FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS

Las autoridades de la Facultad de Ingeniería, por conducto del jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo de 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

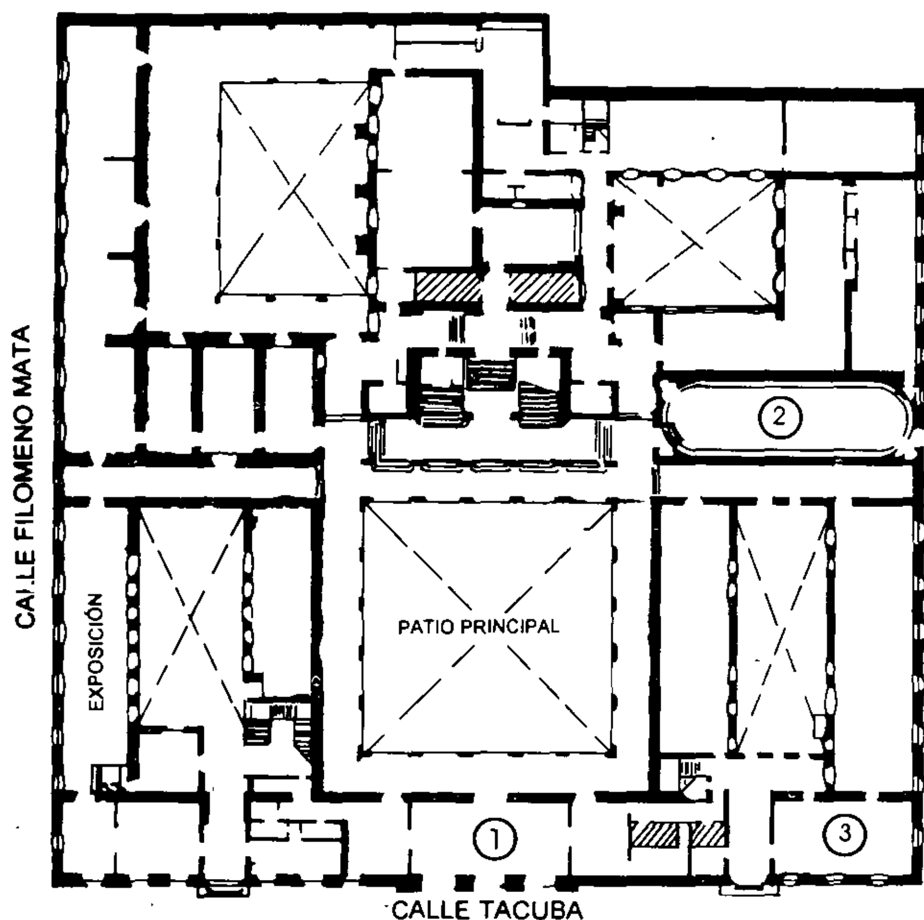
Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

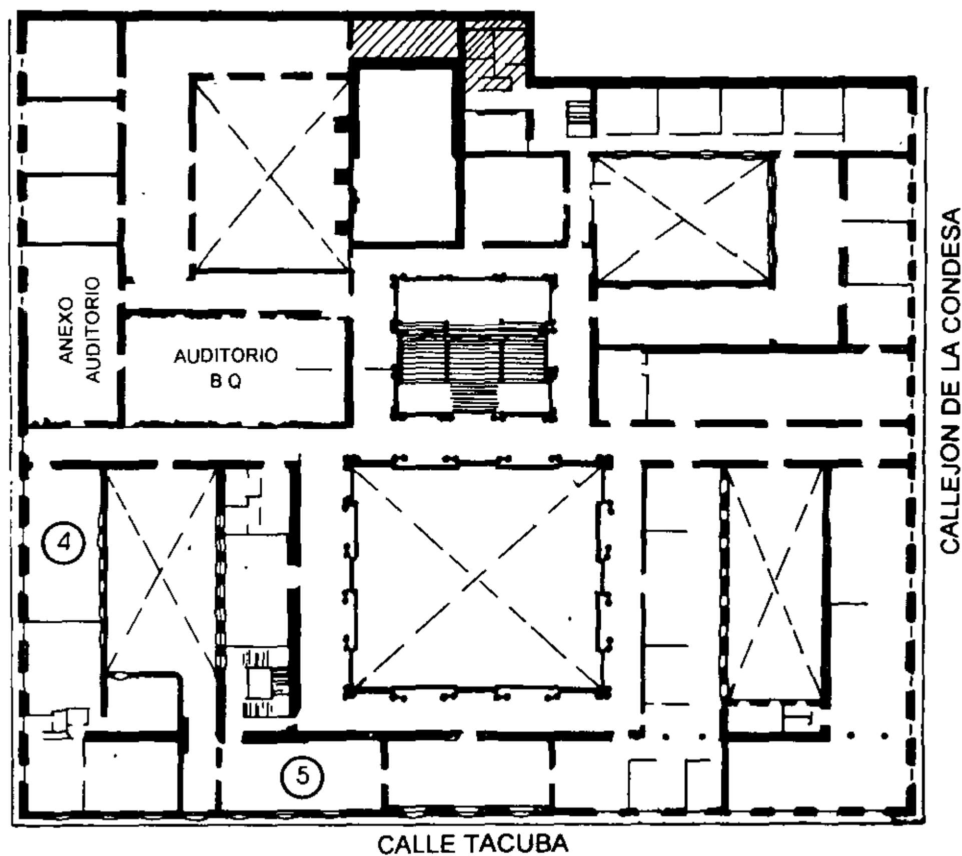
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente
División de Educación Continua.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

PALACIO DE MINERIA

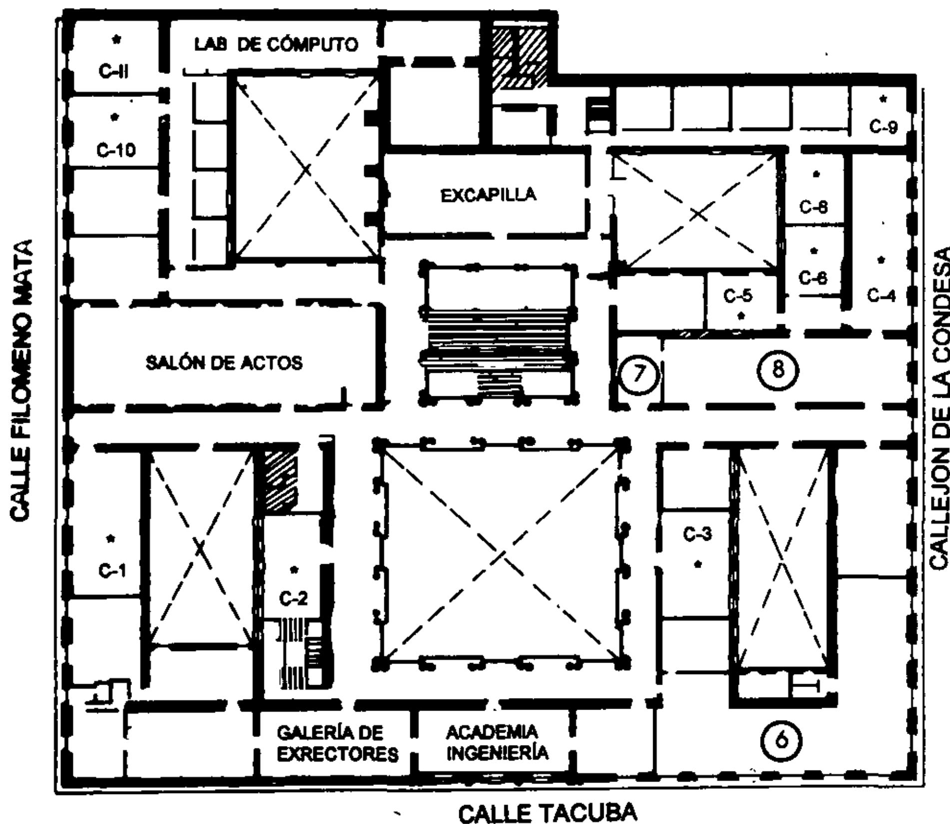




PLANTA BAJA

MEZZANINNE

PALACIO DE MINERIA



Ier. PISO

GUÍA DE LOCALIZACIÓN

1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

SANITARIOS

* AULAS

DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS ABIERTOS

ROBOTS MÓVILES

CHAPTER 1 INTRODUCTION TO EXPERT SYSTEMS

EXPOSITOR : DR. JESÚS SAVAGE CARMONA
1998

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtemoc 06000 Mexico, D.F. APDO Postal M-2285
Telefonos 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

CHAPTER 1 INTRODUCTION TO EXPERT SYSTEMS

1.1 INTRODUCTION

This chapter is a broad introduction to expert systems. The fundamental principles of expert systems are introduced. The advantages and disadvantages of expert systems are discussed and the appropriate areas of application for expert systems are described. The relationship of expert systems to other methods of programming are discussed.

1.2 WHAT IS AN EXPERT SYSTEM

The first step in solving any problem is defining the problem area or **domain** to be solved. This consideration is just as true in artificial intelligence (AI) as in conventional programming. However, because of the mystique formerly associated with AI, there is a lingering tendency to still believe the old adage "It's an AI problem if it hasn't been solved yet". This type of mind-set may have been popular in the 1970's w AI was entirely in a research stage. However, today there are many real-world pı ,ms that are being solved by AI and many commercial applications of AI.

Although general solutions to classic AI problems such as natural language translation, speech understanding, and vision have not been found, restricting the problem domain may still produce a useful solution. For example, it is not difficult to build simple natural language systems if the input is restricted to sentences of the form: noun, verb, and object. Currently, systems of this type work very well in providing a user-friendly interface to many software products such as database systems and spreadsheets. In fact, the parsers associated with popular computer text-adventure games today exhibit an amazing degree of ability in understanding natural language.

As Figure 1-1 shows, AI has many areas of interest. The area of **expert systems** is a very successful approximate solution to the classic AI problem of programming intelligence. Professor Edward Feigenbaum of Stanford University, an early pioneer of expert systems technology, has defined an expert system as "... an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution." (Feigenbaum 82). That is, an expert system is a computer system which **emulates** the decision-making ability of a human expert. The term emulate means that the expert system is intended to act in all respects like a human expert. An emulation is much stronger than a simulation which is only required to act like the real thing in some respects.

lthough a general purpose problem solver still eludes us, expert systems function very well in their restricted domains. As proof of their success, you need only observe the many applications of expert systems today in business, medicine, science, and

engineering as well as all the books, journals, conferences and products devoted to expert systems.

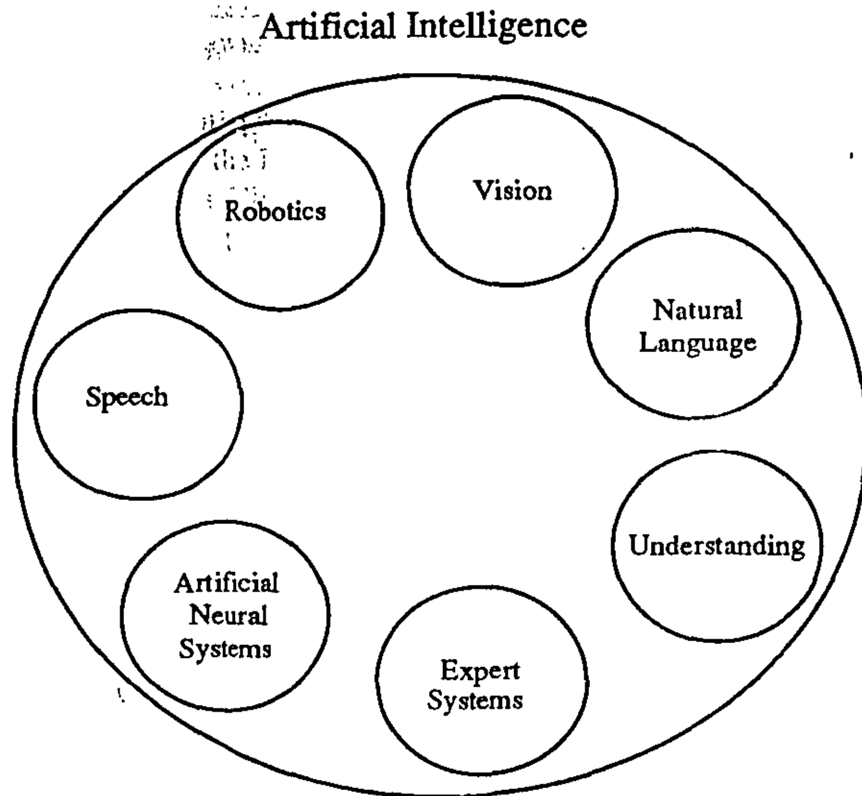


Figure 1-1
Some Areas of Artificial Intelligence

Expert systems is a branch of AI that makes extensive use of specialized knowledge to solve problems at the level of a human **expert**. An expert is a person who has **expertise** in a certain area. That is, the expert has knowledge or special skills that are not known or available to most people. An expert can solve problems that most people cannot solve at all or solve them much more efficiently (but not as cheaply). When expert systems were first developed in the 1970's, they contained expert knowledge exclusively. However, the term expert system is often applied today to any system which uses expert system technology. This expert system technology may include special expert system languages, programs, and hardware designed to aid in the development and execution of expert systems.

The knowledge in expert systems may be either expertise, or knowledge which is generally available from books, magazines, and knowledgeable persons. The terms expert system, **knowledge-based system** or **knowledge-based expert system** are often used synonymously. Most people use expert system simply because it's shorter, even though there may be no expertise in their expert system, only general knowledge.

Figure 1-2 illustrates the basic concept of a knowledge-based expert system. The user supplies facts or other information to the expert system and receives expert advice or **expertise** in response. Internally, the expert system consists of two main components. The knowledge-base contains the knowledge with which the **inference engine** draws conclusions. These conclusions are the expert system's responses to the user's queries for expertise.

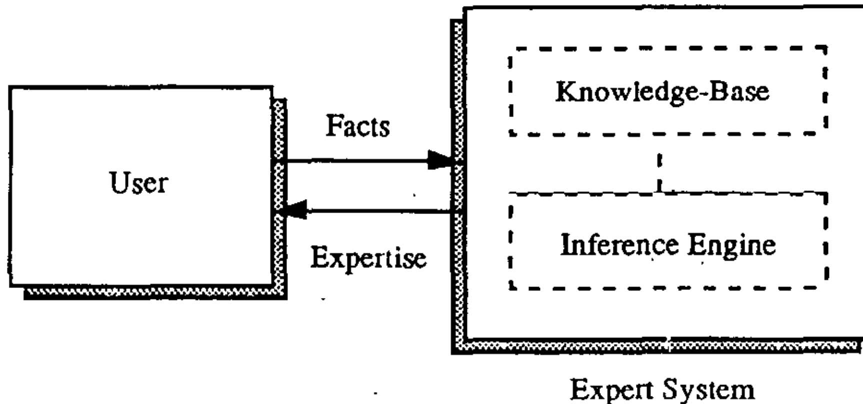


Figure 1-2
Basic Concept of an Expert System Function

Useful knowledge-based systems also have been designed to act as an intelligent assistant to a human expert. These intelligent assistants are designed with expert systems technology because of the development advantages. As more knowledge is added to the intelligent assistant, it acts more like an expert. Developing an intelligent assistant may be a useful milestone in producing a complete expert system. In addition, it may free up more of the expert's time by speeding up the solution of problems.

An expert's knowledge is specific to one **problem domain** as opposed to knowledge about general problem-solving techniques. A problem domain is the special problem area such as medicine, finance, science or engineering and so forth that an expert can solve problems in very well. Expert systems, like human experts, are generally designed to be experts in one problem domain. For example, you would not normally expect a chess expert to have expert knowledge about medicine. Expertise in one problem domain does not automatically carry over to another.

The expert's knowledge about solving specific problems is called the **knowledge domain** of the expert. For example, a medical expert system designed to diagnose infectious diseases will have a great deal of knowledge about certain symptoms caused by infectious diseases. In this case the knowledge domain is medicine and consists of knowledge about diseases, symptoms, and treatments. Figure 1-3 illustrates the relationship between the problem and knowledge domain. Notice that this knowledge domain is entirely included within the problem domain. The portion outside the knowledge domain symbolizes an area in which there is not knowledge about all the problems.

One expert system usually does not have knowledge about other b‑ ‑hes o medicine such as surgery or pediatrics. Although its knowledge of infectiou‑ ‑ase i‑ equivalent to a human expert, the expert system would not know anything about othe knowledge domains unless it was programmed with that domain knowledge.

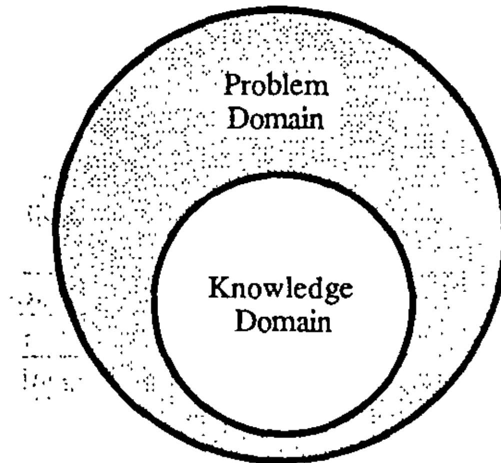


Figure 1-3
A Possible Problem and Knowledge Domain Relationship

In the knowledge domain that it knows about, the expert system reasons ‑akes **inferences** in the same way that a human expert would infer the solution o‑ ‑ ‑roblem. That is, given some facts, a conclusion that follows is inferred. For example, if your spouse hasn't spoken to you in a month, you may infer that he or she had nothing worthwhile to say. However, this is only one of several possible inferences.

As with any new technology, we still have a lot to learn about expert systems. Table 1-1 summarizes the differing views of the participants in a technology. In this table, the technologist may be an engineer or software designer and the technology may be hardware or software. In solving any problem, these are questions that need to be answered or the technology will not be successfully used. Like any other tool, expert systems have appropriate and inappropriate applications. As our experience with expert systems grows, we will discover what these applications are.

Person	*Question*
Manager	What can I use it for?
Technologist	How can I best implement it?
Researcher	How can I extend it?
Consumer	How will it help me? Is it worth the trouble and expense? How reliable is it?

Table 1-1
Differi‑g Views of Technology

1.3 ADVANTAGES OF EXPERT SYSTEMS

Expert systems have a number of attractive features.

- *Increased Availability.* Expertise is available on any suitable computer hardware. In a very real sense, an expert system is the mass production of expertise.
- *Reduced cost.* The cost of providing expertise per user is greatly lowered.
- *Reduced Danger.* Expert systems can be used in environments that might be hazardous for a human.
- *Permanence.* The expertise is permanent. Unlike human experts who may retire, quit or die, the expert system's knowledge will last indefinitely.
- *Multiple expertise.* The knowledge of multiple experts can be made available to work simultaneously and continuously on a problem at any time of day or night. The level of expertise combined from several experts may exceed that of a single human expert (Harmon 85).
- *Increased reliability.* Expert systems increase confidence that the correct decision was made by providing a second opinion to a human expert or break a tie in case of disagreements by multiple human experts. Of course, this method probably won't work if the expert system was programmed by one of the experts. The expert system should always agree with the expert, unless a mistake was made by the expert. However, this may happen if the human expert is tired or under stress.
- *Explanation.* The expert system can explicitly explain in detail the reasoning that led to a conclusion. A human may be too tired, unwilling or unable to do this all the time. This increases the confidence that the correct decision is made.
- *Fast response.* Fast or real-time response may be necessary for some applications. Depending on the software and hardware used, an expert system may respond faster and be more available than a human expert. Some emergency situations may require responses faster than a human and so a real-time expert system is a good choice (Hugh 88) (Ennis 86).
- *Steady, unemotional, and complete response at all times.* This may be very important in real-time and emergency situations when a human expert may not operate at peak efficiency because of stress or fatigue.
- *Intelligent tutor.* The expert system may act as an intelligent tutor by letting the student run sample programs and explaining the system's reasoning.
- *Intelligent Database.* Expert systems can be used to access a database in an intelligent manner (Kerschberg 86) (Schur 88).

The process of developing an expert system has an indirect benefit also since the knowledge of human experts must be put into an explicit form for entering in the computer. Because the knowledge is then explicitly known instead of being implicit in xpert's mind, it can be examined for correctness, consistency and completeness. knowledge may then have to be adjusted or re-examined which improves the quality of the knowledge.

1.4 GENERAL CONCEPTS OF EXPERT SYSTEMS

The knowledge of an expert system may be represented in a number of ways. One common method of representing knowledge is in the form of IF THEN type **rules**, such as

```
IF the light is red THEN stop
```

Although this is a trivial example, many significant expert systems have been built by expressing the knowledge of experts in rules. In fact, the knowledge-based approach to developing expert systems has completely supplanted the early AI approach of the 1950's and 1960's which tried to use sophisticated reasoning techniques with no reliance on knowledge.

Today, a wide range of knowledge-based expert systems have been built. Large systems containing thousands of rules, such as the XCON/R1 system of Digital Equipment Corporation, know much more than any single human expert on how to configure computer systems (McDermott 84). Many small systems for specialized tasks have also been constructed with several hundred rules. These small systems may not operate at the level of an expert but are designed to take advantage of expert systems technology to perform knowledge-intensive tasks. For these small systems, the knowledge may be in books, journals, or other publicly available documentation.

In contrast, a classic expert system embodies unwritten knowledge that n be extracted from an expert by extensive interviews with a **knowledge engineer** over a long period of time. The process of building an expert system is called **knowledge engineering** and is done by a knowledge engineer (Michie 73). Knowledge engineering refers to the acquisition of knowledge from a human expert or other source and its coding in the expert system.

The general stages in the development of an expert system are illustrated in Figure 1-4. The knowledge engineer first establishes a dialog with the human expert in order to elicit the expert's knowledge. This stage is analogous to a system designer in conventional programming discussing the system requirements with a client for whom the program will be constructed. The knowledge engineer then codes the knowledge explicitly in the knowledge-base. The expert then evaluates the expert system and gives a critique to the knowledge engineer. This process iterates until the system's performance is judged to be satisfactory by the expert.

The expression **knowledge-based system** is a better term for the application of knowledge-based technology since it may be used for the creation of either expert systems or knowledge-based systems. However, like the term artificial intelligence, it is common practice today to use the term expert systems when referring to both expert systems and knowledge-based systems, even when the knowledge is not at the level of a human expert.

Expert systems are generally designed very differently from conventional programs because the problems usually have no algorithmic solution and rely on inferer o achieve a reasonable solution. Note that a reasonable solution is about the best we can

expect if no algorithm is available to help us achieve the optimum solution. Since the expert system relies on inference, it must be able to explain its reasoning so that its reasoning can be checked. An **explanation facility** is an integral part of sophisticated expert systems. In fact, elaborate explanation facilities may be designed to allow the user to explore multiple lines of "What if..." type questions, called **hypothetical reasoning** and even translate natural language into rules.

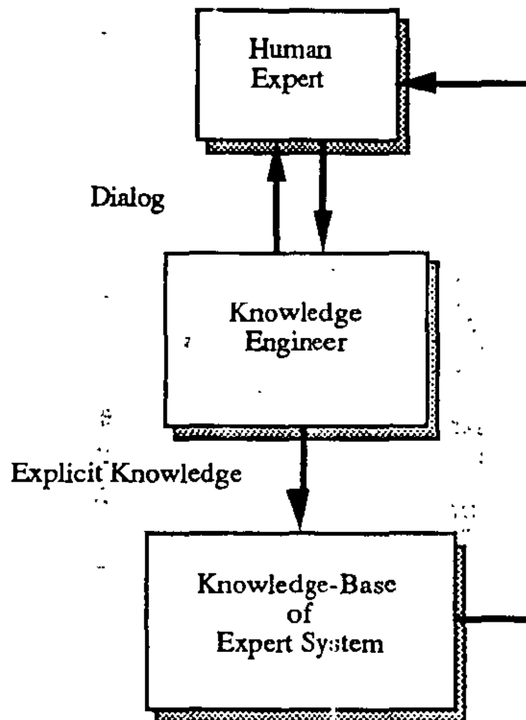


Figure 1-4
Development of an Expert System

Some expert systems even allow the system to learn rules by example, through **rule induction**, in which the system creates rules from tables of data. Formalizing the knowledge of experts into rules is not simple, especially when the expert's knowledge has never been systematically explored. There may be inconsistencies, ambiguities, duplications or other problems with the expert's knowledge that are not apparent until attempts are made to formally represent the knowledge in an expert system.

Human experts also know the extent of their knowledge and qualify their advice as the problem reaches their **limits of ignorance**. A human expert also knows when to "break the rules". Unless expert systems are explicitly designed to deal with uncer- ty, they will make recommendations with the same confidence even if the data they ealing with is inaccurate or incomplete. An expert systems advice, like a human expert's, should degrade gracefully at the boundaries of ignorance instead of abruptly.

A practical limitation of many expert systems today is lack o. **usal knowledge**. That is, the expert systems do not really have an understanding of the underlying causes and effects in a system. It is much easier to program expert systems with **shallow knowledge** based on empirical and heuristic knowledge than **deep knowledge** based on the basic structure, function, and behavior of objects. For example, it is much easier to program an expert system to prescribe an aspirin for a person's headache than program all the underlying biochemical, physiological, anatomical and neurological knowledge about the human body. The programming of a causal model of the human body would be an enormous task and even if successful, the response time of the system would probably be extremely slow because of all the information that the system would have to process.

One type of shallow knowledge is **heuristic knowledge**, where the term heuristic comes from the Greek and means to discover. Heuristics are not guaranteed to succeed in the same way that an algorithm is a guaranteed solution to a problem. Instead, heuristics are rules of thumb or empirical knowledge gained from experience which may aid in the solution but are not guaranteed to work. However, in many fields such as medicine and engineering, heuristics play an essential role in some types of problem solving. Even if an exact solution is known, it may be impractical to use because of cost or time constraints. Heuristics can provide valuable shortcuts that can reduce time and cost.

Another problem with expert systems today is that their expertise is limited to the knowledge domain that the systems know about. Typical expert systems can ener-alize their knowledge by using **analogy** to reason about new situations that the way people can. Although rule induction helps, only limited types of knowledge can be put into an expert system this way. The customary way of building an expert system by having the knowledge engineer repeat the cycle of interviewing the expert, constructing a prototype, testing, interviewing, and so on is a very time consuming and labor intensive task. In fact, this problem of transferring human knowledge into an expert system is so major that it is called the **knowledge acquisition bottleneck**. This is a descriptive term because the knowledge acquisition bottleneck constricts the building of an expert system like an ordinary bottleneck constricts fluid flow into a bottle.

In spite of their present limitations, expert systems have been very successful in dealing with real-world problems that conventional programming methodologies have been unable to solve, especially those dealing with uncertain or incomplete information. The important point is to be aware of the advantages and limitations of this new technology so that it can be appropriately utilized.

1.5 CHARACTERISTICS OF AN EXPERT SYSTEM

An expert system is usually designed to have the following general characteris-

- *High performance.* The system must be capable of responding at a level of competency equal to or better than an expert in the field. That is, the quality of the advice given by the system must be very high.
- *Adequate response time.* The system must also perform in a reasonable time, comparable to or better than the time required by an expert to reach a decision. An expert system that takes a year to reach a decision compared to an expert's time of one hour would not be too useful. The **time constraints** placed on the performance of an expert system may be especially severe in the case of real-time systems, when a response must be made within a certain time interval.
- *Good reliability.* The expert system must be reliable and not prone to crashes or else it will not be used.
- *Understandable.* The system should be able to explain the steps of its reasoning while executing so that it is understandable. Rather than being just a "black box" that produces a miraculous answer, the system should have an explanation capability in the same way that human experts can explain their reasoning. This feature is very important for several reasons.

One reason is that human life and property may depend on the answers of the expert system. Because of the great potential for harm, an expert system must be able to justify its conclusions in the same way a human expert can explain why a certain conclusion was reached. Thus, an explanation facility provides an understandable check of the reasoning for humans.

A second reason for having an explanation facility occurs in the development phase of an expert system to confirm that the knowledge has been correctly acquired and is being correctly used by the system. This is very important in debugging since the knowledge may be incorrectly entered due to typos or be incorrect due to misunderstandings between the knowledge engineer and the expert. A good explanation facility allows the expert and knowledge engineer to verify the correctness of the knowledge. Also, because of the way that typical expert systems are constructed, it is very difficult to read a significant program listing and understand its operation.

An additional source of error may be unforeseen interactions in the expert system, which may be detected by running test cases with known reasoning that the system should follow. As we will discuss in more detail later, multiple rules may apply to a given situation about which the system is reasoning. The flow of execution is not sequential in an expert system so that you cannot just read its code line by line and understand how the system operates. That is, the order in which rules have been entered in the system is not necessarily the order in which they will be executed. The expert system acts much like a parallel program in which the rules are independent knowledge processors.

- *Flexibility.* Because of the large amount of knowledge that an expert system may have, it is important to have an efficient mechanism for adding, changing, and deleting knowledge. One reason for the popularity of rule-based systems is the efficient and modular storage capability of rules.

Depending on the system, an explanation facility may be simple or elab . A simple explanation facility in a rule-based system may simply list all the t. hat made the latest rule execute. More elaborate systems may do the following.

- *List all the reasons for and against a particular hypothesis.* A hypothesis is a goal which is to be proved such as "The patient has a tetanus infection" in a medical diagnostic expert system. In a real problem, there may be multiple hypotheses, just as a patient may have several diseases at once. A hypothesis can also be viewed as a fact whose truth is in doubt and must be proved.
- *List all the hypotheses that may explain the observed evidence.*
- *Explain all the consequences of a hypothesis.* For example, assuming that the patient does have tetanus, there should also be evidence of fever as the infection runs its course. If this symptom is then observed, it adds credibility that the hypothesis is true. If the symptom is not observed, it reduces the credibility of the hypothesis.
- *Give a* ***prognosis*** *or prediction of what will occur if the hypothesis is true.*
- *Justify the questions that the program asks of the user for further information.* These questions may be used to direct the line of reasoning to likely diagnostic paths. In most real problems, it is too expensive or take too long to explore all possibilities, and some way must be provided to guide the search for the correct solution. For example, consider the cost, time and effect of administering all possible medical tests to a patient complaining of a sore throat.
- *Justify the knowledge of the program.* For example, if the program claims that the hypothesis "The patient has a tetanus infection" is true, the user could ask for an explanation. The program might justify this conclusion on the basis of a rule which says that if the patient has a positive blood test for tetanus, the patient has tetanus. Now the user could ask the program to justify this rule. The program could respond by stating that a positive blood test for a disease is proof of the disease.

In this case, the program is actually quoting a **metarule**, which is knowledge about rules. The prefix meta means above or beyond. Some programs such as Meta-DENDRAL have been explicitly created to infer new rules (Buchanan 78). A hypothesis is justified by knowledge and the knowledge is justified by a **warrant** that it is correct. A warrant is essentially a meta-explanation that explains the expert systems explanation of its reasoning.

Knowledge can easily grow **incrementally** in a rule-based system. That is, the knowledge base can grow little by little as rules are added so that the performance and correctness of the system can be continually checked. If the rules are properly designed, the interactions between rules will be minimized or eliminated to protect against unforeseen effects. The incremental growth of knowledge facilitates **rapid prototyping** so that the knowledge engineer can quickly show the expert a working prototype of the expert system. This is an important feature because it main the expert's and management's interest in the project. Rapid prototyping also quickly

exposes any gaps, inconsistencies or errors in the expert's knowledge or the system so that corrections can immediately be made.

1.6 THE DEVELOPMENT OF EXPERT SYSTEMS TECHNOLOGY

AI has many branches concerned with speech, vision, robotics, natural language understanding, and learning and expert systems. The roots of expert systems lie in many disciplines. In particular, one of the major roots of expert systems is the area of human information processing, called **cognitive science**. Cognition is the study of how humans process information. In other words, cognition is the study of how people think, especially when solving problems.

The study of cognition is very important if we want to make computers emulate human experts. Often, experts can't explain how they solve problems—the solution just comes to them. If the expert can't explain how a problem is solved, it's not possible to encode the knowledge in an expert system based on explicit knowledge. In this case, the only possibility is programs that learn by themselves to emulate the expert. These are programs based on induction and artificial neural systems, to be discussed later.

Human Problem-Solving and Productions

The development of expert systems technology draws on a wide background. Table 1-2 is a brief summary of some important developments that have converged in modern expert systems. Whenever possible, the starting dates of projects are used. Many projects extend over many years. These developments are covered in more detail in this chapter and others. The best single reference for all the early systems is the three-volume *Handbook of Artificial Intelligence* (Feigenbaum 81).

In the late 1950's and 1960's, a number of programs were written with the goal of general problem solving. The most famous of these was the **General Problem** Solver created by Newell and Simon and described in a series of papers culminating in their monumental 920-page work on cognition, *Human Problem Solving* (Newell 72).

One of the most significant results demonstrated by Newell and Simon was that much of human problem solving or **cognition** could be expressed by IF THEN type **production rules**. For example, IF it looks like it's going to rain THEN carry an umbrella, or IF your spouse is in a bad mood THEN don't appear happy. A rule corresponds to a small, modular collection of knowledge called a **chunk**. The chunks are organized in a loose arrangement with links to related chunks of knowledge. One theory is that all human memory is organized in chunks. An example of a rule representing a chunk of knowledge is

Year	*Events*
1943	Post production rules; McCulloch and Pitts Neuron Model
1954	Markov Algorithm for controlling rule execution
1956	Dartmouth Conference; Logic Theorist; Heuristic Search; "AI" term coined
1957	Perceptron invented by Rosenblatt; GPS (General Problem Solver) started (Newell, Shaw and Simon)
1958	LISP AI language (McCarthy)
1962	Rosenblatt's *Principles of Neurodynamics* on Perceptions
1965	Resolution Method of automatic theorem proving (Robinson) Fuzzy Logic for reasoning about fuzzy objects (Zadeh) Work begun on DENDRAL, the first expert system (Feigenbaum, Buchanan, et.al.)
1968	Semantic nets, associative memory model (Quillian)
1969	MACSYMA math expert system (Martin and Moses)
1970	Work begins on PROLOG (Colmerauer, Roussell, *et. al.*)
1971	HEARSAY I for speech recognition *Human Problem Solving* popularizes rules (Newell and Simon)
1973	MYCIN expert system for medical diagnosis(Shortliffe, *et. al.*) leading to GUIDON, intelligent tutoring (Clancey) and TEIRESIAS, explanation facility concept (Davis) and EMYCIN, first shell (Van Melle, Shortliffe and Buchanan) HEARSAY II, blackboard model of multiple cooperating experts
1975	Frames, knowledge representation (Minsky)
1976	AM (Artificial Mathematician) creative discovery of math concepts (Lenat) Dempster-Shafer Theory of Evidence for reasoning under uncertainty Work begun on PROSPECTOR expert system for mineral exploration (Duda, Hart, *et. al.*)
1977	OPS expert system shell (Forgy), used in XCON/R1
1978	Work started on XCON/R1 (McDermott, DEC) to configure DEC computer systems Meta-DENDRAL, metarules and rule induction (Buchanan)
1979	Rete Algorithm for fast pattern matching (Forgy) Commercialization of AI begins Inference Corp. fomed (releases ART expert system tool in 1985)
1980	Symbolics, LMI founded to manufacture LISP machines
1982	SMP math expert system; Hopfield Neural Net; Japanese Fifth Generation Project to develop intelligent computers
1983	KEE expert system tool (IntelliCorp)
1985	CLIPS expert system tool (NASA)

Table 1-2
Some Important Events in the History of Expert Systems

```
IF the car doesn't run and
   the fuel gauge reads empty
THEN fill the gas tank
```

Newell and Simon popularized the use of rules to represent human knowledge and showed how reasoning could be done with rules. Cognitive psychologists have used rules as a model to explain human information processing. The basic idea is that sensory input provides stimuli to the brain. The stimuli trigger the appropriate rules of **long-term memory** which produce the appropriate response. Long-term memory is where our knowledge is stored. For example, we all have rules such as

```
IF there is flame THEN there is a fire
IF there is smoke THEN there may be a fire
IF there is a siren THEN there may be a fire
```

Notice that the last two rules are not expressed with complete certainty. The fire may be out, but there may still be smoke in the air. Likewise, a siren does not prove that there is a fire since it may be a false alarm. The stimuli of seeing flames, smelling smoke, and hearing a siren will trigger these and similar types of rules.

Long-term memory consists of many rules having the simple IF THEN structure. In fact, a Grand Master chess expert may know 50,000 or more chunks of knowledge about chess patterns. In contrast to the long-term memory, the **short-term memory** is used for the temporary storage of knowledge during problem solving. Although long-term memory can hold hundreds of thousands or more chunks, the capacity of working memory is surprisingly small—4 to 7 chunks. As a simple example of this, try visualizing some numbers in your mind. Most people can only see 4 to 7 numbers at once. Of course we can memorize many more than 4 to 7 numbers. However, those numbers are kept in long-term memory.

One theory proposes that short-term memory represents the number of chunks that simultaneously can be active and considers human problem-solving as a spreading of these activated chunks in the mind. Eventually, a chunk may be activated with such intensity that a conscious thought is generated and you say to yourself, "Hmm.. something's burning."

The other element necessary for human-problem solving is a **cognitive processor**. The cognitive processor tries to find the rules that will be **activated** by the appropriate stimuli. Not just any rule will do. For example, you wouldn't want to fill your gas tank every time you heard a siren. Only a rule which matched the stimuli would be activated. If there are multiple rules that are activated at once, the cognitive processor must perform a conflict resolution to decide which rule has highest priority. The rule with highest priority will be executed. for example, if both of the following es are activated

```
IF there is a fire THEN leave
IF my clothes are burning THEN put out the fire
```

then the actions of one rule will be executed before the other. The **inference engine** of modern expert systems corresponds to the cognitive processor.

The Newell and Simon model of human problem-solving in terms of long-term memory (rules), short-term memory (working memory), and a cognitive processor (inference engine) is the basis of modern rule-based expert systems.

Rules like these are a type of **production system**. Rule-based production systems are a popular method of implementing expert systems today. The individual rules that comprise a production system are the **production rules**. In designing an expert system, an important factor is the amount of knowledge or **granularity** of the rules. Too little granularity makes it difficult to understand a rule without reference to other rules. Too much granularity makes the expert system difficult to modify since several chunks of knowledge are intermingled in one rule.

Until the mid-sixties, a major quest of AI was to produce intelligent systems that relied little on domain knowledge and greatly on powerful methods of reasoning. Even the name General Problem Solver illustrates the concentration on machines that were not designed for one specific domain but were intended to solve many types of problems. Although the methods of reasoning used by general problem solvers were very powerful, the machines were eternal beginners. When presented with a new domain, they had to discover everything from first principles and were not as good as human experts who relied on domain knowledge for high performance.

An example of the power of knowledge is playing chess. The best chess players are humans despite the fact that computers are much faster than people in doing calculations. Studies have shown (Chase 73) that human expert chess players do not have super powers of reasoning but instead rely on knowledge of chesspiece patterns built up over years of play. As mentioned previously, one estimate places an expert chess player's knowledge at about 50,000 patterns. Humans are very good at recognizing patterns such as pieces on a chessboard. Instead of trying to reason ahead ten or twenty possible moves for every piece as a computer might, the human analyzes the game in terms of patterns that reveal long-term threats while remaining alert for short-term surprise moves.

While domain knowledge is very powerful, it is generally limited to the domain. For example, a person who becomes an expert chess player does not automatically become an expert at solving math problems or even a checkers expert. While some knowledge may carry over to another domain, such as careful planning of moves, this is a skill rather than genuine expertise.

By the early seventies, it became apparent that domain knowledge was the key to building machine problem-solvers that could function at the level of human experts. Although methods of reasoning are important, studies have shown that experts do not primarily rely on reasoning for problem-solving. In fact, reasoning may play a minor role in an expert's problem solving. Instead, experts rely on a vast knowledge of heuristics and experience that they have built up over the years. If an expert cannot solve a problem based on expertise, then it is necessary for the expert to reason from first principles and theory (or more likely ask another expert). The reasoning ability of an expert is generally no better than that of an average person in dealing with a totally

unfamiliar situation. The early attempts at building powerful problem-solvers based only on reasoning have shown that a problem-solver is crippled if it must rely solely on reasoning.

The insight that domain knowledge was the key to building real-world problem solvers led to the success of expert systems. The successful expert systems today are thus knowledge-based expert systems rather than general problem solvers. In addition, the same technology that led to the development of expert systems has also led to the development of knowledge-based systems that do not necessarily contain human expertise.

While expertise is considered knowledge that is specialized and known only to a few, knowledge is generally found in books, periodicals and other widely available resources. For example, the knowledge of how to solve a quadratic equation or perform integration and differentiation is widely available. Knowledge-based computer programs such as MACSYMA and SMP are available to perform automatically these and many other mathematical operations on either numeric or symbolic operands. Other knowledge-based programs may perform process control of manufacturing plants. Today the terms **knowledge-based programming** and expert systems are often used synonymously. In fact, expert systems is now considered an alternative programming model or **paradigm** to conventional algorithmic programming.

he Rise of Knowledge-based Systems

With the acceptance of the knowledge-based paradigm in the 1970's, a number of successful prototype expert systems were created. These systems could interpret mass spectrograms to identify chemical constituents (DENDRAL), diagnose illness (MYCIN), analyze geologic data for oil (DIPMETER) and minerals (PROSPECTOR), and configure computer systems (XCON/R1). The news that PROSPECTOR had discovered a mineral deposit worth $100 million dollars and that XCON/R1 was saving Digital Equipment Corporation (DEC) millions of dollars a year triggered a sensational interest in the new expert systems technology by 1980. The branch of AI that had started off in the 1950's as a study of human information processing had now grown to achieve commercial success by the development of practical programs for real-world use.

The MYCIN expert system was very important for several reasons. First, it demonstrated that AI could be used for practical real-world problems. Second, MYCIN was the testbed of new concepts such as the explanation facility, automatic acquisition of knowledge, and intelligent tutoring that are found in a number of expert systems today. The third reason that MYCIN was important is that it demonstrated the feasibility of the expert system **shell**.

Previous expert systems such as DENDRAL were one–of–a–kind systems in which the knowledge–base was intermingled with the software that applied the knowledge, the inference engine. MYCIN explicitly separated the knowledge–base from the inference engine. This was extremely important to the development of expert system technology since it meant that the essential core of the expert system could be reused.

That is, a new expert system could be built much more rapidly than a DENDRAI system by emptying out the old knowledge and putting in knowledge about the new domain. The part of MYCIN that dealt with inference and explanation, the shell, could then be refilled with knowledge about the new system. The shell produced by removing the medical knowledge of MYCIN was called EMYCIN (Essential or Empty MYCIN).

By the late seventies, the three concepts that are basic to most expert systems today had converged, as shown in Figure 1-5. These concepts are rules, the shell, and knowledge.

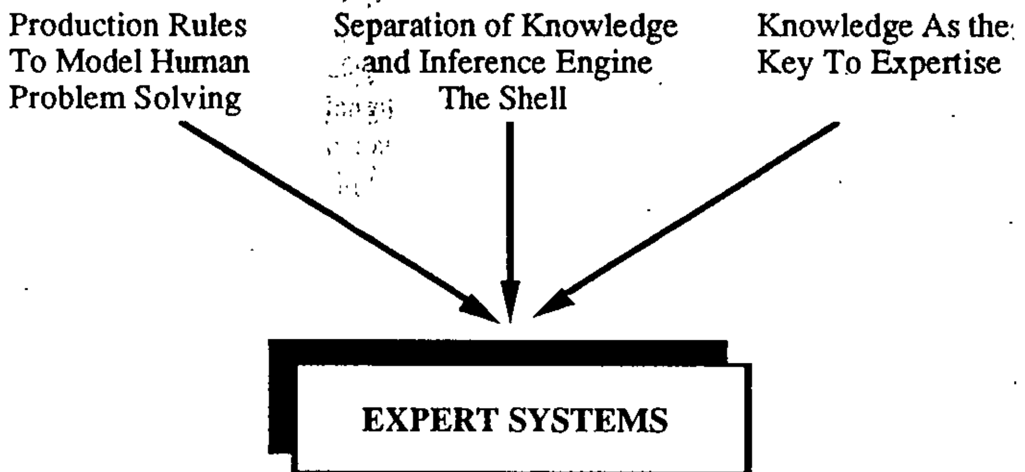


Figure 1-5
Convergence Of Three Important Factors To Create
The Modem Rule-based Expert System

By 1980 new companies started to bring expert systems out of the university laboratory and produce commercial products. Powerful new software for expert system development was introduced such as the Automated Reasoning Tool (ART) by the Inference Corp., the Knowledge Engineering Tool (KEE) by IntelliCorp, and Rulemaster by Radian Corp. In addition, specialized new hardware was developed to run the software with greater speed than ever before. Companies such as Symbolics and LMI introduced computers referred to as LISP machines because they were designed for LISP, the underlying base language of the expert systems development software. In LISP machines, the native assembly language, operating system, and all other fundamental code were done in LISP.

CLIPS

Unfortunately, this high technology also commanded a high price. While a single-user LISP machine was much more powerful and productive than a general purpose machine running LISP, the single-user machine and a single-user software license cost a tot f $100,000. Establishing an AI lab with a half-dozen programmers could easily c half-million dollars.

These objections were overcome in the mid-80's by the introduction of powerful development software such as CLIPS by NASA. CLIPS is written in C for speed and portability, and also uses a powerful pattern matching called the Rete Algorithm. CLIPS is free to Government users and Government contractors. The cost to other users is $312 (as of 1988) which includes the complete 25,000 lines of CLIPS source code.

Any C compiler that supports the standard Kernigan and Richie C language can be used to install CLIPS. CLIPS has been installed on the IBM PC and compatibles, VAX, Hewlett-Packard, Sun, Cray and many other makes of computers. A version for the Macintosh is also available that supports the full Macintosh interface with pull-down windows and mouse support. The speed of CLIPS on a 80386 or 68020 microcomputer is compatible with the state-of-the-art LISP-based development software and LISP machines.

1.7 EXPERT SYSTEMS APPLICATIONS AND DOMAINS

Conventional computer programs are used to solve many types of problems. Generally, these problems have algorithmic solutions that lend themselves well to conventional ograms and programming languages such as FORTRAN, Pascal, Ada, and so on. In any application areas such as business and engineering, numeric calculations are of primary importance. By contrast, expert systems are primarily designed for symbolic reasoning.

While languages such as LISP and PROLOG are also used for symbolic manipulation, they are more general purpose than expert system shells. This does not mean that it is not possible to build expert systems in LISP and PROLOG. In fact, many expert systems have been built with PROLOG and LISP. PROLOG especially has a number of advantages for diagnostic systems because of its built-in backward chaining. Rather, it is more convenient and efficient to build large expert systems with shells and utility programs specifically designed for expert system building. Instead of "re-inventing the wheel" every time a new expert system is to be built, it is more efficient to use specialized tools designed for expert system building than general purpose tools.

Applications of Expert Systems

Expert systems have been applied to virtually every field of knowledge. Some have been designed as research tools while others fulfill important business and industrial functions. One example of an expert system in routine business use is the XCON system of Digital Equipment Corp. (DEC). The XCON system (originally called R1) was veloped in conjunction with John McDermott of Carnegie-Mellon University. XCON is an expert configuring system for DEC computer systems (McDermott 84).

The **configuration** of a computer system means that when a customer places an order, all the right parts—software, hardware, and documentation—are supplied. For

large systems, the customer's system is set up or configured at the factory and t[illegible]d to insure that it meets the customer's requirements. Unlike the purchase of a TV [illegible] or a home computer, there are many options and interconnections possible with a large computer system. In putting together a large system, it's not enough to just ship the requested number of CPU's, disk drives, terminals and so forth. The proper interconnections and cabling must also be supplied and the system checked to verify that it is working correctly.

The XCON system is probably one of the most successful expert systems in routine use and saves DEC millions of dollars a year, reduces time to configure an order, and improves the accuracy of an order. XCON can configure an average order in about two minutes, which is fifteen times faster than a human. Also, while the humans configured orders correctly 70% of the time, XCON has an accuracy of 98%. These are important concerns since configuring a complex computer system at the factory involves considerable effort. It is very expensive to configure a system partially and then find out the system cannot meet the customer's requirements or that other components are needed and shipment will be delayed until the parts arrive.

Hundreds of expert systems have been built and reported in computer journals, books, and conferences. This probably represents only the tip of the iceberg since many companies and military organizations will not report their systems because of proprietary or secret knowledge contained in the systems. Based on the systems described in the open literature, certain broad classes of expert systems applications can be discerned, as shown in Table 1-3.

Class	*General Area*
Configuration	Assemble proper components of a system in the proper way.
Diagnosis	Infer underlying problems based on observed evidence.
Instruction	Intelligent teaching so that a student can ask *Why, How* and *What If* type questions just as if a human was teaching.
Interpretation	Explain observed data.
Monitoring	Compares observed data to expected data to judge performance.
Planning	Devise actions to yield a desired outcome.
Prognosis	Predict the outcome of a given situation.
Remedy	Prescribe treatment for a problem.
Control	Regulate a process. May require interpretation, diagnosis, monitoring, planning, prognosis, and remedies.

Table 1-3
Broad Classes of Expert Systems

Examples of some expert systems are shown in Tables 1-4 through 1-9 (Waterman 86), and in the Historical Bibliography at the end of the chapter.

Name	*Chemistry*
CRYSALIS	Interpret a protein's 3-D structure
DENDRAL	Interpret molecular structure
TQMSTUNE	Remedy Triple Quadruple Mass Spectrometer (keep it tuned)
CLONER	Design new biological molecules
MOLGEN	Design gene-cloning experiments
SECS	Design complex organic molecules
SPEX	Plan molecular biology experiments

Table 1-4
Chemistry Expert Systems

Name	*Electronics*
ACE	Diagnosis telephone network faults
IN-ATE	Diagnosis oscilloscope faults
NDS	Diagnose national communication net
EURISKO	Design 3-D microelectronics
PALLADIO	Design and test new VLSI circuits
REDESIGN	Redesign digital circuits to new
CADHELP	Instruct for computer aided design
SOPHIE	Instruct circuit fault diagnosis

Table 1-5
Electronics Expert Systems

Name	*Medicine*
PUFF	Diagnosis lung disease
VM	Monitors intensive-care patients
ABEL	Diagnosis acid-base/electrolytes
AI/COAG	Diagnosis blood disease
AI/RHEUM	Diagnosis rheumatoid disease
CADUCEUS	Diagnosis internal medicine disease
ANNA	Monitor digitalis therapy
BLUE BOX	Diagnosis/remedy depression
MYCIN	Diagnosis/remedy bacterial infections
ONCOCIN	Remedy/manage chemotherapy patients
ATTENDING	Instruct in anesthetic management
GUIDON	Instruct in bacterial infections

Table 1-6
Medical Expert Systems

Name	*Engineering*
REACTOR	Diagnosis/remedy reactor accidents
DELTA	Diagnosis/remedy GE locomotives
STEAMER	Instruct operation - steam powerplant

Table 1-7
Engineering Expert Systems

Name	*Geology*
DIPMETER	Interpret dipmeter logs
LITHO	Interpret oil well log data
MUD	Diagnosis/remedy drilling problems
PROSPECTOR	Interpret geologic data for minerals

Table 1-8
Geology Expert Systems

Name	*Computer Systems*
PTRANS	Prognosis for managing DEC computers
BDS	Diagnosis bad parts in switching net
XCON	Configure DEC computer systems
XSEL	Configure DEC computer sales order
XSITE	Configure customer site for DEC computers
YES/MVS	Monitor/control IBM MVS operating system
TIMM	Diagnosis DEC computers

Table 1-9
Computer Expert Systems

Appropriate Domains for Expert Systems

Before starting to build an expert system, it is essential to decide if an expert system is the appropriate paradigm. For example, one concern is whether an expert system should be used instead of an alternative paradigm such as conventional programming. The appropriate domain for an expert system depends on a number of factors.

- *Can the problem be solved effectively by conventional programming?* If the answer is yes, then an expert system is not the best choice. For example, consider the problem of diagnosing some equipment. If all the symptoms for all malfunctions are known in advance, then a simple table lookup or decision tree of the fault is adequate. Expert systems are best suited for situations in which there is no efficient algorithmic solution. Such cases are called **ill-structured problen** [illegible]d reasoning may offer the only hope of a good solution.

As an example of an ill-structured problem, consider the case of a person who is thinking about travel and visits a travel agent. While most people have a destination and plans, there are exceptions. Table 1-10 lists some characteristics of an ill-structured problem as indicated by the person's responses to the travel agent's questions.

Travel Agent's Questions	*Responses*
Can I help you?	I'm thinking about going somewhere
Where do you want to go?	I'm not sure where to go
Any particular destination?	I just like to travel; destination's not important
How much can you afford?	I don't have enough money to go
Can you get some money?	I don't know how to get the money
When do you want to go?	I must go soon

Table 1-10
An Example of an Ill-structured Problem

Although this is probably an extreme case, it does illustrate the basic concept of an ill-structured problem. As you can see, an ill-structured problem would not lend itself well to an algorithmic solution because there are so many possibilities.

In dealing with ill-structured problems, there is a danger in that the expert system design may accidentally mirror an algorithmic solution. That is, the development of the expert system may unknowingly discover an algorithmic solution. A clue that this has happened occurs if a solution is found that requires a rigid **control structure**. That is, the rules are forced to execute in a certain sequence by the knowledge engineer explicitly setting the priorities of many rules. Forcing a rigid control structure on the expert system cancels a major advantage of expert system technology, which is dealing with unexpected input that does not follow a predetermined pattern (Parrello 88). That is, expert systems react opportunistically to their input, whatever it is. Conventional programs generally expect input to follow a certain sequence. An expert system with a lot of control often indicates a disguised algorithm and may be a good candidate for recoding as a conventional program.

- *Is the domain well-bounded?* It is very important to have well-defined limits on what the expert system is expected to know and what its capabilities should be. For example, suppose you wanted to create an expert system to diagnose headaches. Certainly medical knowledge of a physician would be put in the knowledge-base. However, for a deep understanding of headaches, you might also put in knowledge about neurochemistry, then its parent area of biochemistry, then chemistry, molecular biophysics, and so forth perhaps down to subnuclear physics. Other domains such as biofeedback, psychology, psychiatry, physiology, exercise, yoga, and stress management may also have pertinent knowledge about headaches. The point of all this is—when do you stop adding domains? The more domains, the more complex the expert system becomes.

In particular, the task of coordinating all the expertise becomes a major task. In the real world, we know from experience how difficult it is to have coordinated teams of experts working on problems, especially when they come up with conflicting recommendations. If we knew how to program well the coordination of expertise, then we could try programming an expert system to have the knowledge of multiple experts. Attempts have been made to coordinate multiple expert systems in the HEARSAY II and HEARSAY III systems. However, it is a complex task that should be viewed more as research rather than producing a deliverable expert system product.

- *Is there a need and a desire for an expert system?* Although it's great experience to build an expert system, it's rather pointless if no one is willing to use it. If there already are many human experts, it's difficult to justify an expert system based on the reason of scarce human expertise. Also, if the experts or users don't want the system, it will not be accepted even if there is a need for it.

Management especially must be willing to support the system. This is even more critical for expert systems than conventional programs because expert systems is a new technology. This means that there are few experienced people and more uncertainty about what can be accomplished. However, the area of expert systems deserves more support because it attempts to solve the problems that cannot be done by conventional programming. The risks are greater but so are the rewards.

- *Is there at least one human expert who is willing to cooperate?* There must be an expert who is willing, and preferably enthusiastic, about the project. Not all experts are willing to have their knowledge examined for faults and then put into a computer. Even if there are multiple experts willing to cooperate in the development, it might be wise to limit the number of experts involved in development. Different experts may have different ways of solving a problem, such as requesting different diagnostic tests. Sometimes, they may even reach different conclusions. Trying to code multiple methods of problem-solving in one knowledge-base may create internal conflicts and incompatibilities.

- *Can the expert explain the knowledge so that it is understandable by the knowledge engineer?* Even if the expert is willing to cooperate, there may be difficulty in expressing the knowledge in explicit terms. As a simple example of this difficulty, can you explain in words how you move a finger? Although you could say it's done by contracting a muscle in the finger, the next question is—how do you contract a finger muscle? The other difficulty in communication between expert and knowledge engineer is that the knowledge engineer doesn't know the technical terms of the expert. This problem is particularly acute with medical terminology. It may take a year or longer for the knowledge engineer to even understand what the expert is talking about, let alone translate that knowledge into explicit computer code.

- *Is the problem-solving knowledge mainly heuristic and uncertain?* Expert systems *are* appropriate when the expert's knowledge is largely heuristic and uncertain. That is, the knowledge may be based on experience, called **experiential knowledge**, and the expert may have to try various approaches in case one doesn't work. In other words, the expert's knowledge may be a trial-and-error approach, rather than one based on logic and algorithms. However, the expert can still solve the problem faster than someone else who is not an expert. This is a good application for expert systems. If the problem can be solved simply by logic and algorithms, it is best solved by a conventional program.

1.8 LANGUAGES, SHELLS AND TOOLS

A fundamental decision in defining a problem is deciding how best to model it. Sometimes experience is available to aid in choosing the best paradigm. For example, experience suggests that a payroll is best done using conventional procedural programming. Experience also suggests that it is preferable to use a commercial package, if available, rather than writing one from scratch. A general guide to selecting a paradigm is to consider the most traditional one first—conventional programming. reason for doing this is because of the vast amount of experience we have with entional programming and the wide variety of commercial packages available. If a problem cannot be effectively done by conventional programming, then turn to non-conventional paradigms such as AI.

Although expert systems is a branch of AI, there are specialized languages for expert systems that are quite different from the commonly used AI languages such as LISP and PROLOG. While many others have been developed, such as IPL-II, SAIL, CONNIVER, KRL and Smalltalk, few are widely used except for research (Scown 85).

An expert system language is a higher order language than languages like LISP or C because it is easier to do certain things, but there is also a smaller range of problems that can be addressed. That is, the specialized nature of expert system languages makes them very suitable for writing expert systems but not for general purpose programming. In many situations, it is even necessary to exit temporarily from an expert system language to perform a function in a procedural language.

The primary functional difference between expert system languages and procedural languages is the focus of representation. Procedural languages focus on providing flexible and robust techniques to represent data. For example, data structures such as arrays, records, linked lists, stacks, queues, and trees are easily created and manipulated. Modern languages such as Modula-2 and Ada are designed to aid in **data abstraction** by providing structures for encapsulation such as modules and packages. This provides a level of abstraction which is then implemented by methods such as operators and ol statements to yield a program. The data and methods to manipulate the data are dy interwoven. In contrast, expert system languages focus on providing flexible and robust ways to represent knowledge. The expert system paradigm allows two levels of abstraction: data abstraction and **knowledge abstraction**. Expert system

languages specifically separate the data from the methods of manipulating the dat n example of this separation is that of facts (data abstraction) and rules (knowleuge abstraction) in a rule-based expert system language.

This difference in focus also leads to a difference in program design methodology. Because of the tight interweaving of data and knowledge in procedural languages, programmers must carefully describe the sequence of execution. However, the explicit separation of data from knowledge in expert system languages requires considerably less rigid control of execution sequence. Typically, an entirely separate piece of code, the inference engine, is used to apply the knowledge to the data. This separation of knowledge and data allows a higher degree of parallelism and modularity.

In choosing a language, a basic question should be whether the problem is knowledge or intelligence intensive. Expert systems rely on a great deal of specialized knowledge or expertise to solve a problem, while AI emphasizes a problem solving approach. Expert systems often rely on a pattern matching within a restricted knowledge domain to guide their execution while AI generally concentrates on searching paradigms in less restricted domains.

The customary way of defining the need for an expert system program is to decide if you want to program the expertise of a human expert. If such an expert exists and will cooperate, then an expert system approach may be successful.

The road to selecting an expert system language is paved with confusion. A few years ago the choice of an expert system language was fairly straightforward. There were only about a half-dozen languages available, and they were generally free or c ı nominal amount from the universities where they were developed.

However, with the explosive commercial growth in the expert systems field since the 1970's, the selection of a language is no longer so simple. Today there are dozens of languages available ranging in price up to $75,000. While it is still possible to obtain some of the older languages such as OPS5 free or at a nominal cost from the universities where they were developed, there is a price to pay in terms of efficiency, lack of modern features and support.

Besides the confusing choice of the many languages available today, the terminology used to describe the languages is confusing. Some vendors refer to their products as "tools," while others refer to "shells" and still others talk about "integrated environments". For clarity in this book, the terms will be defined as follows:

- **language**: A translator of commands written in a specific syntax. An expert system language will also provide an inference engine to execute the statements of the language. Depending on the implementation, the inference engine may provide forward chaining, backward chaining, or both. Under this language definition, LISP is not an expert system language while PROLOG is. However, it is possible to write an expert system language using LISP, and write AI in PROLOG. For that matter you can even write an expert system or AI language in assembly language. Questions of development time, convenience, maintainability, efficiency and speed determine what language software is written in.
- **tool**: A language plus associated utility programs to facilitate the developn debugging, and delivery of application programs. Utility programs may include text and

graphics editors, debuggers, file management, and even code generators. Cross assemblers may also be provided to port the developed code to different hardware. For example, an expert system may be developed on a Digital Equipment Corp. VAX and then cross-assembled to run on a Motorola 68000. Some tools may even allow the use of different paradigms such as forward and backward chaining in one application.

In some cases, a tool may be integrated with all its utility programs in one environment to present a common interface to the user. This approach minimizes the need for the user to leave the environment to perform a task. For example, a simple tool may not provide facilities for file management and so a user would have to exit the tool to give operating system commands. An integrated environment allows easy exchange of data between utility programs in the environment. Some tools do not even require the user to write any code. Instead, the tool allows a user to enter knowledge by examples from tables or spreadsheets and generate the appropriate code itself.

- **shell**: A special purpose tool designed for certain types of applications in which the user must only supply the knowledge base. The classic example of this is the EMYCIN (empty MYCIN) shell. This shell was made by removing the medical knowledge base of the MYCIN expert system.

MYCIN was designed as a backward chaining system to diagnose disease. By simply removing the medical knowledge, EMYCIN was made which could be used as a ell to contain knowledge about other kinds of consultative systems which use back- ard chaining. The EMYCIN shell demonstrated the reusability of the essential MYCIN software such as the inference engine and user interface. This was a very important step in the development of modern expert system technology because it meant that an expert system would not have to be built from scratch for each new application.

There are many ways of characterizing expert systems such as representation of knowledge, forward or backward chaining, support of uncertainty, hypothetical reasoning, explanation facilities and so forth. Unless a person has built a number of expert systems, it is difficult to appreciate all of these features, especially those found in the more expensive tools. The best way to learn expert systems technology is to develop a number of systems with an easy-to-learn language and then invest in a more sophisticated tool if you need its features.

1.9 ELEMENTS OF AN EXPERT SYSTEM

The elements of a typical expert system are shown in Figure 1-6. In a rule-based system, the knowledge base contains the domain knowledge needed to solve problems coded in the form of rules. While rules are a popular paradigm for representing knowledge, other types of expert systems use different representations, as discussed in ıapter 2.

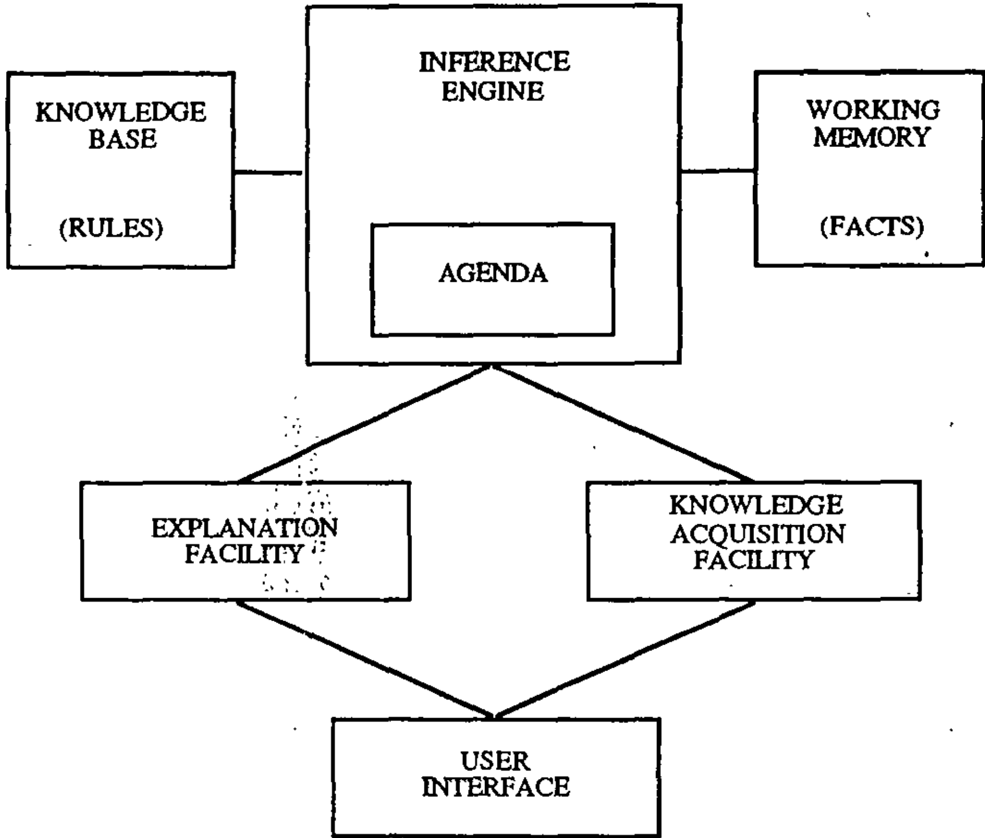


Figure 1-6
Structure of a Rule-Based Expert System

An expert system consists of the following components:

- **user interface** - the mechanism by which the user and the expert system communicate.
- **explanation facility** - explains the reasoning of the system to a user.
- **working memory** - a global database of facts used by the rules.
- **inference engine** -makes inferences by deciding which rules are satisfied by facts, prioritizes the satisfied rules, and executes the rule with the highest priority.
- **agenda** - a prioritized list of rules created by the inference engine, whose patterns are satisfied by facts in working memory.
- **knowledge acquisition facility** - an automatic way for the user to enter knowledge in the system rather than by having the knowledge engineer explicitly code the knowledge.

The knowledge acquisition facility is an an optional feature on many systems. In some expert system tools like KEE and First Class, the tool can learn by rule induction through examples and automatically generate rules. However, the example generally from tabular or spreadsheet type data better suited to decision trees. Ge.

rules constructed by a knowledge engineer can be much more complex than the simple rules from rule induction.

Depending on the implementation of the system, the user interface may be a simple text-oriented display or a sophisticated high-resolution, bit-mapped display. This high-resolution display is commonly used to simulate a control panel with dials and displays.

The knowledge base is also called the **production memory** in a rule-based expert system. As a very simple example, consider the problem of deciding to cross a street. The productions for the two rules are as follows, where the arrows mean that the system will perform the actions on the right of the arrow if the conditions on the left are true.

```
the light is red → stop
the light is green → go
```

The production rules can be expressed in an equivalent pseudocode IF THEN format as:

```
Rule: Red_light
IF
       the light is red
THEN
       stop

Rule: Green_light
IF
       the light is green
THEN
       go
```

Each rule is identified by a name. Following the name is the IF part of the rule. The section of the rule between the IF and THEN part of the rule is called by various names such as the **antecedent**, **conditional part**, **pattern part** or **left-hand-side** (LHS). The individual condition

```
the light is green
```

is called a **conditional element** or a **pattern**.

Some examples of rules from real systems are:

```
MYCIN system for diagnosis of meningitis and
bacteremia (bacterial infections)
IF
    The site of the culture is blood, and
```

```
        The identity of the organism is not known with
           certainty, and
        The stain of the organism is gramneg, and
        The morphology of the organism is rod, and
        The patient has been seriously burned
    THEN
        There is weakly suggestive evidence (.4) that the
            identity of the organism is pseudomonas
```

XCON/R1 for configuring DEC VAX computer systems

```
    IF
        The current context is assigning devices to
           Unibus modules and
        There is an unassigned dual-port disk drive and
        The type of controller it requires is known and
        There are two such controllers, neither of which
           has any devices assigned to it, and
        The number of devices that these controllers can
           support is known
    THEN
        Assign the disk drive to each of the controllers
           and
        Note that the two controllers have been
           associated and that each supports one drive
```

In a rule-based system, the inference engine determines which rule antecedents, if any, are satisfied by the facts. Two general methods of inferencing are commonly used: **forward chaining** and **backward chaining** as the problem solving strategies of expert systems. Other methods used for more specific needs may include means-ends analysis, problem reduction, backtracking, plan-generate-test, hierarchical planning and the least commitment principle, and constraint handling.

Forward chaining is reasoning from facts to the conclusions resulting from those facts. For example, if you see that it is raining before leaving home (the fact), then you should take an umbrella (the conclusion).

Backward chaining involves reasoning in reverse from a hypothesis, a potential conclusion to be proved, to the facts which support the hypothesis. For example, if you have not looked outside and someone enters with wet shoes and an umbrella, your hypothesis is that it is raining. In order to support this hypothesis, you could ask the person if it was raining. If the response is yes, then the hypothesis is proven true and becomes a fact. As mentioned before, a hypothesis can be viewed as a fact whose truth is in doubt and needs to be established. The hypothesis can then be interpreted as a g[illegible] to be proven.

Depending on the design, an inference engine will do either forward or backward chaining. For example, OPS5 and CLIPS are designed for forward chaining while EMYCIN performs backward chaining. Some types of inference engines, such as ART and KEE, offer both. The choice of inference engine depends on the type of problem. Diagnostic problems are better solved with backward chaining while prognosis, monitoring and control are better done by forward chaining.

The working memory may contain facts regarding the current status of the traffic light such as "the light is green" or "the light is red". Either or both of these facts may be in working memory at the same time. If the traffic light is working normally, only one fact will be in memory. However, it is possible that both facts may be in working memory if there is a malfunction in the light. Notice the difference between the knowledge base and working memory. Facts do not interact with one another. The fact "the light is green" has no effect on the fact "the light is red". Instead, our knowledge of traffic lights says that if both facts are simultaneously present, then there is a malfunction in the light.

If there is a fact "the light is green" in working memory, the inference engine will notice that this fact satisfies the conditional part of the green light rule and put this rule on the agenda. If a rule has multiple patterns, then all of its patterns must be simultaneously satisfied for the rule to be placed on the agenda. Some patterns may even be satisfied by specifying the absence of certain facts in working memory.

A rule whose patterns are all satisfied is said to be **activated** or **instantiated**. Multiple activated rules may be on the agenda at the same time. In this case, the inference engine must select one rule for **firing**. The term firing comes from Neurophysiology, the study of the nervous system. An individual nerve cell or **neuron** emits an electrical signal when stimulated. No amount of further stimulation can cause the neuron to fire again for a short time period. This phenomenon is called refraction. Rule-based expert systems are built using refraction in order to prevent trivial loops. That is, if the green light rule kept firing on the same fact over and over again, the expert system would never accomplish any useful work.

Various methods have been invented to provide refraction. In one type of expert system language called OPS5, each fact is given a unique identifier called a **timetag** when it is entered in working memory. After a rule has fired on a fact, the inference engine will not fire on that fact again because its time stamp has been used.

Following the THEN part of a rule is a list of **actions** to be executed when the rule fires. This part of the rule is known as the **consequent** or **right-hand side (RHS)**. When the red light rule fires, it's action "stop" is executed. Likewise, when the green light rule fires, it's action is "go". Specific actions usually include the addition or removal of facts from working memory or printing results. The format of these actions depends on the syntax of the expert system language. For example, in OPS5, ART, and CLIPS, the action to add a new fact called "stop" to working memory would be (assert stop). Because of their LISP ancestry, these languages were designed to require parentheses around patterns and actions.

The inference engine operates in cycles. Various names have been given to describe the cycle such as **recognize-act cycle**, **select-execute cycle**,

situation-response cycle, and **situation-action cycle**. By any name a cycle, the inference engine will repeatedly execute a group of tasks until certain criteria cause execution to cease. The tasks of a cycle for OPS5, a typical expert system shell, are shown in the following pseudocode as **conflict resolution, act, match**, and **check for halt.**

WHILE not done
  Conflict Resolution: If there are activations, then select the one with highest priority else done.
  Act: Sequentially perform the actions on the RHS of the selected activation. Those which change working memory have immediate effect in this cycle. Remove the activation which has just fired from the agenda.
  Match: Update the agenda by checking if the LHS of any rules are satisfied. If so activate them. Remove activations if the LHS of their rules are not satisfied any more.
  Check for Halt: If a halt action is performed or break command given, then done.
END-WHILE
Accept a new user command

Multiple rules may be activated and put on the agenda during one cycle. Also, activations will be left on the agenda from previous cycles unless they are deactivated because their LHS is no longer satisfied. Thus the number of activations on the agenda will vary as execution proceeds. Depending on the program, an activation may always be on the agenda but never selected for firing. Likewise some rules may never become activated. In these cases, the purpose of these rules should be re-examined because the rules are either unnecessary or their patterns were not correctly designed.

The inference engine executes the actions of the highest priority activation on the agenda, then the next highest priority activation and so on until no activations are left. Various priority schemes have been designed into expert system shells. Generally, all shells let the knowledge engineer define the priority of rules.

Agenda conflicts occur when different activations have the same priority and the inference engine must decide on one rule to fire. Different shells have different ways of dealing with this problem. In the original Newell and Simon paradigm, those rules entered first in the system had the highest default priority (Newell 72a). In OPS5, rules with more complex patterns have a higher priority. In ART and CLIPS, rules have the same default priority unless assigned different ones by the knowledge engineer.

At this time, control is returned to the **top-level** command interpreter for the user to give further instructions to the expert system shell. The top-level is the default mode in which the user communicates with the expert system and is indicated by the task "Accept a new user command". It is the top-level which accepts the new command.

The top-level is the user interface to the shell while an expert system application is under development. More sophisticated user interfaces are usually designed for the expert system to facilitate its operation. For example, the expert system may have a user interface for control of a manufacturing plant that shows a block diagram of the plant with a high resolution, bit-mapped color display. Warnings and status messages may appear in flashing colors with simulated dials and gauges. In fact, more effort may go into the design and implementation of the user interface than in the expert system knowledge base, especially in a prototype. Depending on the capabilities of the expert system shell, the user interface may be implemented by rules or in another language called by the expert system.

An explanation facility will allow the user to ask how the system came to a certain conclusion and why certain information is needed. The question of how the system came to a certain conclusion is easy to answer in a rule-based system since a history of the activated rules and contents of working memory can be maintained in a stack. Sophisticated explanation facilities may allow the user to ask "What If" type questions to explore alternate reasoning paths through hypothetical reasoning.

1.10 PRODUCTION SYSTEMS

: of the most popular type of expert system today is the rule-based system. Rules are popular for a number of reasons.

- *Modular nature.* This makes it easy to encapsulate knowledge and expand the expert system by incremental development.
- *Explanation facilities.* It is easy to build explanation facilities with rules since the antecedents of a rule specify exactly what is necessary to activate the rule. By keeping track of which rules have fired, an explanation facility can present the chain of reasoning that led to a certain conclusion.
- *Similarity to the human cognitive process.* Based on the work of Newell and Simon, rules appear to be a natural way of modeling how humans solve problems. The simple IF THEN representation of rules make it easy to explain to experts the structure of the knowledge that you are trying to elicit from them. Other advantages of rules are described in (Hayes-Roth 85).

Rules are a type of production whose origins go back to the 1940's. Because of the importance of rule-based systems, it is worthwhile to examine the development of the rule concept. This will give you a better idea of why rule-based systems are so useful for expert systems.

't Production Systems

Production systems were first used in symbolic logic by Post (Post 43) who originated the name. He proved the important and amazing result that any system of mathematics

or logic could be written as a certain type of production rule system. This result e lished the great capability of production rules for representing major classes of knowledge rather than being limited to a few types. Under the term **rewrite rules**, they are also used in linguistics as a way of defining the grammar of a language. Computer languages are commonly defined using the Backus-Naur Form (BNF) of production rules.

The basic idea of Post was that any mathematical or logic system is simply a set of rules specifying how to change one string of symbols into another set of symbols. That is, given an input string, the antecedent, a production rule could produce a new string, the consequent. This idea is also valid with programs and expert systems where the initial string of symbols is the input data and the output string is some transformation of the input.

As a very simple case, if the input string is "patient has fever" the output string might be "take an aspirin". Note that there is no meaning attached to these strings. That is, the manipulations of the strings is based on syntax and not any semantics or understanding of what a fever, aspirin, and patient represent. A human knows what these strings in terms of the real-world mean but a Post production system is just a way of transforming one string into another. A production rule for this example could be

```
Antecedent → Consequent
person has fever → take aspirin
```

where the arrow indicates the transformation of one string into another. We can interpret this rule in terms of the more familiar IF THEN notation as

```
IF person has fever THEN take aspirin
```

The production rules can also have multiple antecedents. For example,

```
person has fever AND
  fever is greater than 102 → see doctor
```

Note that the special connective AND is not part of the string. The AND indicates that the rule has multiple antecedents.

A Post production system consists of a group of production rules, such as the following (where the numbers in parentheses are for our discussion).

```
(1)   car won't start → check battery
(2)   car won't start → check gas
(3)   check battery AND battery bad → replace battery
(4)   check gas AND no gas → fill gas tank
```

If there is a string "car won't start", the rules (1) and (2) may be used to generate the strings "check battery" and "check gas". However, there is no control mechanism that applies both these rules to the string. Only one rule may be applied, both or none. If there is another string "battery bad" and a string "check battery", then rule (3) may be applied to generate the string "replace battery".

There is no special significance to the order in which rules are written. The rules of our example could also have been written in the following order and it would still be the same system.

```
(4)    check gas AND no gas → fill gas tank
(2)    car won't start → check gas
(1)    car won't start → check battery
(3)    check battery AND battery bad → replace battery
```

Although Post production rules were useful in laying part of the foundation of expert systems, they are not adequate for writing practical programs. The basic limitation of Post production rules for programming is lack of a **control strategy** to guide the application of the rules. A Post system permits the rules to be applied on the strings in any manner because there is no specification given on how the rules should be applied.

As an analogy, suppose you go to the library to find a certain book on expert systems. At the library, you start randomly looking at books on the shelves for the one you want. If the library is fairly large, it will take a very long time to find the book you need. Even if you find the section of books on expert systems, your next random choice could take you to an entirely different section, such as French cooking. The situation becomes even worse if you need material from the first book to help you determine the second book that you need to find. A random search for the second book will also take a long time.

Markov Algorithms

The next advance in applying production rules was made by Markov, who specified a control structure for production systems (Markov 54). A **Markov algorithm** is an ordered group of productions which are applied in order of priority to an input string. If the highest priority rule is not applicable, then the next one is applied and so forth. The Markov algorithm terminates if either (1) the last production is not applicable to a string or (2) a production that ends with a period is applied.

Markov algorithms can also be applied to substrings of a string, starting from the left. For example, the production system consisting of the single rule

```
AB → HIJ
```

when applied to the input string GABKAB produces the new string GHIJKAB. Since the production now applies to the new string, the final result is GHIJKHIJ.

The special character $\wedge$ represents the **null string** of no characters. For example, the production

$A \rightarrow \wedge$

deletes all occurrences of the character A in a string.

Other special symbols represent any single character and are indicated by lower-case letters a, b, c, and so forth. These symbols represent single-character variables and are an important part of modern expert system languages. For example, the rule

$AxB \rightarrow BxA$

will reverse the characters A and B.

The Greek letters α, β, and so forth are used for special punctuation of strings. The Greek letters are used because they are distinct from the alphabet of ordinary letters.

An example of a Markov algorithm that moves the first letter of an input string to the end is shown following (Elson 73). The rules are ordered in terms of highest priority (1), next highest (2) and so forth. The rules are prioritized in the order that they are entered.

(1) $\alpha xy \rightarrow y\alpha x$
(2) $\alpha \rightarrow \wedge$.
(3) $\wedge \rightarrow \alpha$

For the input string ABC, the execution trace is shown in Table 1-11.

Rule	*Success or Failure*	*String*
1	F	ABC
2	F	ABC
3	S	αABC
1	S	BαAC
1	S	BCαA
1	F	BCαA
2	S	BCA

Table 1-11
Execution Trace of a Markov Algorithm

Notice that the α symbol acts analogously to a temporary variable in a conventional programming language. However, instead of holding a value, the α is used as a place holder to mark the progression of changes in the input string. Once its job is done, the α is eliminated by rule 2. The program then ends when rule 2 is applied since there is a period after rule 2.

The Rete Algorithm

Notice that there is a definite control strategy to Markov algorithms with higher priority rules ordered first. As long as the highest priority rule applies, it is used. If not, the Markov algorithm tries lower priority ones. Although the Markov algorithm can be used as the basis of an expert system, it is very inefficient for systems with many rules. The problem of efficiency becomes of major importance if we want to create expert systems for real problems containing hundreds or thousands of rules. No matter how good everything else is about a system, if a user has to wait a long time for a response, the system will not be used. What we really need is an algorithm that knows about all the rules and can apply any rule without having to try each rule sequentially.

A solution to this problem is the **Rete Algorithm** developed by Charles L. Forgy at Carnegie-Mellon University in 1979 for his Ph.D thesis on the OPS (Official Production System at Carnegie-Mellon) expert system shell. The Rete Algorithm is a very fast pattern-matcher that obtains its speed by storing information about the rules in a network. Instead of having to match facts against every rule on every recognize-act cycle, the Rete algorithm only looks for changes in matches on every cycle.This greatly speeds up the matching of facts to antecedents since the static data that doesn't change from cycle to cycle can be ignored. This topic will be discussed further in the c' -s on CLIPS. Fast pattern matching algorithms such as the Rete completed the fc .tion for the practical application of expert systems. Figure 1-7 summarizes the foundations of modern rule-based expert system technologies.

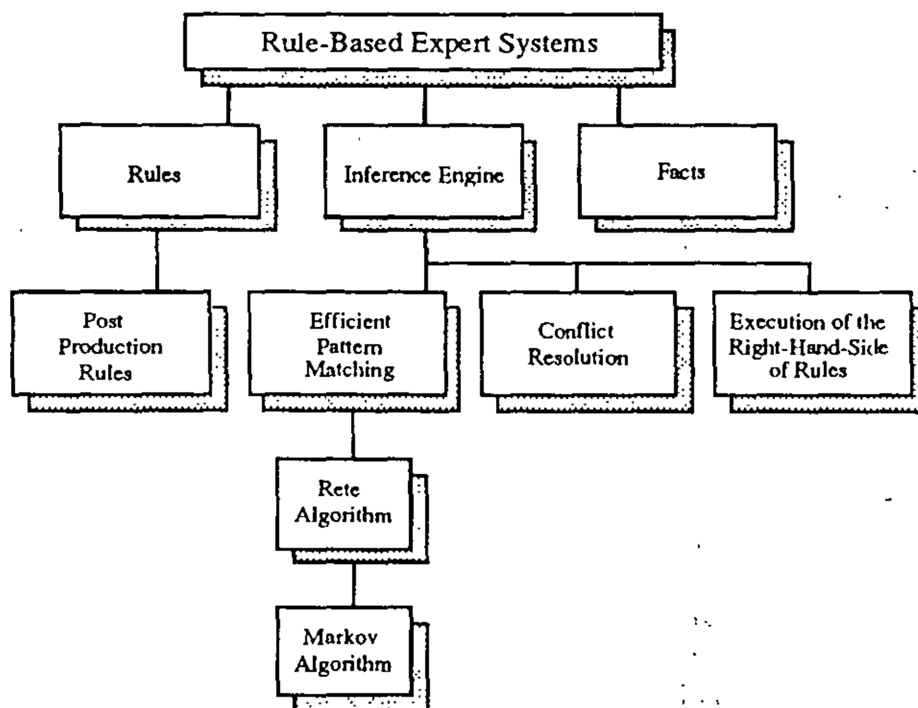


Figure 1-7
Foundations of Modern Rule-based Expert Systems

Different versions of the OPS language and shell have been developed as OPS2, OPS4, and OPS5. A newer commercial version developed by Forgy is OPS83, a very fast shell. However, OPS83 has a significantly different syntax from the OPS5 style and is partly procedural for faster execution.

1.11 PROCEDURAL PARADIGMS

Programming paradigms can be classified as procedural and nonprocedural. Figure 1-8 shows a **taxonomy** or classification of the procedural paradigms in terms of languages. Figure 1-9 shows a taxonomy for nonprocedural paradigms. These figures illustrate the relationship of expert systems to other paradigms. These figures should be considered only as a general guide and not as strict definitions. Some of the paradigms and languages have characteristics that may place them in more than one class. For example, some people consider functional programming a procedural paradigm while others refer to it as declarative (Ghezzi 87a).

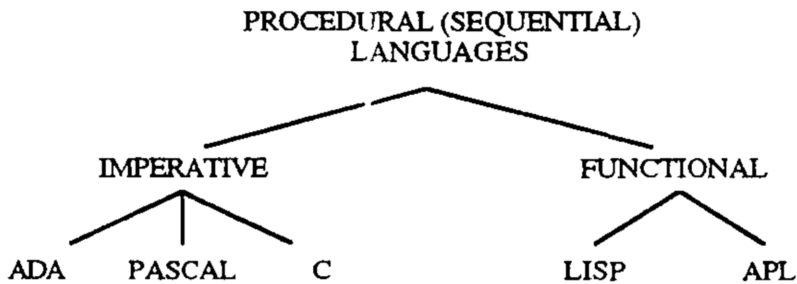


Figure 1-8
Procedural Languages

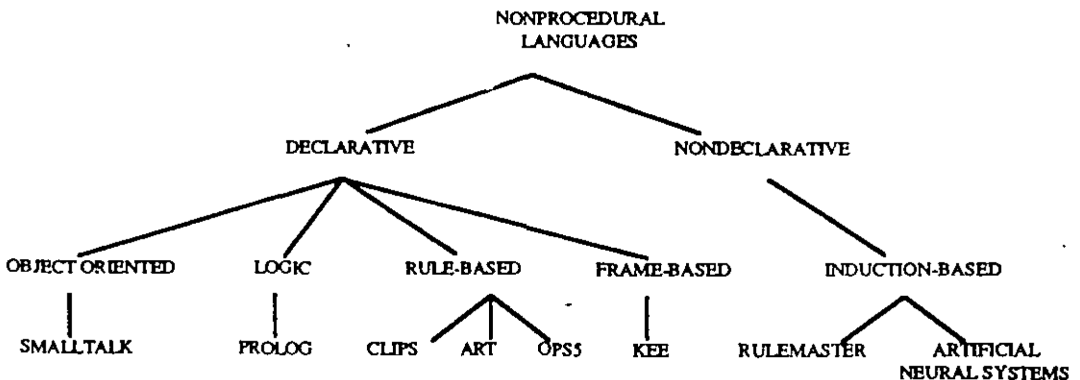


Figure 1-9
Nonprocedural Languages

An **algorithm** is a method of solving a problem in a finite number of steps. The implementation of an algorithm in a program is a **procedural program**. The terms **algorithmic programming,** *procedural programming*, and *conventional programming* are often used synonymously to mean non-AI type programs. A common conception of a procedural program is that it proceeds sequentially, statement by statement, unless a branch instruction is encountered. Another often used synonym for procedural program is **sequential** program. However, the term sequential programming implies too much constraint since all modern programming languages support recursion and so programs may not be strictly sequential.

The distinguishing feature of the procedural paradigm is that the programmer must specify exactly *how* a problem solution must be coded. Even code generators must produce procedural code. In a sense, the use of code generators is **nonprocedural** programming because it removes most or all of the procedural code writing from the programmer. The goal of non-procedural programming is to have the programmer specify *what* the goal is and let the system determine how to accomplish it.

Imperative Programming

The terms **imperative** and **statement-oriented** are used synonymously. Languages such as FORTRAN, Ada, Pascal, Modula-2, COBOL, and BASIC all have the dominant characteristic that statements are imperatives or commands to the computer telling it what to do. Imperative languages developed as a way of freeing the programmer from coding assembly language in the von Neumann architecture. Consequently, imperative languages offer great support to variables, assignment operations, and repetition. These are all low-level operations that modern languages attempt to hide by providing features such as recursion, procedures, modules, packages and so forth. Imperative languages are also characterized by their emphasis on rigid control structure and their associated **top-down** program designs.

A serious problem with all languages is the difficulty of proving the correctness of programs. From the AI standpoint, another serious problem is that imperative languages are not very efficient symbol manipulators. Because the imperative language architecture was molded to fit the von Neumann computer architecture, we have languages that can support number-crunching very well but not symbolic manipulation. However, imperative languages such as C and Ada have been used as the underlying base language to write expert system shells. These languages and the shells built from them run more efficiently and quickly on common general purpose computers than the early shells built using LISP.

Because of their sequential nature, imperative languages are not very efficient for directly implementing expert systems, especially rule-based ones. As an illustration of this problem, consider the problem of encoding the information of a real-world problem with hundreds or thousands of rules. For example, the XCON system used by Digital Equipment Corporation (DEC) to configure computer systems currently has about 7000 rules in its knowledge base. Early unsuccessful attempts were made to code this program in FORTRAN and BASIC before settling on the successful expert systems

approach. The direct way of coding this knowledge in an imperative language would require 7000 IF THEN statements or a very, very long CASE. This style of coding would present major efficiency problems since all 7000 rules need to be searched for matching patterns on every recognize-act cycle. Note that the inference engine and its recognize-act cycle would also have to be coded in the imperative language.

The efficiency of the program could be improved if rules were ordered so that those most likely to be executed were put at the beginning. However, this would require considerable tuning of the system and would change as new rules are added, or old ones deleted and modified. A better method for improving efficiency would be to build a tree of the rule patterns to reduce search time in determining which rules should be activated. Rather than making the programmer manually construct the tree, it should preferably be built automatically by the computer based on the pattern and action syntax of the IF THEN rules. It would also be helpful to have an IF THEN syntax that was more conducive to representing knowledge and had powerful pattern matching tests. This requires the development of a parser to analyze input structure and an interpreter or compiler to execute the new IF THEN syntax.

When all of these techniques for improving efficiency are implemented, the result is a dedicated expert system. If the inference engine, parser and interpreter are removed to provide easy development of other expert systems, they comprise an expert system shell. Of course, instead of doing all of this development from scratch, today it is much easier just to use an existing shell which is documented and extensively tested.

Functional Programming

The nature of **functional programming**, as exemplified by languages such as LISP and APL, is very different from statement-oriented languages with their heavy reliance on elaborate control structures and top-down design. The fundamental idea of functional programming is to combine simple functions to yield more powerful functions. This is essentially a **bottom-up** design in contrast to the common top-down designs of imperative languages.

Functional programming is centered around **functions**. Mathematically, a function is an **association** or rule that maps members of one set, the **domain**, into another set, the **codomain**. An example of a function definition is

```
cube(x) ≡ x*x*x, where x is a real number and
cube is a function with real values
```

The three parts of the function definition are:

(1) the association, $x*x*x$
(2) the domain, real numbers
(3) codomain, real numbers

of the cube function. The symbol ≡ means "is equivalent to" or "is defined as". The following notation is a shorthand way of writing that the cube mapping is from the domain of real numbers, symbolized as $\Re$, to the codomain of real numbers.

```
cube:ℜ → ℜ
```

A general notation for a function *f* that maps from a domain *S* to a domain *T* is $f{:}S \rightarrow T$ (Gersting 82). The **range** of the function *f* is the set of all **images** *f(s)* where *s* is an element of *S*. For the case of the cube function, the images of *s* are *s*s*s* and the range is the set of all real numbers. The range and codomain are the same for the cube function. However, this may not be true for other functions such as the square function, x*x, with domain and codomain of real numbers. Since the range of the square function is only non-negative real numbers, the range and codomain are not the same.

Using set notation, the range of a function can be written as

```
R ≡ {f(s) | s ∈ S}
```

The **curly braces** { } denote a **set**. The **bar**, |, is read as "where". The above state-ʼnt can be read that the range *R* is equivalent to the set of values *f(s)* where every ment *s* is in the set *S*. The association is a set of ordered pairs *(s,t)*, where $s \in S$, $t \in T$, and $t=f(s)$. Every member of *S* must have one and only one element of *T* associated with it. However, multiple *t* values may be associated with a single *s*. As a simple example, every positive number *n* has two square roots, $\pm \sqrt{n}$.

Functions may also be defined recursively as in

```
factorial(n) ≡ n*factorial(n-1)
  where n is an integer and
  factorial is an integer function
```

Recursive functions are commonly used in functional languages such as LISP.

Mathematical concepts and expressions are **referentially transparent** because the meaning of the whole is completely determined from its parts. No synergism is involved between the parts. As an example, consider the functional expression $x+(2*x)$. The result is obviously $3*x$. Both $x+(2*x)$ and $3*x$ give the same results no matter what values are substituted for *x*. Even other functions can be substituted for x and the result is the same. For example, let *h(y)* be some arbitrary function. Then $h(y) + (2 * h(y))$ would still be equivalent to $3*h(y)$.

Now consider the following assignment statement in an imperative computer language such as Pascal.

```
sum := f(x) + x
```

If the parameter x is passed by reference and its value is changed in the function c $f(x)$, what value will be used for x? Depending on how the compiler is written, the value of x might be the original value if it was saved on a stack, or the new value if x was not saved. Another source of confusion occurs if the one compiler evaluates expressions right-to-left while another evaluates left-to-right. In this case, $f(x)+x$ would not evaluate the same as $x+f(x)$ on different compilers even if the same language was used. Other side-effects may occur due to global variables. Thus, unlike mathematical functions, program functions are not referentially transparent.

Functional programming languages were created to be referentially transparent. Five parts make up a functional language:

- **data objects** for the language functions to operate on
- **primitive functions** to operate on the data objects
- **functional forms** to synthesize new functions from other functions
- **application operations** on functions that returns a value and
- **naming procedures** to identify new functions.

Functional languages are generally implemented as interpreters for ease of construction and immediate user response.

In LISP (LISt Processing), data objects are **symbolic expressions** (S-expressions) that are either **lists** or **atoms**, while in **APL** (A Programmin~ Language) the objects are arrays. Examples of lists are

```
(milk eggs cheese)
(shopping (groceries (milk eggs cheese) clothes
(pants)))
()
```

Lists are always enclosed in matching parentheses with spaces separating the elements. The elements of lists can be atoms, such as milk, eggs, and cheese, or embedded lists such as (milk eggs cheese) and (pants). Lists can be split up but atoms cannot. The **empty list**, (), contains no elements and is called **nil**.

There must be some primitive functions provided by the language as a basis for building more complex functions. In the original version of LISP, created by John McCarthy in 1960, there were few primitives, as shown in Table 1-12. Primitives CAR, CDR, CPR, and CTR are acronyms named after the specific hardware registers in the first machine to run LISP. CPR and CTR are now obsolete but the acronyms of the others have remained. Also shown are the **predicates** of LISP. Predicates are special functions that return values representing true and false.

The original version of LISP was called pure LISP because it was purely functional. However, it was also not very efficient for writing programs. Non-functional additions have been made to LISP to increase the efficiency of writing programs. F example, SET acts as the assignment operator, while LET and PROG can be usec

create local variables and execute a sequence of S-expressions. Although these act like functions, they are not functional in the original mathematical sense.

Function	*Predicates*
QUOTE	ATOM
CAR	EQ
CDR	NULL
CPR	
CTR	
CONS	
EVAL	
COND	
LAMBDA	
DEFINE	
LABEL	

Table 1-12
Original LISP Primitives and Functions

Since its creation, LISP has been the leading AI language in the United States. any of the original expert system shells were written in LISP because it is so easy to experiment with LISP. However, conventional computers do not execute LISP very efficiently and execute the shells built using LISP even worse. In order to circumvent this problem, several companies offer machines specifically designed to execute LISP code. These LISP machines use it completely, even as their assembly language. However, the LISP machines cost considerably more than conventional machines and are for single users.

This problem of high cost has an impact on both the development and the **delivery problem**. It is not enough just to develop a great program if it cannot be delivered for use because of high cost. A good development workstation is not necessarily a good delivery vehicle due to speed, power, size, weight, environmental or cost constraints. Some applications may even require that the final code be placed in ROM for reasons of cost and nonvolatility. Putting code into ROM can be a problem with some AI and expert systems tools that require special hardware to run. It's better to consider this possibility in advance rather than have to recode a program later.

An additional problem is that of embedding AI with conventional programming languages such as C, Ada, Pascal and FORTRAN. For example, certain applications which require extensive number-crunching are best done in conventional languages rather than in LISP, or expert systems languages written in LISP or PROLOG.

Unless special provisions are made, expert systems which are written in LISP are erally difficult to embed in anything other than LISP programs. One major consid- .ion in selecting an AI language should be the language in which the tool is written. For reasons of portability, efficiency, and speed, many expert systems tools are now

being written in or converted to C. This also eliminates the problem of requi expensive special hardware for LISP-based applications.

1.12 NONPROCEDURAL PARADIGMS

Nonprocedural paradigms do not depend on the programmer giving exact details for how a problem is to be solved. This is the opposite of the procedural paradigms which specify *how* a function or statement sequence computes. In nonprocedural paradigms, the emphasis is on specifying *what* is to be accomplished and letting the system determine how to accomplish it.

Declarative Programming

The **declarative paradigm** separates the **goal** from the methods used to achieve the goal. The user specifies the goal while the underlying mechanism of the implementation tries to satisfy the goal. A number of paradigms and associated programming languages have been created to implement the declarative model.

Object-oriented Programming

The **object-oriented** paradigm is another case of a paradigm which can be consid partly imperative and partly declarative. The term object-oriented today is used in two different ways. The expression **object-oriented design** is growing in popularity as a programming methodology in imperative programming languages such as Ada and Modula-2. The basic idea is to design a program by considering the data used in the program as objects and then implementing operations on those objects. This is the opposite of top-down design which proceeds by stepwise refinement of a program's control structure. In fact, object-oriented design is essentially what used to be called bottom-up design. Unfortunately, the term bottom-up never sounded quite as impressive as object-oriented.

As an example of object-oriented design, consider the task of writing a program to manage a charge account with an interactive menu (Riley 87). The important data objects are current balance, amount of charge, and amount of payment. Various operations can be defined to act on the data objects. These operations would be add charge, make payment, and add monthly interest. Once all data objects, operations and the menu interface are defined, coding can begin. This object-oriented design methodology is well-suited to a program that has a weak control structure. It would not be as suitable in a program that requires a strong control structure such as a payroll application. For the payroll case there is a definite sequence of steps to follow.

1. Obtain time-card data
2. Account for sick-leave and vacation time
3. Multiply hours worked X rate of pay

4. Multiply overtime hours X overtime rate
5. Add bonus or sales commissions, if applicable
6. Subtract deductions for taxes, insurance, dues and retirement
7. Issue paycheck

Object-oriented design requires no special language features. It can be done in FORTRAN, BASIC, C and so on. Languages such as Ada and Modula-2 support object-oriented design because these languages have features to encapsulate data in modules and packages.

The term **object-oriented programming** was originally used for languages such as Smalltalk, which was specifically designed for objects. The term is now often used to mean programming of an object-oriented design even in a language which has no true object support.

Smalltalk has features to support objects built into the language. In fact, Smalltalk has a programming environment entirely built using objects. Smalltalk is descended from SIMULA 67, a language developed for simulation. SIMULA 67 introduced the concept of **class** which led to the concept of information hiding in modules. A class is essentially a type which is a template that defines the structure of data. In fact, the concept of type in Pascal and Modula-2 came from class. An **instance** of a ...s is a data object that can be manipulated. The term instance has carried over to ...rt systems where it denotes a fact that matches a pattern. Likewise, a rule is said to be instantiated if its LHS is satisfied. The terms activated and instantiated in rule-based systems are synonymous.

Another significant concept that came from SIMULA 67 is **inheritance**. In SIMULA 67, a **subclass** could be defined to inherit the properties of classes. For example, one class may be defined to consist of objects that can be used in a stack and another class defined to be complex numbers. Now a subclass can be easily defined as objects that are complex numbers used in a stack. That is, these objects have inherited properties from both classes above them, called **superclasses**. The concept of inheritance can be extended to organize objects in a hierarchy where objects can inherit from their classes, which can inherit from their classes, and so on. Inheritance is very useful since objects can inherit properties from their classes without the programmer having to specify every property. Many expert systems tools today such as ART and KEE allow inheritance because it is a very powerful tool in constructing large groups of facts.

Logic Programming

One of the first AI applications of computers was in proving logic theorems with the Logic Theorist program of Newell and Simon. This program was first reported at the ...mouth Conference on AI in 1956 and caused a sensation because electronic com-...rs previously had been used only for numeric calculations. Now a computer was actually reasoning the proofs of mathematical theorems, which had been a task that only mathematicians were thought capable of doing.

In the Logic Theorist and its successor, the General Problem Solver (GF Newell and Simon concentrated on trying to implement powerful algorithms that co. solve any problem. While the Logic Theorist was meant only for mathematical theorem proving, GPS was designed to solve any kind of logic problem, including games and puzzles such as chess, Tower of Hanoi, Missionaries and Cannibals, and cryptarithmetic. An example of their famous cryptarithmetic (secret arithmetic) puzzle is

```
  DONALD
+ GERALD
  ROBERT
```

and it is known that D=5. The object is to figure out the arithmetic values of the other letters in the range 0-9.

GPS was the first problem solving program to clearly separate the problem solving knowledge from the domain knowledge. This paradigm of explicitly separating the problem-solving knowledge from the domain knowledge is now used as the basis of expert systems. In expert systems today, the inference engine decides what knowledge should be used and how it should be applied.

Efforts continued to improve mechanical theorem proving. By the early 1970's, it had been discovered that computation is a special case of mechanical, logical deduction (Kowalski 1985). When backward chaining was applied to sentences of the fo "conclusion if conditions", it was powerful enough for significant theorem proving. The conditions can be thought of as representing patterns to be matched as in production rules discussed earlier. Sentences expressed in this form are called Horn clauses after Alfred Horn, who first investigated them. In 1972, the language PROLOG was created by Kowalski, Colmerauer and Roussel to implement logic programming by backward chaining using Horn clauses.

Backward chaining can be used both to express the knowledge in a declarative representation and also control the reasoning process. Typically, backward chaining proceeds by defining smaller **subgoals** that must be satisfied if the initial goal is to be satisfied. These subgoals are then further broken down into smaller subgoals and so forth.

An example of declarative knowledge is the following classic example.

```
All men are mortal
Socrates is a man
```

can be expressed in the Horn clauses

```
someone is mortal
    if someone is a man
Socrates is a man
    if (in all cases)
```

For the sentence about Socrates, the if condition is true in all cases. In other words, the knowledge about Socrates does not require any pattern to match. Contrast this with the mortal case in which someone must be a man for the pattern of the if condition to be satisfied.

Notice that a Horn clause can be interpreted as a procedure that tells how to satisfy a goal. That is, to determine if someone is mortal, it is necessary to determine if someone is a man. As a slightly more complex example,

```
A car needs gas, oil and inflated tires to run
```

can be expressed in a Horn clause as

```
x is a car and runs
  if x has gas and
  if x has oil and
  if x has inflated tires
```

Notice how the problem of determining if a car will run has been reduced to three simpler subproblems or subgoals. Now suppose there is some additional declarative knowledge as follows.

```
The fuel gauge shows not-empty if a car has gas
The dipstick shows not-empty if a car has oil
The air pressure gauge shows at least 20 if a car has
inflated tires
The fuel gauge shows not-empty
The dipstick reads empty
The air pressure gauge shows at least 15
```

These can be translated into the following Horn clauses.

```
x has gas
  if the fuel gauge shows not-empty
x has oil
  if the dipstick shows not-empty
x has inflated tires
  if the air pressure gauge shows at least 20
Fuel gauge is not empty
  if (in all cases)
Dipstick reads empty
Air pressure is at least 15
  if (in all cases)
```

From these clauses, a mechanical theorem prover can prove that the car will :un because there is no oil and insufficient air pressure.

One of the advantages of backward chaining systems is that execution can proceed in parallel. That is, if multiple processors were available they could work on satisfying subgoals simultaneously. This parallel operation can greatly speed up the execution of programs and is one of the reasons that the Japanese chose PROLOG as the programming language for their Fifth Generation Computer project (Moto-Oka 1982). This is an ambitious project to develop the next generation of computers for the 1990's with capabilities far surpassing current models. These machines were to be programmed with declarative techniques, but this approach has run into difficulties. The Fifth Generation project was announced by the Japanese in 1981 with a budget of $850 million dollars. The twenty-six initial goals of the project included natural language translation of Japanese to English and vice-versa, problem-solving and inferencing a thousand times faster than current machines, ability to read handwritten text, understand pictures, intelligent access of huge databases, and to converse in natural language.

PROLOG is more than just a language. At a minimum, PROLOG is a shell since it requires

- an interpreter or inference engine
- a database (facts and rules)
- a form of pattern matching called **unification**
- a **backtracking** mechanism to pursue alternate subgoals if a search to s: y a goal is unsuccessful

More elaborate versions of PROLOG such as TurboPROLOG from Borland are tools because of all the enhancements provided.

As an example of backward chaining, suppose you can pay for the oil to make the car run if you have cash or a credit card. One subgoal is checked to see if you have cash. If there is no fact that you do have cash, the backtracking mechanism will then explore the other subgoal to see if you have a credit card. If you have a credit card, the goal of paying for oil can be satisfied. Notice that the absence of a fact to prove a goal is just as effective, although perhaps less efficient, than a negative fact such as "Dipstick reads empty." Either negative facts or missing facts can cause a goal to be unsatisfied.

If the backtracking and pattern matching mechanisms are not needed by the problem, then the programmer must work around them or code in a different language. One of the advantages of logic programming is executable specifications. That is, specifying the requirements of a problem by Horn clauses produces an executable program. This is very different from conventional programming in which the requirements document does not look at all like the final executable code. Newer imperative languages such as Ada attempt to circumvent this problem by offering the capability of using Ada itself as the program description language (PDL). The PDL can be used as a s' 'on on which to build the rest of the program. If the PDL does not work right, then .ier will the program.

Unlike production rule systems, the order in which subgoals, facts and rules are entered in a PROLOG program has significant effects. Efficiency and therefore speed are affected by the way that PROLOG searches its database. However, there are programs that execute correctly if subgoals, facts, and rules are entered one way but go into an infinite loop or have a run-time error if the order changes (Ghezzi 87b).

Expert Systems

Expert systems can be considered declarative programming because the programmer does not specify how a program is to achieve its goal at the level of an algorithm. For example, in a rule-based expert system, any of the rules may become activated and put on the agenda if its LHS matches the facts. The order that the rules were entered does not affect which rules are activated. Thus, the program statement order does not specify a rigid control flow. Other types of expert systems are based **frames**, discussed in chapter 2, and **inference nets** discussed in chapter 4.

There are a number of differences between expert systems and conventional programs. Table 1-13 lists some typical differences.

Characteristic	*Conventional Program*	*Expert System*
Control by ...	Statement order	Inference engine
Control and data	Implicit integration	Explicit separation
Control Strength	Strong	Weak
Solution by ...	Algorithm	Rules and inference
Solution search	Small or none	Large
Problem solving	Algorithm is correct	Rules
Input	Assumed correct	Incomplete, incorrect
Unexpected input	Difficult to deal with	Very responsive
Output	Always correct	Varies with problem
Explanation	None	Usually
Applications	Numeric, file, and text	Symbolic reasoning
Execution	Generally sequential	Opportunistic rules
Program design	Structured design	Little or no structure
Modifiability	Difficult	Reasonable
Expansion	Done in major jumps	Incremental

Table 1-13
Some Typical Differences between Conventional Programs and Expert Systems

Expert systems are often designed to deal with uncertainty because reasoning is one of the best tools that we have discovered for dealing with uncertainty. The uncertainty ay arise in the input data to the expert system and even the knowledge-base itself. At first this may seem surprising to people used to conventional programming. However, much of human knowledge is heuristic, which means that it may only work correctly part of the time. In addition, the input data may be incorrect, incomplete, inconsistent

and have other errors. Algorithmic solutions are not capable of dealing with siti is like this because an algorithm guarantees the solution of a pi oblem in a finite sei... of steps.

Depending on the input data and the knowledge-base, a expert system may come up with the correct answer, a good answer, a bad answer or n answer at all. While this may seem shocking at first, the alternative is no answer all the time. Again, the important thing to keep in mind is that a good expert syste n will perform no worse than the best problem-solver for problems like this—a hu nan expert—and may do better. If we knew an efficient algorithmic method that was better than an expert system, we would use it. The important thing is to use the be t, and perhaps only, tool for the job.

Nondeclarative Programming

Nondeclarative paradigms are becoming popular. These new paradigms are being increasingly used for a wide variety of applications. They can be used in stand-alone applications or in conjunction with other paradigms.

Induction-based Programming

An application of AI that is attracting interest is **induction-based** programming. In this paradigm, the program learns by example. One application to which this par n has been applied is database access. Instead of the user having to type in the specific values for one or more fields for a search, it is only necessary to select one or more appropriate example fields with those characteristics. The database program infers the characteristics of he data and searches the database for a match.

Expert syst n tools such as 1st-Class and KDS offer induction-learning by which they accept exam les and case studies and automatically generate rules.

1.13 ARTIFICIAL NEURAL SYSTEMS

A new development in programming paradigms arose in the 1980's called **artificial neural systems** (ANS), based on how the brain processes information. This paradigm is sometimes called **connectionism** because it models solutions to problems by training simulated neurons connected in a network. Many researchers are investigating neural nets. The nets hold great potential as the front-end of expert systems that require massive amounts of input from sensors as well as real-time response.

The Traveling Salesman Problem

ANS has had remarkable success in providing real-time response to complex p...rn recognition problems. In one case, a neural net running on an ordinary microco r obtained a very good solution to the Traveling Salesman problem in 0.1 seconds

compared to the optimum solution that required an hour of CPU time on a mainframe (Port 86). The Traveling Salesman problem is important because it is the classic problem faced in optimizing the routing of signals in a telecommunications system. Optimizing routing is important in minimizing the travel time, and thus efficiency and speed.

The basic Traveling Salesman problem is to compute the shortest route through a given list of cities. Table 1-14 shows the possible routes for one to four cities. Notice that the number of routes is proportional to the factorial of the number of cities minus one, (N-1)!.

Number of Cities	*Routes*
1	1
2	1–2-1
3	1–2–3-1 1–3–2-1
4	1–2–3–4-1 1–2–4–3-1 1–3–2–4-1 1–3–4–2-1 1–4–2–3-1 1–4–3–2-1

Table 1-14
Traveling Salesman Problem Routes

While there are 9! = 362880 routes for ten cities, there are 29! = 8.8E30 possible routes for thirty cities. The Traveling Salesman problem is a classic example of **combinatorial explosion** because the number of possible routes increases so rapidly that there are no practical solutions for realistic numbers of cities. If it takes 1 hour of a mainframe CPU time to solve for thirty cities, it will take 30 hours for thirty-one cities and 330 hours for thirty-two cities. These are actually very small numbers when compared to the thousands of telecommunications switches and cities that are used in routing of data packets and real items.

A neural net can solve the ten-city case just as fast as the thirty-city case while a conventional computer takes much longer. For the ten-city case, the neural net came up with one of the two best routes and for the thirty-city case, it came up with one of the best 100,000,000 routes. This is more impressive if it is realized that this route is in the top 1E-22 of the best solutions. Although neural nets may not always give the optimum answer, they can provide a best guess in real-time. In many cases, a 99.999999999999999999 % correct answer in one millisecond is better than a 100 % orrect answer in thirty hours.

Elements of an ANS

An ANS is basically an analog computer that uses simple processing elements connected in a highly parallel manner. The processing elements perform very simple Boolean or arithmetic functions on their inputs. The key to the functioning of an ANS is the **weights** associated with each element. It is the weights which represent the information stored in the system.

A typical artificial neuron is shown in Figure 1-10. The neuron may have multiple inputs but only one output. The human brain contains about 10^{10} neurons and one neuron may have thousands of connections to another. The input signals to the neuron are multiplied by the weights and are summed to yield the total neuron input, I. The weights can be represented as a matrix and identified by subscripts.

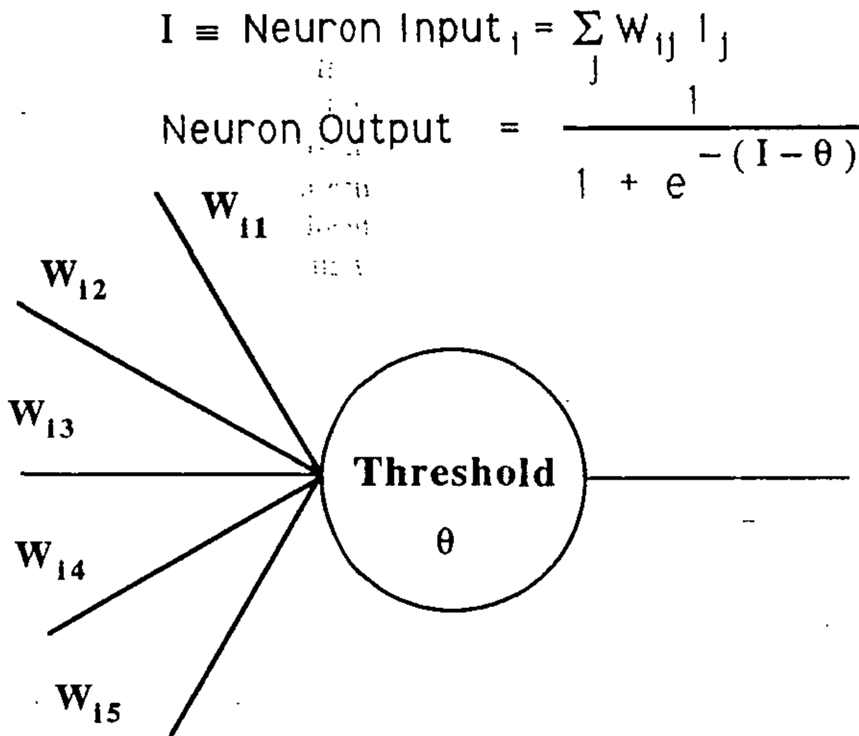


Figure 1-10
A Neuron Processing Element

The neuron output is often taken as a **sigmoid function** of the input. The sigmoid is representative of real neurons which approach limits for very small and very large inputs. The sigmoid is called an **activation function**, and a commonly used function is $(1+e^{-x})^{-1}$. Each neuron also has an associated **threshold value**, θ, which is subtracted from the total input I. Figure 1-11 shows an ANS which can compute the **exclusive-OR (XOR)** of its inputs using a technique called **back-propagati**

The XOR gives a true output only when its inputs are not all true or not all false. The number of nodes in the hidden layer will vary depending on the application and design.

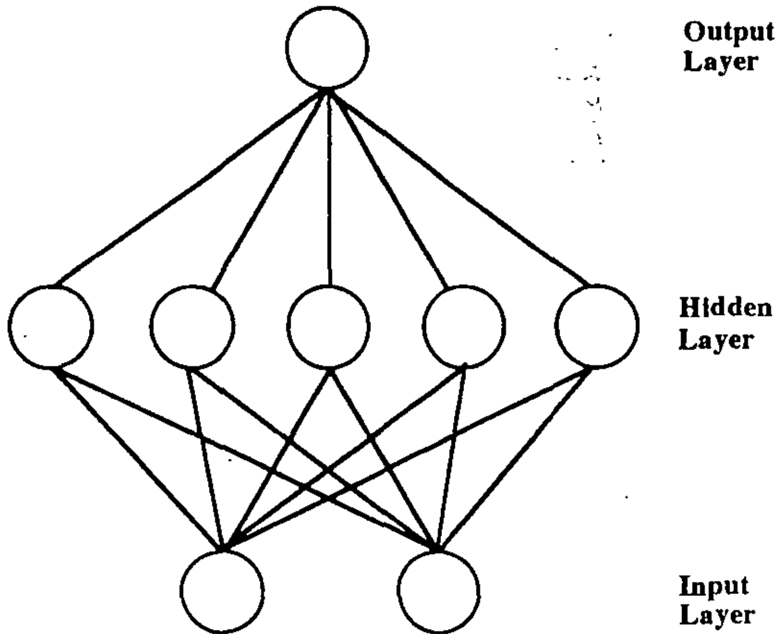


Figure 1-11
A Back-propagation Net

Neural nets are not programmed in the conventional sense. There are about thirteen known neural net learning algorithms, such as **counter-propagation** and back-propagation, to train nets. The programmer "programs" the net by simply supplying the input and corresponding output data. The net learns by automatically adjusting weights in the network which connect the neurons. The weights and the threshold values of neurons determine the propagation of data through the net and so its correct response to the training data. Training the net to the correct responses may take hours or days depending on the number of patterns that the net must learn, the hardware, and software. However, once the learning is accomplished, the net responds very quickly.

If software simulation is not fast enough, ANS can be fabricated in chips for real-time response. Once the network has been trained and the weights determined, a chip can be constructed. AT&T and other companies are fabricating experimental neural net chips containing hundreds of neurons. In the next few years, it is very likely that chips will be fabricated containing thousands of neurons.

Characteristics of ANS

ANS architecture is very different from a conventional computer architecture. In a conventional computer, it is possible to correlate discrete information with memory cells. For example, a Social Security number could be stored as ASCII code in a contiguous

group of memory cells. By examining the contents of this contiguous grou the Social Security number could be directly reconstructed. This reconstruction is possible because there is a one-to-one relationship between each character of the Social Security number and the memory cell that contains the ASCII code of that character.

ANS are modeled after current brain theories in which information is represented by the weights. However, there is no direct correlation between a specific weight and a specific item of stored information. This distributed representation of information is similar to that of a hologram in which the lines of the hologram act as a diffraction grating to reconstruct the stored image when laser light is passed through.

A neural net is a good choice when there is much empirical data and no algorithm exists which provides sufficient accuracy and speed. ANS offers several advantages compared to the storage of conventional computers.

- *Storage is very* ***fault tolerant****.* Portions of the net can be removed and there is only a degradation in quality of the stored data. This occurs because the information is stored in a distributed manner.
- *The quality of the stored image degrades gracefully in proportion to the amount of net removed.* There is no catastrophic loss of information. The storage and quality features are also characteristic of holograms.
- *Data is naturally stored in the form of* ***associative*** *memory.* An associative memory is one in which partial data is sufficient to recall all of the complete stored information. This contrasts with conventional memory in which data is recalled by specifying the address of the data to be recalled. A partial or noisy input may still elicit the complete original information.
- *Nets can extrapolate and interpolate from their stored information.* Training teaches a net to look for significant features or relationships in the data. Afterwards, the net can extrapolate to suggest relationships on new data. In one experiment (Hinton 86), a neural net was trained on the family relationships of twenty-four hypothetical people. Afterwards, the net could also answer correctly relationships about which it had not been trained.
- *Nets have* ***plasticity****.* Even if a number of neurons are removed, the net can be retrained to its original skill level if enough neurons remain. This is also a characteristic of the brain in which portions can be destroyed and the original skill levels can be relearned in time.

These characteristics make ANS very attractive for robot spacecraft, oil field equipment, underwater devices, process control and other applications that need to function a long time in a hostile environment without repair. Besides the issue of reliability, ANS offer the potential of low maintenance cost because of plasticity. Even if hardware repair can be done, it will probably be more cost-effective to reprogram the neural net than replace it.

ANS are generally not well-suited for applications that require number-crunching or an optimum solution. Also, if a practical algorithmic solution exists, an ANS is not a good choice.

Developments in ANS Technology

The origins of ANS started with the mathematical modeling of neurons by McCulloch and Pitts in 1943 (McCulloch 43). An explanation of learning by neurons was given by Hebb in 1949 (Hebb 49). In Hebbian learning, a neuron's efficiency in triggering another neuron increases with **firing**. The term firing means that a neuron emits an electrochemical impulse which can stimulate other neurons connected to it. There is evidence that the conductivity of connections between neurons at their connections, called **synapses**, increases with firing. In ANS, the weight of connections between neurons is changed to simulate the changing conductance of natural neurons.

In 1961, Rosenblatt published an influential book dealing with a new type of artificial neuron system he had been investigating called a **perceptron** (Rosenblatt 61). The perceptron was a remarkable device that showed capabilities for learning and pattern recognition. It basically consisted of two layers of neurons and a simple learning algorithm. The weights had to be manually set in contrast to modern ANS that set the weights themselves based on training. Many researchers entered the field of ANS and began studying perceptrons during the 1960's.

The early perceptron era came to an end in 1969 when Minsky and Papert published a book called *Perceptrons* that showed the theoretical limitations of perceptrons as a general computing machine (Minsky 69). They pointed out a deficiency of the perceptron in being able to compute only 14 of the 16 basic logic functions, which means that a Perceptron is not a general purpose computing device. In particular, they proved a perceptron could not recognize the exclusive-OR. Although they had not seriously investigated multiple layer ANS, they gave the pessimistic view that multiple layers would probably not be able to solve the XOR problem. Government funding of ANS research ceased in favor of the symbolic approach to AI using languages such as LISP and algorithms. New methods of representing symbolic AI information by frames, invented by Minsky, became popular during the 1970's. Further work on perceptrons has continued with new types able to overcome Minsky's objections (Reece 87). Because of their simplicity, perceptrons and other ANS are easy to construct with modern integrated circuit technology.

ANS research continued on a small scale in the 1970's. However, the field finally entered a renaissance starting with the work of Hopfield in 1982 (Hopfield 82). He put ANS on a firm theoretical foundation with the two-layer Hopfield Net and demonstrated how ANS could solve a wide variety of problems. The general structure of a Hopfield Net is shown in Figure 1-12. In particular, he showed how an ANS could solve the Traveling Salesman Problem in constant time as compared to the combinatorial explosion encountered by conventional algorithmic solutions. An electronic circuit form of an ANS could solve the Traveling Salesman Problem in 1 μ second. Other combinatorial optimization problems can easily be done by ANS such as the four-color map, the Euclidean-match (Hopfield 86b), and the transposition code (Tank 85).

An ANS that can easily solve the XOR problem is the **back-propagation** net, also known as the **generalized delta rule** (Rumelhart 86). The back-propagation net is commonly implemented as a three-layer net, although additional layers can be speci-

fied. The layers between the input and output layers are called **hidden layers** becau only the input and output layers are visible to the external world. An important theoretical result from mathematics, the Kolmogorov Theorem, can be interpreted as proving that a three-layer network with n inputs and 2n + 1 neurons in the hidden layer can map any continuous function (Hecht-Nielsen 86).

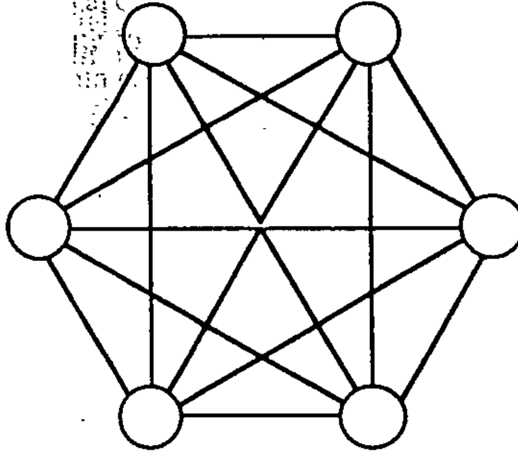


Figure 1-12
A Hopfield Artificial Neural Net

Applications of ANS Technology

A significant example of learning by back-propagation was demonstrated by a neural net that learned correct pronunciation of words from text (Sejnowski 86). The ANS was trained by correcting its output using a Digital Equipment Corp. text to speech device called DECTalk. It required twenty years of linguistic research to devise rules for correct pronunciation used by DECTalk. The ANS taught itself equivalent pronunciation skills overnight by simply listening to the correct pronunciation of speech from text. No linguistic skills were programmed into the ANS.

Investigations of the ANS are under way for recognition of radar targets by electronic and optical computers (Farhat 86). New implementations of neural nets using optical components promise optical computers with speeds millions of times faster than electronic ones. Optical implementation of ANS is attractive because of the inherent parallelism of light. That is, light rays do not interfere with one another as they travel. Huge numbers of photons can easily be generated and manipulated by optical components such as mirrors, lenses, high-speed programmable spatial light modulators, arrays of optical bistable devices that can function as optical neurons, and diffraction gratings. Optical computers designed as ANS appear to be complementary to one another.

Commercial Developments in ANS

A number of new companies and existing firms have been organized to develop ANS technology and products. Nestor markets an ANS product called NestorWriter that can recognize handwritten input and convert it to text using a PC. Other companies such as TRW, SAIC, HNC, Synaptics, Neural Tech, Revelations Research and Texas Instruments market a variety of ANS simulators and hardware accelerator boards to speed up learning. One of the best bargains for getting started in neural computing is volume 3 of Rumelhart's books on Parallel Distributed Processing available from the MIT Press. The book describes a half-dozen ANS simulators and also includes a diskette with software for an IBM PC compatible machine for $27.50. By purchasing a 386 accelerator card for an IBM XT or compatible, a person can have a very powerful system for ANS at low cost.

1.14 CONNECTIONIST EXPERT SYSTEMS AND INDUCTIVE LEARNING

It is possible to build expert systems using ANS. In one system, the ANS is the knowledge-base constructed by training examples from medicine for disease (Gallant 88). In this system, the expert system tries to classify a disease from its symptoms one of the known diseases that the system has been trained on. An inference engine called MACIE (Matrix Controlled Inference Engine) was designed that uses the ANS knowledge-base. The system uses forward chaining to make inferences and backward chaining to query the user for any additional data needed to produce a solution. Although an ANS by itself cannot explain why its weights are set to certain values, MACIE can interpret the ANS and generate IF THEN rules to explain its knowledge.

An ANS expert system such as this uses **inductive learning**. That is, the system **induces** the information in its knowledge-base by example. Induction is the process of inferring the general case from the specific. Besides ANS, there are a number of commercially available expert systems shells that explicitly generate rules from examples. The goal of inductive learning is to reduce or eliminate the knowledge acquisition bottleneck. By placing the burden of knowledge acquisition on the expert system, the development time may be reduced and the reliability may be increased if the system induces rules that were not known by a human.

1.15 SUMMARY

In this chapter we have reviewed the problems and developments which have led to expert systems. These problems that expert systems are used for are generally not able by conventional programs because they lack a known or efficient algorithm. e expert systems are knowledge-based, they can be effectively used for real-world problems that are ill-structured and difficult to solve by other means.

The advantages and disadvantages of expert systems were also discussed in t' context of selecting an appropriate problem domain for an expert system applicati Criteria for selecting appropriate applications were given.

The essentials of an expert system shell were discussed with reference to rule-based expert systems. The basic recognize-act inference engine cycle was described and illustrated by a simple rule example. Finally, the relationship of expert systems to other programming paradigms was described in terms of the appropriate domain of each paradigm. The important point of all this is the concept that expert systems should be viewed as another programming tool that is suitable for some applications and unsuitable for others. Later chapters will describe the features and suitability of expert systems in much more detail.

PROBLEMS

1-1. Identify a person other than yourself who is considered an expert or very knowledgeable. Interview this expert and discuss how well this person's expertise would be modeled by an expert system in terms of each criterion in the section "Advantages of Expert Systems."

1-2. a) Write ten nontrivial rules expressing the expert of problem 1's knowledge.

b) Write a program that will give your expert's advice. Include test results to show that each of the ten rules gives the correct advice. For ease of programming, you may allow the user to provide input from a menu.

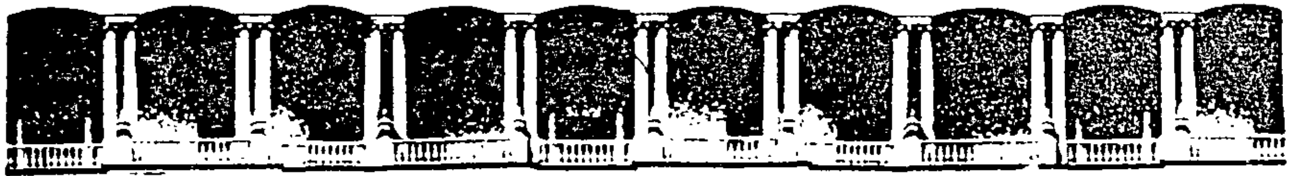
1-3. a) In Newell and Simon's book, *Human Problem Solving*, they mention the 9-Dot Problem. Given 9 dots arranged as follows, how can you draw four lines through all the dots without (a) lifting your pencil from the paper and (b) crossing any dot? (Hint: you can extend the line past the dots)

• • •
• • •
• • •

b) Explain your reasoning (if any) in finding the solution and whether an expert system or some other type of program would be a good paradigm to solve this type of problem.

1-4. Write a program that can solve cryptarithmetic problems. Show the result for the following problem, where D=5.

```
  DONALD
+ GERALD
  ROBERT
```



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS ABIERTOS

ROBOTS MÓVILES

EFFICIENCY IN RULE - BASED LANGUAGES

EXPOSITOR : DR. JESÚS SAVAGE CARMONA
1998

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtemoc 06000 Mexico, D.F. APDO. Postal M-2285
Telefonos 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

CHAPTER 11 EFFICIENCY IN RULE-BASED LANGUAGES

NTRODUCTION

er provides many techniques for increasing the efficiency of a rule-based em which uses the Rete Pattern Matching Algorithm. The reasons for need- cient pattern matching algorithm are discussed before the Rete Algorithm is Several techniques for writing rules more efficiently are discussed.

IE RETE PATTERN MATCHING ALGORITHM

languages such as CLIPS, ART, OPS5, and OPS83 use a very efficient or matching facts against the patterns in rules to determine which rules have itions satisfied. This algorithm is called the **Rete Pattern Matching** 1. Writing efficient CLIPS rules does not require an understanding of the rithm. However, an understanding of the underlying algorithm used in other rule-based languages makes it easier to understand why writing rules more efficient than writing them another way.

ierstand why the Rete Algorithm is efficient, it will be helpful to look at the matching facts to rules in general and examine other algorithms which are ient. Figure 11-1 shows the problem addressed by the Rete Algorithm.

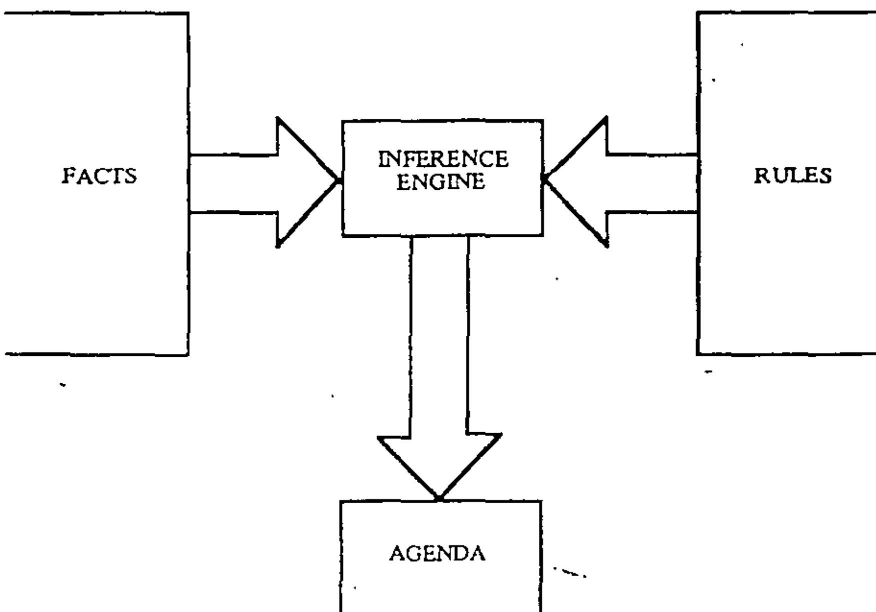


Figure 11-1
Pattern Matching: Rules and Facts

If the matching process only has to occur once, then the solution to the problem straightforward. The inference engine can examine each rule and then search the set of facts to determine if the rule's patterns have been satisfied. If the rule's patterns are satisfied, then the rule can be placed on the agenda. Figure 11-2 shows this approach of having the rules look for the facts needed to match their conditions.

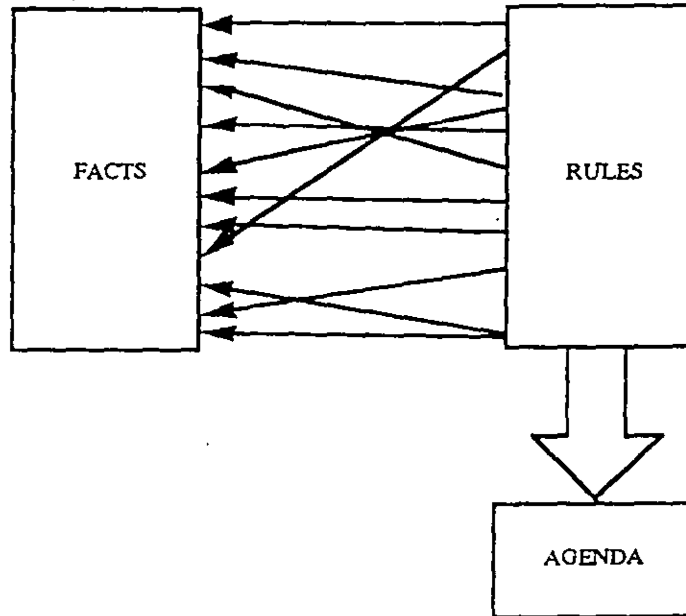


Figure 11-2
Rules Searching for Facts

In rule-based languages, however, the matching process takes place repeatedly. Normally, the fact-list will be modified during each cycle of execution. New facts may be added to the fact-list or old facts may be removed from the fact-list. These changes may cause previously unsatisfied patterns to be satisfied or previously satisfied patterns to become unsatisfied. The problem of matching now becomes an ongoing process. During each cycle, as facts are added and removed, the set of rules that are satisfied must be maintained and updated.

Having the inference engine check each rule to direct the search for facts after each cycle of execution provides a very simple and straightforward technique for solving this problem. The primary disadvantage of such a technique is that it can be very slow. Most rule-based expert systems exhibit a property called **temporal redundancy.** Typically, the actions of a rule will only change a few facts in the fact-list. That is, the facts in the expert system change slowly over time. Each cycle of execution may see only a small percentage of facts either added or removed and so only a small percentage of rules are typically affected by the changes in the fact-list. Thus having the rules drive the search for needed facts requires a lot of unnecessary computation since most of the rules are likely to find the same facts in the current cycle as found in the last cycle. The inefficiency of this approach is shown in Figure 11-3. The shaded area represents the changes that have been made to the fact-list. Unnecessary recomputation could be

by remembering what has already matched from cycle to cycle and computing
: changes necessary for the newly added or removed facts as shown in Figure
is the rules that remain static and the facts that change. Thus the facts should
rules and not the other way around.

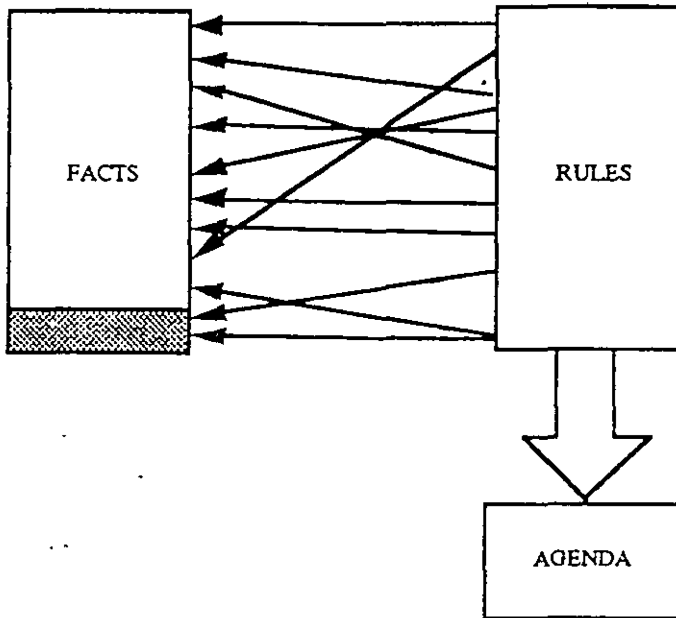


Figure 11-3
Unnecessary Computations when Rules Search for Facts

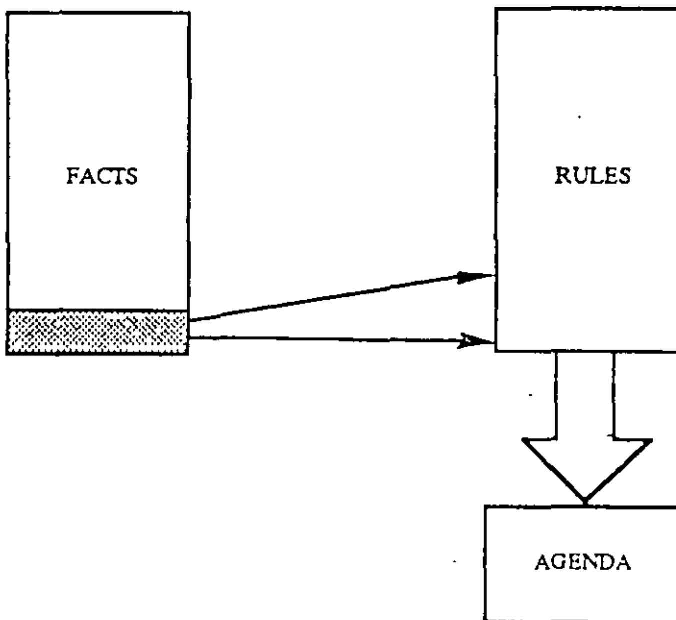


Figure 11-4
Facts Searching for Rules

The Rete Pattern Matching algorithm is designed to take advantage of the tempo. redundancy exhibited by rule-based expert systems. It does this by saving the state of the matching process from cycle to cycle and recomputing the changes in this state only for the changes that occur in the fact-list. That is, if a set of patterns finds two of three required facts in one cycle, a check does not have to be made in the next cycle for the two facts that have already been found. Only the third fact is of interest. The state of the matching process is updated only as facts are added and removed. If the number of facts added and removed is small compared to the total number of facts and patterns, then the process of matching will proceed quickly. As a worst case, if all the facts were to be changed, then the matching process would work as if all the facts were compared against all of the patterns.

If only updates to the fact-list are processed, then each rule must remember what has already matched it. That is, if a new fact has matched the third pattern of a rule, then information about the matches for the first two patterns must be available to finish the matching process. This type of state information indicating the facts that have matched previous patterns in a rule is called a **partial match**. A partial match for a rule is any set of facts which satisfy the rule's patterns beginning with the first pattern of the rule and ending with any pattern up to and including the rule's last pattern. Thus, a rule with three patterns would have partial matches for the first pattern, the first and second patterns, and the first, second, and third patterns. A partial match of all of the patterns of a rule will also be an activation. The other type of state information sa is called a **pattern match**. A pattern match occurs when a fact has satisfied a single pattern in any rule without regard to variables in other patterns that may restrict the matching process.

The primary disadvantage of the Rete Pattern Matching Algorithm is that it is very memory intensive. Simply comparing all of the facts to all of the patterns requires no memory. Saving the state of the system using pattern matches and partial matches, however, can consume considerable amounts of memory. In general, this tradeoff of memory for speed is worthwhile. It is important to note that the Rete Algorithm does not perform a simple search in attempting to match facts against rules. Since memory is used to save both pattern matches and partial matches for the rules, a poorly written rule can not only run slowly, but it can use up considerable amounts of memory. Like errors in procedural languages such as a recursive loop that never bottoms out or nested loops that have a combinatorial number of iterations, this is a commonly made error in rule-based programming.

The Rete Algorithm also improves the efficiency of rule-based systems by taking advantage of **structural similarity** in the rules. Structural similarity refers to the fact that many rules often contain similar patterns or groups of patterns. The Rete Algorithm takes advantage of this feature to increase efficiency by pooling common components together so that they do not have to be computed more than once.

PATHS AND TRAJECTORIES

A **PATH** WILL BE DEFINED AS A LINE IN 2D SPACE THAT IS TO BE TRAVERSED.

A **TRAJECTORY** WILL BE DEFINED AS A TIME HISTORY OF POSITION, VELOCITY, AND ACCELERATION FOR EACH DEGREE OF FREEDOM.

FOR OUR ROBOTS, THE DEGREES OF FREEDOM ARE THE MOTIONS OF THE DRIVE MOTORS AND THE STEER MOTORS.

****

A FIRST PROBLEM IS ASSOCIATED WITH THE HUMAN AND/OR ROBOTIC INTERFACE: NAMELY, HOW TO SPECIFY A PATH (e. g. in the CARTOGRAPHER) AND HOW TO GENERATE A TRAJECTORY.

THE SPECIFICATION SHOULD BE AS "EASY" AS POSSIBLE WITH THE DETAILS LEFT TO THE ON-BOARD AND/OR OFF-BOARD COMPUTERS.

FOR EXAMPLE, THE USER MIGHT JUST SPECIFY THE DESIRED GOAL POSTURE AND THE STARTING POSTURE AND THE "SYSTEM" SHOULD DECIDE ON THE SHAPE OF THE PATH TO GET THERE AND GENERATE THE TRAJECTORIES.

****

THIS PROBLEM CAN BE THOUGHT OF AS A **FORWARD** COMPUTATION AND AN **INVERSE** COMPUTATION.

THE FORWARD COMPUTATION CALCULATES THE ROBOT POSITIONS AND ORIENTATIONS FROM SOME SPECIFIED MOTIONS OF THE MOTORS.

THE INVERSE COMPUTATION CALCULATES THE MOTOR MOTIONS FROM SOME SPECIFIED PATH AND/OR TRAJECTORY OF THE ROBOT.

A TRAJECTORY COULD BE CHARACTERIZED AS A "HISTORICAL TRAJECTORY" OR A "PLANNED" TRAJECTORY.

A HISTORICAL TRAJECTORY IS A TIME HISTORY OF POSITION, SPEED, AND ACCELERATION THAT HAS OCCURRED (IN THE PAST).

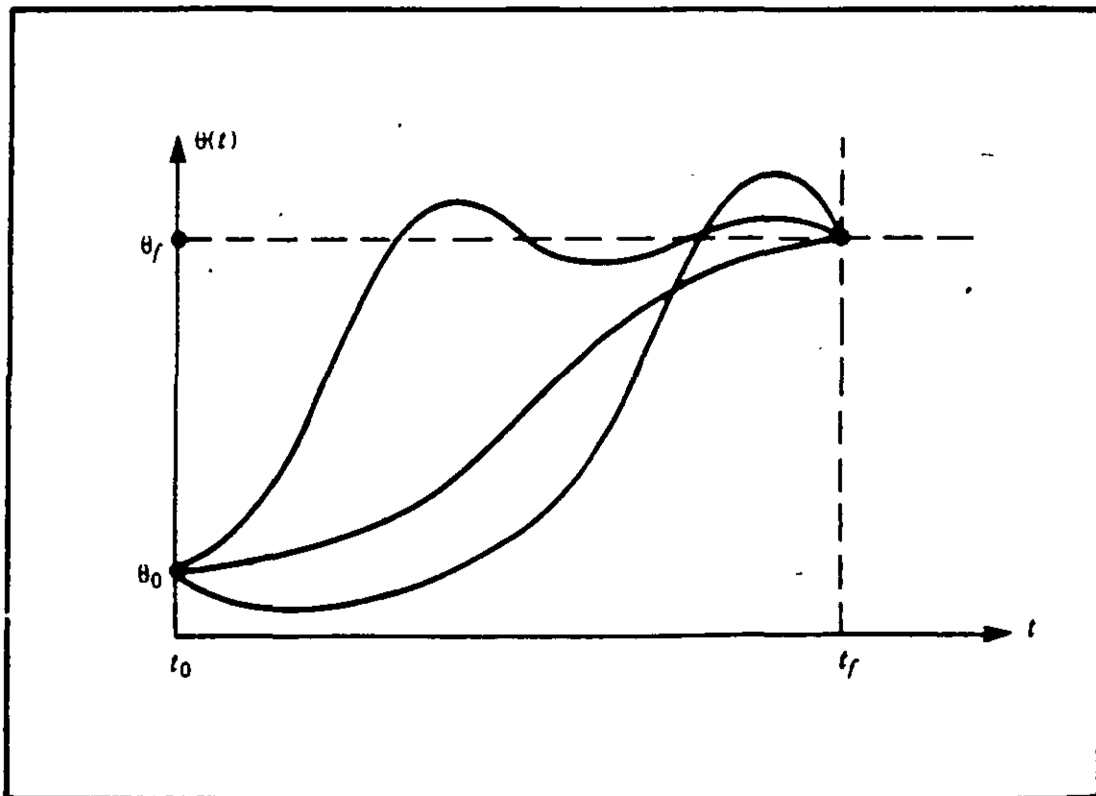
A PLANNED TRAJECTORY IS A SPECIFICATION OF PROPOSED VALUES OF POSITION, SPEED, AND ACCELERATION THAT SHOULD OCCUR (IN THE FUTURE).

****

A PLANNED TRAJECTORY MAY OR MAY NOT BE PHYSICALLY POSSIBLE FOR THE ROBOT TO EXECUTE!

THIS IS ILLUSTRATED WITH A SIMPLE ONE-DIMENSIONAL EXAMPLE:

GIVEN BELOW ARE SEVERAL PROPOSED PATHS FOR A VARIABLE (e. g. AN ANGLE, THETA, THAT MIGHT BE THE ANGLE OF ONE OF THE DRIVE MOTORS OR MIGHT BE THE ANGLE SPECIFYING THE POSE OF THE ROBOT.



Several possible path shapes

CONSIDER THE FOLLOWING PROBLEM:

…VE FROM THE INITIAL POSITION AT t = 0 AND AT REST TO THE FINAL POSITION AT $t = t_f$ AND AT REST SUCH THAT THE PATH IS "SMOOTH".

INITIAL AND FINAL VALUES

$$\theta(0) = \theta_0,$$

$$\theta(t_f) = \theta_f.$$

INITIAL AND FINAL SPEEDS

$$\dot{\theta}(0) = 0,$$

$$\dot{\theta}(t_f) = 0.$$

ASSUMED CUBIC RELATIONSHIP

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3,$$

DERIVATIVES

$$\dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2,$$

$$\ddot{\theta}(t) = 2a_2 + 6a_3 t.$$

INITIAL POSITION

$$\theta_0 = a_0,$$

FINAL POSITION

$$\theta_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3,$$

INITIAL SPEED

$$0 = a_1,$$

FINAL SPEED

$$0 = a_1 + 2a_2 t_f + 3a_3 t_f^2.$$

NOW HAVE 4 EQUATIONS WITH 4 UNKNOWNS:

SOLUTION FOR a_i:

$$
\begin{aligned}
a_0 &= \theta_0, \\
a_1 &= 0, \\
a_2 &= \frac{3}{t_f^2}(\theta_f - \theta_0), \\
a_3 &= -\frac{2}{t_f^3}(\theta_f - \theta_0).
\end{aligned}
$$

****

EXAMPLE: $\theta(0) = 15$
$\theta(3) = 75$

FIND COEFFICIENTS AND PLOT:

COEFFICIENTS:

$$
\begin{aligned}
a_0 &= 15.0, \\
a_1 &= 0.0, \\
a_2 &= 20.0, \\
a_3 &= -4.44.
\end{aligned}
$$

POSITION

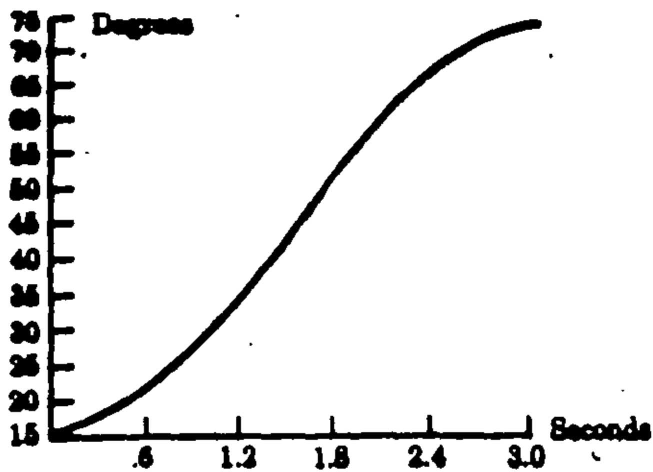
$$\theta(t) = 15.0 + 20.0t^2 - 4.44t^3,$$

SPEED

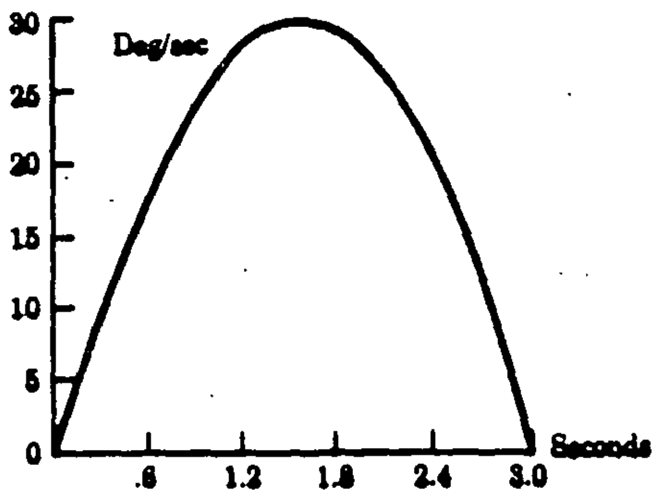
$$\dot{\theta}(t) = 40.0t - 13.33t^2,$$

ACCELERATION

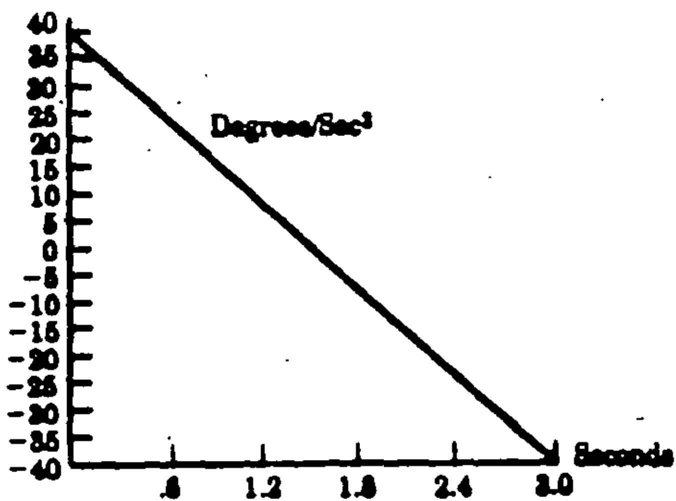
$$\ddot{\theta}(t) = 40.0 - 26.66t.$$



Degrees
75
70
65
60
55
50
45
40
35
30
25
20
15
.6
1.2
1.8
2.4
3.0
Seconds
Position



30
Deg/sec
25
20
15
10
5
0
.6
1.2
1.8
2.4
3.0
Seconds
Velocity



40
35
30
25
20
15
10
5
0
-5
-10
-15
-20
-25
-30
-35
-40
Degrees/Sec²
.6
1.2
1.8
2.4
3.0
Seconds
Acceleration

NOTE THAT THE POSITION IS GIVEN BY A CUBIC CURVE, THE VELOCITY IS PARABOLIC, AND THE ACCELERATION IS LINEAR (THE MAXIMUM ACCELERATION AND THE MAXIMUM DECELLERATION OCCUR AT THE END POINTS).

****

WHAT IF THE MAXIMUM ALLOWED ACCELERATION IS (SAY) 30 DEGREES/SECOND^2 (+ OR -)?

START WITH THE MAXIMUM ACCELERATION OF $\ddot{\theta} = 30 = \text{CONST.}$

THE VELOCITY IS $\dot{\theta} = 30\,t$ (since $\dot{\theta}(0) = 0$)

AND THE POSITION IS GIVEN BY

$$\theta(t) = \frac{30t^2}{2} + \theta(0) = 15t^2 + 15$$

NOTE THAT THE ABOVE SOLUTION IS ONLY VALID UNTIL THE TIME WHEN THE ANGLE "CATCHES UP" TO THE DESIRED TRAJECTORY. THE TIME FOR "CATCHING UP" IS

$$\theta_{30} = 15t^2 + 15 = \theta_{40\downarrow} = 20t^2 - 4.44t^3 + 15$$

$$-5t^2 = -4.44t^3 \Rightarrow t = \frac{5}{4.44} = 1.126 \text{ sec.}$$

ONE FORM OF FULL TRAJECTORY COULD BE GENERATED BY USING "BANG-BANG" CONTROL ON THE ACCELERATION. NOTE THAT WHEN THE ACCELERATION IS LIMITED, THE MAXIMUM SPEED DURING THE TRAJECTORY IS HIGHER TO REACH THE SAME END POINT AT ZERO SPEED! THIS IS GENERALLY TRUE, IF YOU HAVE LESS ACCELERATION AVAILABLE, YOU MUST GO FASTER TO GET TO THE SAME POINT AT THE SAME TIME.

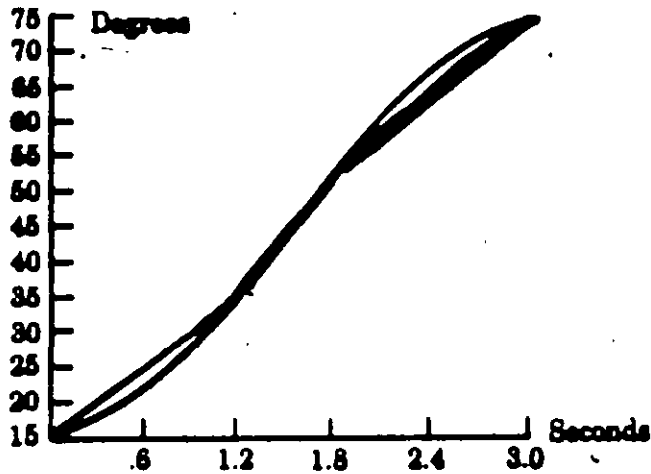
$$\theta_{30} = 15t^2 + 15 = 34.022 = \theta_{40\downarrow}$$

$$\left.\begin{aligned} \dot{\theta}_{30} &= 33.78 \\ \dot{\theta}_{40\downarrow} &= 40t - 13.33t^2 = 28.14 \end{aligned}\right\} \leftarrow \text{HIGHER SPEED}$$

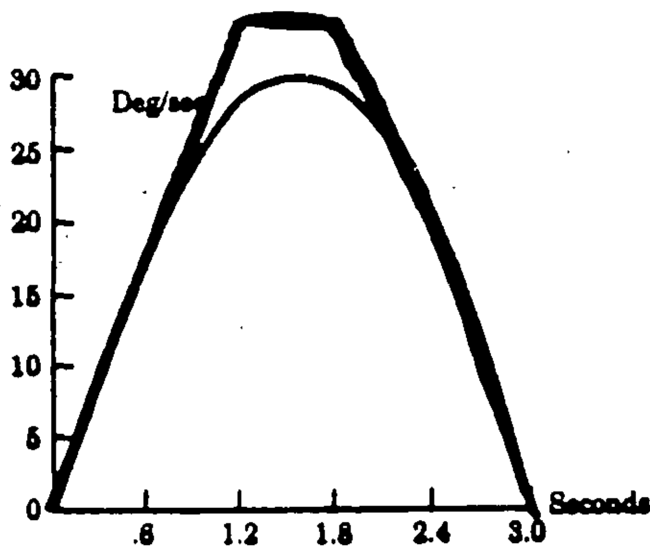
FOR EXAMPLE:

$$\ddot{\theta}_{30+} \rightarrow 0$$

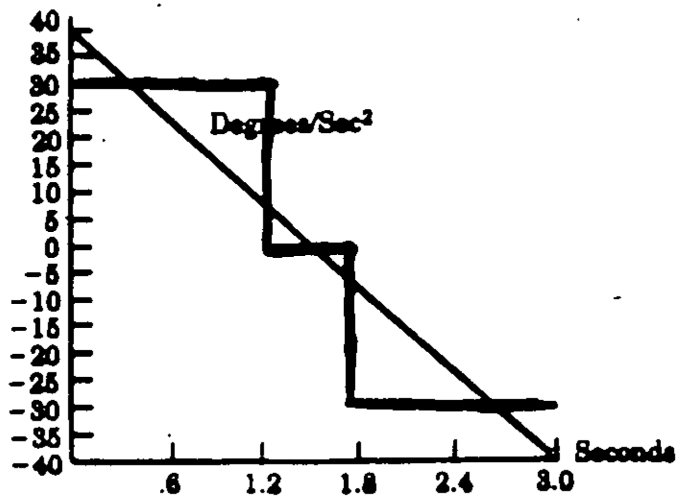
SEE TRAJECTORY



Position



Velocity



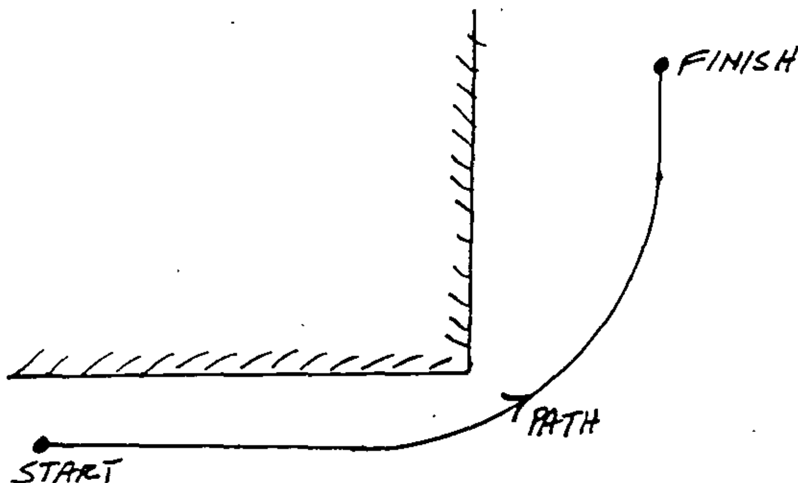
Acceleration

ACCELERATION LIMITED TO 30 DEG/sec²

THE LIMITATION ON ACCELERATION THAT WAS APPLIED IN THE ABOVE EXAMPLE IS KNOWN AS A **CONSTRAINT**. CONSTRAINTS ARE VERY COMMON IN TRAJECTORY GENERATION. THE CONSTRAINTS CAN TAKE MANY FORMS. FOR EXAMPLE, SINCE ACCELERATION IS PROPORTIONAL TO FORCE (OR TORQUE) A LIMIT TO THE AVAILABLE ACCELERATION IS QUITE NATURAL SINCE THE MOTORS MAY BE ONLY ABLE TO DELIVER A LIMITED TORQUE.

CONSTRAINTS COULD ALSO BE APPLIED TO SPEED OR TO POSITION. IN THE CASE OF SPEED THERE COULD BE ANY NUMBER OF REASONS (OFTEN ASSOCIATED WITH SAFETY) TO CONSTRAIN THE SPEED TO VALUES BELOW SOME SPECIFICATION. POSITION COULD BE CONSTRAINED BY REQUIRING THAT THE ANGLE (OR A ROBOT FOR THAT MATTER) NOT PASS A CERTAIN POSITION BEFORE SOME TIME. IN TWO DIMENSIONS, THE POSITION IS VERY OFTEN CONSTRAINED BY THE PRESENCE OF AN OBSTACLE.

THUS, IF A TWO DIMENSIONAL PATH WERE TO BE CONSIDERED LIKE THE ONE SKETCHED BELOW FOR A POINT ROBOT MANEUVERING AROUND A CORNER, THE PROBLEM COULD BE CONSTRAINED BY POSITION (NAMELY THE ROBOT IS NOT ALLOWED TO TOUCH THE WALLS), BY ACCELERATION (THE ROBOT HAS A LIMITED AMOUNT OF TORQUE AVAILABLE) AND BY VELOCITY (EITHER LINEAR OR ANGULAR) BECAUSE THE ROBOT HAS SAFETY SPEED LIMITS OR IT MAY SLIP OR SKID AROUND CORNERS.



THE PATH SKETCHED ABOVE COULD BE TRANSFORMED INTO A TRAJECTORY BY PLACING APPROPRIATE TIME MARKS ON IT. IF THE TIME MARKS ARE ARBITRARY, THEN THE TRAJECTORY MAY NOT BE REALIZABLE BY THE ROBOT. THE "BEST" REALIZABLE TRAJECTORY NEEDS SOME DEFINITION OF THE MEANING OF "BEST". THIS COULD, FOR EXAMPLE BE THE MINIMUM TIME, THE MINIMUM ENERGY, THE MINIMUM DEVIATION FROM THE PLANNED PATH, ETC. THESE CONCEPTS LEAD TO THE FIELD OF **OPTIMUM** TRAJECTORY PLANNING.

AN OPTIMUM TRAJECTORY IS ONE THAT IS "BEST" ACCORDING TO THE FUNCTION TO BE OPTIMIZED AND THAT MEETS ALL OF THE SPECIFIED CONSTRAINTS.

WHAT IF THE MAXIMUM ALLOWED ACCELERATION OF THE SIMPLE ONE-DIMENSIONAL EXAMPLE PROBLEM IS (SAY) 25 DEGREES/SEC^2

AGAIN, START WITH THE MAXIMUM ACCELERATION.

THEN AT THE START: $\ddot{\theta} = 25$ (CONST.)

$$\dot{\theta} = 25t \quad (\text{SINCE } \dot{\theta}(0) = 0)$$

$$\theta = \frac{25t^2}{2} + \theta(0) = 12.5t^2 + 15$$

WE SEE BELOW THAT THE POSITION NEVER CATCHES UP TO THE UNCONSTRAINED CASE. (CUBIC)

$$\theta_{25} = 12.5\,t^2 + 15 = \theta_{40\downarrow} = 20t^2 - 4.44t^3 + 15$$

$$-7.5t^2 = -4.44t^3$$

$$t = \frac{7.5}{4.44} = 1.69 \text{ SECONDS}$$

$$> 1.5 \text{ SECONDS}$$

THE DISTANCE TRAVELLED IN 1.5 SECONDS IS 28.125 DEGREES.

$$\theta_{25} = (12.5)(2.25) + 15 = 43.125$$
$$\Delta\theta_{25} = 28.125$$

THEREFORE, IT IS NOT POSSIBLE TO GET TO THE GOAL (60 DEGREES DISTANCE) IN 3 SECONDS IF STOPPED AT THE END.

THE TIME REQUIRED TO GO 30 DEGREES IS 1.549 SECONDS. THEREFORE, THE SHORTEST TIME FROM START TO GOAL (BOTH STOPPED) IS 1.549 x 2 = 3.098 SECONDS.

$$45 = 12.5t^2 + 15$$
$$12.5t^2 = 30$$
$$t^2 = \pm 1.549 \text{ sec}$$

NOTE THAT THE MAXIMUM SPEED IS 38.725 DEGREES/SECOND.

THE TIME IS LONG TO GET THERE BUT THE MAXIMUM VELOCITY IS HIGH.

THE PREVIOUS EXAMPLE IS A CASE IN WHICH THE ACCELERATION IS INSUFFICIENT TO ALLOW THE TRAJECTORY TO BE COMPLETED WITHIN THE SPECIFIED TIME. IF THIS TRAJECTORY IS ONE LEG OF A TOTAL PATH, THEN THIS RESULT MAY EITHER MAKE THE ARRIVAL TIME AT THE FINAL DESTINATION LATER THAN DESIRED (IF THE OTHER LEGS, FOR EXAMPLE, ARE OPTIMIZED AND COULD ACTUALLY MEET THEIR EXPECTATIONS), OR, ONE MIGHT RE-EXAMINE OTHER LEGS TO SEE IF THEY COULD BE SHORTENED TO "MAKE UP" FOR THE DELAY IN THIS LEG. THUS, WE SEE THAT EACH LEG OF A PATH IS NOT PLANNED INDEPENDENTLY OF THE OTHER LEGS.

THE WORD "PLANNED" IS USED ABOVE. HOW DOES PATH PLANNING OCCUR?

CONSIDER THE FOLLOWING SEQUENCE:

1) RECEIVE INFORMATION CONCERNING START AND GOAL POSITION AND START AND GOAL TIMES.

2) KNOW CONSTRAINTS ON MACHINE (ACCELERATIONS AND VELOCITIES).

3) DETERMINE CONSTRAINTS ON POSITION (CARTOGRAPHER - MAPS).

4) PLAN A PATH TAKING INTO ACCOUNT THE POSITION CONSTRAINTS.

5) TAKING END-POINT TIMES INTO ACCOUNT, DIVIDE TIMES INTO THE LEGS OF THE PATH (FOR EXAMPLE BY TIME BEING PROPORTIONAL TO LEG LENGTH)

6) IF ONE WERE TO STOP AT THE END OF EACH LEG, THEN SOMETHING LIKE THE PREVIOUS EXAMPLES COULD BE USED FOR EACH LEG (IN TWO DIMENSIONS) TO CALCULATE THE TRAJECTORIES FOR EACH LEG.

7) IF SOME LEG(s) EXCEED THE ALLOWED TIME, SEE IF TIME CAN BE MADE UP IN OTHER LEG(S).

8) IF ONE DOES NOT STOP AT THE END OF LEG(s), THEN THESE END POINTS MIGHT BE TREATED AS WAY-POINTS AND AN ITERATIVE PROCEDURE ADOPTED TO BEST MEET THE REQUIREMENTS.

9) THE END RESULT SHOULD BE A PLANNED TRAJECTORY THAT MEETS THE CONSTRAINTS OF THE ROBOT (ACCELERATIONS AND VELOCITIES), THE CONSTRAINTS OF THE ENVIRONMENT (POSITION), THE SPECIFICATIONS OF THE ARRIVAL TIMES, AND MEETS ANY OPTIMAL PRINCIPLES SUCH AS MINIMUM ENERGY, MAXIMUM CLEARANCE, MINIMUM TIME, ETC.

THIS IS A TOUGH PROBLEM

USUALLY, ONE WISHES TO PASS THROUGH A VIA POINT WITHOUT STOPPING, SO WE NEED TO GENERALIZE THE WAY OF GENERATING TRAJECTORIES TO SATISFY CONSTRAINTS. LET'S CONSIDER ONLY THE VIA POINTS AS CONSTRAINTS AND RELIEVE THE CONSTRAINTS FROM THE ACCELERATIONS, ETC.

CONSIDER A ONE DIMENSIONAL PROBLEM, AGAIN, AND CONSIDER FITS TO CUBIC POLYNOMIALS, AGAIN. THE CUBIC EQUATION AND THE CONSTRAINTS ARE GIVEN BELOW. NOTE THAT THE VELOCITIES AT THE END POINTS NOW NOT ZERO.

INITIAL SPEED:

$$\dot{\theta}(0) = \dot{\theta}_0,$$

FINAL SPEED:

$$\dot{\theta}(t_f) = \dot{\theta}_f.$$

CUBIC:

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3,$$

EQUATIONS FOR CONSTRAINTS:

INITIAL POSITION:

$$\theta_0 = a_0,$$

FINAL POSITION:

$$\theta_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3,$$

INITIAL SPEED:

$$\dot{\theta}_0 = a_1,$$

FINAL SPEED:

$$\dot{\theta}_f = a_1 + 2a_2 t_f + 3a_3 t_f^2.$$

SOLUTIONS FOR a_i:

$$a_0 = \theta_0,$$

$$a_1 = \dot{\theta}_0,$$

$$a_2 = \frac{3}{t_f^2}(\theta_f - \theta_0) - \frac{2}{t_f}\dot{\theta}_0 - \frac{1}{t_f}\dot{\theta}_f,$$

$$a_3 = -\frac{2}{t_f^3}(\theta_f - \theta_0) + \frac{1}{t_f^2}(\dot{\theta}_f + \dot{\theta}_0).$$

IF THE DESIRED SPEEDS ARE KNOWN AT EACH VIA POINT, THEN IT IS A SIMPLE MATTER TO APPLY THE EQUATIONS FOR THE a_i TO EACH SEGMENT TO FIND THE SPECIFIC CUBIC EQUATIONS FOR EACH SEGMENT.

HOWEVER, THE DESIRED SPEEDS AT THE VIA POINTS ARE NOT USUALLY KNOWN. THUS, ONE COULD CONSIDER SEVERAL OPTIONS:

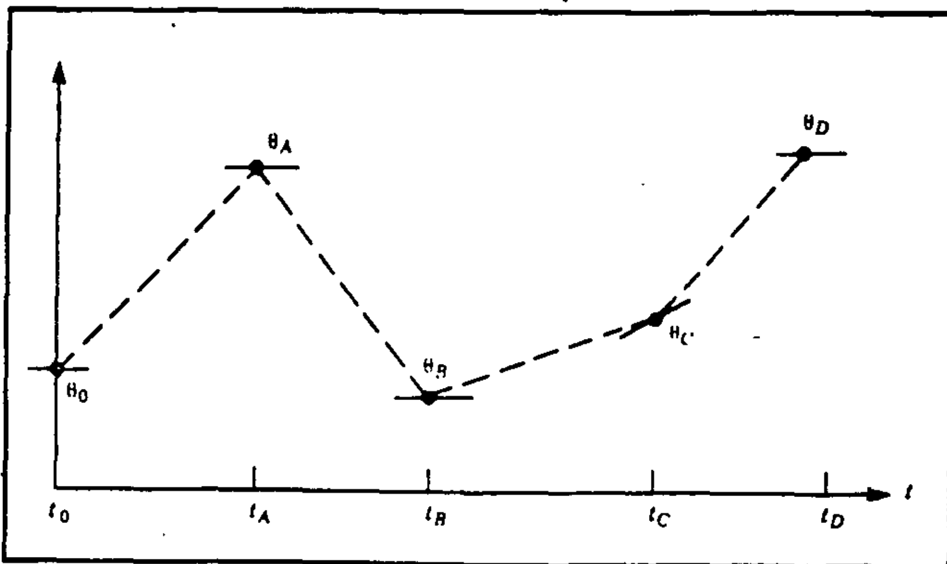
1) THE USER SPECIFIES THE DESIRED SPEED AT EACH VIA POINT. THIS OPTION IS MUCH TOO USER-DEPENDENT FOR OUR AUTONOMOUS MOBILE ROBOTS.

2) THE "SYSTEM" [NAMELY, THE COMPUTER SYSTEM CONTROLLING THE ASPECT OF PLANNING OF TRAJECTORIES] AUTOMATICALLY CHOOSES THE SPEEDS AT THE VIA POINTS BY APPLYING A SUITABLE HEURISTIC.

3) THE "SYSTEM" [NAMELY, THE COMPUTER SYSTEM CONTROLLING THE ASPECT OF PLANNING OF TRAJECTORIES] AUTOMATICALLY CHOOSES THE SPEEDS AT THE VIA POINTS BY APPLYING SOME PRINCIPLES SUCH AS CONTINUOUS ACCELERATION AT VIA POINTS, ACCELERATIONS AND SPEEDS MEET CONSTRAINTS, ETC. AND GENERATE TRAJECTORIES BASED ON THESE PRINCIPLES.

CONSIDER THE PATH SPECIFIED BY VIA POINTS (IN ONE DIMENSION) BELOW. THE SPEEDS AT THE VIA POINTS ARE SHOWN WITH SMALL LINE SEGMENTS REPRESENTING TANGENTS TO THE CURVE AT EACH VIA POINT. THIS CHOICE IS THE RESULT OF APPLYING A CONCEPTUALLY AND COMPUTATIONALLY SIMPLE HEURISTIC.

IMAGINE THE VIA POINTS CONNECTED WITH STRAIGHT LINE SEGMENTS - IF THE SLOPE OF THESE LINES CHANGES SIGN AT THE VIA POINT, CHOOSE ZERO VELOCITY, IF THE SLOPE OF THESE LINES DOES NOT CHANGE SIGN, CHOOSE THE AVERAGE OF THE TWO SLOPES AS THE VIA VELOCITY.



Via points with desired velocities at the points indicated by tangents.

TO ILLUSTRATE ONE METHOD TO PERFORM THE PROBLEM AS SPECIFIED IN 3), CONSIDER THE CASE OF REQUIRING THAT ACCELERATION IS CONTINUOUS AT THE VIA POINT. TO DO THIS, WE REPLACE THE TWO VELOCITY CONSTRAITS AT THE CONNECTION OF THE TWO CUBICS WITH THE TWO CONSTRAINTS THAT a) VELOCITY IS CONTINUOUS AND b) ACCELERATION IS CONTINUOUS. CONSIDER THE EXAMPLE BELOW:

THE PROBLEM IS TO SOLVE FOR THE COEFFICIENTS OF TWO CUBICS WHICH ARE CONNECTED IN A TWO-SEGMENT SPLINE WITH CONTINUOUS ACCELERATION AT THE INTERMEDIATE VIA POINT. THE INITIAL ANGLE IS θ_0, THE VIA POINT IS θ_v, AND THE GOAL POINT IS θ_g

FIRST CUBIC:

$$\theta(t) = a_{10} + a_{11}t + a_{12}t^2 + a_{13}t^3,$$

SECOND CUBIC:

$$\theta(t) = a_{20} + a_{21}t + a_{22}t^2 + a_{23}t^3.$$

EACH CUBIC IS EVALUATED OVER AN INTERVAL STARTING AT t = 0 AND ENDING AT $t = t_{fi}$, WHERE i = 1 or i = 2.

THE CONSTRAINTS ARE:

$$\theta_0 = a_{10},$$

$$\theta_v = a_{10} + a_{11}t_{f1} + a_{12}t_{f1}^2 + a_{13}t_{f1}^3,$$

$$\theta_v = a_{20},$$

$$\theta_g = a_{20} + a_{21}t_{f2} + a_{22}t_{f2}^2 + a_{23}t_{f2}^3,$$

$$0 = a_{11},$$

$$0 = a_{21} + 2a_{22}t_{f2} + 3a_{23}t_{f2}^2,$$

$$a_{11} + 2a_{12}t_{f1} + 3a_{13}t_{f1}^2 = a_{21},$$

$$2a_{12} + 6a_{13}t_{f1} = 2a_{22}.$$

THE PROBLEM IS TO SOLVE EIGHT EQUATIONS WITH EIGHT UNKNOWNS. FOR THE CASE THE $t_f = t_{f1} = t_{f2}$ THE FOLLOWING VALUES FOR THE COEFFICIENTS ARE OBTAINED:

$$a_{10} = \theta_0,$$

$$a_{11} = 0,$$

$$a_{12} = \frac{12\theta_v - 3\theta_g - 9\theta_0}{4t_f^2},$$

$$a_{13} = \frac{-8\theta_v + 3\theta_g + 5\theta_0}{4t_f^3},$$

$$a_{20} = \theta_v,$$

$$a_{21} = \frac{3\theta_g - 3\theta_0}{4t_f},$$

$$a_{22} = \frac{-12\theta_v + 6\theta_g + 6\theta_0}{4t_f^2},$$

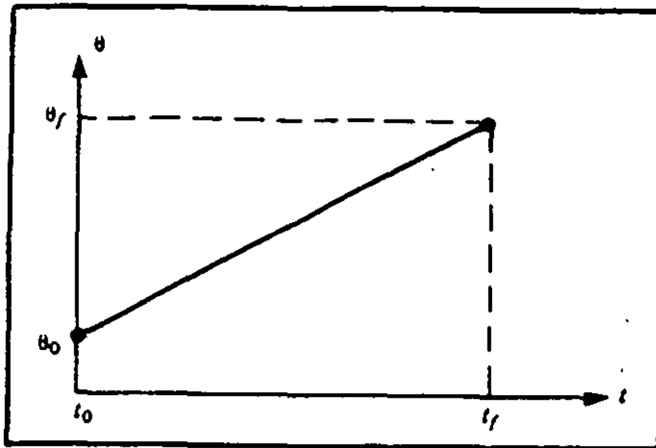
$$a_{23} = \frac{8\theta_v - 5\theta_g - 3\theta_0}{4t_f^3}.$$

FOR THE GENERAL CASE OF n CUBIC SEGMENTS THE EQUATIONS WHICH ARISE FROM INSISTING ON CONTINUOUS ACCELERATION AT THE VIA POINTS MAY BE CAST IN MATRIX FORM WHICH IS SOLVED TO COMPUTE THE SPEEDS AT THE VIA POINTS. THE MATRIX TURNS OUT TO BE TRIDIAGONAL AND EASILY SOLVED (SEE D. ROGERS AND J. A. ADAMS, MATHEMATICAL ELEMENTS FOR COMPUTER GRAPHICS, McGraw-Hill, 1976.)

OBVIOUSLY, HIGHER ORDER POLYNOMIALS COULD BE USED.

LINEAR FUNCTION WITH PARABOLIC BLENDS

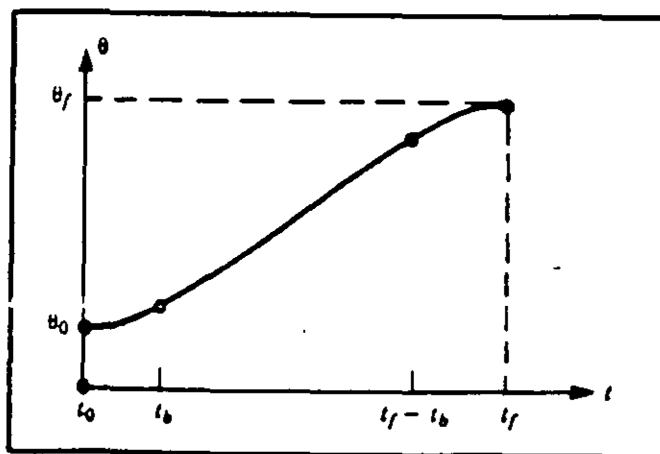
IN THIS CASE, ONE COULD CHOOSE THE PATH SHAPE TO BE LINEAR (SEE FIGURE BELOW). THE PROBLEM WITH THIS CHOICE IS THAT THE SPEEDS ARE DISCONTINUOUS AT THE END POINTS. TO CREATE A SMOOTH PATH WITH CONTINUOUS POSITION AND SPEED, IT IS POSSIBLE TO ADD A PARABOLIC "BLEND" REGION NEAR EACH PATH END POINT.



Linear interpolation requiring infinite acceleration.

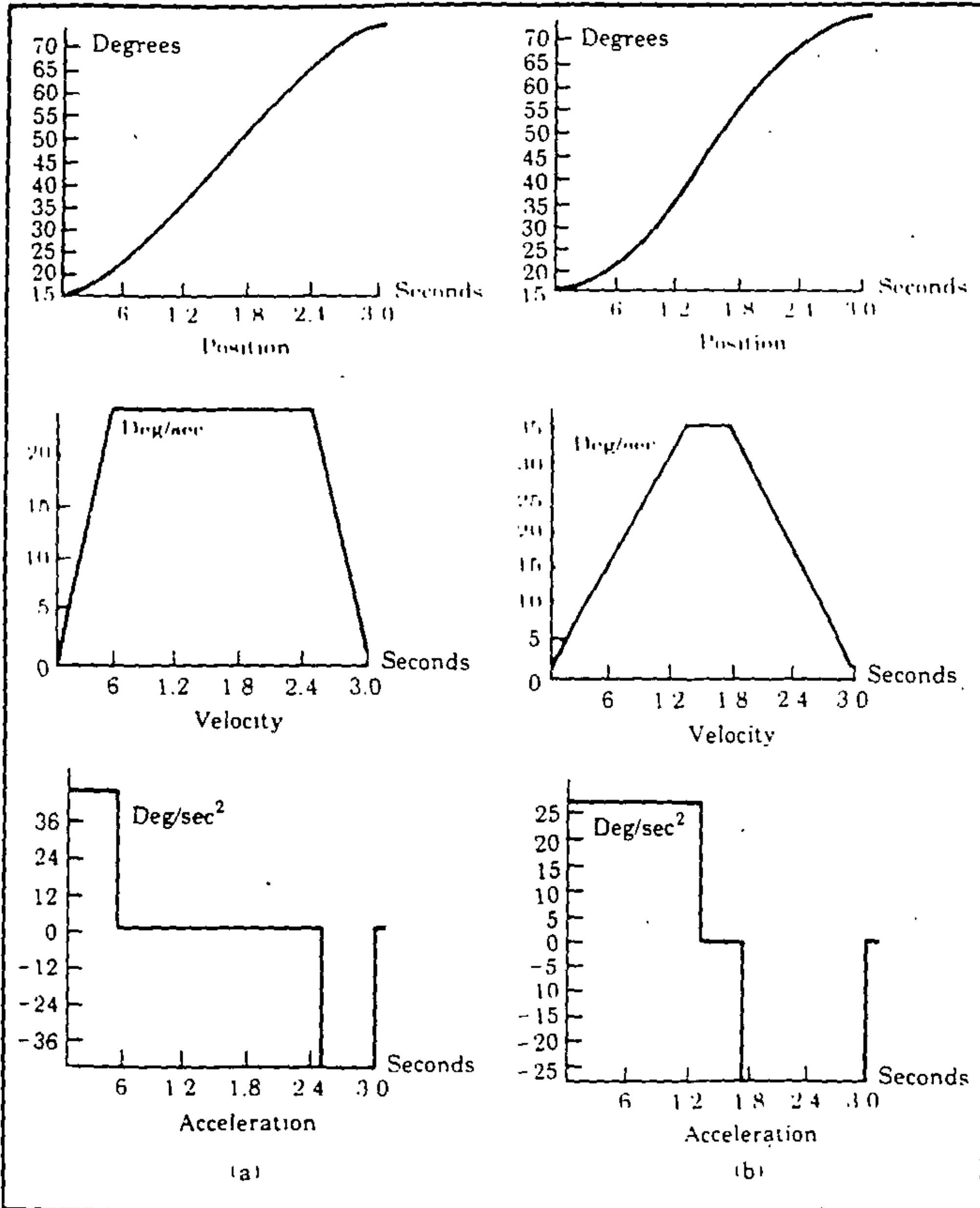
DURING THE BLEND PORTION OF THE TRAJECTORY, CONSTANT ACCELERATION IS USED TO CHANGE SPEED SMOOTHLY. THE FIGURE BELOW SHOWS A SIMPLE TRAJECTORY CONSTRUCTED IN THIS WAY. THE LINEAR FUNCTION AND THE TWO PARABOLIC FUNCTIONS ARE SPLINED TOGETHER SO THAT THE ENTIRE TRAJECTORY IS CONTINUOUS IN POSITION AND SPEED.

FOR AN EXAMPLE, SEE CRAIG, INTRODUCTION TO ROBOTICS, Addison Wesley, 1989.



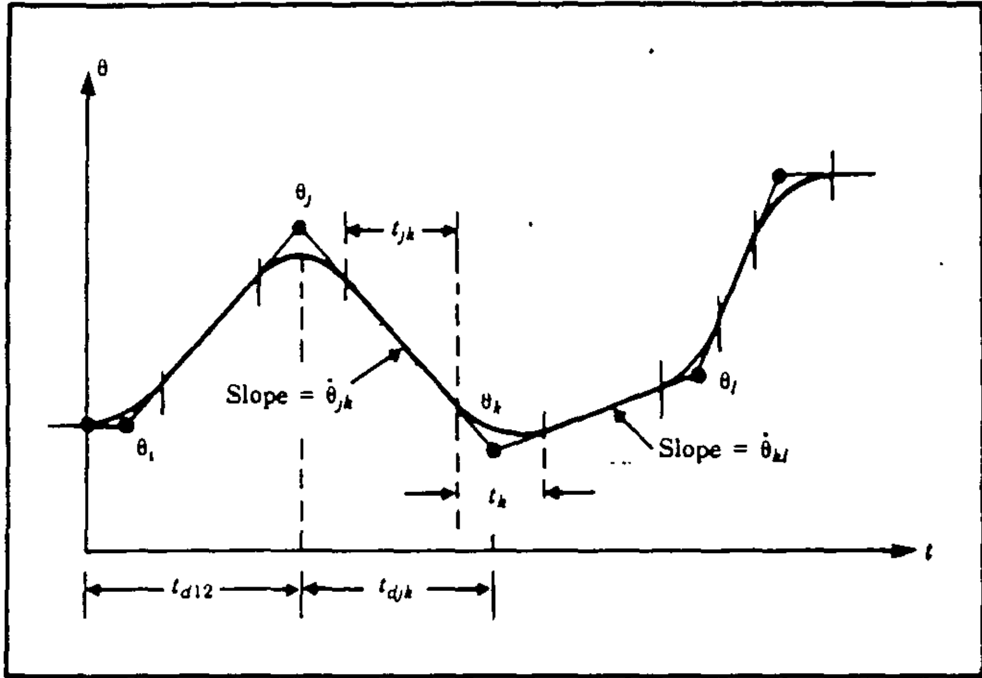
Linear segment with parabolic blends

THE RESULT OF APPLYING LINEAR INTERPOLATION WITH PARABOLIC BLENDS TO AN EXAMPLE PROBLEM:



Position, velocity, and acceleration profiles for linear interpolation with parabolic blends. The set of curves on the left are based on a higher acceleration during the blends than those on the right.

LINEAR FUNCTIONS WITH PARABOLIC BLENDS CAN BE APPLIED TO PATHS WITH VIA POINTS. SEE THE FIGURE BELOW: (AGAIN SEE CRAIG FOR EXAMPLES)



Multisegment linear path with blends.

PILOTING ALONG A SEQUENCE

OF STRAIGHT LINES:

A PATH SPECIFICATION THAT CONSISTS OF A SEQUENCE OF STRAIGHT LINES IS ONE OF THE MOST SIMPLE WAYS TO DEFINE A PATH IN TWO DIMENSIONS.

FOLLOWING THE STRAIGHT LINES REQUIRES A TRANSITION FROM ONE LINE TO ANOTHER. THIS IS ACCOMPLISHED BY A COORDINATE TRANSFORMATION.

VEHICLE (REFERRED TO AS AMR) IS ASSUMED TO BE A POINT AND OBJECTS ARE ENLARGED TO CORRECT FOR THE FINITE SIZE OF THE VEHICLE.

ESCRIBE POSTURE BY (x, y,θ)

(x, y,θ) = posture in XY frame

(x', y',θ') = posture in X'Y' frame

(x_1, y_1) = coordinates of new origin O' in old (XY) coordinate system

θ_1 = angle between new and old X - axes.

[x_1, y_1,θ_1] = transformation from new to old coordinate system.

THE TRANSFORMATION EQUATIONS ARE:

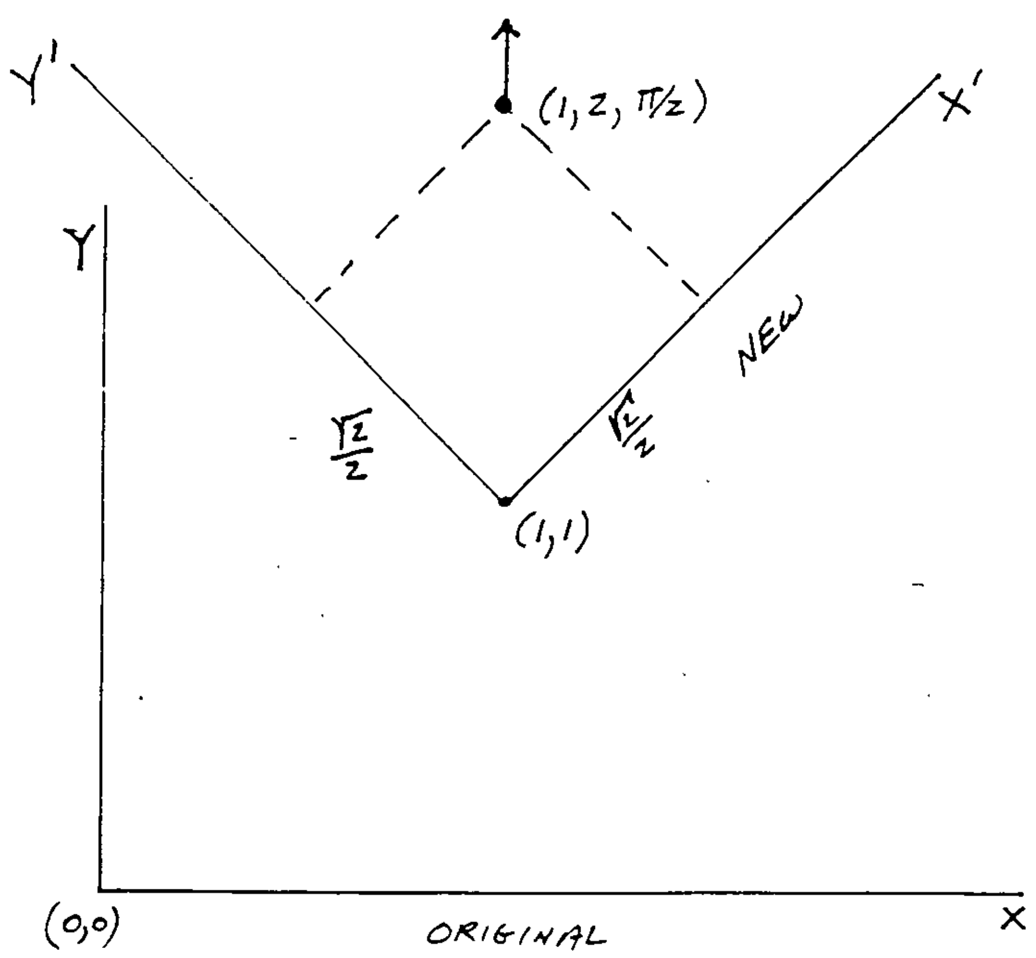
$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix}.$$

$$\begin{pmatrix} \dot{x}' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} \cos\theta_1 & \sin\theta_1 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix} \right).$$

EXAMPLE: THE ORIGINAL AND NEW COORDINATES ARE SHOWN BELOW.

$(X, Y, \theta) = (1, 2, \pi/2)$

$(X_1, Y_1, \theta_1) = (1, 1, \pi/4)$



THE TRANSFORMATION IS APPLIED AS FOLLOWS:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} \cos\theta_1 & \sin\theta_1 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix} \right)$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 \\ 2 \\ \frac{\pi}{2} \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ \frac{\pi}{4} \end{pmatrix} \right)$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ \pi/4 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}}(0) + \frac{1}{\sqrt{2}}(1) + 0\cdot\frac{\pi}{4} \\ -\frac{1}{\sqrt{2}}(0) + \frac{1}{\sqrt{2}}(1) + 0\cdot\frac{\pi}{4} \\ 0\cdot 0 + 0\cdot 1 + \frac{\pi}{4} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ \frac{\pi}{4} \end{pmatrix}$$

SEE THAT THIS IS THE POSTURE IN THE NEW FRAME.

CONSIDER A VEHICLE WITH A FEEDBACK CONTROL SYSTEM SUCH THAT IT TRAVELS ON THE X-AXIS OF THE CURRENT COORDINATE SYSTEM IN THE POSITIVE DIRECTION WITH A GIVEN SPEED V. THE CONTROLLER TRIES TO KEEP THE FOLLOWING CONDITIONS AS NEARLY TRUE AS POSSIBLE:

$$y = 0$$

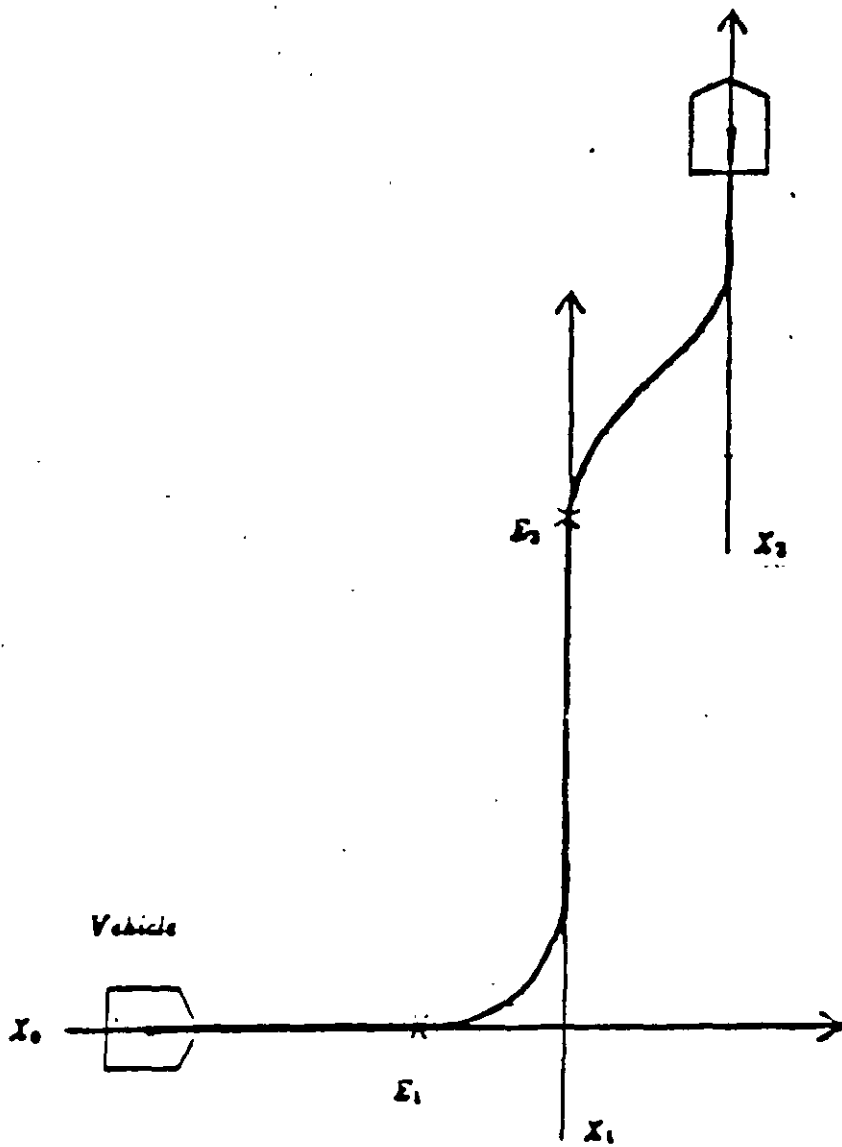
$$dy/dt = 0$$

$$dx/dt = v$$

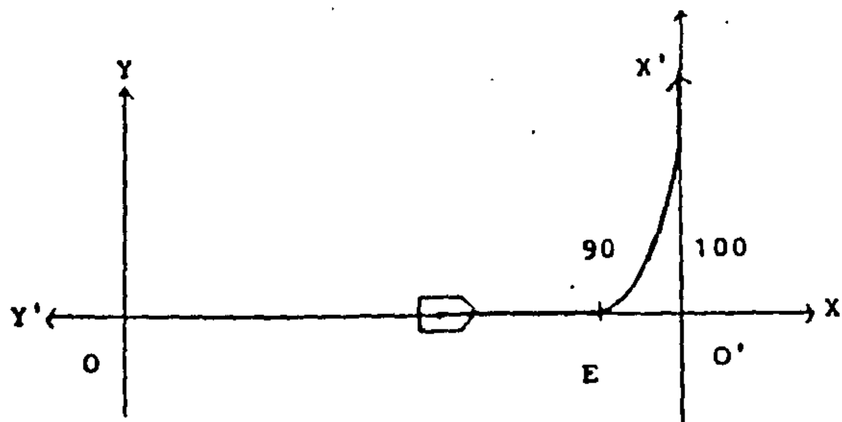
WHEN THE VEHICLE IS TO CHANGE DIRECTION, THEN THE COORDINATE TRANSFORMATION $[x_1, y_1, \quad _1]$ IS ISSUED AT AN EXIT POSITION, E.

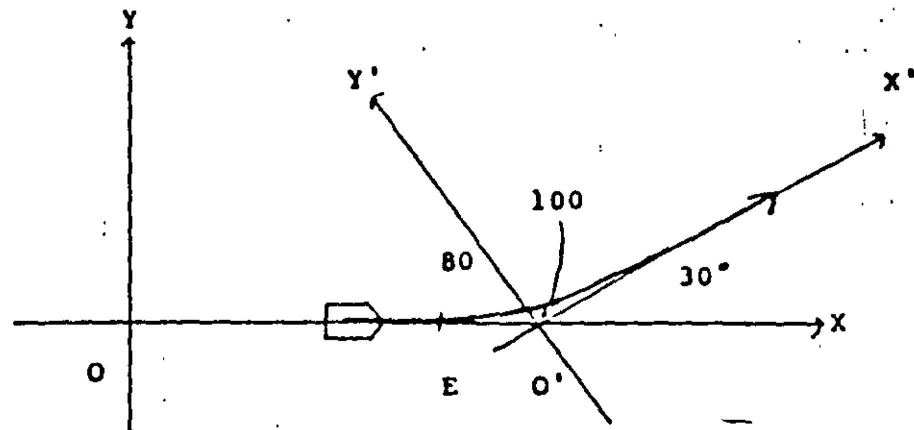
NOW THE FEEDBACK CONTROLLER IS TO USE THE ABOVE CONTROL SCHEME TO CONTROL THE VEHICLE ONTO THE X-AXIS OF THE NEW COORDINATE SYSTEM.

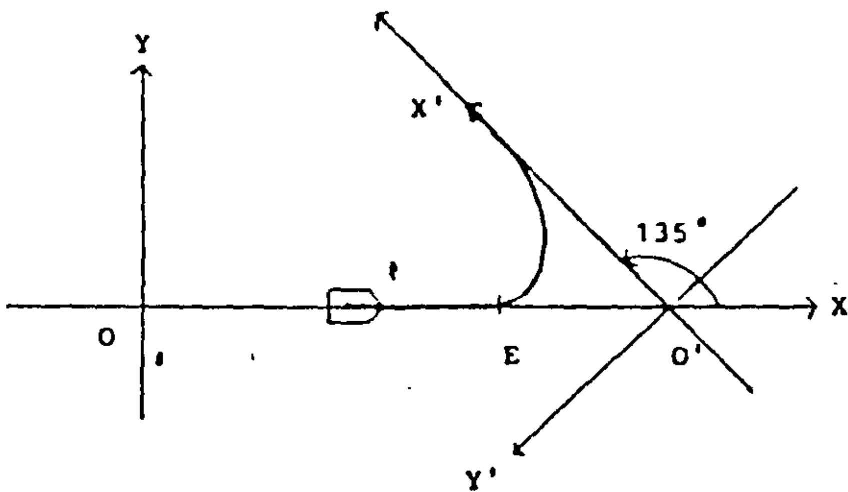
THE FOLLOWING FIGURES ILLUSTRATE SOME OF THE POSSIBILITIES FOR TRANSITIONING FROM ONE PATH TO ANOTHER.

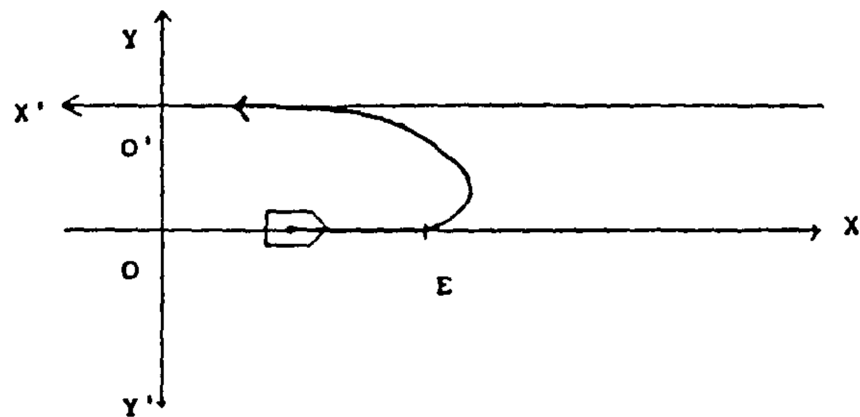


Path specification by directed straight lines.

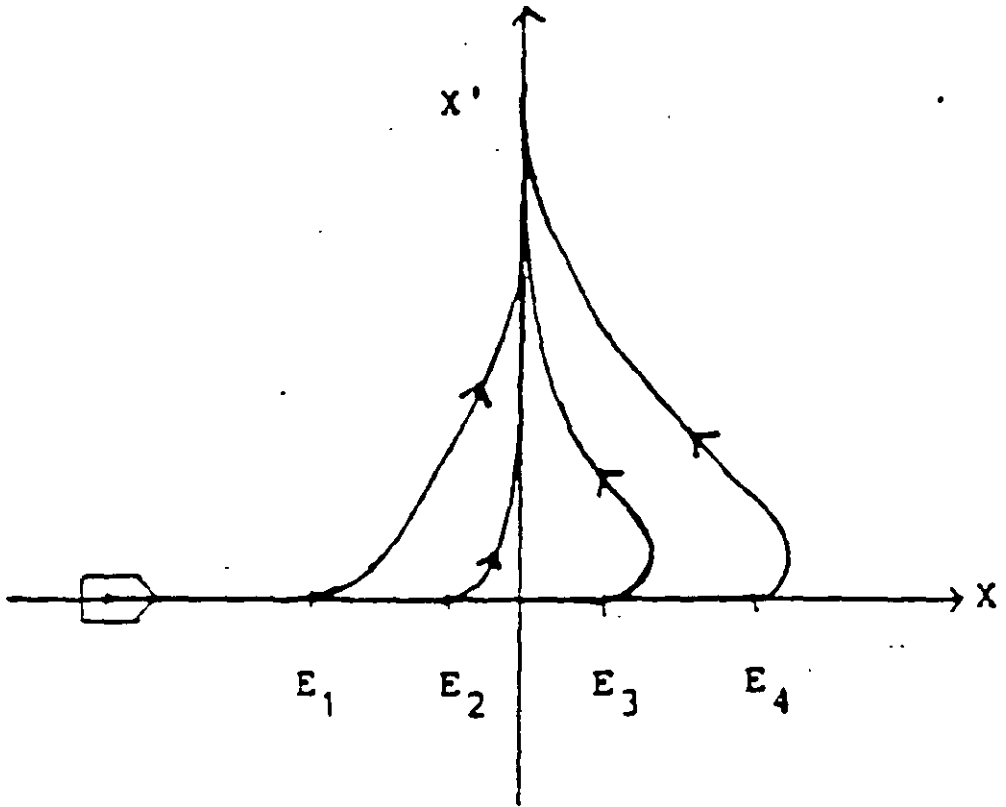




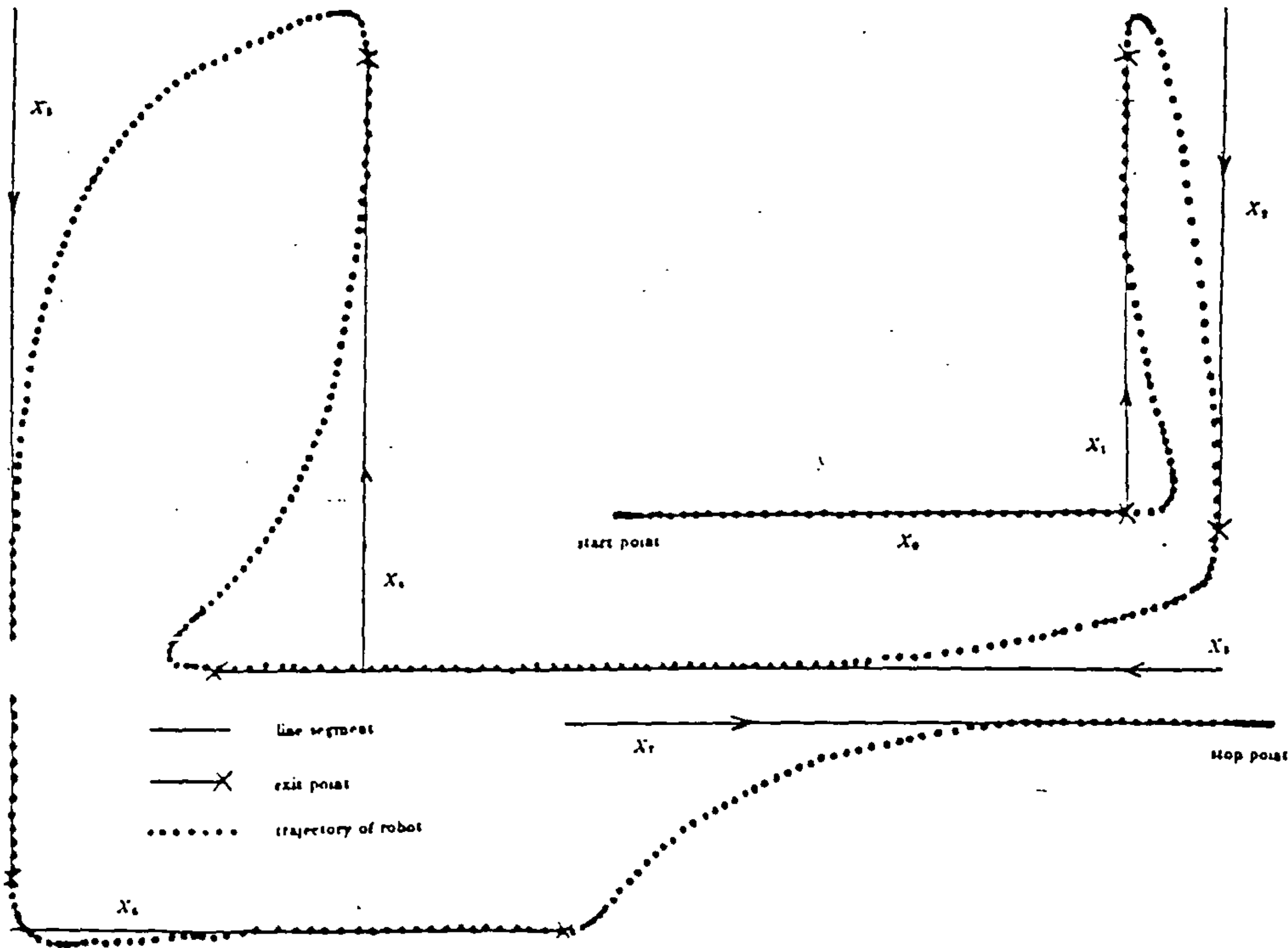




Examples of transformations. (a) [100, 0, 90] (left turn). (b) [100, 0, 30] (left turn by 30°). (c) [100, 0, 135] (left turn by 135°). (e) $[0, 50, \pi]$ (u-turn). (From Ref. 1.)



Effects of different exit positions.



An example of a trajectory generated by the MITCHI simulator.

THE MITCHI ROAD FOLLOWER
THE LUMELSKY BUGS
THE FENG & KROGH PILOTING

CONTENTS

INTRODUCTION (PREPARED BY N. J. McCormick - 1989)

PILOTING BY A SEQUENCE OF STRAIGHT LINES WITHOUT SENSOR FEEDBACK (THE MITCHI ROAD FOLLOWER)

NAVIGATION WITH TACTILE SENSORS (THE LUMELSKY BUGS)

NAVIGATION WITH PROXIMITY SENSORS (FENG & KROGH PILOT)

1. Introduction

1-1. Overview of intelligent control

Operation of an autonomous mobile robot (AMR) requires several hierarchical levels of intelligent control which include *planning, navigation,* and *piloting.* These separate levels of control are needed because not all control aspects can be done in the local AMR environment.

The problem at the top or "planning" level is to define the overall mission for the AMR, including its initial and desired final locations and *a priori* known obstacles and constraints. During the planning operation the global path of the AMR is specified. Also, there may be an overall maximum time specified for the mission.

The role of the "navigation" stage is to find a continuous motion that will take the AMR from its initial position to its desired final position along a locally-defined (i.e., segmented) geometric path within the geometric constraints and that avoids obstacles encountered with the AMR sensors. For the 2-D problem of interest to us, this means that given two locally-defined positions (x_i, y_i) and (x_f, y_f) and the AMR initial orientation θ_i with respect to the x-axis, find a continuous wall-avoiding motion between the two positions or establish that no such motion exists.

In a sense, planning can be considered as "global obstacle avoidance without consideration for the operation of sensors" while navigation enables the AMR to avoid those obstacles.

The navigator also must consider the time required to accomplish the local mission. Since a mobile robot will ordinarily negotiate any given path only once (as opposed to a manipulator, which might perform the same task thousands of times), the navigator wants to plan a path that can be negotiated "quickly" rather than "optimally." In some cases this means that the navigator can use straight lines for the paths, and angular rotations at turns, to plan a path.

A third level in the hierarchy is that of "piloting." This is the problem of following the path prescribed by the AMR navigator. To analyze the motion of the vehicle along a path, it is necessary to link the time variable to the path: the combination of a path and "time marks" along a path gives a "trajectory." At this level of the analysis, the dynamics of the motion of the AMR can become important since the vehicle must be capable of negotiating the path without tipping over. (Of course the AMR must have sufficient power resources to enable it to follow the trajectory, and this leads to additional design considerations.)

One useful analogy may help us better understand the role of the planner, navigator, and pilot of the AMR. This analogy occurs in the transoceanic shipping industry where the planner is in the front office of a company that schedules a vessel for transit between two ports. The front office person typically uses an atlas to plan a great circle path for the ship, subject to known islands and reefs that may be obstacles.

The ship's navigator is the individual responsible for keeping track of the local position of the ship and monitoring its motion with feedback from local sensors (e.g., visual, sonar,

etc.) of obstacles (such as other ships) that have not been foreseen by the front office. The navigator uses knowledge about the current, wind speed, ship speed, etc. to keep track of the ship motion and instructs the pilot on the directional bearing of the ship.

A ship's pilot is the person at the wheel of the ship who must act on the navigator's instructions. The pilot is the one who knows the effects of the ship's momentum on its maneuverability and is responsible for avoiding all obstacles identified by the navigator.

A second analogy is that of flying an airplane. In this case the airline company plans the trips between two destinations, an on-board navigator keeps track of the aircraft location as it moves along a sequence of planned paths between different route checkpoints, and the pilot actually controls the airplane motion.

Finally, it should be remarked that there is no uniform set of words or phrases in the discipline of robotics to describe the different levels of intelligent control, so one is always advised to read the literature carefully before placing an article in one category or another.

1-2. Introduction to navigation and piloting

We shall first look at an algorithm for two-dimensional piloting activities in Section 2. Then in Sections 3 and 4 we examine navigation algorithms.

There are two basic models for navigation, those with complete information and those with incomplete information. For any application where the path planning is not perfect, the latter model is more appropriate, and this is the model of interest to us. With such a model, sensors are needed to provide feedback information to the AMR during its motion and that information is incomplete and imprecise.

Two types of feedback sensors are *tactile,* in which the AMR is allowed to come into contact with obstacles (or to within an allowable distance controlled by sensors), and *proximity* sensors that provide a map of the locally visible obstacles within a finite range. We shall look at navigation algorithms with tactile sensors in Section 3 and with proximity sensors in Section 4.

2. Piloting by a sequence of straight lines without sensor feedback

In this section we deal with how to follow paths with a low-level controller (without sensors). The objective is to balance smooth vehicle control with an efficient path specification. The path description is based on a sequence of directed straight lines and a coordinate transformation. A key feature of the approach is its pure geometrical nature and hardware independence; a key limitation is that it does not take into account the specific characteristics of an AMR. The presentation follows that of Ref. 1.

2-1. Path specification

We assume that the vehicle is a point, and that the objects are appropriately enlarged so as to correct for the finite size of the AMR footprint. (This procedure works best, incidently, with a nearly circular footprint since a constant distance can be used in all directions from the center of the footprint to enlarge obstacles; with a vehicle shaped like a ladder, for example, obstacle "blooming" becomes more messy to implement.)

For paths like that in Fig. 2-1, the path of the AMR is well approximated by a sequence of directed straight lines if a transient portion from one straight line to another can be appropriately generated. (In most cases, in fact, vehicle paths used in an indoor environment have straight-line segments as major components.) The transient portions are determined by a combination of "exit" specifications and characteristics of the feedback control system.

For a directed straight line X, the AMR *posture* (x, y, θ) can be described by its position coordinates (x, y) and vehicle orientation θ, measured counterclockwise from the X-axis. Each directed straight line is the X-axis of a local coordinate system XY.

Motion between two directed line segments, $X \rightarrow X'$, is specified by a coordinate transformation from (x, y, θ) to (x', y', θ'). Also, let (x_1, y_1) denote the coordinates of the origin O' of $X'Y'$ in the XY coordinates, and θ_1 the angle between the new axis X' and the old axis X, as shown in Fig. 2-2.

From Fig. 2-2 it follows that the transformation $[x_1, y_1, \theta_1]$ between the new posture and the old one can be written as a composite of the translation (x_1, y_1) and the rotation θ_1,

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix}. \qquad (2\text{-}1)$$

(Note the use of symbols [and] to define transformation instead of (and) to define posture!) Hence, rearrangement of this equation gives the transformation from the old posture to the new one,

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} \cos\theta_1 & \sin\theta_1 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \left(\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix} \right). \qquad (2\text{-}2)$$

Examples of transformations are shown in Fig. 2-3.

2-2. The MITCHI locomotion command system

Piloting along a path is complicated for the following reasons:

- Since the dead reckoning capability of a vehicle is limited, the AMR must continuously re-calibrate its current posture (x, y, θ).
- The AMR must be capable of modifying the navigator plan in case there are differences between the plan and the environment or moving objects that cannot be described in the navigator's plan.

The MITCHI command system (for "road" in Japanese) is one such system designed to control the AMR through a sequence of commands sent via the communication interface between the vehicle and the master control. A command is a unit of operation; for example, $< go >$ is a command that specifies a transformation $[x_1, y_1, \theta_1]$ together with an *exit* x_e. The motion of the AMR is influenced by the position of the exit, as seen in Fig. 2-4. An arbitrary path for a sequence of straight lines can be defined by a sequence of transformations as in Eq. (2-2) and exits.

The commands are broken down into two types: *slow* commands, such as $< go >$ and other commands that are executed sequentially one after another, and *fast* commands such as $< cancel >$ to cancel an earlier command, or $< stop0 >$, to decelerate immediately until stopped, that need to be executed rapidly.

The vehicle control states in MITCHI are categorized in two classes, *active* and *waiting.* When the vehicle is in a waiting state, any commands stored in the buffer are not executed (on a "first-in-first-out" basis) until a $< start >$ command is received. The three types of active states are *active-stop, moving-stationary,* and *transient.* In the active-stop state, the vehicle remains stationary until it receives a $< go >$ or $< spin >$ command: in the moving-stationary state it travels on an X-axis with a constant speed, and in the transient state it is between an exit and an entry or is accelerating or decelerating.

The MITCHI command system is explained in Table 2-1 and is expressed with the syntax shown in Table 2-2.

2-3. Feedback control without sensors

It is necessary to impose limits on the vehicle acceleration and angular acceleration in order to maintain a reasonable path following capability, and to avoid tipping and excessive power requirements caused by "jack-rabbit" starts. Feedback control is necessary even in a stationary state because there are many types of disturbances to the motion, both internal and external to the vehicle.

To keep the vehicle traveling along an intended axis X with speed $v = dx/dt$, the y-coordinate should be small. Since $dy = \tan\theta\, dx$, and since $\tan\theta \approx \theta$ for θ small, it follows that for a stable solution of Eq. (2-2) to exist when the vehicle speed is constant, it is necessary that

$$dy/dt \approx v\theta. \tag{2-3a}$$

As a second necessary condition for a stable solution, a constraint must be placed on the angular acceleration $\alpha = d\omega/dt$, where the angular velocity $\omega = d\theta/dt$. Since y is coupled to θ via Eq. (2-3a), this can be satisfied by

$$\alpha \approx -k_1 y - k_2\theta - k_3\omega, \tag{2-3b}$$

where k_1, k_2, and k_3 are properly chosen positive constants. After differentiating Eq. (2-3b) and using Eq. (2-3a), it is seen that a stable solution of Eq. (2-2) exists if the third-order differential equation for θ,

$$d^3\theta/dt^3 + (k_2 - k_3^2)d\theta/dt + (vk_1 - k_2k_3)\theta \approx k_1k_3y \tag{2-4}$$

is satisfied for appropriate k_1, k_2, and k_3.

A more complicated set of control equations than this is necessary, however, when v is not constant and y is not necessarily small. The control equation given in Ref. 1 for the acceleration dv/dt is:

$$dv/dt = \begin{cases} 0, & \text{if} \quad v = V_{\max} \quad \text{and} \quad g(y,\theta,v,\omega) > 0, \\ 0, & \text{if} \quad v = -V_{\max} \quad \text{and} \quad g(y,\theta,v,\omega) < 0, \\ g(y,\theta,v,\omega), & \text{otherwise,} \end{cases} \tag{2-5a}$$

where

$$g(y,\theta,v,\omega) = \begin{cases} -A_v, & \text{if} \quad v \geq v_d(y,\theta,\omega), \\ = A_v, & \text{if} \quad v < v_d(y,\theta,\omega), \end{cases}$$

with

$$v_d(y,\theta,\omega) = V_{d0} - c_1|\theta - f_1(y)| - c_2|\omega|.$$

The corresponding control equation for the angular acceleration $d\omega/dt$ is:

$$d\omega/dt = \begin{cases} 0. & \text{if} \quad v = \Omega_{\max} \quad \text{and} \quad f(y,\theta,v,\omega) > 0, \\ 0, & \text{if} \quad v = -\Omega_{\max} \quad \text{and} \quad f(y,\theta,v,\omega) < 0, \\ f(y,\theta,v,\omega), & \text{otherwise,} \end{cases} \tag{2-5b}$$

where

$$f(y,\theta,\omega) = \max[\min[A_\omega, (-f_1(y) - k_2\theta - k_3\omega)]$$

with

$$f_1(y) = k_1 \max[\min[y_0, y], -y_0].$$

The values $V_{\max}$, $\Omega_{\max}$, A_v, A_ω, V_{d0}, c_1, c_2, k_1, k_2, k_3, and y_0 are positive constant parameters. For the "Yamabico 9" of Ref. 1, actual values are

Maximum velocity: $V_{\max} = 35\,\text{cm s}^{-1}$
Maximum angular velocity: $\Omega_{\max} = 2\pi\,\text{s}^{-1}$
Default velocity (changeable by a < *velocity* > command): $v_{d0} = 30\,\text{cm s}^{-1}$
Parameters for deceleration during rotation: $c_1 = 30\,\text{cm s}^{-1}/\pi$ and $c_2 = 30\,\text{cm s}^{-1}/(2\pi\,\text{s}^{-1})$
Acceleration: $A_v = 30\,\text{cm s}^{-2}$
Angular acceleration: $A_\omega = 2\pi\,\text{s}^{-2}$.

The architecture of the feedback control system is shown in Fig. 2-5.

2-4. Example of the MITCHI locomotion command system

An illustration of the MITCHI system with the Yamabico dynamic behavior is shown in Fig. 2-6 for the following control sequence:

$$
\begin{aligned}
&(1)\, G\, 0, 0, 0, 0;\\
&(2)\, G\, 100, 100, 0, 90;\\
&(3)\, G\, 90, 100, 20, 180;\\
&(4)\, G\, 100, 130, 0, -90;\\
&(5)\, G\, 200, 170, 0, -90;\\
&(6)\, G\, 120, 130, 70, 180;\\
&(7)\, G\, 170, 180, 0, 90;\\
&(8)\, G\, 100, 100, 40, 0;\\
&(9)\, P\, 150;\\
&(10)\, S.
\end{aligned}
\tag{2-6}
$$

2-5. Figures and tables for Section 2

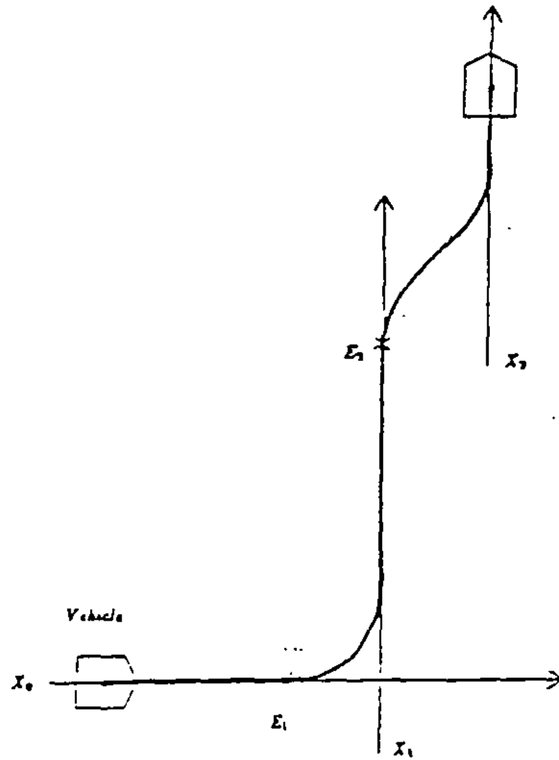


Fig. 2-1. Path specification by directed straight lines. (From Ref. 1.)

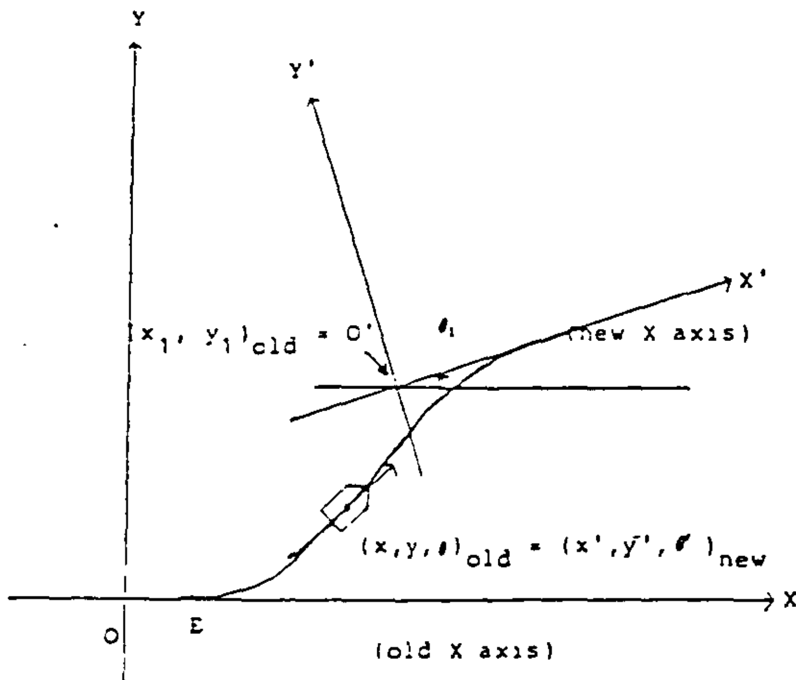


Fig. 2-2. Coordinate transformations (From Ref. 1.)

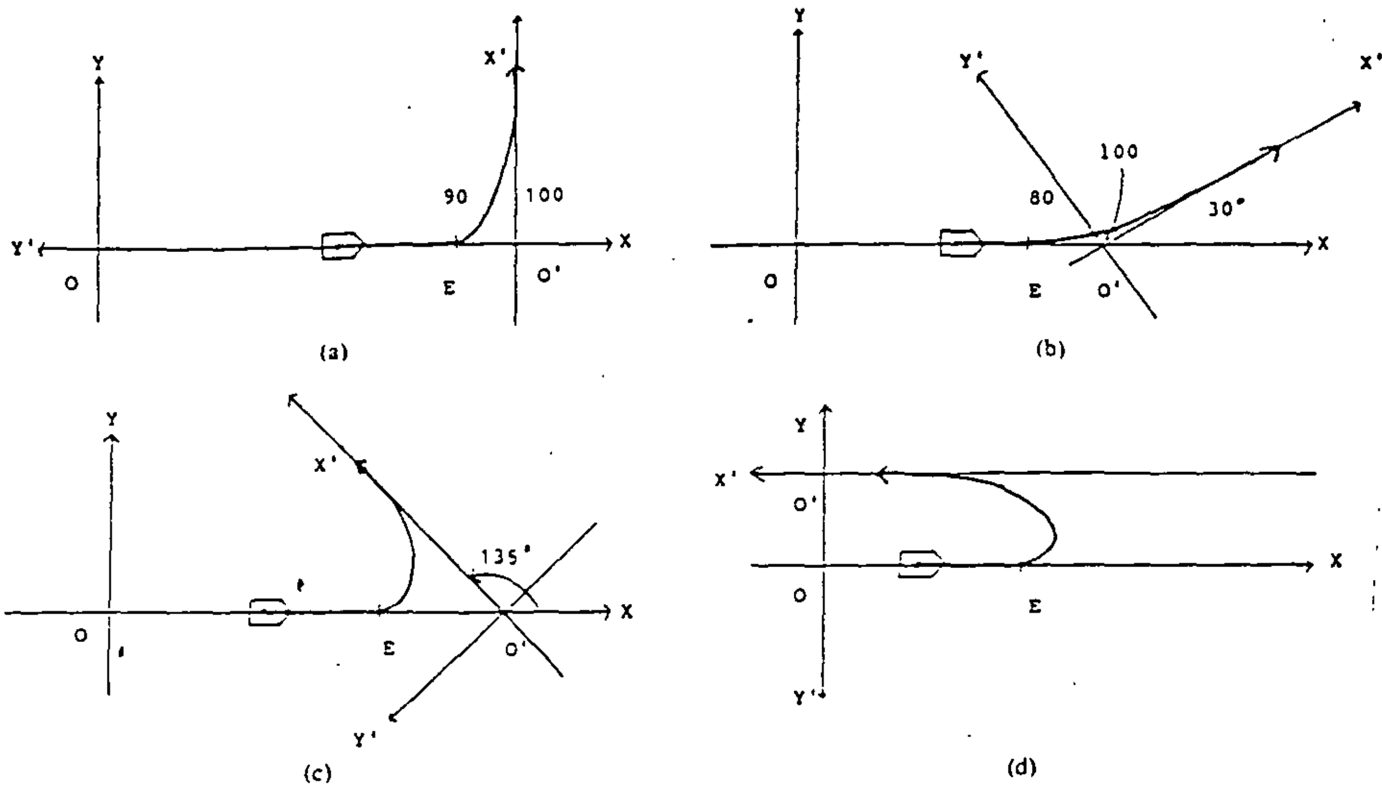


Fig. 2-3. Examples of transformations. (a) [100, 0, 90] (left turn). (b) [100, 0, 30] (left turn by 30°). (c) [100, 0, 135] (left turn by 135°). (e) [0, 50, π] (u-turn). (From Ref. 1.)

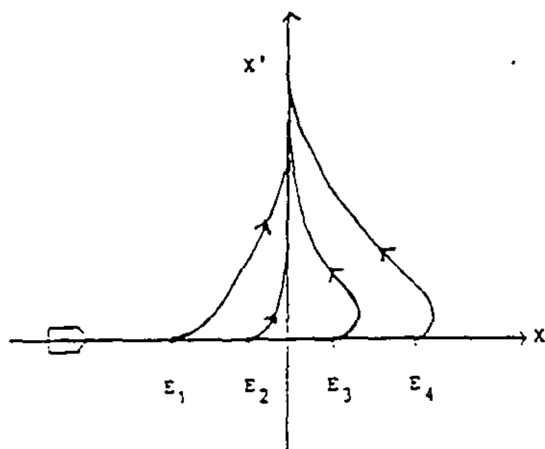


Fig. 2-4. Effects of different exit positions. (From Ref. 1.)

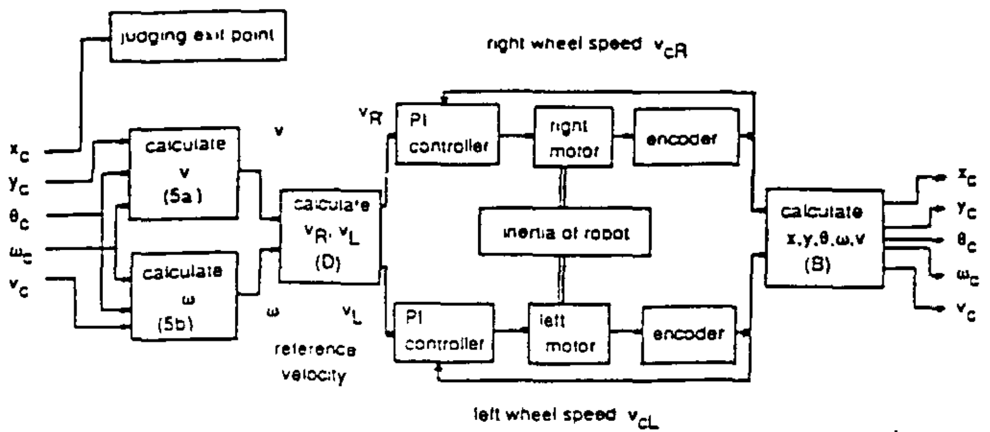


Fig. 2-5. Architecture of feedback control system (From Ref. 1.)

Fig. 2-6. An example of a trajectory generated by the MITCHI simulator. (From Ref. 1)

Table 2-1. MITCHI command system. (From Ref. 1.)

```
⟨command⟩ ::= ⟨slow command⟩ / ⟨fast command⟩
⟨slow command⟩ ::= ⟨go⟩ / ⟨stop⟩ / ⟨reverse⟩ / ⟨spin⟩ /
                   ⟨wait⟩ / ⟨velocity⟩ / ⟨control⟩
⟨go⟩ ::= G⟨e⟩,⟨x⟩,⟨y⟩,⟨θ⟩; / G,⟨x⟩,⟨y⟩,⟨θ⟩;
⟨stop⟩ ::= P⟨x⟩;
⟨reverse⟩ ::= R
⟨spin⟩ ::= N⟨θ⟩;
⟨wait⟩ ::= W
⟨velocity⟩ ::= V⟨v⟩;
⟨control⟩ ::= C⟨p1⟩, ⟨p2⟩;
⟨fast command⟩ ::= ⟨start⟩ / ⟨adjust⟩ / ⟨set⟩ /
                   ⟨global-get⟩ / ⟨stop0⟩ / ⟨velocity0⟩ /
                   ⟨cancel⟩ / ⟨servo⟩ / ⟨freemotor⟩ /
                   ⟨setbrake⟩ / ⟨resetbrake⟩
⟨start⟩ ::= S
⟨adjust⟩ ::= A⟨x⟩,⟨y⟩,⟨θ⟩;
⟨set⟩ ::= H⟨x⟩,⟨y⟩,⟨θ⟩;
⟨global-set⟩ ::= I⟨x⟩,⟨y⟩,⟨θ⟩;
⟨get⟩ ::= T
⟨global-get⟩ ::= J
⟨stop0⟩ ::= Q
⟨velocity0⟩ ::= U⟨v⟩;
⟨cancel⟩ ::= L
⟨servo⟩ ::= E
⟨freemotor⟩ ::= F
⟨setbrake⟩ ::= B
⟨resetbrake⟩ ::= K
⟨x⟩ ::= ⟨integer⟩
⟨y⟩ ::= ⟨integer⟩
⟨θ⟩ ::= ⟨integer⟩
⟨e⟩ ::= ⟨integer⟩
⟨v⟩ ::= ⟨integer⟩
⟨p1⟩ ::= ⟨integer⟩
⟨p2⟩ ::= ⟨integer⟩
```

Table 2-2. Syntax for the MITCHI command system. (From Ref. 1.)

1) ⟨*go*⟩*:* In the *moving-stationary* state, causes the transformation [⟨x⟩, ⟨y⟩, ⟨θ⟩] to be executed and the vehicle to attempt to ride on the new X-axis, when the condition $x \geq$ ⟨e⟩ is satisfied, i.e., the vehicle passes the exit (Figs. 5, 6, and 7). In the *active-stop* state, it causes the transformation [⟨x⟩, ⟨y⟩, ⟨θ⟩] to be immediately executed and the vehicle to attempt to ride on the new X-axis. In this case, the parameter ⟨e⟩ is ignored (Fig. 11).

2) ⟨*stop*⟩*:* Stops the vehicle at a given X coordinate of ⟨x⟩. If the vehicle goes beyond that point, it goes back up to the point.

3) ⟨*reverse*⟩*:* Exchanges the front and back faces of the vehicle. This does not cause any coordinate transformations (Fig. 12).

4) ⟨*spin*⟩*:* Makes the vehicle turn ⟨θ⟩ degrees. This ⟨*spin*⟩ is effective only in the stop state and does not cause any coordinate transformations (Fig. 13).

5) ⟨*wait*⟩*:* Changes the vehicle state from *active* to *wait*.

6) ⟨*velocity*⟩*:* Sets the vehicle's traveling speed to ⟨v⟩ cm/s. The default value is specified in the initialization.

7) ⟨*control*⟩*:* Sets the control parameter ⟨*p1*⟩ of MITCHI the value ⟨*p2*⟩. The details of specification are not given here. Some are discussed in Section V-E.

8) ⟨*start*⟩*:* Changes the vehicle state from *waiting* to *active*.

9) ⟨*adjust*⟩*:* Immediately executes the transformation [⟨x⟩, ⟨y⟩, ⟨θ⟩]. An exit need not be specified as in the slow ⟨*go*⟩ command, because the current position is considered an "exit." This command is useful for dynamically adjusting the vehicle's position or local path in both *moving-stationary* and *stop* states. For example, an ⟨*adjust*⟩ command "A0, 50, 0;" causes an effect similar to the transformation shown in Fig. 6(d).

10) ⟨*set*⟩*:* Changes the current local posture into (⟨x⟩, ⟨y⟩, ⟨θ⟩) immediately. The effect is similar to an ⟨*adjust*⟩ command. Suppose the vehicle is in a stationary state and is at posture (100, 0, 0,). Then a ⟨*set*⟩ command "H100, − 50, 0;" causes the same effect as that of the ⟨*adjust*⟩ command "A0, 50, 0;" as shown above.

11) ⟨*global-set*⟩*:* Changes the current global posture to (⟨x⟩, ⟨y⟩, ⟨θ⟩) immediately.

12) ⟨*get*⟩*:* Asks the vehicle to transmit the current local posture (x, y, θ).

13) ⟨*global-get*⟩*:* Asks the vehicle to transmit the global posture (x, y, θ).

14) ⟨*stop0*⟩*:* Decelerates immediately to stop the vehicle.

15) ⟨*velocity0*⟩*:* Changes the vehicle's traveling speed to ⟨v⟩ cm/s immediately.

16) ⟨*cancel*⟩*:* Clears all commands in the command buffer which have previously been sent from the master.

17) ⟨*servo*⟩*:* Starts motor servoing in order to try to keep its current position even in the stop state.

18) ⟨*freemotor*⟩*:* Kills motor servoing so that the vehicle can be moved manually.

19) ⟨*setbrake*⟩*:* Applies the mechanical brake.

20) ⟨*resetbrake*⟩*:* Releases the mechanical brake.

3. Navigation with tactile sensors

Tactile sensing is the simplest form of sensing in which the AMR, modeled as a point object, is assumed to operate in a straight line from its *start* position S until it reaches its *target* T or interacts with the ith obstacle at *hit point* H_i. (Note that an actual collision with an obstacle need not take place if the "bloomed" obstacles incorporate a nonzero minimum interaction distance as a safety factor.) The AMR then follows the perimeter of the obstacle until it reaches a *leave point* L_i on the obstacle and continues on another straight line journey toward T. Since there is no information to decide in which direction to circumvent the obstacle, one can assume for simplicity that the local direction is always left as in Fig. 3-1.

The presentation of this simple approach to obstacle avoidance follows that of Refs. 2 to 4. We shall consider two algorithms proposed for AMRs with tactile sensors, Bug1 and Bug2. The algorithms differ according to the available information.

3-1. The Bug1 algorithm

The procedure for Bug1 to be executed at any point of a continuous path consists of the following steps (see Fig. 3-2):

1. From the point L_{i-1} (with $L_0 = S$), move toward T along the straight line (S, T) until one of the following occurs:

   (a) T is reached; the procedure stops.

   (b) An obstacle is encountered and a hit point H_i is defined; go to step 2.

2. Using the accepted local direction, follow the obstacle boundary. If T is reached, stop. While traversing, store the coordinates of the current point Q_m located the closest to T. After traversing the whole boundary and returning to H_i, define a new leave point $L_i = Q_m$ and go to step 3.

3. If the straight line segment between L_i and T crosses the obstacle at point L_i, stop because the target is not reachable. Otherwise, follow the shorter path along the boundary to L_i and set $i = i + 1$; go to step 1.

Bug1 requires the algorithm be executed at every point of a continuous path. The procedure uses three registers, R_1, R_2, R_3, to store intermediate information until reaching a new hit point, whereupon all three are reset to zero. Register R_1 is used to store the current coordinates of the point, Q_m, on the boundary at which the distance $d(Q, T)$ between a point on the boundary Q and T is a minimum, while R_2 integrates the length of the obstacle boundary starting at H_i and R_3 integrates the length of the obstacle boundary starting at Q_m.

For Bug1 it is sufficient if the AMR knows its direction toward T and its distance from T instead of its exact coordinates. This can be incorporated into the algorithm by storing in register R_1 the minimum distance $d(Q_m, T)$ of the current point Q_m to T, rather than the coordinates of Q_m; then in step 3 the AMR can compare its current distance to T with that stored in R_1.

It is of interest to check the upper bound of the necessary distance of travel of the AMR to reach T with Bug1. The length of the path will never exceed the limit

$$P = D + 1.5 \sum_i p_i, \tag{3-1}$$

where D is the straight line distance $d(S,T)$ between the points S and T, and p_i is the perimeter of the ith obstacle. Here the sum over i includes all obstacles intersecting the circle of radius D centered at T.

3-2. The Bug2 algorithm

The procedure for Bug2 to be executed at any point of a continuous path consists of the following steps (see Fig. 3-3):

1. From the point L_{j-1} (with $L_0 = S$), move toward T along the straight line (S,T) until one of the following occurs:

   (a) T is reached; the procedure stops.

   (b) An obstacle is encountered and a hit point H_j is defined; go to step 2.

2. Using the accepted local direction, follow the obstacle boundary until:

   (a) T is reached, in which case stop.

   (b) The straight line (S,T) is met at a point Q such that the distance $d(Q,T) < d(H_j,T)$ and the straight line (Q,T) does not cross the current obstacle at the point Q. Define the leave point $L_j = Q_m$, set $j = j+1$, and go to step 1.

   (c) The AMR completely traverses the boundary and returns to H_j without having defined the next hit point H_{j+1}, in which case stop since the target is trapped and cannot be reached.

Unlike Bug1, with the Bug2 algorithm the same obstacle can be hit more than once, as seen in Fig. 3-4. (That is the reason the index j was used for a hit, to distinguish it from the index for obstacle i.) Also, the relationship between the perimeters of the obstacles and the length of the paths generated by Bug2 is not as clear as in the case of Bug1. The path around an obstacle is sometimes shorter than the perimeter, as in Fig. 3-3, but with obstacles as in Figs. 3-4 and 3-5 things can get much worse.

It is of interest to check the upper bound of the necessary distance of travel of the AMR to reach T with Bug2. The length of the path will never exceed the limit

$$P = D + 0.5 \sum_j n_j p_j, \tag{3-2}$$

where D is the distance $d(S,T)$ and n_j is the number of intersections between the straight line (S,T) and the jth obstacle. The sum over i refers to the obstacles intersecting the straight lIne segment (S,T).

While $n_j = 2$ if the jth obstacle is convex, examples of $n_j = 10$ and $n_j = 16$ are shown

in Fig. 3-4. If all obstacles are convex, then in the worst case the length of the path is

$$P = D + \sum_j p_j, \tag{3-3}$$

while on the average the path length is

$$P = D + 0.5 \sum_j p_j. \tag{3-4}$$

On a practical level, Bug1 is rather "conservative" while Bug2 is more "aggressive" and more efficient in many cases.

3-3. Figures and tables for Section 3

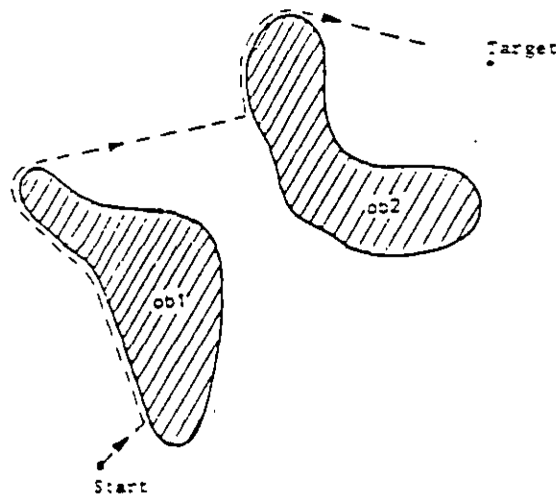


Fig. 3-1. A left local direction AMR path. From Ref. 2.

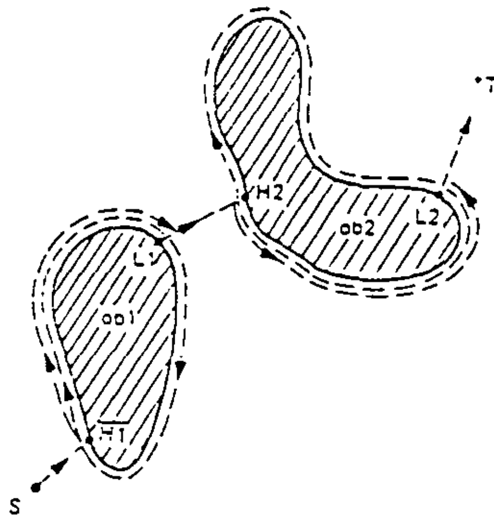


Fig. 3-2. AMR path (dotted lines) under the Bug1 algorithm, with obstacles ob1 and ob2, hit points H_1 and H_2, and leave points L_1 and L_2. From Ref. 4.

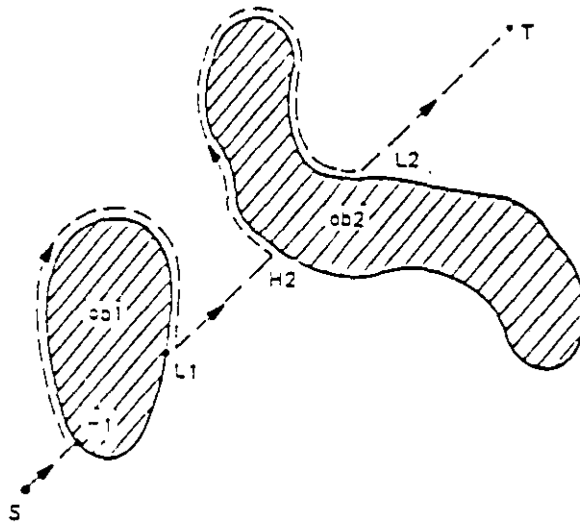


Fig. 3-3. AMR path (dotted line) under the Bug2 algorithm, with obstacles ob1 and ob2, hit points H_1 and H_2, and leave points L_1 and L_2. From Ref. 4.

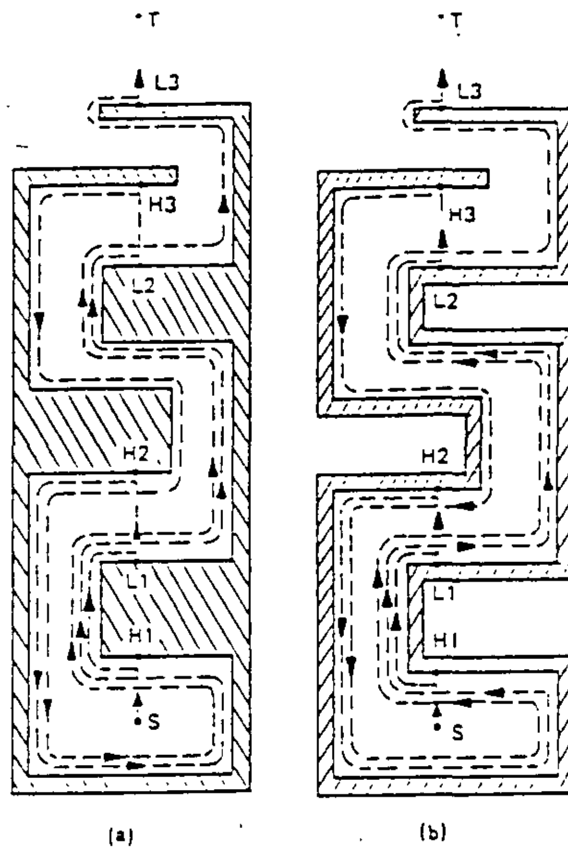


Fig. 3-4. AMR paths (dotted lines) around two maze-like obstacles under the Bug2 algorithm. While both obstacles (a) and (b) are equally complex, the straight line (S, T) crosses the obstacle 10 times, $n_j = 10$, and for (b), $n_j = 16$. (From Ref. 4.)

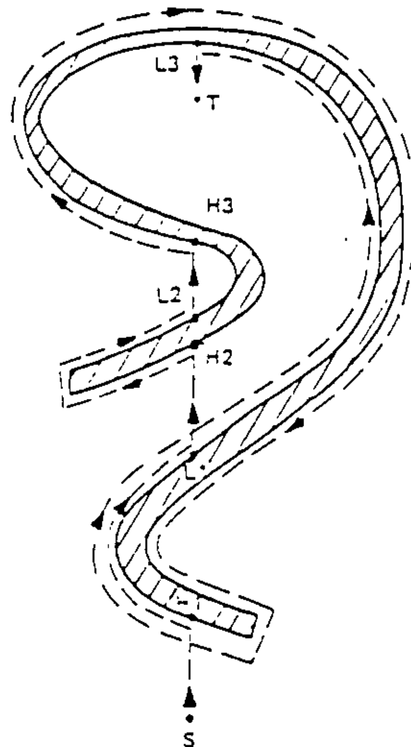


Fig. 3-5. AMR path when S is outside the obstacle and T is inside. From Ref. 4.

BUG-2

4. Piloting with proximity sensors

In an environment where only local information is available and piloting decisions must be made dynamically, in reponse to a changing environment as the vehicle moves along its trajectory, the AMR must be able to make decisions quickly. These decisions must include consideration of the kinematic and dynamic constraints of the AMR. It is within this framework that ***satisficing feedback strategies*** have been developed in Refs. 5 to 7. For such a strategy, it is necessary ***to satisfice:*** "to decide on and pursue a course of action that will satisfy the minimum requirements necessary to achieve a particular goal" (as defined in the Supplement to the Oxford English Dictionary, 1982). Thus satisficing decision strategies are based only on local feedback information and must be done in real time while avoiding obstacles and converging to a goal.

The algorithm necessary to implement the satisficing strategy is broken down into two parts: a *subgoal selection algorithm* (SSA) and a *steering decision algorithm* (SDA). In the first a subgoal is generated that provides a temporary direction to pursue when the final goal is not visible, and in the second a reference trajectory is generated that leads the AMR until the next subgoal is generated.

4-1 Subgoal selection algorithm

The objective of subgoal selection algorithm is to enable the AMR to move using feedback information about the locally visible environment while avoiding collisions with unanticipated obstacles and without stopping to replan the path to the goal. This has advantages over a piloting algorithm without feedback or with tactile feedback which requires the AMR to "stop, look, and move," i.e., to stop and generate a new collision-free path before again moving.

The AMR is modeled as a point object that is to be steered from an initial position $\mathbf{p}_0$ to the final goal $\mathbf{p}_g$ through a 2D space containing a finite number of obstacles. The obstacles are appropriately "bloomed" convex polygons; the minimum physical distance between points on any two obstacles must exceed the smallest circle encompassing the AMR (so that the vehicle is capable of passing between them). The AMR is assumed to have an SDA and a servo control system capable of following a reference trajectory with a known accuracy, provided that trajectory satisfies the necessary kinematic constraints. Also, estimates of the AMR position, orientation, and velocity in the global coordinate frame must be available for piloting decisions at regular time intervals.

For the kth iteration of the SSA, the AMR position is denoted by $\mathbf{p}_k$ and a local map is presumed to be created that consists of a set $O(\mathbf{p}_k)$ of line segments representing obstacle faces that are visible from $\mathbf{p}_k$, as illustrated in Fig. 4-1. (Note that the AMR need not be stopped at $\mathbf{p}_k$ since the time delay caused by operation of the sensors and map processor can be accounted for by the evaluation of the safeness of the subgoals.) The dynamic state of the AMR at time t, including its position, can be denoted by $\mathbf{x}(t)$.

In order for the SSA to generate the next subgoal, it is necessary to define a candidate subgoal $\mathbf{g}^*$ to be *safe* at time t with respect to local map $O(\mathbf{p}_k)$ and the current dynamic state $\mathbf{x}(t)$ if the SDA can bring the AMR to rest at $\mathbf{g}^*$ along a collision-free trajectory in

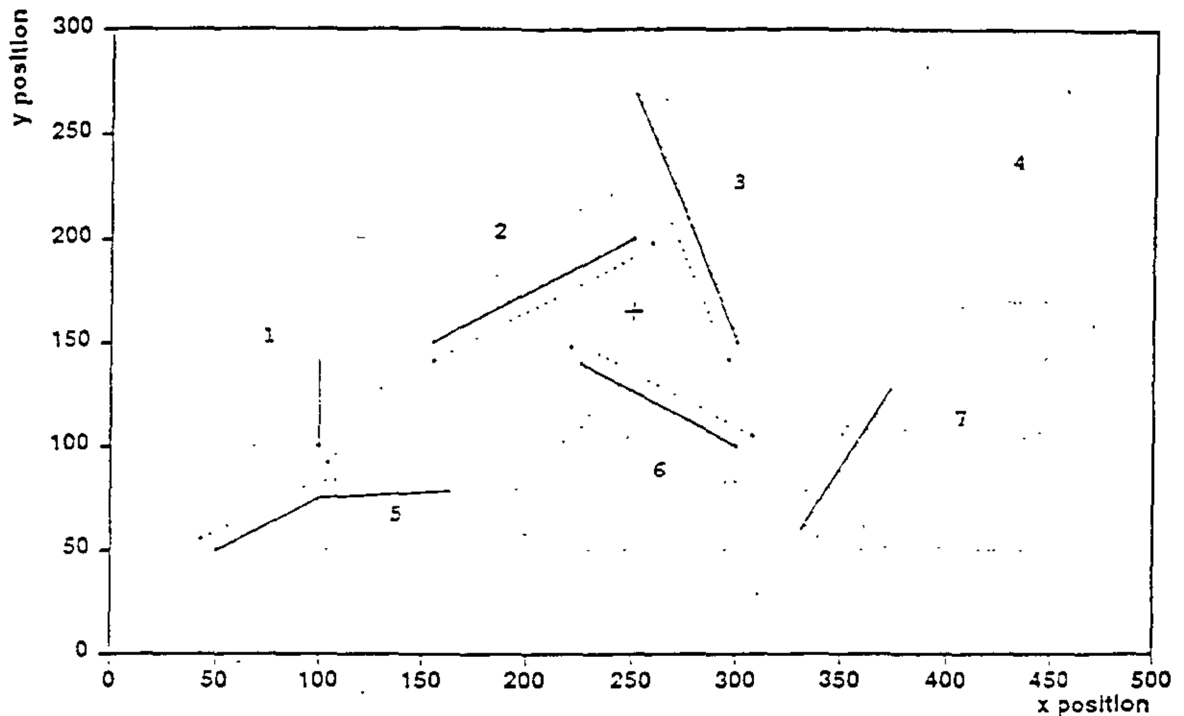


Fig. 4-1. Local obstacle map from the position $p_k(+)$, together with the corresponding "bloomed" local map (dotted lines). The bold lines indicate the faces of the visible obstacles. From Ref. 7.

$O(p_k)$. Note that the safeness of a subgoal is evaluated at the current time t, which is different from the time at which the sensor data for local map $O(p_k)$ were taken.

The SSA has been developed using functions and data structures in a pseudoprogramming language in which small bold-faced letters represent points in the plane, large bold-faced letters represent other data structures, and small capitals are used for function names, with brackets $<,>$ delimiting function arguments. A straight line (unbounded) containing two points **a** and **b** is denoted by **ab**, with the open or closed termination of a line segment indicted by a parenthesis or bracket, respectively. For example, (**ab**] denotes the line segment from **a** to **b** that contains **b** but not **a**, while [**ab**) is the ray beginning at and including **a** and reaching **b**.

The following 10 pages is an explanation of the SSA taken directly from pp. 78-87 of Ref. 7, including at the end a summary of definitions of all the notation and functions. We will not discuss the proof of convergence of the algorithm given in Refs. 5 and 6.

The function INT<**ab**.**cd**> equals the point at which the two lines **ab** and **cd** intersect. If the arguments of INT<...> do not intersect the function returns the symbol NIL, which is used throughout the algorithm definition to indicate empty sets or vacuous conditions.

At the kth execution of the SSA, the local map consists of a set $O(\mathbf{p}_k)$ of line segments representing obstacle faces which are visible from $\mathbf{p}_k$. Connected lines in $O(\mathbf{p}_k)$ are faces of the same obstacle. Given two points **a** and **b**, the function OBS<**a**.**b**> is defined as the line segment (obstacle face) in $O(\mathbf{a})$ which intersects the line segment (**ab**), along with all other faces in $O(\mathbf{a})$ connected to the intersected face. When point **b** is visible from point **a**, OBS<**a**.**b**> equals NIL.

Obstacle *edges* in the local map with respect to the position $\mathbf{p}_k$ are defined as the extreme points of the connected lines in $O(\mathbf{p}_k)$, that is, they are the extreme points on each connected set of obstacle faces visible from $\mathbf{p}_k$. We denote the set of edges at iteration k by $\mathbf{E}(\mathbf{p}_k)$. Note that if obstacle A is only partially visible from $\mathbf{p}_k$, due to the presence of another obstacle, B, then at least one extreme point on obstacle A is not actually visible. To distinguish edges which are actually vertices of the obstacle models from extreme points arising from obstructions, we let $\mathbf{E}'(\mathbf{p}_k)$ denote the set of edges which are actually visible extreme points on the visible obstacles in $O(\mathbf{p}_k)$. Given two points **a** and **b**, if OBS<**a**,**b**> = NIL, then the two extreme points on the connected lines in OBS<**a**,**b**> correspond to two edges in $\mathbf{E}(\mathbf{p}_k)$, but not necessarily in $\mathbf{E}'(\mathbf{p}_k)$. From the

perspective of p_k. we distinguish the two edges on CBS<p_k,b> as "right edge" and "left edge", with the obvious meaning. Note that a particular physical obstacle can lead to more than one set of connected faces in $O(p_k)$ when it is partially obstructed from view by other obstacles, but it can lead to no more than two visible edges in $E'(p_k)$.

Subgoals are always associated with obstacle edges or vertices in a local map. A subgoal **g** and its associated obstacle edge (or vertex) **e** make up the fundamental data structure referred to as a subgoal-edge pair, denoted by (**g**,**e**). For an edge $\mathbf{e} \in E_K$, the associated subgoal is defined as a point **g** a distance $\varepsilon \geq 0$ from **e**. The precise locations of subgoals **g** with respect to associate edges **e** are given below in the descriptions of the functions NEXT_VERTEX, CHOOSE_EDGE, OTHER_EDGE, EXTEND_FACE, OTHER_VERTEX, and SUBGOAL. We choose $\varepsilon \leq \kappa/2$ to assure that the subgoal for an edge is not on another obstacle (which must be at least a distance κ from **e**). The purpose of setting the subgoal a certain distance from the edge is to guide the AMR around a vertex to a vantage point from which it is guaranteed that further subgoals will be generated.

Subgoal-edge pairs generated in the SSA are stored on a dynamic stack S which is operated on using the standard functions POP<S> and PUSH<(g,e),S>. In the remainder of the paper (g,e) denotes the subgoal-edge pair currently being pursued by the SDA, and (g^T,e^T) denotes the top of the stack S, that is, (g^T,e^T)=POP<S>. The function HEIGHT<S> equals the number of subgoal-edge pairs in the stack S.

$C\{a_1,....a_n\}$ denotes the open convex hull of a set of n points in the plane $\{a_1,....a_n\}$. **L** is a particular subset of subgoals generated by the SSA. The meaning of the points in **L** is described below in the discussion of step 4 of the SSA.

The SSA is given in Fig. 5-2. In the remainder of this section we discuss each step in turn, providing further definitions of variables, data structures and functions as needed.

The algorithm is initialized in step 1 when k=0 and the AMR is at rest at p_o. If there are no obstacles between the AMR and the final goal p_g, the subgoal g is set equal to p_g and the SSA terminates. It is assumed that the SDA can drive the AMR from rest to any visible point. Thus, p_g is a safe subgoal at this stage and no other subgoals are needed. When p_g is not visible initially, it is pushed on the stack (with a default value of p_g for the edge), L and h (defined below) are initialized, and the algorithm is continued at step 5. It will be shown in the following section that in this case more subgoals will be put on the stack in step 6 and a safe subgoal-edge pair will be generated.

The SSA terminates at step 2 when the final goal is already being pursued (no further subgoals are needed), or when the ray from the goal on top of the stack g^T through p_k does not intersect the ray from the current edge **e** through the goal **g**. As illustrated in

```
step 1:  if (k=0) then
             if (OBS<p_o,p_t>=NIL) then
                 (g,e) ← (p_t,p_t)
                 exit
             else
                 PUSH<(p_t,p_t),S>
                 L ← NIL
                 h ← p_o
                 (g', e') ← CHOOSE_EDGE<OBS<p_o, p_t>>
                 goto step 5
step 2:  if ( g=p_t or INT<[g^T p_k, (eg>=NIL) then
             exit
step 3:  SAVE<S,(g,e),[ab],L,h>
             if (HEIGHT<S> ≥ 3) then
                 goto step 6
step 4:  if (HEIGHT<S>=1) then
             if (OBS<p_k,p_t>=NIL) then
                 goto step 7
             if (O(p_k) ∩ C(p_k,g,e) ≠ NIL) then
                 goto step 8
             (g', e') ← NEXT_VERTEX < O(p_k), (g, e)>
             if (INT < (ge'), (ep_t) > ≠ NIL or e'=e) then
                 if (OBS<p_k,p_t> ∩ C(g,e,p_t)=NIL) then
                     goto step 8
                 else
                     (g', e') ← CHOOSE_EDGE<OBS<p_k, p_t>>
                     L ← L ∪ {h}
                     h ← g
             if (INT<(hg'],(p_t x> ≠ NIL and INT<(hg'],(p_t x)>=NIL for some x ∈ L) then
                 (g',e') ← OTHER_EDGE<O(p_k),(g',e')>
         else
             if (O(p_k) ∩ C(p_k,g^T,e^T) ≠ NIL) then
                     goto step 6
             (g', e') ← EXTEND_FACE < O(p_k), (g^T, e^T)>
             if (INT<(hg'],(p_t x> ≠ NIL and INT<(hg'],(p_t x)>=NIL for some x ∈ L) then
                 (g',e') ← OTHER_VERTEX<O(p_k),(g',e')>
             POP<S>
step 5:  [ab] ← [p_k e']
         (g, e) ← (p_k, p_k)
         PUSH < (g', e'), S >
step 6:  while (O(p_k) ∩ C(p_k,g^T, e^T) ≠ NIL or OBS<p_k,g^T> ≠ NIL)
             (g',e') ← CREATEG<p_k,(g^T,e^T),(g,e),[ab]>
             if ((g',e')=NIL) then
                 goto step 8
             else
                 PUSH<(g',e'),S>
step 7:  (g,e) ← POP<S>
         if (SAFE<g, x(t), O(p_k)>then
             exit
step 8:  RESTORE<S,(g,e),[ab],L,h>
         end
```

Figure 5-2: Subgoal Selection Algorithm (SSA)

Fig. 5-3, this second condition assures that a new subgoal is sought only after the AMR has gone beyond the current edge **e** so that visibility of the subgoal g^T on the top of the stack is not blocked by the obstacle face that produced **e**.

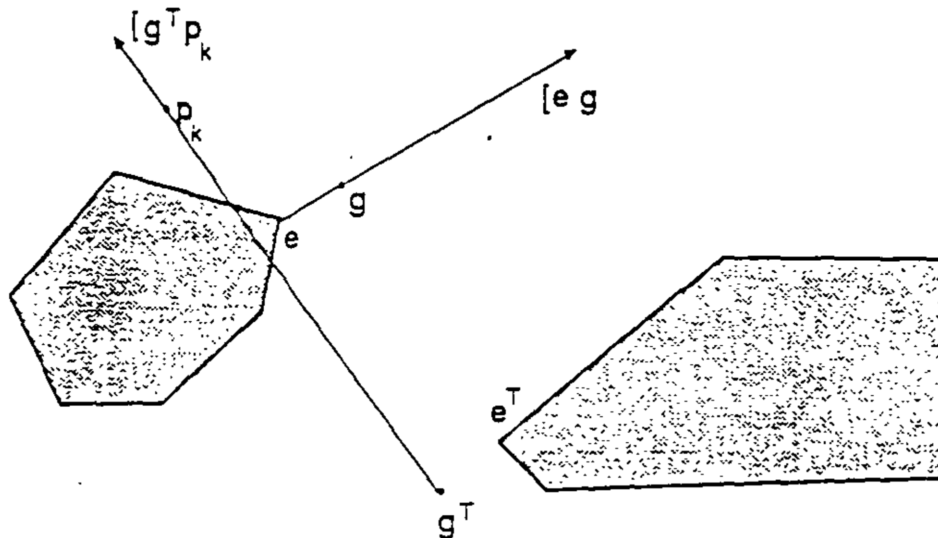


Figure 5-3: Condition tested in step 2: p_k must be past current edge e before the visibility of subgoal g^T is evaluated.

In step 3 the function SAVE stores the current *context* which consists of the stack **S**, the current subgoal-edge pair (g,e), line segment [ab], a set of previous subgoals **L** and vector **h** ([ab], **L**, and **h** are defined below). After the context is saved in step 3, the algorithm jumps to step 6 if there are three or more subgoal-edge pairs in the stack **S**. If it is determined in steps 4 through 7 that a new subgoal should not be selected on the current iteration, the context stored in step 3 is retrieved by the function RESTORE in step 8.

Step 4 is executed for two mutually exclusive cases, namely, when only the final goal p_g remains in the stack (HEIGHT<S>=1) or when the stack contains one subgoal in addition to the final goal (HEIGHT<S>=2). In the first case, if the final goal is visible from p_k, the algorithm jumps to step 7 to evaluate whether the final goal is safe. If the final goal is not visible, the next condition in step 4 assures the line segment (e, g) is visible. If it is not visible, the SSA jumps to step 8 to restore the original context. Otherwise, step 4 continues and the function NEXT_VERTEX generates a subgoal-edge pair (**g′**, **e′**) where **e′** is the next visible vertex on the obstacle containing the current edge **e** in the direction of g. The associated subgoal g′ is on the extension of the face containing **e′**, a distance ε from e′. If the current edge e is an extreme point in $O(p_k)$, then NEXT_VERTEX returns (g′, e′)=(g, e), the current subgoal-edge pair.

The next If condition in step 4 determines whether the edge e′ is on a face between the current edge and the goal. This condition is illustrated in Fig. 5-4. If INT<(ge′),(ep_g)> = NIL or e′=e, the current edge is the last one that must be circumvented on that obstacle. In this case, before choosing an edge on another obstacle, a test is performed to see if the obstacle between p_k and p_g intersects the region C(g,e,p_g). The purpose of this test is to assure that the next obstacle lies between the current subgoal and p_g. If OBS<p_k, p_g>∩ C(g,e,p_g)=NIL, the SSA jumps to step 8 and terminates. If OBS<p_k,p_g>∩C(g,e,p_g) ≠ NIL, a subgoal-edge pair is generated by the function CHOOSE_EDGE from one of the two edges for OBS<p_k,p_g>. Since only local information is being used in the SSA, there is no "optimal" choice between these two edges. In our implementation of CHOOSE_EDGE we select the edge closest to p_g. We show in the following section that the SSA steers the AMR to the final goal no matter which edge is selected by CHOOSE_EDGE.

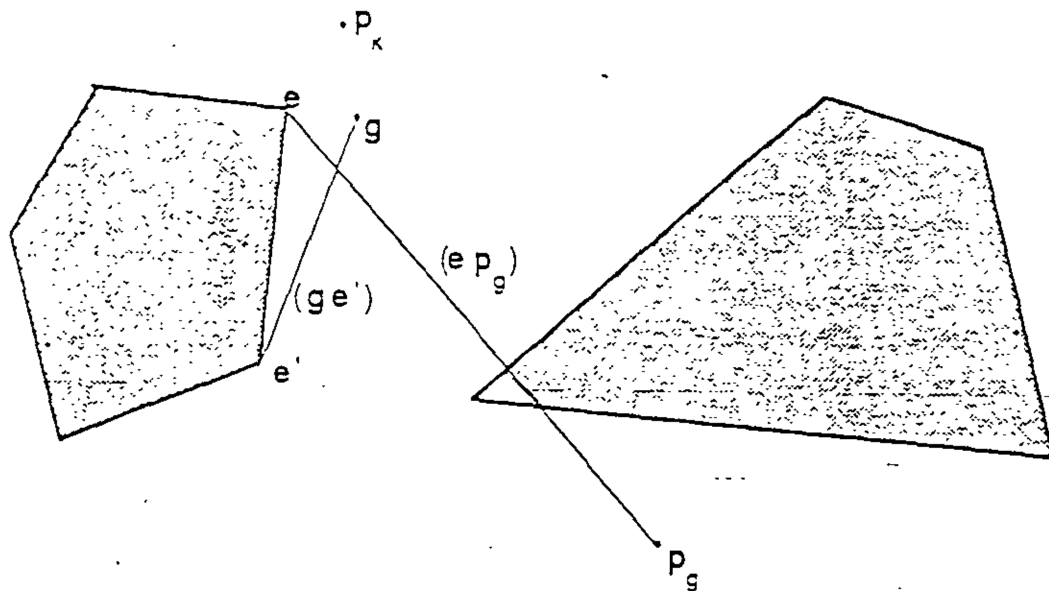


Figure 5-4: Condition tested in step 4: if vertex e′ is not a face blocking the view of p_g, a new obstacle is used to generate a subgoal.

When a new obstacle is used to generate the potential subgoal-edge pair, **h** is added to the set **L**. The variable **h** is then set equal to the current subgoal **g**. Thus, **h** is the final subgoal generated in step 4 for a particular obstacle before switching to a new obstacle and **L** is the set of previous values of **h**. The set of subgoals **L** serves as a "memory" of where the AMR has been so that it does not cycle around the final goal indefinitely. This is assured by the next condition tested in step 4 which, in words,

checks to see if the subgoal g' will take the AMR "behind" a previous subgoal in L. This condition is illustrated in Fig. 5-5. If it occurs, the function OTHER_EDGE generates a subgoal-edge pair from the other edge on the obstacle containing e'. This concludes step 4 when HEIGHT<S>=1.

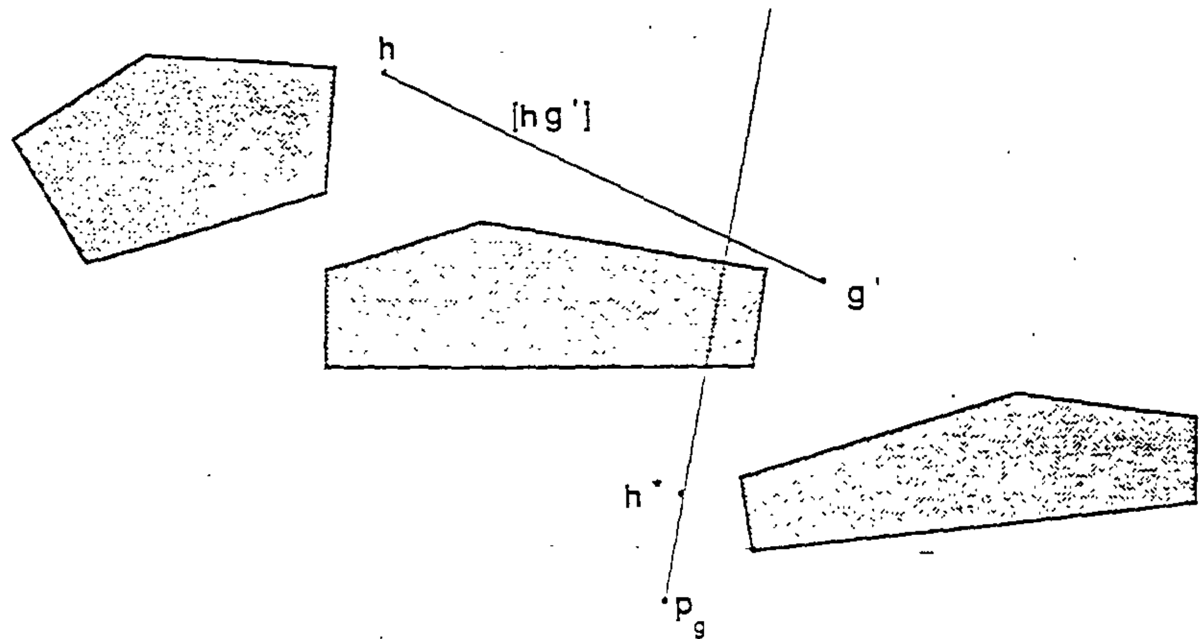


Figure 5-5: Example of candidate subgoal g' taking the line [hg'] "behind" a previous subgoal h^* in L in step 4.

In the second case of step 4 (when HEIGHT<S>=2), if the line segment $(g^T e^T)$ is not completely visible from p_k (which is the case when there are obstacles in $C(p_k, g^T, e^T)$, the SSA jumps to step 6. Otherwise, function EXTEND_FACE chooses e' as the most extreme visible point on the face colinear with $(g^T e^T)$ in the direction of g^T from e^T. The subgoal g' is generated as the extension of this face a distance ε from e'. Before going to step 5 to push this new subgoal edge pair on the stack, the same test is applied as described above for case one of step 4 to make sure the subgoal generated by EXTEND_FACE is not taking the AMR behind a previous subgoal in L. If it is, the other vertex of the face is chosen by OTHER_VERTEX to generate the subgoal-edge pair (g', e'). This subgoal-edge pair will replace the subgoal-edge pair on top of the stack (g^T, e^T) which is "popped" and discarded.

Step 5 updates the appropriate variables with the subgoal-edge pair (g', e') generated in step 4. The line [ab] is set equal to line segment $[p_k e']$, which is guaranteed not to intersect the interior of any obstacles. The current edge e is set equal to p_k for use in

the function CREATEG in step 6. The subgoal-edge pair (g',e') is then pushed on the stack. We shall see that the line segment [**ab**] provides the "memory" required to assure that if any other subgoal-edge pairs are pushed on top of (g',e'), they will guide the AMR to a point from which the subgoal g' is visible.

In step 6, the function CREATEG generates subgoal-edge pairs when the stack already contains a subgoal-edge pair generated previously in step 4. The function CREATEG is defined in Fig. 5-6. An edge selected by CREATEG must satisfy conditions related to its proximity to the current edge **e**, the edge e^T on top of the stack and the line [**ab**). If no such edge exists, NIL is returned. The function CREATEG chooses the edge e' in $E'(\mathbf{p}_k)$ (the set of visible edges) which is closest to the line [**ab**] while being in the region $C\{g,e,g^T,e\}$]. The significance of these conditions is explained in the following section where we prove the trajectory converges to the final goal in finite time.

```
CREATEG<p_k,(g^T,e^T),(g,e),[ab],E'(p_k)>
    A ← E'(p_k) ∩ {C{e,g, e^T,g^T} ∪ [g,g^T]}
    If (A=NIL) then
        (g',e') ← NIL
    else
        e' ← MIN_DIST <A,[ab]>
        g' ← SUBGOAL<e',[ab]>
    return (g',e')
```

Figure 5-6: Function CREATEG

As illustrated in Fig. 5-7, function SUBGOAL generates a subgoal g' a distance ε from **e'** on a line perpendicular to, and in the direction of, the line segment [**ab**]. Subgoal-edge pairs are generated by CREATEG and pushed on the stack until the line segment [$\mathbf{g}^T\mathbf{e}^T$) is visible. If CREATEG returns NIL, the algorithm jumps to step 8 and the original context is restored.

In step 7 the top of the stack is popped and the (visible) subgoal is evaluated by the function SAFE which returns the logical condition TRUE if the SDA can drive the AMR to **g** along a collision free path given the current system state $\mathbf{x}(t_k)$ and the local obstacle map $O(\mathbf{p}_k)$. If **g** is a safe subgoal, the algorithm terminates and this new subgoal is pursued by the SDA. If **g** is not safe, step 8 is executed and the original context is restored.

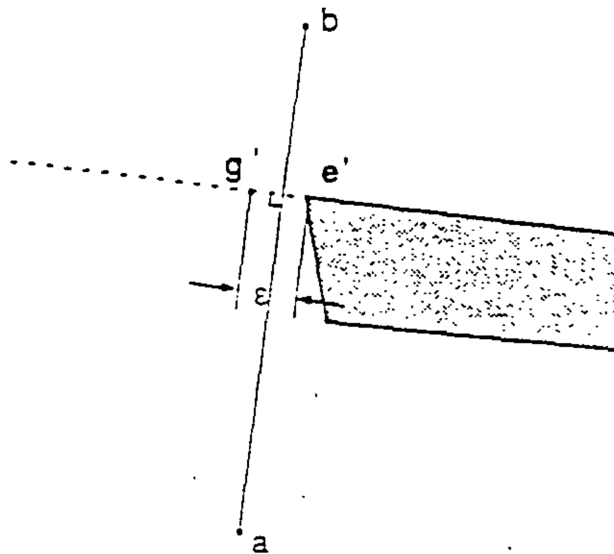


Figure 5-7: Location of subgoal g′ generated by function SUBGOAL for edge e′.

In the following, we list the notation, parameters and functions that have been defined in this section. They will be frequently refered to in the next section.

Notation

NIL
empty set or vacuous result

ε
distance between a subgoal g and its associated edge e

ab
straight line (unbounded) through points **a** and b

[ab
ray from point **a** through point b containing point **a**

(ab]
line segment from point a to point **b** containing b but not **a**

κ
minimum distance between obstacles

(g,e)
subgoal-edge pair currently being pursued

(g^T, e^T)
subgoal-edge pair on top of stack S

h
most recent extreme subgoal generated in step 4

P_0
initial position

p_g
final goal position

p_k
position at beginning of the kth execution of the SSA

x_k
dynamic state of the AMR at beginning of the kth iteration

$C(a_1,\ldots,a_n)$
open convex hull of points $a_1,\ldots,a_n$

E(a)
set of obstacle edges in O(a), that is, extreme points of the connected obstacle faces in O(a)

E'(a)
set of visible edges in E(a)

L
set of extreme subgoals generated in step 4 of SSA

O(a)
local map: set of obstacle faces (line segments) visible from point **a**

S
stack of subgoal-edge pairs maintained by SSA

Functions

CHOOSE_EDGE<OBS<p_k,p_g>>
returns subgoal-edge pair (g',e') with e' one of the two edge in E(p_k) on OBS<p_k,p_g> and g' a distance E from e' on extension of the fact terminating at e'

CREATEG<p_k,(g^T,e^T),(g,e),[ab],E'(p_k)>
generates subgoal-edge pairs in step 6, as defined in section 3

EXTEND_FACE<O(p_k),(g,e)>
returns subgoal-edge pair (g',e') where e' is the vertex (extreme point) of the face in O(p_k) containing **e** in direction of g and g' is defined as in CHOOSE_EDGE

HEIGHT<S>
number of subgoal-edge pairs in stack **S**

MIN_DIST<A,[ab]>
chooses one point in set A closest to line [ab]

NEXT_VERTEX<O(p_k),(g,e)>
returns subgoal-edge pair (g',e') with e' as the next vertex on obstacle in O(p_k) containing **e** in the direction of g and g' a distance ε from e' on extension of face [ee']

OBS<a,b>
set of obstacle faces in local map O(a) connected to and including a face intersected by line segment (ab)

OTHER_EDGE<O(p_k),(g,e)>
returns subgoal-edge pair (g',e') where e' is the other edge in E(p_k) on the obstacle in O(p_k) with edge e', and g' is defined as in CHOOSE_EDGE

OTHER_VERTEX<$O(p_k)$,(g,e)>
returns subgoal-edge pair (g′,e′) where e′ is the other vertex (extreme point) for the face in O(p) containing e, and g′ is defined as in CHOOSE_EDGE

POP<S>
removes and returns top subgoal-edge pair from stack S

PUSH<(g,e),S>
puts subgoal-edge pair (g,e) on stack S

RESTORE<S,(ge),[ab],L,h>
sets arguments equal to values store by SAVE

SAFE<g,x(t),$O(p_k)$>
returns logical TRUE if the SCA can bring the AMR to a stop at g from state x(t) along a path in the visible obstacle-free space in $O(p_k)$

SAVE<S,(g,e),[ab],L,h>
stores context of SSA

SUBGOAL<e,[ab]>
generates a subgoal g a distance ε from e on the line perpendicular to, and in the direction of, line segment [ab]

4-2. Steering decision algorithm

A steering decision algorithm must necessarily depend upon the type of AMR. The two general types are the omnidirectional AMR (OAMR) in which the vehicle is capable of pure rotation as well as translation, and the conventionally-steered AMR (CAMR) with front wheel steering and rear wheel drive. Since the Denning robots in our laboratory are of the OAMR type, we shall only examine the SDA for this type; fortunately, the algorithm is easier than that for the CAMR. The following 12 pages is an explanation taken directly from pp. 11-22 of the Ref. 6 manuscript, which also contains a discussion of the SDA for a CAMR which is not included here.

4. An SDA for Omnidirectional Mobile Robots

In this section, attention is focused on the realization of the SDA for an omnidirectional autonomous mobile robot (OAMR). OAMR is characterized by three degrees of freedom in the plane, in contrast to the two degrees of freedom of normal wheel-based AMRs. Starting from rest, an OAMR is capable of translating along any direction in the plane. In addition, it can also rotate. Examples of AMRs constructed with Omnidirectional characteristics include the Unimation robot [33], Uranus [34], and the Ilonator Cart [30].

4.1. OAMR Reference Model

For the purpose of reference trajectory generation, we consider the OAMR as a point in the global coordinate frame. When navigating through obstacle environments, the finite size of the OAMR can be compensated for by means of expanding the obstacle regions in the local map. (Simulation examples using expanded obstacle maps to navigate AMRs of finite size are given in Section 6.) Because of the omnidirectional property, we consider only the position of the OAMR when generating the reference trajectory. Thus, the reference state variables are the

position vector $r=p$, and velocity vector $\dot{r}=v$, of the robot in R^2. The steering vector is $u=a$, the acceleration of the point p, also in R^2. Thus, we can model the reference trajectory dynamics by two-dimensional double integrator dynamics:

$$\begin{pmatrix} \dot{p} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} 0 & I \\ 0 & 0 \end{pmatrix} \begin{pmatrix} p \\ v \end{pmatrix} + \begin{pmatrix} 0 \\ a \end{pmatrix}. \quad (2)$$

Integrating equation (2), we obtain the reference trajectory for the time interval $t \in [t_n, t_{n+1}]$ as:

$$p(t) = p(t_n) + (t-t_n)v(t_n) + 0.5(t-t_n)^2 a(t_n) \quad (3)$$

$$v(t) = v(t_n) + (t-t_n)a(t_n)$$

We represent the OAMR's operating limits by magnitude constraints on the acceleration and velocity vectors given by:

$$\|a\| \le a_{max} \quad (4)$$

and

$$\|v\| \le v_{max} . \quad (5)$$

These limits constitute the system constraint set U_s for the OAMR.

4.2. OAMR Free-space

Before defining the free-space for the OAMR, we first introduce some notation. A straight, unbounded line containing two points a and b is denoted by ab. The open or closed termination of a line segments is indicated by a parenthesis or bracket, respectively. For example, (ab] denotes the line segment from a to b which contains b but not a, and [ab denotes the ray beginning at and including a passing through b. In the following, we describe a set of rules for constructing the free-space for the OAMR.

The OAMR free-space is defined in association with the current subgoal g, the current reference state, and the current local map of the environment $O(t_k)$. Consider the situation when a subgoal g has just become the currently pursued subgoal. Suppose the OAMR is currently at reference state p, v and beginning to pursue the subgoal g as shown in figure 4-1. The local map is indicated in the figure by the bold lines on the obstacles, which are the obstacle faces visible from the AMR position p. The subgoal selection algorithm guarantees that g is a safe and visible from p. We choose one of the free-space boundaries to be the line segment [pg]. The second boundary is determined by the ray emitting from p in the direction of the velocity vector v. These boundary choices are motivated by the following observation. With reference to figure 4-1, when the velocity vector v is pointing into the half plane to the right of the line pg, then the trajectory toward the subgoal should stay in the right half-plane determined

by pg, otherwise the resulting path would be longer than necessary. A similar argument can be made for the undesirability of the trajectory being to the right of the ray emitting from p in the direction of v.

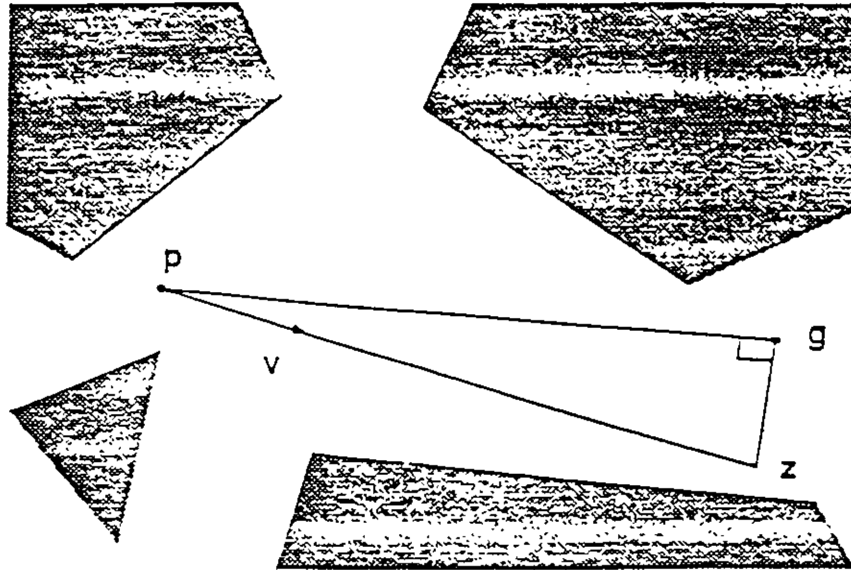


Figure 4-1: Construction of free-space for CAMR, case 1.

The last boundary is defined by a line segment gz where the point z is chosen such that gz ⊥ gp and zp is colinear with v. Choosing the boundary gz to be perpendicular to gp constrains the trajectory from overshooting the goal. For the case shown in figure 4-1, the free space is defined as the convex hull of three points p, g, and z denoted by C(p,g,z).

For the case shown in figure 4-2, the above definition would results in a free-space that intersects obstacle regions. In this case, to exclude the obstacle region from C(p,g,z), we rotate [gz around g towards p until $C(p,g,z) \cap O = \emptyset$. A free-space that has its vertex z to the right(left) of the subgoal g as viewed from the point p is called a right(left)-hand free-space, as is the case in figure 4-1(4-2).

Once an initial free-space is defined for a subgoal, the subsequent free-spaces for the subgoal are defined with respect to the first free-space as the CAMR moves toward the subgoal. Consider the situation shown in figure 4-3. The first free-space is defined for the subgoal g at t_n as $C(p(t_n),g,z(t_n))$. For the CAMR state $p(t_n)$, $v(t_n)$, the free-space for the subgoal g is defined as $C(p(t_1),g,z(t_n))$ where the point $z(t_n)$ is the intersection between the extension of the velocity vector $v(t_n)$ from $p(t_n)$ and the boundary $gz(t_n)$. Thus, the sequence of free-spaces for a subgoal all have a single common boundary [$gz(t_n)$] defined when the fir

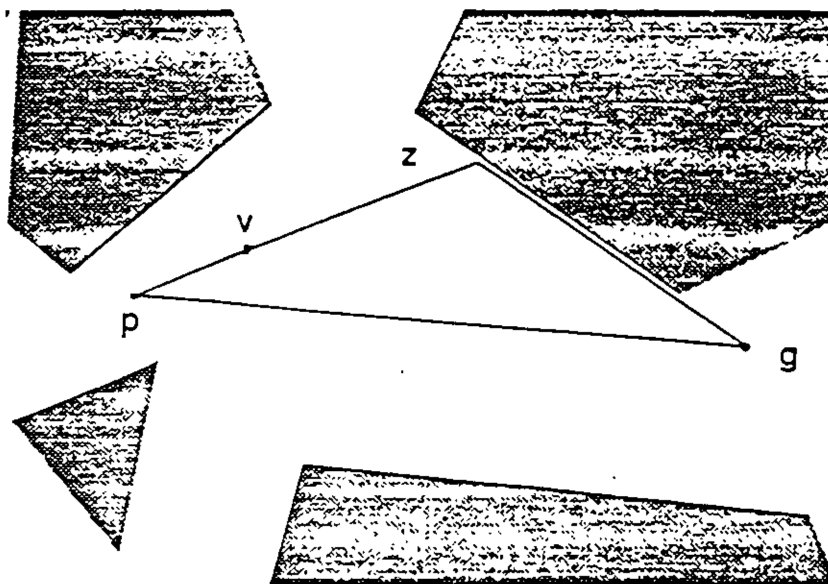


Figure 4-2: Construction of free-space for OAMR, case 2.

free-space for the subgoal is constructed. Each free-space may differ from the others in the other two boundaries which are determined by the subgoal and the OAMR state at the time when the free-space is constructed. Therefore, we can simplify the notation for the free-spaces for a particular subgoal to $C(t_n)$. For example, in figure 4-3, the free-space $C(p(t_n),g,z(t_n))$ is denoted by $C(t_n)$ and the free-space $C(p(t_{n+k}),g,z(t_{n+k}))$ is denoted by $C(t_{n+k})$.

The free-space can be represented analytically as a set of linear inequalities of the form

$$C(t_n) = \{q \mid [n_i(t_n)]^T q \leq d_i(t_n) \quad i=1,2,3\},$$

where the vectors $n_i(t_n)$ are normal to the boundaries $[gz]$, $[zp(t_n)]$, $[gp(t_n)]$, respectively, and the constants $d_i(t_n)$ determine the boundary lines in the plane.

4.3. OAMR Default Maneuver and Safe States

The default maneuver for the OAMR is defined by applying the maximum constant deceleration opposing the current velocity of the OAMR until the OAMR comes to a stop. This maneuver results in a straight line trajectory. For an OAMR with velocity $v(t_o)$, the time required to bring the OAMR to a stop using the default maneuver is:

$$t^* = \frac{\|v(t_o)\|}{a_{max}}. \tag{6}$$

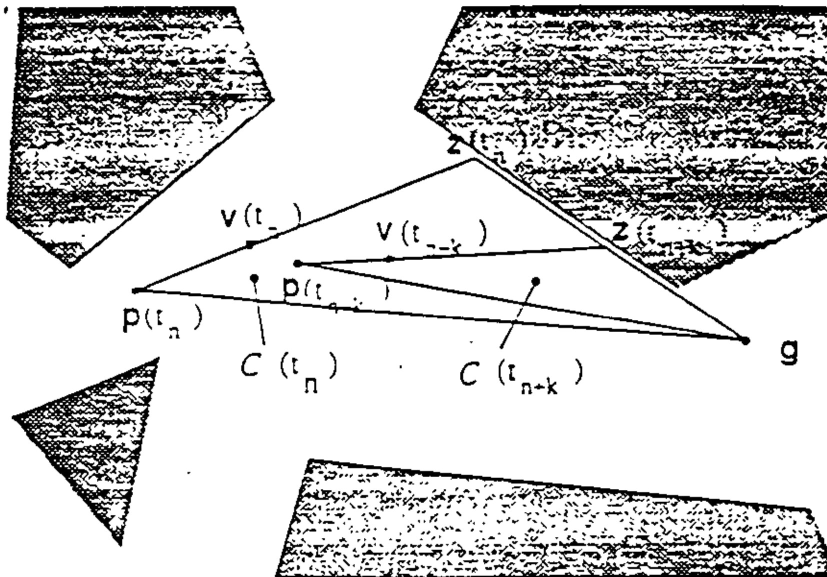


Figure 4-3: Construction of free-space for CAMR after the initial free-space.

The steering vector for the CAMR default maneuver is given by

$$a(t) = -\frac{a_{max}v(t_o)}{\|v(t_o)\|} \quad \text{for} \quad t_o \le t \le t^*.$$

We denote the position of the CAMR when it comes to a rest by $p_f^*(p(t_o),v(t_o))$, which is given by:

$$p_f^*(p(t_o),v(t_o)) = p(t_o) + \frac{\|v(t_o)\|}{a_{max}}v(t_o). \tag{7}$$

By the general definition of safe state given in section 2, an CAMR state in a free-space is safe if the CAMR can be brought to a stop along the default maneuver trajectory which lies entirely within the free space. Therefore, an CAMR state $v(t_o)$, $p(t_o)$ is a safe state in a free space $C(t')$ if

- $p(t_o) \in C(t')$ and
- $p_f^*(p(t_o),v(t_o)) \in C(t')$.

4.4. OAMR Environment Constraints

In this section, we present a method for mapping the constraints on the state of the OAMR due to the free-space boundaries into constraints on the steering vector at each time interval to produce the environmental constraint set U_e for the SDA. We first consider the mapping of the obstacle constraints at a time to when a new subgoal **g** is to be pursued.

Given that $\mathbf{p}(t_o)$, $\mathbf{v}(t_o)$ is safe in $C(t_o)$ (which is guaranteed by the SSA), we want to constraint the steering vector $\mathbf{a}(t_o)$ so that the state $\mathbf{p}(t_1)$, $\mathbf{v}(t_1)$ remains safe in $C(t_o)$. In other words, the steering vector $\mathbf{a}(t_o)$ is constrained so that the default maneuver trajectory starting from $\mathbf{p}(t_1)$, $\mathbf{v}(t_1)$ remains within the boundaries of $C(t_o)$. Using the definition of the safe state for $\mathbf{p}(t_1)$ and $\mathbf{v}(t_1)$, the free-space constraint can be expressed as

$$\mathbf{n}(t_o)_i^T \mathbf{p}_f^*(\mathbf{p}(t_1),\mathbf{v}(t_1)) \le d_i(t_o). \tag{8}$$

where $i = 1, 2$, and 3 for the first, second, and third boundaries of the free-space $C(t_o)$ respectively. By equations (3) and (7), the term $\mathbf{p}^*(t^*(\mathbf{p}(t_1),\mathbf{v}(t_1)),\mathbf{p}(t_1),\mathbf{v}(t_1))$ can be written in terms of $\mathbf{p}(t_o)$, $\mathbf{v}(t_o)$ and $\mathbf{a}(t_o)$ as:

$$\mathbf{p}_f^*(\mathbf{p}(t_1),\mathbf{v}(t_1))$$
$$= \mathbf{p}(t_o) + \Delta\mathbf{v}(t_o) + \frac{1}{2}\mathbf{a}(t_o)\Delta^2 + \frac{\|\mathbf{v}(t_o)+\Delta\mathbf{a}(t_o)\|}{a_{max}}(\mathbf{v}(t_o)+\Delta\mathbf{a}(t_o)).$$

Substituting this expression for $\mathbf{p}_f^*(\mathbf{p}(t_1),\mathbf{v}(t_1))$ in inequality (8) we obtain the following constraint on $\mathbf{a}(t_o)$:

$$\mathbf{n}_i(t_o)^T[\mathbf{p}(t_o) + \Delta\mathbf{v}(t_o) + \frac{1}{2}\mathbf{a}(t_o)\Delta^2 + \frac{\|\mathbf{v}(t_o)+\Delta\mathbf{a}(t_o)\|}{a_{max}}(\mathbf{v}(t_o)+\Delta\mathbf{a}(t_o))] \le d_i(t_o). \tag{9}$$

for $i = 1, 2$, and 3. These constraints on $\mathbf{a}(t_o)$ guarantee the trajectory remains in the free space for $t \in [t_o, t_1]$ and that the AMR is in a safe state at $t=t_1$.

Although the constraints on the state of the OAMR have been mapped into the steering vector space, the resulting constraints (9) are nonlinear. These constraints are too complicated for application in a real-time feedback scheme. Thus, we simplify these constraints to obtain linear constraints of the form

$$\mathbf{c}_i^T\mathbf{a}(t_o) \le \bar{a}_i.$$

This simplification is described as follows:

First, consider the case for $i = 1$ corresponding to the free-space boundary **gz**. This constraint is nonlinear due to the magnitude constraint on the acceleration $\mathbf{a}(t_n)$. By considering only the component of the OAMR velocity normal to the boundary, the nonlinear constraint can be approximated by a linear constraint. If we apply the constraint (9) only to the components of the

OAMR reference state and the steering vector that are perpendicular to the third boundary of the free-space $C(t_o)$, the constraint becomes a scalar inequality:

$$[\Delta a_1(t_n)]^2+(2\Delta v_1(t_n)+\Delta^2 a_{max})a_1(t_n)+(v_1(t_n))^2+2a_{max}(\Delta v_1(t_n)) \leq 2a_{max}d_1$$

where $a_1(t_o)$, $v_1(t_o)$, and $p_1(t_0)$ are the projections of $\mathbf{a}(t_o)$, $\mathbf{v}(t_o)$, and $\mathbf{p}(t_o)$ onto $\mathbf{n}_1(t_o)$. This quadratic inequality can be represented by the linear inequality

$$\mathbf{c}_1(t_o)^T\mathbf{a}(t_n) \leq \overline{a}_1(t_o) \tag{11}$$

where $\mathbf{c}_1(t_o)=\mathbf{n}_1(t_o)$ and

$$\overline{a}_1(t_o) = \frac{-(2+\Delta a_{max})-\sqrt{\Delta^2 a_{max}+\Delta v_1(t_o)a_{max}-3(\Delta v_1(t_o)-2d_1(t_o))a_{max}}}{2\Delta}$$

The linear approximations to the nonlinear constraint (9) are plotted for comparison in figure 4-4. We note that although the approximation is an outer approximation, it is very close to the original constraint in the vicinity of the system constraint set U_s (the circular acceleration limit), and can be compensated for in the steering vector selection procedure.

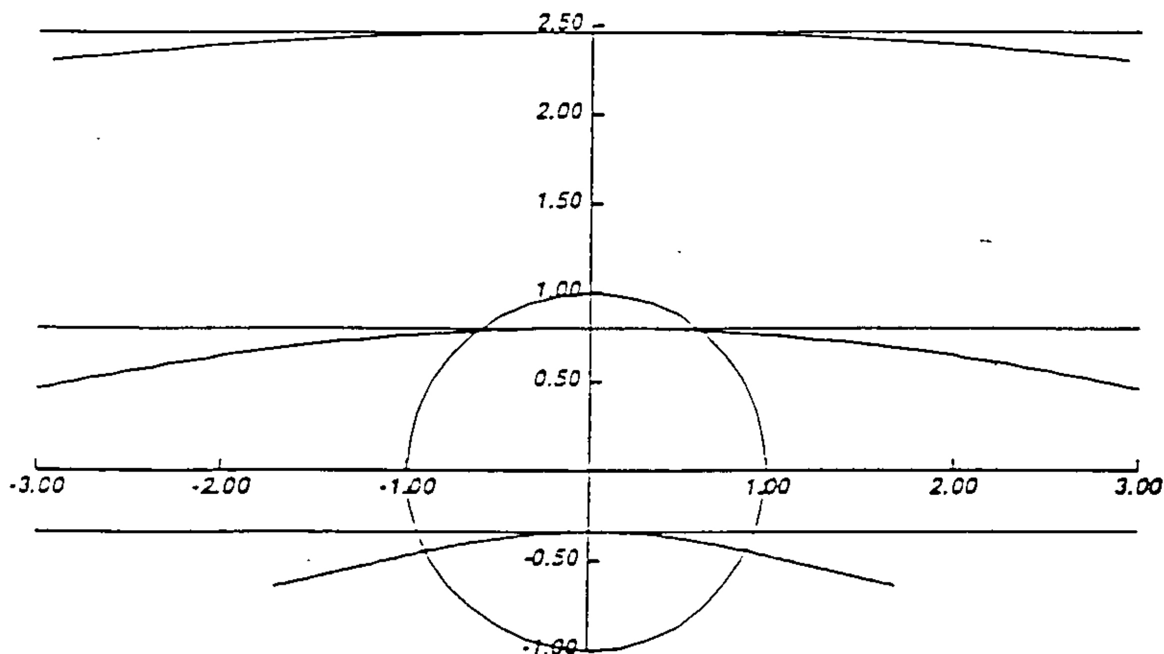


Figure 4-4: Constraints due to the third boundary of the free-space and their linear approximations.

From the geometry of the free-space, we note that the constraint (9) for $i = 2$ (corresponding to the free-space boundary [pz]) can be replaced by simple linear constraint. To illustrate the approach, we consider the right-handed free-space $C(t_o)$ shown in figure 4-5. The purpose of constraint (9) is to constrain $a(t_o)$ so that the default maneuver trajectory starting from the state $p(t_1)$, $v(t_1)$ remains in the left half plane defined by the line $p(t_o)z$. Since the default maneuver at time t_1 would generate a trajectory that is colinear with $v(t_1)$, constraining the velocity vector $v(t_1)$ to point into the left-half plane, away from the line $zp(t_o)$, has the same effect as constraint (9). Thus, we can replace constraint 9 for $i = 2$ by

$$c_2(t_o)^T v(t_1) = c_2(t_o)^T a(t_o) \leq 0 \tag{12}$$

where $c_2(t_o) = n_2(t_o)$. This constraint is consistent with the goal to reach the subgoal g since in order to reach the goal we must effect a change in the direction of the velocity vector so that it rotates toward the direction of the subgoal, as is required by the constraint (12).

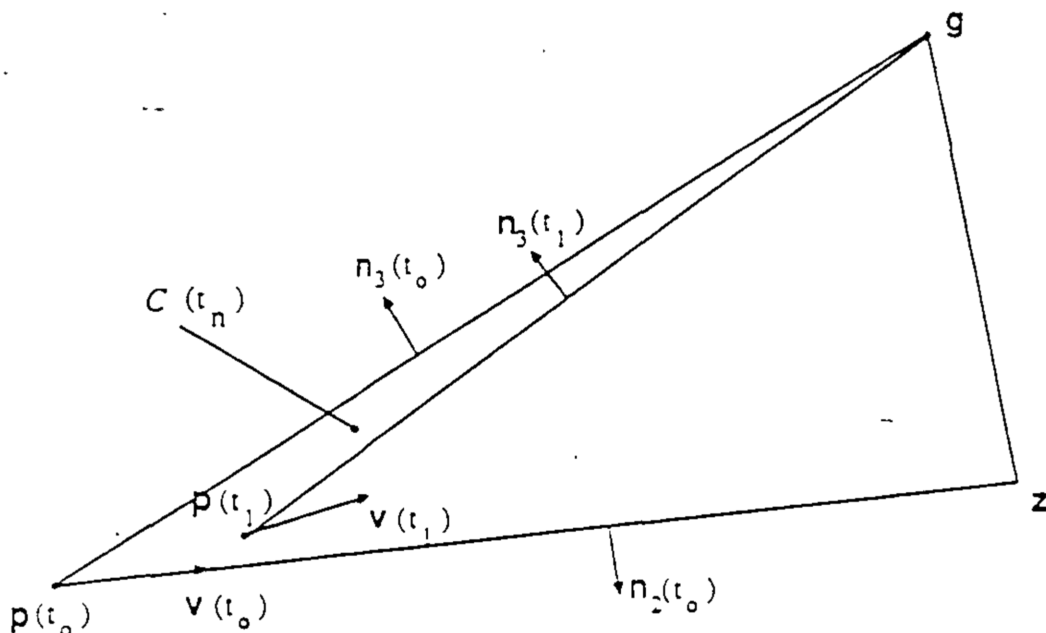


Figure 4-5: Simplification of the constraints due to free-space boundary.

By geometrical observation, we note that the constraint 9 for $i = 3$ (corresponding to boundary [pg]) can be replaced by linear constraint similarly as in the case of $i = 2$. Again without loss of generality, we consider the situation show in figure 4-5. The constraint (8) for $i = 3$ is designed to prohibit the default maneuver trajectory starting from the state $p(t_1)$, $v(t_1)$ from crossing the first boundary of the free-space $C(t_o)$, that is, the line $[gp(t_o)]$. The same effect can be achieved if we constraint the velocity vector $v(t_1)$ to point to the right of the line $[gp(t_1)]$ as shown in figure

4-5. This would ensure that the default maneuver starting from $p(t_1)$, $v(t_1)$ be unable to exit the free-space through boundary $[\bar{g}p(t_o)]$. The constraint on the velocity vector $v(t_1)$ can expressed as the inequality:

$$[^l(g-p(t_1))]^T v(t_1) \le 0. \tag{13}$$

where $^l(g-p(t_1))$ is the vector $(g-p(t_1))$ rotated 90 degrees to the left. In fact, $^l(g-p(t_1)) = \|(g-p(t_1))\| n_3(t_1)$, where $n_3(t_1)$ is the unit vector that defines the orientation of the first boundary of the free-space $C(t_1)$. Therefore, the inequality (13) can be written as:

$$c_3(t_o)^T a(t_o) \le \bar{a}_3 \tag{14}$$

where $c_3(t_o) = g-p(t_o)+0.5\delta v(t_o)$ and $\bar{a}_3 = -\frac{(g-p(t_o))^T v(t_o)}{\Delta \|g-p(t_o)-0.5\Delta v(t_o)\|}$. This is a linear constraint on $a(t_o)$ that is parallel to the vector $g-p(t_o)+0.5\Delta v(t_o)$.

Thus, we have transformed the constraints due to the obstacles, as represented by the free-space, into constraints on the steering vector **a** given by the environmental constraint set $U_e = \{a \mid c_i^T a \le \bar{a}_i \text{ for } i=1,2,3\}$. We note that the three boundaries of the set U_e are completely determined by the state of the OAMR, the current subgoal and the third boundary of the free-space.

4.5. Objective Vector and Steering Vector Selection

Having transformed the obstacle constraints into the environment constraint set U_e and represented the system constraint by the set U_s, the acceleration vector $a(t_o)$ for generating the reference trajectory over the time interval $[t_o, t_1]$ can be computed by choosing an acceleration vector $a(t_o)$ that approximates the objective vector $d^*(t_o)$ subject to the constraint $a(t_o) \in U_e \cap U_s$. The objective vector is to produce a "good" reference trajectory to the subgoal. It is desirable that the objective vector produces a sequence of steering vectors leading to a direct and smooth trajectory to the goal. To this end, we construct the objective vector from two components, based on a generalized potential field, as

$$d^*(t_n) = d_a(g^*, v(t_n), p(t_n)) + \gamma d_r(C, v(t_n), p(t_n)) \tag{15}$$

where d_a is an attractive potential gradient determined from the goal, and d_r is the repulsive potential gradient caused by generalized potential fields assigned to the free space boundaries [16].

The attractive potential gradient vector towards the subgoal is constructed heuristically based on the time-optimal control solution to the double integrator dynamics in one dimensional space [15]. Give the subgoal g, and the OAMR reference state $p(t_o)$, $v(t_o)$ the attraction vector d_a is defined to be:

$$d_a(g^*, p(t_o), v(t_o)) = \tau_1 n_1 + \tau_2 n_2 \tag{16}$$

where n_1 and n_2 are unit vectors that are normal and colinear, respectively, to $[gp(t_n)]$ subject to:

$$n_1^T v(t_o) \leq 0, \text{ and}$$

$$n_2^T [g - p(t_o)] \leq 0$$

The gains t_i, $i=1,2$, are the minimum times required to drive the one-dimensional double integrator from an initial state $n_i^T p(t_o)$, $n_i^T v(t_o)$ to the final state $n_i^T g, n_i^T v$ with maximum acceleration magnitude of a_{max}. Thus the gains τ_1 and τ_2 reflect the urgency to accelerate in each direction in order to bring the AMR to rest at the subgoal.

The repulsive potential gradient $d_r(C, v(t_o), p(t_o))$ is the sum of repulsive potential gradients from each of the three boundaries of the free space $C(t_o)$. A repulsive accelerating vector is normal to the boundary which generates it. Assuming the geometric relationship between the AMR reference state $p(t_n)$, $v(t_n)$ and the *ith* boundary of the $C(t_n)$ at a distance of d_i away from the position p, the repulsive potential gradient for the *ith* boundary is given by:

$$d_{ri} = -\frac{n_i}{\tau_M - \tau_m} \tag{17}$$

where n_i is a unit vector normal to the *ith* boundary and is pointed away from the free space. The parameter τ_m is the time it takes to reduce the speed of the AMR towards the *ith* boundary to zero with maximum deceleration, given by

$$\tau_m = \frac{v(t_o)^T n_i}{a_{max}}. \tag{18}$$

The parameter τ_M is the time it takes to reduce the speed to zero just before hitting the boundary with a constant deceleration, given by

$$\tau_M = \frac{2d_i}{[v(t_o)]^T n_i}. \tag{19}$$

Substituting τ_m and τ_M in equation (17) with expressions (18) and (19) and summing over i from 1 to 3, we obtain the repulsive potential gradient for the free-space boundaries as,

$$d_r = \sum_{i=1}^{3} \frac{a_{max}(n_i^T v(t_o)) n_i}{2 d_i a_{max} - (n_i^T v(t_o))^2}.$$

The parameter γ in equation (15) is used to balance the effects of the attractive and the repulsive vectors on the objective vector. Simulation experiments show that 0.1 is a reasonable

value for γ.

The final stage of the SDA is to select a steering vector to approximated the objective ve subject to the constraints posted by U_e and U_s. This problem is illustrated in figure 4-6. ... system constraint set U_s is show as the circle. The two linear boundaries (due to the free-space boundaries 1 and 3) of the environmental constraint set U_e are shown. The other boundary is located very far from the origin of the steering vector space in this case, and is not shown in the figure. The task is to select a steering vector $a(t_o)$ that best approximates the objective vector $d^*(t_o)$. This problem can be formulated as an optimization problem by defining a objective

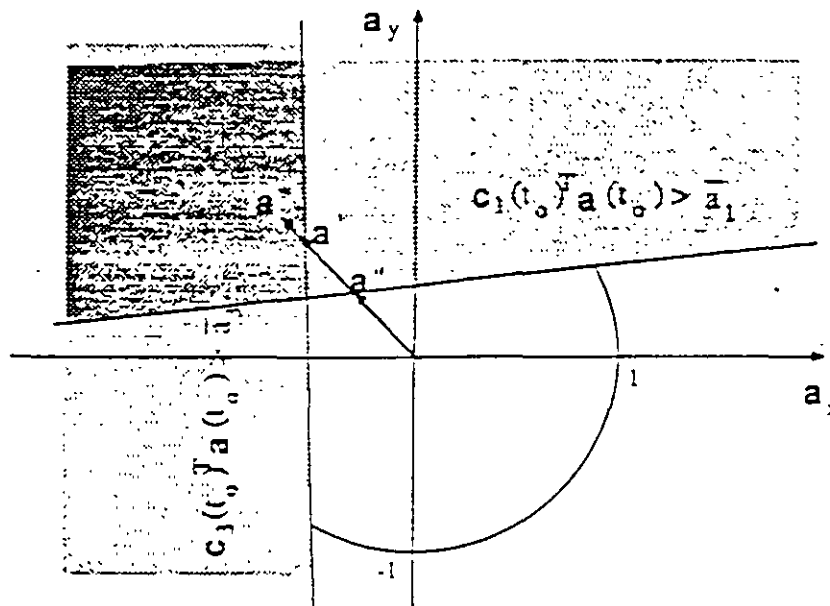


Figure 4-6: Solving the steering vector.

function and solved using standard numerical procedures, However, because of the nonlinear constraint U_s, such a formulation does not lead to a simple solution procedure. On the other hand, the special structure of the constraints can be exploited to allow a reasonable steering vector to be selected with minimal computation burden using the hueristic procedure presented in figure 4-7.

The computation requirement for this procedure is very small. It needs to test at most 3 linear inequalities and solve at most 2 linear equations (represented by the procedure LINESOLVE). The early activation of the default maneuver at line 1 in figure 4-7 compensates for the outer approximation of the constraint due third boundary of the free-space. As shown by the simulation examples in section 6, the default maneuver is not normally invoked.

```
1   if  (ā₁ < 0)
        a = -a_max v(t_n)/||v(t_n)||
    else
2       a* = a_max d*;
3       if  (c₃ᵀa* ≤ ā₃)
4           a(t_n)=LINESOLVE(⊥a*,0,c₃,ā₃)
5           if  (c₁ᵀa(t_n) ≤ ā₁)
                a(t_n)=LINESOLVE(c(t_n),ā₁,⊥a(t_n),0)
6       else if ( c₂ᵀa(t_n) ≤ ā₂)
            a(t_n)=LINESOLVE(⊥a*,0,c₂,ā₂)
            if  (c₁ᵀa(t_n) ≤ ā₁)
                a(t_n)=LINESOLVE(c(t_n),ā₁,⊥a(t_n),0)
7       else
            a(t_n)=a*
            if  (c₁ᵀa(t_n) ≤ ā₁)
                a(t_n)=LINESOLVE(c(t_n),ā₁,⊥a(t_n),0)
```

Figure 4-7: Procedure for selecting the steering vector subject to the environmental and system constraints

References

30. D. Feng, et al, "The Servo-Control System for an Omnidirectional Mobile Robot", *IEEE International Conf on Robotics and Automation*, 1989.

33. B. Carlisle, *Developments in Robotics*, IFS Publishing Ltd., Kempston, Bedfordshire, England, 1983, ch. Omni-Directional Mobile Robot.

34. H.P. Moravec, "Autonomous Mobile Robots Annual Report - 1985", Robotics Institute Technical Report CMU-RI-MRL 86-1, Carnegie Mellon University, Jan. 1986.

References

1. Y. Kanayama and S.'I. Yuta, "Vehicle Path Specification by a Sequence of Straight Lines," IEEE J. Robotics and Automation **4**, 265-276 (1988).

2. V. J. Lumelsky and A.A. Stepanov, "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment," IEEE Trans. Automatic Control **31**, 1058-1063 (1986).

3. V. Lumelsky, "Algorithmic Issues of Sensor-Based Robot Motion Planning," IEEE Trans. Decision and Control, 1796-1801 (1987).

4. V.J. Lumelsky and A.A. Stepanov, "Path-Planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shape," Algorithmica **2**, 403-430 (1987).

5. B.H. Krogh and D. Feng, "Dynamic Generation of Subgoals for Autonomous Mobile Robots Using Local Feedback Information," IEEE Trans. Automatic Control **34**, 483-493 (1989).

6. D. Feng and B.H. Krogh, "Satisficing Feedback Strategies for Local Navigation of Autonomous Mobile Robots," IEEE Trans. Systems, Man and Cybernetics (to be published).

7. D. Feng, "Satisficing Feedback Strategies for Local Navigation of Autonomous Mobile Robots," Ph.D. thesis, Carnegie Mellon University (1989).

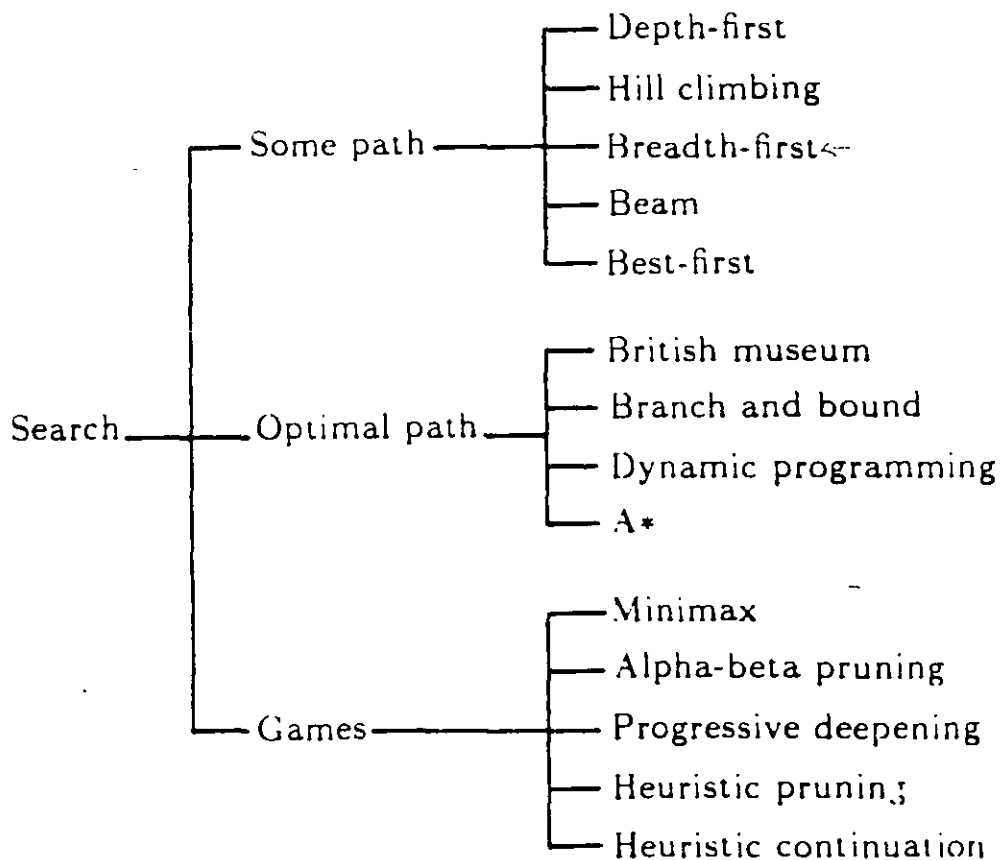
SEARCHING FOR PATHS AND OTHER THINGS

ARTIFICIAL INTELLIGENCE

P. H. WINSTON

ONE MAY SEARCH FOR:

1) SOME PATH
2) OPTIMAL PATH
3) WAYS TO WIN THE GAME



Search procedures. Many procedures address the problem of finding satisfactory paths. Others concentrate on the harder problem of finding optimal paths. Procedures for games differ from ordinary path-finding procedures, because games involve adversaries.

THE BASIC SEARCH PROBLEM:

GIVENS:

1) THE STARTING POINT (NODE)

2) THE GOAL POINT (NODE)

3) A MAP OF NODES AND CONNECTIONS

GOALS:

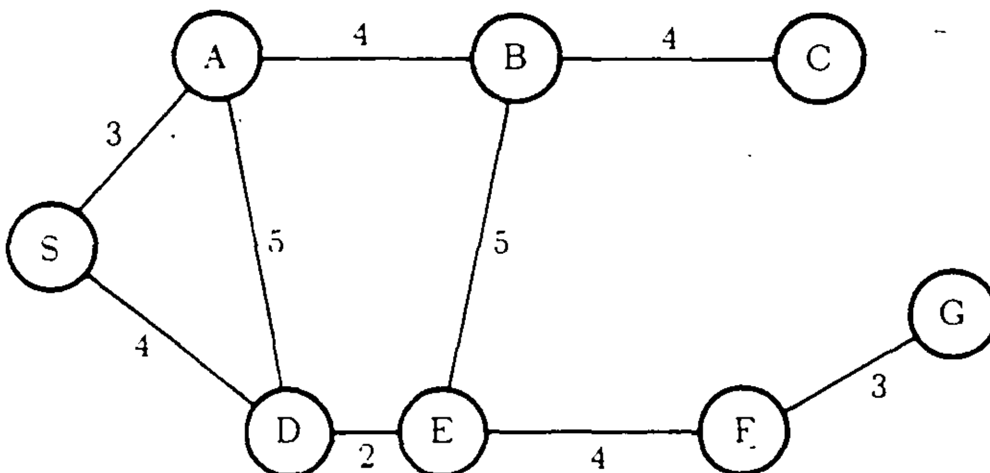
1) FIND SOME PATH OR FIND THE "BEST" PATH [MAYBE SHORTEST]

SOME PATH MAY BE THE CORRECT PROCEDURE IF THE TERRAIN IS TRAVERSED ONLY SELDOM AND THE JOB OF FINDING THE BEST PATH IS DIFFICULT.

BEST PATH MAY BE THE CORRECT PROCEDURE IF THE TERRAIN IS TRAVERSED OFTEN.

2) TRAVERSE THE PATH

A COLLECTION OF NODES AND LINKS AS SHOWN BELOW IS CALLED A "NET".



A basic search problem. A path is to be found from the start node. S. to the goal node. G. Search procedures explore nets like these. learning about connections and distances as they go.

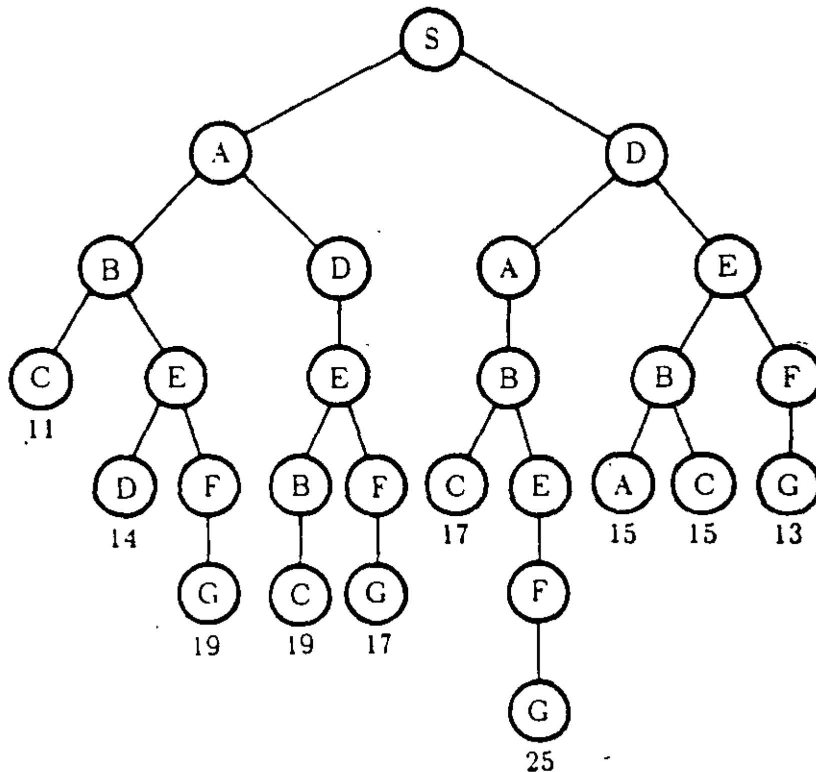
A COLLECTION OF NODES (MAY BE REPEATING) AND BRANCHES IS CALLED A "TREE". THE TREE BELOW IS FORMED FROM THE PREVIOUS NET.

NODES CLOSER TO THE TOP OF THE TREE ARE CALLED "PARENTS".

THE NODE AT THE TOP WITH NO PARENT IS CALLED THE "ROOT NODE".

THE NODES AT THE BOTTOM WITH NO CHILDREN ARE CALLED "TERMINAL NODES".

THE TREE BELOW HAS EIGHT "LEVELS".



A tree made from a net. Nets are made into trees by tracing out all possible paths to the point where they reenter previously visited nodes. Node S is the root node. Node S is also a parent node, with its children being A and D, and an ancestor node, with all other nodes in the tree being descendants. The nodes with no children are the terminal nodes. The numbers beside the terminal nodes are accumulated distances.

DEPTH FIRST SEARCH:

1) FORM A ONE-ELEMENT QUEUE CONSISTING OF THE ROOT NODE

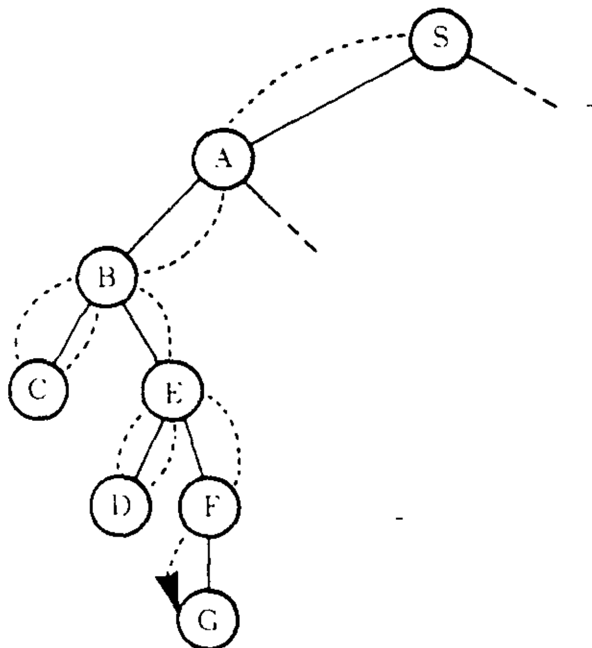
2) UNTIL THE QUEUE IS EMPTY OR THE GOAL IS REACHED, DETERMINE IF THE FIRST ELEMENT IN THE QUEUE IS THE GOAL NODE.

2a) IF THE FIRST ELEMENT IS THE GOAL NODE, DO NOTHING

2b) IF THE FIRST ELEMENT IS NOT THE GOAL NODE, REMOVE THE FIRST ELEMENT FROM THE QUEUE AND ADD THE FIRST ELEMENT'S CHILDREN, IF ANY , TO THE *FRONT* OF THE QUEUE

3) IF THE GOAL NODE IS FOUND ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

SEE EXAMPLE BELOW:



An example of depth-first search. One alternative is selected and pursued at each node until the goal is reached or a node is reached where further downward motion is impossible. When further downward motion is impossible, search is restarted at the nearest ancestor node with unexplored children.

HILL CLIMBING:

1) FORM A ONE-ELEMENT QUEUE CONSISTING OF THE ROOT NODE

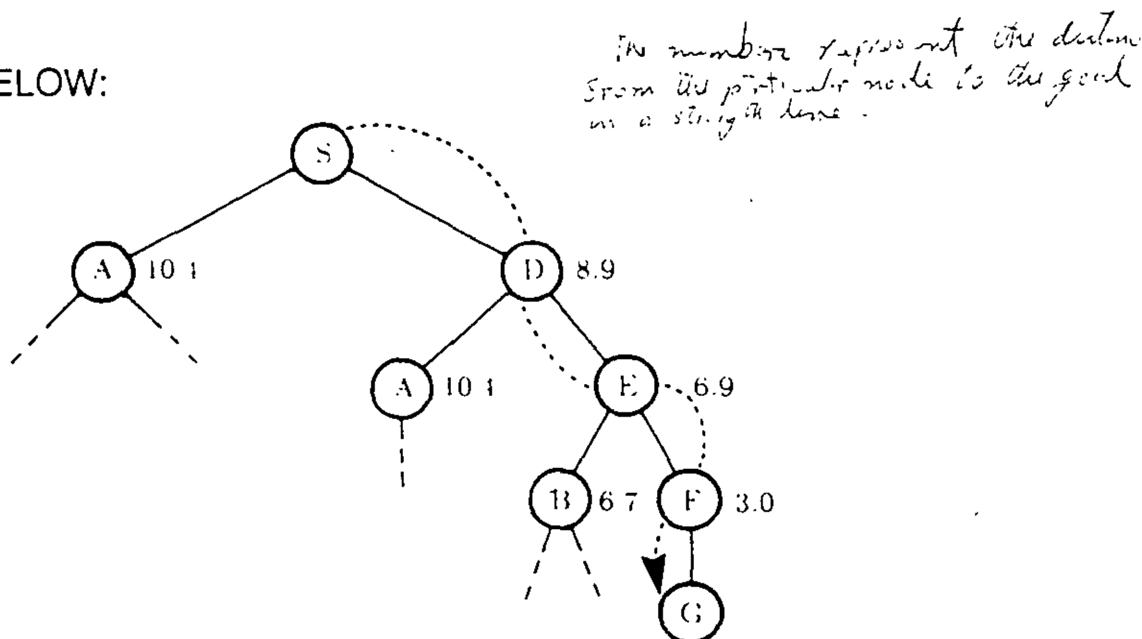
2) UNTIL THE QUEUE IS EMPTY OR THE GOAL IS REACHED, DETERMINE IF THE FIRST ELEMENT IN THE QUEUE IS THE GOAL NODE.

2a) IF THE FIRST ELEMENT IS THE GOAL NODE, DO NOTHING

2b) IF THE FIRST ELEMENT IS NOT THE GOAL NODE, REMOVE THE FIRST ELEMENT FROM THE QUEUE, *SORT THE FIRST ELEMENT'S CHILDREN, IF ANY, BY ESTIMATED REMAINING DISTANCE*, AND ADD THE FIRST ELEMENT'S CHILDREN, IF ANY, TO THE *FRONT* OF THE QUEUE

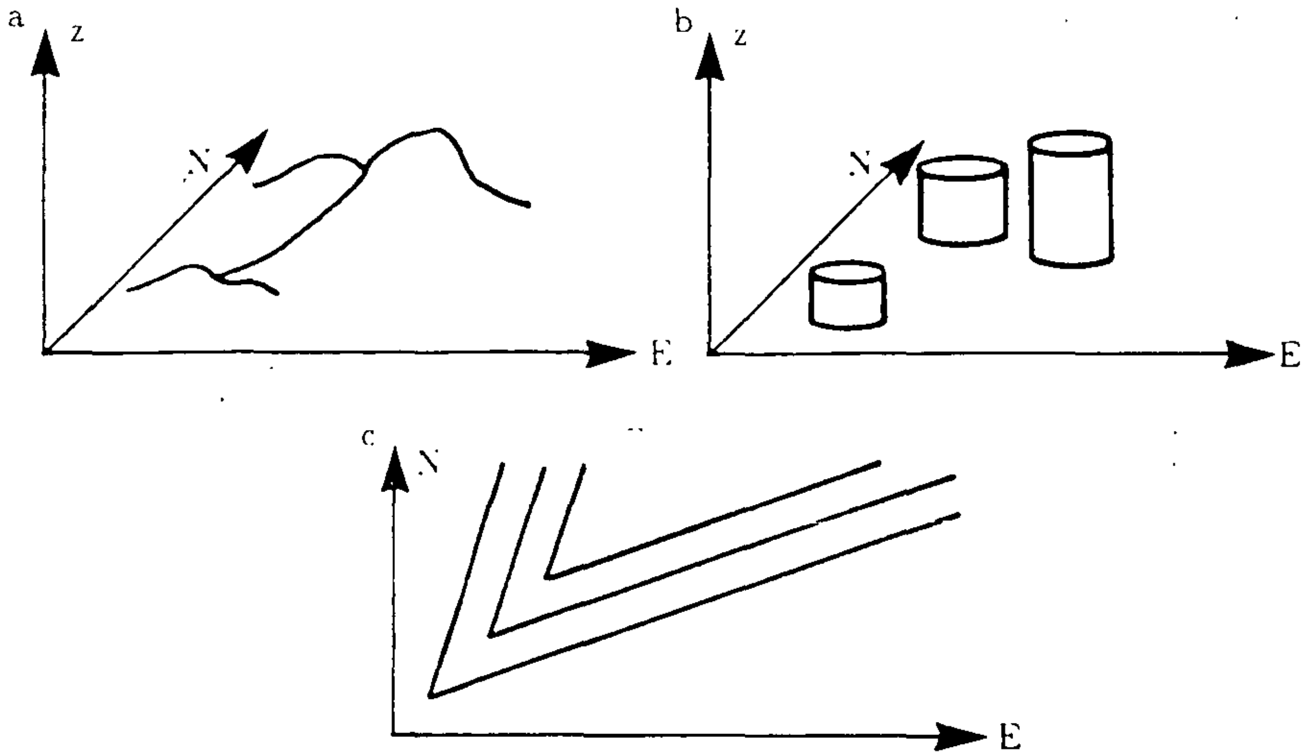
3) IF THE GOAL NODE IS FOUND ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

SEE EXAMPLE BELOW:



An example of hill climbing. Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded. The numbers beside the nodes are straight-line distances to the goal node.

PROBLEMS WITH HILL CLIMBING:



Hill climbing is a bad idea in difficult terrain. In *a*, foothills stop progress. In *b*, plains cause aimless wandering. In *c*, with the terrain described by a contour map, all ridge points look like peaks because all four east-west and north-south probe directions lead to lower quality measurements.

BREADTH FIRST SEARCH:

1) FORM A ONE-ELEMENT QUEUE CONSISTING OF THE ROOT NODE

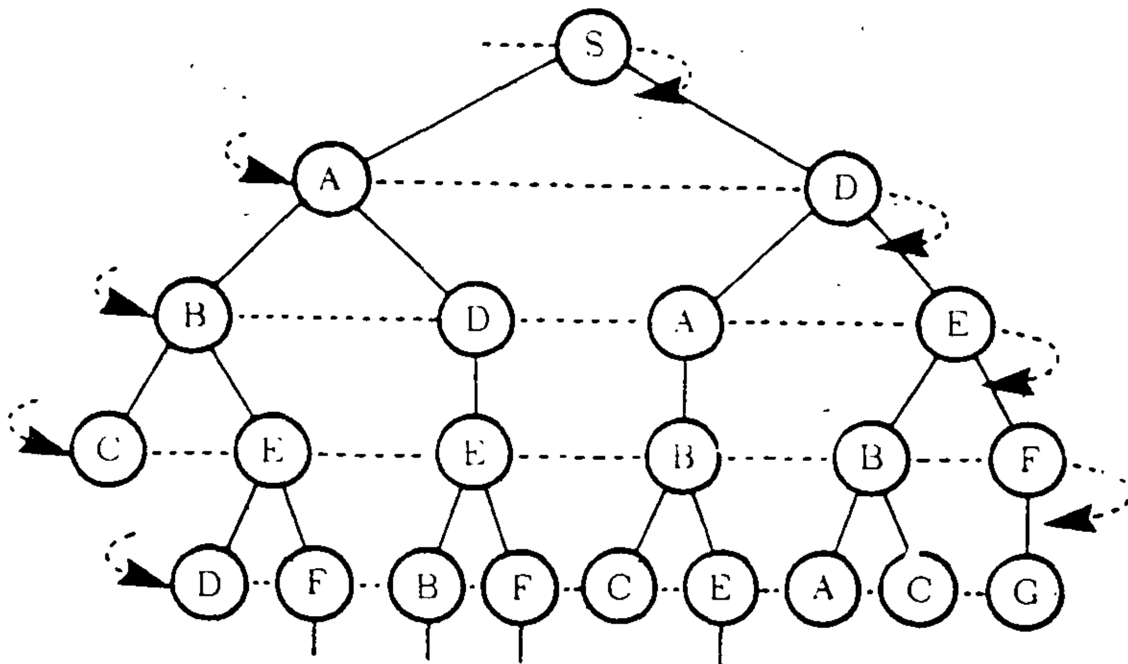
2) UNTIL THE QUEUE IS EMPTY OR THE GOAL IS REACHED, DETERMINE IF THE FIRST ELEMENT IN THE QUEUE IS THE GOAL NODE.

2a) IF THE FIRST ELEMENT IS THE GOAL NODE, DO NOTHING

2b) IF THE FIRST ELEMENT IS NOT THE GOAL NODE, REMOVE THE FIRST ELEMENT FROM THE QUEUE AND ADD THE FIRST ELEMENT'S CHILDREN, IF ANY, *TO THE BACK OF THE QUEUE.*

3) IF THE GOAL NODE IS FOUND ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

SEE EXAMPLE BELOW:



An example of breadth-first search. Downward motion proceeds level by level until the goal is reached

BEAM SEARCH:

LIKE BREADTH FIRST BUT SEARCH ONLY MOVES DOWNWARD FROM THE BEST *W* NODES AT EACH LEVEL WHERE *W* IS CHOSEN EXTERNAL TO THE SEARCH ALGORITHM

SEE EXAMPLE BELOW:

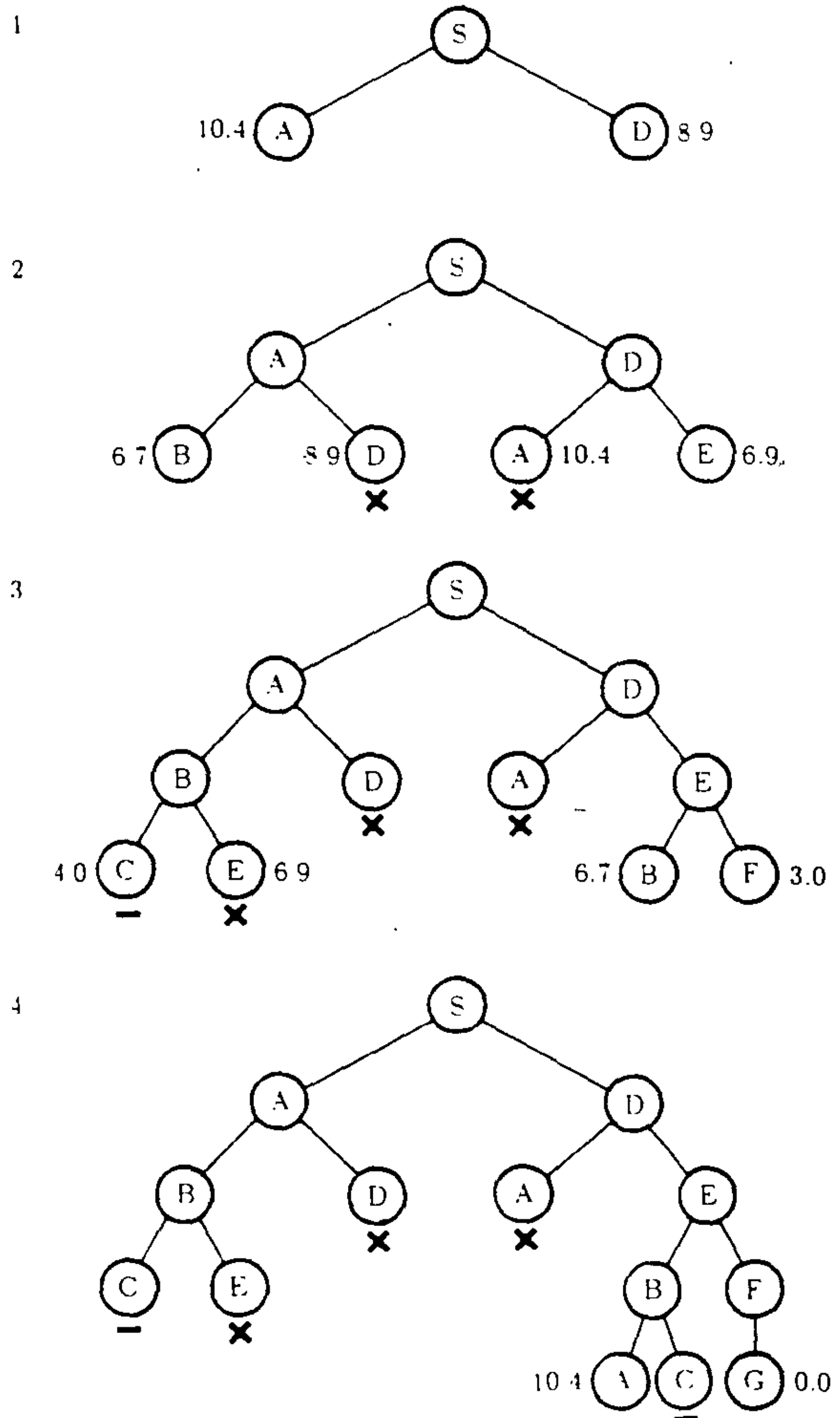


Figure 4-8. An example of beam search. Investigation spreads through the search tree level by level, but only the best w nodes are expanded, where $w = 2$ here. The numbers beside the nodes are straight-line distances to the goal node.

BEST FIRST SEARCH:

1) FORM A ONE-ELEMENT QUEUE CONSISTING OF THE ROOT NODE

2) UNTIL THE QUEUE IS EMPTY OR THE GOAL IS REACHED, DETERMINE IF THE FIRST ELEMENT IN THE QUEUE IS THE GOAL NODE.

2a) IF THE FIRST ELEMENT IS THE GOAL NODE, DO NOTHING

2b) IF THE FIRST ELEMENT IS NOT THE GOAL NODE, REMOVE THE FIRST ELEMENT FROM THE QUEUE AND ADD THE FIRST ELEMENT'S CHILDREN, IF ANY, *TO THE QUEUE, AND SORT THE ENTIRE QUEUE BY ESTIMATED REMAINING DISTANCE.*

3) IF THE GOAL NODE IS FOUND ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

FIND THE SHORTEST PATH:

1) EXHAUSTIVE SEARCH (EITHER DEPTH FIRST OR BREADTH FIRST)

2) BRANCH AND BOUND SEARCH (EXPAND THE LEAST COST PARTIAL PATH)

BRANCH AND BOUND SEARCH:

1) FORM A QUEUE OF PARTIAL PATHS. LET THE INITIAL QUEUE CONSIST OF THE ZERO-LENGTH, ZERO-STEP PATH FROM THE ROOT NODE TO NOWHERE.

2) UNTIL THE QUEUE IS EMPTY OR THE GOAL HAS BEEN REACHED, DETERMINE IF THE FIRST PATH IN THE QUEUE REACHES THE GOAL NODE.

2a) IF THE FIRST PATH REACHES THE GOAL NODE, DO NOTHING.

2b) IF THE FIRST PATH DOES NOT REACH THE GOAL NODE:

2b1) REMOVE THE FIRST PATH FROM THE QUEUE

2b2) FORM NEW PATHS FROM THE REMOVED PATH BY EXTENDING ONE STEP

2b3) ADD THE NEW PATHS TO THE QUEUE

2b4) SORT THE QUEUE BY COST ACCUMULATED SO FAR, WITH LEAST-COST PATHS IN FRONT

3) IF THE GOAL NODE HAS BEEN FOUND, ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

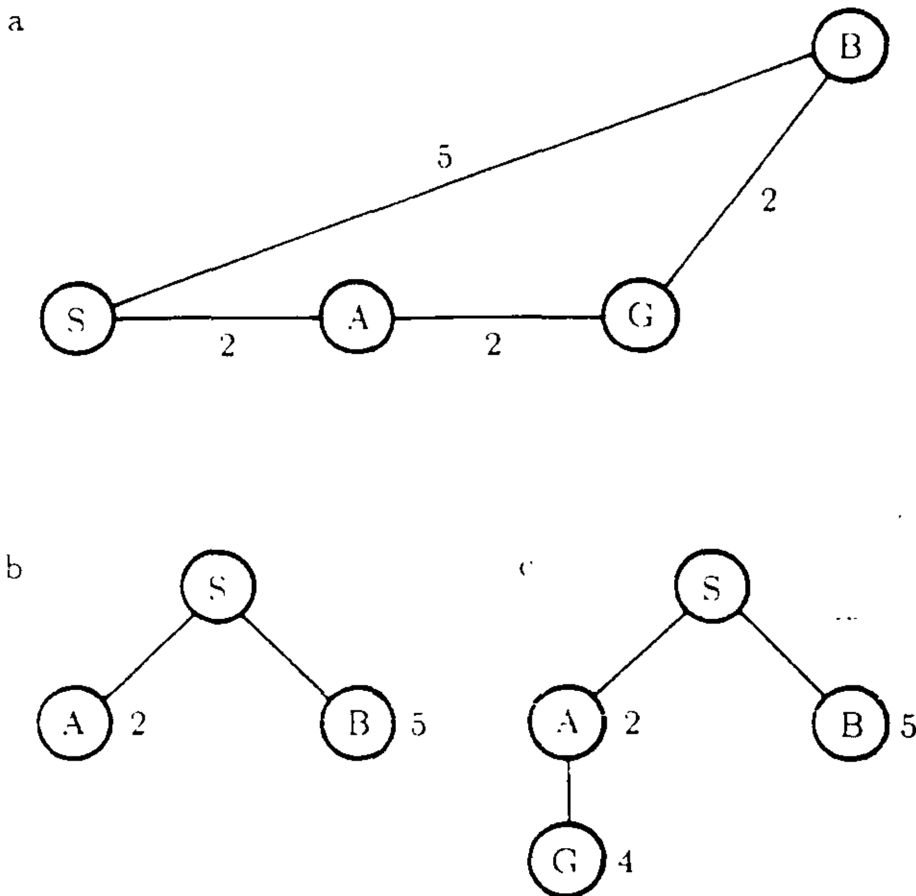


Figure 4-10. In branch-and-bound search, the node expanded is the one at the end of the shortest path leading to an open node. Expansion continues until there is a path reaching the goal that is of length equal to or shorter than all incomplete paths terminating at open nodes. A sample net is shown in *a*, along with partially developed search trees in *b* and *c*. The numbers beneath the nodes in the trees are accumulated distances. In *b*, node A might just as well be expanded, for even if a satisfactory path through B is found, there may be a shorter one through A. In *c*, however, it makes no sense to expand node B, because there is a complete path to the goal that is shorter than the path ending at B.

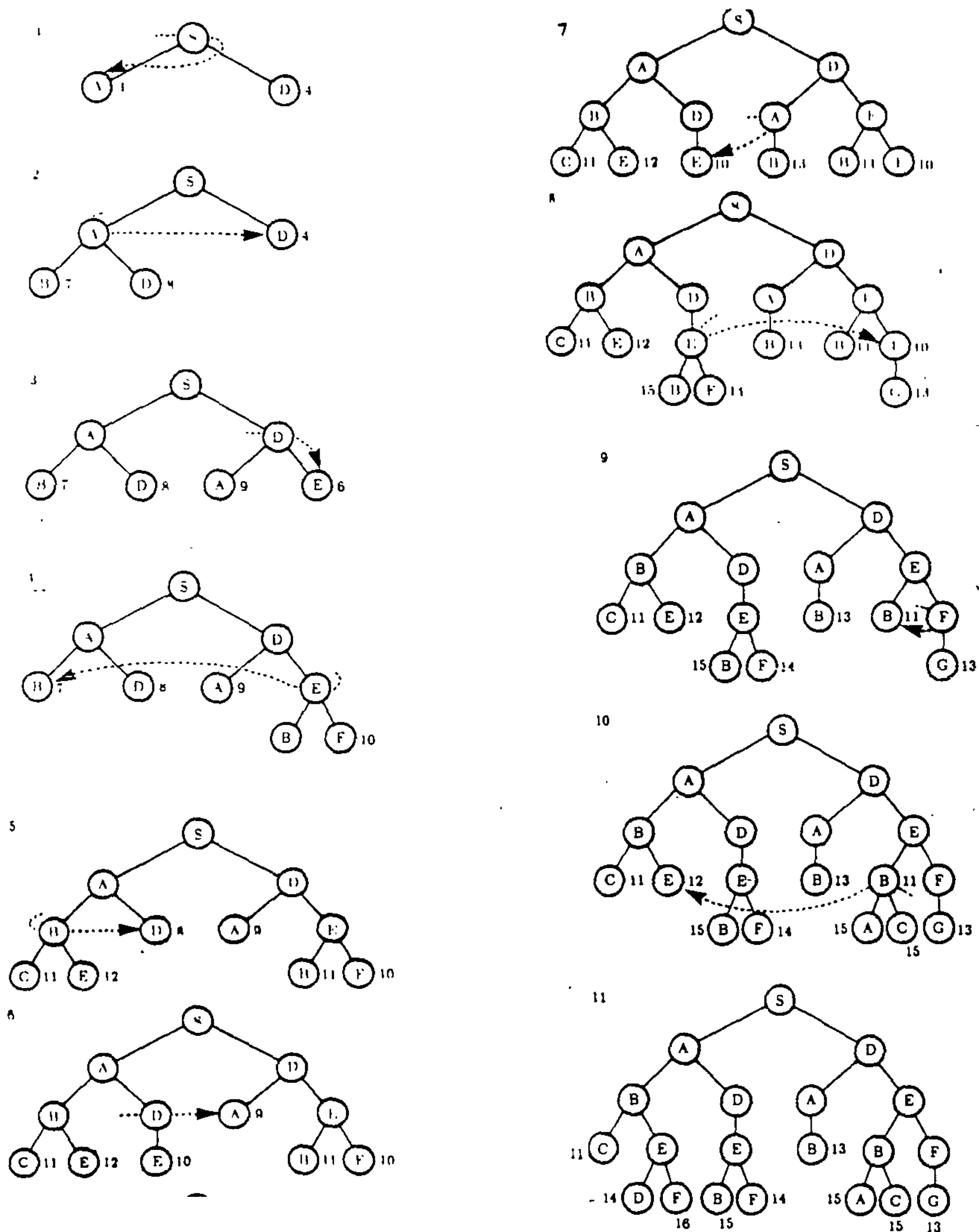


Figure 4-11. Branch-and-bound search determines that path S–D–E–F–G is optimal. The numbers beside the nodes are accumulated distances. Search stops when all partial paths to open nodes are as long as or longer than the complete path S–D–E–F–G.

BRANCH AND BOUND WITH UNDERESTIMATES:

1) FORM A QUEUE OF PARTIAL PATHS. LET THE INITIAL QUEUE CONSIST OF THE ZERO-LENGTH, ZERO-STEP PATH FROM THE ROOT NODE TO NOWHERE.

2) UNTIL THE QUEUE IS EMPTY OR THE GOAL HAS BEEN REACHED, DETERMINE IF THE FIRST PATH IN THE QUEUE REACHES THE GOAL NODE.

2a) IF THE FIRST PATH REACHES THE GOAL NODE, DO NOTHING.

2b) IF THE FIRST PATH DOES NOT REACH THE GOAL NODE:

2b1) REMOVE THE FIRST PATH FROM THE QUEUE

2b2) FORM NEW PATHS FROM THE REMOVED PATH BY EXTENDING ONE STEP

2b3) ADD THE NEW PATHS TO THE QUEUE

2b4) SORT THE QUEUE BY THE *SUM OF COST* ACCUMULATED SO FAR *AND A LOWER-BOUND ESTIMATE OF THE COST REMAINING,* WITH LEAST-COST PATHS IN FRONT

3) IF THE GOAL NODE HAS BEEN FOUND, ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

SEE EXAMPLE FOLLOWING:

DEFINE:

e(total path length) = ESTIMATE OF TOTAL PATH LENGTH

d(already traveled) = KNOWN DISTANCE ALREADY TRAVELED

e(distance remaining) = ESTIMATE OF DISTANCE REMAINING

e(total path length) = d(already traveled) + e(distance remaining)

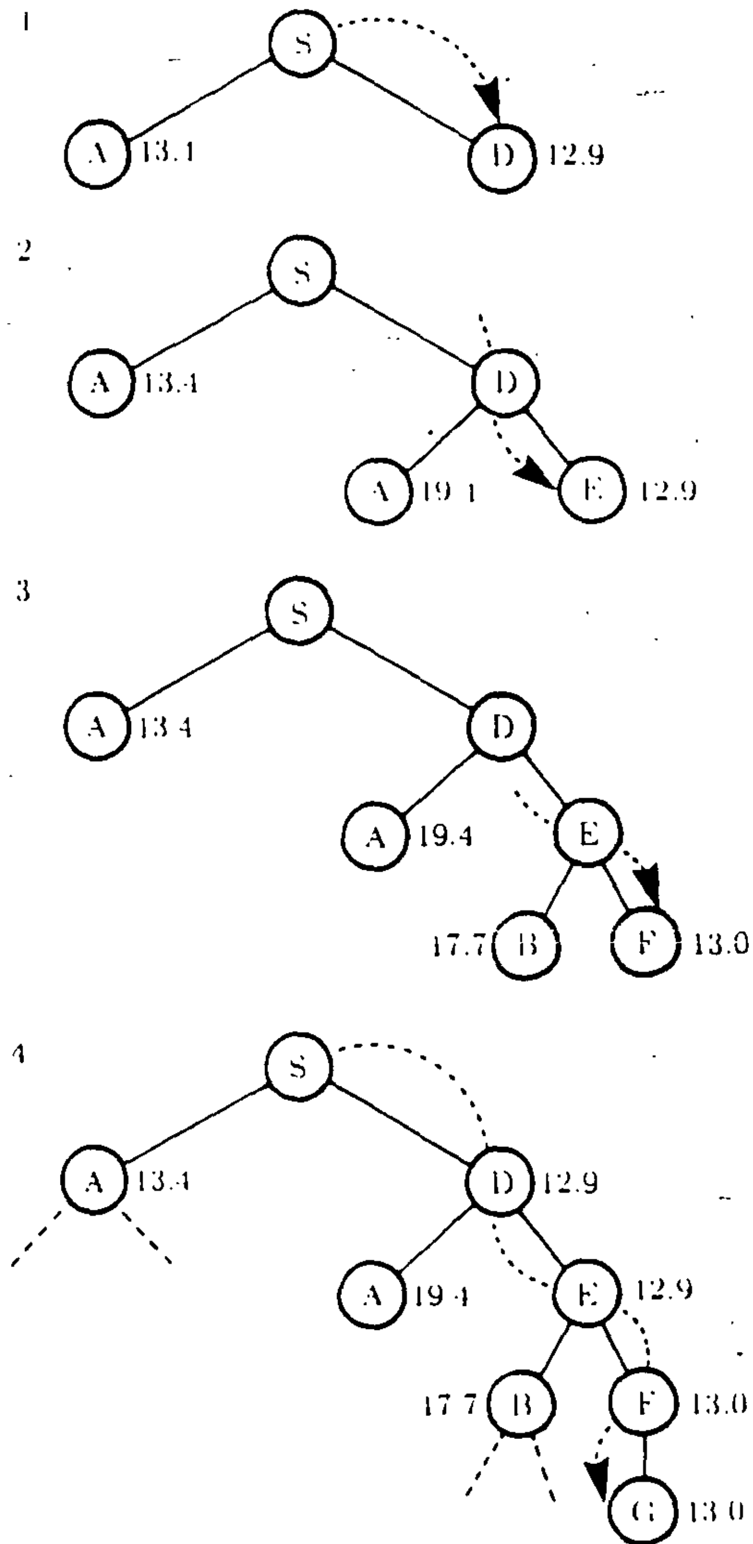
AN OVERESTIMATE OF e(distance remaining) CAN BE VERY BAD!

AN UNDERESTIMATE CAN NOT CAUSE THE CORRECT PATH TO BE OVERLOOKED.

DEFINE u( ) TO BE AN UNDERESTIMATE OF THE ARGUMENT

u(total path length) = d(already traveled) + u(distance remaining)

SEE EXAMPLE FOLLOWING:

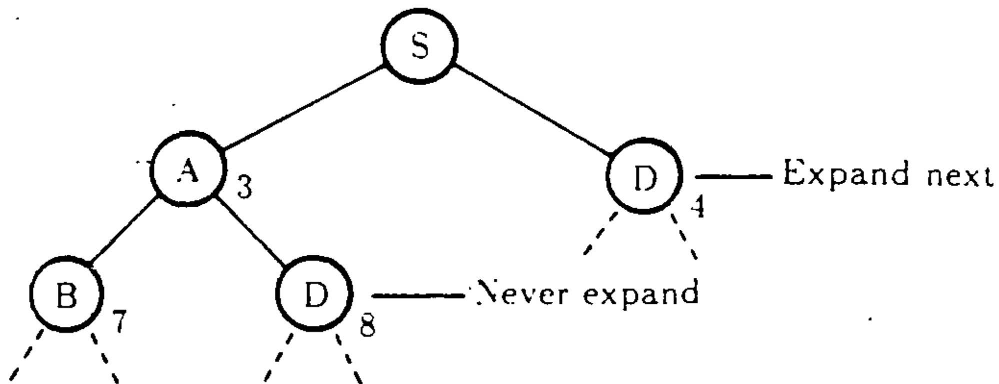


Branch-and-bound search augmented by underestimates determines that the path S D E F G is optimal. The numbers beside the nodes are accumulated distances plus underestimates of distances remaining. Underestimates quickly push up the lengths associated with bad paths. In this example, many fewer nodes are expanded than with branch-and-bound search operating without underestimates.

SEARCHING USING THE DYNAMIC PROGRAMMING PRINCIPLE:

"WHEN LOOKING FOR THE BEST PATH FROM S TO G, ALL PATHS FROM S TO ANY INTERMEDIATE NODE, I, OTHER THAN THE MINIMUM-LENGTH PATH FROM S TO I, CAN BE IGNORED."

SEE EXAMPLE BELOW:



An illustration of the dynamic-programming principle. The numbers beside the nodes are accumulated distances. There is no point to expanding the instance of node D at the end of S-A-D because getting to the goal via the instance of D at the end of S-D is obviously better.

BRANCH AND BOUND WITH DYNAMIC PROGRAMMING:

1) FORM A QUEUE OF PARTIAL PATHS. LET THE INITIAL QUEUE CONSIST OF THE ZERO-LENGTH, ZERO-STEP PATH FROM THE ROOT NODE TO NOWHERE.

2) UNTIL THE QUEUE IS EMPTY OR THE GOAL HAS BEEN REACHED, DETERMINE IF THE FIRST PATH IN THE QUEUE REACHES THE GOAL NODE.

   2a) IF THE FIRST PATH REACHES THE GOAL NODE, DO NOTHING.

   2b) IF THE FIRST PATH DOES NOT REACH THE GOAL NODE:

      2b1) REMOVE THE FIRST PATH FROM THE QUEUE

      2b2) FORM NEW PATHS FROM THE REMOVED PATH BY EXTENDING ONE STEP

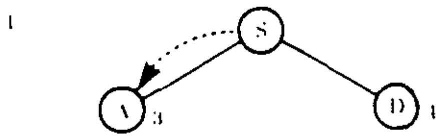
      2b3) ADD THE NEW PATHS TO THE QUEUE

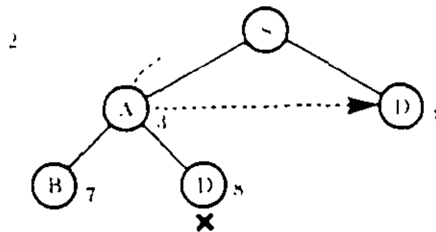
      2b4) SORT THE QUEUE BY COST ACCUMULATED SO FAR, WITH LEAST-COST PATHS IN FRONT.

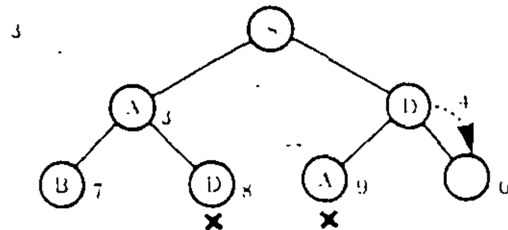
      2b5) *IF TWO OR MORE PATHS REACH A COMMON NODE, DELETE ALL THOSE PATHS EXCEPT THE ONE THAT REACHES THE COMMON NODE WITH THE MINIMUM COST.*

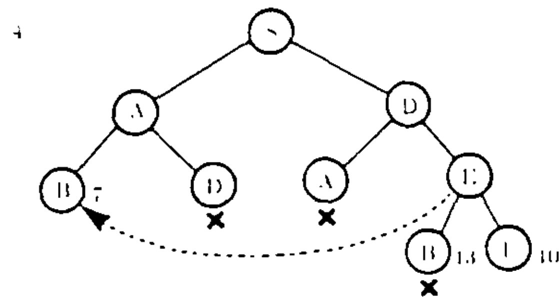
3) IF THE GOAL NODE HAS BEEN FOUND, ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

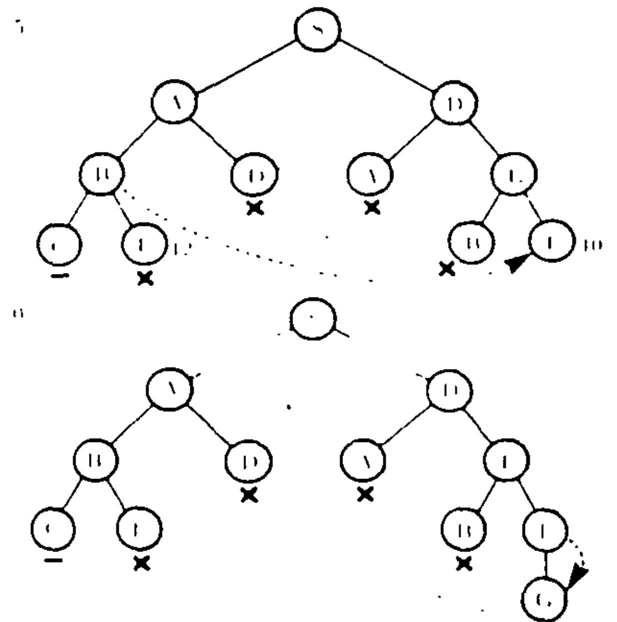
BRANCH AND BOUND WITH DYNAMIC PROGRAMMING EXAMPLE:











Branch-and-bound search, augmented by dynamic programming, determines that path S-D-E-F G is optimal. The numbers beside the nodes are accumulated distances. Many nodes, those crossed out, are found to be redundant. Fewer nodes are expanded than with branch-and-bound search operating without dynamic programming.

THE A* ALGORITHM (IMPROVED BRANCH AND BOUND):

1) FORM A QUEUE OF PARTIAL PATHS. LET THE INITIAL QUEUE CONSIST OF THE ZERO-LENGTH, ZERO-STEP PATH FROM THE ROOT NODE TO NOWHERE.

2) UNTIL THE QUEUE IS EMPTY OR THE GOAL HAS BEEN REACHED, DETERMINE IF THE FIRST PATH IN THE QUEUE REACHES THE GOAL NODE.

2a) IF THE FIRST PATH REACHES THE GOAL NODE, DO NOTHING.

2b) IF THE FIRST PATH DOES NOT REACH THE GOAL NODE:

2b1) REMOVE THE FIRST PATH FROM THE QUEUE

2b2) FORM NEW PATHS FROM THE REMOVED PATH BY EXTENDING ONE STEP

2b3) ADD THE NEW PATHS TO THE QUEUE

2b4) SORT THE QUEUE BY *SUM OF* COST ACCUMULATED SO FAR *AND A LOWER BOUND ESTIMATE OF THE COST REMAINING*, WITH LEAST-COST PATHS IN FRONT.

2b5) *IF TWO OR MORE PATHS REACH A COMMON NODE, DELETE ALL THOSE PATHS EXCEPT THE ONE THAT REACHES THE COMMON NODE WITH THE MINIMUM COST.*

3) IF THE GOAL NODE HAS BEEN FOUND, ANNOUNCE SUCCESS; OTHERWISE ANNOUNCE FAILURE.

Scientific Applications

F.N. Fritsch
Editor

An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles

Tomás Lozano-Pérez and Michael A. Wesley
IBM Thomas J. Watson Research Center

This paper describes a collision avoidance algorithm for planning a safe path for a polyhedral object moving among known polyhedral objects. The algorithm transforms the obstacles so that they represent the locus of forbidden positions for an arbitrary reference point on the moving object. A trajectory of this reference point which avoids all forbidden regions is free of collisions. Trajectories are found by searching a network which indicates, for each vertex in the transformed obstacles, which other vertices can be reached safely.

Key Words and Phrases: path finding, collision-free paths, polyhedral objects, polyhedral obstacles, graph searching, growing objects

CR Categories: 3.15, 3.64, 3.66, 8.1

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Authors' present addresses: T. Lozano-Perez, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139; M.A. Wesley, IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

© 1979 ACM 0001-0782/79/1000-0560 $00.75

1. Introduction

The problem of avoiding collisions when operating on computer models of physical objects is central to model-based manipulation systems. This paper describes an algorithm for planning safe, that is collision-free, paths for a polyhedral object among similarly described obstacles.[1] The algorithm is required to:

(1) find safe paths that might involve going near obstacles, and
(2) guarantee that these paths are short relative to a prespecified distance metric.

The simplest collision avoidance algorithms fall into the generate and test paradigm. A simple path from start to goal, usually a straight line, is hypothesized and then the path is tested for potential collisions. If collisions are detected, a new path is proposed, possibly using information about the detected collision to help hypothesize the new path. This is repeated until no collisions are detected along the path. Roughly, the three steps in this type of algorithm are

(1) calculate the volume swept out by the moving object along the proposed path,
(2) determine the overlap between the swept volume and the obstacles, and
(3) propose a new path.

The second step, determining the overlap between the swept volume and the obstacles, is also known as an intersection or interference calculation [2, 3]. Current computer modeling techniques employ large numbers of simple surfaces to model accurately even the most common objects. It can be quite difficult to determine whether two such models overlap. This general method, which we will call the *swept volume* method, has a more fundamental drawback. The problem is in the relationship between the second and third steps. Each proposed path provides only local information about potential collisions, for example, the shape of the intersections of the volumes involved, or the identity of the obstacle giving rise to the collision. This information suggests local path changes but is not sufficient to determine when a radically different path would be better. This lack of a global view can result in an expensive search of the space of possible paths with a very large upper bound on the worst case length of the path.

A radical alternative to the swept volume method is to compute explicitly the constraints on the position of the moving object relative to the obstacles. The desired trajectory is the shortest path which satisfies all the position constraints. If the objects are modeled as collections of convex polyhedra, the position constraints can be stated in terms of the position of the vertices of the mo-

[1] We will henceforth use the term "polyhedron" for closed figures bounded by "flats" in two or three dimensions.

ing object relative to the planes of the obstacle surfaces. The trajectory problem can then be posed as an optimization problem as in Ignat'yev [5]. The difficulty with this formulation is that these position constraints, although linear, do not all apply simultaneously. It is not necessary for each point on the moving object to be outside *all* the planes of the obstacles; it is sufficient for each point to be outside *at least one* of the planes of each obstacle. This property makes traditional linear optimization methods inapplicable.

The algorithm presented in this paper is closely related to the optimization approach. The constraints on the position of an arbitrary reference point on the moving object are computed. Polyhedral obstacles in two or three dimensions give rise to sets of polyhedral *forbidden regions*, that is, regions corresponding to positions of the reference point where collisions would occur. This transformation reduces the problem of finding a safe path for the polyhedron to the simpler problem of finding a safe path for a point. This last task is accomplished by finding a path through a graph connecting vertices of the forbidden regions.

The technique of computing the position constraints on an object as constraints on a reference point is extremely powerful and has been applied independently to different problems. It has been used by Udupa [9] for planning safe paths for computer-controlled manipulators, by Lozano-Perez [6] for identifying feasible grasp points on an object, and by Adamowicz and Albano [1] for two-dimensional template layout.

Udupa uses a simple "growing" transformation on obstacles to compute approximations to the forbidden regions for the three-dimensional reference point of a three degree of freedom subset of a manipulator. The system maintains a variable resolution description of the legal positions of the reference point (the *free space*). Safe paths for the subset manipulator are found by recursively introducing intermediate goals into a straight line path until the complete path is in free space. This method has two drawbacks:

(1) Because the complete manipulator has more than three degrees of freedom, the three-dimensional forbidden regions cannot model all the constraints on the manipulator. When a trajectory fails, Udupa's system makes a correction using manipulator-dependent heuristics. The use of heuristics tends to limit the performance of the algorithm in cluttered spaces.
(2) The recursive path finder uses only local information to determine a safe path and therefore suffers from some of the same drawbacks as the swept volume method.

The algorithm presented in this paper uses a more accurate growing operation to compute the forbidden regions in both two and three dimensions. It introduces a graph searching technique for path finding which produces optimum two-dimensional paths when only translations are involved. This technique is then generalized to deal with three-dimensional obstacles and extended to deal uniformly with more than three degrees of freedom. The resulting algorithm no longer guarantees optimum paths. This algorithm has been used to plan safe trajectories for a seven degree of freedom manipulator. These trajectories have been successfully executed.

A detailed survey of previous work in collision avoidance, specifically in connection with computer-controlled manipulators, can be found in Udupa [9].

The nature of the models used for the obstacles affects the details of any collision avoidance algorithm. For concreteness, the detailed discussions and examples in this paper assume that all objects are modeled as sets of, possibly overlapping, convex polyhedra. Any object can be modeled to any desired degree of accuracy in this fashion. A method for finding collision-free paths for a single convex polyhedron among sets of convex polyhedra can be simply extended to plan safe paths for a complex moving object among complex obstacles. The extension involves finding the constraints due to each of the convex components of the moving object relative to each of the components of all obstacles. The constraints for the composite moving object are the union of the constraints on its components.

The collision avoidance algorithm is defined for three dimensions. However, the presentation is easier to follow in two dimensions; for clarity the next sections first develop the complete algorithm for the two-dimensional case and then consider the extension to three dimensions. Section 2 presents a simple form of the algorithm for the case of a polygonal object translating in the plane among polygonal obstacles. Section 3 considers the effect of allowing the moving object to rotate as well as translate. Section 4 deals with more complex moving objects with more degrees of freedom. Section 5 discusses generalization to three dimensions. Discussion of the two steps of the algorithm that are directly affected by the choice of modeling methodology is relegated to the appendices. These steps will be functionally described in the body of the paper.

2. Collision Avoidance on the Plane

Consider the problem, shown in Figure 1, of moving a point object A from position S to position G while avoiding the obstacles (shown shaded); the shortest collision-free path from S to G is also shown. The important property of this path is that it is composed of straight lines joining the origin to the destination via a possibly empty sequence of vertices of obstacles. In the case of motion in the plane with arbitrary polygonal objects, the shortest collision-free path connecting any two accessible points always has this property.

The undirected graph $VG(N, L)$ is defined: The node

Fig. 1.

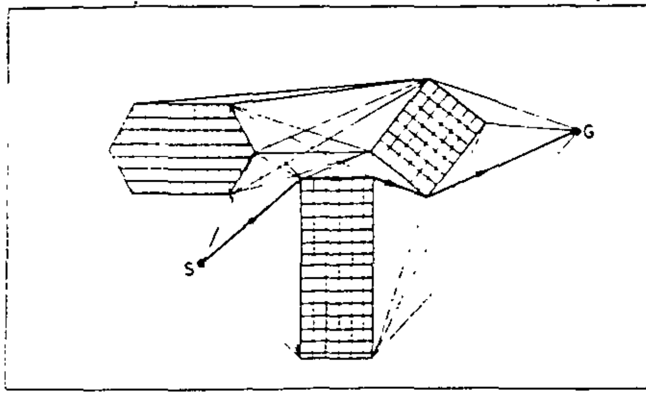


Fig. 2.

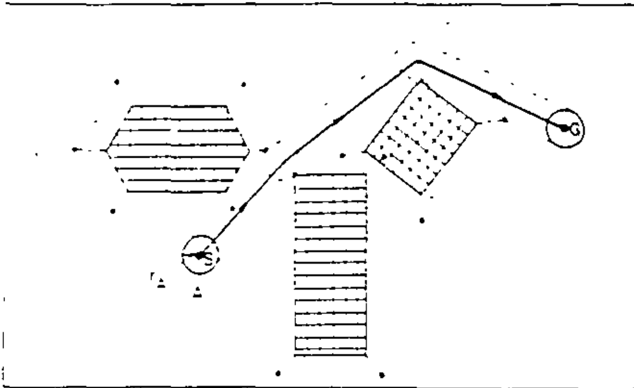


Fig. 3(a).

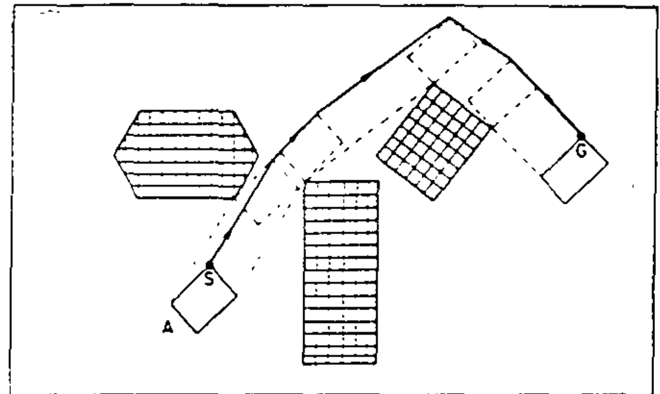
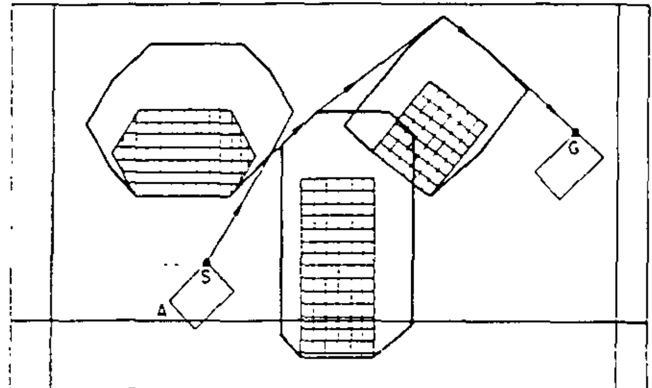


Fig 3(b).



set N is $V \cup \{S, G\}$ where V is the set of all vertices of obstacles and the Link set L is the set of all links (n_i, n_j) such that a straight line connecting the ith element of N to the jth does not overlap any obstacle. The graph VG(N, L) is called the *visibility graph* (VGRAPH) of N since connected vertices in the graph can see each other. The VGRAPH is shown in Figure 1. The shortest collision-free path from S to G on the plane is the shortest path in the VGRAPH from the node corresponding to S to that corresponding to G when the euclidean metric is used on the links. We will call this method for finding collision-free paths for a point by finding the shortest path in a visibility graph the VGRAPH algorithm. This method was used for navigating SHAKEY [8], an early robot vehicle, and is also described in some detail in Ignat'yev [5].

The simplicity of the VGRAPH algorithm stems from the fact that the moving object A is a point. This is a good approximation for moving objects which are small in relation to the obstacles, but causes problems otherwise. Ignat'yev [5, p. 241] puts it as follows

> The robot begins to move from the point y_0 (S in our example) along the direction to the x_{19} (a vertex). Here he must consider his dimensions in order not to run into the obstacles and walls

This paper shows how a more general form of the collision avoidance problem can be reduced to the VGRAPH problem. In other words, it concerns how the robot "must consider his dimensions."

A simple generalization of the problem in Figure 1 is to make the moving object A a circle with nonnegligible radius r_A. The VGRAPH algorithm can be adapted to this situation by moving the vertices away from the obstacles so that they are at least r_A away from all the sides (Figure 2). Moving A so that its center point moves through the new displaced vertices will still produce a minimum distance, collision-free path. Notice, however, that the path found is different from that in Figure 1. This technique of displacing the vertices was also used in SHAKEY [8].

The VGRAPH algorithm requires that the moving object be a point; the obstacles then represent the forbidden regions for the position of that point. If the moving object is not a point, a new set of obstacles must be computed which are the forbidden regions of some reference point on the moving object. These new obstacles must describe the locus of positions of this reference point which would cause a collision with any of the original obstacles. The displaced vertices of Figure 2 are, in fact, approximations to the vertices of these new obstacles when the reference point is the center of A.

The operation of computing a new obstacle O' from an original obstacle O and a moving object A will be called *growing O by A*. This name reflects the fact that

the obstacles are being grown so that the moving object can be shrunk to the reference point. The result of growing a set of obstacles by A will be indicated by GOS(A), i.e., the *Grown Obstacle Set* of A. Note that the growing operation is closely related to that of deriving the path of a machine tool to cut out a part.

Consider the situation in Figure 3(a). The same obstacles in Figures 1 and 2 are shown but the moving object A is now a rectangular solid. Figure 3(b) shows the obstacles after they have been grown by A. It also shows the shortest collision-free path for A's reference point from S to G. This figure demonstrates how the process of growing obstacles allows representing A as a point. Notice that the boundary of the obstacle space is treated as an obstacle and is also grown, thus avoiding paths which involve moving outside the space.

The growing operation was defined as computing the locus of positions of the moving object's reference point that would cause a collision with a given obstacle. The position of the moving object has been interpreted as its (x, y) position, i.e., the grown obstacles are polygons in (x, y) space. This is an arbitrary but natural choice. Different types of moving objects would call for different choices. Figure 4(a) shows one such case in an (x, y) coordinate system. The moving object A can rotate about a fixed point and can change length. This defines a polar coordinate system (r, α). Figure 4(b) shows the region of the (r, α) space which is forbidden to the tip of A by the presence of the obstacle in Figure 4(a); an alternative way of representing this region is shown in (x, y) coordinates in Figure 4(c). The choice of representation depends on:

(1) the ease of computing the forbidden regions, i.e., growing the obstacles, versus
(2) the ease of building the VGRAPH from the grown obstacles.

The use of polyhedra as the basic unit of shape description influences our choice of obstacle representation. Polyhedra (polygons when on the plane) have boundaries which are linear equations in the coordinate variables. This property makes them computationally attractive. In this section we have represented objects as polygons in a planar cartesian coordinate system. The natural choice is to express the grown obstacles in the same space, thus making the growing operation a mapping from polyhedra to polyhedra. Notice that in Figure 4(b) the object O was interpreted as a polygon in (x, y) space and the resulting grown obstacle O' in (r, α) is not a polygon in that space.

Another factor in the choice of obstacle representation is the shape of the path between two nodes in a VGRAPH. A link connecting two nodes in the VGRAPH implies that the path between the corresponding locations does not overlap any of the obstacles. Paths have so far been shown as straight lines in cartesian space; since the grown obstacles were in this coordinate space, the use of straight lines simplifies the detection of overlap. Of course, paths could be more complicated curves which are best expressed in different coordinate systems. For example, the object in Figure 4 might move in straight lines in the (r, α) system. In that case it might be more efficient to use the polar form of the grown obstacles in detecting overlap.

The choice of representation for the grown obstacles depends on the geometric details of the application domain. The choice should be made so as to simplify the overall computation. For the sake of simplicity the next section will continue to assume that the grown obstacles are polygons in (x, y) space.

Fig. 4(a)

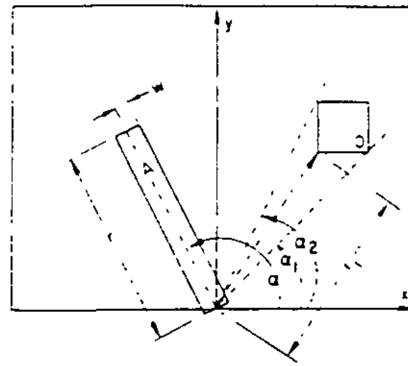


Fig. 4(b)

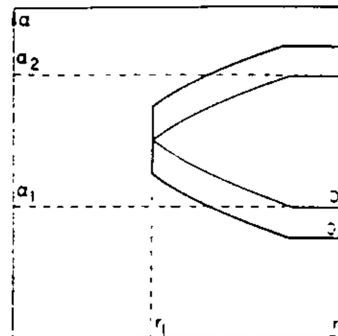


Fig. 4(c)

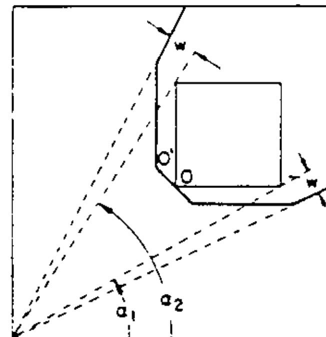


Fig. 5.

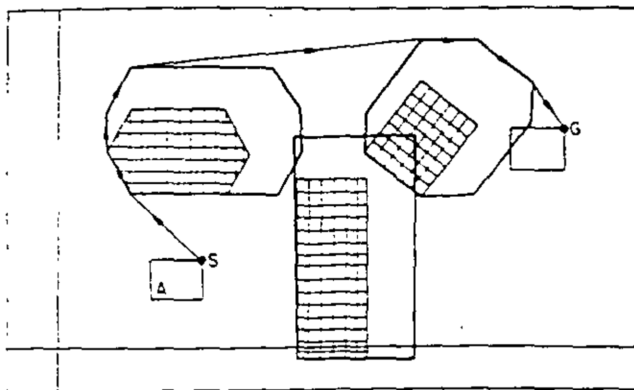
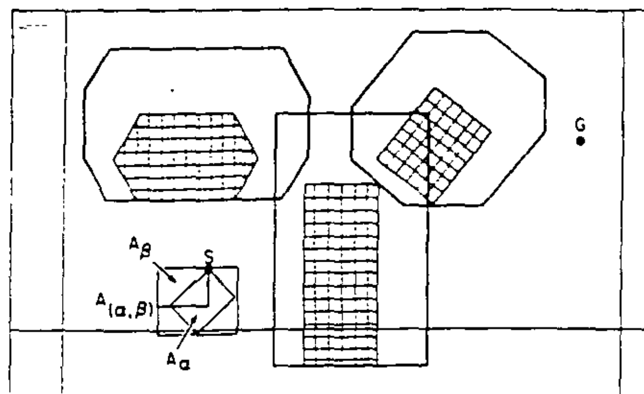


Fig. 6.



3. The Effect of Rotation

It is important to notice that the growing operation as shown in Figures 2 and 3 is sensitive to the orientation of A. This was not apparent in Figure 2 because the moving object was a circle. The orientation dependence follows from the fact that a grown obstacle is defined as the forbidden region for a *reference* point. The position of a point on the plane can encode only two degrees of freedom, whereas differentiating the legality of two positions of A with different orientations requires at least three degrees of freedom. Figure 5 shows that a different orientation of A from that in Figure 3 will produce different grown obstacles and a different path. To make the orientation explicit, we will denote the result of growing all the obstacles with a moving object A, whose orientation parameter is the angle α, $GOS(A_\alpha)$. The set of vertices of these grown obstacles will be called V_α.

To summarize, any position of A at orientation α for which A's reference point is outside all the elements of the grown obstacle set is free of collisions. The sides of each obstacle in $GOS(A_\alpha)$ are computed by tracing the path of A's reference point around each of the original objects while keeping A in contact with the obstacle. Before two objects collide they must first touch; therefore any position of the reference point that would cause a collision must be inside the obstacle, and any position outside must be safe. Clearly this condition presupposes that the orientation of A does not change.

Consider the problem of moving object A from position S with orientation α to G with a different orientation β. A safe trajectory cannot be found by simply computing a path that is free of collisions in $GOS(A_\alpha)$ and $GOS(A_\beta)$ since, in changing the orientation from α to β, A must pass through the whole range of intermediate orientations. One way to find a path requires knowing what positions on the plane will allow the desired rotation to take place. The algorithm can then plan a path from the start to one of these positions, rotate to the desired orientation, and move in that orientation to the goal.

For a position to allow a change in orientation there must be no overlap between the rotating object in any of its intermediate orientations and any of the obstacles. Figure 6 shows the area that A traverses in going from orientation α to β; this area may be approximated by another polygon $A_{(\alpha,\beta)}$ shown rectangular for simplicity. This new object, called an *envelope*, can be used to grow a new obstacle set $GOS(A_{(\alpha,\beta)})$, also shown in Figure 6, which represents the forbidden regions for the reference point of A in any of the orientations within the interval $[\alpha, \beta]$. We will refer to this as a *transition obstacle set*. By analogy to the vertex set V_α, the set $V_{(\alpha,\beta)}$ represents the set of vertices of obstacles in the transition obstacle set. In general we can associate with all the elements of a vertex set an orientation interval (possibly singular) as well as a position.

The problem in Figure 6 can now be solved by:

(1) finding a path starting with orientation α at S which avoids the obstacles in $GOS(A_\alpha)$ and which ends at a point clear of the obstacles in $GOS(A_{(\alpha,\beta)})$,
(2) rotating to orientation β, and
(3) finding a path to G avoiding the obstacles in $GOS(A_\beta)$.

This can be stated as a VGRAPH problem of finding the shortest path from S to G in a visibility graph defined as follows:

$$VG_{\alpha\beta}(N_{\alpha\beta}, L_{\alpha\beta})$$

where

$$N_{\alpha\beta} = V_{(\alpha,\alpha)} \cup V_{(\alpha,\beta)} \cup V_{(\beta,\beta)}$$
$$V_{(\alpha,\alpha)} = V_\alpha \cup \{S\}$$
$$V_{(\beta,\beta)} = V_\beta \cup \{G\}$$
$V_{(\alpha,\beta)}$ defined as above

and

$$L_{\alpha\beta} = \{(n_i, n_j)\}$$

$n_i \in V_{(a,b)}$ and $n_j \in V_{(c,d)}$ where a, b, c, d are either α or β

such that the following *visibility conditions* hold on the link:

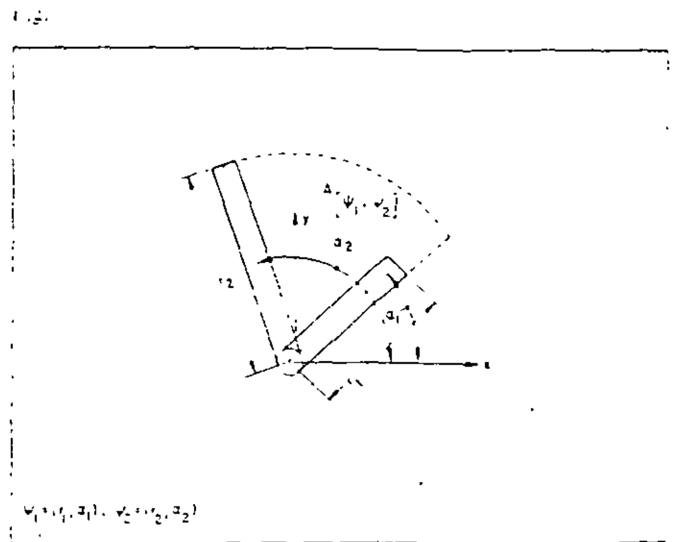
(1) the orientation intervals $[a, b]$ and $[c, d]$ must not be disjoint.
(2) n_i is outside all the obstacles in GOS($A_{[a,b]}$).
(3) n_j is outside all the obstacles in GOS($A_{[c,d]}$).
(4) the path from n_i to n_j either:
 (a) does not overlap any obstacle in GOS($A_{[a,b]}$), or
 (b) does not overlap any obstacle in GOS($A_{[c,d]}$).

A solution path in $VG_{\alpha,\beta}$ is a sequence of nodes starting at S and ending at G:

$S, n_1, n_2, \ldots, n_k, G$

in which adjacent nodes are connected by a link in $L_{\alpha,\beta}$. *Each* $n_i \in V_{a_i,b_i}$ is defined such that if n_i is outside all obstacles in GOS($A_{[a_i,b_i]}$), then the reference point of the moving object A can be at position n_i in any orientation within the interval $[a, b]$ without danger of collisions. Following the link from n_i to n_{i+1} means that the reference point of A must make the corresponding translation. Also, if n_i and n_{i+1} belong to different vertex sets, V_{a_i,b_i} and $V_{c,d}$ respectively, then a change of orientation may also be required. The conditions on $L_{\alpha,\beta}$ require that the orientation intervals corresponding to the endpoints of a link must not be disjoint. This means that there is some orientation x, such that if $a \leq b$ and $c \leq d$ then $\max(a, c) \leq x \leq \min(b, d)$, for which A can safely be at either node of the link. Moving along the link requires first rotating to the orientation x and then translating from the first node to the second. Since the translation happens in an orientation compatible with both nodes of the link, the visibility conditions on the link require only checking for overlap with the obstacles in the obstacle set of *either* one of the nodes. Alternatively, if the path from n_i to n_{i+1} is outside all obstacles in both GOS($A_{[a,b]}$) and GOS($A_{[c,d]}$), then the rotation may take place in conjunction with the translation along the link.

The use of transition sets, e.g., GOS($A_{[\alpha,\beta]}$), has two important drawbacks. The shortest solution path in $VG_{\alpha,\beta}$ is no longer guaranteed to be an optimum solution to the original problem, and failure to find a solution path in the VGRAPH does not necessarily mean that no safe trajectory exists. The reasons are twofold. The first and most basic is that paths found in this VGRAPH will change the orientation of the moving object only at locations where the full rotation can be performed. If the optimum path involves traversing a narrow passage where the orientation of A must be within a small subrange of the orientations between α and β, then this path could not be a solution path in this version of the VGRAPH algorithm. Secondly, even if the first problem were avoided, the current formulation considers orientations only in the range $[\alpha, \beta]$; it could not negotiate a passage where the moving object could only fit at an orientation outside the specified range. The latter problem can be solved simply by expanding the orientation interval, but only at the expense of making the former problem worse.



The two problems mentioned above can be alleviated by replacing the single transition obstacle set, GOS ($A_{[\alpha,\beta]}$), by the union of several other obstacle sets, each generated with a smaller orientation interval for the moving object. In this fashion the range of legal orientations can also be extended beyond the interval $[\alpha, \beta]$. As the number of transition obstacle sets increases, the VGRAPH becomes a better match to the original problem. Unfortunately, the computational burden also increases rapidly. Each new obstacle set requires growing all the obstacles with the moving object in a new configuration, though the growing operation can be speeded up by using approximations, as will be shown later. Also, the added vertices from the extra obstacle sets make searching the visibility graph much more time consuming. Alternatively, it may be possible to derive automatically transition sets to handle narrow passages specifically, and combine these with wider-range transition sets.

4. More Degrees of Freedom

Transition obstacle sets can be used whenever the moving object has more degrees of freedom than can be represented by a point in the obstacle coordinate space. The only requirement is that it be possible to compute an envelope $A_{[x,y]}$ which is an object of the same type as A, e.g., a polygon, such that any point inside an A_z, $x \leq z \leq y$, is also inside $A_{[x,y]}$. This object then can be used in the growing operation to generate a transition obstacle set. There are no other restrictions on the nature of the parameter range $[x, y]$; in particular, it need not be an orientation range and both x and y may also be vectors. A point outside all of the obstacles in GOS($A_{[x,y]}$) indicates a position where each of A_z's *configuration parameters*, z_i, can safely take on values such that $x_i \leq z_i \leq y_i$.

Figure 7 repeats the example of Figure 4 except that now the moving object can translate in x and y as well as rotate and change its length. The choice of a coordinate system for the grown obstacles will also determine which of the coordinate variables is to be used for the configuration parameters. For example, if the grown obstacles are represented as polygons in (x, y), then (r, θ) are configuration parameters and vice-versa.

Configuration parameters can also be used to deal with a moving object whose shape can change due to changes in the relative positions of its components. The object shown in Figure 8 is composed of two rectangles that are free to rotate about a common point. The shape of this object relative to a stationary obstacle can be described by:

(1) the shape of its components.
(2) their relative displacements.
(3) the two angles θ and ρ indicated in Figure 8.

In this example only the angles can change during a motion, therefore the obstacle set for this moving object must be parameterized by the value of both θ and ρ. Generally the configuration parameters describe not only the global orientation or position of the object but also the relative positions of its components.

In general, objects need not be grown in the full dimensional configuration space: instead, repeated use is made of operations on lower dimensional, partitioned, configuration spaces which allows the growing operation to work in a convenient subspace of the full configuration space. The VGRAPH algorithm described in Sections 2 and 3 remains unchanged except that the scalar parameters and intervals are replaced by vector parameters and intervals.

Fig. 8.

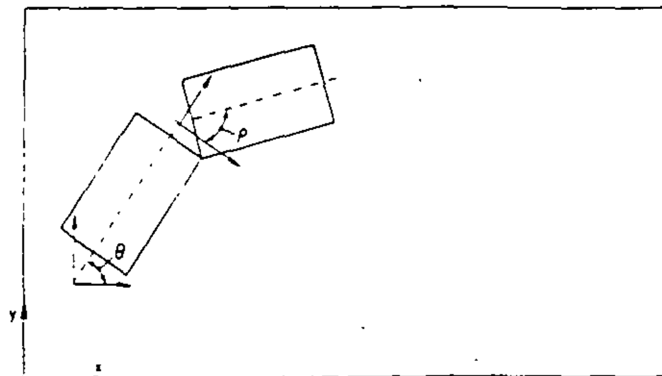
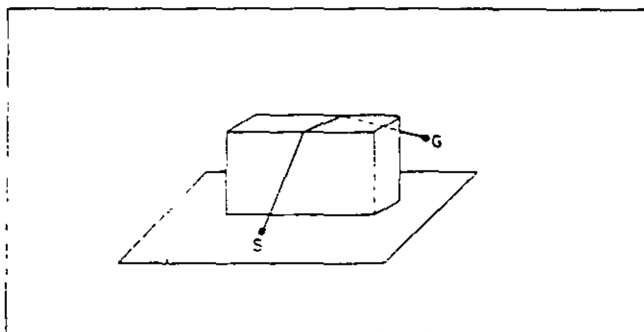


Fig. 9.



5. Collision Avoidance in Three Dimensions

The VGRAPH algorithm has so far been presented as an algorithm for collision avoidance on the plane. This section examines how three-dimensional obstacles affect the algorithm. This generalization does not affect the statement of the algorithm but does affect the details of the obstacle growing and graph searching. These subjects are discussed in the appendices.

The generalization to three dimensions has an unfortunate side effect: The shortest path around a polyhedral obstacle does not in general traverse only vertices of the polyhedron (Figure 9). That is, the shortest path in a VGRAPH whose node set contains only vertices of the grown obstacles is not guaranteed to be the shortest collision-free path. In general, the shortest path will involve going via points on edges of the obstacles. Our approach is to introduce additional vertices along edges of the grown obstacles so that no edge is longer than a prespecified maximum length. This method generally results in a good approximation to the optimum path.

The use of three-dimensional obstacles also has a significant effect on the execution time of the algorithm. The three-dimensional growing operation is much more time consuming than the corresponding operation in two dimensions. Grown obstacles in three dimensions are generally much more complex than the underlying objects (Appendix 1). The larger vertex sets also increase the time necessary to search the visibility graph. These effects make the use of approximations necessary for practical applications.

A great saving can be realized by using the detailed growing operation sparingly. Many application domains have the property that the moving object need only be close to obstacles at a small number of points along the path. These *care points* usually include the start and goal of the path. Elsewhere the requirements on the path are less strict, in fact, it is often undesirable to move close to the obstacles when away from the care points. This property can easily be exploited in the VGRAPH algorithm, instead of executing the detailed growing operation on each of the known obstacles, it need only be executed on those obstacles close to the care points. Away from the care points drastic approximations can safely be used. Complex objects, built up from many polyhedra, can be approximated by a single enclosing polyhedron. The moving object can be similarly approximated so as to further simplify the process. In addition, a very simple form of the growing operation (Appendix 1) can be used

Fig. 10.

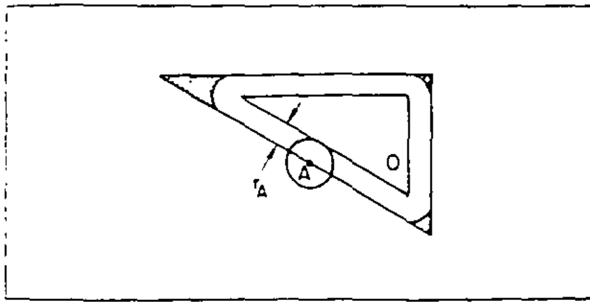
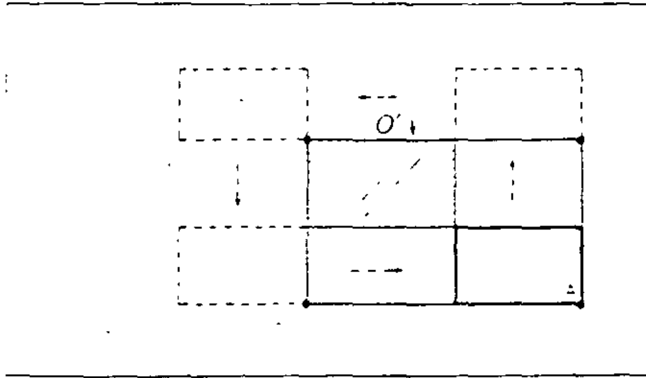


Fig. 11



which, at the expense of accuracy, is faster and results in simpler objects.

The key to using this approximation technique is an effective way of determining which objects are close to the care points. Clearly a care point is close to an object if it is inside or close to one of the sides of the grown obstacle resulting from it. This means that the moving object when located at the care point is either inside or close to a side of the object. Approximating both the moving and the stationary objects will cause the care point to be inside the grown obstacle. This condition can be used as a criterion for careful growing. When the moving object is large relative to the obstacles, approximating it as a single object results in detailed growing of too many obstacles. The larger the moving object, the worse a simple approximation is likely to be. In particular, some part of the moving object, relatively far from the care point, will cause the grown obstacle to include the care point. The solution is to have a hierarchic decomposition of the moving object; that is, if the test fails for the roughest description, then use a slightly better approximation. In this way the source of potential collision can be better isolated. The other components of the moving object which are not involved need not be considered carefully. Udupa [9] proposed a similar variable level of detail approximation scheme.

Another way to increase the efficiency of the algorithm is to use heuristics in the graph search operation. This is discussed briefly in Appendix 2 which deals with searching the VGRAPH.

6. Summary and Discussion

This paper has shown how the simple visibility graph algorithm used for navigation of SHAKEY [8] can be extended to more general collision avoidance problems. The mechanism necessary to achieve this involves growing the obstacles and shrinking the moving object to a point. This approach has the desirable property of providing two subproblems, growing the obstacles and searching a visibility graph, which can be pursued independently. A description of our current approach to these problems is included in the appendices.

The most important remaining problem with the VGRAPH algorithm is the quantization of configuration parameters into intervals. Paths that require almost continuous changes of orientation as well as position require small quantization intervals, resulting in many transition obstacle sets, and are therefore expensive to compute.

The VGRAPH algorithm as described in this paper has been implemented in PL/1 on an IBM 370/168. It has been used to plan collision-free trajectories for a seven degree of freedom computer-controlled manipulator [10]; these trajectories have been successfully executed in the laboratory.

Appendix 1: Growing the Obstacles

This appendix describes the component of the VGRAPH algorithm that computes from an obstacle description the shape of the forbidden regions for the position of the moving object's reference point. This is called growing the obstacle. The ideas will be developed in two dimensions and then extended to their three-dimensional counterparts. In the initial two-dimensional case the degrees of freedom used for *growing* will be the x and y position of the moving object.

Consider growing a polygonal obstacle by a circular solid as in Figure 10. The simplest growing algorithm moves each of the sides of the original obstacle by a constant amount r_A and then intersects the lines to obtain the vertices of the grown polygon. The drawbacks of this algorithm are twofold:

(1) It works well only for moving objects that are nearly circular.
(2) It generates wasted space near pointed corners, as seen by the dark shaded regions in Figure 10. This problem can be alleviated by clipping the corners of the grown polygon.

This was the form of the growing algorithm used by Udupa [9].

Fig 12(a).

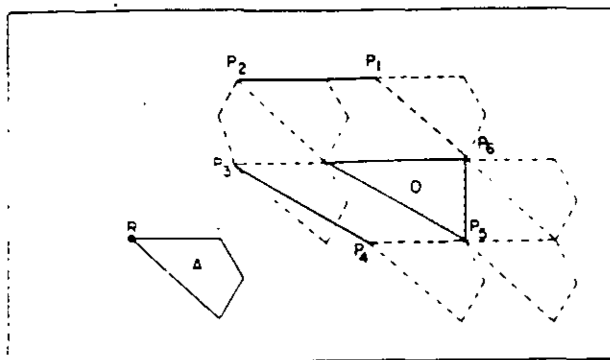
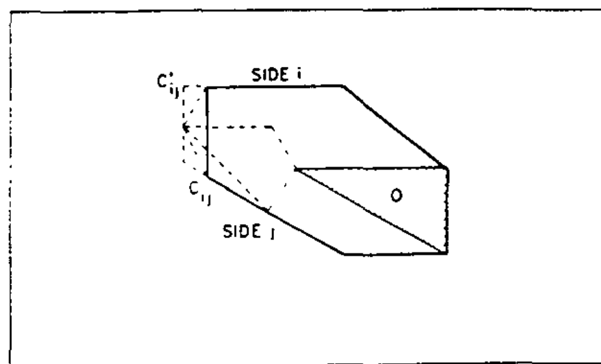


Fig. 12(b).



A simple variation of this procedure will solve problem 1 above. Figure 11 shows a convex polygon O and a moving object A. both are rectangular and both are aligned with the global coordinate axes. R, the reference point of A, coincides with one of the vertices of A. The boundary of the forbidden region for the position of A is the locus of positions of R for which A is in contact with O. This locus defines another convex polygon O' shown in Figure 11. Clearly any point inside this polygon implies a collision between A and O. This grown polygon has side $nside_i$ corresponding to each side $side_i$ of the original obstacle. The distance from $nside_i$ to $side_i$ is the perpendicular distance of R from $side_i$ minus the perpendicular distance to the point where A would first contact $side_i$. The distance from $side_i$ to this contact point on A is the minimum perpendicular distance of all of the vertices of A from $side_i$. Once the sides are displaced by this amount, the lines can be intersected to generate the grown polygon.

This method only makes use of the distance from the reference point to the contact point for a side. In polygons with interior angles of less than a right angle the method described above produces wasted space at the vertices. This waste can be reduced by simply cutting the corner at a conservative distance. A more accurate growing procedure can be obtained by determining the actual locus of motion of R along each $side_i$ as the contact point slides along the side. Figure 12(a) shows the line segments traced out by this procedure (bold lines). Notice that the line segments do not intersect and that the endpoints of these line segments correspond to the position of R when the contact point of A with O is at a vertex of O. These positions will be referred to as *maximal locus points*.

To complete the figure, notice that the locus of R as A moves from its contact point with $side_i$ to its contact point with the adjacent $side_j$ traces successive edges of A between the two contact points on A. In the course of connecting all the maximal locus points, all the edges of A are traced out in reverse order (heavy dashed lines). A simple algorithm for growing convex polygons exists which is based on merging a list of displaced edges of O with a reverse order list of displaced edges of A. To simplify the geometry of the grown object, the locus of R between successive contact points can be conservatively estimated by a straight line c'_{ij} which is parallel to the line c_{ij} connecting the maximal locus points but displaced to the position of the point on the actual locus furthest from the line c_{ij} as shown in Figure 12(b).

The *approximate* method for growing a convex polyapproach is to grow each face of the polyhedron independently and then introduce new faces to complete the grown polyhedron. The steps in the process of growing a rectangular solid O with a rotated rectangular solid A are shown in Figure 13. The locus of R as the contact point of A moves along each edge of $face_i$ is called the *maximal locus edge*, Figure 13(a). Such edges define potential new edges for the faces of the grown polyhedron. Each edge of O generates two adjacent maximal loci. These edges have to be connected in a manner analogous to the way in which maximal locus points are connected in a grown polygon, Figure 12(b). The edge has to be displaced to compensate for points on A which are closer to O than the plane defined by the two adjacent maximal locus edges and passing through the corresponding contact points. Faces are introduced to connect each pair of edges of the grown faces arising from a common edge, Figure 13(b). These new faces introduce new edges, each of which connects two points on the grown faces arising from a common vertex of A. All the edges corresponding to a single vertex also define a new set of faces, Figure 13(c). The total number of faces in a polyhedron grown from an object O in this fashion is equal to the sum of the numbers of faces, edges, and vertices of O.

The operation of growing a polyhedron is related to an operation known as *mixing* polyhedra [7]. A mixed polyhedron is the set of points which can be expressed as a linear combination of points from the two starting polyhedra. A polyhedron isomorphic to a grown obstacle can be obtained by mixing the underlying obstacle with a negative image of the moving object, as in convolution.

Fig. 13(a).

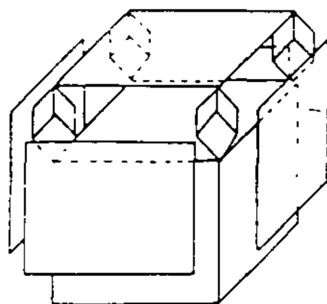


Fig. 13(b).

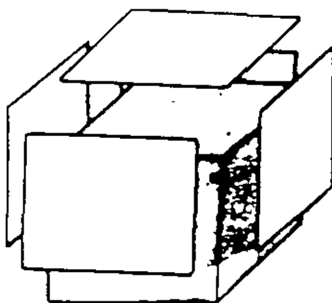
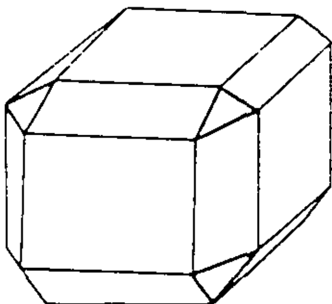


Fig. 13(c)



Appendix 2: Finding a Path

A generalized *visibility graph* VG(N, L) contains a node set N and a link set L_{ij} of links between node pairs (n_i, n_j) for which a visibility function $l(n_i, n_j)$ is true. A node is a representation of a region in an n-dimensional parameter space: for each node the associated region is represented by two n-dimensional parameter vectors ϕ_1 and ϕ_2. Individual elements of the difference vector $\delta\phi = \phi_1 - \phi_2$ will be zero when the corresponding parameter has a fixed value, or nonzero when the parameter has a range of values.

The path finding problem is defined as: Given a node set N with an associated parameter vector set, a start node n_s, and a goal node n_g, find a sequence of nodes from n_s to n_g, by way of an ordered set of intermediate nodes $n_1, \ldots, n_k$, which may be null, such that the visibility function for each node pair in sequence:

$$L_1 = l(n_s, n_1)$$
$$L_2 = l(n_1, n_2)$$
$$\ldots$$
$$L_{k+1} = l(n_k, n_g)$$

is true, and that a cost function

$$C = \sum c(n_i, n_j)$$

where $c(n_i, n_j) \geq 0$ is minimized.

A direct approach to finding an optimum path is to enumerate all possible paths and choose one for which C is minimum. For node sets whose cardinality is of practical interest (e.g., > 50) the computational load of the direct approach is prohibitive, and more efficient heuristic based search methods may be used.

The A^* algorithm of Hart et al. [4] allows use of efficient heuristic information. For each node, an estimate *hhat* is made of the cost h to travel from the node to the goal. Initially, n_s is placed on a list of candidate nodes for examination (the OPEN list). At each step of the algorithm, the node with minimum total path cost estimate (i.e., actual cost of reaching the node along the trial path plus *hhat*) is moved onto a CLOSE list and its minimum cost estimate visible successor nodes are placed on the OPEN list.

Hart et al. have shown that the A^* algorithm finds an optimum path when *hhat* is a lower bound estimate of the true cost h. When the estimator for *hhat* is zero and some $c(n_i, n_j) \rightarrow = 0\ n_i, n_j \in N$, the estimate gives no heuristic information to assist in the choice of a path; as $hhat \rightarrow h$, the heuristic information increases and the average number of unsuccessful trial paths is reduced. In the context of this paper, the lower bound requirement for *hhat*, i.e., $hhat \leq h$, may be met by assuming $l(n_i, n_g)$ to be true and computing $hhat_i = c(n_i, n_g)$.

The cost function $c(n_i, n_j)$ may be tailored to suit the requirements of a particular problem environment, for example:

- distance to be traveled in a subspace of the parameter space;
- functions of distances in parameter space, for example, time to complete a change based on allowable rate of change of parameters, with the option of selecting the limiting (i.e., slowest) dimension.
- special costs may be assigned to particular node sequence pairs to allow, for example, costs to be assigned depending on whether the pair allows the motion to proceed without a speed change, as opposed to pairs requiring a change of speed or an intermediate halt.

The form of the visibility function $l(n_i, n_j)$ depends on the semantics of the parameters. Two mutually exclusive classes of parameters are considered as described in Section 3:

- those that may vary continuously, that is, those embodied in the growing operation;
- those that may occupy only discrete ranges, that is, those that are represented by transition obstacle sets.

Note that this distinction between continuous and discrete parameters is an artifact introduced to simplify the handling of spaces of high dimensionality (i.e., $n > 3$) and that the partitioning of the parameter set is not unique: In general, either linear or rotary motions may be represented by continuous or discrete ranges. In the case of discrete range parameters, specific values must be chosen to enable l and *hhat* to be evaluated. In all cases of parameter change, a path function defines the motion effect of the change, as either linear or nonlinear motions in parameter space.

In the formalization for path planning described in the body of this paper, an obstacle A is grown under some parameter dependent transformation to produce GOS($A_{\phi 1,\phi 2}$) where ($\phi 1$, $\phi 2$) represents a range of parameters. Three parameters represent continuous motion, and the rest represent discrete motions. The visibility function is line of sight in three-dimensional orthogonal cartesian space, with the provision that visibility is possible only when the discrete parameter ranges at the start and end of the path segment overlap, and values assigned to these parameters are in the overlap region.

In many practical situations, the computational cost of evaluating l is very much greater than that of evaluating *hhat* for candidate successor nodes. In such cases it is computationally efficient to select a candidate successor node in terms of minimum *hhat* before computing l.

Acknowledgments. We would like to thank P. Will for providing the support, guidance, and encouragement for this work. D. Grossman and L. Lieberman contributed useful discussions and criticism as well as programming advice and assistance. Conversations with R. Taylor triggered the development of the multiple obstacle set approach.

Received June 1978; revised August 1979

References

1. Adamowicz, M., and Albano, A. Nesting two-dimensional shapes in rectangular modules. *Comptr. Aided Design 8*, 1 (Jan. 1976), 27-33.
2. Boyse, J.W. Interference detection among solids and surfaces. *Comm. ACM 22*, 1 (Jan. 1979), 3-9.
3. Braid, I.C. *Designing with Volumes.* Cantab Press, Cambridge, England, 1973.
4. Hart, P., Nilsson, N.J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernetics SSC-4*, 2 (July 1968), 100-107.
5. Ignat'yev, M.B., Kulakov, F.M., and Pokrovskiy, A.M. Robot manipulator control algorithms. Rep. No. JPRS 59717, NTIS, Springfield, Va., Aug. 1973.
6. Lozano-Pérez, T. The design of a mechanical assembly system. Rep. No. AI-TR-397, Artif. Intell. Lab., MIT, Cambridge, Mass., Dec. 1976.
7. Lyusternik, L.A. *Convex Figures and Polyhedra.* Dover Publications, N.Y., 1963. (Translated from the Russian by T.J. Smith; original copyright Moscow, 1956.)
8. Nilsson, N.J. A mobile automaton: An application of artificial intelligence techniques. Proc. Int. Joint Conf. Artif. Intell., 1969, pp. 509-520.
9. Udupa, S. Collision detection and avoidance in computer controlled manipulators. Ph.D. Th., Calif. Inst. of Technology, Pasadena, Calif., 1977.
10. Will, P.M., and Grossman, D.D. An experimental system for computer controlled mechanical assembly. *IEEE Trans. Comptrs* (1975), 879-888.

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS ABIERTOS

ROBOTS MÓVILES

SISTEMAS EXPERTOS

EXPOSITOR : DR. JESÚS SAVAGE CARMONA
1998

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtemoc 06000 Mexico, D.F. APDO. Postal M-2285
Telefonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

CHAPTER 7 INTRODUCTION TO CLIPS

7.1 INTRODUCTION

This chapter begins the introduction of the practical concepts necessary to build an expert system. The previous chapters have discussed the background, history, definitions, terminology, concepts, tools, and applications of expert systems. In short, they have provided an understanding of what expert systems are and what they can do. This theoretical framework of concepts and algorithms is essential in building expert systems. However, there are many practical aspects to building expert systems which must be learned by doing. Building an expert system is much like writing a program in a procedural language. Knowing how an algorithm works is not equivalent to being able to write a procedural program to perform that algorithm. Similarly, capturing an expert's knowledge is not equivalent to building an expert system. For this reason, practical experience using an expert system tool is invaluable in learning about expert systems.

The expert system language that will be used to demonstrate various concepts in the rest of the book is CLIPS. This chapter describes the basic components of an expert system (as discussed in Chapter 1) that are found within CLIPS. The basic elements of CLIPS are:

1. **fact-list**: global memory for data
2. **knowledge-base**: contains all the rules
3. **inference engine**: controls overall execution

These three components of CLIPS will provide the emphasis of this chapter. The first component, facts, will be covered in detail. Adding, removing, browsing, and tracing facts will be discussed. Following this will be a discussion on how the rules of a CLIPS program interact with facts to make a program execute. Finally, the interaction of multiple rules will be demonstrated.

To provide the basis for many examples throughout this chapter, a hypothetical expert system that monitors the activities of an industrial plant will be used. This expert system will monitor and respond to a range of possible emergencies. One such emergency would be a fire. Another would be a flood.

7.2 CLIPS

CLIPS is a forward chaining rule-based language that has inferencing and representation capabilities similar to those of OPS5. Syntactically, CLIPS very closely resembles a subset of ART. ART, however, is a multi-paradigm tool which provides forward and backward chaining, hypothetical reasoning, a schema representation language, and

object-oriented programming among other features. CLIPS provides only the forward chaining capabilities of ART. Backward chaining, hypothetical reasoning, a schema representation language and object-oriented programming are not provided in CLIPS.

CLIPS was designed at NASA/Johnson Space Center with the specific purposes of providing high portability, low cost, and easy integration with external systems. CLIPS was written using the C programming language to facilitate these objectives. CLIPS is an acronym for C Language Integrated Production System. The C Language portion of the acronym can be somewhat misleading, however, since there is also an ADA version of CLIPS.

Because of its high portability, CLIPS has been installed on a wide variety of computers ranging from PCs to CRAY supercomputers. The majority of examples shown in this and the following chapters should work on any computer on which CLIPS has been installed. It is recommended, however, that CLIPS users have some knowledge of the operating system of the computer on which CLIPS is being used. For example, the method for specifying files tends to vary from one operating system to another. When commands may vary depending upon the machine or operating system, this will be noted.

7.3 NOTATION

This chapter and the following chapters will use the same notation for describing the syntax of various commands and constructs that are introduced. This notation consists of three different types of text which are to be entered.

The first type of notation is for words and characters that are to be entered exactly as shown. Anything that is not enclosed by the symbol pairs < >, [ ], or { }, should be entered exactly as shown. For example, consider the syntax description shown following.

```
(example)
```

This syntax description means that *(example)* should be entered as shown. To be exact, the character '(' should be entered first, followed by the character 'e', then 'x', 'a', 'm', 'p', 'l', 'e' and finally the character ')'.

Brackets, [ ], indicate that the contents of the brackets are optional. For example, the syntax description

```
(example [1])
```

indicates that the 1 found within the brackets is optional. So the following entry would be consistent with the syntax

```
(example)
```

as would this entry

```
(example 1)
```

The less than and greater than symbols, < > indicate that a replacement is to be made with the value specified by the words found within the < >. For example, finding the following within a syntax description

```
<integer>
```

indicates that a substitution should be made with an actual integer value. Following the previous examples, the syntax description

```
(example <integer>)
```

could be replaced with

```
(example 1)
```

```
(example 5)
```

or

```
(example -20)
```

or many more entries which contained the characters "(example ", followed by an integer, followed by the character ')'. It is important to note that spaces shown in the syntax description should also be included in the entry.

Another notation is indicated by the symbol pair << >> enclosing a description. This indicates that the description can be replaced with zero or more occurrences of the value specified. Spaces should be placed after each occurrence of a value. For example, the syntax description

```
<<integer>>
```

could be replaced with

```
1
```

```
1 2
```

or

```
1 2 3
```

or with any number of integers, or with nothing at all.

A description enclosed by three less than and three greater than symbols indicates that one or more of the values specified by the description should by used in place of the syntax description. Note that for this notation, the syntax description

```
<<<integer>>>
```

is equivalent to the syntax description

```
<integer> <<integer>>
```

Braces, { }, indicate that one and only one of the items contained within the braces should be used in place of this syntax description. The items within the braces are separated by commas. For example, the syntax description

```
{all,none,some}
```

could be replaced with

```
all
```

or

```
none
```

or

```
some
```

7.4 FIELDS

As a knowledge base is constructed, CLIPS is required to read input from the keyboard and files to execute commands and load programs. As CLIPS reads characters from the keyboard or files, it groups them together into **tokens**. Tokens represent groups of characters that have special meaning to CLIPS. Some tokens such as left and right parentheses consist of only one character.

The group of tokens known as **fields** are of particular importance. There are three types of fields. A **word** is one type of field that starts with a printable ASCII acter

and is followed by zero or more characters with certain exceptions. A word may not begin with any of the following characters or groups of characters

```
<   |   &   $   ?   +   -   (   )   ;
```

In addition, a word may not contain any of the following characters

```
<   |   &   (   )   ;
```

These characters have special meaning to CLIPS, and so act as **delimiters** to indicate the end of a word.

Examples of valid words are the following.

```
emergency
fire
emergency-fire
activate_sprinkler_system
notify-fire-department
shut-down-electrical-junction-387
!?#$^*
```

Notice how the underscore and dash characters are used to tie words together to make them a single field. The last word shown demonstrates that words can have unusual characters placed within them.

CLIPS will preserve the uppercase and lowercase letters it finds in tokens. CLIPS is said to be **case-sensitive** because it distinguishes between uppercase and lowercase letters. For example, the following word fields are considered by CLIPS to be different

```
fire
FIRE
Fire
```

The second type of field is the **string**. A string must begin and end with double quotes. The quotes are part of the field. There can be zero or more characters of any kind between the double quotes including characters normally used by CLIPS as delimiters. The following shows some definitions of strings.

```
"Activate the sprinkler system."
"Shut down electrical junction 387."
"!?#$^"
"<-;()  +-"
```

Spaces normally act as delimiters in CLIPS to separate fields (such as words) and other tokens. Additional spaces used between tokens are discarded. Spaces included as

part of a string, however, are preserved. For example, the following st s are considered by CLIPS to be four distinct strings.

```
"fire"
"fire "
" fire"
" fire "
```

If the surrounding double quotes were removed, CLIPS would consider each of the lines to contain the same word since spaces other than those used as delimiters would be ignored.

Since double quotes are used to delimit strings, it is not possible to directly place a double quote within a string. For example, the following line

```
""fire""
```

would be interpreted by CLIPS as the three separate tokens shown following

```
""
fire
""
```

since double quotes act as delimiters.

Within a string, double quotes can be included by using the backslash operator, \. For example, the line

```
"\"fire\""
```

will be interpreted by CLIPS as the string field

```
""fire""
```

Only a single field is created because the backslash character prevents the following double quotes from acting as a delimiter. The backslash character itself may be placed within a string by using two backslashes in succession. For example, the line

```
"\\fire\\"
```

will be interpreted by CLIPS as the string field

```
"\fire\"
```

The third type of field is the **numeric field** or simply **number**, ıich represents a floating point number. All numbers in CLIPS, including integer values,

are stored as floating point numbers. A numeric field consists of three parts: the sign, the value, and the exponent. The sign and the exponent are optional. The sign is either + or -. The value contains one or more digits with a single optional decimal point contained with the digits. The exponent consists of the letter e or E followed by an optional + or - followed by one or more digits.

The following are all examples of valid CLIPS numbers

```
1
1.5
.7
+3
-1
65
3.5e10
```

7.5 FACTS

In order to solve a problem, a CLIPS program must have data or information with whic' can reason. A "chunk" of information in CLIPS is called a **fact**. A fact cons .f one or more fields enclosed in matching left and right parentheses. The following are all examples of facts

```
(single-field)
(two fields)
(speed 38 mph)
(cost 78 dollars 23 cents)
(name "John Doe")
```

Although CLIPS will accept any combination of fields as a fact, good programming style dictates that a method be used to insure that facts are represented in a meaningful manner. For example, consider the following sets of facts

```
(fire)
(flood)
(Tuesday)
(Wednesday)

(emergency-fire)
(emergency-flood)
 'ay-Tuesday)
 .ay-Wednesday)

(emergency fire)
```

```
(emergency flood)
(day Tuesday)
(day Wednesday)
```

In the first set of facts, the facts (fire) and (flood) could be used to indicate t
presence of these occurrences, as the facts (Tuesday) and (Wednesday) could be used
indicate days of the week. Notice that the facts (fire) and (flood) are related by bo
being types of emergencies and the facts (Tuesday) and (Wednesday) are both days of t
week. However, any relationship between the facts (fire) and (flood) and between t
facts (Tuesday) and (Wednesday) can only be inferred. No explicit statement indicates
relationship between the facts (fire) and (flood) exists any more than a relationsh
between the facts (fire) and (Tuesday) exists.

The second set of facts goes a step further and implies a relationship between fac
by using similar components as the first part of each word. Each fact, however, onl
contains a single field. Since fields are treated as indivisible items, a rule will not b
able to access the prefix part of the field. These facts, then, while being recognized by
person as being similar, cannot be recognized by a rule as being similar.

The third set of facts explicitly declares the relationship between fire and flood ar
between Tuesday and Wednesday. The first field of the facts, emergency and day, is use
to describe the relationship of the following fields. When used this way, the first fie
is called a **relation name**. The remaining fields of the fact are then used for specif
values. In this case, fire and flood are types of emergencies, and Tuesday and esda
are days. The use of a relation name explicitly states the relationship between fire an
flood and the relationship between Tuesday and Wednesday.

7.6 FACT TEMPLATES

Groups of facts having the same relation can be described for purposes o
documentation by using a **template**. A template for a fact shows the relation nam
followed by one or more general items or specific items. For example, the template fo
the *emergency* facts is

```
(emergency <type>)
```

where the symbol <type> indicates a general item that could be replaced by a specifi
item such as fire or flood. More than one template can be used to indicate variations of
a set of facts. For instance, the following facts

```
(action activate sprinkler-system)
(action activate fire-alarm)
(action notify fire-department)
(action shutdown-electrical-power)
```

could be represented with the following three templates

```
(action activate <device>)
(action notify <group>)
(action <specific-action>)
```

More than one field can be represented with a template. The fact

```
(computer-components cpu disk-drive terminal)
```

could be described with the template

```
(computer-components <<component-list>>)
```

where <<component-list>> represents zero or more fields. The template could be slightly modified to represent more than one component list. For example, these facts

```
(components computer cpu disk-drive terminal)
(components car tires engine body gas-tank)
```

a [illegible] scribed by this template

```
(components <item> <<component-list>>)
```

Templates are convenient for documenting facts and the expected fields that are found in them. They are not accepted by CLIPS as commands and should only be used for documenting CLIPS code.

7.7 ENTERING AND EXITING CLIPS

CLIPS can be entered by issuing the appropriate run command for the machine on which CLIPS has been installed. The CLIPS prompt should appear as follows.

```
CLIPS>
```

At this point, commands can be entered directly to CLIPS. This mode, in which direct commands can be entered, is called the **top-level**.

The normal mode of leaving CLIPS is with the **exit** command. The syntax of this command is

```
(exit)
```

Notice that the word *exit* is enclosed within matching parentheses. Many based languages draw their origins from LISP which uses parentheses as delimi Since CLIPS is based on a language that was originally developed using LISP machines, it retains these delimiters. The word *exit* without enclosing parentheses has quite a different meaning than the word *exit* with enclosing parentheses. The parentheses around *exit* indicate that *exit* is a command to be executed and not just the word *exit*. We'll see later that parentheses serve as important delimiters for commands.

For now, it is only important to remember that each CLIPS command must having a matching number of left and right parentheses and that the parentheses are properly balanced.

The final step in executing a CLIPS command after the command has been entered with properly balanced parentheses is to press the carriage return key. The carriage return key may also be pressed before or after any token has been entered. Pressing the carriage return key after entering the characters 'ex', but before entering the characters 'it' would create two tokens: a token for the word *ex* and a token for the word *it*.

The following command sequence demonstrates a sample session of entering CLIPS and then immediately exiting. The example shown is for an IBM PC™ using MS-DOS in which the CLIPS executable is stored on a disk in drive A and the current drive is also A. The name of the CLIPS executable is assumed to be CLIPS. Output displayed by MS-DOS or CLIPS is shown in regular type. All input that must be typed by you is shown in bold. The carriage return key is indicated by the character ↵. CLIPS is case sensitive, so it is important to type upper and lower case lette actly as they appear.

```
A>CLIPS↵
          CLIPS (V4.20 4/29/88)
CLIPS> (exit)↵
A>
```

7.8 ADDING AND REMOVING FACTS

The group of all facts known to CLIPS is stored in the fact-list. Facts representing information can be added and removed from the fact-list. New facts can be added to the fact-list using the **assert** command. The syntax of the *assert* command is

```
(assert <<<fact>>>)
```

As an example, let's consider a hypothetical expert system that monitors the activities of an industrial plant. This expert system will monitor and respond to a range of possible emergencies. One such emergency would be a fire. A fact representing that a fire emergency exists could be added to the fact-list by using the following command

```
CLIPS> (a: sert (emergency fire))↵
CLIPS>
```

The **facts** command can be used to display the facts in the fact-list. The basic syntax of the *facts* command is

```
(facts)
```

For example,

```
CLIPS> (facts)↵
f-1  (emergency fire)
CLIPS>
```

The term "f-1" is the **fact identifier** assigned to the fact by CLIPS. Every fact that is inserted into the fact-list is assigned a unique fact identifier starting with the letter f and followed by an integer called the **fact-index**.

CLIPS does not accept duplicate entries of a fact. Therefore, attempting to place a second (emergency fire) fact into the fact-list will have no result. Of course, other facts th 'o not duplicate existing facts can be easily added to the fact-list. For example,

```
CLIPS> (assert (emergency flood))↵
CLIPS> (facts)↵
f-1  (emergency fire)
f-2  (emergency flood)
CLIPS>
```

As the syntax of the *assert* command indicates, more than one fact can be asserted using a single *assert* command. For example, the command

```
CLIPS> (assert (emergency fire)↵
               (emergency flood))↵
CLIPS>
```

will assert two facts into the fact-list.

An important point to realize is that identifiers in the fact-list are not necessarily sequential. Just as facts can be added to the fact-list, there is also a way to remove facts. As facts are removed from the fact-list, the deleted fact identifiers will be missing in the fact-list. So the fact-identifiers may not be strictly sequential as a CLIPS program executes.

Since large numbers of facts can be contained in the fact-list, it is often useful to l .le view only a portion of the fact-list. This can be accomplished using additional arguments for the *facts* command. The complete syntax for the *facts* command is

```
(facts [<start> [<end> [<maximum>]]])
```

where <start>, <end>, and <maximum> are positive integers. Notice that the syntax the *facts* command allows from zero to three arguments. It no arguments are specifie all facts are displayed. If the <start> argument is specified, all facts with fact-index greater than or equal to <start> are displayed. If <start> and <end> are specified, facts with fact-indexes greater than or equal to <start> and less than or equal to <en are displayed. Finally, if <maximum> is specified along with <start> and <end>, more than <maximum> facts will be displayed.

Just as facts can be added to the fact-list, they can also be removed. Removir facts from the fact-list is called **retraction** and is done with the **retract** comman The syntax of the *retract* command is

```
(retract <<<fact-index>>>)
```

The fact indexes of one or more facts to be retracted are included as the arguments of tl *retract* command. For example, suppose the fact-list is

```
f-1  (emergency fire)
f-3  (emergency flood)
f-4  (action activate-sprinkler-system)
```

The command

```
(retract 3)
```

would remove the fact (emergency flood). Similarly, the command

```
(retract 1)
```

will retract the fact with identifier f-1 which is (emergency fire) and the command

```
(retract 4)
```

will retract the fact (action activate-sprinkler-system). Notice that to retract a fact , th fact-index and not the position of the fact in the fact-list must be specified. Attemptin to retract a non-existent fact will produce the following error message

```
Fact <fact-index> does not exist
```

For example,

```
CLIPS> (retract 4)↵
CLIPS> (retract 4)↵
```

```
Fact 4 does not exist
CLIPS>
```

A single *retract* command can be used to retract multiple facts at once. For example, the command

```
(retract 3 1 4)
```

will retract all three facts.

7.9 THE COMPONENTS OF A RULE

In order to accomplish useful work, an expert system must have rules as well as facts. Now that fact assertions and retractions have been discussed, it's possible to see how rules work.

Rules can be typed directly into CLIPS or they can be loaded in from a file of rules created by an editor. For information on how to use the editor of CLIPS, refer to Appendix F. Care must be taken when using a word processor to be sure that it does not introduce document formatting characters into the source code since such characters will not be accepted by CLIPS. For example, the non-document mode should be used with Wordstar™ on an IBM PC, and the SAVE AS... with text only option should be used on a Macintosh™ with MacWrite™.

Initially, the examples shown will be rules entered directly into CLIPS from the top-level. The pseudocode for one of the possible rules in the industrial plant monitoring expert system is shown following

```
IF the emergency is a fire
THEN the action is to activate the sprinkler system
```

Shown following is the rule expressed in CLIPS syntax. The rule can be entered by typing it in after the CLIPS prompt.

```
(defrule fire-emergency "An example rule"
   (emergency fire)
   =>
   (assert (action activate-sprinkler-system)))
```

If the rule is entered correctly as shown, then the CLIPS prompt reappears. Otherwise, an error message, most likely indicating a misspelled keyword or misplaced parenthesis, will appear.

Shown following is the same rule with comments added to match the parts of the rule. Comments begin with a semicolon and continue until a carriage return. Comments are ignored by CLIPS and will be discussed in greater detail later.

```
; Rule header
(defrule fire-emergency "An example rule"
   ; Patterns
   (emergency fire)
   ; THEN arrow
   =>
   ; Actions
   (assert (action activate-sprinkler-system)))
```

The general format of a rule is

```
(defrule <rule name> [<optional comment>]
   <<patterns>>    ; Left-Hand Side (LHS) of the rule
   =>
   <<actions>>)    ; Right-Hand Side (RHS) of the rule
```

The entire rule must be surrounded by parentheses. Each of the patterns and actior of the rule must be surrounded by parentheses. A rule may have multiple patterns an actions. The parentheses surrounding patterns and actions must be properly balanced they are nested. In the *fire-emergency* rule, there is one pattern and one action.

The header of the rule consists of three parts. The rule must start with **defrul** keyword. Following defrule must be the name of the rule. The name can b y vali CLIPS word. If a rule is entered with a rule name that is the same as an existing rul then the new rule replaces the old rule. In this rule, the rule name is *fire-emergenc* Next is an optional comment within double quotes. For this rule, the comment is "A example rule". The comment is normally used to describe the purpose of the rule o any other information the programmer desires. Unlike comments beginning with semicolon, the comment following the rule name is not ignored and can be displaye along with the rest of the rule (using the *ppru*le command introduced in Section 7.11).

After the rule header are zero or more **patterns** or conditional elements. Eac pattern consists of one or more fields. In the *fire-emergency* rule, the pattern i (emergency fire). The fields are *emergency* and *fire*. CLIPS attempts to match th patterns of rules against facts in the fact-list. If all the patterns of a rule match facts the rule is **activated** and put on the **agenda**, the collection of activated rules. Ther may be zero or more rules in the agenda. If a rule has no patterns, then the specia pattern (initial-fact) will be added as a pattern for the rule. The reason for using thi pattern will become clear when the *reset* command is discussed later.

The symbol => that follows the patterns in a rule is called an **arrow**. It is forme by typing the equal sign followed by the greater than sign. The arrow is a symbo representing the beginning of the THEN part of an IF-THEN rule. The part of the rul before the arrow is called the left-hand side (**LHS**) and the part after the arrow is calle the right-hand side (**RHS**).

The last part of a rule is the list of actions that will be executed whe e rul **fires**. A rule may have no actions. This is not particularly useful, but it can be done

In our example, the one action is to assert the fact (action activate-sprinkler-system). The term fires means that CLIPS executes the actions of a rule from the agenda. A program normally ceases execution when there are no rules on the agenda. When there are multiple rules on the agenda, CLIPS automatically determines which is the appropriate rule to fire. CLIPS orders the rules on the agenda in terms of increasing priority and fires the rule with the highest priority, called **salience**. Salience will be discussed in more detail in Chapter 9.

7.10 THE AGENDA AND EXECUTION

A CLIPS program can be made to run with the **run** command. The syntax of the *run* command is

```
(run [<limit>])
```

where the optional argument <limit> is the maximum number of rules to be fired. If <limit> is not included or <limit> is -1, then rules will be fired until no rules are left on the agenda. Otherwise, execution of rules will cease after <limit> number of rules ·e fired.

When a CLIPS program is run, the rule with the highest salience on the agenda is fired. If there is only one rule on the agenda, that rule will fire. Since the conditional element of the *fire-emergency* rule is satisfied by the fact (emergency fire), the *fire-emergency* rule should fire when the program is run.

If the *run* command is attempted when the *fire-emergency* rule has been entered after the fact (emergency fire), then the *fire-emergency* rule does not fire. The reason CLIPS does not fire the rule has to do with the way CLIPS is designed. The design of CLIPS is such that rules only see facts that have been entered *after* the rules. Thus newly entered rules will not "see" the facts that are currently on the fact-list. Only new facts that are entered will be seen by the rule. This means that a rule can only be activated by facts that are asserted after the rule is entered.

Activating the Rule

To activate the *fire-emergency* rule, the fact (emergency fire) must be asserted after the rule has been entered. At first thought, this seems trivial since the fact (emergency fire) just needs to be asserted. However, if the (emergency fire) fact is already in the fact-list, asserting a new (emergency fire) fact will have no effect. In order to re-assert this fact, the original version of the fact must be retracted. Once this is done, a new (emergency fire) fact can be asserted since no duplication will occur. This new version of the fact 'l have a different fact-index from the original and the *fire-emergency* rule will be ,ed on the agenda.

The list of rules on the agenda can be displayed with the **agenda** command. The syntax of the *agenda* command is

```
(agenda)
```

If no activations are on the agenda, the CLIPS prompt will reappear after th *agenda* command is issued. If the *fire-emergeny* rule had been activated by th (emergency fire) fact with a fact index of 2, an *agenda* command would produce th following output

```
CLIPS> (agenda)↵
0   fire-emergency  f-2
CLIPS>
```

The 0 indicates the salience of the rule on the agenda. After the salience comes th name of the rule followed by the fact identifiers that match the rule. In this case, ther is only one fact identifier, f-2.

Rules and Refraction

With the *fire-emergency* rule on the agenda, the *run* command will now cause the rul to fire and produce the following output

```
CLIPS> (run)↵
1 rules fired
Run time is 0.0167 seconds
CLIPS>
```

The message "1 rules fired" is an information message generated by the run command indicating the number of rules that were fired. The message following that indicates the amount of time taken to fire the rules. The timing message may not appear on all machines on which CLIPS runs, since some machines do not have internal clocks. The fact (action activate-sprinkler-system) will be added to the fact-list as the action of the rule.

An interesting question occurs at this point. What if the *run* command is issued again? There is a rule and there is a fact which satisfies the rule and so the rule should fire again. However, a *run* command attempted now will produce no results. Checking the agenda will verify that no rules are fired because there were no rules on the agenda.

The rule didn't fire again because of the way that CLIPS is designed. CLIPS was programmed with a characteristic of a nerve cell (**neuron**). After a neuron transmits a nerve impulse (fires), no amount of stimulation will make it fire for some time. This phenomenon is called **refraction** and is very important in expert systems.

Without refraction, expert systems would always be caught in trivial loops. That is, as soon as a rule fired, it would keep on firing on that same fact over and over again. In the real world, the stimulus that caused the firing would eventually disappear. For example, the fire would eventually be put out by the sprinkler system or n out

by itself. However, in the computer world, once a fact is entered in the fact-list, it stays there until explicitly removed.

If necessary, the rule be made to fire again by retracting the fact (emergency fire) and asserting it again. Basically, CLIPS remembers the fact identifiers that triggered a rule into firing and will not activate that rule again with the exact same combination of fact identifiers. Identical sets of fact identifiers must match one-for-one in both order and fact indices. In Chapter 8, examples will shown how a single fact can match a pattern in more than one way. In this case, several activations with the same set of fact identifiers for a single rule can be placed on the agenda, one for each distinct match.

7.11 COMMANDS USED WITH RULES

Displaying the Rules in the Knowledge Base

The **rules** command is used to display the current list of rules maintained by CLIPS. Its syntax is

```
(rules)
```

For example,

```
CLIPS> (rules)↵
fire-emergency
CLIPS>
```

The **pprule** (pretty print rule) command is used to display the text representation of a rule. Its syntax is

```
(pprule <rule-name>)
```

Its single argument specifies the name of the rule to be displayed. For example,

```
CLIPS> (pprule fire-emergency)↵
(defrule fire-emergency "An example rule"
   (emergency fire)
   =>
   (assert (action activate-sprinkler-system)))
CLIPS>
```

CLIPS puts different parts of the rule on different lines for the sake of readability. The patterns before the arrow are still considered the LHS and the actions after the arrow are still the RHS of the rule.

Loading Rules from a File

A file of rules made with a text editor can be loaded into CLIPS using the **load** command. The syntax of the *load* command is

```
(load <file-name>)
```

where <file-name> is a string containing the name of the file to be loaded.

Assuming that the fire-emergency rule was stored in a file called fire.clp on drive B of an IBM PC, then the following command would load the rule into CLIPS

```
(load "B:fire.clp")
```

Of course, the specification of a file name will be machine dependent and so this example should be taken only as a guide. A problem that may occur in loading is due to the backslash character used on some operating systems as a directory path separator. Since CLIPS interprets the backslash as an escape character, two backslashes must be used to create a single backslash in a string. For example, normally a pathname might be written as

```
B:\usr\clips\fire.clp
```

To preserve the backslash characters, however, the pathname would have to be written as shown in the following command

```
(load "B:\\usr\\clips\\fire.clp")
```

Rules do not have to all be kept in a single file. They can be stored in more than one file and loaded using several load commands.

Saving Rules to a File

CLIPS also provides the opposite of the *load* command. The **save** command allows the set of rules stored in CLIPS to be saved to a disk file. The syntax of the *save* command is

```
(save <file-name>)
```

For example, the following command will save the *fire-emergency* rule to a file called fire.clp on drive B

```
(save "B:fire.clp")
```

The *save* command will save all the rules in CLIPS to the specified file. It is not possible to save specified rules to a file. The *save* and *load* commands also save and load deffacts (which will be discussed shortly). Normally, if an editor is used to create and modify the rules, there is no need to use the *save* command since the rules will be saved while using the editor. Sometimes, however, it is convenient to enter rules directly at the CLIPS prompt and then save the rules to a file.

7.12 COMMENTING RULES

It's a good idea to include comments in a CLIPS program. Sometimes, rules can be difficult to understand and comments can be used to explain to the reader what the rule is doing. Comments are also used for good documentation of programs and will be very helpful in lengthy programs.

A comment in CLIPS is any text that begins with a semicolon and ends with a carriage return. The following is an example of comments in the fire program.

```
;**********************************
;*                                *
;* Programmer: G. D. Riley        *
;*                                *
;* Title    : The Fire Program    *
;*                                *
;* Date    : 10/31/88             *
;*                                *
;**********************************

; Facts used in this program use the
; following templates

; (emergency <type-of-emergency>)
; (action <action-to-be-performed>)

; The purpose of this rule is to activate
; the sprinkler system if there is a fire

(defrule fire-emergency "An example rule" ; IF
   ; there is a fire emergency
   (emergency fire)
   =>                                         ; THEN
   ; activate the sprinkler system
   (assert (action activate-the-sprinkler-system)))
```

Loading this rule into CLIPS and then pretty printing it with the *pprul* man will demonstrate that every comment starting with a semicolon is elimin... in th CLIPS program. The only comment that is retained is the one in quotes after the rul name. That is, CLIPS will ignore the semicolon comments as it's loading th program.

7.13 THE PRINTOUT COMMAND

Besides asserting facts in the RHS of rules, the RHS can also be used to print ou information using the **printout** command. The syntax of the *printout* command is

```
(printout <logical-name> <<print-items>>)
```

where <logical-name> indicates the output destination of the *printout* command an <<<print-items>>> are the zero or more items to be printed by this command.

The following rule demonstrates the use of the *printout* command

```
(defrule fire-emergency
   (emergency fire)
   =>
   (printout t "Activate the sprinkler system"
            crlf))
```

It is very important to include the letter **t** following the *printout* command as thi argument indicates the destination of the output. This destination is also referred to as **logical name**. In this case, the logical name *t* tells CLIPS to send the output to the **standard output device** of the computer. Generally, the standard output device i the terminal. However, this may be redefined so that the standard output device is som other device such as a modem or printer. In Chapter 10, the concept of logical name will be fully introduced and the *printout* command will be used for disk I/O.

The arguments following the logical name are items to be printed by the *printou* command. The string

```
"Activate the sprinkler system"
```

will be printed at the terminal without the enclosing quotes. The word **crlf** is treated specially by the *printout* command. It forces a carriage return/line feed. This is useful in improving the appearance of output by formatting it on different lines.

7.14 USING MULTIPLE RULES

Until now, only the simplest type of program consisting of just one rule has been shown. However, expert systems consisting of only one rule are not too useful. Practical expert systems may consist of hundreds or thousands of rules. In addition to the *fire-emergency* rule, the expert system monitoring the industrial plant might also include a rule for emergencies in which flooding has occurred. The expanded set of rules now looks like

```
(defrule fire-emergency
   (emergency fire)
   =>
   (printout t "Activate the sprinkler system"
               crlf))

(defrule flood-emergency
   (emergency flood)
   =>
   (printout t "Shut down electrical equipment"
               crlf))
```

Once these rules have been entered into CLIPS, asserting the fact (emergency fire) and then issuing a *run* command will produce the output "Activate the sprinkler system". Asserting the fact (emergency flood) and issuing a *run* command will produce the output "Shut down the electrical equipment."

Capturing the Real World in a Rule

As long as the fires and floods are the only two emergencies which must be handled, then the rules above would be sufficient. However, the real world is not quite that simple. For instance, not all fires can be extinguished using water. Some fires may require chemical extinguishers. What should be done if a fire produces poisonous gas or an explosion may occur? Should an off-site fire department be notified in addition to the on-site firemen? Does it matter which floor of the building the fire is on? Activating the water sprinklers on the second floor might cause water damage to both the first and second floors. Electrical power to equipment might have to be shut off on both floors. A fire on the first floor may only require that power be shut off to equipment on the first floor. If the building is flooded, are there watertight doors that can be shut to prevent damage? What should be done in case a plant break-in is detected? If all of these situations are included as rules, have all possibilities now been covered?

Unfortunately, the answer is no. In the real world, things don't always operate perfectly. Capturing all pertinent knowledge in an expert system can be quite difficult. In the best case, it may be possible to recognize most major emergencies and to

provide rules to allow the expert system to recognize when it doesn't r nize : emergency.

Rules with Multiple Patterns

Most real-world heuristics are too complicated to be expressed as a rule with just single pattern. For example, activating the sprinkler system for any type of fire migl not only be wrong—it could be dangerous. Fires involving ordinary combustibles suc as paper, wood, and cloth (class A fires) can be extinguished using water or water-base extinguishers. However, fires involving flammable and combustible liquids, grease, and similar materials (class B fires) must be extinguished using a different method suc as a carbon-dioxide extinguisher.

Rules with more than one pattern could be used to express these conditions. Th first pattern would determine that a fire emergency exists. The second pattern woul determine whether the fire was a class A fire or a class B fire. In addition, more rule could be used to determine whether a specific burning material constituted either a cla A or B fire. Two rules for class A and B fire emergencies are shown as follows

```
(defrule class-A-fire-emergency
   (emergency fire)
   (fire-class A)
   =>
   (printout t "Activate sprinkler system" crlf))

(defrule class-B-fire-emergency
   (emergency fire)
   (fire-class B)
   =>
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

Both rules have two patterns. Both patterns in each of the rules must be satisfie by facts in the fact-list for the rule to fire. Entering both rules and then asserting th facts (emergency fire) and (fire-class A) will cause the *class-A-fire-emergency* rule to b placed on the agenda. Running the system would then produce the output "Activat sprinkler system".

Any number of patterns can be placed in a rule. The important point to realize i that the rule is placed on the agenda only if *all* the patterns are satisfied by facts. Thi type of restriction is called a **logical AND**. Because the patterns are of the logica AND type, the rule will not fire if only one of the patterns is satisfied. All the fact must be present before the LHS of a rule is satisfied and the rule is placed on the agenda.

7.__ THE DEFFACTS CONSTRUCT

It is often very convenient to be able to automatically assert a set of facts instead of typing in the same assertions from the top-level. This is particularly true for facts which are known to be true before running a program (i.e., the initial knowledge). Running test cases to debug a program is another instance in which it is useful to automatically assert a group of facts. Groups of facts which represent initial knowledge can be defined using the define facts keyword, **deffacts**. For example, the following deffacts statement provides initial information about the type of emergency

```
(deffacts status "Some facts about the emergency"
   (emergency fire)
   (fire-class A))
```

The general format of a deffacts is

```
(deffacts <deffacts name> [<optional comment>]
   <<facts>>)
```

Following the deffacts keyword is the required name of this deffacts statement. Any v ymbolic field can be used as the name. In this case, the name chosen was *status*. Fo...wing the name is an optional comment in double quotes. Like the optional comment of a rule, this comment will be retained with the deffacts after it's been loaded by CLIPS. After the name or comment are the facts that will be asserted in the fact-list by this deffacts statement.

The facts in a deffacts statement are asserted using the CLIPS **reset** command. The syntax of the *reset* command is

```
(reset)
```

Assuming that the *status* deffacts had been entered, the following dialog shows how the *reset* command adds the facts to the fact-list

```
CLIPS> (reset)↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (emergency fire)
f-2     (fire-class A)
CLIPS>
```

The output shows the facts from the deffacts statement and a new fact generated by th et command called **initial-fact**. Upon startup, CLIPS automatically defines a deffacts statement with the following format

```
(deffacts initial-fact
   (initial-fact))
```

Thus, even if you have not defined any deffacts statements, a reset will assert tl fact (initial-fact). The fact-identifier of the initial-fact is always f-0. The utility ((initial-fact) lies in starting the execution of a program. A CLIPS program will n start running unless there are rules whose LHS are satisfied by facts. The *res* command asserts a default startup fact as well as asserting the facts in deffac statements. As stated previously, a rule without any patterns has the patter (initial-fact) added to its LHS. Thus any rule without LHS patterns will be placed o the agenda when a reset is performed.

The *reset* command is the key method for starting or restarting an expert system i CLIPS. The *reset* command removes all activated rules from the agenda and all fact from the fact-list. It then asserts the facts from existing deffacts statements. Normally this will cause the fact (initial-fact) to be asserted, since this is a fact found in th *initial-facts* deffacts provided for you by CLIPS.

There are some additional commands that are useful with deffacts. For exampl **list-deffacts** will list the names of deffacts that are currently loaded in CLIPS Another useful command is **ppdeffact** that prints the facts stored in a deffacts. Th syntax of these two commands are

```
(list-deffacts)
(ppdeffact <deffacts-name>)
```

For example

```
CLIPS> (list-deffacts)↵
initial-fact
status
CLIPS> (ppdeffact  status)↵
(deffacts status "Some facts about the emergency"
   (emergency fire)
   (fire-class A))
CLIPS>
```

7.16 REMOVING DEFFACTS AND RULES

Deffacts statements can be removed from CLIPS by using the **undeffacts** command. The syntax of this command is

```
(undeffacts <deffacts-name>)
```

For example, to remove the *status* deffacts, the command

```
(undeffacts status)
```

would be used. A reset now would only add the fact (initial-fact). The *undeffacts status* command has erased the status deffacts from memory. To regain a deffacts statement after an undeffacts command, the deffacts statement must be re-entered.

Even the *initial-fact* deffacts can be remove using undeffacts command as shown following.

```
(undeffacts initial-fact)
```

Now, whenever a *reset* command is performed, the initial-fact will not appear.

CLIPS also allows rules to be selectively removed by using the **excise** command. The syntax of the *excise* command is

```
(excise <rule-name>)
```

For example, to get rid of the *fire-emergency* rule, the following command with the name of the rule as an argument would be used.

```
(excise fire-emergency)
```

The **clear** command can be used to remove all information contained in the CLIPS environment. It restores the original state of CLIPS just as if it has just been started up. The syntax of the *clear* command is

```
(clear)
```

The *clear* command causes all of the facts in the fact-list to be removed as well as removing all rule and deffacts currently contained in CLIPS.

7.17 THE WATCH COMMAND

The **watch** command is useful for debugging programs. The syntax of this command is

```
(watch {facts,rules,activations,all})
```

Facts, rules, and activations may be watched in any combination to provide the appropriate amount of debugging information. The *watch* command can be used more than once to watch more than one feature of CLIPS execution. The word *all* can be used to enable all of the watch features (facts, rules, and activations).

If facts are being watched, CLIPS will automatically print a messag[illegible] licatir that an update has been made to the fact-list whenever facts are asserted or retracted. Tl following command dialog illustrates the use of this debugging command.

```
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (assert (emergency fire))↵
==> f-1   (emergency fire)
CLIPS> (assert (emergency flood))↵
==> f-2   (emergency flood)
CLIPS> (retract 2)
<== f-2   (emergency flood)
```

If activations are being watched, CLIPS will automatically print a messag whenever an activation has been added to or removed from the agenda. If rules are bein watched, CLIPS will print a message whenever a rule is fired. Note that watchin activations will not cause a message to be displayed when a rule is fired.

In addition, if the rules are being watched while loading a program, CLIPS wi display a message such as

```
Compiling rule: cut-power-to-electrical-fire +j+j+j
```

As the rules are being loaded, CLIPS is compiling the source code. The "+j+j+j" strin at the end of the message is some information from CLIPS about the internal structur of the compiled rules. This information will be useful to know for tuning a progran and will be discussed in Chapter 11 which deals with efficiency.

The effects of a *watch* command may be turned off by using the correspondin **unwatch** command. The syntax of the *unwatch* command is

```
(unwatch {facts,rules,activations,all})
```

7.18 THE MATCHES COMMAND

CLIPS has a debugging command called **matches** which indicates which patterns in a rule match facts. This capability is useful for debugging cases in which a rule appears to have all of its patterns satisfied, but is nonetheless not activated. Those patterns which do not match prevent the rule from becoming activated. One common reason that a pattern won't match a fact is misspelling an element either in the pattern or the assertion of the fact. The syntax of the *matches* command is

```
(matches <rule-name>)
```

The argument of the *matches* command is the name of the rule to be checked for matches. The following rule will be used to demonstrate the *matches* command.

```
; Class C Fires involve energized electrical
; equipment. Remove the power to such a fire
; if possible.

(defrule cut-power-to-electrical-fire
   ; Emergency is a fire
   (emergency fire)
   ; Fire involves energized electrical equipment
   (fire-class C)
   ; Power can be safely removed
   (power can-be-safely-removed)
   =>
   (printout t "Remove power from the fire" crlf))
```

The following command sequence shows how the *matches* command is used. Notice that the *watch facts* command is used to provide immediate feedback when facts asserted.

```
CLIPS> (watch facts)↵
CLIPS> (assert (emergency fire))↵
==> f-1     (emergency fire)
CLIPS> (matches cut-power-to-electrical-fire)↵
Matches for Pattern 1
 f-1
Matches for Pattern 2
 None
Matches for Pattern 3
 None
Partial matches for patterns 1 - 2
 None
Activations
 None
```

The *matches* command considers the patterns to be ordered sequentially as follows.

```
(defrule cut-power-to-electrical-fire
   (emergency fire)                      ;Pattern 1
   (fire-class C)                        ;Pattern 2
   (power can-be-safely-removed)         ;Pattern 3
   =>
   (printout t "Remove power from the fire" crlf))
```

The fact with fact-identifier f-1 matches the first pattern in the rule a: so reported by the *matches* command. Given that a rule has N patterns, the term **partia matches** refers to any set of matches of the first N-1 patterns with facts. That is, th partial matches begin with the first pattern in a rule and end with any pattern up to bu not including the last (N'th) pattern. As soon as one partial match can't be made CLIPS does not check any further. For example, a rule with four patterns would hav partial matches of the first and second patterns, and also the first, second and thir patterns. If all N patterns match, the rule will be activated.

The following commands continue this example

```
CLIPS> (assert (fire-class C))↲
=> f-2     (fire-class C)
CLIPS> (matches cut-power-to-electrical-fire)↲
Matches for Pattern 1
 f-1
Matches for Pattern 2
 f-2
Matches for Pattern 3
 None
Partial matches for patterns 1 - 2
 f-1  f-2
Activations
 None
```

Notice that with the second fact asserted, there is now a partial match for patterns 1 and 2. Of course, if the fact (power can-be-safely-removed) had been asserted instead of (fire-class C), there would have been no partial match for patterns 1 - 2.

This next command sequence asserts the final fact to match the third pattern in the rule.

```
CLIPS> (assert (power can-be-safely-removed))↲
=> f-3     (power can-be-safely-removed)
CLIPS> (matches cut-power-to-electrical-fire)↲
Matches for Pattern 1
 f-1
Matches for Pattern 2
 f-2
Matches for Pattern 3
 f-3
Partial matches for patterns 1 - 2
 f-1  f-2
Activations
 f-1  f-2  f-3
```

With the addition of the third and final pattern, the Activations information shows the fact-identifiers which cause an activation of the rule.

7.19 THE SET-BREAK COMMAND

CLIPS has a debugging command called **set-break** which allows execution to be halted before any rule from a specified group of rules is fired. A rule which halts execution before being fired is called a **breakpoint**. The syntax of the set-break command is

```
(set-break <rule-name>)
```

where <rule-name> is the name of a rule to make a breakpoint. As an example, consider the following rules

```
(defrule first
   =>
   (assert (fire second)))

(defrule second
   (fire second)
   =>
   (assert (fire third)))

(defrule third
   (fire third)
   =>)
```

The following command dialog shows execution of the rules without any breakpoints set.

```
CLIPS> (watch all)↵
CLIPS> (reset)↵
==> f-0      (initial-fact)
==> Activation 0      first: f-0
CLIPS> (run)↵
FIRE    1 first: f-0
==> f-1      (fire second)
==> Activation 0      second: f-1
FIRE    2 second: f-1
==> f-2      (fire third)
==> Activation 0      third: f-2
FIRE    3 third: f-2
```

```
3 rules fired
CLIPS>
```

All three rules fire in succession when the *run* command is issued. The following command dialog demonstrates the use of the *set-break* command to halt execution.

```
CLIPS> (set-break second)↵
CLIPS> (set-break third)↵
CLIPS> (watch all)↵
CLIPS> (reset)↵
<== Activation 0      first: f-0
==> f-0      (initial-fact)
==> Activation 0      first: f-0
CLIPS> (run)↵
FIRE    1 first: f-0
==> f-1      (fire second)
==> Activation 0      second: f-1
Breaking on rule second
1 rules fired
CLIPS> (run)↵
FIRE    1 second: f-1
==> f-2      (fire third)
==> Activation 0      third: f-2
Breaking on rule third
1 rules fired
CLIPS> (run)↵
FIRE    1 third: f-2
1 rules fired
CLIPS>
```

In this case, execution halts before the rules *second* and *third* are allowed to fire. Notice that at least one rule must be fired by the *run* command before a breakpoint will stop execution. For example, after the rule *second* has halted execution, it does not halt execution again when the *run* command is given.

The **show-breaks** command can be used to list all breakpoints. Its syntax is

```
(show-breaks)
```

The **remove-break** command can be used to remove breakpoints. Its syntax is

```
(remove-break [<rule-name>])
```

If <rule-name> is provided as an argument, only the breakpoint for that rule wi removed. Otherwise, all breakpoints will be removed.

7.20 SUMMARY

This chapter introduced the fundamental components of CLIPS. Facts are the first component of a CLIPS system. Facts are made up of fields which are either a word, string, or number. The first field of a fact is normally used to indicate the type of information stored in a fact and is called a relation. Fact templates can be used to document the type of information that is stored in a fact. Commands for adding, removing and displaying facts were discussed. The deffacts construct can be used to specify facts as initial knowledge.

Rules are the second component of a CLIPS system. A rule is divided into a LHS and a RHS. The LHS of a rule can be thought of as the IF portion and the RHS can be thought of as the THEN portion of a rule. Rules can have multiple patterns and actions. Commands are available for displaying the list of rules and the text of an individual rule.

The third component of CLIPS is the inference engine. Rules that have their patterns satisfied by facts produce an activation which is placed on the agenda. Refraction prevents rules from constantly being activated by old facts.

Fact assertions and retractions, rule firings, and activations can be traced by using the *watch* command. The *matches* command will display the facts that have matched the patterns of a rule as well as the list of partial matches for a rule. The *set-break* command allows execution to be halted before a rule is fired. The *printout* command can be used to output information from the RHS of a rule. The *clear* command is used to re-initialize the state of the CLIPS environment.

PROBLEMS AND PROGRAMS

7-1. Convert the following sentences to facts in a deffacts statement. For each group of related facts, build a fact template which describes a more general relationship.

```
The father of John is Tom.
The mother of John is Susan.
The parents of John are Tom and Susan.
Tom is a father.
Susan is a mother.
John is a son.
Tom is a male.
Susan is a female.
John is a male.
```

Describe a fact template for a fact containing information about a set. The [illegible] template should include information about the name or description of the set, the list of elements in the set and whether it is a subset of another set. Represent the following sets as facts using the format specified by your fact templates.

```
A = { 1, 2, 3 }
B = { 1, 2, 3, 4, 5 }
C = { red, green, yellow, blue }
D = { CPU, memory, I/O, address bus,
      data bus, video display }
```

7-3. A sparsely populated array is an array that contains relatively few non-ze elements. It is more efficiently represented as a linked list or tree. How might sparsely populated array be represented using facts? Describe the templates used for t facts to represent the array. What are the possible disadvantages of representing an arra using facts as opposed to using an array data structure in a procedural language?

7-4. Convert the general net representing airline routes as shown in Figure 2-4 (to a series of facts in a deffacts statement. Use a single template to describe the facts.

7-5. Convert the semantic net representing a family as shown in Figure 2-4 (b) a series of facts in a deffacts statement. Use several templates to describe the fac produced.

7-6. How might facts be used to represent Object-Attribute-Value triplets an Attribute-Value pairs? Give examples for the sentences in Problem 7-1.

7-7. Convert the semantic net shown in Figure 2-5 to a series of facts in a deffact statement. Use several templates to describe the facts. For example, the IS-A links an AKO links should be relation names and each should have its own template.

7-8. Convert the binary decision tree representing animal classificatio information of Figure 3-3 to a series of facts in a deffacts statement. Show how th links between the nodes can be represented. Do the leafs of the tree require a differen representation than the other nodes of the tree?

7-9. Explain the output from the *matches* command for the following comman sequence.

```
CLIPS> (assert (b))
CLIPS> (defrule matcher (a) (b) (c) =>)
CLIPS> (assert (a) (c))
CLIPS> (matches matcher)
```

7-10. Convert the decision tree shown in Figure 3-3 to a series of CLIPS rules Create patterns to match facts using the template (question <query-string> <answer>) For example, if the answer to the question represented by the root node in the tree wa no, the fact representing this information would be (question "Is it very ' no)

What information that was contained in the decision tree is now lost by representing the tree as rules in this manner?

7-11. Plants require many different types of nutrients for proper growth. Three of the most important plant nutrients that are provided by fertilizer are nitrogen, phosphorus, and potassium. A deficiency in one of these nutrients will produce various symptoms. Translate the following heuristics below to rules which determine nutrient deficiency. Assume that the plant is normally green.

```
A plant with stunted growth may have a nitrogen
   deficiency.
A plant that is pale yellow in color may have a
   nitrogen deficiency.
A plant that has reddish-brown leaf edges may have a
   nitrogen deficiency.
A plant with stunted root growth may have a
   phosphorus deficiency.
A plant with a spindly stalk may have a phosphorus
   deficiency.
A plant that is purplish in color may have a
   phosphorus deficiency.
A plant that has delayed in maturing may have a
   phosphorus deficiency.
A plant with leaf edges that appear scorched may have
   a potassium deficiency.
A plant with weakened stems may have a potassium
   deficiency.
A plant with shriveled seeds or fruits may have a
   potassium deficiency.
```

Include comments which define the templates used for the facts in the rules. The input to the program should be made by asserting symptoms as facts. The output should indicate which nutrient deficiencies exist by printing to the terminal. Implement a method so that multiple printouts for a single deficiency caused by more than one symptom are avoided. Test your program with the following inputs:

```
The plant has stunted root growth.
The plant is purplish in color.
```

7-12. Fires are classified according to their principal burning material. Translate the following information to rules for determining fire class.

```
Type A fires involve ordinary combustibles such as
   paper, wood, and cloth.
```

```
Type B fires involve flammable and combustible
   liquids, greases, and similar materials.
Type C fires involve energized electrical equipment.
Type D fires involve combustible metals such as
   magnesium, sodium, potassium, titanium, and
   zirconium.
```

The type of extinguisher that should be used on a fire depends upon the fire class. Translate the following information to rules.

```
Class A fires should be extinguished with
   heat-absorbing or combustion-retarding
   extinguishers such as water or water based liquids
   and dry chemicals.
Class B fires should be extinguished by excluding
   air, inhibiting the release of combustible vapors,
   or interrupting the combustion chain reaction.
   Extinguishers include dry chemicals, carbon
   dioxide, foam, and bromotrifluoromethane.
Class C fires should be extinguished with a
   non-conducting agent to prevent shock. If
   possible the power should be cut. Extinguishe.
   include dry chemicals, carbon dioxide, and
   bromotrifluoromethane.
Class D fires should be extinguished with smothering
   and heat-absorbing chemicals that do not react
   with the burning metals. Such chemicals include
   trimethoxyboroxine and screened graphitized coke.
```

Include comments which define the templates used for the facts in the rules. The input to the program should be made by asserting the type of burning material as a fact. The output should indicate which extinguishers may be used on the fire and other actions that should be taken such as cutting off the power. Show that your program works for a material from each of the fire classes.

CHAPTER 8 PATTERN MATCHING

8.1 INTRODUCTION

The type of rules shown in Chapter 7 illustrate simple pattern matching of patterns to facts. This chapter will introduce several concepts which provide very powerful capabilities for matching and manipulating facts. The first concept will be the use of variables in pattern matching. Secondly, field constraints will be discussed and the use of functions in CLIPS will then be introduced and basic arithmetic will be covered.

8.2 VARIABLES

Just as with other programming languages, CLIPS has **variables** available to store values. Variables in CLIPS are always written in the syntax of a question mark followed by a symbolic field name. Variable names follow the syntax of a word with the exception that they must begin with a character. For good programming style, variables should be given meaningful names. Some examples of variables are as follows.

```
?speed
?sensor
?value
?noun
?color
```

Spaces should not be used between the question mark and symbolic field name. As will be discussed later, a question mark by itself has its own use.

Before a variable can be used, it must be assigned a value. The following rule shows a case in which the variable ?x is used in a printout command without having been assigned a value.

```
(defrule test
   (initial-fact)
   =>
   (printout t ?x crlf))
```

In the above example, notice how initial-fact is used to trigger the rule. This is convenient since (initial-fact) can be asserted by simply using the *reset* command. In fact, for this rule, it is not necessary to place the (initial-fact) pattern on the LHS, since a rule with no patterns automatically has the (initial-fact) pattern added to its LHS.

When the rule is entered, CLIPS will respond with the error message

```
Undefined variable x referenced on rhs of rule
```

CLIPS gives an error message because it cannot find a value which has been id to ?x. The terms **bound** and **bind** are used to describe the assignment of a value to a variable.

One common use of variables is to bind a value on the LHS and then assert the bound variable on the RHS. The following rule demonstrates this use

```
(defrule grandfather
   (is-a-grandfather ?name)
   =>
   (assert (is-a-man ?name)))
```

If the fact (is-a-grandfather Jack) is asserted and the program is run, the grandfather rule will assert the fact (is-a-man Jack) because the variable ?name was bound to Jack.

Of course, a variable can be used more than once as the following rule shows

```
(defrule grandfather
   (is-a-grandfather ?name)
   =>
   (assert (is-a-father ?name)
           (is-a-man ?name)))
```

If the fact (is-a-grandfather Jack) is asserted and the rule is fired, the facts (is-a- Jack) and (is-a-man Jack) will be placed in the fact-list since the variable ?name is used twice.

Variables are also commonly used in printing output, as in

```
(defrule grandfather
   (is-a-grandfather ?name)
   =>
   (printout t ?name " is a grandfather" crlf))
```

If the fact (is-a-grandfather Jack) is asserted and the rule is executed, the following output would be produced

```
Jack is a grandfather
```

8.3 FACT ADDRESSES AND RETRACTION

Retraction is very useful in expert systems and is usually done on the RHS rather than in the top-level. Before a fact can be retracted, it must be specified to CLIPS. To retract a fact from a rule, the **fact-address** must first be bound to a variable on the LHS.

There is a big difference between binding a variable to the contents of a fact binding a variable to the fact-address. In the examples of patterns previously used, such

as (is-a-grandfather ?name), a variable was bound to the value of a field. That is, ?name was bound to Jack. However, if the fact whose contents are (is-a-grandfather Jack) is to be removed, then CLIPS must know the address of the fact to be retracted.

The address of the fact is specified using the **left arrow** operator, "<-". This operator is made by typing a "<" symbol followed by a "-".

Shown following is an example of fact retraction from a rule

```
(defrule modify-grandfather-fact
   ?old-fact <- (is-a-grandfather Jack)
   =>
   (retract ?old-fact)
   (assert (has-a-grandchild Jack)
           (is-a-man Jack)))
```

If the fact (is-a-grandfather Jack) is asserted and the rule is executed, then the fact (is-a-grandfather Jack) will be retracted from the fact-list and the two facts (has-a-grandchild Jack) and (is-a-man Jack) will be asserted into the fact-list. The pattern in the rule assigns the address of (is-a-grandfather Jack) to the variable ?old-fact. The *retract* command then removes the fact (is-a-grandfather Jack) whose address was gned to ?old-fact.

Variables can be used to pick up a fact value at the same time as an address. For example

```
(defrule modify-grandfather-fact
   ?old-fact <- (is-a-grandfather ?name)
   =>
   (retract ?old-fact)
   (assert (has-a-grandchild ?name)
           (is-a-man ?name)))
```

The above rule will fire when ?name is assigned to any field. For example, if the facts (is-a-grandfather Brian), (is-a-grandfather Steve), and (is-a-grandfather Gary) are asserted into the fact-list, the rule will produce an activation for each of the three facts.

The *assert* command can be used with *retract* command to produce some interesting effects. The following rule demonstrates a simple looping effect. It can be activated by asserting the fact (loop-fact).

```
(defrule simple-loop
   ?old-fact <- (loop-fact)
   =>
   (printout t "Looping" crlf)
   (retract ?old-fact)
   (assert (loop-fact)))
```

The program executes an infinite loop since it asserts a new (loop-fact) a[illegible] re-tracting the fact bound to ?old-fact. Note that to make a new (loop-fact) the individual fields must be asserted. CLIPS is not designed to allow assertions of entire facts with statements like (assert (?old-fact)). After the new (loop-fact) is asserted, the rule then fires on the new (loop-fact).

The only way to stop an infinite loop like this is by interrupting CLIPS using a Control-C or other appropriate interrupt command for the computer on which CLIPS is running. In extreme cases on personal computers it may be necessary to reboot and then restart CLIPS.

The program will not produce an infinite loop if the *retract* command is removed since the attempted assertions would then be duplicates of the fact already in memory.

8.4 MULTIPLE USE OF VARIABLES

The *modify-grandfather-fact* rule can only be activated by a fact with two fields. A fact matching the template

```
(is-a-grandfather <grandfather> <grandchild>)
```

such as (is-a-grandfather Jack John), will not match the rule because it contain[illegible]ee fields. In order to match against a template containing the name of both the gran[illegible]r and the grandchild, the rule must be modified to match against three fields. One approach to modify the rule might be to use two variables called ?name to match the two name fields in (is-a-grandfather Jack John).

```
(defrule modify-grandfather-fact
   ?old-fact <- (is-a-grandfather ?name ?name)
   =>
   (retract ?old-fact)
   (assert (has-a-grandchild ?name ?name)
           (is-a-man ?name)))
```

However, this rule won't fire on (is-a-grandfather Jack John) because the pattern (is-a-grandfather ?name ?name) specifies that the two fields must be the same since the same variable ?name is used for both fields.

The way to bind both names is to use two different variables, as follows

```
(defrule modify-grandfather-fact
   ?old-fact <- (is-a-grandfather ?grandfather
                                  ?grandchild)
   =>
   (retract ?old-fact)
   (assert (has-a-grandchild ?grandfather
```

```
                              ?grandchild)
            (is-a-man ?grandfather)))
```

This rule will become activated when the fact (is-a-grandfather Jack John) is asserted.

This rule will also fire if Jack has a grandchild named after him. That is, the fact (is-a-grandfather Jack Jack) will also cause the rule to become activated. Different variables are allowed to have the same values.

Notice that the pattern (is-a-grandfather ?grandfather ?grandchild) matched any two symbolic fields for the variables while (is-a-grandfather ?name ?name) required that the two fields bound to the variables were the same.

Variables that are used in multiple patterns have an important and useful property. The first time that a variable is bound to a value, it retains that value within the rule. The following rule and deffacts demonstrate this principle. Notice that the same variable, ?father, is used in both patterns of the *determine-grandfather* rule.

```
(defrule determine-grandfather
   (the-father-of ?grandchild is ?father)
   (the-father-of ?father is ?grandfather)
   =>
   (printout t "The grandfather of " ?grandchild
               " is " ?grandfather crlf))

(deffacts father-of-relations
   (the-father-of Jill is John)
   (the-father-of Bob is John)
   (the-father-of John is Jack)
   (the-father-of Bill is Fred)
   (the-father-of Sally is Fred)
   (the-father-of Fred is Gene)
```

When CLIPS is reset and the facts from the deffacts construct are asserted, the first pattern will match all six facts. The variable ?father will be bound to the names John, John, Jack, Fred, Fred, and Gene respectively for each of the six facts. When CLIPS tries to match the second pattern of the rule, only the facts with a second field bound to one of these values will be able to match. Of the six facts, only the fact (the-father-of John is Jack) and (the-father-of Fred is Gene) will be able to match the second pattern for facts which matched the first pattern.

8.5 SINGLE FIELD WILDCARDS

.her than having to specify a specific field of a fact to trigger a rule, a general pattern can be specified by using a **wildcard**. For example, suppose a database of information

exists about the eye and hair color of a group of people. Each fact in the database is specified by the following template

```
(person <name> <eye-color> <hair-color>)
```

and the database could be the described by the deffacts

```
(deffacts persons
   (person John brown black)
   (person Jill blue blond)
   (person Jack green brown)
   (person Jane brown brown)
   (person Fred green red))
```

where, for example, the fact (person John brown black) indicates that John has brown eyes and black hair. Rules can be written to search the fact-list for people meeting certain specifications. One such specification could be to write a rule which finds all people who have brown hair. For this specification, the color of the person's eyes is unimportant.

This type of situation in which only part of the fact is needed to determine a match is very common. To solve this problem, a wildcard can be used for the fields are unimportant.

The simplest form of wildcard is called a **single-field wildcard** and is shown by a question mark, "?". A single-field wildcard stands for one field. The following rule shows how a wildcard can be used with the previous deffacts to find all people with brown hair.

```
(defrule find-brown-haired-people
   (person ?name ? brown)
   =>
   (printout t ?name " has brown hair" crlf))
```

The pattern includes a wildcard to indicate that the color of the person's eyes is not important. So long as the hair color of the person is brown, the rule will be satisfied and fire.

Notice now the importance of not including a space between the question mark and the symbolic name of a variable. Doing this to the pattern

```
(person ?name ? brown)
```

would produce the pattern

```
(person ? name ? brown)
```

which is completely different. The first pattern would match a fact with four fields. The first field of the fact would be the word *person* and the fourth field of the fact would be the word *brown*. The second pattern would match a fact with five fields whose first field was the word *person*, third field was the word *name*, and fifth field was the word *brown*.

Suppose it is desired to match against a fact with the following template

```
(person <name> <eye-color> <hair-color>
        <nationality>)
```

to find all persons with brown hair. Since this template has an additional field, an additional wildcard must be added to the previous pattern. Thus the pattern

```
(person ?name ? brown ?)
```

would have to be used. The important point to remember is that each field of the pattern must match to a field in the fact.

8.6 BLOCKS WORLD

n example to demonstrate variables bindings, a program to move blocks in a simple blocks world will be built. This type of program is analogous to the classic blocks world program in which the knowledge domain is restricted to blocks [Firebaugh 88]. An example like this is a good example of planning and might be applied to automated manufacturing where a robot arm is manipulating parts.

The only thing of interest in blocks world are blocks. A single block may be stacked upon another block. The goal of a complex blocks world program would be to rearrange the stacks of block into a goal configuration with the minimum number of moves. For this example, a number of simplifying restrictions will be made. The first of these restrictions is that only one primary goal is allowed and this goal can only be to move one block on top of another block. With this restriction, it is rather trivial to determine the optimal moves to achieve the goal. If the goal is to move block x on top of block y, then move all blocks (if any) on top of block x to the floor and all blocks (if any) on top of block y to the floor and then move block x on top of block y.

The second restriction will be that any goal must not have already been achieved. That is, the goal cannot be to move block x on top of block y if block x is already on top of block y. This is a rather simple condition to check, however, the appropriate syntax to test for this condition is not introduced until Chapter 9.

To begin to solve this problem, it will be useful to set up a configuration of blocks which can be used for testing the program. Figure 8-1 shows the configuration [illegible] will be used. There are two stacks in this configuration. The first stack has block [illegible] top of block B on top of block C. The second stack has block D on top of block E on top of block F.

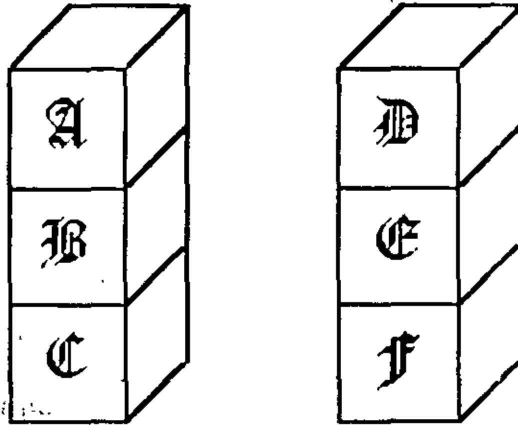


Figure 8-1

To determine which type of rules would be useful in solving the problem, a step by step completion of a blocks world goal would be useful. What steps must be taken to move block C on top of block E? The easiest solution for this problem would be to directly move block C on top of block E. This rule, however, could only be applied if both block C and block E had no blocks on top of them. The pseudo code for this rule would be

```
RULE MOVE-DIRECTLY
IF   The goal is to move block ?upper on top of
       block ?lower and
     block ?upper is the top block in its stack and
     block ?lower is the top block in its stack,
THEN Move block ?upper on top of block ?lower.
```

The *move-directly* rule cannot be used in this case, however, since blocks A and B are on top of block C, and block D is on top of block E. In order to allow the *move-directly* rule to move block C on top of block E, blocks A, B, and D must be moved to the floor. The blocks are moved to the floor since this is the easiest step to take to get them out of the way. The simple blocks world does not require that the blocks be restacked and only a single primary goal is allowed, so there is no need to stack blocks when they are moved out of the way. This rule can be expressed as two pseudocode rules: one rule to clear blocks off the block to be moved and one rule to clear blocks off the block to be stacked on.

```
RULE CLEAR-UPPER-BLOCK
IF   The goal is to move block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the flo
```

```
RULE CLEAR-LOWER-BLOCK
IF   The goal is to move another block on top of
       block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor
```

The *clear-upper-block* rule will work to clear the blocks off block C. It will first determine that block B needs to be moved to the floor. In order to move block B to the floor, this same rule will determine that block A needs to be moved to the floor. Similarly, the *clear-lower-block* rule will determine that block D needs to be moved to the floor in order to move something on top of block E.

Now there are goals to move blocks A, B, and D to the floor. Blocks A and D can be moved directly to the floor. If written properly, the *move-directly* rule might be able to handle moving blocks on top of the floor as well as other blocks. However, since the floor is really not a block, it may be necessary to treat the floor differently. The following pseudocode rule will handle the special case of moving a block to the floor

```
RULE MOVE-TO-FLOOR
IF   The goal is to move block ?upper on top of the
       floor and
     block ?upper is the top block in its stack,
THEN Move block ?upper on top of the floor.
```

The *move-to-floor* rule can now move blocks A and D to the floor. Once block A is moved to the floor, the *move-to-floor* rule can be activated to move block B to the floor. With blocks A, B, and D on the floor, blocks C and E are now the top blocks in their stacks and it is possible to use the *move-directly* rule to move block C on top of block E.

Now that the rules have been written using pseudocode, the facts to be used by the rules should be determined. The types of facts needed cannot always be determined without some prototyping. In this case, however, the pseudocode rules point out several types of facts that will be needed. For example, the information about which blocks are on top of other blocks is crucial. This information could be described with the following template

```
(on-top-of <upper-block> <lower-block>)
```

and the facts described by this template would be

```
(on-top-of A B)
(on-top-of B C)
(on-top-of D E)
(on-top-of E F)
```

Since it is also important to know which block is at the top of a stack and which block is at the bottom of a stack, it would be useful to include the facts

```
(on-top-of nothing A)
(on-top-of C floor)
(on-top-of nothing D)
(on-top-of F floor)
```

The words *nothing* and *floor* have special meaning in these facts. The facts (on-top-of nothing A) and (on-top-of nothing D) indicate that A and D are the top blocks in their stacks. Similarly, the facts (on-top-of C floor) and (on-top-of F floor) indicate that blocks A and F are the bottom blocks in their stacks. Including these facts, however, does not necessarily solve the problem of determining the top and bottom blocks in a stack. If the rules are not written correctly, the words *floor* and *nothing* might be mistaken as the names of blocks. Facts that indicate the name of the blocks might be useful. The following template

```
(block <name>)
```

can be used to identify the blocks from the special words *nothing* and *floor*. The facts described by this template would be

```
(block A)
(block B)
(block C)
(block D)
(block E)
(block F)
```

Finally, a fact is needed to describe the block moving goals that are being processed. These goals could be described with the template

```
(move-goal <block> on-top-of <target-destination>)
```

and the initial goal using this template would be

```
(move-goal C on-top-of E)
```

Note that the word *on-top-of* has been included in the template. This word serves no other purpose in the template and fact other than providing increased readability. The target destination in the move goal could also be the floor.

With the facts and templates now defined, the initial configuration of the blocks world can be described by the following deffacts.

```
(deffacts initial-state
   (block A)
   (block B)
   (block C)
   (block D)
   (block E)
   (block F)
   (on-top-of nothing A)
   (on-top-of A B)
   (on-top-of B C)
   (on-top-of C floor)
   (on-top-of nothing D)
   (on-top-of D E)
   (on-top-of E F)
   (on-top-of F floor)
   (move-goal C on-top-of E))
```

Notice that information is included in the *on-top-of* facts to indicate which blocks are the top blocks and which blocks are the bottom blocks. The *block* facts help dis sh the blocks from the words *floor* and *nothing*. The goal for the program is to mov block C on top of block E.

The *move-directly* rule is written as follows

```
(defrule move-directly
   ?goal <- (move-goal ?block1 on-top-of ?block2)
   (block ?block1)
   (block ?block2)
   (on-top-of nothing ?block1)
   ?stack-1 <- (on-top-of ?block1 ?block3)
   ?stack-2 <- (on-top-of nothing ?block2)
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (on-top-of ?block1 ?block2))
   (assert (on-top-of nothing ?block3))
   (printout t ?block1 " moved on top of " ?block2
               "." crlf))
```

The first three patterns determine that there is a goal to move a block on top of another block. Patterns two and three insure that a goal to move a block onto the floor will not be processed by this rule. The fourth and sixth patterns check that the blocks are top blocks in their stacks. The fifth pattern matches against information necessary to update the stack from which the moving block is being taken. The actions of the rule update the stack information for the two stacks and prints a message. The

block beneath the moved block is now the top block in that stack, and the bloc :ing moved is now the top block in the stack to which it was moved.

The *move-to-floor* rule is implemented as follows

```
(defrule move-to-floor
   ?goal <- (move-goal ?block1 on-top-of floor)
   (block ?block1)
   (on-top-of nothing ?block1)
   ?stack <- (on-top-of ?block1 ?block2)
   =>
   (retract ?goal ?stack)
   (assert (on-top-of ?block1 floor))
   (assert (on-top-of nothing ?block2))
   (printout t ?block1 " moved on top of floor."
              crlf))
```

This rule is similar to the *move-directly* rule with the exception that it is not necessary to update some information about the floor since it is not a block.

The *clear-upper-block* rule is implemented as follows

```
(defrule clear-upper-block
   (move-goal ?block1 on-top-of ?)
   (block ?block1)
   (on-top-of ?block2 ?block1)
   (block ?block2)
   =>
   (assert (move-goal ?block2 on-top-of floor)))
```

The *clear-lower-block* rule is implemented as follows

```
(defrule clear-lower-block
   (move-goal ? on-top-of ?block1)
   (block ?block1)
   (on-top-of ?block2 ?block1)
   (block ?block2)
   =>
   (assert (move-goal ?block2 on-top-of floor)))
```

The program is now complete with the *move-directly*, *move-to-floor*, *clear-upper-block*, and *clear-lower-block* rules and the *initial-state* deffacts. The following output shows a sample run of this blocks world program.

```
CLIPS> (reset)↵
CLIPS> (run)↵
```

```
A moved on top of floor.
B moved on top of floor.
D moved on top of floor.
C moved on top of E.
7 rules fired
CLIPS>
```

Blocks A and B are first moved to the floor to clear block C. Block D is then moved to the floor to clear block E. Finally, block C can be moved on top of block E to solve the primary goal.

This example has demonstrated building a program using a step by step method. First, pseudorules were written using English-like text. Second, the pseudorules were used to determine the types of facts that would be required. Templates describing the facts were designed, and the initial knowledge for the program was coded using these templates. Finally, the pseudorules were translated to CLIPS rules using the fact templates as a guide for translation.

Typically, the development of an expert system requires a great deal more prototyping and iterative development than was shown in developing this example. It is not always possible to determine the best method for representing facts or the types of rules th[illegible]ill be needed to build an expert system. Following a consistent methodology, h[illegible]er, can aid in the development of an expert system even when a great deal of prototyping and iteration needs to be performed.

8.7 MULTIFIELD WILDCARDS AND VARIABLES

Multifields

Multifield wildcards and variables can be used to match against zero or more fields of a pattern. The **multifield wildcard** is a dollar sign followed by a question mark, "$?", and represents zero or more occurrences of a field. Note that ordinary variables and wildcards match exactly one field. This is a slight, but very important difference. The following rule and deffacts show how multifield wildcards can be used to find all occurrences of facts that represent a list of items, regardless of the size of the list.

```
(defrule find-list
   (list $?)
   =>
   (printout t "Found a list" crlf))

(deffacts different-lists
   (list a)
   (list b d)
   (list a b c)
```

```
(list a c c e)
(list c d b))
```

This program, when run, will fire the *find-list* rule five times since there are five *list* facts.

Wildcards have another important use because they can be attached to a symbolic field to create a variable such as ?x, $?x, ?name, $?name. Single field variables are preceded by a "?", while multifield variables are preceded by a "$?".

As an example of a multifield variable, the following version of the *find-list* rule also prints out the item field(s) of the matching fact because a variable is equated to the item field(s) that match.

```
(defrule find-list
   (list $?items)
   =>
   (printout t "Found a list with items: " $?items
               crlf))
```

When this rule is fired, the items found in the lists will be printed. The multifield wildcard can match any number of fields. No matter how many item fields that a list has, the multifield wildcard will match against all of them.

Multifield variables can also be used to find all lists that contain a particu em field, even if the position of the item field is not known in advance. Suppose we want to find all *list* facts containing the item c. The following facts from the *different-lists* deffacts all contain the item c

```
(list a b c)
(list a c c e)
(list c d b)
```

Notice that the fact (list a c c e) would actually match twice since there are two c's in the item list. This represents a case in which two activations of the same rule are placed on the agenda with the exact same fact identifiers. Normally, CLIPS will not allow this to happen, but in this case, the facts have been matched in different ways because multifield variables were used.

The following rule will match all facts containing the item c and print out the item fields found before and after the item c.

```
(defrule find-item-c
   (list $?before c $?after)
   =>
   (printout t "Items before c: " $?before crlf)
   (printout t "Items after c: " $?after crlf))
```

The pattern matches any *list* facts that have an item c anywhere in them.

Single and multifield wildcards can be combined. For example, the pattern

```
(list ? $? c ?)
```

means that the first and last fields can be anything, and that the field just before the last field must be the word *c*. The pattern also requires that the matching fact must have at least four fields. Table 8-1 shows which of the facts found in the *different-lists* deffacts will match this pattern.

Fact	*Matches*
(list c)	No
(list c d)	No
(list a c e)	Yes
(list a c d b)	No
(list a d c b)	Yes

Table 8-1
Matches for the Pattern (list ? $? c ?)

N that the $? can match zero or more occurrences of a field. Matching zero occurrences means that there can be a match with no fields. So a pattern like (list $?items) will match any *list* fact. In contrast, the ? requires a field to match.

Although multifield variables can be essential for pattern matching in many cases, their overuse can cause a lot of inefficiency because of increased memory requirements and slower execution. As a general rule of style, $? should only be used when the length of the fields being matched against is not known. The $? should not be used as a convenience to avoid typing several single field wildcards. Chapter 11, which discusses efficiency, has more details on the efficient use of multifield wildcards and variables.

Implementing a Stack

A **stack** is an ordered data structure to which items can be added and removed. Items are added and removed to one "end" of the stack. A new item can be pushed (added) to the stack or the last item added can be popped (removed) from the stack. In a stack, the first value added is the last item to be removed and the last item added is the first item to be removed.

A useful analogy for a stack is a set of cafeteria trays. New trays are added (pushed) on the top of the trays already in the stack. The last trays added to the top of the stack will be the first trays removed (popped).

 is relatively easy to implement a stack capable of performing push and pop ope ons using multifield variables. First of all, a *stack* fact similar to the *list* facts previously discussed can be used. The following rule will push a value onto the *stack* fact.

```
(defrule push-value
   ?push-value <- (push-value ?value)
   ?stack <- (stack $?rest)
   =>
   (retract ?push-value ?stack)
   (assert (stack ?value $?rest))
   (printout t "Pushing value " ?value crlf))
```

Two rules are necessary to implement the pop action: one for an empty stack and another if the stack has a value to pop.

```
(defrule pop-value-valid
   ?pop-value <- (pop-value)
   ?stack <- (stack ?value $?rest)
   =>
   (retract ?pop-value ?stack)
   (assert (stack $?rest))
   (printout t "Popping value " ?value crlf))

(defrule pop-value-invalid
   ?pop-value <- (pop-value)
   (stack)
   =>
   (retract ?pop-value)
   (printout t "Popping from empty stack" crlf))
```

These rules could easily be changed to push and pop values to named stacks. For example, the patterns

```
?push-value <- (push-value ?value)
?stack <- (stack $?rest)
```

could be replaced with

```
?push-value <- (push-value ?name ?value)
?stack <- (stack ?name $?rest)
```

where ?name represents the name of the stack.

Blocks World Revisited

Using multifield variables and wildcards, the blocks world problem can be reimplemented in a much easier fashion. Each stack can be represented by a sing ct as

shown below. The operations on moving the blocks are very similar to those used in the push/pop example.

```
(deffacts initial-state
   (stack A B C)
   (stack D E F)
   (move-goal C on-top-of E)
   (stack))
```

The empty *stack* fact is included to prevent this fact from being added later, for example, when a stack has one block in it and that block is moved on top of another stack.

The rules for the blocks world program using multifield variables are shown below.

```
(defrule move-directly
   ?goal <- (move-goal ?block1 on-top-of ?block2)
   ?stack-1 <- (stack ?block1 $?rest1)
   ?stack-2 <- (stack ?block2 $?rest2)
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (stack $?rest1))
   (assert (stack ?block1 ?block2 $?rest2))
   (printout t ?block1 " moved on top of "
               ?block2 "." crlf))

(defrule move-to-floor
   ?goal <- (move-goal ?block1 on-top-of floor)
   ?stack-1 <- (stack ?block1 $?rest)
   =>
   (retract ?goal ?stack-1)
   (assert (stack ?block1))
   (assert (stack $?rest))
   (printout t ?block1 " moved on top of floor."
               crlf))

(defrule clear-upper-block
   (move-goal ?block1 on-top-of ?)
   (stack ?top $? ?block1 $?)
   =>
   (assert (move-goal ?top on-top-of floor)))

(defrule clear-lower-block
   (move-goal ? on-top-of ?block1)
   (stack ?top $? ?block1 $?)
```

```
   =>
   (assert (move-goal ?top on-top-of floor)))
```

8.8 FIELD CONSTRAINTS

The NOT Field Constraint

Chapter 7 demonstrated elementary pattern matching using the industrial plant monitoring system example. In addition to those elementary pattern matching capabilities, CLIPS has more powerful pattern matching operators. These additional pattern matching capabilities will be introduced by reconsidering the problem of determining groups of people with certain hair and eye colors. Suppose, for example, it is necessary to find all people who do not have brown hair. One way to do this is to write rules for each type of hair color. For example, the following rule finds people with black hair.

```
(defrule black-hair-is-not-brown-hair
   (person ?name ? black)
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

Another rule would cover people having blonde hair.

```
(defrule blonde-hair-is-not-brown-hair
   (person ?name ? blonde)
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

Yet another rule could be written to find people having red hair. The problem with writing the rules in this manner, however, is that the condition being checked is that the hair is not brown. The previous rules attempt to test this condition in a round-about manner. That is, determine all the hair colors other than brown and write a rule for each one. If all the colors can be specified, then this technique will work. For this example, however, it would be simpler to assume that hair color could be anything (even purple or green).

A way of handling this case is to use a **field constraint** to restrict the values a field may have on the LHS. The field constraint put limits on the values a field may have on the LHS. The field constraint acts like operators on patterns.

One type of field constraint is called a **logical constraint**. There are three types of logical constraints. The first type of logical constraint is called a **NO**

constraint. Its symbol is the **tilde,** "~". The NOT constraint acts on the one value that immediately follows it. It prevents that value from matching a field of the pattern.

The rule that looks for people without brown hair can be written much more easily using the NOT constraint as shown in the following rule.

```
(defrule person-without-brown-hair
   (person ?name ? ~brown)
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

By using the NOT constraint, this one rule does the work of many others that required specifying each hair color.

The OR Field Constraint

The second logical constraint is the **OR constraint**, represented by the bar "|". The OR constraint is used to allow one or more possible values to match a field of a pattern.

For example, the following rule finds all people with either black or brown hair ıg the OR constraint.

```
(defrule black-or-brown-hair
   (person ?name ? brown | black)
   =>
   (printout t ?name " has dark hair" crlf))
```

Asserting the facts (person Joe blue brown) and (person Mark brown black) would place an instantiation for each of the facts on the agenda.

The AND Field Constraint

The third type of logical constraint is the **AND constraint**. The AND constraint is different from the logical AND discussed in Chapter 7. The symbol of the AND constraint is the ampersand, "&". The AND constraint is normally used only with the other constraints since otherwise it's not of much practical use.

One case in which the AND constraint is useful is to place additional constraints with the binding instance of a variable. Suppose, for example, that a rule is triggered by a *person* fact with hair color of brown or black. The pattern to find such a fact can be expressed easily using the OR constraint as shown in the previous example. ᴴ~wever, how can the value of the hair color be identified? The solution is to bind a able to the color that is matched using the AND constraint and then print out the variable. This is a case in which the AND constraint is useful, as shown following.

```
(defrule black-or-brown-hair
   (person ?name ? ?color&brown|black)
   =>
   (printout t ?name " has " ?color " hair" crlf))
```

The variable ?color will be bound to whatever color is matched by the brown|black field.

The AND constraint is also useful with the NOT constraint. For example, the following rule triggers when a person's hair color is not black and not brown.

```
defrule black-or-brown-hair
   (person ?name ? ?color&~brown&~black)
   =>
   (printout t ?name " has " ?color " hair" crlf))
```

The following rule illustrates a case in which the AND constraint is *not* useful.

```
(defrule person-with-black-and-blonde
   (person ?name ? black&blonde)
   =>
   (printout t ?name " has black and blonde hair"
               crlf))
```

This rule will never be triggered since there is no possible fact whose hair color value is both black and blonde (assuming, of course, that a person's natural hair color can only be one color). The pattern will not match on black, blonde, or even blackblonde.

The following rule illustrates another case where the AND constraint is not very useful.

```
(defrule person-with-black-hair
   (person ?name ? black&~blonde)
   =>
   (printout t ?name " has black hair" crlf))
```

The pattern will match a person whose hair color is black and not blonde. In this case, the rule can be triggered by the fact (person Mark brown black) since black is black and not blonde. However, it's much more efficient to just write the rule as

```
(defrule person-with-black-hair
   (person ?name ? black)
   =>
   (printout t ?name " has black hair" crlf))
```

since hair that is black cannot be blonde anyway.

Combining Field Constraints

Field constraints can be mixed together with variables and other literal values to provide very powerful pattern matching capabilities. Suppose, for example, a rule is needed which determines if two people exist with the following conditions. The first person has either blue or green eyes and does not have black hair. The second person does not have the same color eyes as the first person and has either red hair or the same color hair as the first person. The following rule will match these constraints.

```
(defrule complex-eye-hair-match
   (person ?name1 ?eyes1&blue|green ?hair1&~black)
   (person ?name2&~?name1 ?eyes2&~?eyes1
           ?hair1&red|?hair1)
   =>
   (printout t ?name1 " has " ?eyes1 " eyes and "
               ?hair1 " hair" crlf)
   (printout t ?name2 " has " ?eyes2 " eyes and "
               ?hair2 " hair" crlf))
```

This example is worth studying in some detail. The field ?eyes1&blue|green binds the first person's eye color to the variable ?eyes1 if the eye color value of the fact being matched is either blue or green. The field ?hair1&~black binds the variable ?hair1 if the hair color value of the fact being matched is not black.

The pattern ?name2&~?name1 performs a very useful operation. It binds the name value of the person fact to the variable ?name2 if it is not the same as the value of ?name1. If the names are unique tags, this will insure that both patterns do not match the same fact. For these two patterns, this cannot happen since the eye color of the second person must be different from the eye color of the first person. However, it is a useful technique to understand and has many useful applications. Chapter 12 will show how this same technique can be accomplished using fact addresses to make sure that the facts are different. This will allow the rule to work for people with the same names, but different hair and eye colors.

The pattern field ?eyes2&~?eyes1 performs much the same test as the previous field. The final field, ?hair2&red|?hair1, binds the hair color value of the second person to the variable ?hair2 if the hair color is either red or the same value as the variable ?hair1. Note that a variable must already be bound if it used as part of an OR field constraint.

Variables will be bound only if they are the first condition in a field and only if they occur singly or are tied to the other conditions by an AND field constraint. For nple, the following rule

```
(defrule bad-variable-use
   (person ?name ? red|?hair)
   =>
   (printout t ?name " has " ?hair " hair " crlf))
```

will produce an error since ?hair is not bound.

As a final note for combining constraints, keep in mind the combinations of constraints that perform no useful purpose. As shown previously, using the AND constraint to tie together literal constants will always produce a matching failure unless the constants are identical. Similarly, tying together negated literals using the OR constraint will always succeed in a pattern match if the literals are different.

8.9 FUNCTIONS AND EXPRESSIONS

Elementary Math Functions

Besides dealing with symbolic facts, CLIPS can also perform calculations. However, keep in mind that an expert system language like CLIPS is not designed for number-crunching. Although the math functions of CLIPS are very powerful, they are primarily meant for modification of numbers that are being reasoned about by the application program. Other languages such as FORTRAN are bet' 'or number-crunching in which little or no reasoning about the numbers is done. .'S provides the elementary arithmetic operators as shown in Table 8-2. All arithmetic is done in floating-point and all numbers are stored in floating-point.

Arithmetic Operators	*Meaning*
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Table 8-2 CLIPS Elementary Arithmetic Operators

Numeric expressions are represented in CLIPS according to the style of LISP. In LISP and CLIPS, a numeric expression that would customarily be written as 2 + 3 must be written in **prefix form**, (+ 2 3). The customary way of writing numeric expressions is called **infix form** because the math operators are in between the **operands** or **arguments**. In the prefix form of CLIPS, the operator must go before the operands and parentheses must surround the numeric expression.

It is relatively easy to convert from infix to prefix format. For example, suppose two points are to be checked to see if they have a positive slope. In the customary x way, this can be written as

```
(y2 - y1) / (x2 - x1)   >   0
```

Note that the greater than symbol, >, is a CLIPS function which will be introduced later. It is used to determine if its first argument is greater than its second argument. In order to write this expression in prefix form, it will be useful to start by thinking of the numerator as (Y) and the denominator as (X). So the expression above can be written as

```
(Y) / (X)   >   0
```

The prefix form for division is then

```
(/ (Y)  (X))
```

since in prefix the operator comes before the argument. Now the result of the division must be tested to determine if it is greater than 0. So the prefix form is as follows.

```
(> (/ (Y)  (X)) 0))
```

In infix form, Y = y2 - y1. But the prefix form needs to be used, since a prefix expression is being created. So (- y2 y1) will be used for (Y) and (- x2 x1) for (X). Replacing (Y) and (X) by their prefix forms will yield the final expression of whether the two points have a positive slope.

```
(> (/ (- y2 y1)  (- x2 x1)) 0)
```

The simplest way to evaluate a numeric expression in CLIPS (as well as any other expression) is to evaluate the expression at the top-level prompt. For example, entering (+ 2 2) at the CLIPS prompt would produce the following output.

```
CLIPS> (+ 2 2)↵
4
CLIPS>
```

The output shows the correct response of 4. In general, any CLIPS expression can be entered at the top level to be evaluated. In fact, many of the commands already discussed such as the *facts* and *agenda* commands are simply functions that, when evaluated, produce output as a side effect. Most functions, such as the addition function, have a **return value**. This return value can be a number, word, string, or even a multifield value. Other functions, such as the *facts* and *agenda* commands have no return value. Functions without a return value typically have what is called a **side effect**. The side effect of the *facts* command is to list the facts in the fact-list.

The other arithmetic functions also work at the top-level. The following output demonstrates the evaluation of some other expressions.

```
CLIPS> (+ 2 3)↵
5
CLIPS> (- 2 3)↵
-1
CLIPS> (* 2 3)↵
6
CLIPS> (/ 2 3)↵
0.66666669
CLIPS> (** 2 3)↵
8
CLIPS>
```

Note that the answer for division will probably show a round-off error in the last digit. This result may vary from machine to machine.

Variable Numbers of Arguments

Prefix notation allows variable numbers of arguments to be represented quite simply. Many CLIPS functions accept a variable number of arguments. The arguments in a numeric expression can be extended beyond two for all operators except exponentiation. The same sequence of arithmetic calculations is performed for more than two arguments. The following examples entered at the top level shows how three argu ; are used. Evaluation proceeds from left to right.

```
CLIPS> (+ 2 3 4)↵
9
CLIPS> (- 2 3 4)↵
-5
CLIPS> (* 2 3 4)↵
24
CLIPS> (/ 2 3 4)↵
0.16666667
CLIPS>
```

Once again note that the answer for division may differ slightly depending upon the machine being used.

Precedence and Nesting Expressions

One important fact about CLIPS and LISP calculations is that there is no built-in precedence of arithmetic operations. In other computer languages, multiplication and division rank higher than addition and subtraction and the computer does the higher ranked operations first. However, in LISP and CLIPS, there is no built-in pre(ıce.

Everything is simply evaluated from left to right with parentheses determining precedence.

Mixed calculations can be done in prefix notation. For example, suppose the following infix expression is to be evaluated

```
2 + 3 * 4
```

The customary evaluation is to multiply 3 by 4 and then add the result to 2. However, in CLIPS, the precedence must be explicitly written. The expression could be evaluated by entering the following at the top level

```
CLIPS> (+ 2 (* 3 4))↵
14
CLIPS>
```

In this rule, the expression in the innermost parentheses is evaluated first and so 3 is multiplied by 4. The result is then added to 2. If the desired evaluation is (2 + 3) * 4, where again addition is done first, the top level expression would be

```
CLIPS> (* (+ 2 3) 4)↵
20
CLIPS>
```

Using Expressions Within Asserts

In general, expressions may be freely embedded within other expressions. It would therefore seem natural to use an expression within an assert in the following manner

```
CLIPS> (assert (answer (+ 2 2)))↵
CLIPS>
```

The expected result would be the addition of the fact (answer 4) to the fact-list. Attempting this assertion, however, will produce an error. To understand this error, one must look back to the origin of CLIPS. The syntax of CLIPS is based very closely upon the syntax of ART. ART allows the use of embedded lists within facts. For example, the following facts are all valid ART facts:

```
(tools (hammer wrench pliers))
(names (Paul Tom Harry) (Martha Anne Jane Janet))
(answer (+ 2 2))
```

All of these facts are invalid in CLIPS since they contain embedded lists. Notice the fact (answer (+ 2 2)). Since this is a valid fact with an embedded list, namely (+ 2 2), how does ART specify an expression to be evaluated? The answer is that ART uses

an "escape" character to indicate that a list is an expression to be evaluated and an embedded list. To maintain compatibility, CLIPS uses this same escape character even though embedded lists are not allowed in CLIPS.

The escape character used to signify an expression to be evaluated is the "=" operator. The operator may also be associated with the words "equal to the value of.". For example, the following command

```
CLIPS> (assert (answer =(+ 2 2)))↵
CLIPS>
```

will correctly assert the fact (answer 4) because the = operator uses the value of 4 returned by the expression (+ 2 2). Note that the = operator is only used within *assert* commands (and sometimes within patterns as will be shown in Chapter 9), and it should only be used at the beginning of an expression to be evaluated (never within an expression).

8.10 SUMMING VALUES USING RULES

As a simple example of using functions to perform calculations, consider the problem of summing up the area of a group of rectangles. The height and width of the rectangles can be specified using the following template.

```
(rectangle <height> <width>)
```

and the sum of rectangle areas could be specified with the following template.

```
(sum <total-area>)
```

A deffacts containing sample information is

```
(deffacts initial-information
   (rectangle 10 6)
   (rectangle 7 5)
   (rectangle 6 8)
   (rectangle 2 5)
   (sum 0))
```

An initial attempt to produce a rule to sum the rectangle might be as follows:

```
(defrule sum-rectangles
   (rectangle ?height ?width)
   ?sum <- (sum ?total)
   =>
```

```
   (retract ?sum)
   (assert (sum =(+ ?total (* ?height ?width)))))
```

This rule, however, will loop endlessly. Retracting the *sum* fact and then reasserting it will produce a loop with a single *rectangle* fact. One solution to solve the problem would be to retract the *rectangle* fact after its area was added to the *sum* fact. This would prevent the rule from firing off the same *rectangle* fact with a different *sum* fact. If the *rectangle* fact needs to be preserved, however, a different approach is required. A temporary fact containing the area to be added to the sum is created for each *rectangle* fact. This temporary fact can then be retracted which prevents an endless loop. The modified program is as follows:

```
(defrule sum-rectangles
   (rectangle ?height ?width)
   =>
   (assert (add-to-sum =(* ?height ?width))))

(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (assert (sum =(+ ?total ?area))))
```

The following output shows how these two rules interact to sum the areas of the rectangles.

```
CLIPS> (watch all)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (rectangle 10 6)
==> Activation 0      sum-rectangles: f-1
==> f-2     (rectangle 7 5)
==> Activation 0      sum-rectangles: f-2
==> f-3     (rectangle 6 8)
==> Activation 0      sum-rectangles: f-3
==> f-4     (rectangle 2 5)
==> Activation 0      sum-rectangles: f-4
==> f-5     (sum 0)
CLIPS> (run)↵
FIRE    1 sum-rectangles: f-4
==> f-6     (add-to-sum 10)
==> Activation 0      sum-areas: f-5,f-6
FIRE    2 sum-areas: f-5,f-6
```

```
<== f-5       (sum 0)
<== f-6       (add-to-sum 10)
==> f-7       (sum 10)
FIRE     3 sum-rectangles: f-3
==> f-8       (add-to-sum 48)
==> Activation 0         sum-areas: f-7,f-8
FIRE     4 sum-areas: f-7,f-8
<== f-7       (sum 10)
<== f-8       (add-to-sum 48)
==> f-9       (sum 58)
FIRE     5 sum-rectangles: f-2
==> f-10      (add-to-sum 35)
==> Activation 0         sum-areas: f-9,f-10
FIRE     6 sum-areas: f-9,f-10
<== f-9       (sum 58)
<== f-10      (add-to-sum 35)
==> f-11      (sum 93)
FIRE     7 sum-rectangles: f-1
==> f-12      (add-to-sum 60)
==> Activation 0         sum-areas: f-11,f-12
FIRE     8 sum-areas: f-11,f-12
<== f-11      (sum 93)
<== f-12      (add-to-sum 60)
==> f-13      (sum 153)
8 rules fired
CLIPS>
```

The *sum-rectangles* rule is activated four times when the *rectangle* facts are asserted as a result of the *reset* command. Each time the *sum-rectangles* rule is fired, it asserts a fact which activates the *sum-areas* rule. The *sum-areas* rules adds the area to the running total and removes the *add-to-sum* fact. Since the *sum-rectangles* rule does not pattern match against the *sum* fact, it is not reactivated when a new *sum* fact is asserted.

8.11 THE BIND FUNCTION

Quite often it is useful to store a value in a temporary variable to avoid recalculation. Avoiding recalculation can often be crucial when functions produce side effects. The **bind** function can be used to bind the value of a variable to the value of an expression. The syntax of the *bind* function is

```
(bind <variable> <value>)
```

The bound variable, <variable>, can be either a multifield variable or a single field variable. The new value, <value>, should either be a single field value or an expression. If <value> is an expression, its return value should be a single field value for a single field variable. A multifield variable can accept either a single field or a multifield return value. For example, the *sum-areas* rule could printout the total sum and the area being added to it for each rectangle.

```
(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (printout t "Adding " ?area " to " ?total crlf)
   (printout t "New sum is " (+ ?total ?area) crlf)
   (assert (sum =(+ ?total ?area))))
```

Notice that the expression (+ ?total ?area) can be embedded within the *printout* command and that it does not require the = operator. The expression is used twice in the RHS of the rule. Replacing the two separate evaluations with one *bind* function would 'nate unnecessary calculations. The rule rewritten with the *bind* function is

```
(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (printout t "Adding " ?area " to " ?total crlf)
   (bind ?new-total (+ ?total ?area))
   (printout t "New sum is " ?new-total crlf)
   (assert (sum ?new-total)))
```

In addition to creating new variables for use on the RHS of a rule, the *bind* function can also be used to rebind the value of a variable used in the LHS of a rule.

8.12 USING FUNCTIONS ON THE RHS

Commands that are entered at the top-level prompt and actions performed from the RHS of a rule are all CLIPS functions. Commands have been used to describe functions that are primarily entered at the top-level prompt and actions are used to describe 'ions that are primarily used on the RHS of a rule. In general, any function which . be entered as a command at the top-level can also be used on the RHS of a rule as an action and vice-versa. The only major exception are the defrule and deffacts

constructs which cannot be used on the RHS of a rule. The following is an e: ple of a rule which displays the facts in the fact-list as its RHS action.

```
(defrule print-facts
   =>
   (facts))
```

Note that some commands such as *reset*, *run*, *clear*, and *excise* are dangerous to use on the RHS because they may modify the current state of an executing rule. For example, a rule may be deleted in the process of executing the actions on its RHS, and so the results are unpredictable.

8.13 SUMMARY

In this chapter, the concept of variables was introduced. The use of variables for retrieving information from 'acts was discussed as well as the use of variables for pattern matching on the LHS of a rule. Variables can be used to store the fact addresses of patterns on the LHS of a rule so that the fact bound to the pattern can be retracted on the RHS of the rule. Single field wildcards can be used in place of variables when the field to be matched against can be anything and its value is not needed later in " LHS or RHS of the rule. Multifield variables and wildcards allow more than one in a pattern to be matched against.

Field constraints allow the negation and combination of more than one constraint for a given field. The NOT field constraint is used to prevent matching against certain values. The AND field constraint is used to guarantee that all of a series of matching conditions are true. The OR field constraint is used to guarantee that at least one of a series of matching conditions are true.

Functions can be entered into the CLIPS top level command loop or used on the LHS or RHS of a rule. Many functions, such as some of the arithmetic functions, can have a variable number of arguments. Function calls can be nested within other function calls. The bind command allows variables to be bound on the RHS of a rule.

PROBLEMS AND PROGRAMS

8-1. Given the following fact templates for facts describing a family tree

```
(father-of <father> <child>)
(mother-of <mother> <child>)
(male <person>)
(female <person>)
(wife-of <wife> <husband>)
(husband-of <husband> <wife>)
```

write rules which will infer the following relations

a) Uncle, aunt
b) Cousin
c) Grandparent
d) Grandfather, grandmother
e) Sister, brother
f) Ancestor

8-2. An industrial plant has ten sensors with id numbers 1 through 10. Each sensor has either good or bad status. Build a template for representing the sensors and write one or more rules which will print a warning message if three or more sensors have bad status. Test your rules with sensors 3 and 5 bad, sensors 2, 8 and 9 bad, and sensors 1, 3, 5 and 10 bad. What must be done to prevent the warning message from being displayed multiple times?

8-3. Given facts which describe the hair and eye color of a person as shown by the following fact template

```
(person <name> <eye-color> <hair-color>)
```

and a fact describing a hair or eye color being sought as shown by the following fact template

```
(looking-for <eye-color> <hair-color>)
```

write one or more rules which will

a) Print a message indicating that a person has either the same eye color or same hair color that is being sought.
b) Print a message indicating that a person has both the same eye color and hair color being sought.

8-4. Rewrite the stack rules found in Section 8.7 to work with named stacks.

8-5. In a stack, the first value added is the last value to be removed and the last value added is the first value to be remove. A queue works in the opposite manner. The first value added is the first value removed and the last value added is the last value removed. Write rules which will add and remove values to a queue. Assume that only one queue exists.

Write one or more rules which will generate all of the permutations of a base fact and print them out. For example, the fact

```
(base-fact red green blue)
```

should generate the output

```
Permutation is red green blue
Permutation is red blue green
Permutation is green red blue
Permutation is green blue red
Permutation is blue red green
Permutation is blue green red
```

8-7. Given sets represented using the following fact template

```
(set <name> <<elements>>)
```

write one or more rules which will

a) Compute the union of two sets given a fact of the format (find-union <set-1> <set-2>).
b) Compute the intersection of two sets given a fact of the format (find-intersection <set-1> <set-2>).

Note that when computing the union and intersection, duplicate elements should not be allowed to appear in the union or intersection of the sets.

8-8. Given a series of facts describing shapes with the following templates

```
(square <id-name> <side-length>)
(rectangle <id-name> <width> <length>)
(circle <id-name> <radius>)
(triangle <id-name> <base-length> <height>)
```

write one or more rules which will compute the sum of

a) The area of the shapes.
b) The perimeter of the shapes.

Test the output of the rules with the following deffacts

```
(deffacts test-8-8
   (square A 3)
   (square B 5)
   (rectangle C 5 7)
   (circle D 2)
```

```
(circle E 6)
(triangle F 3 4))
```

8-9. Given information about the name, eye color, hair color, and nationality of a person from a group in the following format

```
(person <name> <eye-color> <hair-color>
       <nationality>)
```

write one rule which will identify

a) Anyone with blue or green eyes who has brown hair and is from France.
b) Anyone who does not have blue eyes or black hair and does not have the same color hair and eyes.
c) Two people. The first having brown or blue eyes, not having blond hair, and a German nationality. The second having green eyes and the same hair color as the first person. The second person's eyes may be brown if the first person's hair is brown.

10. Convert the following infix expressions to prefix expressions.

a) `(3 + 4) * (5 + 6) + 7`
b) `(5 * (5 + 6 + 7)) - ((3 * (4 / 9) + 2) / 8)`
c) `6 - 9 * 8 / 3 + 4 - (8 - 2 - 3) * 6 / 7`

8-11. Write rules which will take a finite state machine from its present state to its next state given a fact of the form

```
(input <value>)
```

The state machine and its arcs should be represented as facts. The input fact should be retracted when the machine goes to its next state. Test your rules and fact representations on the finite state machines shown in Figures 3-5 and 3-6.

8-12. Write a set of rules for classifying syllogisms by mood and figure. For example, the syllogism

```
No M is P
Some M is not S
∴ Some S is P
```

of type EOI-3. The input for the rules should be facts representing the major and minor premises and the conclusion. The output should a printed statement of the mood and figure.

8-13. Create rules for implementing the decision procedure for determinin; a syllogism is valid. Test your program on the syllogism from Problem 8-12.

BIBLIOGRAPHY

Morris W. Firebaugh, *Artificial Intelligence: A Knowledge-Based Approach*, boyd & fraser Pub. Co., pp. 224 - 226, 1988.

CHAPTER 9 CONTROL TECHNIQUES

9.1 INTRODUCTION

This chapter introduces various techniques for controlling the execution of rules. Groups of rules are used to build more powerful and complex programs than have been shown in previous chapters. Techniques for input, comparing values, and generating loops will be demonstrated. Rule salience is introduced as a method for prioritizing rules, and techniques for specifying control knowledge using rules is demonstrated. Finally, logical pattern operators are introduced. These operators allow a single rule to perform the function of several rules and provide the capability to allow matching against the absence of facts.

9.2 THE GAME OF STICKS

A simple two player game called Sticks will be used as an example in this chapter to demonstrate various techniques of control in a rule-based language. The object of Sticks … avoid being forced to take the last stick in a pile of sticks. Each player may take …, or 3 sticks on their turn. The trick (or heuristic) to winning this game can be found by noticing that you can force your opponent to lose if it is their turn and there are 2, 3, or 4 sticks remaining. Thus a player who has 5 sticks remaining on their move has lost. To force the other player to 5 sticks, always leave a number of sticks equal to a multiple of 4 plus 1 at the end of your turn. At the end of your turn, force the pile to 5, 9, 13, etc. sticks. If you move first and the pile is set to one of the "losing" numbers, then you cannot win unless your opponent makes a mistake. If you move first and the pile is not set to a "losing" number, then you can always win.

9.3 INPUT TECHNIQUES

Before beginning to play Sticks, the program must determine some information. The program will play against a human opponent, so it must be determined who will get the first move. In addition, the starting size of the pile must be determined. This information could be placed in a deffacts construct. However, it is quite simple to ask the program's opponent for it by input from the keyboard. CLIPS allows information to be read from the keyboard using the **read** function. The basic syntax of the *read* function requires no arguments. The following example shows how the *read* function is used to input data.

```
(deffacts initial-phase
   (phase choose-player))
```

```
(defrule player-select
   (phase choose-player)
   =>
   (printout t "Who moves first (Computer: c "
               "Human: h)? ")
   (assert (player-select =(read))))

(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))
```

Both rules use the pattern (phase choose-player) to indicate their applicability only when a specific fact is in the fact-list. This type of pattern is called a **control pattern** because it is specifically used to control when the rule is applicable. A **control fact** is used to trigger a control pattern. Since the control pattern for these rules contains only literal fields, the control fact must match the pattern exactly. In this case, the control fact to trigger these rules must be the fact (phase choose-player). This control fact will be useful for error correction when the input received by the read function does not match the expected values of "c" or "h" for "computer" and "human". By retracting and re-asserting the control fact, the *player-select* rule can be retriggered.

The following output shows how the *player-select* and *good-player-choice* rules work to determine who should move first.

```
CLIPS> (watch facts)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (phase choose-player)
CLIPS> (run)↵
Who moves first (Computer: c Human: h)? c↵
==> f-2     (player-select c)
<== f-1     (phase choose-player)
<== f-2     (player-select c)
==> f-3     (player-move c)
2 rules fired
CLIPS>
```

Notice that the *read* function requires a carriage return before it will read the token entered. The *read* function can only be used to input a single field at a time. All extra characters entered after the first field up to the carriage return are discarded. For example, if the *player-select* rule tried to read both the first player and the number of sticks in the pile with the following input,

```
c 15↵
```

only the first field, "c", will be read. To read all of the input, both fields must be enclosed within double quotes. Of course, once the input is within double quotes, it is a single literal field. The individual fields "c" and 15 cannot be easily accessed.

The read function allows fields that are not words, strings, or numbers, such as parentheses, to be entered. Such fields are placed within double quotes and treated as strings. The following command line dialog demonstrates this capability

```
CLIPS> (read)↵
(↵
"("
CLIPS>
```

Other input functions are also available, including a function which allows an entire line of data to be read as a string. In addition, information can also be written to and read from a file. These capabilities will be discussed in greater detail in Chapter 10.

The preceding rules work properly if the correct response of "c" or "h" is entered, but if an incorrect response is entered no error checking is performed. There are many uations in which an input request should be repeated to correct faulty input. One way programming an input request loop is shown in the following example. The following rules use the control fact (phase choose-player) to indicate that a player selection needs to be made.

```
(deffacts initial-phase
   (phase choose-player))

(defrule player-select
   (phase choose-player)
   =>
   (printout t "Who moves first (Computer: c "
               "Human: h)? ")
   (assert (player-select =(read))))

(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))

(defrule bad-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&~c&~h)
```

```
=>
(retract ?phase ?choice)
(assert (phase choose-player))
(printout t "Choose c or h." crlf))
```

Once again, note the use of the control pattern (phase choose-player). It provides the basic control for the input loop and also prevents this group of rules from firing during other phases of execution in the program.

Notice how the two rules, *player-select* and *bad-player-choice*, work together in that each rule supplies the facts necessary to activate the other. If the response to the player question is incorrect, the *bad-player-choice* rule will retract the control fact (phase choose-player) and then reassert it causing the *player-select* rule to be reactivated.

9.4 PREDICATE FUNCTIONS

A **predicate function** is defined to be any function which is intended to return a value of either true or false. CLIPS treats the value of zero as being false and *any* other value as true. Predicate functions may either be **predefined functions** or **user-defined functions**. Predefined functions are those functions already provided by CLIPS. User-defined functions or **external functions** are functions other than predefined functions that are written in C or another language and linked with CLIPS. Table 9-1 lists the predefined **logical predicate functions** provided by CLIPS.

Function Name	*Purpose*
not	boolean not function
and	boolean and function
or	boolean or function

Table 9-1
Logical Predicate Functions

The syntax of these functions is shown following

```
(and <<<predicate-function>>>)
(or  <<<predicate-function>>>)
(not <predicate-function>)
```

Table 9-2 lists the predefined **comparison predicate functions** provided by CLIPS.

Function Name	*Purpose*
eq	equal (any)
neq	not equal (any)
=	equal (numeric)
!=	not equal (numeric)
>=	greater than or equal
>	greater than
<=	less than or equal
<	less than

Table 9-2
Comparison Predicate Functions

The syntax of these functions is shown following

```
(eq  <any-value> <any-value>)
(neq <any-value> <any-value>)
(=   <numeric-value> <numeric-value>)
(!=  <numeric-value> <numeric-value>)
(>=  <numeric-value> <numeric-value>)
(>   <numeric-value> <numeric-value>)
(<=  <numeric-value> <numeric-value>)
(<   <numeric-value> <numeric-value>)
```

All of the comparison functions except the *eq* and *neq* functions will give an error message if given a non-numeric value as an argument. The *eq* and *neq* functions should be used for comparing values which may be non-numeric.

Table 9-3 lists the predefined **type predicate functions** provided by CLIPS.

Function Name	*Purpose*
numberp	argument is number
stringp	argument is string
wordp	argument is word
integerp	argument is an integer
evenp	argument is even
oddp	argument is odd

Table 9-3
Type Predicate Functions

The type predicate functions can be used to determine the type of a field. The syntax of these functions is shown following.

```
(numberp  <value>)
(stringp  <value>)
(wordp    <value>)
(integerp <value>)
(evenp    <value>)
(oddp     <value>)
```

9.5 TEST PATTERNS

There are many cases in which it is useful to repeat a calculation or other information processing. The common way of doing this is by setting up a loop. In the previous example, a loop was set up to repeat until the user responded correctly to a question. But there are also many situations in which a loop needs to terminate automatically as the result of the evaluation of an arbitrary expression.

The **test pattern** provides a very powerful way to evaluate expressions on the LHS of the rule. Instead of pattern matching against a fact in the fact-list, the test pattern evaluates an expression. The outermost function of the expression must be a predicate function. If the expression evaluates to true, the test pattern is satisfied. If the expression evaluates to false, the test pattern is not satisfied. A rule will be triggered only if all its test patterns are satisfied, along with other patterns. The syntax of the test pattern is shown following.

```
(test <predicate-function>)
```

As an example, suppose that it is the human player's turn. If only one stick remains in the pile, the human player has lost. If there is more than one stick, the human player should be asked how many sticks are to be removed from the pile. The rule which asks the human player how many sticks should be removed from the pile needs to check that there is more than one stick remaining in the pile. The > comparison predicate function can be used to express this constraint as shown in the following test pattern

```
(test (> ?size 1))
```

where ?size is the number of sticks remaining in the pile.

Once the human player has stated the number of sticks to be removed, the response needs to be checked to insure that it is valid. The number of sticks taken must be an integer and must be greater than or equal to one and less than or equal to three. A player cannot take more sticks than are in the pile and must be forced to take the last stick. The *and* logical predicate function can be used to express all of these constraints as shown in the following test pattern

```
(test (and (integerp ?choice)
           (>= ?choice 1)
           (<= ?choice 3)
           (< ?choice ?size)))
```

where ?choice would contain the number of sticks to be taken and ?size is the number of sticks remaining in the pile.

Likewise, an invalid number of sticks to take would be a value that is not an integer, less than one or greater than three, or greater than or equal to the number of sticks remaining. This can be expressed using the *or* logical function as shown in the following test pattern

```
(test (or  (not (integerp ?choice))
           (< ?choice 1)
           (> ?choice 3)
           (>= ?choice ?size)))
```

where ?choice would contain the number of sticks to be taken and ?size is the number of sticks remaining in the pile.

The following rules use the preceding test patterns to check that the human player taken a valid number of sticks. The *pile-size* facts used in these rules stores the information about the number of sticks remaining in the pile.

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   ; Human player only has a choice when there is
   ; more than one stick remaining in the pile
   (test (> ?size 1))
   =>
   (printout t
             "How many sticks do you wish to take? ")
   (assert (human-takes =(read))))

(defrule good-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (and (integerp ?choice)
              (>= ?choice 1)
              (<= ?choice 3)
              (< ?choice ?size)))
   =>
   (retract ?whose-turn ?number-taken)
```

```
   (printout t "Human made a valid move" crlf))

(defrule bad-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (or (not (integerp ?choice))
             (< ?choice 1)
             (> ?choice 3)
             (>= ?choice ?size)))
   =>
   (printout t "Human made an invalid move" crlf)
   (retract ?whose-turn ?number-taken)
   (assert (player-move h)))
```

This program will end when a valid choice for the number of sticks taken has been entered. Once again, a control fact is used to retrigger a rule to allow invalid responses to be reentered. The *bad-human-move* rule will reassert the control fact (player-move h) to reactivate the *get-human-move* rule. The human player can then reenter the number of sticks to be taken from the pile.

9.6 THE PREDICATE FIELD CONSTRAINT

The **predicate field constraint**, ":", is useful for performing tests directly within patterns. In many ways it is similar to performing a test directly after a pattern, although in some cases, as will be discussed in Chapter 11, it is more efficient to use the predicate field constraint than it is to use a test pattern. The predicate field constraint can be used just like a literal field constraint. It can stand by itself in a field or be used as one part of a more complex field using the NOT, AND, and OR field constraints. The predicate field constraint is always followed by a function for evaluation. As with a test pattern, this function should be a predicate function.

As an example, the *get-human-move* rule in the previous section used the following two patterns to check that the pile contained at least one stick.

```
(pile-size ?size)
(test (> ?size 1))
```

These two pattern could be replaced with the single pattern

```
(pile-size ?size&:(> ?size 1))
```

The predicate field constraint can be used in a pattern field by itself. Howev ιere are few situations in which this is useful. Typically a variable is bound and then tested

using the predicate field constraint. When reading a pattern it is useful to think of the predicate field constraint as meaning *such that.* For example, the pattern field shown previously

```
?size&:(> ?size 1)
```

could be read as "bind ?size such that ?size is greater than 1".

One application of the predicate field constraint is error checking of data. For example, the following rule checks that a data item is numeric before adding it to a running total

```
(defrule add-sum
   (data-item ?value&:(numberp ?value))
   ?old-total <- (total ?total)
   =>
   (retract ?old-total)
   (assert (total =(+ ?total ?value))))
```

The second field of the first pattern can be read as "bind ?value such that ?value is a ber." The following two rules both perform the same kind of error checking. They onstrate the use of the NOT and OR field constraints with the predicate field constraint.

```
(defrule found-symbol-1
   (data-item ?item&:(stringp ?item)|:(wordp ?item))
   =>
   (printout t "Found symbol or word " ?item crlf))

(defrule found-symbol-2
   (data-item ?item&~:(numberp ?item))
   =>
   (printout t "Found symbol or word " ?item crlf))
```

The rule *found-symbol-1* checks that a data item is either a string or a word by using the *stringp* and *wordp* type predicate functions. The rule *found-symbol-2* performs the same check, but uses the *numberp* type predicate function and then negates the return value.

9.7 THE EQUALITY FIELD CONSTRAINT

..e **equality field constraint**, "=", allows the return value of a function to be used for comparison inside a pattern. The equality field constraint can be used in conjunction with the NOT, AND, and OR field constraints as well as the predicate field constraint.

Like the predicate field constraint, the equality field constraint must be followed ı function. However, the function does not have to be a predicate function. The only restriction is that the function must have a single field return value. The following rule shows how the equality constraint can be used in the Sticks program to determine the number of sticks that the computer player should remove from the pile.

```
(deffacts take-sticks-information
   (computer-take 1 sticks-if-remainder 1)
   (computer-take 1 sticks-if-remainder 2)
   (computer-take 2 sticks-if-remainder 3)
   (computer-take 3 sticks-if-remainder 0))

(defrule computer-move
   ?whose-turn <- (player-move c)
   ?pile <- (pile-size ?size)
   (test (> ?size 1))
   (computer-take ?number sticks-if-remainder
                  =(mod ?size 4))
   =>
   (retract ?whose-turn ?pile)
   (assert (pile-size =(- ?size ?number)))
   (assert (player-move h)))
```

The *computer-move* rule determines the appropriate number of sticks for the computer to take on its turn. The first pattern insures that the rule is applicable only when it is the computer's move. The second and third patterns check to see that the remaining number of sticks is greater than one. If there had only been one remaining stick, then the computer would be forced to take it and lose. The final pattern works in conjunction with the *take-sticks-information* deffacts to determine the appropriate number of sticks to take. The **mod** function, short for modulus, returns the integer remainder of its first argument divided by its second. When reading a pattern it is useful to think of the equality field constraint as meaning *is equal to.* For example, the pattern field shown above

```
=(mod ?size 4)
```

could be read as "The field is equal to ?size modulus 4".

The computer will try to take enough sticks to set the remainder to one when the total number of sticks is divided by four. If the remainder is one when the computer has begun its move, then it has lost unless its opponent makes a mistake. It only takes one stick in this case to make the game last longer (hoping that the human player will make a mistake). In all other cases, the computer can take the appropriate number of sticks to cause its opponent to lose.

To demonstrate how the equality field constraint works in more detail, let's consider a specific example. Assume that the pile size is 7 and that it is the computer's move. The first two patterns of the *computer-move* rule will be satisfied with the last pattern remaining. The function expression (mod ?size 4) will have the value of 7 substituted for ?size leaving the expression (mod 7 4) which evaluates to 3. The *computer-move* rule now "appears" as follows

```
(defrule computer-move
   ?whose-turn <- (player-move c)
   ?pile <- (pile-size ?size)
   (test (> ?size 1))
   (computer-take ?number sticks-if-remainder 3)
   =>
   (retract ?whose-turn ?pile)
   (assert (pile-size =(- ?size ?number)))
   (assert (player-move h)))
```

The fourth pattern will match the *computer-take* fact with 3 in the fourth field. For this case, the computer will take two sticks leaving a pile size of 5. After the human 'ayer's next move, the computer can force a loss.

As another example of the equality field constraint, suppose a rule has the following patterns.

```
(data length ?y)
(data width ?x)
(test (or (= (?x (+ 5 ?y)) (= ?x (- 12 ?y)))))
```

The second and third patterns can be read as "?x is equal to 5 + ?y or ?x is equal to 12 - ?y". So if the facts

```
(data length 4)
(data width 9)
```

are asserted then ?y will be bound to 4, and ?x will be bound to 9. Note that this application of "=" in the test pattern is completely different from the use of the "=" as the equality field constraint in the computer-move rule. In the computer-move rule, the "=" substitutes a return value into a pattern. This is called **pattern substitution.** In the test pattern above, the "=" is used as a predicate comparison function. So the symbol "=" can be used in two different ways.

The two patterns

```
(data width ?x)
(test (or (= (?x (+ 5 ?y)) (= ?x (- 12 ?y)))))
```

can be expressed as a single pattern using the equality field constraint, show
following

```
(data width ?x&=(+ 5 ?y)|=(- 12 ?y))
```

The field

```
?x&=(+ 5 ?y)|=(- 12 ?y)
```

can be read as "bind ?x such that ?x is equal to ?y plus 5 or ?x is equal to 12 minu
?y".

9.8 THE STICKS PROGRAM

All of the basic techniques needed for completing the Sticks program have been dis
cussed. The complete listing of the Sticks program in shown in Appendix G. A sampl
run of the program after it has been loaded is shown following

```
CLIPS> (reset)↵
CLIPS> (run)↵
Who moves first (Computer: c Human: h)? c↵
How many sticks in the pile? 15↵
Computer takes 2 stick(s).
13 stick(s) left in the pile.
How many sticks do you wish to take? 3↵
10 stick(s) left in the pile.
Computer takes 1 stick(s).
9 stick(s) left in the pile.
How many sticks do you wish to take? 2↵
7 stick(s) left in the pile.
Computer takes 2 stick(s).
5 stick(s) left in the pile.
How many sticks do you wish to take? 1↵
4 stick(s) left in the pile.
Computer takes 3 stick(s).
1 stick(s) left in the pile.
You must take the last stick!
You lose!
15 rules fired
CLIPS>
```

9.9 SALIENCE

Up to this point, control facts have been used to indirectly control the execution of programs. CLIPS provides a direct way of control through **salience**. Normally, the agenda acts like a stack. That is, the most recent activation placed on the agenda is the first to fire. Salience allows more important rules to stay at the top of the agenda regardless of when the rules were added to the agenda. Lower salience rules are pushed onto the agenda below higher salience rules.

CLIPS has a keyword called salience which is used to set the priority of rules. The salience is set using a numeric value ranging from the smallest value of -10,000 to the highest of 10,000. If a rule has no salience explicitly assigned by the programmer, then CLIPS assumes a salience of 0. Notice that a salience of 0 is midway between the largest and smallest salience values. A salience of 0 does not mean that the rule has no salience but rather that it has an intermediate priority level. A newly activated rule is placed on the agenda before all rules with an equal or lesser salience and below all rules with a greater salience.

One use of salience is to force rules to fire in a sequential fashion. Consider the following set of rules in which no salience values are declared.

```
(defrule fire-first
   (priority first)
   =>
   (printout t "Print first" crlf))

(defrule fire-second
   (priority second)
   =>
   (printout t "Print second" crlf))

(defrule fire-third
   (priority third)
   =>
   (printout t "Print third" crlf))
```

The order in which the rules fire is dependent upon the order in which the facts that satisfy the LHS's of the rules are asserted. For example, if the rules are entered, then the following commands will produce the output shown below.

```
CLIPS> (reset)↵
CLIPS> (assert (priority first))↵
CLIPS> (assert (priority second))↵
CLIPS> (assert (priority third))↵
CLIPS> (run)↵
Print third
```

```
Print second
Print first
3 rules fired
CLIPS>
```

Notice the order of output statements. First "Print third" is printed, then "Print second", and finally "Print first". The reason this occurs is that the activated rules are placed on the agenda, which acts like a stack (last on is first off). The first fact (priority first) activates the rule *fire-first*. When the second fact is asserted, it activates the rule *fire-second*, which is stacked on top of the activation for rule *fire-first*. Finally, the third fact is asserted and its activated rule, *fire-third*, is stacked on top of the activation for rule *fire-second*.

In CLIPS, rules of equal salience which are activated by different patterns are prioritized based on the stack order of facts. Rules are fired from the agenda from the top of the stack down. So rule *fire-third* is fired first because it's on the top of the stack, then rule *fire-second* and finally rule *fire-first*. If the order in which the facts are asserted is reversed, then the order in which the rules are fired will also be reversed. This is shown by the following output.

```
CLIPS> (reset)↵
CLIPS> (assert (priority third))↵
CLIPS> (assert (priority second))↵
CLIPS> (assert (priority first))↵
CLIPS> (run)↵
Print first
Print second
Print third
3 rules fired
CLIPS>
```

One important point is that if two or more rules having the same salience are all activated by the same fact, there is no guarantee of which order that the rules will be placed on the agenda.

Salience can be used to force the rules to fire in the order *fire-first*, *fire-second*, and then *fire-third*, no matter what order the activating facts are asserted. This can be accomplished by declaring salience values as shown following

```
(defrule fire-first
   (declare (salience 30))
   (priority first)
   =>
   (printout t "Print first" crlf))
```

```
(defrule fire-second
   (declare (salience 20))
   (priority second)
   =>
   (printout t "Print second" crlf))

(defrule fire-third
   (declare (salience 10))
   (priority third)
   =>
   (printout t "Print third" crlf))
```

Regardless of the order in which the priority facts are asserted, the agenda will always be ordered the same. Performing the *agenda* command after asserting the priority facts would produce the following output.

```
30  fire-first  f-1
20  fire-second  f-2
10  fire-third  f-3
```

:e how the salience values have rearranged the priority of rules in the agenda. When the program is run, the order of rule firing will always be *fire-first*, *fire-second*, and then *fire-third*.

9.10 PHASES AND CONTROL FACTS

The purest concept of a rule-based expert system is one in which the rules act opportunistically whenever they are applicable. Most expert systems, however, have some procedural aspect to them. The Sticks program, for example, had different rules which were applicable depending upon whether it was the human or computer's move. The control for this program was handled by facts which indicated whose turn it was. These control facts allow information about the control structure of the program to be embedded into the rules of domain knowledge. However, doing this has a drawback. The knowledge about the control of the rules is intermixed with the knowledge about how to play the game. This is not a major drawback in the case of the Sticks program, because it is a small program. However, for larger programs involving hundreds or thousands of rules, the intermixing of domain knowledge and control knowledge make development and maintenance a major problem.

As an example, consider the problem of performing **fault detection, isolation ~ecovery** of a a system such as an electronic device. Using an electronic device as ꭓample, fault detection is the process of recognizing that the electronic device is not working properly. Isolation is the process of determining the components of the device that have caused the fault. Recovery is the process of determining the steps

necessary to correct the fault, if possible. Typically for this type of problem, xpert system will have rules to determine if a fault has occurred, other rules will ᵢ ᵢe the cause of the fault, and still other rules will determine how to recover from the fault. The cycle will then loop back. Figure 9-1 shows an example of control flow in this type of system.

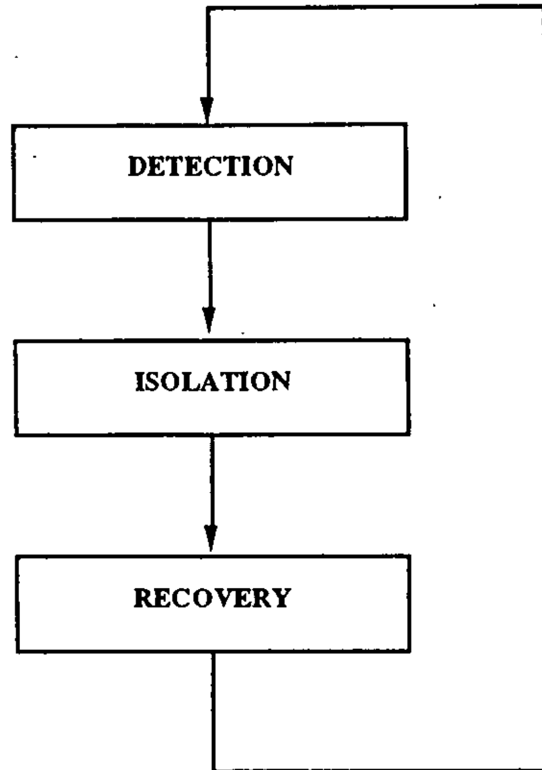


Figure 9-1
Different Phases for Fault Detection, Isolation, and Recovery Problem

Implementing the flow of control in this system can be attempted in at least three ways. The first would be to embed the control knowledge directly into the rules. For instance, the detection rules would include rules indicating when the isolation phase should be entered. Each group of rules would be given a pattern indicating in which phase it would be applicable. This technique has two drawbacks. First, as already mentioned, control knowledge is being embedded into the domain knowledge rules which makes them more difficult to understand. Second, it is not always very easy to determine when a phase is completed. This generally requires writing a rule which is applicable only when all the other rules have fired.

The second approach is to use salience to organize the rules as shown in Figure 9-2. This approach also has two major drawbacks. First, control knowledge is still being embedded into the rules using salience. Second, this approach does not guarantee the correct order of execution. Detection rules will always fire before isolat' ules.

However, when the isolation rules begin firing they might cause a detection rule to become activated and immediately fire because of its higher salience.

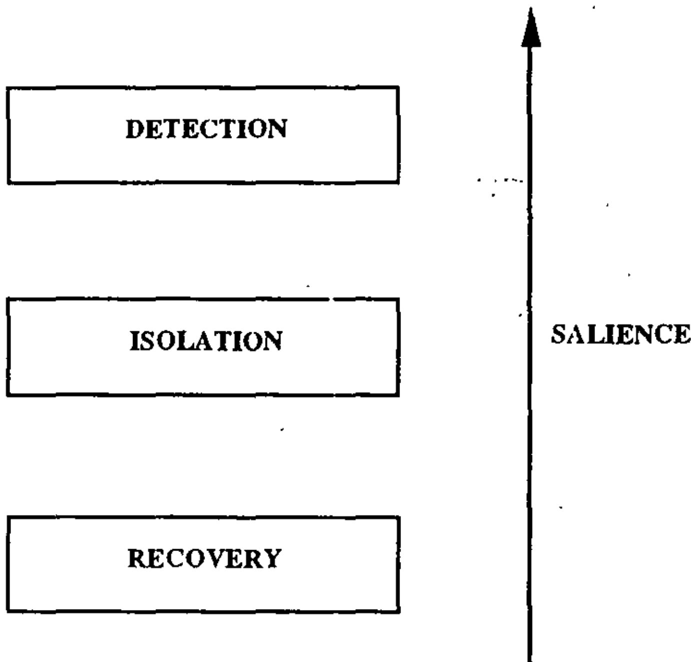


Figure 9-2
Assignment of Salience for Different Phases

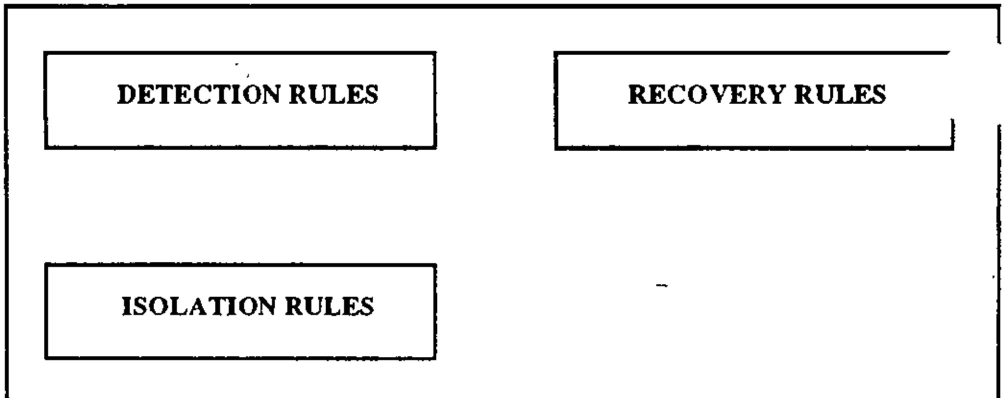
A better approach in controlling the flow of execution is to separate the control knowledge from the domain knowledge as shown in Figure 9-3. Using this approach, each rule is given a control pattern which indicates its applicable phase. Control rules are then written to transfer control between the different phases as shown below.

```
(defrule detection-to-isolation
   (declare (salience -10))
   ?phase <- (phase detection)
   =>
   (retract ?phase)
   (assert (phase isolation)))

(defrule isolation-to-recovery
   (declare (salience -10))
   ?phase <- (phase isolation)
   =>
   (retract ?phase)
   (assert (phase recovery)))
```

```
(defrule recovery-to-detection
   (declare (salience -10))
   ?phase <- (phase recovery)
   =>
   (retract ?phase)
   (assert (phase detection)))
```

Each of the rules for a particular phase is then given a control pattern which checks that appropriate control fact is present for the rule to be applicable. For example, a recovery rule might look like the following.

```
(defrule find-fault-location-and-recovery
   (phase recovery)
   (recovery-solution switch-device ?replacement on)
   =>
   (printout t "Switch device " ?replacement " on "
               crlf))
```



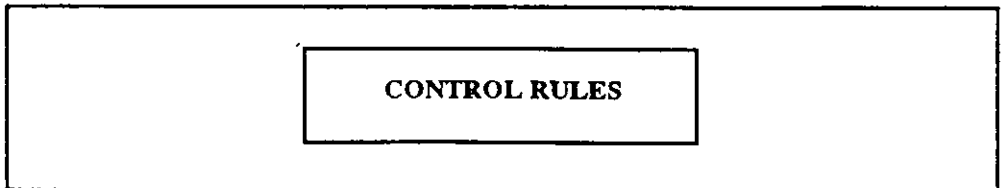


Figure 9-3
Separation of Expert Knowledge from Control Knowledge

A **salience hierarchy** is a description of the salience values used by an expert system. Each level in a salience hierarchy corresponds to a specific set of rules that are all given the same salience. If the rules for detection, isolation and recovery are given a default salience of zero, then the salience hierarchy is shown in Figure 9-4. Notice that while the fact (phase detection) is in the fact-list, the *detection-to-isolation* rule will be on the agenda. However, since it has a lower salience than the detection rules, it will not fire until all of the detection rules have had an opportunity to fire. The following output shows a sample run of the three previous control rules

```
CLIPS> (assert (phase detection))↵
CLIPS> (watch rules)↵
CLIPS> (run 10)↵
FIRE    1 detection-to-isolation: f-1
FIRE    2 isolation-to-recovery: f-2
FIRE    3 recovery-to-detection: f-3
FIRE    4 detection-to-isolation: f-4
FIRE    5 isolation-to-recovery: f-5
FIRE    6 recovery-to-detection: f-6
FIRE    7 detection-to-isolation: f-7
FIRE    8 isolation-to-recovery: f-8
FIRE    9 recovery-to-detection: f-9
FIRE   10 detection-to-isolation: f-10
rule firing limit reached
10 rules fired
CLIPS>
```

Notice that the control rules just keep firing in sequence since there are no domain knowledge rules to be applied during any of the phases. If there were domain knowledge rules for any of these phases, then the domain knowledge rules would be applied for activated rules during their appropriate phase.

The previous control rules could be more generically written with a deffacts construct and a single rule as shown following.

```
(deffacts control-information
   (phase detection)
   (phase-after detection isolation)
   (phase-after isolation recovery)
   (phase-after recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   (phase-after ?current-phase ?next-phase)
   =>
```

```
   (retract ?phase)
   (assert (phase ?next-phase)))
```

or it could be written using a sequence of phases to be cycled through as shown following.

```
(deffacts control-information
   (phase detection)
   (phase-sequence isolation recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?current-phase <- (phase ?current-phase)
   (phase-sequence ?next-phase $?other-phases)
   =>
   (retract ?current-phase)
   (assert (phase ?next-phase))
   (assert (phase-sequence $?other-phases
                           ?next-phase)))
```

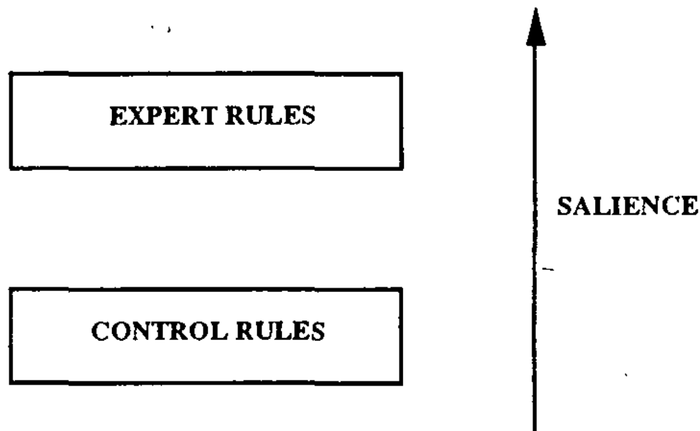


Figure 9-4
Salience Hierarchy using Expert and Control Rules

Additional levels can easily be added to the salience hierarchy. Figure 9-5 shows a hierarchy with two additional levels. The constraint rules represent rules that detect illegal or unproductive states that may occur in the expert system. For example, an expert system scheduling people to various tasks may produce a schedule that violates a constraint. Instead of allowing the lower salience rules to continue working on the schedule, the constraint rules will immediately remove violations in the schedule. As another example, the user may enter in response to a series of questions a series of legal values that result in an illegal value being generated. The constraint rules can be used to detect such violations.

The query rules shown in the diagram represent rules that ask the user particular questions to aid the expert system in determining an answer. These rules have lower salience than the expert rules because it is undesirable to ask the user a question that can be determined by the expert rules. Thus, the query rules are only fired when no more information can be derived by the expert rules.

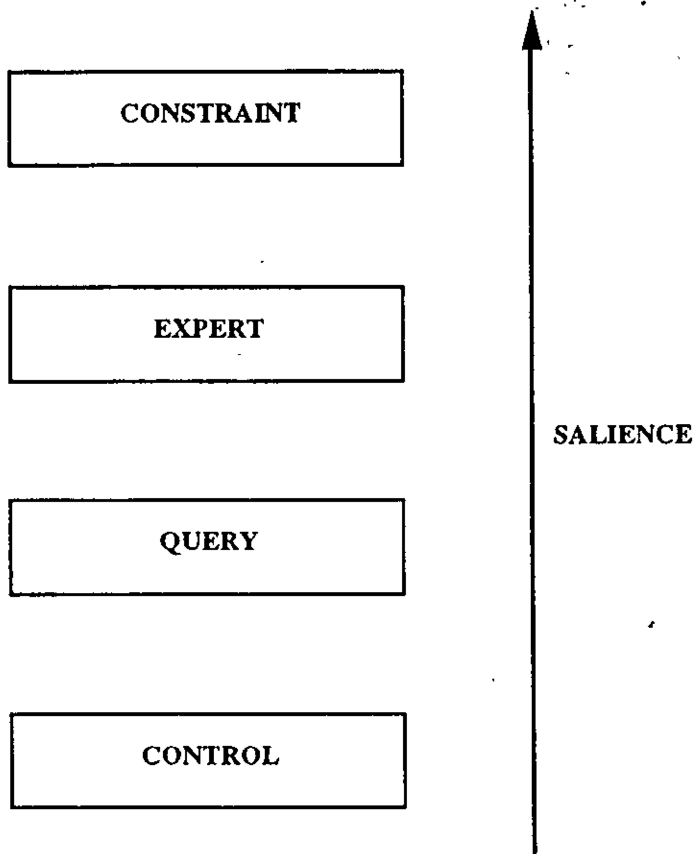


Figure 9-5
Four Level Salience Hierarchy

9.11 MISUSE OF SALIENCE

Although salience is a powerful tool for controlling execution, it can be easily abused. In particular, people who are just learning rule-based programming tend to overuse salience because it gives them explicit control of execution. It is more like the procedural programming they are used to, in which statements execute sequentially.

Overuse of salience results in a poorly coded program. The main advantage of a r[illegible]based program is that the programmer does not have to worry about controlling [illegible]tion. A well-designed rule-based program has a natural mode of execution that permits the inference engine to guide rule firings in an optimum manner. Salience

interferes with the normal operation of the inference engine by increasing the rity of certain rules. While salience is very useful at times, it should not be overuseo.

In general, *any* salience value used in a rule should correspond to a level in the salience hierarchy of the expert system. The range of salience values from -10,000 to 10,000 is somewhat misleading. Rarely should more than seven salience values ever be required for coding an expert system and most well-coded expert systems need no more than three or four salience values. Using salience as a "quick fix" to get rules to fire in the proper order should be avoided. Chapter 11 contains some suggestions on ways to avoid overusing salience in rules.

9.12 THE PATTERN LOGICAL OR

So far all the rules shown have an **implicit logical AND** between the patterns. That is, a rule will not be triggered unless all of the patterns are true. However, CLIPS also provides the capability of specifying an **explicit logical AND** condition and also an **explicit logical OR** condition on the LHS.

As an example of a logical OR, consider the following rules for use by the industrial plant monitoring system (as discussed in Chapter 7) which are first written without a logical OR. Then you will see how to rewrite them with a logical OR.

```
(defrule shut-off-electricity-1
   (emergency flood)
   =>
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
   (fire-class C)
   =>
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-3
   (sprinkler-systems active)
   =>
   (printout t "Shut off the electricity" crlf))
```

Rather than writing three separate rules, they can all be combined into the following rule, using a logical OR.

```
(defrule shut-off-electricity
   (or  (emergency flood)
        (fire-class C)
        (sprinkler-systems active))
   =>
```

```
   (printout t "Shut off the electricity" crlf))
```

The group of patterns enclosed within the logical OR element is called a **logic block.** The one rule using the logical OR is equivalent to the previous three rules. Asserting the three facts matching the patterns of this rule would cause the rule to be triggered three times, once for each of the facts.

Other patterns can be included outside the logical OR block and will be part of the implicit logical AND. For example, the following rule

```
(defrule shut-off-electricity
   (electrical-power on)
   (or  (emergency flood)
        (fire-class C)
        (sprinkler-systems active))
   =>
   (printout t "Shut off the electricity" crlf))
```

is equivalent to the three rules shown following.

```
(defrule shut-off-electricity-1
   (electrical-power on)
   (emergency flood)
   =>
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
   (electrical-power on)
   (fire-class C)
   =>
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-3
   (electrical-power on)
   (sprinkler-systems active)
   =>
   (printout t "Shut off the electricity" crlf))
```

As can be seen, the logical OR is similar to a true logical OR. In a true logical OR, the rule *shut-off-electricity* would only be triggered once if one or more conditions are true. However, in the logical OR of CLIPS, the *shut-off-electricity* rule will be ered for each OR pattern that is true, since the one rule using the logical OR is valent to three rules.

Since multiple triggerings of a logical OR will occur with multiple facts, a natural question is how to prevent the problem of more than one triggering. For example,

multiple printouts of "Shut off the electricity" appearing for multiple are unnecessary. After all, the power only needs to be shut off once, regardle. f the number of reasons to shut it off.

Perhaps the most appropriate manner to prevent the other rules from firing is to change the rule to update the fact-list indicating that the electricity has been shut off. The modified rule is shown following.

```
(defrule shut-off-electricity
   ?power <- (electrical-power on)
   (or  (emergency flood)
        (fire-class C)
        (sprinkler-systems active))
   =>
   (retract ?power)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))
```

The rule now only fires once because the fact containing information about the electrical power is retracted when the rule fires. This will remove other activations of the rule made by the other patterns. If it is also necessary to remove the supporting reason for shutting off the power from the fact-list, then the rule would have to be written as follows.

```
(defrule shut-off-electricity
   ?power <- (electrical-power on)
   (or  ?reason <- (emergency flood)
        ?reason <- (fire-class C)
        ?reason <- (sprinkler-systems active))
   =>
   (retract ?power ?reason)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))
```

Notice that all patterns within the logical OR are bound to the same variable, ?reason. At first, this may appear to be an error since the same variables are normally not assigned to patterns of an ordinary rule. But a rule using a logical OR is different. Since a logical OR will produce multiple rules, the above rule is equivalent to the following three rules.

```
(defrule shut-off-electricity-1
   ?power <- (electrical-power on)
   ?reason <- (emergency flood)
   =>
   (retract ?power ?reason)
```

```
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
   ?power <- (electrical-power on)
   ?reason <- (fire-class C)
   =>
   (retract ?power ?reason)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-3
   ?power <- (electrical-power on)
   ?reason <- (sprinkler-systems active)
   =>
   (retract ?power ?reason)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))
```

By looking at the three rules, if can be seen that the same variable name was necessary to match the (retract ?power ?reason) action. For example, suppose separate names had been assigned to the patterns such as ?reason1, ?reason2, and ?reason3 and then a (retract ?power ?reason1 ?reason2 ?reason3) action was used.

```
(defrule shut-off-electricity
   ?power <- (electrical-power on)
   (or  ?reason1 <- (emergency flood)
        ?reason2 <- (fire-class C)
        ?reason3 <- (sprinkler-systems active))
   =>
   (retract ?power ?reason1 ?reason2 ?reason3)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))
```

The (retract ?power ?reason1 ?reason2 ?reason3) action would be in error because two out of three variables would not be defined in the equivalent rules.

9.13 THE PATTERN LOGICAL AND

The logical AND is opposite in concept to the logical OR. Instead of any patterns triggering a rule, the logical AND requires that all of the patterns be matched to facts. Normally there is an implicit logical AND for the patterns of a rule. A rule such as

```
(defrule electrical-fire
   (emergency fire)
   (fire-class C)
   =>
   (printout t "Shut off the electricity" crlf))
```

could also be written with an explicit logical AND as

```
(defrule electrical-fire
   (and (emergency fire)
        (fire-class C))
   =>
   (printout t "Shut off the electricity" crlf))
```

Of course, there is no advantage to writing a rule with an explicit logical AND. It's easier to use the implicit logical AND of patterns if it is not necessary to specify a logical AND. The logical AND is provided so that it can be used with the other logical functions to make more complex patterns. For example, it can be used with a logical OR to require groups of multiple conditions to be true, as shown in the following example.

```
(defrule shut-off-electricity
   ?power <- (electrical-power on)
   (or  (emergency flood)
        (and (emergency fire)
             (fire-class C))
        (sprinkler-systems active))
   =>
   (retract ?power)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))
```

So the rule *shut-off-electricity* is equivalent to the three following rules,

```
(defrule shut-off-electricity-1
   ?power <- (electrical-power on)
   (emergency flood)
   =>
   (retract ?power)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
   ?power <- (electrical-power on)
```

```
   (emergency fire)
   (fire-class C)
   =>
   (retract ?power)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-3
   ?power <- (electrical-power on)
   (sprinkler-systems active)
   =>
   (retract ?power)
   (assert (electrical-power off))
   (printout t "Shut off the electricity" crlf))
```

9.14 THE PATTERN LOGICAL NOT

Sometimes it is useful to be able to pattern match against the absence of a particular fact in the fact-list. CLIPS allows the specification of the absence of a fact as a pattern in the LHS using the logical **NOT**. As a simple example, the monitoring expert system might have two rules for reporting its status.

```
If the monitoring status is to be reported and
there is an emergency being handled
then report the type of the emergency

If the monitoring status is to be reported and
there is no emergency being handled
then report that no emergency is being handled
```

The logical NOT can be conveniently applied to the simple rules above as follows.

```
(defrule report-emergency
   (report-status)
   (emergency ?type)
   =>
   (printout t "Handling " ?type " emergency" crlf))

(defrule no-emergency
   (report-status)
   (not (emergency ?))
   =>
   (printout t "No emergency being handled" crlf))
```

Notice that these two rules are mutually exclusive. That is, they cannot both be on the agenda at the same time because the second pattern in both rules cannot be satisfied simultaneously.

Variables can also be used within a negated pattern to produce some interesting effects. Consider the following rule which looks for the largest number out of a group of facts representing numbers.

```
(defrule largest-number
   (number ?x)
   (not (number ?y&:(> ?y ?x)))
   =>
   (printout t "Largest number is " ?x crlf))
```

The first pattern will bind to all of the *number* facts, however, the second pattern will not allow the rule to become activated for any but the fact with the largest value for ?x.

Note that variables first bound within a logical NOT are limited in scope to that logical NOT. For example, the following rule

```
(defrule demonstrate-not-1
   (pair ?x ?y)
   (pair ?y ?z)
   (not (triplet ?x ?x ?z))
   =>)
```

will be activated if the fact-list is

```
f-1  (pair 3 4)
f-2  (pair 4 5)
f-3  (triplet 4 4 6)
```

The first pattern will match f-1, the second pattern will match f-2, and the third pattern will look for instances of the fact (triplet 3 3 5) which does not exist and so this pattern will also match.

If the third pattern is move to the front as shown following, however, the rule will not match using the same set of facts.

```
(defrule symmetric-pair-2
   (not (triplet ?x ?x ?y))
   (pair ?x ?y)
   (pair ?y ?z)
   =>)
```

The first pattern will be matched by f-3 thus failing to satisfy the logical NOT. Thus the placement of a NOTed pattern within a rule can be very important.

Variables that were previously bound in another pattern without a logical NOT can be freely used within a NOTed pattern. However, variables that are first bound within a NOTed pattern cannot be used outside of that pattern. These variables are bound only for the purpose of determining if any fact matches the pattern. If no fact matches the pattern, there is no real binding and so the variable cannot be used outside of the pattern.

Since variables cannot be used outside the logical NOT, it follows that the *find-largest-number* rule cannot be rewritten using a test as shown below.

```
(defrule largest-number
   (number ?x)
   (not (number ?y))
   (test (> ?y ?x))
   =>
   (printout t "Largest number is " ?x crlf))
```

Finally, a logical NOT can only be used to negate a single pattern. Multiple 'erns, the logical AND, and the logical OR cannot be used with a logical NOT.

9.15 SUMMARY

In this chapter, various concepts for controlling the flow of execution were introduced. The read function was used to demonstrate how a simple control loop for input could be created using control facts that were retracted and then asserted again. Test patterns along with predicate functions can be used on the LHS of a rule to provide more powerful pattern matching capabilities. In addition, test patterns can be used to maintain a control loop. The predicate field constraint allows test expressions to be placed directly within a pattern. The equality field constraint is used to compare a field to a value returned by a function. The Sticks program demonstrates several of these control techniques.

Salience provides a mechanism for even more complex control structures. Salience can be used to prioritize rules such that the activated rule with the highest salience is fired first. Salience can be combined with control facts in order to separate expert knowledge from control knowledge.

Logical pattern operators can be used to express several rules as a single rule. The logical AND and OR can be arbitrarily nested to express complex conditions needed to satisfy a rule. The logical NOT allows pattern matching against the absence of a fact. 're must be exercised with the logical NOT since variables that are first bound within . NOTed pattern have their scope limited to that pattern.

PROBLEMS AND PROGRAMS

9-1. Add rules to the Sticks program which will ask if the human player wishes to play again after the game has finished. Do not use salience.

9-2. Modify the Sticks program such that the control rules are separated from the rules of playing the game. Use salience to give the control rules a lower priority.

9-3. Modify the Sticks program to allow two human players to play the game in addition to the computer playing against a human.

9-4. Rewrite the following rules as a single rule using logical pattern operators.

```
(defrule rule-1
   (fact-a)
   (fact-d)
   =>)

(defrule rule-2
   (fact-b)
   (fact-c)
   (fact-e)
   (fact-f)
   =>)

(defrule rule-3
   (fact-a)
   (fact-e)
   (fact-f)
   =>)

(defrule rule-4
   (fact-b)
   (fact-c)
   (fact-d)
   =>)
```

9-5 Determine if the variable x is referenced properly for each of the following rules. Explain your answer.

a)

```
(defrule example-1
   (not (fact ?x))
   (test (> ?x 4)).
```

```
      =>)
```

b)

```
(defrule example-2
   (not (fact ?x&:(> ?x 4)))
   =>)
```

c)

```
(defrule example-3
   (not (fact ?x))
   (fact ?y&:(> ?y ?x)))
   =>)
```

d)

```
(defrule example-4
   (not (fact ?x))
   (fact ?x&:(> ?x 4)))
   =>)
```

Since logical pattern operators cannot be placed inside of a logical pattern Γ operator, show how the following rule can be expressed as a group of valid CLIPS rules.

```
(defrule invalid-pattern-operator-rule
   (not (and (a ?x)
             (b ?x)))
   =>
   (assert (c)))
```

Hint: Using salience, it can be done with four rules.

9-7. Add a rule to the blocks world program found in Section 8.6 which would remove a move goal if it has already been satisfied.

9-8. Rewrite the blocks world program found in Section 8.6 so that it can rearrange the blocks from any initial state of stacked blocks to any goal state of stacked blocks. For example, if the initial state of the blocks was

```
(stack A B C)
(stack D E F)
```

One possible goal state might be

```
(stack D C B)
```

```
(stack A)
(stack F E)
```

9-9. Write a program using the logical AND, OR, and NOT's for the AND/OR tree of getting to work shown in Figure 3-10. Test it for all branches.

9-10. Implement a finite state machine driver using rules. Given the current state and the next input, the rules should determine the transition to the next state. Implement the rules such that the knowledge about the states and transitions are stored as facts. Make sure that an error state is entered for any invalid input. Test your rules as follows.

a) Implement the finite state machine shown in Figure 3-5. Use only deffacts constructs in addition to the rules written above.
b) Implement the finite state machine shown in Figure 3-6. Use only deffacts constructs in addition to the rules written above.

```
   =>
   (retract ?phase)
   (printout t "Continue? " crlf)
   (bind ?answer (read))
   (if (or (eq ?answer y) (eq ?answer yes))
      then (assert (phase continue))
      else (assert (phase halt))))
```

Notice that the *if* function is used to convert the user's yes/no response to a fa indicating the type of action to be taken. In this case the action is to either continue (halt.

The While Function

The syntax of the *while* function is

```
(while <predicate-function> [do]
   <<<actions>>>)
```

where <predicate-function> is a predicate function to be evaluated and <<<actions>>> represent one or more functions. The actions of the *while* function comprise the bod of the loop. The keyword **do** is optional.

The part of the while function represented by <predicate-function> represents condition that is tested before the actions in the body are executed. If th while-condition is true, the actions in the body will be executed. If the while-conditio is false, execution continues with the next statement after the *while* function, if an The condition of the *while* function will be checked each time the body is executed t determine if the body should be executed again.

The *while* function can be used with the *if* function to implement input erro checking on the RHS of a rule. The following modification to the rule *continue-chec* uses the *while* function to continue looping until an appropriate answer is received.

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? " crlf)
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no)) do
      (printout t "Continue? " crlf)
      (bind ?answer (read)))
   (if (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

CLIPS is designed to be an efficient rule-based language. The *if* and *while* procedural functions are intended for judicious use only. Writing a lengthy procedural program on the RHS of a rule defeats the entire purpose of using a rule-based language. In general, the *if* and *while* functions should be used for performing simple tests and loops on the RHS of a rule. Complex nesting of *if* and *while* functions on the RHS of a rule should be avoided.

The Halt Function

The *halt* function can be used on the RHS of a rule to stop the execution of rules on the agenda. It requires no arguments. When called, no further actions will be executed from the RHS of the rule being fired and control will return to the top-level prompt. The agenda will contain any remaining rules that were activated when the *halt* function was called.

As an example, the *continue-check* rule could replace the action

```
(assert (phase halt))
```

with the following action

```
(halt)
```

which would stop the execution of rules.

The *halt* function is particularly useful for halting execution when the intent to restart execution later using the *run* command. Consider the following modification to the *continue-check* rule

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? " crlf)
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no)) do
      (printout t "Continue? " crlf)
      (bind ?answer (read)))
   (assert (phase continue))
   (if (neq ?answer yes)
      then (halt)))
```

Notice that the rule asserts the fact (phase continue) regardless of the user's response to the continue question. Asserting this fact will place the appropriate rules on the agenda to continue execution. However, if the reply to the continue question was not yes, execution will be stopped by the *halt* function. The user could then

examine the rules and facts; later restarting execution where it had left off by issuing a *run* command.

10.3 FILE I/O FUNCTIONS AND LOGICAL NAMES

The Open Function

In addition to keyboard input and terminal output, CLIPS can also read from and write to files. Before a file can be accessed for reading or writing, it must be opened using the **open** function. The number of files that can be open simultaneously is dependent on your operating system and hardware.

The syntax of the *open* function is

```
(open <file-name> <file-ID> [<file-access>])
```

As an example,

```
(open "input.dat" data "r")
```

The first argument of open, <file-name>, is a string representing the name of the file on your computer. Generally, this will be the physical file name known to the operating system. For the example shown, the file-name is "input.dat".

The second argument, <file-ID>, is the **logical name** under which CLIPS knows the file. The logical name is a global name by which CLIPS can access the file from any rule or the top-level prompt. Although the logical name could be the same as the file-name, it is good idea to use a different name to avoid confusion. For the example shown, the logical name *data* was used to refer to the file "input.dat". Other meaningful names, such as input or file-data, could have been used.

One advantage of using a logical name is that a different file name can easily be substituted without making major changes to the program. Since the file name is only used in the *open* function and is later referred to by its logical name, only the *open* function need be changed to read from a different file.

The third argument, <file-access>, is a string representing one of the four possible modes of file access. Table 10-1 list the file access modes.

Mode	*Action*
"r"	read access only
"w"	write access only
"r+"	read and write access
"a"	append access only

Table 10-1
File Access Modes

If <file-access> is not included as an argument, the default value of "r" will be used. Some access modes may not be meaningful to certain operating systems. Most operating systems will support read access (for inputting data) and write access (for outputting data). Read and write access and append access (for appending output data to the end of a file) may not always be available.

It is important to remember that the consequences of opening a file may vary from one machine to another. For example, operating systems which do not support multiple file versions, such as MS-DOS on an IBM PC or UNIX, will replace an existing file if that file is opened using write access. In contrast, VMS on a VAX will create a new version of a file if it already exists and is opened using write access.

The *open* function acts as a predicate function. It returns true if a file was successfully opened, otherwise false is returned. The return value can be used to perform error checking. For example, if the user supplies the name of a file that does not exist and an attempt is made to open the file using read access, the *open* function will return a value of false. The rule that opens the file could detect this and perform the appropriate actions to prompt the user for another file name.

The Close Function

Once access is no longer needed to a file, it should be closed. Unless a file is closed, there is no guarantee that the information written to it will be saved. Also, the longer a file is open, the greater is the chance of a power loss or other malfunction which would prevent the information from being saved.

The general form of the close function is

```
(close [<file-ID>])
```

where the optional argument <file-ID> specifies the logical name of the file to be closed. If <file-ID> is not specified, all open files will be closed. As an example,

```
(close data)
```

will close the file known to CLIPS by the logical file-name *data*. The statements

```
(close input)
(close output)
```

will close the files associated with the logical names *input* and *output*. Note that separate statements are necessary to close specific files.

When using files, it is important to remember that each opened file should eventually be closed with the *close* function. In particular, if a command is not issued to close a file, the data written to it may be lost. CLIPS will not prompt you to close an open file. The only safeguard built into CLIPS for closing files that have been

inadvertently left open is that all open files will be closed when an *exit* command is issued.

However, a problem arises if CLIPS is terminated with an interrupt control character such as Control C. Depending on the operating system, the data written to open files may not be saved. That's why it's always a good idea to exit CLIPS with an *exit* command rather than an interrupt. A graceful exit will usually save any data in files inadvertently left open. As another safeguard, you can always include a rule with a very low salience which closes all open files. The salience of this rule should be made low enough that all other rules in the system fire before it does.

Reading and Writing to a File

In all the examples so far, all input has been read from the keyboard and all output has been sent to the terminal. However, the use of logical names allow input and output to and from other sources. In Chapter 7, the *printout* function used the logical name *t* to send output to the screen. Other logical names can be used with the *printout* function to send output to destinations other than the screen.

When used as a logical name for an output function, the logical name *t* writes output to the standard output device which is usually the terminal. Similarly, when used as the logical name for an input function, the logical name *t* reads input from the **standard input device** which is normally the keyboard. For example, the *read* function introduced in Chapter 9 defaults to reading from the standard input device if it is given no arguments. The general format of the read function is

```
(read [<logical-name>])
```

As an example of using logical names with the *read* and *printout* functions, let's simulate monitoring the oil pressure of an oil pump. Values will be read in from a file indicating the oil pressure of the pump, then the values will be analyzed, and the analysis will be written to the terminal as well as a **trace file**. The trace file will keep a record of all important information that has been written to the terminal.

The first step in solving this problem is to open the input and output files. The following rule will do this

```
(defrule open-files
   =>
   (open "oil.dat" oil "r")
   (open "trace.txt" trace "w")
   (assert (phase read-oil-pressure)))
```

The input data is stored in the file "oil.dat" and tracing information will be sent to the output file "trace.txt". The logical name *oil* is now associated with the file "oil.dat" and the logical name *trace* is now associated with the file "trace.txt". Note that the "r" option did not have to used for opening the "oil.dat" file since this is the default. The

fact (read-oil-pressure) will be used to create a loop for reading the oil pressure values. The rule used for reading the value is shown following

```
(defrule read-data
   ?phase <- (phase read-oil-pressure)
   =>
   (retract ?phase)
   (bind ?value (read oil))
   (if (eq ?value EOF)
      then (assert (phase close-files))
      else (assert (oil-value ?value))))
```

Notice that the *read* function uses an additional argument to specify the logical name from which input is to be read. In this case, the logical name *oil* tells the *read* function to access the file "oil.dat" for input. After a value is read from the file, it is compared against the **EOF** symbolic field. CLIPS returns this value for input functions when an attempt is made to read past the end of the file. In this case, if the EOF field is returned, the *read-data* rule asserts a fact indicating that the files should be closed. Otherwise, it asserts the oil pressure value that was read from the file.

If the end of the file was reached, then the following rule will close both the input file "oil.dat" and the output file "trace.txt" that were opened by the *open-files* rule.

```
(defrule close-files
   ?phase <- (phase close-files)
   =>
   (retract ?phase)
   (close oil)
   (close trace))
```

The following two rules analyze the oil pressure value asserted by the *read-data* rule. For simplicity, we will assume that a value greater than or equal to 60 is good and any value less than 60 is bad. The first pattern of the rule *good-oil-value* checks for an oil value greater than or equal to 60, while the first pattern of the rule *bad-oil-value* checks for an oil value less than 60.

```
(defrule good-oil-value
   ?data <- (oil-value ?pressure&:(>= ?pressure 60))
   =>
   (retract ?data)
   (printout t "Oil pressure of " ?pressure
              " is good " crlf)
   (printout trace ?pressure " good " crlf)
   (assert (read-oil-pressure)))
```

```
(defrule bad-oil-value
   ?data <- (oil-value ?pressure&:(< ?pressure 60))
   =>
   (retract ?data)
   (printout t "Oil pressure of " ?pressure
               " is bad " crlf)
   (printout trace ?pressure " bad " crlf)
   (assert (read-oil-pressure)))
```

Notice that each rule has two *printout* actions. One *printout* action uses the logical name *t* to send output to the terminal. The other *printout* action uses the logical name *trace* to send the output to the trace file "trace.txt". Both rules then assert the *read-oil-pressure* fact which sends the program through another input loop. This will continue until the end of file is reached for the "oil.dat" file.

If the following values are stored in the "oil.dat" file,

```
71
64
60
56
58
61
```

then the following output represents a sample run of the program

```
CLIPS> (watch rules)↵
CLIPS> (reset)↵
CLIPS> (run)↵
FIRE     1 open-files: f-0
FIRE     2 read-data: f-1
FIRE     3 good-oil-value: f-2
Oil pressure of 71 is good
FIRE     4 read-data: f-3
FIRE     5 good-oil-value: f-4
Oil pressure of 64 is good
FIRE     6 read-data: f-5
FIRE     7 good-oil-value: f-6
Oil pressure of 60 is good
FIRE     8 read-data: f-7
FIRE     9 bad-oil-value: f-8
Oil pressure of 56 is bad
FIRE    10 read-data: f-9
FIRE    11 bad-oil-value: f-10
Oil pressure of 58 is bad
```

```
FIRE    12 read-data: f-112
FIRE    13 good-oil-value: f-12
Oil pressure of 61 is good
FIRE    14 read-data: f-13
FIRE    15 close-files: f-14
CLIPS>
```

After running this program, the file "trace.txt" should contain the following information

```
71 good
64 good
60 good
56 bad
58 bad
61 good
```

which indicates that the tracing information has been properly written to the trace file.

10.4 OTHER I/O FUNCTIONS

The Format Function

There are many times when it is desirable to format output from a CLIPS program, such as arranging data in tables. While the *printout* function is useful, there is a function specifically designed for formatting, called **format**, which provides a wide variety of formatting styles. The syntax of the *format* function is

```
(format <logical-name> <control-string>
        <<parameters>>)
```

The *format* function has several parts. The <logical-name> is the logical name where the output is sent. The default standard output device can be specified with the logical name *t*. Next comes a **control string** which must be contained within double quotes. The control string consists of **format flags** which indicate how the parameters to the *format* function should be printed. Following the control string is a list of parameters. The number of format flags in the control string will determine how many parameters should be specified.

An example of a *format* statement in CLIPS is shown by the following command which prints the integer forms of two numbers reserving 15 spaces for each number. Note that there are no spaces between the "%" signs.

```
CLIPS> (format t "Integer%n%15d%15d%n" 5.25
10)↵
Integer
                    5              10
CLIPS>
```

Format flags always begin with a "%" sign. Ordinary strings such as "Integer" can also be put in the control string and will be printed in the output. Some format flags do not format parameters. In this example, the "%n" is used to put a carriage return/linefeed in the output. For this example, the *format* function has printed the integer form of its parameters: 5.25 and 10. Besides constants, the *format* function can also evaluate and print expressions used as arguments. Notice in the output that the number 5.25 was printed in its integer form of 5 since the fractional part of a number is not allowed in an integer format.

Note that there is no space between the "Integer" and the control flag "%n". Although it would help readability to put a space in between, CLIPS would also print the space since that is a valid character. Another flag format is the "%15d". The "15" specifies in how many columns the parameter is to be printed in. After the "15" is the part of the format flag which specifies how the output is to be displayed, such as "d" for an integer specification. The general specification of a format flag is

%-M.Nx

where "-" is optional and means to **left justify**. The default is to **right justify**.

M specifies the field width in columns. At least M characters will be output. Spaces are normally used to pad the output to make up the M columns unless M starts with a 0, in which case zeroes are used. If the output exceeds M columns, then the format function will expand the field as needed.

N is an optional specification of the number of digits past the decimal point that will be printed. The default is six digits past the decimal point for floating-point numbers.

x specifies the display format specification. Table 10-2 gives the display format specifications.

Shown following are some top-level *format* commands to illustrate output formats. The text Float, Exponential, General, Octal, and Hexadecimal in the control strings of the *format* function examples are not required keywords. They were just chosen as headings for the output in these examples.

```
CLIPS> (format t "Float%n%15f%15f%15f%n" 1 5.25
10)↵
Float
       1.000000       5.250000      10.000000
CLIPS> (format t "Exponential%n%15e%15e%n" 5.25
10)↵
Exponential
  5.250000e+000  1.000000e+001
CLIPS> (format t "General%n%15g%15g%n" 5.25
10)↵
General
           5.25             10
CLIPS> (format t "Octal%n%15o%15o%n" 5.25 10)↵
Octal
              5             12
CLIPS> (format t "Hexadecimal%n%15x%15x%n" 5.25
10)↵
Hexadecimal
              5              a
CLIPS> (format t "String%n%15s%15s%n" "efg"
hij)↵
String
            efg            hij
```

Character	*Meaning*
d	Integer
f	Floating-point
e	Exponential (in power-of-ten format)
g	General (numeric).Display in whatever format is shorter.
o	Octal. Unsigned number (N specifier not applicable)
x	Hexadecimal. Unsigned number (N specifier not applicable)
s	String. Quoted strings will be stripped of quotes.
n	Carriage return/linefeed
%	The "%" character itself

Table 10-2
Display Format Specifications

The Readline Function

The **readline** function can be used to read an entire line of input. Its syntax is

```
(readline [<logical-name>])
```

As with the *read* function, the logical name is optional. If no logical name is provided or the logical name *t* is used, input will be read from the standard input device. The *readline* function will return as a string the next line of input from the input source associated with the logical name up to and including the carriage return. The following rule illustrates the use of the *readline* function.

```
(defrule get-name
   =>
   (printout t "What is your name? ")
   (assert (name =(readline))))
```

The following command dialog illustrates the use of the *readline* function in the *get-name* rule.

```
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (run)↵
What is your name? John Smith↵
==> f-1     (name "John Smith")
1 rules fired
CLIPS>
```

The *readline* function will return the word *EOF* if the end of file has been reached. This will only occur when the logical name used by readline is associated with a file.

Loading and Saving Facts

It is often useful to be able to load facts from a file or save facts to a file. The **load-facts** and **save-facts** functions provide these capabilities. The syntax of these two functions is

```
(load-facts <file-name>)
(save-facts <file-name>)
```

The *load-facts* function will load in a group of facts stored in the file specified by <file-name>. The facts in the file should be in the standard format of a fact. That is, each fact should begin with a left parenthesis, have one or more fields after the left parenthesis, and end with a right parenthesis. For example, if the file "facts.dat" contained the following

```
(data 34)
(data 89)
(data 64)
(data 34)
```

then the command

```
(load-facts "facts.dat")
```

would load the facts contained in this file.

The *save-facts* function can be used to save all of the facts in the fact-list to the file specified by <file-name>. The facts are stored in the format required by the *load-facts* function. If you only want to save certain facts, then it is necessary to write rules which will match against the facts to be saved and store them in a file in the format required by the *load-facts* function. The following rules illustrate how the *data* facts shown above could be selectively saved to a file.

```
(defrule start-save-data
   ?phase <- (phase start-save)
   =>
   (retract ?phase)
   (printout t "Save data in which file? ")
   (bind ?file-name (readline))
   (if (open ?file-name save-file "w")
       then
      (assert (phase write-data))
      else
      (printout t "Unable to open file" crlf)))

(defrule save-data-info
   (phase write-data)
   (data $?rest)
   =>
   (printout save-file "(data " $?rest ")" crlf))

(defrule finish-save-data
   (declare (salience -1))
   ?phase <- (phase write-data)
   =>
   (retract ?phase)
   (close save-file))
```

The rule *start-save-data* is activated by the control fact (phase start-save). It prompts for a file name in which the *data* facts should be stored and then opens the file. If it is able to open the file, it asserts the fact (phase write-data) to indicate that the *data* facts should be written out to the file. The rule *save-data-info* will be activated once for every *data* fact in the fact-list. It writes each *data* fact to the file in the format expected by the *load-facts* function. The rule *finish-save-data* closes the file once all the facts have been saved. It is given a lower salience than the *save-data-info* rule to allow all of

the facts to be written to the file before it is closed. If rules had been included to perform switching from one phase to the next phase, the *finish-save-data* rule could have been placed in the phase following the *write-data* phase and the salience value of -1 would not have been required for the rule.

10.5 STRING FUNCTIONS

CLIPS provides several functions for manipulating strings. These functions are **str_assert, str_cat, str-index,** and **sub-string**. The syntax of these functions is shown following.

```
(str_assert <string>)
(str_cat <<items>>)
(str-index <string-1> <string-2>)
(sub-string <integer-1> <integer-2> <string>)
```

The Str_assert Function

The *str_assert* function takes a string as argument and breaks it up into fields which are asserted as a fact. For example, the top-level command

```
(str_assert "data-values 1 5 6")
```

would cause the assertion of the fact

```
(data-values 1 5 6)
```

Notice that there are no double quotes around the asserted fact. The fact consists of four fields while the original string was one field.

Spaces, carriage returns, or tabs within a string are used as delimiters by the *str_assert* function in determining where to create fields. A quoted string may be included by using the "\" as an escape character. For example,

```
(str_assert "data-values green \"red blue\" black")
```

would cause the fact

```
(data-values green "red blue" black)
```

to be asserted.

The Str_cat Function

The *str_cat* function constructs a single quoted string from one or more individual items. The items can be words, numbers, strings, bound variables or functions which return a word, string, or numbers. For example, the following top-level command would produce the output shown

```
CLIPS> (str_cat "data-" 9 ".txt")↵
"data-9.txt"
CLIPS>
```

Notice that the fields are all run together. If it is necessary to have a space between the words in the fact, a space should be entered between the double quotes as shown following.

```
CLIPS> (str_cat colors " " red " " green " "
blue)↵
"colors red green blue"
CLIPS>
```

The Str-index Function

The *str-index* function returns the position of the first string within the second string. If the string is not found, then zero is returned. For example,

```
CLIPS> (str-index "red" "blueredgreen")↵
5
CLIPS> (str-index "red" "bluegreen")↵
0
CLIPS>
```

The Sub-string Function

The *sub-string* function returns a portion of a string. It requires three arguments. The first two arguments are the beginning and ending indices of the substring. The third argument is the string from which the substring is to be taken. For example,

```
CLIPS> (sub-string 3 5 "a string")↵
"str"
CLIPS>
```

10.6 MULTIFIELD FUNCTIONS

CLIPS provides a number of functions that are useful with multifield variables. These functions are **mv-append**, **mv-delete**, **length**, **nth**, **member**, **subset**, **mv-subseq**, **str-explode**, and **str-implode**. The syntax is as follows.

```
(mv-append <<items>>)
(mv-delete <field-index> <multifield value>)
(length <multifield value>)
(nth <field-index> <multifield value>)
(member <field> <multifield value>)
(subset <multifield value-1>
        <multifield value-2>)
(mv-subseq <field-index-1> <field-index-2>
           <multifield value>)
(str-explode <string>)
(str-implode <multifield value>)
```

The Mv-append and Mv-delete Functions

The *mv-append* function constructs a single multifield value from individual items.The items can be words, numbers, strings, bound single or multifield variables or functions which return a word, string, numbers, or multifields. The *mv-delete* function deletes an indexed field from a multifield value. For example,

```
CLIPS> (mv-append a b 9 "red")↵
a b 9 "red"
CLIPS> (mv-delete 3 (mv-append a b 9 "red"))↵
a b "red"
CLIPS>
```

If no arguments are passed to *mv-append*, a multifield value of length zero is created. A multifield value of length zero is referred to as **null**, which literally means "nothing." Only a multifield variable can be bound to null. Notice that *mv-append* is used to generate a multifield value for the *mv-delete* function. Since variables cannot be used at the top-level, there is no other way to generate a multifield constant.

The Length Function

The *length* function returns the number of fields in a multifield value. For example,

```
CLIPS> (length (mv-append a b 9 "red"))↵
4
CLIPS>
```

The Nth Function

The *nth* function returns the field of the multifield specified by the first argument. If the field does not exist, the value **nil** is returned. For example,

```
CLIPS> (nth 2 (mv-append a b 9 "red"))↵
b
CLIPS> (nth 5 (mv-append a b 9 "red"))↵
nil
CLIPS>
```

The Member Function

The *member* function determines if a specific field is contained within a multifield value. If the field is found, an index to the field is returned, otherwise zero is returned. For example,

```
CLIPS> (member 9 (mv-append a b 9 "red"))↵
3
CLIPS> (member 5 (mv-append a b 9 "red"))↵
0
CLIPS>
```

The Subset Function

The *subset* function determines if one multifield value is a subset of another multifield value. A set such as (a b c) is considered a subset of (c b a), (b a c), and so forth. A null multifield value is a subset of *everything*, including itself. The value 1 is returned if the first argument is a subset of the second argument, otherwise the value 0 is returned. For example,

```
CLIPS> (subset (mv-append a b c)
               (mv-append c b a))↵
1
CLIPS> (subset (mv-append a b b c)
               (mv-append b c a))↵
1
CLIPS> (subset (mv-append a d c)
               (mv-append c a))↵
0
CLIPS>
```

The Mv-subseq Function

The *mv-subseq* function returns a portion of a multifield value. It requires three arguments. The first two arguments are the beginning and ending indices of the subsequence. The third argument is the multifield value from which the subsequence is to be taken. For example,

```
CLIPS> (mv-subseq 2 4 (mv-append a b c d e))↵
b c d
CLIPS>
```

The Str-explode Function

The *str-explode* function converts a string to a multifield value. For example,

```
CLIPS> (str-explode "a b c d e")↵
a b c d e
CLIPS>
```

The Str-implode Function

Conversely, the *str-implode* function converts a multifield value to a string. For example,

```
CLIPS> (str-implode (mv-append a b c d e))↵
"a b c d e"
CLIPS>
```

10.7 EXTENDED MATH FUNCTIONS

In addition to the standard CLIPS math functions of +, -, * and /, there are a large number of additional mathematical functions provided by the CLIPS extended math package. The extended math functions are normally provided with CLIPS, but may be excluded from some CLIPS systems to conserve memory.

Trigonometric Functions

Table 10-3 show the list of trigonometric functions provided by the extended math package. Each of the trigonometric functions expects a single argument. Where applicable, each trigonometric function expects its argument to be in radians.

Function Name	*Purpose*
cos	cosine
sin	sine
tan	tangent
sec	secant
csc	cosecant
cot	cotangent
acos	arccosine
asin	arcsine
atan	arctangent
asec	arcsecant
acsc	arccosecant
acot	arccotangent
cosh	hyperbolic cosine
sinh	hyperbolic sine
tanh	hyperbolic tangent
sech	hyperbolic secant
csch	hyperbolic cosecant
coth	hyperbolic cotangent
acosh	hyperbolic arccosine
asinh	hyperbolic arcsine
atanh	hyperbolic arctangent
asech	hyperbolic arcsecant
acsch	hyperbolic arccosecant
acoth	hyperbolic arccotangent

Table 10-3
Trigonometric Functions

The function **pi** can be used to obtain the value of π in radians. It requires no arguments. For example,

```
CLIPS> (pi)↵
3.14159274
CLIPS> (cos (pi))↵
-1
CLIPS>
```

Conversion Functions

The functions **deg-rad**, **rad-deg**, **deg-grad**, and **grad-deg** are provided for converting between degrees, radians and gradients. The syntax of these functions is as follows

```
(deg-rad <argument-in-degrees>)
(rad-deg <argument-in-radians>)
(deg-grad <argument-in-degrees>)
(grad-deg <argument-in-gradients>)
```

The *deg-rad* function converts its argument from degrees to radians. The *rad-deg* function converts its argument from radians to degrees. The *deg-grad* function converts its argument from degrees to gradients. The *grad-deg* function converts its argument from gradients to degrees.

Min and Max Functions

The **min** and **max** functions can be used to find the minimum and maximum value from a group of numeric values. The syntax of these functions is

```
(min <<numbers>>)
(max <<numbers>>)
```

Both functions expects one or more numeric arguments. The *min* function will return the smallest value of all its arguments and the *max* function will return the largest value of all its arguments. For example

```
CLIPS> (min 3 1 8 5)↵
1
CLIPS> (max 3 1 8 5)↵
8
CLIPS>
```

Logarithm Functions

The functions **log**, **log10**, and **exp** are useful for performing mathematical operations with logarithms. The syntax of these functions is

```
(log <number>)
(log10 <number>)
(exp <number>)
```

Each function takes a single numeric argument. The *log* function returns the logarithm base e of its argument. The *log10* function returns the logarithm base 10 of its argument. The *exp* function returns e^n where n is the argument passed to the function.

Predicate Functions

The functions *evenp*, *oddp*, and *integerp* are provided as part of the extended math package. These functions were introduced in Chapter 9 and are listed in Table 9-3 which shows a list of CLIPS type predicate functions.

Miscellaneous Functions

Several other miscellaneous functions are provided by the extended math package. They are **sqrt**, **trunc**, **abs**, **mod**, and ******. The syntax of these functions is

```
(sqrt <number>)
(trunc <number>)
(abs <number>)
(mod <number> <number>)
(** <number> <number>)
```

The *sqrt* function returns the square root of its argument. The *trunc* function truncates its argument so that it is the floating point equivalent of an integer value. The *abs* function returns the absolute value of its argument. The *mod* function returns the remainder that results from dividing its first argument by its second argument (the first argument modulus the second argument). The ** function returns the value of its first argument raised to the power of its second argument. For example,

```
CLIPS> (sqrt 9)↵
3
CLIPS> (trunc 11.3)↵
11
CLIPS> (abs -5)↵
5
CLIPS> (mod 9 2)↵
1
CLIPS> (** 3 2)↵
9
CLIPS>
```

10.8 UTILITY COMMANDS

The System Command

The **system** command allows the execution of operating system commands from within CLIPS. The syntax of the system command is

```
(system <<single-field arguments>>)
```

For example, the following rule will give a directory listing for a specified directory on a machine using the UNIX operating system.

```
(defrule list-directory
   (list-directory ?directory)
   =>
   (system "ls " ?directory))
```

For this example, the first argument to the *system* command, "ls ", is the UNIX command for listing a directory. Notice that a space is included after the characters "ls". The *system* command simply appends all its arguments together as strings before allowing the operating system to process the command. Any spaces needed for the operating system command must be included as part of an argument in the *system* command call.

The effects of the *system* command may vary from one operating system to another. Not all operating systems provide the functionality for implementing the *system* command, so you cannot rely on this command being available in CLIPS.

The Batch Command

The **batch** command allows commands and responses that would normally have to be entered at the top-level prompt to be read directly from a file. The syntax of the *batch* command is

```
(batch <file-name>)
```

For example, suppose the following dialog shows the commands and responses which must be entered to run a CLIPS program. Remember that the bold face letters indicate keys entered by you.

```
CLIPS> (load "rules1.clp")↵
CLIPS> (load "rules2.clp")↵
CLIPS> (load "rules3.clp")↵
CLIPS> (reset)↵
CLIPS> (run)↵
How many iterations? 10↵
Starting value? 1↵
End value? 20↵
Completed
CLIPS>
```

The commands and responses needed to run the program could be stored in a file as shown following.

```
(load "rules1.clp")↲
(load "rules2.clp")↲
(load "rules3.clp")↲
(reset)↲
(run)↲
10↲
1↲
20↲
```

If the file with the commands and responses was named "commands.bat", the following dialog shows how the *batch* command can be used. Notice again that the bold face letters indicate keys entered by you.

```
CLIPS> (batch "commands.bat")↲
CLIPS> (load "rules1.clp")↲
CLIPS> (load "rules2.clp")↲
CLIPS> (load "rules3.clp")↲
CLIPS> (reset)↲
CLIPS> (run)↲
How many iterations? 10↲
Starting value? 1↲
End value? 20↲
Completed
CLIPS>
```

Once all the commands and responses have been read from a batch file, keyboard interaction at the top-level prompt is returned to normal.

When run under operating systems that support command line arguments for executables (such as UNIX), CLIPS can automatically execute commands from a batch file upon startup. Assuming that the CLIPS executable can be executed by typing "clips", the syntax for executing a batch file upon startup is

```
clips -f <file-name>
```

If an *exit* command is processed in the startup batch file, control will be returned to the operating system. Otherwise, CLIPS will enter its interactive state displaying the top-level prompt "CLIPS>". Unlike the *batch* command, only commands can be processed from a startup batch file. Responses to input functions cannot be stored in a startup batch file. Furthermore, the CLIPS prompt and the command being executed will not be displayed until CLIPS enters its interactive state.

The Dribble-on and Dribble-off Commands

The **dribble-on** command can be used to store a record of all output to the terminal and all input from the keyboard. The syntax of the *dribble-on* command is

```
(dribble-on <file-name>)
```

Once the dribble-on command has been executed, all output sent to the terminal and all input entered at the keyboard will be echoed to the file specified by <file-name> as well as to the terminal.

The effects of the *dribble-on* command can be turned off with the **dribble-off** command. The syntax of the *dribble-off* command is

```
(dribble-off)
```

10.9 THE GENSYM FUNCTION

CLIPS has a function called **gensym** which returns a unique word every time it's called. The general form of the value returned by the *gensym* function is shown following.

```
gen<number>
```

where <number> is a suffix numeral supplied by CLIPS each time the *gensym* function is called. The numbers used by the *gensym* function always start at 1 when CLIPS is first started and increment by 1 each time the *gensym* function is called until CLIPS is exited. The following command sequence demonstrates the use of the *gensym* function.

```
CLIPS> (gensym)↵
gen1
CLIPS> (gensym)↵
gen2
CLIPS> (gensym)↵
gen3
CLIPS>
```

The **setgen** function can be used to reset the gensym count or to set it to a specific value. The *setgen* function takes a single argument, the starting number to be appended to the word "gen". For example,

```
CLIPS> (setgen 10)↵
CLIPS> (gensym)↵
gen10
```

```
CLIPS> (gensym)↲
gen11
CLIPS> (setgen 3)↲
CLIPS> (gensym)↲
gen3
CLIPS>
```

The usefulness of the gensym function will be demonstrated in Chapter 12 using an example that demonstrates how uncertainty can be represented in CLIPS.

10.10 SUMMARY

This chapter introduced a wide variety of functions. CLIPS provides functions for manipulating strings and multifield values. A number of math functions are provided by the CLIPS extended math package.

The *if* and *while* functions can be used for flow of control on the RHS of a rule. The *halt* function can be used to halt the execution of rules. Overuse of these functions on the RHS is poor programming style.

The *open* and *close* functions can be used to open and close files. Opened files are associated with a logical name. Logical names can be used in most functions that perform input and output to more than one type of physical device. Both the *printout* and *read* functions used logical names. The *printout* function can output to the terminal and files. The *read* function can input from the keyboard and files. The *format* and *readline* functions also accept logical names. The *format* function allows more control over the appearance of output. The *readline* function can be used to read an entire line of data. The *save-facts* and *load-facts* functions can be used to save facts to a file and load facts from a file.

The *system* command allows operating system commands to be executed from within CLIPS. The *batch* commands allows a series of commands and responses stored in a file to replace normal keyboard input. The *dribble-on* and *dribble-off* commands allow a record of terminal output to be stored in a file.The *gensym* function can be used to generate unique words.

PROBLEMS AND PROGRAMS

10-1. Modify the oil pump program in Section 10.3 to ask for the name of the input and output files. Check to see if the files can be opened. If they cannot, prompt the user for another file name.

10-2. Rewrite the following rule into one or more rules that do not use the if and while functions. Verify that your rules perform the same actions by comparing the output and final fact-list of your rules with the following rule.

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? " crlf)
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no)) do
      (printout t "Continue? " crlf)
      (bind ?answer (read)))
   (if (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

10-3. Write a program that saves a specified group of facts. The facts to be save should be specified with a fact using the following template

```
(save-relations <file-name> <<relation-names>>)
```

where <file-name> indicates the file in which the facts should be stored and <<relation-names>> are the relation names of the facts to be saved. Only those facts beginning with one of the relation names should be saved. Verify your program using the following deffacts construct.

```
(deffacts test-data
  (save-relations "facts.sav" data numbers)
  (numbers 10 20 30)
  (words a b c)
  (data a b 34 c 34)
  (numbers 23 22)
  (words x y z)
  (strings "abc" "def" "ghi")
  (data a b c d 34)
  (data a b))
```

10-4. Write a program which will read a data file containing a list of people's names and ages and create a new file containing the list of people's names and ages sorted in ascending order by age. The program should prompt for both the input and output files. For example, the following input file

```
John Carter 34
Mary Jane Campbell 25
Bill Smith 37
Janis Page 22
```

should create the following output file

```
Janis Page 22
Mary Jane Campbell 25
John Carter 34
Bill Smith 37
```

CHAPTER 11 EFFICIENCY IN RULE-BASED LANGUAGES

11.1 INTRODUCTION

This chapter provides many techniques for increasing the efficiency of a rule-based expert system which uses the Rete Pattern Matching Algorithm. The reasons for needing an efficient pattern matching algorithm are discussed before the Rete Algorithm is explained. Several techniques for writing rules more efficiently are discussed.

11.2 THE RETE PATTERN MATCHING ALGORITHM

Rule-based languages such as CLIPS, ART, OPS5, and OPS83 use a very efficient algorithm for matching facts against the patterns in rules to determine which rules have their conditions satisfied. This algorithm is called the **Rete Pattern Matching Algorithm.** Writing efficient CLIPS rules does not require an understanding of the Rete Algorithm. However, an understanding of the underlying algorithm used in CLIPS and other rule-based languages makes it easier to understand why writing rules one way is more efficient than writing them another way.

To understand why the Rete Algorithm is efficient, it will be helpful to look at the problem of matching facts to rules in general and examine other algorithms which are not as efficient. Figure 11-1 shows the problem addressed by the Rete Algorithm.

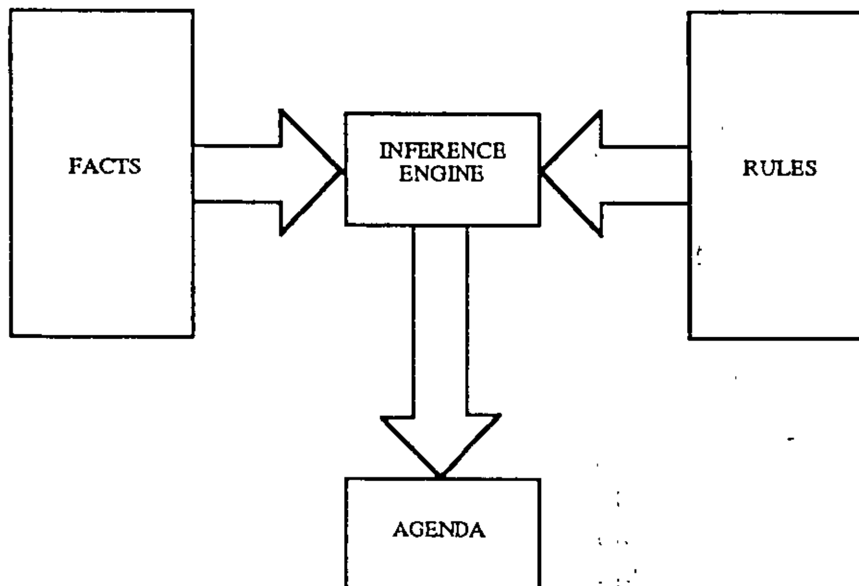


Figure 11-1
Pattern Matching: Rules and Facts

If the matching process only has to occur once, then the solution to the probl(straightforward. The inference engine can examine each rule and then search the st. .. facts to determine if the rule's patterns have been satisfied. If the rule's patterns are satisfied, then the rule can be placed on the agenda. Figure 11-2 shows this approach of having the rules look for the facts needed to match their conditions.

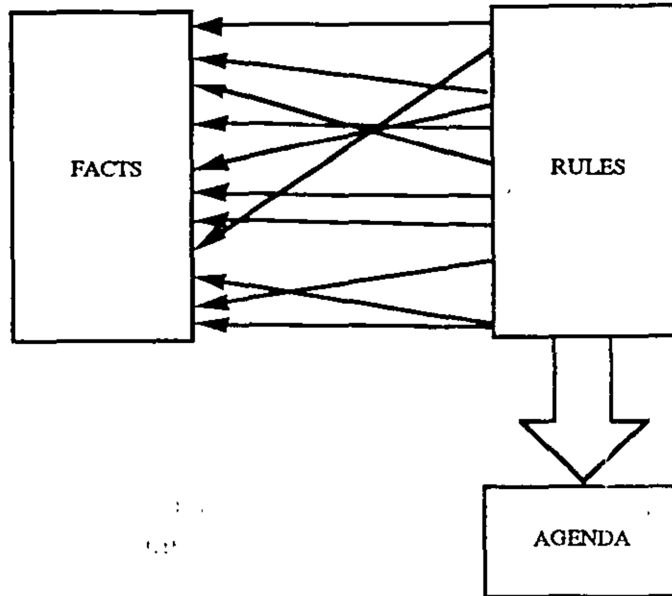


Figure 11-2
Rules Searching for Facts

In rule-based languages, however, the matching process takes place repeatedly. Normally, the fact-list will be modified during each cycle of execution. New facts may be added to the fact-list or old facts may be removed from the fact-list. These changes may cause previously unsatisfied patterns to be satisfied or previously satisfied patterns to become unsatisfied. The problem of matching now becomes an ongoing process. During each cycle, as facts are added and removed, the set of rules that are satisfied must be maintained and updated.

Having the inference engine check each rule to direct the search for facts after each cycle of execution provides a very simple and straightforward technique for solving this problem. The primary disadvantage of such a technique is that it can be very slow. Most rule-based expert systems exhibit a property called **temporal redundancy**. Typically, the actions of a rule will only change a few facts in the fact-list. That is, the facts in the expert system change slowly over time. Each cycle of execution may see only a small percentage of facts either added or removed and so only a small percentage of rules are typically affected by the changes in the fact-list. Thus having the rules drive the search for needed facts requires a lot of unnecessary computation since most of the rules are likely to find the same facts in the current cycle as found in the last cycle. T' inefficiency of this approach is shown in Figure 11-3. The shaded area represents t changes that have been made to the fact-list. Unnecessary recomputation could be

avoided by remembering what has already matched from cycle to cycle and computing only the changes necessary for the newly added or removed facts as shown in Figure 11-4. It is the rules that remain static and the facts that change. Thus the facts should find the rules and not the other way around.

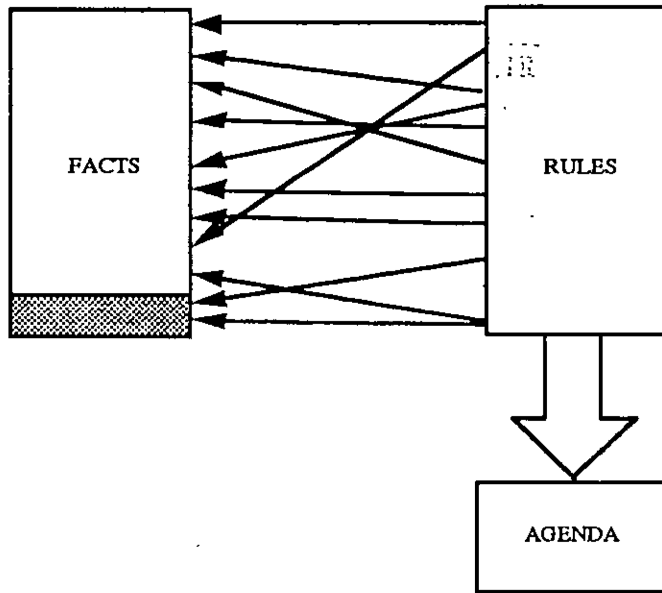


Figure 11-3
Unnecessary Computations when Rules Search for Facts

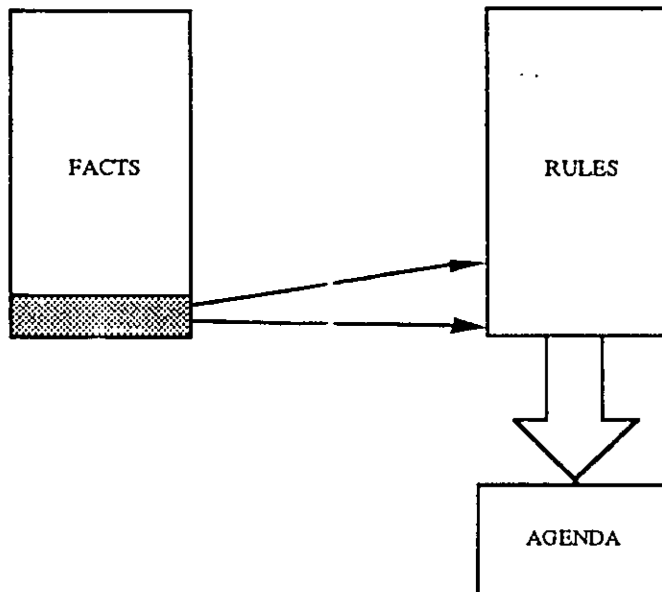


Figure 11-4
Facts Searching for Rules

The Rete Pattern Matching algorithm is designed to take advantage of the temporal redundancy exhibited by rule-based expert systems. It does this by saving the state of the matching process from cycle to cycle and recomputing the changes in this state only for the changes that occur in the fact-list. That is, if a set of patterns finds two of three required facts in one cycle, a check does not have to be made in the next cycle for the two facts that have already been found. Only the third fact is of interest. The state of the matching process is updated only as facts are added and removed. If the number of facts added and removed is small compared to the total number of facts and patterns, then the process of matching will proceed quickly. As a worst case, if all the facts were to be changed, then the matching process would work as if all the facts were compared against all of the patterns.

If only updates to the fact-list are processed, then each rule must remember what has already matched it. That is, if a new fact has matched the third pattern of a rule, then information about the matches for the first two patterns must be available to finish the matching process. This type of state information indicating the facts that have matched previous patterns in a rule is called a **partial match**. A partial match for a rule is any set of facts which satisfy the rule's patterns beginning with the first pattern of the rule and ending with any pattern up to and including the rule's last pattern. Thus, a rule with three patterns would have partial matches for the first pattern, the first and second patterns, and the first, second, and third patterns. A partial match of all of the patterns of a rule will also be an activation. The other type of state information saved is called a **pattern match**. A pattern match occurs when a fact has satisfied a single pattern in any rule without regard to variables in other patterns that may restrict the matching process.

The primary disadvantage of the Rete Pattern Matching Algorithm is that it is very memory intensive. Simply comparing all of the facts to all of the patterns requires no memory. Saving the state of the system using pattern matches and partial matches, however, can consume considerable amounts of memory. In general, this tradeoff of memory for speed is worthwhile. It is important to note that the Rete Algorithm does not perform a simple search in attempting to match facts against rules. Since memory is used to save both pattern matches and partial matches for the rules, a poorly written rule can not only run slowly, but it can use up considerable amounts of memory. Like errors in procedural languages such as a recursive loop that never bottoms out or nested loops that have a combinatorial number of iterations, this is a commonly made error in rule-based programming.

The Rete Algorithm also improves the efficiency of rule-based systems by taking advantage of **structural similarity** in the rules. Structural similarity refers to the fact that many rules often contain similar patterns or groups of patterns. The Rete Algorithm takes advantage of this feature to increase efficiency by pooling common components together so that they do not have to be computed more than once.

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS ABIERTOS

ROBOTS MÓVILES

CHAPTER 1 INTRODUCTION TO EXPERT SYSTEMS

EXPOSITOR : DR. JESÚS SAVAGE CARMONA
1998

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtemoc 06000 Mexico, D.F. APDO. Postal M-2285
Telefonos 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

CHAPTER 1 INTRODUCTION TO EXPERT SYSTEMS

1.1 INTRODUCTION

This chapter is a broad introduction to expert systems. The fundamental principles of expert systems are introduced. The advantages and disadvantages of expert systems are discussed and the appropriate areas of application for expert systems are described. The relationship of expert systems to other methods of programming are discussed.

1.2 WHAT IS AN EXPERT SYSTEM

The first step in solving any problem is defining the problem area or **domain** to be solved. This consideration is just as true in artificial intelligence (AI) as in conventional programming. However, because of the mystique formerly associated with AI, there is a lingering tendency to still believe the old adage "It's an AI problem if it hasn't been solved yet". This type of mind-set may have been popular in the 1970's when AI was entirely in a research stage. However, today there are many real-world problems that are being solved by AI and many commercial applications of AI.

Although general solutions to classic AI problems such as natural language translation, speech understanding, and vision have not been found, restricting the problem domain may still produce a useful solution. For example, it is not difficult to build simple natural language systems if the input is restricted to sentences of the form: noun, verb, and object. Currently, systems of this type work very well in providing a user-friendly interface to many software products such as database systems and spreadsheets. In fact, the parsers associated with popular computer text-adventure games today exhibit an amazing degree of ability in understanding natural language.

As Figure 1-1 shows, AI has many areas of interest. The area of **expert systems** is a very successful approximate solution to the classic AI problem of programming intelligence. Professor Edward Feigenbaum of Stanford University, an early pioneer of expert systems technology, has defined an expert system as "... an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution." (Feigenbaum 82). That is, an expert system is a computer system which **emulates** the decision-making ability of a human expert. The term emulate means that the expert system is intended to act in all respects like a human expert. An emulation is much stronger than a simulation which is only required to act like the real thing in some respects.

Although a general purpose problem solver still eludes us, expert systems function very well in their restricted domains. As proof of their success, you need only observe the many applications of expert systems today in business, medicine, science, and

engineering as well as all the books, journals, conferences and products de' d to expert systems.

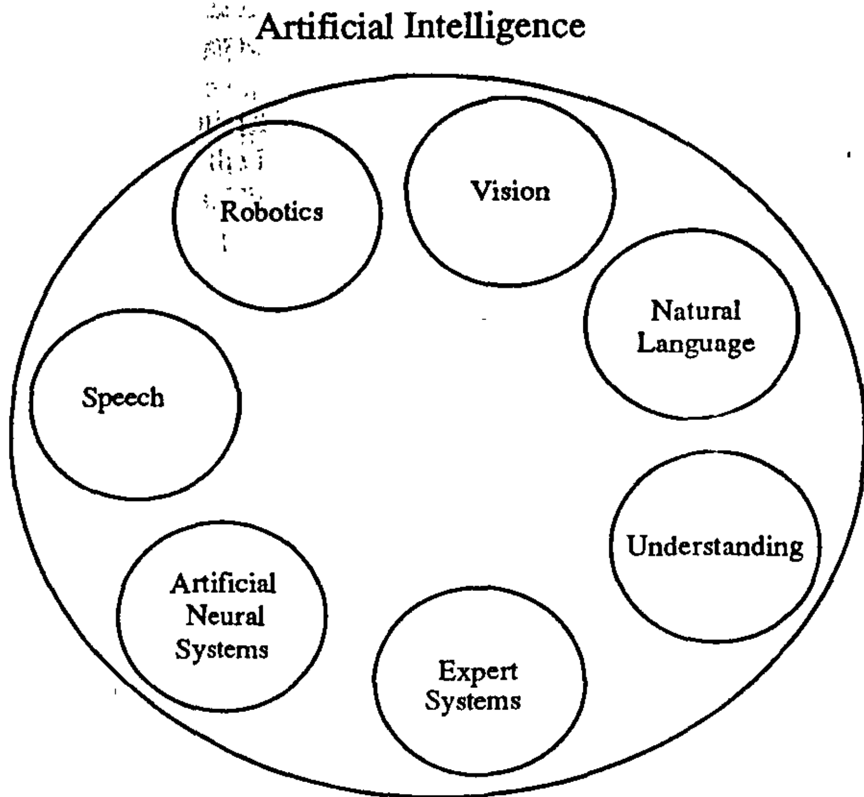


Figure 1-1
Some Areas of Artificial Intelligence

Expert systems is a branch of AI that makes extensive use of specialized knowledge to solve problems at the level of a human **expert**. An expert is a person who has **expertise** in a certain area. That is, the expert has knowledge or special skills that are not known or available to most people. An expert can solve problems that most people cannot solve at all or solve them much more efficiently (but not as cheaply). When expert systems were first developed in the 1970's, they contained expert knowledge exclusively. However, the term expert system is often applied today to any system which uses expert system technology. This expert system technology may include special expert system languages, programs, and hardware designed to aid in the development and execution of expert systems.

The knowledge in expert systems may be either expertise, or knowledge which is generally available from books, magazines, and knowledgeable persons. The terms expert system, **knowledge-based system** or **knowledge-based expert system** are often used synonymously. Most people use expert system simply because it's shorter, even though there may be no expertise in their expert system, only ge—ral knowledge.

Figure 1-2 illustrates the basic concept of a knowledge-based expert system. The user supplies facts or other information to the expert system and receives expert advice or **expertise** in response. Internally, the expert system consists of two main components. The knowledge-base contains the knowledge with which the **inference engine** draws conclusions. These conclusions are the expert system's responses to the user's queries for expertise.

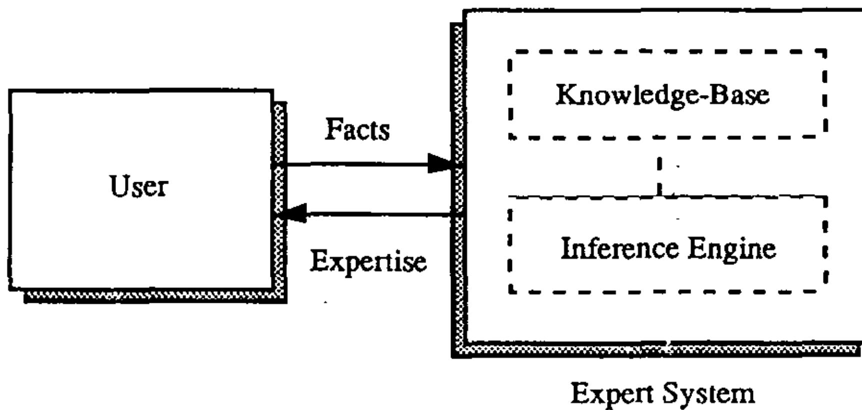


Figure 1-2
Basic Concept of an Expert System Function

Useful knowledge-based systems also have been designed to act as an intelligent assistant to a human expert. These intelligent assistants are designed with expert systems technology because of the development advantages. As more knowledge is added to the intelligent assistant, it acts more like an expert. Developing an intelligent assistant may be a useful milestone in producing a complete expert system. In addition, it may free up more of the expert's time by speeding up the solution of problems.

An expert's knowledge is specific to one **problem domain** as opposed to knowledge about general problem-solving techniques. A problem domain is the special problem area such as medicine, finance, science or engineering and so forth that an expert can solve problems in very well. Expert systems, like human experts, are generally designed to be experts in one problem domain. For example, you would not normally expect a chess expert to have expert knowledge about medicine. Expertise in one problem domain does not automatically carry over to another.

The expert's knowledge about solving specific problems is called the **knowledge domain** of the expert. For example, a medical expert system designed to diagnose infectious diseases will have a great deal of knowledge about certain symptoms caused by infectious diseases. In this case the knowledge domain is medicine and consists of knowledge about diseases, symptoms, and treatments. Figure 1-3 illustrates the relationship between the problem and knowledge domain. Notice that this knowledge domain is entirely included within the problem domain. The portion outside the knowledge domain symbolizes an area in which there is not knowledge about all the problems.

One expert system usually does not have knowledge about other b es o medicine such as surgery or pediatrics. Although its knowledge of infectious ase i. equivalent to a human expert, the expert system would not know anything about othe knowledge domains unless it was programmed with that domain knowledge.

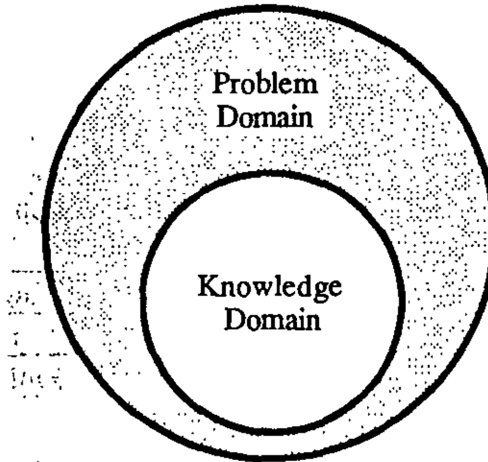


Figure 1-3
A Possible Problem and Knowledge Domain Relationship

In the knowledge domain that it knows about, the expert system reasons kes **inferences** in the same way that a human expert would infer the solution o. prob-lem. That is, given some facts, a conclusion that follows is inferred. For example, if your spouse hasn't spoken to you in a month, you may infer that he or she had nothing worthwhile to say. However, this is only one of several possible inferences.

As with any new technology, we still have a lot to learn about expert systems. Table 1-1 summarizes the differing views of the participants in a technology. In this table, the technologist may be an engineer or software designer and the technology may be hardware or software. In solving any problem, these are questions that need to be answered or the technology will not be successfully used. Like any other tool, expert systems have appropriate and inappropriate applications. As our experience with expert systems grows, we will discover what these applications are.

Person	*Question*
Manager	What can I use it for?
Technologist	How can I best implement it?
Researcher	How can I extend it?
Consumer	How will it help me? Is it worth the trouble and expense? How reliable is it?

Table 1-1
Differing Views of Technology

1.3 ADVANTAGES OF EXPERT SYSTEMS

Expert systems have a number of attractive features.

- *Increased Availability.* Expertise is available on any suitable computer hardware. In a very real sense, an expert system is the mass production of expertise.
- *Reduced cost.* The cost of providing expertise per user is greatly lowered.
- *Reduced Danger.* Expert systems can be used in environments that might be hazardous for a human.
- *Permanence.* The expertise is permanent. Unlike human experts who may retire, quit or die, the expert system's knowledge will last indefinitely.
- *Multiple expertise.* The knowledge of multiple experts can be made available to work simultaneously and continuously on a problem at any time of day or night. The level of expertise combined from several experts may exceed that of a single human expert (Harmon 85).
- *Increased reliability.* Expert systems increase confidence that the correct decision was made by providing a second opinion to a human expert or break a tie in case of disagreements by multiple human experts. Of course, this method probably won't work if the expert system was programmed by one of the experts. The expert system should always agree with the expert, unless a mistake was made by the expert. However, this may happen if the human expert is tired or under stress.
- *Explanation.* The expert system can explicitly explain in detail the reasoning that led to a conclusion. A human may be too tired, unwilling or unable to do this all the time. This increases the confidence that the correct decision is made.
- *Fast response.* Fast or real-time response may be necessary for some applications. Depending on the software and hardware used, an expert system may respond faster and be more available than a human expert. Some emergency situations may require responses faster than a human and so a real-time expert system is a good choice (Hugh 88) (Ennis 86).
- *Steady, unemotional, and complete response at all times.* This may be very important in real-time and emergency situations when a human expert may not operate at peak efficiency because of stress or fatigue.
- *Intelligent tutor.* The expert system may act as an intelligent tutor by letting the student run sample programs and explaining the system's reasoning.
- *Intelligent Database.* Expert systems can be used to access a database in an intelligent manner (Kerschberg 86) (Schur 88).

The process of developing an expert system has an indirect benefit also since the knowledge of human experts must be put into an explicit form for entering in the computer. Because the knowledge is then explicitly known instead of being implicit in expert's mind, it can be examined for correctness, consistency and completeness. knowledge may then have to be adjusted or re-examined which improves the quality of the knowledge.

1.4 GENERAL CONCEPTS OF EXPERT SYSTEMS

The knowledge of an expert system may be represented in a number of ways. One common method of representing knowledge is in the form of IF THEN type **rules**, such as

```
IF the light is red THEN stop
```

Although this is a trivial example, many significant expert systems have been built by expressing the knowledge of experts in rules. In fact, the knowledge-based approach to developing expert systems has completely supplanted the early AI approach of the 1950's and 1960's which tried to use sophisticated reasoning techniques with no reliance on knowledge.

Today, a wide range of knowledge-based expert systems have been built. Large systems containing thousands of rules, such as the XCON/R1 system of Digital Equipment Corporation, know much more than any single human expert on how to configure computer systems (McDermott 84). Many small systems for specialized tasks have also been constructed with several hundred rules. These small systems may not operate at the level of an expert but are designed to take advantage of expert systems technology to perform knowledge-intensive tasks. For these small systems, the knowledge may be in books, journals, or other publicly available documentation

In contrast, a classic expert system embodies unwritten knowledge that [illegible] be extracted from an expert by extensive interviews with a **knowledge engineer** over a long period of time. The process of building an expert system is called **knowledge engineering** and is done by a knowledge engineer (Michie 73). Knowledge engineering refers to the acquisition of knowledge from a human expert or other source and its coding in the expert system.

The general stages in the development of an expert system are illustrated in Figure 1-4. The knowledge engineer first establishes a dialog with the human expert in order to elicit the expert's knowledge. This stage is analogous to a system designer in conventional programming discussing the system requirements with a client for whom the program will be constructed. The knowledge engineer then codes the knowledge explicitly in the knowledge-base. The expert then evaluates the expert system and gives a critique to the knowledge engineer. This process iterates until the system's performance is judged to be satisfactory by the expert.

The expression **knowledge-based system** is a better term for the application of knowledge-based technology since it may be used for the creation of either expert systems or knowledge-based systems. However, like the term artificial intelligence, it is common practice today to use the term expert systems when referring to both expert systems and knowledge-based systems, even when the knowledge is not at the level of a human expert.

Expert systems are generally designed very differently from conventional programs because the problems usually have no algorithmic solution and rely on inferen[illegible]o achieve a reasonable solution. Note that a reasonable solution is about the best we can

expect if no algorithm is available to help us achieve the optimum solution. Since the expert system relies on inference, it must be able to explain its reasoning so that its reasoning can be checked. An **explanation facility** is an integral part of sophisticated expert systems. In fact, elaborate explanation facilities may be designed to allow the user to explore multiple lines of "What if..." type questions, called **hypothetical reasoning** and even translate natural language into rules.

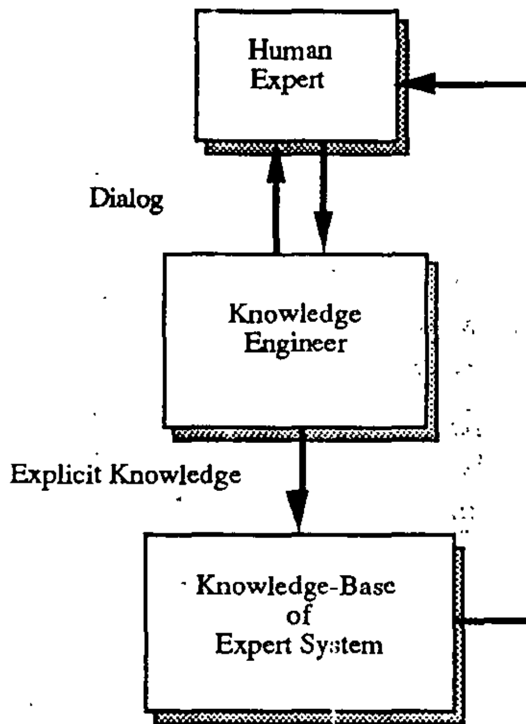


Figure 1-4
Development of an Expert System

Some expert systems even allow the system to learn rules by example, through **rule induction**, in which the system creates rules from tables of data. Formalizing the knowledge of experts into rules is not simple, especially when the expert's knowledge has never been systematically explored. There may be inconsistencies, ambiguities, duplications or other problems with the expert's knowledge that are not apparent until attempts are made to formally represent the knowledge in an expert system.

Human experts also know the extent of their knowledge and qualify their advice as the problem reaches their **limits of ignorance**. A human expert also knows when to "break the rules". Unless expert systems are explicitly designed to deal with uncertainty, they will make recommendations with the same confidence even if the data they dealing with is inaccurate or incomplete. An expert systems advice, like a human expert's, should degrade gracefully at the boundaries of ignorance instead of abruptly.

A practical limitation of many expert systems today is lack of [illegible]usal **knowledge**. That is, the expert systems do not really have an understanding of the underlying causes and effects in a system. It is much easier to program expert systems with **shallow knowledge** based on empirical and heuristic knowledge than **deep knowledge** based on the basic structure, function, and behavior of objects. For example, it is much easier to program an expert system to prescribe an aspirin for a person's headache than program all the underlying biochemical, physiological, anatomical and neurological knowledge about the human body. The programming of a causal model of the human body would be an enormous task and even if successful, the response time of the system would probably be extremely slow because of all the information that the system would have to process.

One type of shallow knowledge is **heuristic knowledge**, where the term heuristic comes from the Greek and means to discover. Heuristics are not guaranteed to succeed in the same way that an algorithm is a guaranteed solution to a problem. Instead, heuristics are rules of thumb or empirical knowledge gained from experience which may aid in the solution but are not guaranteed to work. However, in many fields such as medicine and engineering, heuristics play an essential role in some types of problem solving. Even if an exact solution is known, it may be impractical to use because of cost or time constraints. Heuristics can provide valuable shortcuts that can reduce time and cost.

Another problem with expert systems today is that their expertise is limi[illegible] to the knowledge domain that the systems know about. Typical expert systems can[illegible]ner-alize their knowledge by using **analogy** to reason about new situations that the way people can. Although rule induction helps, only limited types of knowledge can be put into an expert system this way. The customary way of building an expert system by having the knowledge engineer repeat the cycle of interviewing the expert, constructing a prototype, testing, interviewing, and so on is a very time consuming and labor intensive task. In fact, this problem of transferring human knowledge into an expert system is so major that it is called the **knowledge acquisition bottleneck**. This is a descriptive term because the knowledge acquisition bottleneck constricts the building of an expert system like an ordinary bottleneck constricts fluid flow into a bottle.

In spite of their present limitations, expert systems have been very successful in dealing with real-world problems that conventional programming methodologies have been unable to solve, especially those dealing with uncertain or incomplete information. The important point is to be aware of the advantages and limitations of this new technology so that it can be appropriately utilized.

1.5 CHARACTERISTICS OF AN EXPERT SYSTEM

An expert system is usually designed to have the following general characteristi[illegible]

- *High performance.* The system must be capable of responding at a level of competency equal to or better than an expert in the field. That is, the quality of the advice given by the system must be very high.
- *Adequate response time.* The system must also perform in a reasonable time, comparable to or better than the time required by an expert to reach a decision. An expert system that takes a year to reach a decision compared to an expert's time of one hour would not be too useful. The **time constraints** placed on the performance of an expert system may be especially severe in the case of real-time systems, when a response must be made within a certain time interval.
- *Good reliability.* The expert system must be reliable and not prone to crashes or else it will not be used.
- *Understandable.* The system should be able to explain the steps of its reasoning while executing so that it is understandable. Rather than being just a "black box" that produces a miraculous answer, the system should have an explanation capability in the same way that human experts can explain their reasoning. This feature is very important for several reasons.

One reason is that human life and property may depend on the answers of the expert system. Because of the great potential for harm, an expert system must be able justify its conclusions in the same way a human expert can explain why a certain conclusion was reached. Thus, an explanation facility provides an understandable check of the reasoning for humans.

A second reason for having an explanation facility occurs in the development phase of an expert system to confirm that the knowledge has been correctly acquired and is being correctly used by the system. This is very important in debugging since the knowledge may be incorrectly entered due to typos or be incorrect due to misunderstandings between the knowledge engineer and the expert. A good explanation facility allows the expert and knowledge engineer to verify the correctness of the knowledge. Also, because of the way that typical expert systems are constructed, it is very difficult to read a significant program listing and understand its operation.

An additional source of error may be unforeseen interactions in the expert system, which may be detected by running test cases with known reasoning that the system should follow. As we will discuss in more detail later, multiple rules may apply to a given situation about which the system is reasoning. The flow of execution is not sequential in an expert system so that you cannot just read its code line by line and understand how the system operates. That is, the order in which rules have been entered in the system is not necessarily the order in which they will be executed. The expert system acts much like a parallel program in which the rules are independent knowledge processors.

Flexibility. Because of the large amount of knowledge that an expert system may have, it is important to have an efficient mechanism for adding, changing, and deleting knowledge. One reason for the popularity of rule-based systems is the efficient and modular storage capability of rules.

Depending on the system, an explanation facility may be simple or elab. A simple explanation facility in a rule-based system may simply list all the fa that made the latest rule execute. More elaborate systems may do the following.

- *List all the reasons for and against a particular hypothesis.* A hypothesis is a goal which is to be proved such as "The patient has a tetanus infection" in a medical diagnostic expert system. In a real problem, there may be multiple hypotheses, just as a patient may have several diseases at once. A hypothesis can also be viewed as a fact whose truth is in doubt and must be proved.
- *List all the hypotheses that may explain the observed evidence.*
- *Explain all the consequences of a hypothesis.* For example, assuming that the patient does have tetanus, there should also be evidence of fever as the infection runs its course. If this symptom is then observed, it adds credibility that the hypothesis is true. If the symptom is not observed, it reduces the credibility of the hypothesis.
- *Give a* ***prognosis*** *or prediction of what will occur if the hypothesis is true.*
- *Justify the questions that the program asks of the user for further information.* These questions may be used to direct the line of reasoning to likely diagnostic paths. In most real problems, it is too expensive or take too long to explore all possibilities, and some way must be provided to guide the search for the correct solution. For example, consider the cost, time and effect of administ all possible medical tests to a patient complaining of a sore throat.
- *Justify the knowledge of the program.* For example, if the program claim the hypothesis "The patient has a tetanus infection" is true, the user could ask for an explanation. The program might justify this conclusion on the basis of a rule which says that if the patient has a positive blood test for tetanus, the patient has tetanus. Now the user could ask the program to justify this rule. The program could respond by stating that a positive blood test for a disease is proof of the disease.

In this case, the program is actually quoting a **metarule**, which is knowledge about rules. The prefix meta means above or beyond. Some programs such as Meta-DENDRAL have been explicitly created to infer new rules (Buchanan 78). A hypothesis is justified by knowledge and the knowledge is justified by a **warrant** that it is correct. A warrant is essentially a meta-explanation that explains the expert systems explanation of its reasoning.

Knowledge can easily grow **incrementally** in a rule-based system. That is, the knowledge base can grow little by little as rules are added so that the performance and correctness of the system can be continually checked. If the rules are properly designed, the interactions between rules will be minimized or eliminated to protect against unforeseen effects. The incremental growth of knowledge facilitates **rapid prototyping** so that the knowledge engineer can quickly show the expert a working prototype of the expert system. This is an important feature because it maint the expert's and management's interest in the project. Rapid prototyping also quickly

exposes any gaps, inconsistencies or errors in the expert's knowledge or the system so that corrections can immediately be made.

1.6 THE DEVELOPMENT OF EXPERT SYSTEMS TECHNOLOGY

AI has many branches concerned with speech, vision, robotics, natural language understanding, and learning and expert systems. The roots of expert systems lie in many disciplines. In particular, one of the major roots of expert systems is the area of human information processing, called **cognitive science**. Cognition is the study of how humans process information. In other words, cognition is the study of how people think, especially when solving problems.

The study of cognition is very important if we want to make computers emulate human experts. Often, experts can't explain how they solve problems—the solution just comes to them. If the expert can't explain how a problem is solved, it's not possible to encode the knowledge in an expert system based on explicit knowledge. In this case, the only possibility is programs that learn by themselves to emulate the expert. These are programs based on induction and artificial neural systems, to be discussed later.

Human Problem-Solving and Productions

The development of expert systems technology draws on a wide background. Table 1-2 is a brief summary of some important developments that have converged in modern expert systems. Whenever possible, the starting dates of projects are used. Many projects extend over many years. These developments are covered in more detail in this chapter and others. The best single reference for all the early systems is the three-volume *Handbook of Artificial Intelligence* (Feigenbaum 81).

In the late 1950's and 1960's, a number of programs were written with the goal of general problem solving. The most famous of these was the **General Problem Solver** created by Newell and Simon and described in a series of papers culminating in their monumental 920-page work on cognition, *Human Problem Solving* (Newell 72).

One of the most significant results demonstrated by Newell and Simon was that much of human problem solving or **cognition** could be expressed by IF THEN type **production rules**. For example, IF it looks like it's going to rain THEN carry an umbrella, or IF your spouse is in a bad mood THEN don't appear happy. A rule corresponds to a small, modular collection of knowledge called a **chunk**. The chunks are organized in a loose arrangement with links to related chunks of knowledge. One theory is that all human memory is organized in chunks. An example of a rule representing a chunk of knowledge is

Year	*Events*
1943	Post production rules; McCulloch and Pitts Neuron Model
1954	Markov Algorithm for controlling rule execution
1956	Dartmouth Conference; Logic Theorist; Heuristic Search; "AI" term coined
1957	Perceptron invented by Rosenblatt; GPS (General Problem Solver) started (Newell, Shaw and Simon)
1958	LISP AI language (McCarthy)
1962	Rosenblatt's *Principles of Neurodynamics* on Perceptions
1965	Resolution Method of automatic theorem proving (Robinson) Fuzzy Logic for reasoning about fuzzy objects (Zadeh) Work begun on DENDRAL, the first expert system (Feigenbaum, Buchanan, et.al.)
1968	Semantic nets, associative memory model (Quillian)
1969	MACSYMA math expert system (Martin and Moses)
1970	Work begins on PROLOG (Colmerauer, Roussell, *et. al.*)
1971	HEARSAY I for speech recognition *Human Problem Solving* popularizes rules (Newell and Simon)
1973	MYCIN expert system for medical diagnosis(Shortliffe, *et. al.*) leading to GUIDON, intelligent tutoring (Clancey) and TEIRESIAS, explanation facility concept (Davis) and EMYCIN, first shell (Van Melle, Shortliffe and Buchanan) HEARSAY II, blackboard model of multiple cooperating experts
1975	Frames, knowledge representation (Minsky)
1976	AM (Artificial Mathematician) creative discovery of math concepts (Lenat) Dempster-Shafer Theory of Evidence for reasoning under uncertainty Work begun on PROSPECTOR expert system for mineral exploration (Duda, Hart, *et. al.*)
1977	OPS expert system shell (Forgy), used in XCON/R1
1978	Work started on XCON/R1 (McDermott, DEC) to configure DEC computer systems Meta-DENDRAL, metarules and rule induction (Buchanan)
1979	Rete Algorithm for fast pattern matching (Forgy) Commercialization of AI begins Inference Corp. fomed (releases ART expert system tool in 1985)
1980	Symbolics, LMI founded to manufacture LISP machines
1982	SMP math expert system; Hopfield Neural Net; Japanese Fifth Generation Project to develop intelligent computers
1983	KEE expert system tool (IntelliCorp)
1985	CLIPS expert system tool (NASA)

Table 1-2
Some Important Events in the History of Expert Systems

```
IF the car doesn't run and
   the fuel gauge reads empty
THEN fill the gas tank
```

Newell and Simon popularized the use of rules to represent human knowledge and showed how reasoning could be done with rules. Cognitive psychologists have used rules as a model to explain human information processing. The basic idea is that sensory input provides stimuli to the brain. The stimuli trigger the appropriate rules of **long-term memory** which produce the appropriate response. Long-term memory is where our knowledge is stored. For example, we all have rules such as

```
IF there is flame THEN there is a fire
IF there is smoke THEN there may be a fire
IF there is a siren THEN there may be a fire
```

Notice that the last two rules are not expressed with complete certainty. The fire may be out, but there may still be smoke in the air. Likewise, a siren does not prove that there is a fire since it may be a false alarm. The stimuli of seeing flames, smelling smoke, and hearing a siren will trigger these and similar types of rules.

Long-term memory consists of many rules having the simple IF THEN structure. ... fact, a Grand Master chess expert may know 50,000 or more chunks of knowledge about chess patterns. In contrast to the long-term memory, the **short-term memory** is used for the temporary storage of knowledge during problem solving. Although long-term memory can hold hundreds of thousands or more chunks, the capacity of working memory is surprisingly small—4 to 7 chunks. As a simple example of this, try visualizing some numbers in your mind. Most people can only see 4 to 7 numbers at once. Of course we can memorize many more than 4 to 7 numbers. However, those numbers are kept in long-term memory.

One theory proposes that short-term memory represents the number of chunks that simultaneously can be active and considers human problem-solving as a spreading of these activated chunks in the mind. Eventually, a chunk may be activated with such intensity that a conscious thought is generated and you say to yourself, "Hmm.. something's burning."

The other element necessary for human-problem solving is a **cognitive processor**. The cognitive processor tries to find the rules that will be **activated** by the appropriate stimuli. Not just any rule will do. For example, you wouldn't want to fill your gas tank every time you heard a siren. Only a rule which matched the stimuli would be activated. If there are multiple rules that are activated at once, the cognitive processor must perform a conflict resolution to decide which rule has highest priority. The rule with highest priority will be executed. for example, if both of the following les are activated

```
IF there is a fire THEN leave
IF my clothes are burning THEN put out the fire
```

then the actions of one rule will be executed before the other. The **inference** ne of modern expert systems corresponds to the cognitive processor.

The Newell and Simon model of human problem-solving in terms of long-term memory (rules), short-term memory (working memory), and a cognitive processor (inference engine) is the basis of modern rule-based expert systems.

Rules like these are a type of **production system**. Rule-based production systems are a popular method of implementing expert systems today. The individual rules that comprise a production system are the **production rules**. In designing an expert system, an important factor is the amount of knowledge or **granularity** of the rules. Too little granularity makes it difficult to understand a rule without reference to other rules. Too much granularity makes the expert system difficult to modify since several chunks of knowledge are intermingled in one rule.

Until the mid-sixties, a major quest of AI was to produce intelligent systems that relied little on domain knowledge and greatly on powerful methods of reasoning. Even the name General Problem Solver illustrates the concentration on machines that were not designed for one specific domain but were intended to solve many types of problems. Although the methods of reasoning used by general problem solvers were very powerful, the machines were eternal beginners. When presented with a new domain, they had to discover everything from first principles and were not as good as human experts who relied on domain knowledge for high performance.

An example of the power of knowledge is playing chess. The best chess r' ers are humans despite the fact that computers are much faster than people in doin, u-lations. Studies have shown (Chase 73) that human expert chess players do not have super powers of reasoning but instead rely on knowledge of chesspiece patterns built up over years of play. As mentioned previously, one estimate places an expert chess player's knowledge at about 50,000 patterns. Humans are very good at recognizing patterns such as pieces on a chessboard. Instead of trying to reason ahead ten or twenty possible moves for every piece as a computer might, the human analyzes the game in terms of patterns that reveal long-term threats while remaining alert for short-term surprise moves.

While domain knowledge is very powerful, it is generally limited to the domain. For example, a person who becomes an expert chess player does not automatically become an expert at solving math problems or even a checkers expert. While some knowledge may carry over to another domain, such as careful planning of moves, this is a skill rather than genuine expertise.

By the early seventies, it became apparent that domain knowledge was the key to building machine problem-solvers that could function at the level of human experts. Although methods of reasoning are important, studies have shown that experts do not primarily rely on reasoning for problem-solving. In fact, reasoning may play a minor role in an expert's problem solving. Instead, experts rely on a vast knowledge of heuristics and experience that they have built up over the years. If an expert cannot solve a problem based on expertise, then it is necessary for the expert to reasor -om first principles and theory (or more likely ask another expert). The reasoning ab. of an expert is generally no better than that of an average person in dealing with a totally

unfamiliar situation. The early attempts at building powerful problem-solvers based only on reasoning have shown that a problem-solver is crippled if it must rely solely on reasoning.

The insight that domain knowledge was the key to building real-world problem solvers led to the success of expert systems. The successful expert systems today are thus knowledge-based expert systems rather than general problem solvers. In addition, the same technology that led to the development of expert systems has also led to the development of knowledge-based systems that do not necessarily contain human expertise.

While expertise is considered knowledge that is specialized and known only to a few, knowledge is generally found in books, periodicals and other widely available resources. For example, the knowledge of how to solve a quadratic equation or perform integration and differentiation is widely available. Knowledge-based computer programs such as MACSYMA and SMP are available to perform automatically these and many other mathematical operations on either numeric or symbolic operands. Other knowledge-based programs may perform process control of manufacturing plants. Today the terms **knowledge-based programming** and expert systems are often used synonymously. In fact, expert systems is now considered an alternative programming model or **paradigm** to conventional algorithmic programming.

The Rise of Knowledge-based Systems

With the acceptance of the knowledge-based paradigm in the 1970's, a number of successful prototype expert systems were created. These systems could interpret mass spectrograms to identify chemical constituents (DENDRAL), diagnose illness (MYCIN), analyze geologic data for oil (DIPMETER) and minerals (PROSPECTOR), and configure computer systems (XCON/R1). The news that PROSPECTOR had discovered a mineral deposit worth $100 million dollars and that XCON/R1 was saving Digital Equipment Corporation (DEC) millions of dollars a year triggered a sensational interest in the new expert systems technology by 1980. The branch of AI that had started off in the 1950's as a study of human information processing had now grown to achieve commercial success by the development of practical programs for real-world use.

The MYCIN expert system was very important for several reasons. First, it demonstrated that AI could be used for practical real-world problems. Second, MYCIN was the testbed of new concepts such as the explanation facility, automatic acquisition of knowledge, and intelligent tutoring that are found in a number of expert systems today. The third reason that MYCIN was important is that it demonstrated the feasibility of the expert system **shell**.

Previous expert systems such as DENDRAL were one-of-a-kind systems in which the knowledge-base was intermingled with the software that applied the knowledge, the inference engine. MYCIN explicitly separated the knowledge-base from the inference engine. This was extremely important to the development of expert system technology since it meant that the essential core of the expert system could be reused.

That is, a new expert system could be built much more rapidly than a DENDRAL e system by emptying out the old knowledge and putting in knowledge about the new domain. The part of MYCIN that dealt with inference and explanation, the shell, could then be refilled with knowledge about the new system. The shell produced by removing the medical knowledge of MYCIN was called EMYCIN (Essential or Empty MYCIN).

By the late seventies, the three concepts that are basic to most expert systems today had converged, as shown in Figure 1-5. These concepts are rules, the shell, and knowledge.

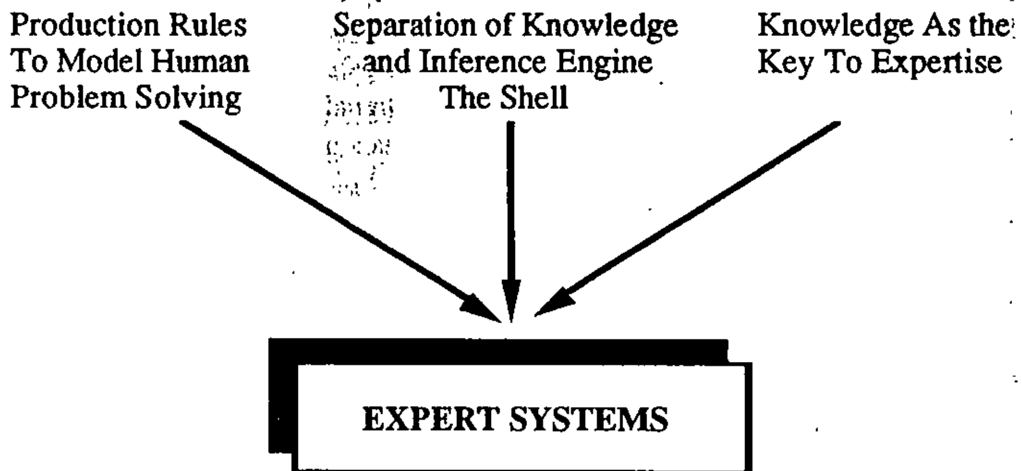


Figure 1-5
Convergence Of Three Important Factors To Create
The Modern Rule-based Expert System

By 19:30 new companies started to bring expert systems out of the university laboratory : nd produce commercial products. Powerful new software for expert system development was introduced such as the Automated Reasoning Tool (ART) by the Inference Corp., the Knowledge Engineering Tool (KEE) by IntelliCorp, and Rulemaster by Radian Corp. In addition, specialized new hardware was developed to run the software with greater speed than ever before. Companies such as Symbolics and LMI introduced computers referred to as LISP machines because they were designed for LISP, the underlying base language of the expert systems development software. In LISP machines, the native assembly language, operating system, and all other fundamental code were done in LISP.

CLIPS

Unfortunately, this high technology also commanded a high price. While a single-user LISP machine was much more powerful and productive than a general purpose machine running LISP, the single-user machine and a single-user software license cost a tot- f $100,000. Establishing an AI lab with a half-dozen programmers could easily c half-million dollars.

These objections were overcome in the mid-80's by the introduction of powerful development software such as CLIPS by NASA. CLIPS is written in C for speed and portability, and also uses a powerful pattern matching called the Rete Algorithm. CLIPS is free to Government users and Government contractors. The cost to other users is $312 (as of 1988) which includes the complete 25,000 lines of CLIPS source code.

Any C compiler that supports the standard Kernigan and Richie C language can be used to install CLIPS. CLIPS has been installed on the IBM PC and compatibles, VAX, Hewlett-Packard, Sun, Cray and many other makes of computers. A version for the Macintosh is also available that supports the full Macintosh interface with pull-down windows and mouse support. The speed of CLIPS on a 80386 or 68020 microcomputer is compatible with the state-of-the-art LISP-based development software and LISP machines.

1.7 EXPERT SYSTEMS APPLICATIONS AND DOMAINS

Conventional computer programs are used to solve many types of problems. Generally, these problems have algorithmic solutions that lend themselves well to conventional grams and programming languages such as FORTRAN, Pascal, Ada, and so on. In any application areas such as business and engineering, numeric calculations are of primary importance. By contrast, expert systems are primarily designed for symbolic reasoning.

While languages such as LISP and PROLOG are also used for symbolic manipulation, they are more general purpose than expert system shells. This does not mean that it is not possible to build expert systems in LISP and PROLOG. In fact, many expert systems have been built with PROLOG and LISP. PROLOG especially has a number of advantages for diagnostic systems because of its built-in backward chaining. Rather, it is more convenient and efficient to build large expert systems with shells and utility programs specifically designed for expert system building. Instead of "re-inventing the wheel" every time a new expert system is to be built, it is more efficient to use specialized tools designed for expert system building than general purpose tools.

Applications of Expert Systems

Expert systems have been applied to virtually every field of knowledge. Some have been designed as research tools while others fulfill important business and industrial functions. One example of an expert system in routine business use is the XCON system of Digital Equipment Corp. (DEC). The XCON system (originally called R1) was developed in conjunction with John McDermott of Carnegie-Mellon University. XCON is an expert configuring system for DEC computer systems (McDermott 84).

The **configuration** of a computer system means that when a customer places an order, all the right parts—software, hardware, and documentation—are supplied. For

large systems, the customer's system is set up or configured at the factory and ' ' to insure that it meets the customer's requirements. Unlike the purchase of a TV . or a home computer, there are many options and interconnections possible with a large computer system. In putting together a large system, it's not enough to just ship the requested number of CPU's, disk drives, terminals and so forth. The proper interconnections and cabling must also be supplied and the system checked to verify that it is working correctly.

The XCON system is probably one of the most successful expert systems in routine use and saves DEC millions of dollars a year, reduces time to configure an order, and improves the accuracy of an order. XCON can configure an average order in about two minutes, which is fifteen times faster than a human. Also, while the humans configured orders correctly 70% of the time, XCON has an accuracy of 98%. These are important concerns since configuring a complex computer system at the factory involves considerable effort. It is very expensive to configure a system partially and then find out the system cannot meet the customer's requirements or that other components are needed and shipment will be delayed until the parts arrive.

Hundreds of expert systems have been built and reported in computer journals, books, and conferences. This probably represents only the tip of the iceberg since many companies and military organizations will not report their systems because of proprietary or secret knowledge contained in the systems. Based on the systems described in the open literature, certain broad classes of expert systems applications c-- be discerned, as shown in Table 1-3.

Class	*General Area*
Configuration	Assemble proper components of a system in the proper way.
Diagnosis	Infer underlying problems based on observed evidence.
Instruction	Intelligent teaching so that a student can ask *Why*, *How* and *What If* type questions just as if a human was teaching.
Interpretation	Explain observed data.
Monitoring	Compares observed data to expected data to judge performance.
Planning	Devise actions to yield a desired outcome.
Prognosis	Predict the outcome of a given situation.
Remedy	Prescribe treatment for a problem.
Control	Regulate a process. May require interpretation, diagnosis, monitoring, planning, prognosis, and remedies.

Table 1-3
Broad Classes of Expert Systems

Examples of some expert systems are shown in Tables 1-4 through 1-9 (Waterman 86), and in the Historical Bibliography at the end of the chapter.

Name	*Chemistry*
CRYSALIS	Interpret a protein's 3-D structure
DENDRAL	Interpret molecular structure
TQMSTUNE	Remedy Triple Quadruple Mass Spectrometer (keep it tuned)
CLONER	Design new biological molecules
MOLGEN	Design gene-cloning experiments
SECS	Design complex organic molecules
SPEX	Plan molecular biology experiments

Table 1-4
Chemistry Expert Systems

Name	*Electronics*
ACE	Diagnosis telephone network faults
IN-ATE	Diagnosis oscilloscope faults
NDS	Diagnose national communication net
EURISKO	Design 3-D microelectronics
PALLADIO	Design and test new VLSI circuits
REDESIGN	Redesign digital circuits to new
CADHELP	Instruct for computer aided design
SOPHIE	Instruct circuit fault diagnosis

Table 1-5
Electronics Expert Systems

Name	*Medicine*
PUFF	Diagnosis lung disease
VM	Monitors intensive-care patients
ABEL	Diagnosis acid-base/electrolytes
AI/COAG	Diagnosis blood disease
AI/RHEUM	Diagnosis rheumatoid disease
CADUCEUS	Diagnosis internal medicine disease
ANNA	Monitor digitalis therapy
BLUE BOX	Diagnosis/remedy depression
MYCIN	Diagnosis/remedy bacterial infections
ONCOCIN	Remedy/manage chemotherapy patients
ATTENDING	Instruct in anesthetic management
GUIDON	Instruct in bacterial infections

Table 1-6
Medical Expert Systems

Name	*Engineering*
REACTOR	Diagnosis/remedy reactor accidents
DELTA	Diagnosis/remedy GE locomotives
STEAMER	Instruct operation - steam powerplant

Table 1-7
Engineering Expert Systems

Name	*Geology*
DIPMETER	Interpret dipmeter logs
LITHO	Interpret oil well log data
MUD	Diagnosis/remedy drilling problems
PROSPECTOR	Interpret geologic data for minerals

Table 1-8
Geology Expert Systems

Name	*Computer Systems*
PTRANS	Prognosis for managing DEC computers
BDS	Diagnosis bad parts in switching net
XCON	Configure DEC computer systems
XSEL	Configure DEC computer sales order
XSITE	Configure customer site for DEC computers
YES/MVS	Mon tor/control IBM MVS operating system
TIMM	Diagnosis DEC computers

Table 1-9
Computer Expert Systems

Appropriate Domains for Expert Systems

Before starting to build an expert system, it is essential to decide if an expert system is the appropriate paradigm. For example, one concern is whether an expert system should be used instead of an alternative paradigm such as conventional programming. The appropriate domain for an expert system depends on a number of factors.

- *Can the problem be solved effectively by conventional programming?* If the answer is yes, then an expert system is not the best choice. For example, consider the problem of diagnosing some equipment. If all the symptoms for all malfunctions are known in advance, then a simple table lookup or decision tree of the fault is adequate. Expert systems are best suited for situations in which there is no efficient algorithmic solution. Such cases are called **ill-structured problem** d reasoning may offer the only hope of a good solution.

As an example of an ill-structured problem, consider the case of a person who is thinking about travel and visits a travel agent. While most people have a destination and plans, there are exceptions. Table 1-10 lists some characteristics of an ill-structured problem as indicated by the person's responses to the travel agent's questions.

Travel Agent's Questions	*Responses*
Can I help you?	I'm thinking about going somewhere
Where do you want to go?	I'm not sure where to go
Any particular destination?	I just like to travel; destination's not important
How much can you afford?	I don't have enough money to go
Can you get some money?	I don't know how to get the money
When do you want to go?	I must go soon

Table 1-10
An Example of an Ill-structured Problem

Although this is probably an extreme case, it does illustrate the basic concept of an ill-structured problem. As you can see, an ill-structured problem would not lend itself well to an algorithmic solution because there are so many possibilities.

In dealing with ill-structured problems, there is a danger in that the expert system design may accidentally mirror an algorithmic solution. That is, the development of the expert system may unknowingly discover an algorithmic solution. A clue that this has happened occurs if a solution is found that requires a rigid **control structure**. That is, the rules are forced to execute in a certain sequence by the knowledge engineer explicitly setting the priorities of many rules. Forcing a rigid control structure on the expert system cancels a major advantage of expert system technology, which is dealing with unexpected input that does not follow a predetermined pattern (Parrello 88). That is, expert systems react opportunistically to their input, whatever it is. Conventional programs generally expect input to follow a certain sequence. An expert system with a lot of control often indicates a disguised algorithm and may be a good candidate for recoding as a conventional program.

- *Is the domain well-bounded?* It is very important to have well-defined limits on what the expert system is expected to know and what its capabilities should be. For example, suppose you wanted to create an expert system to diagnose headaches. Certainly medical knowledge of a physician would be put in the knowledge-base. However, for a deep understanding of headaches, you might also put in knowledge about neurochemistry, then its parent area of biochemistry, then chemistry, molecular biophysics, and so forth perhaps down to subnuclear physics. Other domains such as biofeedback, psychology, psychiatry, physiology, exercise, yoga, and stress management may also have pertinent knowledge about headaches. The point of all this is—when do you stop adding domains? The more domains, the more complex the expert system becomes.

In particular, the task of coordinating all the expertise becomes a major task. I e real world, we know from experience how difficult it is to have coordinated teams of experts working on problems, especially when they come up with conflicting recommendations. If we knew how to program well the coordination of expertise, then we could try programming an expert system to have the knowledge of multiple experts. Attempts have been made to coordinate multiple expert systems in the HEARSAY II and HEARSAY III systems. However, it is a complex task that should be viewed more as research rather than producing a deliverable expert system product.

- *Is there a need and a desire for an expert system?* Although it's great experience to build an expert system, it's rather pointless if no one is willing to use it. If there already are many human experts, it's difficult to justify an expert system based on the reason of scarce human expertise. Also, if the experts or users don't want the system, it will not be accepted even if there is a need for it.

Management especially must be willing to support the system. This is even more critical for expert systems than conventional programs because expert systems is a new technology. This means that there are few experienced people and more uncertainty about what can be accomplished. However, the area of expert systems deserves more support because it attempts to solve the problems that cannot be done by conventional programming. The risks are greater but so are the rewards.

- *Is there at least one human expert who is willing to cooperate?* There must be an expert who is willing, and preferably enthusiastic, about the project. Not all experts are willing to have their knowledge examined for faults and then put into a computer. Even if there are multiple experts willing to cooperate in the development, it might be wise to limit the number of experts involved in development. Different experts may have different ways of solving a problem, such as requesting different diagnostic tests. Sometimes, they may even reach different conclusions. Trying to code multiple methods of problem-solving in one knowledge-base may create internal conflicts and incompatibilities.

- *Can the expert explain the knowledge so that it is understandable by the knowledge engineer?* Even if the expert is willing to cooperate, there may be difficulty in expressing the knowledge in explicit terms. As a simple example of this difficulty, can you explain in words how you move a finger? Although you could say it's done by contracting a muscle in the finger, the next question is—how do you contract a finger muscle? The other difficulty in communication between expert and knowledge engineer is that the knowledge engineer doesn't know the technical terms of the expert. This problem is particularly acute with medical terminology. It may take a year or longer for the knowledge engineer to even understand what the expert is talking about, let alone translate that knowledge into explicit computer code.

- *Is the problem-solving knowledge mainly heuristic and uncertain?* Expert systems *are* appropriate when the expert's knowledge is largely heuristic and uncertain. That is, the knowledge may be based on experience, called **experiential knowledge**, and the expert may have to try various approaches in case one doesn't work. In other words, the expert's knowledge may be a trial-and-error approach, rather than one based on logic and algorithms. However, the expert can still solve the problem faster than someone else who is not an expert. This is a good application for expert systems. If the problem can be solved simply by logic and algorithms, it is best solved by a conventional program.

1.8 LANGUAGES, SHELLS AND TOOLS

A fundamental decision in defining a problem is deciding how best to model it. Sometimes experience is available to aid in choosing the best paradigm. For example, experience suggests that a payroll is best done using conventional procedural programming. Experience also suggests that it is preferable to use a commercial package, if available, rather than writing one from scratch. A general guide to selecting a paradigm is to consider the most traditional one first—conventional programming. ason for doing this is because of the vast amount of experience we have with ntional programming and the wide variety of commercial packages available. If a problem cannot be effectively done by conventional programming, then turn to non-conventional paradigms such as AI.

Although expert systems is a branch of AI, there are specialized languages for expert systems that are quite different from the commonly used AI languages such as LISP and PROLOG. While many others have been developed, such as IPL-II, SAIL, CONNIVER, KRL and Smalltalk, few are widely used except for research (Scown 85).

An expert system language is a higher order language than languages like LISP or C because it is easier to do certain things, but there is also a smaller range of problems that can be addressed. That is, the specialized nature of expert system languages makes them very suitable for writing expert systems but not for general purpose programming. In many situations, it is even necessary to exit temporarily from an expert system language to perform a function in a procedural language.

The primary functional difference between expert system languages and procedural languages is the focus of representation. Procedural languages focus on providing flexible and robust techniques to represent data. For example, data structures such as arrays, records, linked lists, stacks, queues, and trees are easily created and manipulated. Modern languages such as Modula-2 and Ada are designed to aid in **data abstraction** by providing structures for encapsulation such as modules and packages. This provides a level of abstraction which is then implemented by methods such as operators and ol statements to yield a program. The data and methods to manipulate the data are ly interwoven. In contrast, expert system languages focus on providing flexible and robust ways to represent knowledge. The expert system paradigm allows two levels of abstraction: data abstraction and **knowledge abstraction**. Expert system

languages specifically separate the data from the methods of manipulating the dat n example of this separation is that of facts (data abstraction) and rules (knowledge abstraction) in a rule-based expert system language.

This difference in focus also leads to a difference in program design methodology. Because of the tight interweaving of data and knowledge in procedural languages, programmers must carefully describe the sequence of execution. However, the explicit separation of data from knowledge in expert system languages requires considerably less rigid control of execution sequence. Typically, an entirely separate piece of code, the inference engine, is used to apply the knowledge to the data. This separation of knowledge and data allows a higher degree of parallelism and modularity.

In choosing a language, a basic question should be whether the problem is knowledge or intelligence intensive. Expert systems rely on a great deal of specialized knowledge or expertise to solve a problem, while AI emphasizes a problem solving approach. Expert systems often rely on a pattern matching within a restricted knowledge domain to guide their execution while AI generally concentrates on searching paradigms in less restricted domains.

The customary way of defining the need for an expert system program is to decide if you want to program the expertise of a human expert. If such an expert exists and will cooperate, then an expert system approach may be successful.

The road to selecting an expert system language is paved with confusion. A few years ago the choice of an expert system language was fairly straightforward. T e were only about a half-dozen languages available, and they were generally free or nominal amount from the universities where they were developed.

However, with the explosive commercial growth in the expert systems field since the 1970's, the selection of a language is no longer so simple. Today there are dozens of languages available ranging in price up to $75,000. While it is still possible to obtain some of the older languages such as OPS5 free or at a nominal cost from the universities where they were developed, there is a price to pay in terms of efficiency, lack of modern features and support.

Besides the confusing choice of the many languages available today, the terminology used to describe the languages is confusing. Some vendors refer to their products as "tools," while others refer to "shells" and still others talk about "integrated environments". For clarity in this book, the terms will be defined as follows:

- **language**: A translator of commands written in a specific syntax. An expert system language will also provide an inference engine to execute the statements of the language. Depending on the implementation, the inference engine may provide forward chaining, backward chaining, or both. Under this language definition, LISP is not an expert system language while PROLOG is. However, it is possible to write an expert system language using LISP, and write AI in PROLOG. For that matter you can even write an expert system or AI language in assembly language. Questions of development time, convenience, maintainability, efficiency and speed determine what language software is written in.
- **tool**: A language plus associated utility programs to facilitate the developm debugging, and delivery of application programs. Utility programs may include text and

graphics editors, debuggers, file management, and even code generators. Cross assemblers may also be provided to port the developed code to different hardware. For example, an expert system may be developed on a Digital Equipment Corp. VAX and then cross-assembled to run on a Motorola 68000. Some tools may even allow the use of different paradigms such as forward and backward chaining in one application.

In some cases, a tool may be integrated with all its utility programs in one environment to present a common interface to the user. This approach minimizes the need for the user to leave the environment to perform a task. For example, a simple tool may not provide facilities for file management and so a user would have to exit the tool to give operating system commands. An integrated environment allows easy exchange of data between utility programs in the environment. Some tools do not even require the user to write any code. Instead, the tool allows a user to enter knowledge by examples from tables or spreadsheets and generate the appropriate code itself.

- **shell**: A special purpose tool designed for certain types of applications in which the user must only supply the knowledge base. The classic example of this is the EMYCIN (empty MYCIN) shell. This shell was made by removing the medical knowledge base of the MYCIN expert system.

MYCIN was designed as a backward chaining system to diagnose disease. By simply removing the medical knowledge, EMYCIN was made which could be used as a shell to contain knowledge about other kinds of consultative systems which use backward chaining. The EMYCIN shell demonstrated the reusability of the essential MYCIN software such as the inference engine and user interface. This was a very important step in the development of modern expert system technology because it meant that an expert system would not have to be built from scratch for each new application.

There are many ways of characterizing expert systems such as representation of knowledge, forward or backward chaining, support of uncertainty, hypothetical reasoning, explanation facilities and so forth. Unless a person has built a number of expert systems, it is difficult to appreciate all of these features, especially those found in the more expensive tools. The best way to learn expert systems technology is to develop a number of systems with an easy-to-learn language and then invest in a more sophisticated tool if you need its features.

1.9 ELEMENTS OF AN EXPERT SYSTEM

The elements of a typical expert system are shown in Figure 1-6. In a rule-based system, the knowledge base contains the domain knowledge needed to solve problems coded in the form of rules. While rules are a popular paradigm for representing knowledge, other types of expert systems use different representations, as discussed in Chapter 2.

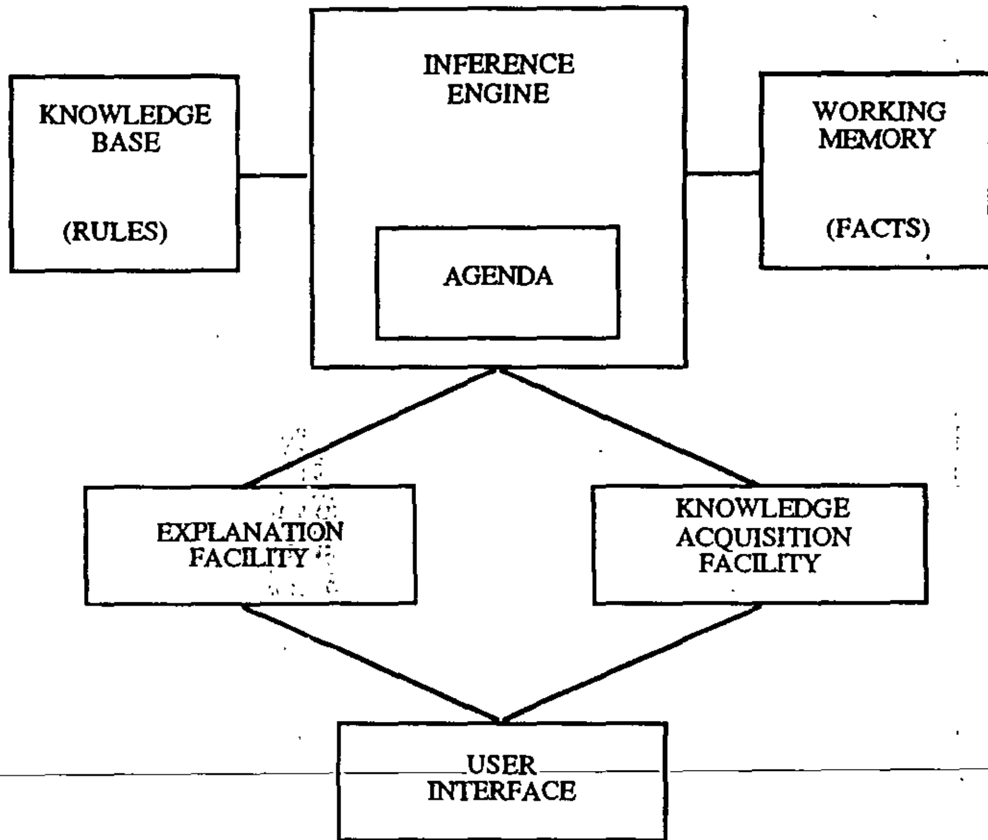


Figure 1-6
Structure of a Rule-Based Expert System

An expert system consists of the following components:

- **user interface** - the mechanism by which the user and the expert system communicate.
- **explanation facility** - explains the reasoning of the system to a user.
- **working memory** - a **global database** of facts used by the rules.
- **inference engine** -makes inferences by deciding which rules are satisfied by facts, prioritizes the satisfied rules, and executes the rule with the highest priority.
- **agenda** - a prioritized list of rules created by the inference engine, whose patterns are satisfied by facts in working memory.
- **knowledge acquisition facility** - an automatic way for the user to enter knowledge in the system rather than by having the knowledge engineer explicitly code the knowledge.

The knowledge acquisition facility is an an optional feature on many systems. In some expert system tools like KEE and First Class, the tool can learn by rule induction through examples and automatically generate rules. However, the example generally from tabular or spreadsheet type data better suited to decision trees. Ge

rules constructed by a knowledge engineer can be much more complex than the simple rules from rule induction.

Depending on the implementation of the system, the user interface may be a simple text-oriented display or a sophisticated high-resolution, bit-mapped display. This high-resolution display is commonly used to simulate a control panel with dials and displays.

The knowledge base is also called the **production memory** in a rule-based expert system. As a very simple example, consider the problem of deciding to cross a street. The productions for the two rules are as follows, where the arrows mean that the system will perform the actions on the right of the arrow if the conditions on the left are true.

```
the light is red → stop
the light is green → go
```

The production rules can be expressed in an equivalent pseudocode IF THEN format as:

```
Rule: Red_light
IF
       the light is red
THEN
       stop

Rule: Green_light
IF
       the light is green
THEN
       go
```

Each rule is identified by a name. Following the name is the IF part of the rule. The section of the rule between the IF and THEN part of the rule is called by various names such as the **antecedent**, **conditional part**, **pattern part** or **left-hand-side** (LHS). The individual condition

```
the light is green
```

is called a **conditional element** or a **pattern**.

Some examples of rules from real systems are:

```
MYCIN system for diagnosis of meningitis and
bacteremia (bacterial infections)
IF
    The site of the culture is blood, and
```

```
    The identity of the organism is not known with
       certainty, and
    The stain of the organism is gramneg, and
    The morphology of the organism is rod, and
    The patient has been seriously burned
THEN
    There is weakly suggestive evidence (.4) that the
        identity of the organism is pseudomonas
```

XCON/R1 for configuring DEC VAX computer systems

```
IF
    The current context is assigning devices to
       Unibus modules and
    There is an unassigned dual-port disk drive and
    The type of controller it requires is known and
    There are two such controllers, neither of which
       has any devices assigned to it, and
    The number of devices that these controllers can
       support is known
THEN
    Assign the disk drive to each of the controllers
       and
    Note that the two controllers have been
       associated and that each supports one drive
```

In a rule-based system, the inference engine determines which rule antecedents, if any, are satisfied by the facts. Two general methods of inferencing are commonly used: **forward chaining** and **backward chaining** as the problem solving strategies of expert systems. Other methods used for more specific needs may include means-ends analysis, problem reduction, backtracking, plan-generate-test, hierarchical planning and the least commitment principle, and constraint handling.

Forward chaining is reasoning from facts to the conclusions resulting from those facts. For example, if you see that it is raining before leaving home (the fact), then you should take an umbrella (the conclusion).

Backward chaining involves reasoning in reverse from a hypothesis, a potential conclusion to be proved, to the facts which support the hypothesis. For example, if you have not looked outside and someone enters with wet shoes and an umbrella, your hypothesis is that it is raining. In order to support this hypothesis, you could ask the person if it was raining. If the response is yes, then the hypothesis is proven true and becomes a fact. As mentioned before, a hypothesis can be viewed as a fact whose truth is in doubt and needs to be established. The hypothesis can then be interpreted as a goal to be proven.

Depending on the design, an inference engine will do either forward or backward chaining. For example, OPS5 and CLIPS are designed for forward chaining while EMYCIN performs backward chaining. Some types of inference engines, such as ART and KEE, offer both. The choice of inference engine depends on the type of problem. Diagnostic problems are better solved with backward chaining while prognosis, monitoring and control are better done by forward chaining.

The working memory may contain facts regarding the current status of the traffic light such as "the light is green" or "the light is red". Either or both of these facts may be in working memory at the same time. If the traffic light is working normally, only one fact will be in memory. However, it is possible that both facts may be in working memory if there is a malfunction in the light. Notice the difference between the knowledge base and working memory. Facts do not interact with one another. The fact "the light is green" has no effect on the fact "the light is red". Instead, our knowledge of traffic lights says that if both facts are simultaneously present, then there is a malfunction in the light.

If there is a fact "the light is green" in working memory, the inference engine will notice that this fact satisfies the conditional part of the green light rule and put this rule on the agenda. If a rule has multiple patterns, then all of its patterns must be simultaneously satisfied for the rule to be placed on the agenda. Some patterns may even be sfied by specifying the absence of certain facts in working memory.

A rule whose patterns are all satisfied is said to be **activated** or **instantiated**. Multiple activated rules may be on the agenda at the same time. In this case, the inference engine must select one rule for **firing**. The term firing comes from Neurophysiology, the study of the nervous system. An individual nerve cell or **neuron** emits an electrical signal when stimulated. No amount of further stimulation can cause the neuron to fire again for a short time period. This phenomenon is called refraction. Rule-based expert systems are built using refraction in order to prevent trivial loops. That is, if the green light rule kept firing on the same fact over and over again, the expert system would never accomplish any useful work.

Various methods have been invented to provide refraction. In one type of expert system language called OPS5, each fact is given a unique identifier called a **timetag** when it is entered in working memory. After a rule has fired on a fact, the inference engine will not fire on that fact again because its time stamp has been used.

Following the THEN part of a rule is a list of **actions** to be executed when the rule fires. This part of the rule is known as the **consequent** or **right-hand side (RHS)**. When the red light rule fires, it's action "stop" is executed. Likewise, when the green light rule fires, it's action is "go". Specific actions usually include the addition or removal of facts from working memory or printing results. The format of these actions depends on the syntax of the expert system language. For example, in OPS5, ART, and CLIPS, the action to add a new fact called "stop" to working memory would be sert stop). Because of their LISP ancestry, these languages were designed to require parentheses around patterns and actions.

The inference engine operates in **cycles**. Various names have been given to describe the cycle such as **recognize-act cycle**, **select-execute cycle**,

situation-response cycle, and **situation-action cycle**. By any name a cycle, the inference engine will repeatedly execute a group of tasks until certain criteria cause execution to cease. The tasks of a cycle for OPS5, a typical expert system shell, are shown in the following pseudocode as **conflict resolution, act, match,** and **check for halt.**

WHILE not done

- **Conflict Resolution**: If there are activations, then select the one with highest priority else done.
- **Act**: Sequentially perform the actions on the RHS of the selected activation. Those which change working memory have immediate effect in this cycle. Remove the activation which has just fired from the agenda.
- **Match**: Update the agenda by checking if the LHS of any rules are satisfied. If so activate them. Remove activations if the LHS of their rules are not satisfied any more.
- **Check for Halt**: If a halt action is performed or break command given, then done.

END-WHILE

Accept a new user command

Multiple rules may be activated and put on the agenda during one cycle. Also, activations will be left on the agenda from previous cycles unless they are deactivated because their LHS is no longer satisfied. Thus the number of activations on the agenda will vary as execution proceeds. Depending on the program, an activation may always be on the agenda but never selected for firing. Likewise some rules may never become activated. In these cases, the purpose of these rules should be re-examined because the rules are either unnecessary or their patterns were not correctly designed.

The inference engine executes the actions of the highest priority activation on the agenda, then the next highest priority activation and so on until no activations are left. Various priority schemes have been designed into expert system shells. Generally, all shells let the knowledge engineer define the priority of rules.

Agenda conflicts occur when different activations have the same priority and the inference engine must decide on one rule to fire. Different shells have different ways of dealing with this problem. In the original Newell and Simon paradigm, those rules entered first in the system had the highest default priority (Newell 72a). In OPS5, rules with more complex patterns have a higher priority. In ART and CLIPS, rules have the same default priority unless assigned different ones by the knowledge engineer.

At this time, control is returned to the **top-level** command interpreter for the user to give further instructions to the expert system shell. The top-level is the default mode in which the user communicates with the expert system and is indicated by the task "Accept a new user command". It is the top-level which accepts the new command.

The top-level is the user interface to the shell while an expert system application is under development. More sophisticated user interfaces are usually designed for the expert system to facilitate its operation. For example, the expert system may have a user interface for control of a manufacturing plant that shows a block diagram of the plant with a high resolution, bit-mapped color display. Warnings and status messages may appear in flashing colors with simulated dials and gauges. In fact, more effort may go into the design and implementation of the user interface than in the expert system knowledge base, especially in a prototype. Depending on the capabilities of the expert system shell, the user interface may be implemented by rules or in another language called by the expert system.

An explanation facility will allow the user to ask how the system came to a certain conclusion and why certain information is needed. The question of how the system came to a certain conclusion is easy to answer in a rule-based system since a history of the activated rules and contents of working memory can be maintained in a stack. Sophisticated explanation facilities may allow the user to ask "What If" type questions to explore alternate reasoning paths through hypothetical reasoning.

1.10 PRODUCTION SYSTEMS

, of the most popular type of expert system today is the rule-based system. Rules are popular for a number of reasons.

- *Modular nature.* This makes it easy to encapsulate knowledge and expand the expert system by incremental development.
- *Explanation facilities.* It is easy to build explanation facilities with rules since the antecedents of a rule specify exactly what is necessary to activate the rule. By keeping track of which rules have fired, an explanation facility can present the chain of reasoning that led to a certain conclusion.
- *Similarity to the human cognitive process.* Based on the work of Newell and Simon, rules appear to be a natural way of modeling how humans solve problems. The simple IF THEN representation of rules make it easy to explain to experts the structure of the knowledge that you are trying to elicit from them. Other advantages of rules are described in (Hayes-Roth 85).

Rules are a type of production whose origins go back to the 1940's. Because of the importance of rule-based systems, it is worthwhile to examine the development of the rule concept. This will give you a better idea of why rule-based systems are so useful for expert systems.

~`st Production Systems

Production systems were first used in symbolic logic by Post (Post 43) who originated the name. He proved the important and amazing result that any system of mathematics

or logic could be written as a certain type of production rule system. This result e lished the great capability of production rules for representing major classes of knowledge rather than being limited to a few types. Under the term **rewrite rules**, they are also used in linguistics as a way of defining the grammar of a language. Computer languages are commonly defined using the Backus-Naur Form (BNF) of production rules.

The basic idea of Post was that any mathematical or logic system is simply a set of rules specifying how to change one string of symbols into another set of symbols. That is, given an input string, the antecedent, a production rule could produce a new string, the consequent. This idea is also valid with programs and expert systems where the initial string of symbols is the input data and the output string is some transformation of the input.

As a very simple case, if the input string is "patient has fever" the output string might be "take an aspirin". Note that there is no meaning attached to these strings. That is, the manipulations of the strings is based on syntax and not any semantics or understanding of what a fever, aspirin, and patient represent. A human knows what these strings in terms of the real-world mean but a Post production system is just a way of transforming one string into another. A production rule for this example could be

```
Antecedent → Consequent
person has fever → take aspirin
```

where the arrow indicates the transformation of one string into another. We can interpret this rule in terms of the more familiar IF THEN notation as

```
IF person has fever THEN take aspirin
```

The production rules can also have multiple antecedents. For example,

```
person has fever AND
   fever is greater than 102 → see doctor
```

Note that the special connective AND is not part of the string. The AND indicates that the rule has multiple antecedents.

A Post production system consists of a group of production rules, such as the following (where the numbers in parentheses are for our discussion).

```
(1)   car won't start → check battery
(2)   car won't start → check gas
(3)   check battery AND battery bad → replace battery
(4)   check gas AND no gas → fill gas tank
```

If there is a string "car won't start", the rules (1) and (2) may be used to generate the strings "check battery" and "check gas". However, there is no control mechanism that applies both these rules to the string. Only one rule may be applied, both or none. If there is another string "battery bad" and a string "check battery", then rule (3) may be applied to generate the string "replace battery".

There is no special significance to the order in which rules are written. The rules of our example could also have been written in the following order and it would still be the same system.

```
(4)    check gas AND no gas → fill gas tank
(2)    car won't start → check gas
(1)    car won't start → check battery
(3)    check battery AND battery bad → replace battery
```

Although Post production rules were useful in laying part of the foundation of expert systems, they are not adequate for writing practical programs. The basic limitation of Post production rules for programming is lack of a **control strategy** to guide the application of the rules. A Post system permits the rules to be applied on the strings in any manner because there is no specification given on how the rules should be applied.

As an analogy, suppose you go to the library to find a certain book on expert systems. At the library, you start randomly looking at books on the shelves for the one you want. If the library is fairly large, it will take a very long time to find the book you need. Even if you find the section of books on expert systems, your next random choice could take you to an entirely different section, such as French cooking. The situation becomes even worse if you need material from the first book to help you determine the second book that you need to find. A random search for the second book will also take a long time.

Markov Algorithms

The next advance in applying production rules was made by Markov, who specified a control structure for production systems (Markov 54). A **Markov algorithm** is an ordered group of productions which are applied in order of priority to an input string. If the highest priority rule is not applicable, then the next one is applied and so forth. The Markov algorithm terminates if either (1) the last production is not applicable to a string or (2) a production that ends with a period is applied.

Markov algorithms can also be applied to substrings of a string, starting from the left. For example, the production system consisting of the single rule

```
AB → HIJ
```

when applied to the input string GABKAB produces the new string GHIJKAB. Since the production now applies to the new string, the final result is GHIJKHIJ.

The special character $\wedge$ represents the **null string** of no characters. For example, the production

$$A \rightarrow \wedge$$

deletes all occurrences of the character A in a string.

Other special symbols represent any single character and are indicated by lower-case letters a, b, c, and so forth. These symbols represent single-character variables and are an important part of modern expert system languages. For example, the rule

$$AxB \rightarrow BxA$$

will reverse the characters A and B.

The Greek letters α, β, and so forth are used for special punctuation of strings. The Greek letters are used because they are distinct from the alphabet of ordinary letters.

An example of a Markov algorithm that moves the first letter of an input string to the end is shown following (Elson 73). The rules are ordered in terms of highest priority (1), next highest (2) and so forth. The rules are prioritized in the order that they are entered.

(1) $\alpha xy \rightarrow y\alpha x$
(2) $\alpha \rightarrow \wedge .$
(3) $\wedge \rightarrow \alpha$

For the input string ABC, the execution trace is shown in Table 1-11.

Rule	*Success or Failure*	*String*
1	F	ABC
2	F	ABC
3	S	αABC
1	S	BαAC
1	S	BCαA
1	F	BCαA
2	S	BCA

Table 1-11
Execution Trace of a Markov Algorithm

Notice that the α symbol acts analogously to a temporary variable in a conventional programming language. However, instead of holding a value, the α is used as a place holder to mark the progression of changes in the input string. Once its job is done, the α is eliminated by rule 2. The program then ends when rule 2 is applied since there is a period after rule 2.

The Rete Algorithm

Notice that there is a definite control strategy to Markov algorithms with higher priority rules ordered first. As long as the highest priority rule applies, it is used. If not, the Markov algorithm tries lower priority ones. Although the Markov algorithm can be used as the basis of an expert system, it is very inefficient for systems with many rules. The problem of efficiency becomes of major importance if we want to create expert systems for real problems containing hundreds or thousands of rules. No matter how good everything else is about a system, if a user has to wait a long time for a response, the system will not be used. What we really need is an algorithm that knows about all the rules and can apply any rule without having to try each rule sequentially.

A solution to this problem is the **Rete Algorithm** developed by Charles L. Forgy at Carnegie-Mellon University in 1979 for his Ph.D thesis on the OPS (Official Production System at Carnegie-Mellon) expert system shell. The Rete Algorithm is a very fast pattern-matcher that obtains its speed by storing information about the rules in a network. Instead of having to match facts against every rule on every recognize-act cycle, the Rete algorithm only looks for changes in matches on every cycle.This greatly speeds up the matching of facts to antecedents since the static data that doesn't change from cycle to cycle can be ignored. This topic will be discussed further in the c' ·s on CLIPS. Fast pattern matching algorithms such as the Rete completed the fι ation for the practical application of expert systems. Figure 1-7 summarizes the foundations of modern rule-based expert system technologies.

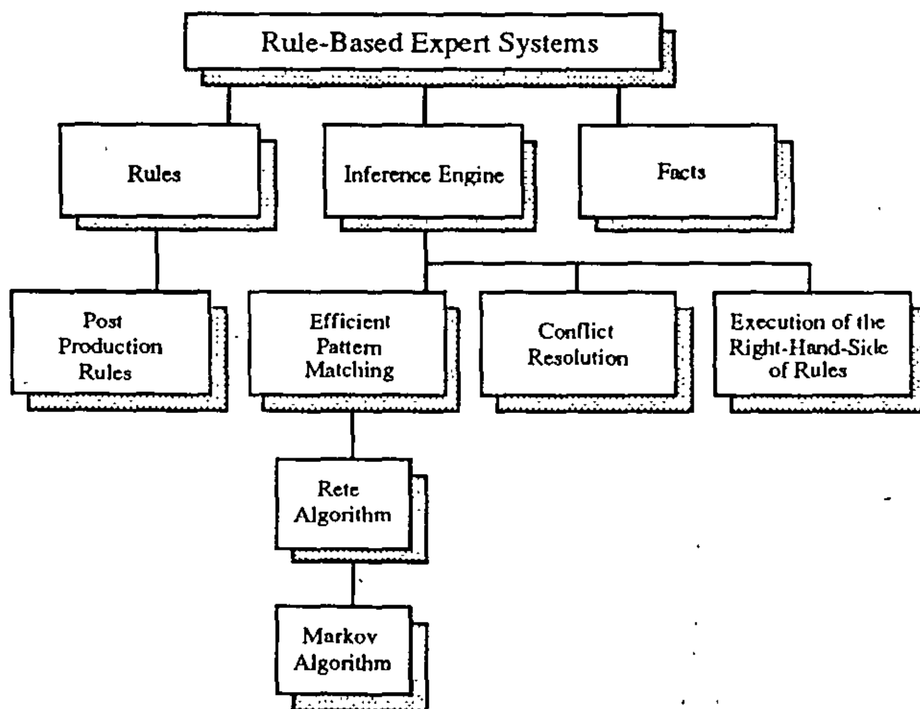


Figure 1-7
Foundations of Modern Rule-based Expert Systems

Different versions of the OPS language and shell have been developed as OPS2, OPS4, and OPS5. A newer commercial version developed by Forgy is OPS83, a very fast shell. However, OPS83 has a significantly different syntax from the OPS5 style and is partly procedural for faster execution.

1.11 PROCEDURAL PARADIGMS

Programming paradigms can be classified as procedural and nonprocedural. Figure 1-8 shows a **taxonomy** or classification of the procedural paradigms in terms of languages. Figure 1-9 shows a taxonomy for nonprocedural paradigms. These figures illustrate the relationship of expert systems to other paradigms. These figures should be considered only as a general guide and not as strict definitions. Some of the paradigms and languages have characteristics that may place them in more than one class. For example, some people consider functional programming a procedural paradigm while others refer to it as declarative (Ghezzi 87a).

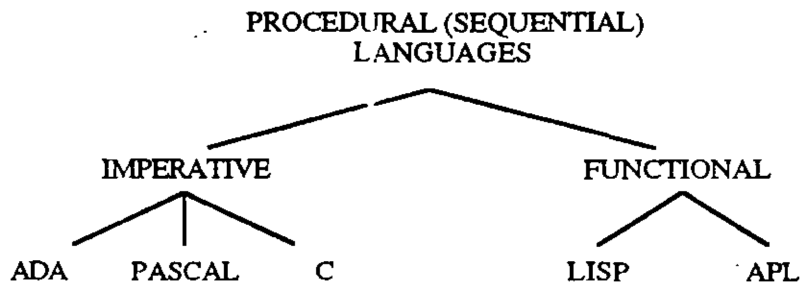


Figure 1-8
Procedural Languages

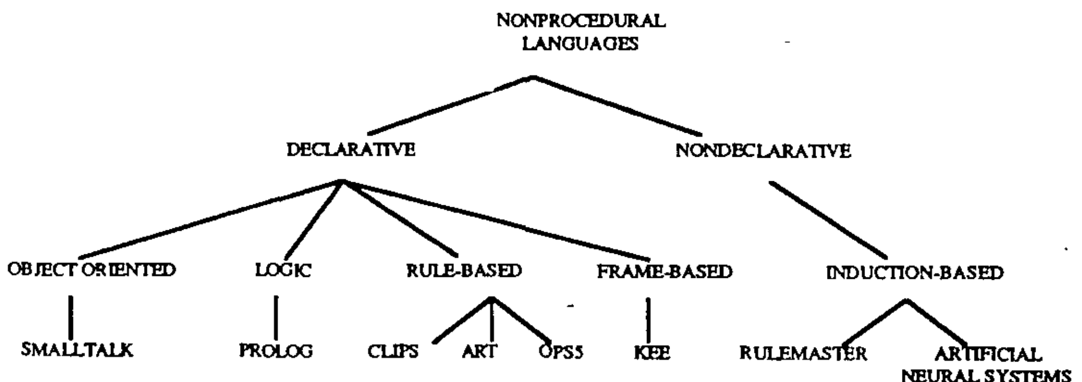


Figure 1-9
Nonprocedural Languages

An **algorithm** is a method of solving a problem in a finite number of steps. The implementation of an algorithm in a program is a **procedural program**. The terms **algorithmic programming,** *procedural programming*, and *conventional programming* are often used synonymously to mean non-AI type programs. A common conception of a procedural program is that it proceeds sequentially, statement by statement, unless a branch instruction is encountered. Another often used synonym for procedural program is **sequential** program. However, the term sequential programming implies too much constraint since all modern programming languages support recursion and so programs may not be strictly sequential.

The distinguishing feature of the procedural paradigm is that the programmer must specify exactly *how* a problem solution must be coded. Even code generators must produce procedural code. In a sense, the use of code generators is **nonprocedural** programming because it removes most or all of the procedural code writing from the programmer. The goal of non-procedural programming is to have the programmer specify *what* the goal is and let the system determine how to accomplish it.

Imperative Programming

The terms **imperative** and **statement-oriented** are used synonymously. Languages ·h as FORTRAN, Ada, Pascal, Modula-2, COBOL, and BASIC all have the domi- ıt characteristic that statements are imperatives or commands to the computer telling it what to do. Imperative languages developed as a way of freeing the programmer from coding assembly language in the von Neumann architecture. Consequently, imperative languages offer great support to variables, assignment operations, and repetition. These are all low-level operations that modern languages attempt to hide by providing features such as recursion, procedures, modules, packages and so forth. Imperative languages are also characterized by their emphasis on rigid control structure and their associated **top-down** program designs.

A serious problem with all languages is the difficulty of proving the correctness of programs. From the AI standpoint, another serious problem is that imperative languages are not very efficient symbol manipulators. Because the imperative language architecture was molded to fit the von Neumann computer architecture, we have languages that can support number-crunching very well but not symbolic manipulation. However, imperative languages such as C and Ada have been used as the underlying base language to write expert system shells. These languages and the shells built from them run more efficiently and quickly on common general purpose computers than the early shells built using LISP.

Because of their sequential nature, imperative languages are not very efficient for directly implementing expert systems, especially rule-based ones. As an illustration of this problem, consider the problem of encoding the information of a real-world problem ;ith hundreds or thousands of rules. For example, the XCON system used by Digital ;uipment Corporation (DEC) to configure computer systems currently has about 7000 rules in its knowledge base. Early unsuccessful attempts were made to code this program in FORTRAN and BASIC before settling on the successful expert systems

approach. The direct way of coding this knowledge in an imperative language w l require 7000 IF THEN statements or a very, very long CASE. This style of coding would present major efficiency problems since all 7000 rules need to be searched for matching patterns on every recognize-act cycle. Note that the inference engine and its recognize-act cycle would also have to be coded in the imperative language.

The efficiency of the program could be improved if rules were ordered so that those most likely to be executed were put at the beginning. However, this would require considerable tuning of the system and would change as new rules are added, or old ones deleted and modified. A better method for improving efficiency would be to build a tree of the rule patterns to reduce search time in determining which rules should be activated. Rather than making the programmer manually construct the tree, it should preferably be built automatically by the computer based on the pattern and action syntax of the IF THEN rules. It would also be helpful to have an IF THEN syntax that was more conducive to representing knowledge and had powerful pattern matching tests. This requires the development of a parser to analyze input structure and an interpreter or compiler to execute the new IF THEN syntax.

When all of these techniques for improving efficiency are implemented, the result is a dedicated expert system. If the inference engine, parser and interpreter are removed to provide easy development of other expert systems, they comprise an expert system shell. Of course, instead of doing all of this development from scratch, today it is much easier just to use an existing shell which is documented and extensively tested.

Functional Programming

The nature of **functional programming**, as exemplified by languages such as LISP and APL, is very different from statement-oriented languages with their heavy reliance on elaborate control structures and top-down design. The fundamental idea of functional programming is to combine simple functions to yield more powerful functions. This is essentially a **bottom-up** design in contrast to the common top-down designs of imperative languages.

Functional programming is centered around **functions**. Mathematically, a function is an **association** or rule that maps members of one set, the **domain**, into another set, the **codomain**. An example of a function definition is

```
cube(x) ≡ x*x*x, where x is a real number and
cube is a function with real values
```

The three parts of the function definition are:

(1) the association, $x*x*x$
(2) the domain, real numbers
(3) codomain, real numbers

of the cube function. The symbol ≡ means "is equivalent to" or "is defined as". The following notation is a shorthand way of writing that the cube mapping is from the domain of real numbers, symbolized as $\Re$, to the codomain of real numbers.

$$\text{cube}: \Re \rightarrow \Re$$

A general notation for a function *f* that maps from a domain *S* to a domain *T* is $f{:}S \rightarrow T$ (Gersting 82). The **range** of the function *f* is the set of all **images** *f(s)* where *s* is an element of *S*. For the case of the cube function, the images of *s* are *s*s*s* and the range is the set of all real numbers. The range and codomain are the same for the cube function. However, this may not be true for other functions such as the square function, x*x, with domain and codomain of real numbers. Since the range of the square function is only non-negative real numbers, the range and codomain are not the same.

Using set notation, the range of a function can be written as

$$R \equiv \{f(s) \mid s \in S\}$$

The **curly braces** { } denote a **set**. The **bar**, |, is read as "where". The above state-nt can be read that the range *R* is equivalent to the set of values *f(s)* where every ment *s* is in the set *S*. The association is a set of ordered pairs *(s,t)*, where $s \in S$, $t \in T$, and $t=f(s)$. Every member of *S* must have one and only one element of *T* associated with it. However, multiple *t* values may be associated with a single *s*. As a simple example, every positive number *n* has two square roots, $\pm\sqrt{n}$.

Functions may also be defined recursively as in.

```
factorial(n) ≡ n*factorial(n-1)
  where n is an integer and
  factorial is an integer function
```

Recursive functions are commonly used in functional languages such as LISP.

Mathematical concepts and expressions are **referentially transparent** because the meaning of the whole is completely determined from its parts. No synergism is involved between the parts. As an example, consider the functional expression $x+(2*x)$. The result is obviously $3*x$. Both $x+(2*x)$ and $3*x$ give the same results no matter what values are substituted for *x*. Even other functions can be substituted for x and the result is the same. For example, let *h(y)* be some arbitrary function. Then $h(y) + (2 * h(y))$ would still be equivalent to $3*h(y)$.

Now consider the following assignment statement in an imperative computer language such as Pascal.

```
sum := f(x) + x
```

If the parameter *x* is passed by reference and its value is changed in the function c *f(x)*, what value will be used for *x*? Depending on how the compiler is written, the value of *x* might be the original value if it was saved on a stack, or the new value if *x* was not saved. Another source of confusion occurs if the one compiler evaluates expressions right-to-left while another evaluates left-to-right. In this case, *f(x)+x* would not evaluate the same as *x+f(x)* on different compilers even if the same language was used. Other side-effects may occur due to global variables. Thus, unlike mathematical functions, program functions are not referentially transparent.

Functional programming languages were created to be referentially transparent. Five parts make up a functional language:

- **data objects** for the language functions to operate on
- **primitive functions** to operate on the data objects
- **functional forms** to synthesize new functions from other functions
- **application operations** on functions that returns a value and
- **naming procedures** to identify new functions.

Functional languages are generally implemented as interpreters for ease of construction and immediate user response.

In LISP (LISt Processing), data objects are **symbolic expressions** (S-expressions) that are either **lists** or **atoms**, while in **APL** (A Programm Language) the objects are arrays. Examples of lists are

```
(milk eggs cheese)
(shopping (groceries (milk eggs cheese) clothes
(pants)))
()
```

Lists are always enclosed in matching parentheses with spaces separating the elements. The elements of lists can be atoms, such as milk, eggs, and cheese, or embedded lists such as (milk eggs cheese) and (pants). Lists can be split up but atoms cannot. The **empty list**, (), contains no elements and is called **nil**.

There must be some primitive functions provided by the language as a basis for building more complex functions. In the original version of LISP, created by John McCarthy in 1960, there were few primitives, as shown in Table 1-12. Primitives CAR, CDR, CPR, and CTR are acronyms named after the specific hardware registers in the first machine to run LISP. CPR and CTR are now obsolete but the acronyms of the others have remained. Also shown are the **predicates** of LISP. Predicates are special functions that return values representing true and false.

The original version of LISP was called pure LISP because it was purely functional. However, it was also not very efficient for writing programs. Non-functional additions have been made to LISP to increase the efficiency of writing programs. For example, SET acts as the assignment operator, while LET and PROG can be used

create local variables and execute a sequence of S-expressions. Although these act like functions, they are not functional in the original mathematical sense.

Function	*Predicates*
QUOTE	ATOM
CAR	EQ
CDR	NULL
CPR	
CTR	
CONS	
EVAL	
COND	
LAMBDA	
DEFINE	
LABEL	

Table 1-12
Original LISP Primitives and Functions

Since its creation, LISP has been the leading AI language in the United States. ..ny of the original expert system shells were written in LISP because it is so easy to experiment with LISP. However, conventional computers do not execute LISP very efficiently and execute the shells built using LISP even worse. In order to circumvent this problem, several companies offer machines specifically designed to execute LISP code. These LISP machines use it completely, even as their assembly language. However, the LISP machines cost considerably more than conventional machines and are for single users.

This problem of high cost has an impact on both the development and the **delivery problem**. It is not enough just to develop a great program if it cannot be delivered for use because of high cost. A good development workstation is not necessarily a good delivery vehicle due to speed, power, size, weight, environmental or cost constraints. Some applications may even require that the final code be placed in ROM for reasons of cost and nonvolatility. Putting code into ROM can be a problem with some AI and expert systems tools that require special hardware to run. It's better to consider this possibility in advance rather than have to recode a program later.

An additional problem is that of embedding AI with conventional programming languages such as C, Ada, Pascal and FORTRAN. For example, certain applications which require extensive number-crunching are best done in conventional languages rather than in LISP, or expert systems languages written in LISP or PROLOG.

Unless special provisions are made, expert systems which are written in LISP are ..nerally difficult to embed in anything other than LISP programs. One major consid-..tion in selecting an AI language should be the language in which the tool is written. For reasons of portability, efficiency, and speed, many expert systems tools are now

being written in or converted to C. This also eliminates the problem of requi expensive special hardware for LISP-based applications.

1.12 NONPROCEDURAL PARADIGMS

Nonprocedural paradigms do not depend on the programmer giving exact details for how a problem is to be solved. This is the opposite of the procedural paradigms which specify *how* a function or statement sequence computes. In nonprocedural paradigms, the emphasis is on specifying *what* is to be accomplished and letting the system determine how to accomplish it.

Declarative Programming

The **declarative paradigm** separates the **goal** from the methods used to achieve the goal. The user specifies the goal while the underlying mechanism of the implementation tries to satisfy the goal. A number of paradigms and associated programming languages have been created to implement the declarative model.

Object-oriented Programming

The **object-oriented** paradigm is another case of a paradigm which can be consid partly imperative and partly declarative. The term object-oriented today is used in two different ways. The expression **object-oriented design** is growing in popularity as a programming methodology in imperative programming languages such as Ada and Modula-2. The basic idea is to design a program by considering the data used in the program as objects and then implementing operations on those objects. This is the opposite of top-down design which proceeds by stepwise refinement of a program's control structure. In fact, object-oriented design is essentially what used to be called bottom-up design. Unfortunately, the term bottom-up never sounded quite as impressive as object-oriented.

As an example of object-oriented design, consider the task of writing a program to manage a charge account with an interactive menu (Riley 87). The important data objects are current balance, amount of charge, and amount of payment. Various operations can be defined to act on the data objects. These operations would be add charge, make payment, and add monthly interest. Once all data objects, operations and the menu interface are defined, coding can begin. This object-oriented design methodology is well-suited to a program that has a weak control structure. It would not be as suitable in a program that requires a strong control structure such as a payroll application. For the payroll case there is a definite sequence of steps to follow.

1. Obtain time-card data
2. Account for sick-leave and vacation time
3. Multiply hours worked X rate of pay

4. Multiply overtime hours X overtime rate
5. Add bonus or sales commissions, if applicable
6. Subtract deductions for taxes, insurance, dues and retirement
7. Issue paycheck

Object-oriented design requires no special language features. It can be done in FORTRAN, BASIC, C and so on. Languages such as Ada and Modula-2 support object-oriented design because these languages have features to encapsulate data in modules and packages.

The term **object-oriented programming** was originally used for languages such as Smalltalk, which was specifically designed for objects. The term is now often used to mean programming of an object-oriented design even in a language which has no true object support.

Smalltalk has features to support objects built into the language. In fact, Smalltalk has a programming environment entirely built using objects. Smalltalk is descended from SIMULA 67, a language developed for simulation. SIMULA 67 introduced the concept of **class** which led to the concept of information hiding in modules. A class is essentially a type which is a template that defines the structure of data. In fact, the concept of type in Pascal and Modula-2 came from class. An **instance** of a class is a data object that can be manipulated. The term instance has carried over to expert systems where it denotes a fact that matches a pattern. Likewise, a rule is said to be instantiated if its LHS is satisfied. The terms activated and instantiated in rule-based systems are synonymous.

Another significant concept that came from SIMULA 67 is **inheritance**. In SIMULA 67, a **subclass** could be defined to inherit the properties of classes. For example, one class may be defined to consist of objects that can be used in a stack and another class defined to be complex numbers. Now a subclass can be easily defined as objects that are complex numbers used in a stack. That is, these objects have inherited properties from both classes above them, called **superclasses**. The concept of inheritance can be extended to organize objects in a hierarchy where objects can inherit from their classes, which can inherit from their classes, and so on. Inheritance is very useful since objects can inherit properties from their classes without the programmer having to specify every property. Many expert systems tools today such as ART and KEE allow inheritance because it is a very powerful tool in constructing large groups of facts.

Logic Programming

One of the first AI applications of computers was in proving logic theorems with the Logic Theorist program of Newell and Simon. This program was first reported at the Dartmouth Conference on AI in 1956 and caused a sensation because electronic computers previously had been used only for numeric calculations. Now a computer was actually reasoning the proofs of mathematical theorems, which had been a task that only mathematicians were thought capable of doing.

In the Logic Theorist and its successor, the General Problem Solver (G' Newell and Simon concentrated on trying to implement powerful algorithms that co solve any problem. While the Logic Theorist was meant only for mathematical theorem proving, GPS was designed to solve any kind of logic problem, including games and puzzles such as chess, Tower of Hanoi, Missionaries and Cannibals, and cryptarithmetic. An example of their famous cryptarithmetic (secret arithmetic) puzzle is

```
  DONALD
+ GERALD
  ROBERT
```

and it is known that D=5. The object is to figure out the arithmetic values of the other letters in the range 0-9.

GPS was the first problem solving program to clearly separate the problem solving knowledge from the domain knowledge. This paradigm of explicitly separating the problem-solving knowledge from the domain knowledge is now used as the basis of expert systems. In expert systems today, the inference engine decides what knowledge should be used and how it should be applied.

Efforts continued to improve mechanical theorem proving. By the early 1970's, it had been discovered that computation is a special case of mechanical, logical deduction (Kowalski 1985). When backward chaining was applied to sentences of the f "conclusion if conditions", it was powerful enough for significant theorem proving. The conditions can be thought of as representing patterns to be matched as in production rules discussed earlier. Sentences expressed in this form are called Horn clauses after Alfred Horn, who first investigated them. In 1972, the language PROLOG was created by Kowalski, Colmerauer and Roussel to implement logic programming by backward chaining using Horn clauses.

Backward chaining can be used both to express the knowledge in a declarative representation and also control the reasoning process. Typically, backward chaining proceeds by defining smaller **subgoals** that must be satisfied if the initial goal is to be satisfied. These subgoals are then further broken down into smaller subgoals and so forth.

An example of declarative knowledge is the following classic example.

```
All men are mortal
Socrates is a man
```

can be expressed in the Horn clauses

```
someone is mortal
    if someone is a man
Socrates is a man
    if (in all cases)
```

For the sentence about Socrates, the if condition is true in all cases. In other words, the knowledge about Socrates does not require any pattern to match. Contrast this with the mortal case in which someone must be a man for the pattern of the if condition to be satisfied.

Notice that a Horn clause can be interpreted as a procedure that tells how to satisfy a goal. That is, to determine if someone is mortal, it is necessary to determine if someone is a man. As a slightly more complex example,

```
A car needs gas, oil and inflated tires to run
```

can be expressed in a Horn clause as

```
x is a car and runs
  if x has gas and
  if x has oil and
  if x has inflated tires
```

Notice how the problem of determining if a car will run has been reduced to three simpler subproblems or subgoals. Now suppose there is some additional declarative 'nowledge as follows.

```
The fuel gauge shows not-empty if a car has gas
The dipstick shows not-empty if a car has oil
The air pressure gauge shows at least 20 if a car has
inflated tires
The fuel gauge shows not-empty
The dipstick reads empty
The air pressure gauge shows at least 15
```

These can be translated into the following Horn clauses.

```
x has gas
  if the fuel gauge shows not-empty
x has oil
  if the dipstick shows not-empty
x has inflated tires
  if the air pressure gauge shows at least 20
Fuel gauge is not empty
  if (in all cases)
Dipstick reads empty
Air pressure is at least 15
  if (in all cases)
```

From these clauses, a mechanical theorem prover can prove that the car will run because there is no oil and insufficient air pressure.

One of the advantages of backward chaining systems is that execution can proceed in parallel. That is, if multiple processors were available they could work on satisfying subgoals simultaneously. This parallel operation can greatly speed up the execution of programs and is one of the reasons that the Japanese chose PROLOG as the programming language for their Fifth Generation Computer project (Moto-Oka 1982). This is an ambitious project to develop the next generation of computers for the 1990's with capabilities far surpassing current models. These machines were to be programmed with declarative techniques, but this approach has run into difficulties. The Fifth Generation project was announced by the Japanese in 1981 with a budget of $850 million dollars. The twenty-six initial goals of the project included natural language translation of Japanese to English and vice-versa, problem-solving and inferencing a thousand times faster than current machines, ability to read handwritten text, understand pictures, intelligent access of huge databases, and to converse in natural language.

PROLOG is more than just a language. At a minimum, PROLOG is a shell since it requires

- an interpreter or inference engine
- a database (facts and rules)
- a form of pattern matching called **unification**
- a **backtracking** mechanism to pursue alternate subgoals if a search to s y a goal is unsuccessful

More elaborate versions of PROLOG such as TurboPROLOG from Borland are tools because of all the enhancements provided.

As an example of backward chaining, suppose you can pay for the oil to make the car run if you have cash or a credit card. One subgoal is checked to see if you have cash. If there is no fact that you do have cash, the backtracking mechanism will then explore the other subgoal to see if you have a credit card. If you have a credit card, the goal of paying for oil can be satisfied. Notice that the absence of a fact to prove a goal is just as effective, although perhaps less efficient, than a negative fact such as "Dipstick reads empty." Either negative facts or missing facts can cause a goal to be unsatisfied.

If the backtracking and pattern matching mechanisms are not needed by the problem, then the programmer must work around them or code in a different language. One of the advantages of logic programming is executable specifications. That is, specifying the requirements of a problem by Horn clauses produces an executable program. This is very different from conventional programming in which the requirements document does not look at all like the final executable code. Newer imperative languages such as Ada attempt to circumvent this problem by offering the capability of using Ada itself as the program description language (PDL). The PDL can be used as a sk on on which to build the rest of the program. If the PDL does not work right, then er will the program.

Unlike production rule systems, the order in which subgoals, facts and rules are entered in a PROLOG program has significant effects. Efficiency and therefore speed are affected by the way that PROLOG searches its database. However, there are programs that execute correctly if subgoals, facts, and rules are entered one way but go into an infinite loop or have a run-time error if the order changes (Ghezzi 87b).

Expert Systems

Expert systems can be considered declarative programming because the programmer does not specify how a program is to achieve its goal at the level of an algorithm. For example, in a rule-based expert system, any of the rules may become activated and put on the agenda if its LHS matches the facts. The order that the rules were entered does not affect which rules are activated. Thus, the program statement order does not specify a rigid control flow. Other types of expert systems are based **frames**, discussed in chapter 2, and **inference nets** discussed in chapter 4.

There are a number of differences between expert systems and conventional programs. Table 1-13 lists some typical differences.

Characteristic	*Conventional Program*	*Expert System*
Control by ...	Statement order	Inference engine
Control and data	Implicit integration	Explicit separation
Control Strength	Strong	Weak
Solution by ...	Algorithm	Rules and inference
Solution search	Small or none	Large
Problem solving	Algorithm is correct	Rules
Input	Assumed correct	Incomplete, incorrect
Unexpected input	Difficult to deal with	Very responsive
Output	Always correct	Varies with problem
Explanation	None	Usually
Applications	Numeric, file, and text	Symbolic reasoning
Execution	Generally sequential	Opportunistic rules
Program design	Structured design	Little or no structure
Modifiability	Difficult	Reasonable
Expansion	Done in major jumps	Incremental

Table 1-13
Some Typical Differences between Conventional Programs and Expert Systems

Expert systems are often designed to deal with uncertainty because reasoning is one of the best tools that we have discovered for dealing with uncertainty. The uncertainty may arise in the input data to the expert system and even the knowledge-base itself. At first this may seem surprising to people used to conventional programming. However, much of human knowledge is heuristic, which means that it may only work correctly part of the time. In addition, the input data may be incorrect, incomplete, inconsistent

and have other errors. Algorithmic solutions are not capable of dealing with situations like this because an algorithm guarantees the solution of a problem in a finite series of steps.

Depending on the input data and the knowledge-base, an expert system may come up with the correct answer, a good answer, a bad answer or no answer at all. While this may seem shocking at first, the alternative is no answer all the time. Again, the important thing to keep in mind is that a good expert system will perform no worse than the best problem-solver for problems like this—a human expert—and may do better. If we knew an efficient algorithmic method that was better than an expert system, we would use it. The important thing is to use the best, and perhaps only, tool for the job.

Nondeclarative Programming

Nondeclarative paradigms are becoming popular. These new paradigms are being increasingly used for a wide variety of applications. They can be used in stand-alone applications or in conjunction with other paradigms.

Induction-based Programming

An application of AI that is attracting interest is **induction-based** programming. In this paradigm, the program learns by example. One application to which this paradigm has been applied is database access. Instead of the user having to type in the specific values for one or more fields for a search, it is only necessary to select one or more appropriate example fields with those characteristics. The database program infers the characteristics of the data and searches the database for a match.

Expert system tools such as 1st-Class and KDS offer induction-learning by which they accept examples and case studies and automatically generate rules.

1.13 ARTIFICIAL NEURAL SYSTEMS

A new development in programming paradigms arose in the 1980's called **artificial neural systems** (ANS), based on how the brain processes information. This paradigm is sometimes called **connectionism** because it models solutions to problems by training simulated neurons connected in a network. Many researchers are investigating neural nets. The nets hold great potential as the front-end of expert systems that require massive amounts of input from sensors as well as real-time response.

The Traveling Salesman Problem

ANS has had remarkable success in providing real-time response to complex pattern recognition problems. In one case, a neural net running on an ordinary microcomputer obtained a very good solution to the Traveling Salesman problem in 0.1 seconds

compared to the optimum solution that required an hour of CPU time on a mainframe (Port 86). The Traveling Salesman problem is important because it is the classic problem faced in optimizing the routing of signals in a telecommunications system. Optimizing routing is important in minimizing the travel time, and thus efficiency and speed.

The basic Traveling Salesman problem is to compute the shortest route through a given list of cities. Table 1-14 shows the possible routes for one to four cities. Notice that the number of routes is proportional to the factorial of the number of cities minus one, (N-1)!.

Number of Cities	*Routes*
1	1
2	1–2-1
3	1–2–3-1 1–3–2-1
4	1–2–3–4-1 1–2–4–3-1 1–3–2–4-1 1–3–4–2-1 1–4–2–3-1 1–4–3–2-1

Table 1-14
Traveling Salesman Problem Routes

While there are 9! = 362880 routes for ten cities, there are 29! = 8.8E30 possible routes for thirty cities. The Traveling Salesman problem is a classic example of **combinatorial explosion** because the number of possible routes increases so rapidly that there are no practical solutions for realistic numbers of cities. If it takes 1 hour of a mainframe CPU time to solve for thirty cities, it will take 30 hours for thirty-one cities and 330 hours for thirty-two cities. These are actually very small numbers when compared to the thousands of telecommunications switches and cities that are used in routing of data packets and real items.

A neural net can solve the ten-city case just as fast as the thirty-city case while a conventional computer takes much longer. For the ten-city case, the neural net came up with one of the two best routes and for the thirty-city case, it came up with one of the best 100,000,000 routes. This is more impressive if it is realized that this route is in the top 1E-22 of the best solutions. Although neural nets may not always give the optimum answer, they can provide a best guess in real-time. In many cases, a 99.9999999999999999999 % correct answer in one millisecond is better than a 100 % correct answer in thirty hours.

Elements of an ANS

An ANS is basically an analog computer that uses simple processing elements connected in a highly parallel manner. The processing elements perform very simple Boolean or arithmetic functions on their inputs. The key to the functioning of an ANS is the **weights** associated with each element. It is the weights which represent the information stored in the system.

A typical artificial neuron is shown in Figure 1-10. The neuron may have multiple inputs but only one output. The human brain contains about 10^{10} neurons and one neuron may have thousands of connections to another. The input signals to the neuron are multiplied by the weights and are summed to yield the total neuron input, I. The weights can be represented as a matrix and identified by subscripts.

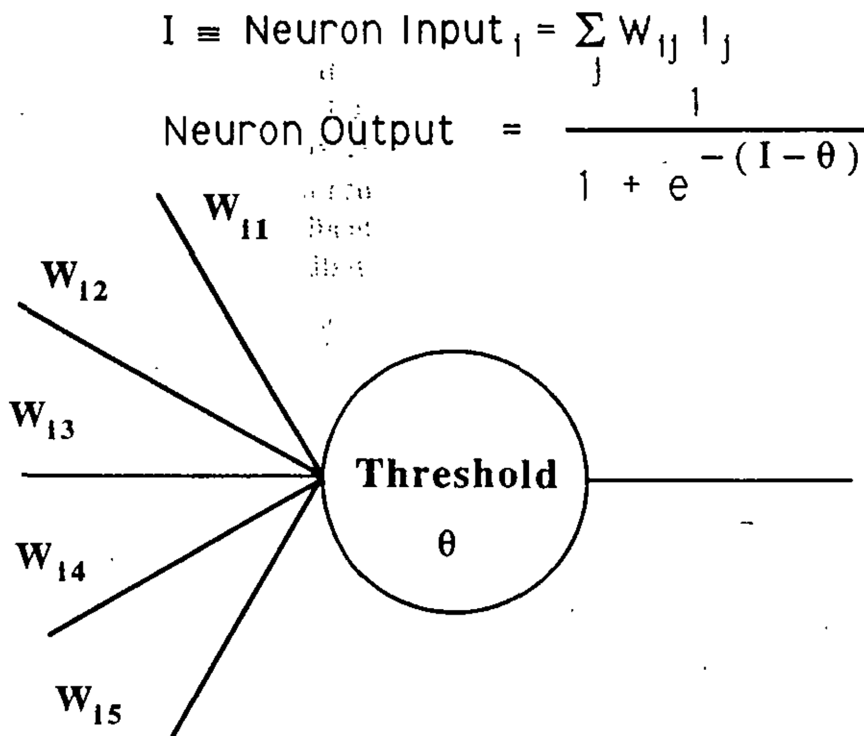


Figure 1-10
A Neuron Processing Element

The neuron output is often taken as a **sigmoid function** of the input. The sigmoid is representative of real neurons which approach limits for very small and very large inputs. The sigmoid is called an **activation function**, and a commonly used function is $(1+e^{-x})^{-1}$. Each neuron also has an associated **threshold value**, θ, which is subtracted from the total input I. Figure 1-11 shows an ANS which can compute the **exclusive-OR (XOR)** of its inputs using a technique called **back-propagatic**

The XOR gives a true output only when its inputs are not all true or not all false. The number of nodes in the hidden layer will vary depending on the application and design.

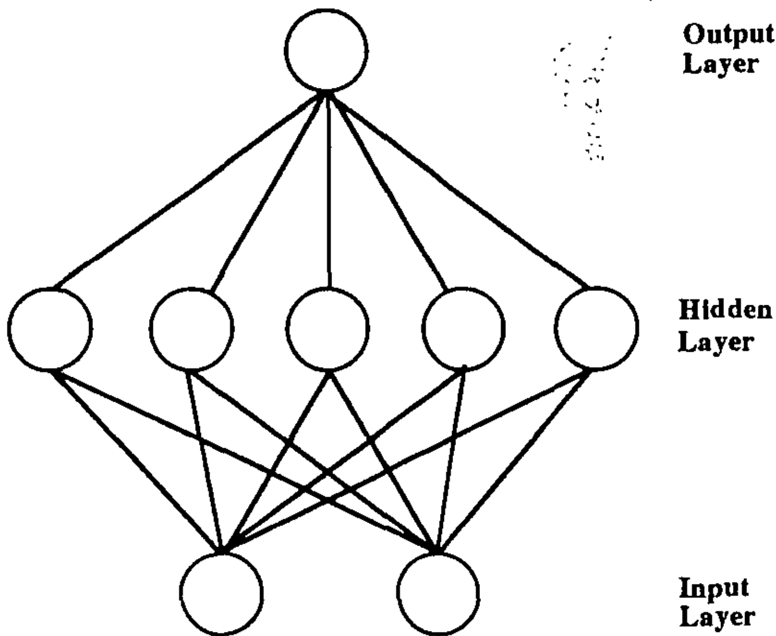


Figure 1-11
A Back-propagation Net

Neural nets are not programmed in the conventional sense. There are about thirteen known neural net learning algorithms, such as **counter-propagation** and back-propagation, to train nets. The programmer "programs" the net by simply supplying the input and corresponding output data. The net learns by automatically adjusting weights in the network which connect the neurons. The weights and the threshold values of neurons determine the propagation of data through the net and so its correct response to the training data. Training the net to the correct responses may take hours or days depending on the number of patterns that the net must learn, the hardware, and software. However, once the learning is accomplished, the net responds very quickly.

If software simulation is not fast enough, ANS can be fabricated in chips for real-time response. Once the network has been trained and the weights determined, a chip can be constructed. AT&T and other companies are fabricating experimental neural net chips containing hundreds of neurons. In the next few years, it is very likely that chips will be fabricated containing thousands of neurons.

Characteristics of ANS

ANS architecture is very different from a conventional computer architecture. In a conventional computer, it is possible to correlate discrete information with memory cells. For example, a Social Security number could be stored as ASCII code in a contiguous

group of memory cells. By examining the contents of this contiguous group the Social Security number could be directly reconstructed. This reconstruction is possible because there is a one-to-one relationship between each character of the Social Security number and the memory cell that contains the ASCII code of that character.

ANS are modeled after current brain theories in which information is represented by the weights. However, there is no direct correlation between a specific weight and a specific item of stored information. This distributed representation of information is similar to that of a hologram in which the lines of the hologram act as a diffraction grating to reconstruct the stored image when laser light is passed through.

A neural net is a good choice when there is much empirical data and no algorithm exists which provides sufficient accuracy and speed. ANS offers several advantages compared to the storage of conventional computers.

- *Storage is very **fault tolerant**.* Portions of the net can be removed and there is only a degradation in quality of the stored data. This occurs because the information is stored in a distributed manner.
- *The quality of the stored image degrades gracefully in proportion to the amount of net removed.* There is no catastrophic loss of information. The storage and quality features are also characteristic of holograms.
- *Data is naturally stored in the form of **associative memory**.* An associative memory is one in which partial data is sufficient to recall all of the complete stored information. This contrasts with conventional memory in which data is recalled by specifying the address of the data to be recalled. A partial or noisy input may still elicit the complete original information.
- *Nets can extrapolate and interpolate from their stored information.* Training teaches a net to look for significant features or relationships in the data. Afterwards, the net can extrapolate to suggest relationships on new data. In one experiment (Hinton 86), a neural net was trained on the family relationships of twenty-four hypothetical people. Afterwards, the net could also answer correctly relationships about which it had not been trained.
- *Nets have **plasticity**.* Even if a number of neurons are removed, the net can be retrained to its original skill level if enough neurons remain. This is also a characteristic of the brain in which portions can be destroyed and the original skill levels can be relearned in time.

These characteristics make ANS very attractive for robot spacecraft, oil field equipment, underwater devices, process control and other applications that need to function a long time in a hostile environment without repair. Besides the issue of reliability, ANS offer the potential of low maintenance cost because of plasticity. Even if hardware repair can be done, it will probably be more cost-effective to reprogram the neural net than replace it.

ANS are generally not well-suited for applications that require number-crunching or an optimum solution. Also, if a practical algorithmic solution exists, an ANS is not a good choice.

Developments in ANS Technology

The origins of ANS started with the mathematical modeling of neurons by McCulloch and Pitts in 1943 (McCulloch 43). An explanation of learning by neurons was given by Hebb in 1949 (Hebb 49). In Hebbian learning, a neuron's efficiency in triggering another neuron increases with **firing.** The term firing means that a neuron emits an electrochemical impulse which can stimulate other neurons connected to it. There is evidence that the conductivity of connections between neurons at their connections, called **synapses**, increases with firing. In ANS, the weight of connections between neurons is changed to simulate the changing conductance of natural neurons.

In 1961, Rosenblatt published an influential book dealing with a new type of artificial neuron system he had been investigating called a **perceptron** (Rosenblatt 61). The perceptron was a remarkable device that showed capabilities for learning and pattern recognition. It basically consisted of two layers of neurons and a simple learning algorithm. The weights had to be manually set in contrast to modern ANS that set the weights themselves based on training. Many researchers entered the field of ANS and began studying perceptrons during the 1960's.

The early perceptron era came to an end in 1969 when Minsky and Papert published a book called *Perceptrons* that showed the theoretical limitations of perceptrons as a general computing machine (Minsky 69). They pointed out a deficiency of the perceptron in being able to compute only 14 of the 16 basic logic functions, which means that a Perceptron is not a general purpose computing device. In particular, they proved a perceptron could not recognize the exclusive-OR. Although they had not seriously investigated multiple layer ANS, they gave the pessimistic view that multiple layers would probably not be able to solve the XOR problem. Government funding of ANS research ceased in favor of the symbolic approach to AI using languages such as LISP and algorithms. New methods of representing symbolic AI information by frames, invented by Minsky, became popular during the 1970's. Further work on perceptrons has continued with new types able to overcome Minsky's objections (Reece 87). Because of their simplicity, perceptrons and other ANS are easy to construct with modern integrated circuit technology.

ANS research continued on a small scale in the 1970's. However, the field finally entered a renaissance starting with the work of Hopfield in 1982 (Hopfield 82). He put ANS on a firm theoretical foundation with the two-layer Hopfield Net and demonstrated how ANS could solve a wide variety of problems. The general structure of a Hopfield Net is shown in Figure 1-12. In particular, he showed how an ANS could solve the Traveling Salesman Problem in constant time as compared to the combinatorial explosion encountered by conventional algorithmic solutions. An electronic circuit form of an ANS could solve the Traveling Salesman Problem in 1 μ second. Other combinatorial optimization problems can easily be done by ANS such as the four-color map, the Euclidean-match (Hopfield 86b), and the transposition code (Tank 85).

An ANS that can easily solve the XOR problem is the **back-propagation** net, also known as the **generalized delta rule** (Rumelhart 86). The back-propagation net is commonly implemented as a three-layer net, although additional layers can be speci-

fied. The layers between the input and output layers are called **hidden layers** beca… only the input and output layers are visible to the external world. An important theoretical result from mathematics, the Kolmogorov Theorem, can be interpreted as proving that a three-layer network with n inputs and 2n + 1 neurons in the hidden layer can map any continuous function (Hecht-Nielsen 86).

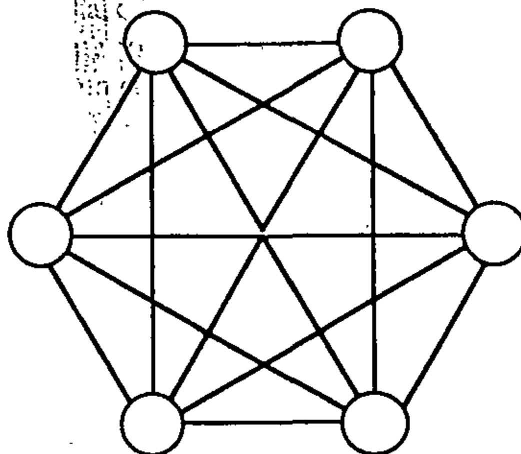


Figure 1-12
A Hopfield Artificial Neural Net

Applications of ANS Technology

A significant example of learning by back-propagation was demonstrated by a neural net that learned correct pronunciation of words from text (Sejnowski 86). The ANS was trained by correcting its output using a Digital Equipment Corp. text to speech device called DECTalk. It required twenty years of linguistic research to devise rules for correct pronunciation used by DECTalk. The ANS taught itself equivalent pronunciation skills overnight by simply listening to the correct pronunciation of speech from text. No linguistic skills were programmed into the ANS.

Investigations of the ANS are under way for recognition of radar targets by electronic and optical computers (Farhat 86). New implementations of neural nets using optical components promise optical computers with speeds millions of times faster than electronic ones. Optical implementation of ANS is attractive because of the inherent parallelism of light. That is, light rays do not interfere with one another as they travel. Huge numbers of photons can easily be generated and manipulated by optical components such as mirrors, lenses, high-speed programmable spatial light modulators, arrays of optical bistable devices that can function as optical neurons, and diffraction gratings. Optical computers designed as ANS appear to be complementary to one another.

Commercial Developments in ANS

A number of new companies and existing firms have been organized to develop ANS technology and products. Nestor markets an ANS product called NestorWriter that can recognize handwritten input and convert it to text using a PC. Other companies such as TRW, SAIC, HNC, Synaptics, Neural Tech, Revelations Research and Texas Instruments market a variety of ANS simulators and hardware accelerator boards to speed up learning. One of the best bargains for getting started in neural computing is volume 3 of Rumelhart's books on Parallel Distributed Processing available from the MIT Press. The book describes a half-dozen ANS simulators and also includes a diskette with software for an IBM PC compatible machine for $27.50. By purchasing a 386 accelerator card for an IBM XT or compatible, a person can have a very powerful system for ANS at low cost.

1.14 CONNECTIONIST EXPERT SYSTEMS AND INDUCTIVE LEARNING

It is possible to build expert systems using ANS. In one system, the ANS is the knowledge-base constructed by training examples from medicine for disease (Gallant 88). In this system, the expert system tries to classify a disease from its symptoms one of the known diseases that the system has been trained on. An inference engine called MACIE (Matrix Controlled Inference Engine) was designed that uses the ANS knowledge-base. The system uses forward chaining to make inferences and backward chaining to query the user for any additional data needed to produce a solution. Although an ANS by itself cannot explain why its weights are set to certain values, MACIE can interpret the ANS and generate IF THEN rules to explain its knowledge.

An ANS expert system such as this uses **inductive learning**. That is, the system **induces** the information in its knowledge-base by example. Induction is the process of inferring the general case from the specific. Besides ANS, there are a number of commercially available expert systems shells that explicitly generate rules from examples. The goal of inductive learning is to reduce or eliminate the knowledge acquisition bottleneck. By placing the burden of knowledge acquisition on the expert system, the development time may be reduced and the reliability may be increased if the system induces rules that were not known by a human.

1.15 SUMMARY

In this chapter we have reviewed the problems and developments which have led to expert systems. These problems that expert systems are used for are generally not solvable by conventional programs because they lack a known or efficient algorithm. Since expert systems are knowledge-based, they can be effectively used for real-world problems that are ill-structured and difficult to solve by other means.

The advantages and disadvantages of expert systems were also discussed in the context of selecting an appropriate problem domain for an expert system application. Criteria for selecting appropriate applications were given.

The essentials of an expert system shell were discussed with reference to rule-based expert systems. The basic recognize-act inference engine cycle was described and illustrated by a simple rule example. Finally, the relationship of expert systems to other programming paradigms was described in terms of the appropriate domain of each paradigm. The important point of all this is the concept that expert systems should be viewed as another programming tool that is suitable for some applications and unsuitable for others. Later chapters will describe the features and suitability of expert systems in much more detail.

PROBLEMS

1-1. Identify a person other than yourself who is considered an expert or very knowledgeable. Interview this expert and discuss how well this person's expertise would be modeled by an expert system in terms of <u>each</u> criterion in the section "Advantages of Expert Systems."

1-2. a) Write ten nontrivial rules expressing the expert of problem 1's knowledge.

b) Write a program that will give your expert's advice. Include test results to show that each of the ten rules gives the correct advice. For ease of programming, you may allow the user to provide input from a menu.

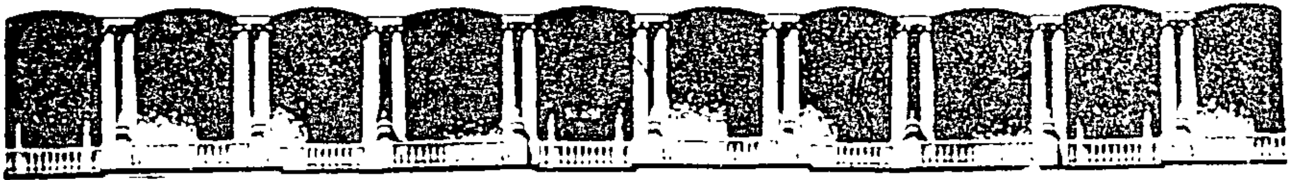
1-3. a) In Newell and Simon's book, *Human Problem Solving*, they mention the 9-Dot Problem. Given 9 dots arranged as follows, how can you draw four lines through all the dots without (a) lifting your pencil from the paper and (b) crossing any dot? (Hint: you can extend the line past the dots)

```
•   •   •
•   •   •
•   •   •
```

b) Explain your reasoning (if any) in finding the solution and whether an expert system or some other type of program would be a good paradigm to solve this type of problem.

1-4. Write a program that can solve cryptarithmetic problems. Show the result for the following problem, where D=5.

```
  DONALD
+ GERALD
  ROBERT
```



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS ABIERTOS

ROBOTS MÓVILES

BRAITENBERG CREATURES

EXPOSITOR : DR. JESÚS SAVAGE CARMONA
1998

Braitenberg Creatures

David W. Hogg, Fred Martin, Mitchel Resnick

MIT Media Laboratory
20 Ames Street
Cambridge, MA 02139
617-253-7143

June 5, 1991

Abstract

This paper describes 12 autonomous "creatures" built with Electronic Bricks. Electronic Bricks are specially-modified LEGO bricks with simple electronic circuits inside. Although each Electronic Brick is quite simple, the bricks can be combined to form robotic creatures with interesting and complex behaviors, similar to the fictional machines described in Valentino Braitenberg's book *Vehicles* (1984).

Introduction

This paper describes a collection of artificial "creatures" made from LEGO bricks. The creatures were inspired by Valentino Braitenberg's book *Vehicles* (1984).

In *Vehicles*, Braitenberg describes a set of thought experiments in which increasingly complex vehicles are built from simple mechanical and electronic components. Each of these imaginary vehicles in some way mimics intelligent behavior, and each one is given a name that corresponds to the behavior it imitates: "Fear," "Love," "Values," "Logic," etc. Braitenberg uses these thought experiments to explore psychological ideas and the nature of intelligence. Progressing through the book, the reader sees very intricate behaviors emerge from the interaction of simple component parts. In a sense, Braitenberg "constructs" intelligent behavior—a process he calls "synthetic psychology."

Our work follows Braitenberg in spirit. However, instead of Braitenberg's imaginary motors and sensors, we have developed a set of real components known as **Electronic Bricks**—LEGO bricks with electronic circuits inside. These bricks can be connected together to form a wide variety of artificial "creatures," much like Braitenberg's vehicles. Of course, our creatures do not reach the levels of complexity that Braitenberg's do. On the other hand, our creatures exist in the real world: people can actually interact with and play with them. Our creatures do not live only in the world of scientific publications.

We designed our creature-construction kit particularly for children. Children can easily connect the bricks together into new configurations, to create new creatures with new behaviors. In doing so, children can explore, in a playful way, the same deep ideas that Braitenberg describes in his book. In trying to construct particular behaviors, children explore the emergence of complex behaviors from simple components. At the same time, children confront fundamental questions about intentionality. Creatures built from Electronic Bricks fall on the fuzzy boundary between animals and machines, forcing students to come to terms with how machines can be like animals, and vice versa (Resnick and Martin 1990).

This creature-construction activity serves as a prototypical example of the "constructionist" approach to learning (Papert 1980). According to the constructionist paradigm, people are most likely to make deep

connections with new ideas when they are involved in constructing meaningful artifacts. In this case, children learn important ideas about living systems not just by observing creatures, but by building them.

Electronic Bricks

Electronic Bricks are LEGO bricks in which we have placed digital electronic circuits with inputs and outputs. They fall into three categories: *action bricks* (such as **motor bricks**), *sensor bricks* (such as **threshold sound sensors**), and *logic bricks* (such as **and bricks**). When connected by a wire, the output of one brick controls the input of another.

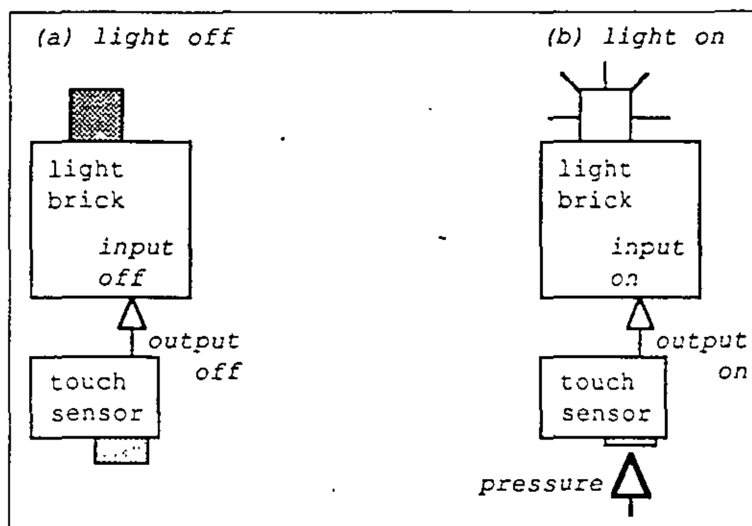


Figure 1: A simple **Electronic Brick** setup.

Imagine that we have a **touch sensor**. Its output is *on* when its button is pressed, and its output is *off* at all other times. If this output is attached to the input of a **light brick**, then the input of the **light brick** will be on when the button is pressed (see figure 1(b)), and off otherwise (see figure 1(a)). This extremely simple setup provides us with a light that turns on when you press the **touch sensor** and turns off again when you let go.

In this paper, we use certain terms that have specific meanings in the **Electronic Brick** universe. These terms will be printed throughout the article in boldface, and they are all defined in the glossary at the end. Included in the glossary are descriptions of each individual type of **Electronic Brick**.

Simple creatures

The first six creatures, *Timid*, *Indecisive*, *Paranoid*, *Dogged*, *Insecure*, and *Driven*, are all quite simple. They generally have only one sensor and rather limited electronic "brains."

Timid

the shadow seeker

The simplest combination of **Electronic Bricks** possible is one motor attached to one sensor. Happily, even the simplest combinations can produce interesting behaviors.

Timid is a tiny car with one **motor brick** and one **threshold light sensor**, pointing up. The motor brick has two inputs: one controls its power, one controls its direction. The output of the sensor is attached to the *power input* of the **motor brick**. When the light falling on the sensor is in excess of its *threshold*, its output turns on. Because this output is plugged into the power input of the motor, the motor turns on. Hence if sufficient light is falling on the **threshold light sensor**, the car drives forward; otherwise it stands still. The threshold is set with a dial on the side of the sensor.

If the dial on the **threshold light sensor** is set correctly, *Timid* will run when it can "see" the room lights, and stop when it cannot. When the lights are turned on, *Timid* drives until it gets into shadow, at which point it stops. If whatever is casting the shadow is moved, *Timid* will start driving again until it enters another shadow.

Indecisive

the shadow edge finder

Indecisive is identical to *Timid*, except the output of the **threshold light sensor** goes to the *direction input* of the **motor brick** instead of the power input. In this case, the motor is on all of the time, but it goes in one direction when the input is on and in the other when the input is off. This means that *Indecisive* runs forward when it sees the room lights, and backwards when it does not.

When let loose, *Indecisive* drives forward (assuming that the room lights are on) until it gets to a shadow cast by a chair, table, someone's hand, etc. At this point, the **threshold light sensor** no longer sees the overhead lights, and its output switches off. The motor reverses, and the creature runs back into the light. Now the light sensor is illuminated again and *Indecisive* reverts to forward motion, returning to the shadow. In the shadow, the sensor output turns off again, and the vehicle backs up again. It oscillates back and forth at shadow edges.

Paranoid

the shadow-fearing robot

The structure of this creature is what we call a **turtle** Turtles have two motors, one that drives the wheels on the left side (the *left motor*), and one that drives the wheels on the right side (the *right motor*). If the two motors are doing different things, then the **turtle** will turn. That is, if the left motor is driving forward, and the right motor is off, the **turtle** will move forward and to the right. If the left motor is driving forward, and the right motor is driving backward, the **turtle** will pivot in place to the right. The turtle will move in a straight line only if both motors are on and turning in the same direction.

This particular **turtle** has one sensor, a **threshold light sensor**, whose threshold is set so that its output is on in the light and off in the shadow, just as it is for *Timid* and *Indecisive*. The **threshold light sensor** points up, and it is on an arm that sticks out forward on the vehicle. Its output is connected to the direction input of the left motor. When its output is on, the left and right motors turn in the same direction (the direction of the right motor defaults to forward, the direction of the left motor is set to forward by the output of the **threshold light sensor**), and when the output is off, they turn in opposite directions.

When *Paranoid* is set down in a lit room, it drives straight forward until its protruding **threshold light sensor** enters a shadow. When this happens, the sensor's output switches from on to off and the left wheel reverses. At this point, the left and right wheels are turning in opposite directions. This forces the **turtle** to pivot to the left (see figure 2). It swings around to the left until the protruding sensor has swung back out of the shadow. At this point the left wheel returns to forward motion, and *Paranoid* is off again.

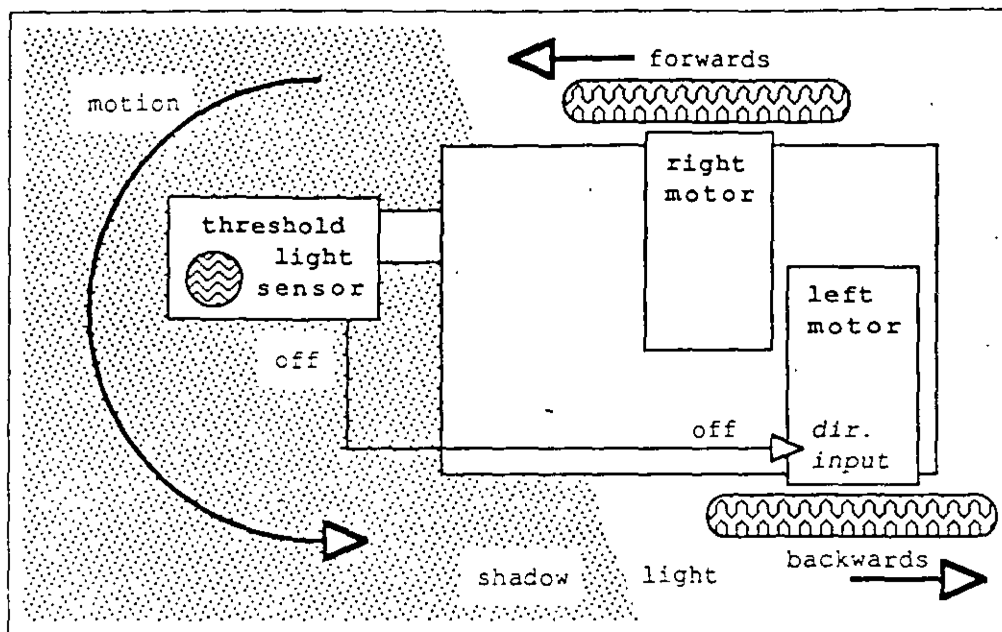


Figure 2: *Paranoid* turning.

Dogged

the obstacle avoider

This vehicle is a small car with a single **motor brick** that drives it forward or backward. It has two **touch sensors**, one facing front and one back, each connected to a bumper. When the bumper is pressed, the touch sensor is activated, and its output turns on. It also has an **or brick** and a **flip-flop brick**. The **flip-flop brick** has two states, an *on state* and an *off state*. In the on state, its ouptut is on continuously, and in the off state, the output is off continuously. It changes state each time its input is turned on. So if the input is off and then turns on, the brick will change state, and it won't change state again until the input turns off and on again. The outputs of the **touch sensors** go to the inputs of the **or brick** and its output goes to the **flip-flop brick**. The **flip-flop brick**'s output is connected to the direction input of the motor.

When *Dogged* is started, it runs either forward or backward, depending on the state of the **flip-flop brick**. When either the front or back bumper is pressed (and hence a **touch sensor**'s output turns on), the **or brick**'s output turns on, which changes the state of the **flip-flop brick**, and the creature reverses direction. In other words, *Dogged* changes direction every time either bumper gets pressed. Let loose, the vehicle will run across the floor until it hits something. When it does, the bumper will be pressed and it will reverse direction and run away. It will continue until it hits something in the other direction. It reverses and runs back to the first obstacle. In this way, it will fall into a pattern of running very quickly back and forth between two objects over and over again.

Insecure

the wall follower

This vehicle is a **turtle**, like *Paranoid*, but it uses the **whisker brick** to sense its surroundings. The **whisker brick** is a brick with a thin plastic strip (*whisker*) pointing out of it. When the strip becomes sufficiently bent, the output of the sensor turns on. This "bending threshold" is set with a dial on the side of the brick. The whisker protrudes over *Insecure*'s left side. The output of the **whisker brick** goes to the power input of the left motor and to an **inverter brick**. An **inverter brick**'s output is on if its input is off, and is off if its input is on. The output of the **inverter brick** is attached to the power input of the right motor. This setup ensures that exactly one motor is on at any time, since the output of the sensor controls one motor and the inverse of that output controls the other.

When *Insecure* is put down in an open area, the **whisker brick**'s output will be off, the right motor will be driving, and it will turn in circles. Imagine, however, that it is put down so that there is a wall close to it on its left. The right motor will drive, turning the vehicle left and forwards, until the whisker touches the wall. Once this happens, the output of the **whisker brick** will turn on, and the left motor will drive; the inverter will turn off and the right motor will stop. *Insecure* will now move forwards and right. Soon it will have turned far enough right that the whisker will no longer be touching the wall, and it will start turning back.

Insecure slowly edges its way along walls and around the bases of pillars.

Driven

the light seeker

Driven is our standard light seeking vehicle. It is a **turtle**. It has a **differential light sensor** mounted on it, facing forward. The **differential light sensor** is a brick with two outputs. When the left side of the sensor is receiving more light than the right side, the *left output* will be on. When the right side is receiving more light than the left side, the *right output* will be on. The left output of the **differential light sensor** is attached to the power input of the right motor, and the right output to the power input of the left motor. Thus, the left motor turns on when the right side of the sensor is brighter, and the right motor turns on when the left side is brighter.

Driven moves towards a bright light by successive right and left turns. If the light is on the left, the left output of the **differential light sensor** will be on and so the right motor will be running, moving the creature forward and rotating it left. Once the vehicle has turned far enough that the light is on its right, it starts moving forward and right. *Driven* slowly wiggles its way towards light sources.

More complex creatures

In this section we present 4 creatures, all of which are a bit more sophisticated than the previous ones.

Persistent

the light seeker with a collision algorithm

This vehicle is the same as *Driven*, but with an attempt to give it a method for avoiding obstacles.

In addition to *Driven*'s **differential light sensor**, *Persistent* has a bumper on its front which is attached to a **touch sensor**. This sensor plugs into a **timer brick**. The **timer brick** has one input and one output.

Each time the input turns on, the output turns on for a period of time and then turns off again. The period is set with a dial on the side of the brick. The output of the **timer brick** is attached to the direction inputs on the motors.

When the creature is let loose, it starts moving towards the light source in the same way that *Driven* does. However, if it collides with something on its way, the bumper will get pressed. When this happens, the **touch sensor** will be activated, the **timer brick** will turn on and the motor directions will be reversed. They will stay reversed for the duration of the **timer brick**'s period.

Upon collision, *Persistent* backs up for a short period of time and then the **timer brick** switches off. The creature resumes travelling towards the light. However, *Persistent* fails in its goal of avoiding obstacles, since the creature generally resumes travelling along the same path it was on when it made the contact. It usually collides with the same object that it did before, and it does this over and over again.

Attractive and Repulsive

The leading and following pair

This example actually involves two Braitenberg creatures.

Attractive is a creature with one **motor brick** and a **threshold light sensor** pointing towards the rear. The **threshold light sensor**'s output goes to the power input of the motor. It drives forward when the sensor's output is on. *Attractive* is a very quick vehicle.

Repulsive is a slow creature with one **motor brick** and a bank of bright lights. It continually drives forward. When *Repulsive*'s lights get sufficiently close to *Attractive*'s **threshold light sensor**, the output of the sensor will turn on, and *Attractive* will move.

Imagine that *Attractive* and *Repulsive* are set down in a line, with *Repulsive* facing *Attractive*'s back. *Repulsive* will drive slowly forward until its lights come within the range of *Attractive*'s sensor. The sensor's output will turn on and *Attractive* will run quickly away from *Repulsive* until it is out of range and so the motor stops again. Soon, however, *Repulsive* will have come within range again.

Consistent

the four-state turtle

This vehicle is a **turtle** with one **threshold sound sensor**. When the **threshold sound sensor**'s tiny microphone "hears" a sufficiently loud noise, its output turns on for a very short time. Its threshold is set with a dial on the side of the brick. The sensor's output goes to a **flip-flop brick**. The output of that **flip-flop brick** goes to another **flip-flop brick**, as well as the direction input of the left motor. The second **flip-flop brick**'s output goes to the direction input of the right motor.

Every time the **threshold sound sensor** is triggered, the output of the first **flip-flop brick** changes state (either from on to off or off to on). So, every *second* time the sensor is triggered, the output of the first **flip-flop brick** turns from off to on (as opposed to changing from on to off). Whenever the output of the first **flip-flop brick** changes from off to on, the output of the second **flip-flop brick** changes state. In other words, the output of the first **flip-flop brick** changes state every time that the sensor is triggered, and the output of the second **flip-flop brick** changes every second time. This means that *Consistent* has four different "mental" states: on-on, off-on, on-off, off-off.

Because the output of the first **flip-flop brick** is attached to the direction input of the right motor, and the output of the second is attached to the direction input of the left motor, these "mental" states each correspond to one of four states of motion. The first corresponds to forward motion, the second to turning left, the third to turning right, the fourth to moving backwards. If one claps loud enough in the vicinity of

Consistent, it will change state. In fact, as one claps repeatedly, *Consistent* will cycle through its four states in this order.

Inhumane

the mousetrap

Inhumane is designed to capture mice. It consists of a long, thin tunnel with a door at one end. The door can be moved by turning on a **motor brick**. Near the other end of the tunnel, there is a light source on one side and a **threshold light** sensor on the other. Some mouse food is placed at the very end, past the light beam and sensor. If a mouse enters the chamber, he or she must break the light beam to get to the bait. The light is always on, and the sensor's threshold is set so that the output is on when the light's path is unobstructed, and off when it is blocked.

The layout of **Electronic Bricks** that make up *Inhumane*'s "brain" is shown in figure 3.

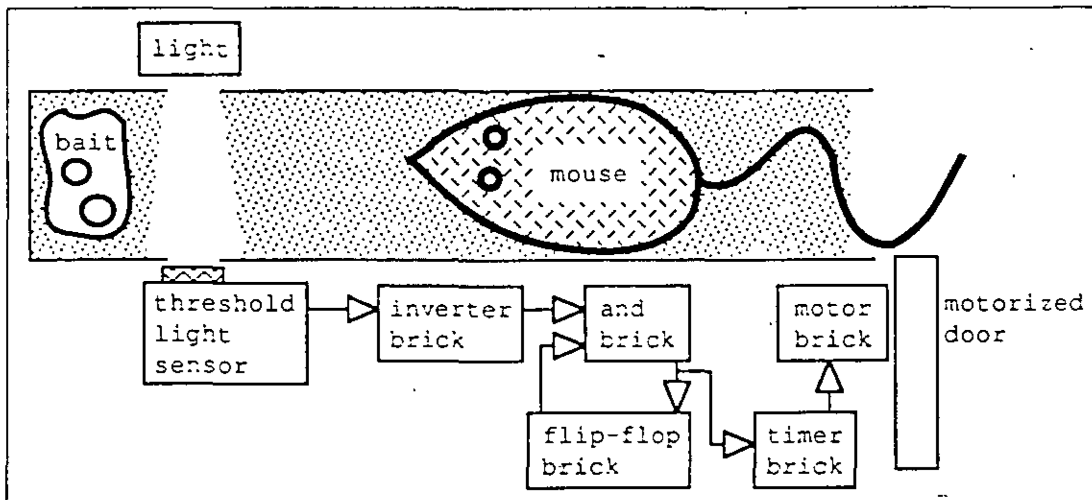


Figure 3: The layout of *Inhumane*.

Imagine that we start with the trap's door open and the **flip-flop brick**'s output on. In this situation, one of the **and brick**'s inputs is on. If the beam is broken by a mouse, the **threshold light sensor**'s output changes to off, the **inverter brick**'s to on, and then both of the **and brick** inputs are on. The **and brick** turns on the inputs of the **flip-flop brick** and the **timer brick**. The **timer brick**'s output turns on for just long enough to close the door. The **flip-flop brick** changes state and therefore one of the inputs to the **and brick** is now off.

Now the mouse is incarcerated, and the circuit is impotent. The mouse can run back and forth past the beam all he or she likes, but with one input to the **and brick** off (the one from the **flip-flop brick**), the **timer brick** will never turn on again. In other words, when the **flip-flop brick** is in its off state, the **threshold light sensor** has no effect on it, whereas when it was in its on state, the sensor could indeed affect it. This circuit is directly analogous to a physical trap, because most traps begin in a loaded state (**flip-flop brick** on), get triggered (light beam is broken), and then close, entrapping the victim (**flip-flop brick** off). After the trap is sprung, no amount of fiddling with the trigger will reopen it (re-breaking the beam does not change the **flip-flop brick**'s state).

In practice, *inhumane* caught several mice in our building. Most were able to chew their way out, leaving

a pile of LEGO dust behind. However, we did indeed relocate one mouse to a more natural habitat. Soon, however, our creation was outmoded by more efficient methods of pest control.

Philosophical creatures

These two machines value thought over action. The first, *Frantic*, does nothing but think: it only observes itself. The second, *Observant*, observes its environment, but does not change it.

Frantic

the negative feedback loop

Frantic consists of a **light brick**, an **inverter brick** and a **threshold light sensor**. A **light brick** is an **Electronic Brick** with one input and a small light. When the input is on, the light is on; when the input is off, the light is off.

The **threshold light sensor** points at the **light brick**, and its output goes to the **inverter brick**. The **inverter brick**'s output goes to the **light brick**. Now there is a "paradox": if the light is on, the sensor's output is on, the **inverter brick**'s output is off, and the light turns off. If the light is off, then the sensor's output turns off, the **inverter brick**'s output turns on, and the light turns on.

As one might predict, the light blinks on and off at a frenzied rate.

Observant

the creature sensitive to the direction of a sound

By comparing the outputs of two **threshold sound sensors**, *Observant* can find the direction from which a sound comes. The circuit does not lend itself to verbal description, so we present a diagram: figure 4.

In air, sound travels a foot in about one millisecond. With the **threshold sound sensors** approximately one foot apart, *Observant* can indeed tell the direction from which the sound came since the sound takes about a millisecond to get to the more distant sensor. That is, if the sound comes from the left, and hits the left sensor first, the output of the left **timer brick** will turn on, and if it comes from the right, the output of the right **timer brick** will turn on. The circuit restores itself when the timing cycles of the **timer bricks** run out.

It is interesting to note the conceptual similarity between the electronics in *Inhumane* and *Observant*. Both creatures have circuits that start in one state, and in the presence of some stimulus "drop" into some new state which the initial triggering stimulus is unable to change. Both circuits are analogous to traps. But because *Observant* uses **timer bricks** instead of **flip-flop bricks**, it restores automatically to its initial state after a specific time.

Acknowledgments

The creatures described in this paper were designed by Ryan Evans, David Hogg, Fred Martin, Seymour Papert, Mitch Resnick and Brian Silverman.

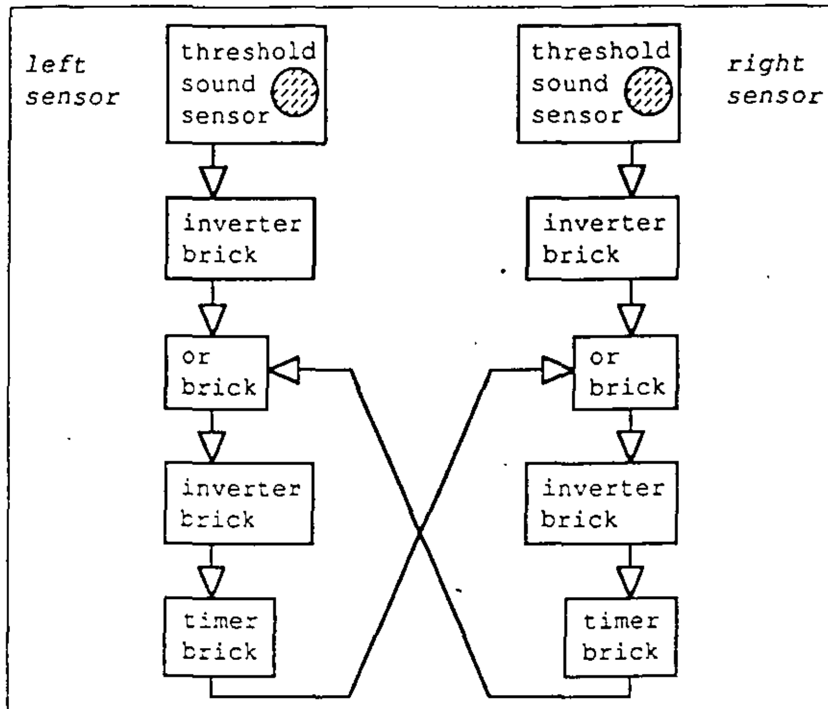


Figure 4: *Observant's* "brain."

References

[1] Braitenberg, Valentino (1984) *Vehicles: Experiments in Synthetic Psychology.* The MIT Press, Cambridge, Massachusetts.

[2] Papert, Seymour (1980) *Mindstorms: Children, Computers, and Powerful Ideas* Basic Books, New York.

[3] Resnick, Mitchel, and Martin, Fred (1990) "Children and Artificial Life," E&L Memo No. 10, MIT Media Laboratory, Cambridge, Massachusetts.

Glossary

And brick: An **Electronic Brick** with two inputs and one output. If both inputs are on, the output is on; otherwise it is off.

Electronic Bricks: These are the subject of this paper. They are LEGO bricks with simple electronic circuits inside, and *inputs* and *outputs* that allow you to connect them together. The *inputs* and *outputs* are digital (only on or off). There are *action bricks* (such as **motor bricks**), *sensor bricks* (such as **threshold sound sensors**), and *logic bricks* (such as **and bricks**). When connected by a wire, the *output* of one brick controls the *input* of another.

Differential light sensor: An **Electronic Brick** with two light sensors and two outputs. When the sensor on the left side of the brick is more brightly lit than the right, then its *left output* is on, if the right is brighter, then its *right output* is on. If the light levels on the two sides are extremely close, then both outputs are on.

Flip-flop brick: An **Electronic Brick** with one input and one output. This brick has two states: an ***on state*** and an ***off state.*** When the input to this brick turns on, it changes state. That is, if it is in the ***off state*** and the input changes from off to on, it changes to the ***on state***, and it will stay that way until the input turns off and then on again. When this happens, it will change back to the ***off state.*** When in the ***off state,*** its output is off, and in the ***on state***, its output is on. The **flip-flop brick** is like a toggle switch.

Motor brick: An **Electronic Brick** with two inputs, and a motor shaft which can be attached to wheels or other LEGO machinery. If the ***direction input*** is on, the shaft turns in one direction. If it is off, it turns in the other. If the ***power input*** is on, it runs. If it is off, it does not. ***Note:*** If there is nothing attached to the power input, the motor defaults to on.

Inverter brick: An **Electronic Brick** with one input and one output. When the input is on, the output is off. When the input is off, the output is on.

Light brick: An **Electronic Brick** with a tiny lightbulb. It has one input. When the input is on, the light is on; when the input is off, the light is off.

Or brick: An **Electronic Brick** with two inputs and one output. If both inputs are off, the output is off: otherwise it is on.

Threshold light sensor: An **Electronic Brick** with a light sensor and one output. When the light falling on the sensor is above a certain level, the output is on; otherwise it is off. The threshold is set with a dial on the side of the brick.

Threshold sound sensor: An **Electronic Brick** with a tiny microphone and one output. When the sound picked up by the microphone is sufficiently loud, the output turns on for a short time. The threshold is set with a dial on the side of the brick.

Timer brick: An Electronic Brick with one input and one output. Each time the input turns on, the output turns on for a period of time. The period is set with a dial on the side of the brick.

Touch sensor: An **Electronic Brick** with a button on its side and one output. When the button is depressed. the output is on: otherwise it is off.

Turtle: A type of vehicle with wheels on the left side and wheels on the right side. **Turtles** have two **motor bricks**. one that drives the wheels on the left side (the *left motor*), and one that drives the wheels on the right side (the *right motor*). If the two motors are doing different things, then the **turtle** will turn. For example, if the left motor is driving forward. and the right motor is off, the **turtle** will move forward and to the right. If the left motor is driving forward. and the right motor is driving backward, the **turtle** will pivot to the right in place. It will only move in a straight line if both motors are on and turning in the same direction.

Whisker brick: An **Electronic Brick** with one output and a thin plastic strip sticking out one side. When the strip gets bent far enough, the output turns on; otherwise it is off. The bending threshold is set with a dial on the side of the brick.

World modeling and path planning for autonomous mobile robots

Glen E. Monaghan

Department of Electrical Engineering, US Air Force Academy
DFEE, USAF Academy, Colorado Springs, Colorado, 80840

Abstract

An autonomous mobile robot that is capable of planning "intelligent" paths through its world must have a suitable representation, or model, of that world and must be able to interpret its sensory inputs in terms of that world model. Furthermore, the robot must be able to modify its planned movements (and world model) based on differences between current perceptions of the world and the existing model.

This paper describes a two-part world model and a method for path planning that are particularly suitable for use in known, man-made environments. The basic model treats the robot's world as a hierarchical network of logical regions with arbitrary polygonal boundaries. Because the regions need not be convex, they may correspond one-to-one with intuitively partitioned, but irregularly shaped, real-world regions such as rooms, hallways, and open work areas. Obstructions within regions are also modeled by arbitrary polygons. A "complex configuration space" is derived from this basic model and is used to plan goal-oriented paths in a top-down fashion. Dynamic replanning is used to cope with changes in the environment.

Introduction

The techniques described below were developed during preliminary efforts to design and build an autonomous mobile robot for aircraft service and maintenance. The initial work entered on general navigation which, for an autonomous mobile robot, consists of cartography, path planning, and collision-free motion. A robot navigator is a task- or function-oriented path-finding system that is capable of developing and maintaining an accurate model of the world, accepting motion-related commands, and planning and replanning paths to guide the robot mechanism through its world to accomplish the given tasks.

Emphasis was placed on finding good paths through known, man-made environments such as storage rooms, hallways, hangars, and flight lines. Planning a path that is as direct as possible, while circumnavigating known obstacles and avoiding inadvertent entrapment, is known as the find-path problem.

The find-path problem is well known in robotics, with solutions falling into two broad categories. One approach models, and plans paths around, any obstacles in the world, while the other basic approach models, and plans paths through, the open spaces between obstacles. In both cases, the three-dimensional real world normally is reduced into a two-dimensional floor plan representation. Many implementations of the first approach are based on the classic configuration space method.

Classic configuration space models

The classic configuration space method considers only a single reference point of the moving object (robot), such as its center, and determines the boundaries of the space which the reference point can occupy near obstacles without causing any other part of the moving object to collide with an obstacle[1]. This locus of "free space" is known as the configuration space. Essentially, this classic configuration ("free") space is the complement of the union of all obstacles, although it also depends heavily on the shape and orientation of the moving object. Paths which restrict the reference point to the current configuration space are guaranteed to be collision free for the moving object in the real world.

Three problems are usually associated with configuration spaces. First, the cross sections of free-standing obstacles must be convex polygons. Since many typical real-world areas and objects are not convex, they must be "built up" from unions of convex polygons, which increases both model size and processing times. Second, the paths found by this method often pass near obstacles, so positioning errors in a dead-reckoning robot mechanism could lead to collisions. Finally, rotation of the moving object leads to inefficiencies, unless the rotation can be ignored (usually possible because the moving object is approximately circular and its reference point is the object's center). When rotation can not be ignored, either the configuration space must be recalculated after every rotation or else

the space must be sufficiently conservative to allow for any rotation, which "wastes" f space near objects and may prohibit otherwise good paths between closely spaced obstacles

Direct free space models

The rationale behind the other general approach to the find-path problem is that, since the path must remain in unobstructed space, the unobstructed space should be modeled directly rather than as the complement of the union of all obstacles. Two methods found in the literature for directly modeling free space include use of overlapped "generalized cones" to represent the two-dimensional spaces (sweepable areas) between obstacles in the world model[2], and use of non-overlapped convex polygons to partially tile the same areas[3]. The cones (or convex polygons) represent available pathways, so the find-path problem reduces to finding a match between the areas (cones or polygons) of free space and the area swept by the object as it moves toward the goal. Advantages claimed for this approach are that paths tend to pass midway between obstacles rather than on near-tangents, and that free space is modeled directly. The generalized cone approach also has the ability to deal with rotation of arbitrarily-shaped objects without recomputing the free space. (The convex polygon method simply assumes rotation can be ignored.)

Although both of the cited free-space methods reduce the likelihood of collision (compared with classic configuration space methods), they also increase path length, especially for widely spaced obstacles. Also, although the free space is modeled directly, the number of cones or convex polygons required to represent it increases rapidly with the number of obstacles present, and the cones/polygons bear little relationship with intuitive partitions of the real world. Finally, if the robot mechanism is approximately circular (its width and length are approximately equal), path planning using a configuration space method can be computationally more efficient than the cone method.

Other methods

A number of other efforts to solve the find-path problem have been reported in the literature, but were found to be of little interest here for various reasons. Most were either concerned only with paths through completely unknown worlds, or else were dependent on complete vision systems. Both of these cases are needlessly inefficient in the inter application for this project.

The solution to the find-path problem that is described in this paper was developed for a robot, approximately circular, that inhabits a man-made environment containing many objects and distinctly separate areas. The world is modeled using a "complex configuration space" to represent free space in a straightforward way, with a minimum number of elements in the model. Each element corresponds with a floor plan view of an intuitively-defined real-world area, such as a single room or work area.

Approach

The general approach taken in this paper to solve the find-path problem is as follows. The world inhabited by the robot mechanism is described (modeled) as a **network of regions**. Each region has its own natural orientation and a bounded floor plan. Regional bounds, much like the walls of a room or the painted lines around a work area in a factory, serve to define meaningful areas, each area containing a group of (zero or more) spatially-related objects. A ring of distance-measuring sonars spaced at a fixed height around the robot's body serve as the main sensors. Because of the wide beam and fixed elevation of each sonar sensor, the robot does not have a video-quality three dimensional image of its world. Rather, it "sees" the world as a low resolution, two dimensional plot of the distances to surrounding obstacles. Therefore, a region's floor plan is described in terms of boundaries of the area and objects, as viewed at the sensor height. **Passageways**, areas of overlap between regions, are the only places where the robot mechanism may transition across (non-physical) boundaries between regions.

Given a command to move to a new position within the current region, the navigation system finds a path, if one exists, that avoids intermediate obstacles and stays within the region's bounds. For efficiency, commands to move to positions outside the current region involve **hierarchical path planning**. First, a limited-detail **global path** is planned across the network of regions to find a sequence of passageways leading from the current region to the goal region. Then, local paths are planned between successive passageways to traverse each intermediate region. The end points for each straight-line leg of the local path passed to the robot mechanism-driver, which is expected to use feedback from the vari sensor subsystems to keep it on course, within some tolerance, while avoiding unforseen obstacles. The sensor subsystems are also expected to report back to the navigation system on previously unknown impasses so that the world model may be updated[4].

The world model

e robot's world is divided into logical regions with arbitrary polygonal boundaries. ause regions do not need to be convex, they may correspond one-to-one with intuitively rtitioned, but irregularly shaped, regions of the real world, such as rooms, walkways, parking areas, and so forth. All objects in the robot's world, whether they are physical or non-physical, are also modeled by arbitrary polygons. The polygons are restricted only in that edges of a given polygon may not cross each other, like a figure-8, although they may touch. This restriction is quite reasonable since it in no way limits the actual shapes that may be represented, only the way they are represented. The restriction simply reflects the fact that the sides of real-world objects may narrow to negligible thickness and even merge, but they can never cross through each other.

escribing a polygonal area.

One of the simplest and most compact ways to represent a polygon is with an ordered list f its vertices. Successive pairs of vertices define successive edges of the polygon and, y imposing a consistent order on the vertices, they also define the "inside" and "outside" f the edges, with respect to the whole polygon. These definitions are simple applications f basic geometry. In particular, a straight line segment can be drawn joining any two oints, any such segment can be extended in a straight line, and a straight line divides the lane containing it.

The general equation of a straight line, given in parametric form, is:

$$X = X_0 + mt \qquad Y = Y_0 + nt \tag{1}$$

here t is the parameter and (X_0, Y_0) is the point on the line corresponding to a zero value f t. By adopting the convention that t ranges from 0 at one end of an edge (call it K) to at the other end (call it L), Eq (1) can be rewritten as:

$$X = X_K + (X_L - X_K)t \qquad Y = Y_K + (Y_L - Y_K)t \tag{2}$$

rametric form can be converted to the implicit form to yield a single equation for ine containing the two points, K and L:

$$(Y_K - Y_L)X + (X_L - X_K)Y + (X_K Y_L - X_L Y_K) = 0 \tag{3}$$

en, points on one side of the edge-extended line satisfy the inequality:

$$(Y_K - Y_L)X + (X_L - X_K)Y + (X_K Y_L - X_L Y_K) < 0 \tag{4}$$

ile points on the other side satisfy:

$$(Y_K - Y_L)X + (X_L - X_K)Y + (X_K Y_L - X_L Y_K) > 0 \tag{5}$$

In brief, Eq (3) describes a line containing two points, K and L. Points on one side or e other of the line will satisfy either Eq (4) or Eq (5). Points that lie on the line ntaining K and L will satisfy Eq (3) and, furthermore, points that actually lie on the ge, between K and L, will simultaneously satisfy both parts of Eq (2) with a single value parameter t, between 0 and 1, inclusive. The convention used in this paper is to list e vertices of an edge such that the side facing the free space satisfies Eq (5), while the de toward "forbidden" space satisfies Eq (4). (See Figure 1.) Depending on the ordering the vertices, polygonal objects may enclose either free or forbidden space.

basic world model

Given this background, it is easy to see that a polygonal model of a free-standing tacle has its vertices specified with a clockwise order. Such polygons completely lose forbidden space and are surrounded by free space, so they exclude the robot from the as that they occupy.

Conversely, because the exterior border of a region encloses free space, the vertices of exterior border are listed in counterclockwise order. This permits closed figures that d ribed by counterclockwise ordering of vertices to define large areas of free space, ich smaller areas of forbidden space (obstacles) are subtracted out with clockwise g of vertices. Each region, then, has exactly one exterior (counterclockwise) border y have zero or more interior (clockwise) borders that represent forbidden areas and tacles.

The edges of the polygonal figures that comprise the basic world model may be eithe sensor detectable or non-sensor detectable. Detectable edges, termed physical, represen solid objects that are detectable by the robot's sonars. Since not all logically separate areas in the real world are even partially bounded by physical objects such as walls, and because passageways must be unobstructed for the robot to pass through, the non-sensor detectable, perimeter, edge is introduced. Although not detectable by the robot's sonars, perimeter edges are used three ways.

First, perimeter edges may form a closed polygon with vertices ordered clockwise, to form a virtual obstacle (one not detectable by the main sensors) that excludes the robot from an area. Second, they may form a closed polygon with counterclockwise order, to define the exterior border of a logical region that lacks any sensor-detectable boundaries. Finally, an open series of perimeter edges may be used to link physical edges to complete a closed polygon, with both the perimeter and physical vertices having the same order. A very common usage of such a perimeter edge is to complete an otherwise solid (sensor detectable) exterior border of a region, thereby forming a passageway to another region (that is, the perimeter represents the threshold of a doorway).

Each region, then, is described by its exterior border, its interior borders (if any), and any passageways to adjacent regions (Figure 2). The arrangement of regions in the robot's world is described by the network of regions. This network specifies the connectivity of the regions in terms of the passageways that directly connect them.

To summarize the basic world model, the real world is divided into logical areas of open space, called regions. Each region has one exterior border and zero or more interior borders due to obstacles in the area. All areas are modeled by arbitrary polygons with edges that represent combinations of sensor detectable and non-detectable real world objects. Regions overlap slightly at passageways, which are defined to be perimeter edges that are the only places where the robot can safely exit the current region. The connectivity of the regions is specified by the network of regions, in terms of the passageways between adjacent regions.

The complex configuration space

To reduce the computational requirements of the find-path solution, the basic wor. model is transformed into the complex configuration space, a proper subset of the basic world model in which all paths are guaranteed to be collision-free. The complex configuration space is similar to the convex polygons used by Crowley[3] to directly model free space, except that it need not have a convex exterior border and it may include interior borders. (It is because of these two features that this configuration space is termed "complex".)

In general, a configuration space consists of that free space which may be occupied by a reference point on the robot without producing a collision between the robot and the various objects in the region[5]. Since the cross section of the robot for which this system was designed is a regular dodecagon, the robot is sufficiently close to being circular that it can be modeled by a circle with the reference point at the center. This simplification permits rotation of the robot to be ignored, so only a single complex configuration space need be calculated for each region.

The derivation of the complex configuration space is a version of the "growing" process described by Lozano-Perez[1]. Each edge in a region is displaced by a fixed distance into the free space and reconnected to form grown structures. Because the regions are not required to be convex and because regions may contain free-standing obstacles, the grown structures must be "pruned" to prevent cross-over of the structures, caused by edges in the basic world model that face each other with a separation less than the growing distance (Figure 3). By displacing edges more than the (greatest) radius of the robot mechanism, and then pruning overlaps, collision-free motion is guaranteed for the actual robot when its reference point stays on or within the grown structures.

Corners in a complex configuration space

Traditional find-path solutions that use configuration space are based on decomposing the world model into sets of convex polygons that represent obstacles. All vertices in the space are then considered to be possible waypoints of a path[1]. The large number of convex polygons needed for typical world models leads to a combinatorial explosion of possib' paths when applied to regions of reasonable complexity. Use of a complex configurati space reduces the search space of possible paths by reducing the possible waypoints to those vertices that "divide" free space.

In a complex configuration space (one polygonal exterior border and zero or more non-overlapping, polygonal interior borders), the junction of any two adjacent edges that meet

at an angle greater than 180 degrees (measured through free space) is said to divide free space, and is an inside corner. The simple region shown in Figure 4 has one interior angle that is greater than 180 degrees, forming an inside corner. The figure is not convex, but it can be formed from the three convex polygons A, B, and C. The shortest paths between points in A and points in C consist of two legs, one from the point in A to the inside corner, V, and the other from V to the point in C. No other waypoints should be considered for minimal length paths. This same concept applies to regions with arbitrarily complex boundaries. It is never necessary to consider a vertex as a waypoint on a minimal indirect path unless that vertex is an inside corner (Figure 5).

To further simplify matters, interior nodes are derived from the set of inside corners. When two inside corners occur "negligibly close" to each other and each divides the same localized free space, one of the corners or an entirely new median point may be chosen to replace the original pair (Figure 6). This replacement is called an interior node. After merging all possible inside corners, the new nodes and any remaining corners are all considered to be interior nodes and are combined to form the network of visible nodes.

The network of visible nodes describes all possible direct paths between interior nodes within the region. Because every point in a complex configuration space is visible from at least one interior node, the network of visible nodes can be used to find a minimal indirect path between any two obstructed points within the space. Such a path consists of one leg to an interior node, zero or more legs to other interior nodes, and one leg from the last interior node to the goal.

In summary, all edges of the basic world model are shifted by a safe amount into the free space to form the complex configuration space. The space is complex in that it has one polygonal exterior border and may have non-overlapping polygonal interior borders. Some points within the space, the interior nodes, serve to divide the free space, and the mutually-visible interior nodes are described by the network of visible nodes. This network is used to find minimal indirect paths through the region when direct paths are not possible. Figure 7 shows the complex configuration space derived from Figure 2.

Using the world model to find paths

The solution to the find-path problem is a hierarchical one. Intra-regional path finding ("local find-path") is developed first and then extended to include inter-regional path finding ("global find-path").

The local find-path algorithm

If the goal is within the same region as the robot, then only the free space associated with that region need be considered to find a local path; the rest of the world outside the region can be ignored. Because "local" path finding refers to the situation where the goal and the robot's current position are both in the same complex configuration space, find-path must verify that the goal position for each new path actually lies inside the complex configuration space.

If the current and goal positions are both within the same configuration space, the local path is direct if no edges of the complex configuration space intervene between the current position and the goal. If there are no intervening edges, the local path is called "direct" and is simply the straight line from the current position to the goal. However, direct paths are not guaranteed because configuration spaces are, in general, complex polygons.

When intervening edges of the complex configuration space prevent a direct path, a search must be made of the space for an indirect path that is minimal by some metric. The metric used here is Euclidean distance. As is described in the derivation of the complex configuration space, a shortest indirect path through the space proceeds from the start to the goal point, passing through only inside corners for waypoints. When interior nodes are used instead of inside corners, paths are not guaranteed to be absolutely minimal length for the complex configuration space. However, a near-minimal path will be found, possibly in considerably less time because of the smaller number of nodes to search. Therefore, the local find-path algorithm finds all the interior nodes with an unobstructed view of the starting point and searches from them through the region's network of visible nodes for an interior node with an unobstructed view of the goal. The shortest path through the network to the goal is the indirect path selected by the local find-path routine.

The procedure just described for finding indirect paths is an application of classical state-space search, which is thoroughly discussed in the literature. The state-space graph for local find-path is the network of visible nodes, plus the current and goal locations, within a region's complex configuration space. The search procedure used in the local find-path solution is the A* search algorithm, which has been called "the optimal search for an optimal solution"[6]. Euclidean distance is the cost metric used. The A* algorithm is

applied in the local find-path solution only when an obstruction exists between the start and goal locations.

The global find-path algorithm

If the goal is not within the current region, then reaching the goal is a three-step process. The first step is to find a global path from the starting region to the goal region. This is facilitated by the network of regions which describes passageways between adjacent regions. The second step is to iteratively find and follow local paths to each successive passageway along the global path until the region containing the goal is reached. The final step is to find and follow a local path from the last passageway to the goal itself.

The network of regions is a high-level "map" used to find a reasonably good over-all path from the start to the goal location. Searching this network is analogous to the search procedure used in the local find-path algorithm, except it uses the network of regions rather than the network of visible nodes. Because of this close similarity, global find-path uses the same basic A* search algorithm used by local find-path.

Building on the find-path solutions

A number of mid-level, task-oriented commands build upon the find-path solutions and low-level mechanism control instructions. The first task-oriented command is the basic "goto XY" function which uses local find-path to move about within a region. A similar command is "goto Region". This function combines local and global find-path to move the robot into the desired region. The last of the basic motion commands is "goto XY in Region". As one might expect, this function simply commands a "goto Region" followed by "goto XY" to reach a specific place in the distant region.

A data base of known "things" provides memory for the location of various parts and supplies. The task "bring the Widget" is accomplished by getting the robot's current region and XY location, recalling the last-known region and XY location of the Widget, performing a "goto XY in Region" to find the Widget, performing a "goto XY in Region" to return to the original location and, finally, updating the data base to reflect the movement of the Widget.

In each case, paths are planned before motion begins, so there is no guarantee that paths can be completed as planned. For example, a recently moved obstacle may block a passageway. In such cases, the robot will try to circumnavigate the blockage and stay on the original path. However, if the path deviation becomes excessive, the old plan is scrapped and a new path is planned from the current position to the goal.

Conclusion

This paper describes a world model and find-path solution that are suitable for use in man-made environments. For efficiency, both the model and find-path solution are organized hierarchically. The two-part model also supports higher level, task-oriented commands that build on the basic find-path solution.

References

1. Lozano-Perez, T. and M.A. Wesley. "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," Communications of the ACM, 22 (10): 560-570 (October 1979).

2. Brooks, R.A. "Solving the Find-Path Problem by Good Representation of Free Space," Proceedings of the National Conference on Artificial Intelligence: 381-386. The American Association for Artificial Intelligence. (August 1982).

3. Crowley, J.L. "Position Estimation for an Intelligent Mobile Robot," 1983 Annual Research Review. Pittsburgh, PA: Robotics Institute, Carnegie-mellon University, 1984.

4. Monaghan, G.E. Navigation for an Autonomous Mobile Robot. MS thesis. Wright-Patterson AFB, OH: School of Engineering, Air Force Institute of Technology, December 1984. (AFIT/GE/ENG/84D-47).

5. Brady, M., et al. Robot Motion: Planning and Control. Cambridge: The MIT Press 1982.

6. Gardner, A. "Heuristic State-space Search" in The Handbook of Artificial Intelligence, Volume I, edited by A. Barr and E. A. Feigenbaum. Los Altos, CA: William Kaufmann, Inc., 1981.

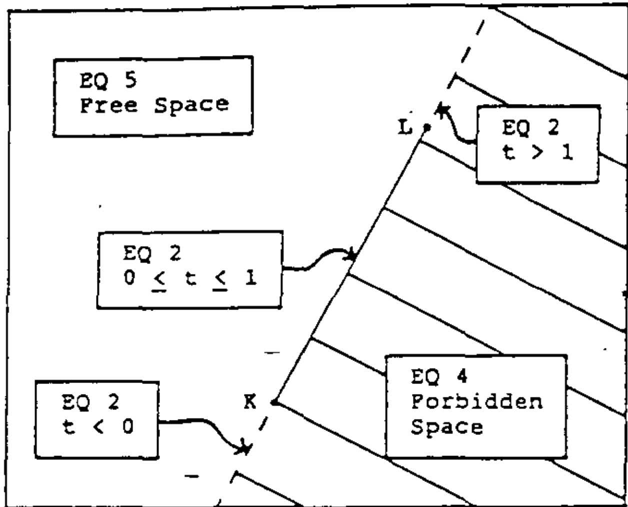


Figure 1. The interpretation of line segment KL and its related equations.

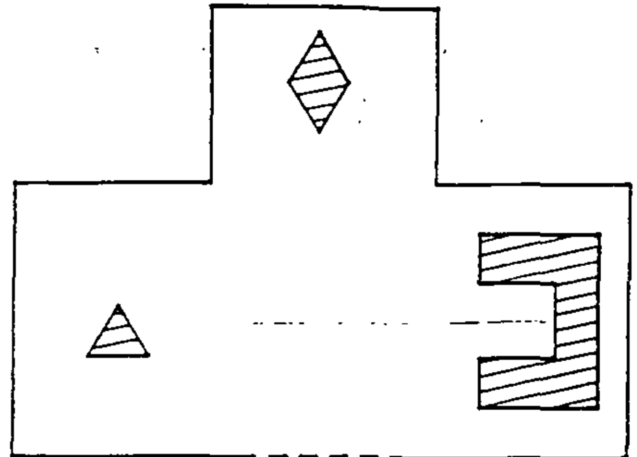


Figure 2. A sample region showing an exterior border with one passageway and three enclosed objects.

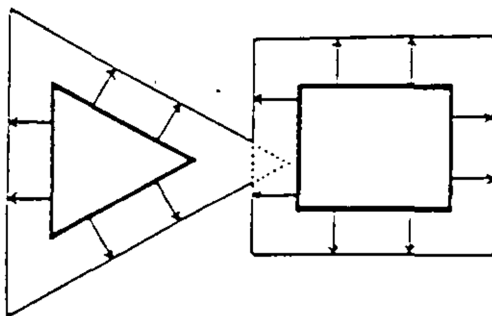


Figure 3. Grown structures which overlap must be pruned, possibly merging several objects into one.

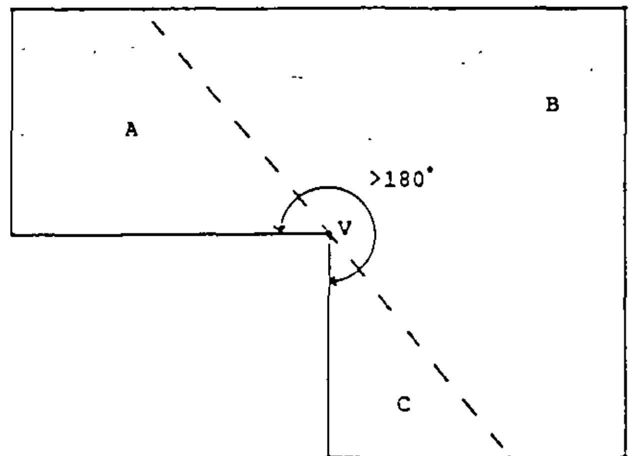


Figure 4. A simple, non-convex polygon with a single inside corner, V. The non-convex area can be decomposed into convex polgons "separated" by V.

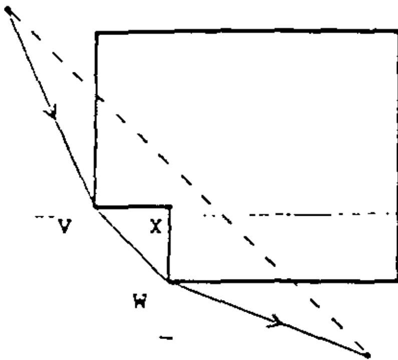


Figure 5. The shortest path around an obstacle passes through one or more inside corners (e.g., V, W). Outside corners (e.g., X) are never considered.

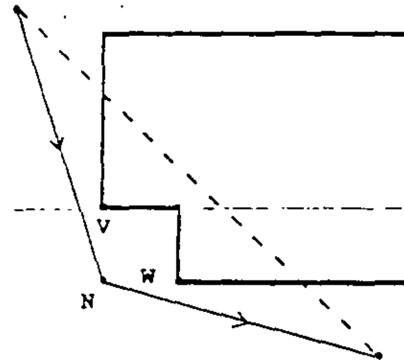


Figure 6. Two inside corners that are sufficiently close (e.g., V, W) may be replaced by a single interior node (N) to reduce the search space for indirect paths.

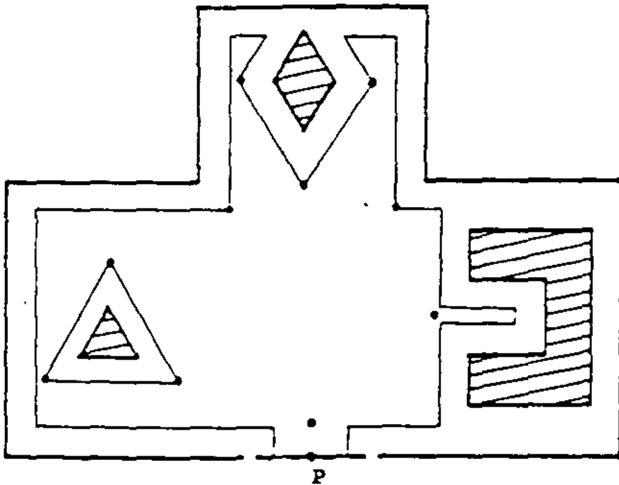


Figure 7. The complex configuration space is shown superimposed on the region from Figure 2. Interior nodes are highlighted and the passageway (P) is marked.

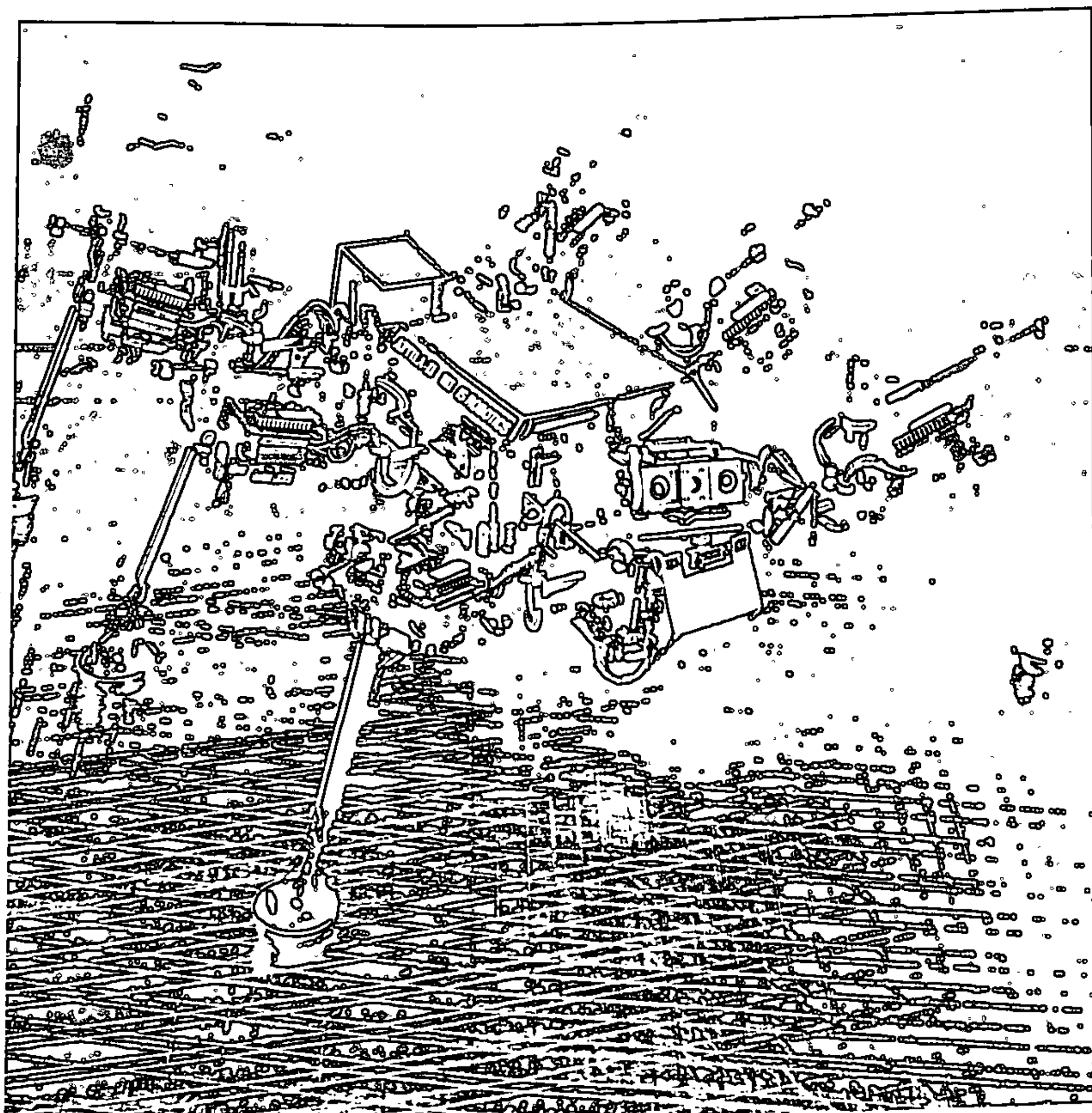
DECEMBER 1991
$3.95

SCIENTIFIC AMERICAN

Quantizing the birth of the universe.

Reusable fuels from solar energy.

When the first horse was broken to the bit.



Insectoids: can artificial intelligence give robots the IQ of a smart bug?

Researchers are trying to build machines that emulate the reasoning and self-awareness of humans, but in the real world even the competence of a mayfly eludes them—for now.

Squish... Whirr... Scrape. Scrape. Scrape. A robot is buttering toast. It's doing a fairly good job, considering that it is just learning.

Although humans do it easily, by machine standards the simple act of buttering toast is very difficult indeed. It is also a task that stands proxy for a host of other problems that confront machines aspiring to get along in the real world. A conventional robot, which knows only how to move its gripper along a specific path, would almost certainly make a mess of both toast and butter.

Buttering toast requires continuous modification of actions based on sensory feedback. Too much resistance, ease off the pressure; too little, change the angle of the knife, and so on. Calculating the correct path in advance would call for precise measurement of the viscosity of the butter and of the toast surface, a nonlinear finite-element model of the spreading process as well as several hours of time on a Cray supercomputer.

This experimental robot in Stanley J. Rosenschein's laboratory at Teleos Research in Palo Alto, Calif., is one of a new breed of machines that represent the leading edge of artificial-intelligence (AI) research. Unlike AI systems based on vast stores of arcane know-how, such as expert systems, this nascent tribe is supposed to act with perception and be imbued with common sense. The effort is pushing the state of the art not only of machine vision and sensing but also of automated reasoning, planning, knowledge representation and natural-language understanding.

These are the subdisciplines that artificial intelligence split into a generation ago, when it became clear that it would take more than simple programming to produce a computer that could be regarded as intelligent. Then, two years ago, AI pioneer Allen Newell of Carnegie Mellon University issued a call for researchers to put the field back together again, to build what he called "integrated intelligent systems." The prowess demonstrated by AI's parts, he said, was now sufficient to build a whole.

In response, Newell says, "a whole bunch of interesting characters emerged from the woodwork." Most of them were already working on "autonomous vehicles" or "intelligent agents" or just strange things. They shared the common goal

FORCE-SENSING ROBOT can spread butter, but it still can't taste breakfast. Interaction with the world is largely an unsolved problem for machines that aspire to human-like intelligence and even a sense of [illegible] existence

How Not to Butter Toast

of attempting to build not just "smart" computers but mechanical creatures that could function independently in the world.

Two major issues daunt those who have responded to Newell's call. Researchers disagree on the fundamental issue of what constitutes intelligent behavior. Second, they divide into at least two major camps on the question of how to get from the current state of the art to whatever they think intelligence may be. Traditional AI researchers such as Newell believe in reasoning, learning and symbolic processing. More sophisticated algorithms and faster hardware, they predict confidently, will eventually engender smart machines. Meanwhile young turks such as Rodney A. Brooks of the Massachusetts Institute of Technology avoid at all costs anything that might look like rationality by designing mechanical creatures that act entirely according to reflex.

Even if a consensus existed among AI researchers (a suggestion that people in the field regard as humorous), putting artificial intelligence back together would not be easy. Expert systems are solving a myriad of problems, from diagnosing the ills of diesel engines to rating midsize banks. Automated planners schedule aircraft maintenance and help design assembly lines. Natural-language systems can parse most sentences as well as a person can. But experience demonstrates that modules from these diverse disciplines cannot simply be combined and work smoothly together.

At one recent workshop, reports Kenneth D. Forbus of Northwestern University, "thing" builders spoke optimistically of an autonomous machine that could perform a variety of tasks and survive unaided in the world for a year. "So I went around and asked people, 'How long does your system last?'"

The answer was that no one had yet built anything that survived more than a few hours. The causes of death are as varied as the machines themselves. Sometimes the software just grinds to a halt, Forbus says, or sometimes the batteries run out. Most often, though, an autonomous system simply gets into a situation from which either reasoning or servomotors cannot extricate it—say, stuck under a chair.

In blunt terms, the integrated-intelligent systems researchers have yet to build a machine with the survival abilities of a mayfly—much less the sensory and behavioral smarts to find another mayfly and mate with it.

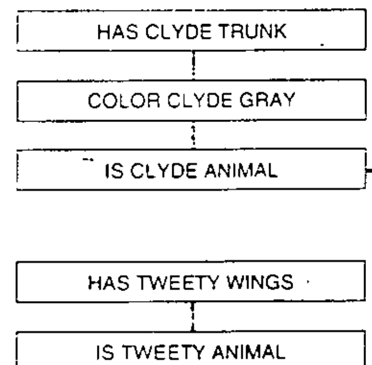
Modules from different parts of AI do not work together, because the re-

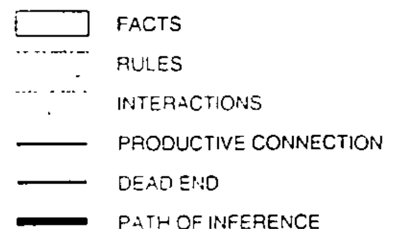
Knowledge Representation

Artificial-intelligence programs usually represent their knowledge in the form of logical assertions in a specialized computer language. Some assertions are simple facts—(HAS TWEETY WINGS) (ISA TWEETY BIRD)— whereas others, typically called rules, encode information about connections. (IMPLIES (HUNGRY CLYDE) (GOAL CLYDE (AND (POSSESS CLYDE ?X) (EDIBLE ?X)) (CONSUME CLYDE ?X))) [Rough translation: "If Clyde is hungry, he will want to possess something that is edible and consume it."]

An "inference engine" responds to new assertions by searching its knowledge base for additional facts based on rules and existing facts. For example, if a system were told that Clyde is in fact hungry, it would assert that he has the appropriate goals. That conclusion drawn, the system might search its knowledge base for edible objects and then assert that Clyde wanted to possess and eat one of them (*below*).

Reasoning from knowledge and





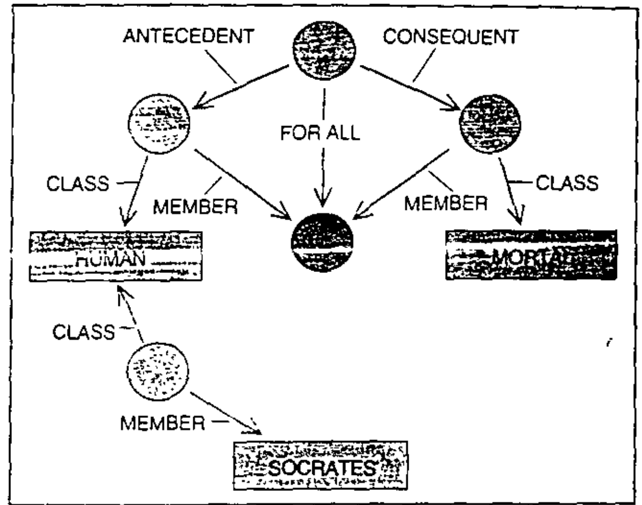
21

rules to new knowledge is typically called forward chaining. Inference systems can also chain backward from existing knowledge to answer queries. For example, if asked whether Clyde wanted to possess a crescent wrench, the system would see that being hungry can be a reason for wanting to possess an object, and so it would check whether Clyde is hungry and whether crescent wrenches are edible. Failing either of those possibilities, it would continue searching for other rules that might eventually imply (GOAL CLYDE (POSSESS CLYDE CRESCENT-WRENCH)).

Encoding facts and actions is a tricky problem. Consider the "Gravity drowned" problem: When a natural-language understanding researcher was first trying to represent the concept of falling, he translated "X fell" as "Gravity carried X downward." Elsewhere in his system was a rule that if you fell in a river and could not swim, and had no friends to rescue you, you drowned. Since gravity as a force of nature has neither arms, legs nor friends, it meets its unfortunate—and improbable—end.

Other facts, such as "Fish swim," are not really facts at all, so attempting to encode them yields endless trouble. Some fish walk on their fins, and others are baked. Researchers have turned to "default reasoning" ("Fish usually swim") and "nonmonotonic inference" (retracting any conclusions that may have depended on the belief that a particular fish swims) to produce automated deductions in the presence of uncertainty and error. The success of both techniques is mixed.

If the objects in a knowledge base have complex relations to one another— and they usually do—researchers often resort to semantic network representations (*above right*). Nets consist of nodes that stand for the object in question and of links that show nodal connections. Links can denote simple relations such as IS-A or PART-OF or more complex ones such as LEGALLY-ENTITLED-TO, BELIEVES or OPERATES-ON. A program seeking to deduce information about a particular object in the semantic network need only follow the appropriate links.



Semantic nets can also use a technique called spreading activation to model human cognitive behaviors such as free association. Spreading activation passes markers outward from nodes representing particular objects or concepts. When the markers from different nodes cross, the paths back to the parent nodes represent a chain of associations between the two objects

QUERY
(GOAL CLYDE
(POSSESS CLYDE CRESCENT-WRENCH))

IMPLIES (AND (IS ?X ANIMAL)
(NOT-EATEN-LATELY ?X))
(HUNGRY ?X)

IMPLIES (HUNGRY ?X)
(GOAL ?X (AND (POSSESS ?X ?Y)
(EDIBLE ?Y)))

IMPLIES (IS TOOL ?X)
(NOT (EDIBLE ?X))

IMPLIES (GOAL ?X USE-TOOL-TO-OPEN)
(GOAL ?X (AND (POSSESS ?X ?Y)
(IS TOOL ?X)))

ANSWER
(QUERY (NOT-EATEN-LATELY CLYDE))

EDIBLE PEANUTS
EDIBLE BROCCOLI
EDIBLE BRAZIL NUTS
EDIBLE CASHEWS
IS CRESCENT-WRENCH TOOL

IMPLIES (AND (GOAL ?X (POSSESS ?X ?Y))
(NOT (KNOW ?X (LOCATION ?Y))))
(GOAL ?X (FIND-LOCATION ?Y))

AND (CONTAINS CRATE-34 PEANUTS)
(CLOSED CRATE-34)

IMPLIES (AND (GOAL ?X POSSESS ?Y)
(IN-CLOSED-CONTAINER ?Y))
(GOAL ?X USE-TOOL-TO-OPEN)

RODNEY A. BROOKS of M.I.T. and thoughtless friend Genghis are leading exponents of the belief that intelligent behavior has no need of reason.

searchers who built them have answered questions that those in other specialties were not asking, and they have for the most part neglected other issues that are turning out to be crucial for integrated intelligence. For example, says Thomas M. Mitchell of Carnegie Mellon, the underlying paradigm of most machine vision algorithms is simply misguided. "Much of the work was how to get a program to interpret a [single] picture" and identify the objects in it. But artificial creatures that exist in the world must interpret a stream of images, each differing only slightly from the one before it. Furthermore, they may care less about the identity of an object than whether they should avoid running into it.

Although machine vision systems are being used in commercial robots, until recently specialists in the field largely ignored the question of vision for mobile machines. Mitchell, for one, believes problems ranging from binocular vision to the identification of ambiguously shaped objects can be solved simply by moving a robot's eyes [illegible] computing power. [illegible] work by [illegible] H. Ballard of the University of Rochester on "animate vision," which has shown that many image-processing [illegible] can be simplified immensely [illegible] taking into account [illegible] [illegible] heads and can fix both eyes on the object of their attention.

Then there is the problem posed by so-called systems of truth maintenance and default reasoning, which attempt to cope with uncertain or contradictory information. Take, for example, the statement: "Birds fly." That's true, as Marvin L. Minsky of M.I.T., among others, has observed, unless they are penguins or ostriches or moas or dodoes, or unless they have had their wings clipped or are confined to a small box, or unless they are dead and nailed to a perch, or unless they have been psychologically conditioned from fledglinghood to stay on the ground, and so on. Instead of cogitating furiously in an attempt to determine whether it should [illegible] a priori that a particular bird [illegible] Mitchell says, "a robot can just [illegible] to discover if it can fly."

Attila the Bug

Brooks is the most visible member of [illegible] that wants to consign [illegible] vision, truth maintenance and [illegible] of the rest of artificial intelligence to the dustbin of history. They contend that AI's separate subdisciplines have essentially been looking under the lamppost—choosing entire [illegible] of study for their conceptual elegance [illegible] rather than for their usefulness. In papers titled "Intelligence without Reason" or "Elephants Don't Play Chess," he argues that the fancy logic and knowledge-representation techniques beloved of artificial intelligencers have no place in machines designed to live in the world.

Instead Brooks pins his hopes on what he calls "subsumption architecture," in which complex behaviors such as exploration of the environment are built up from simple ones like moving a single leg. Subsumption architecture relies largely on the nature of the outside world rather than sophisticated reasoning to structure the robot's actions. For example, if the robot encounters an obstacle, the important thing is to go around it, not to determine how it got there. The robot may not even need to remember that the obstacle is there—after all, it will detect the obstacle perfectly well the next time it approaches it.

Such a procedural knowledge representation, says M.I.T. researcher Pattie Maes, avoids the conventional AI impasse of trying to construct and maintain a consistent logical model of the outside world. "Making the correspondence between perception and internal representations is too hard," she says.

Brooks's insectoid robots, dubbed Attila and Hannibal, contain multiple microprocessors and work by coupling sensors, processors and actuators [illegible] a tight loop [see "Mathematical Recreations," SCIENTIFIC AMERICAN, July]. The legs, for example, spend most of their time running a program that checks their position and keeps them standing. If conditions trigger the robot's walking behavior, a central processor sends "Walk" signals to the legs but does not actually coordinate their behavior. The individual programs built into each leg operate independently. Maes contends that the "emergent behavior" of these parallel processes may be a better mimic for what actually happens in the brain than the "folk psychology" of a centralized, all-controlling conscious mind.

As simple as it seems, subsumption can generate fairly complex activity. One of the robots that the group designed wandered the halls of the M.I.T. Artificial Intelligence Laboratory scavenging soda cans (at least once, the published claim goes) without any explicit concept of what it was doing. The "approach" module brought the robot's gripper close to anything can-shaped, and the gripper simply closed whenever anything appeared between its claws.

Brooks has suggested that tiny mobile robots, weighing no more than a kilogram, might perform such tasks as

exploring the terrain of a distant planet (or this one) more efficiently than larger, individually more reliable systems.

Working along these lines, M.I.T. graduate student Maja Mataric designed a software module that produces a subsumption-style map of a robot's surroundings. Instead of the usual data structure containing objects and their locations, or perhaps even a robot-sensor view of the world, the subsumption map is a collection of little computational processes, one of which is activated for each "interesting" place in the robot's world.

Maes has even designed a subsumption-based architecture that incorporates "beliefs" and "motivations." A robot based on her designs would be potentially able to answer the question "Why did Attila cross the road?" The robot could tell a human being where it thought a particular object was located or what caused it to go from one room to another. Much like the "behavior" agents that drive walking or scavenging behavior, the "belief" agents in Maes's so-called agent network architecture govern the robot's actions at a high level. For example, a "battery low" belief agent might suppress exploration in favor of a "return to base" behavior.

Predictably, the makers of more conventional, less photogenic systems are charitable toward Brooks at best, hostile at worst. Nils J. Nilsson of Stanford University, who directed AI work at SRI International 20 years ago in the days of Shakey, the first mobile robot, retreats to W. H. Auden when pressed for an opinion. "Those who will not reason/Perish in the act:/Those who will not act/Perish for that reason."

Mitchell is blunter. Brooks and his colleagues, he says, "are very seductive, but they have very bad ideas." Although he admits that Brooks's team has been "surprisingly successful in building systems that behave well," Mitchell contends that the subsumption architecture is better suited to building thermostats than intelligent agents.

Yet one of the most approving comments comes from Newell, whose SOAR architecture is the apotheosis of elegant reasoning and knowledge representation. Newell contends that SOAR and subsumption are not really so very different, because SOAR, too, operates essentially on a stimulus-response basis except when its rules run into an impasse. He praises the idea of "reactivity" that he sees in the little insectoids.

Regardless of whether Attila and its kin scuttle to fame or blunder into a philosophical dead end, or do both, Brooks's work raises some fundamental issues about the nature of intelligence. What does it mean to say that a system knows something? Is it enough for the system to behave as if it knows not to run into walls, or must it have some explicit representation of structural engineering?

Researchers may find it simpler to think in terms of logical propositions and inference, Rosenschein says, but that does not mean everything inside a machine must be represented in terms of explicit logic. "Some people are impressed by the precision of logical languages and their capacity for rubbing facts up against one another," he says, but in real machines elegance must defer to efficiency. Determining the truth of a logical proposition is an "NP-hard" problem, one that in principle cannot be solved in any reasonable time. That is not a good thing for a mobile robot to be computing.

The problem, Rosenschein contends, is that designers in pursuit of logical elegance mix static facts, which are always true, with dynamic information about the immediate state of the world. Facts that can change, such as who works in what office or whether the elevator door is open, benefit from explicit representation. But constants such as gravity or the impenetrability of material objects need not be explicitly coded. A system has only to act as if it knows them. Rosenschein and his colleagues have developed software tools that build the static knowledge into the structure of an autonomous system.

But the designer of an intelligent machine must still decide what knowledge the system should appear to have and how that knowledge should appear to be represented. The issues are far from resolved. Standard logic, for example, can represent facts, but it cannot represent intentions or beliefs about the world or about the results of particular actions, says Stuart C. Shapiro of the State University of New York at Buffalo. Richer logical formalisms bring with them the danger of creating paradoxes that can bring a reasoning engine to its knees, he notes [*see box below*].

Furthermore, Shapiro says, the logic of belief and the logic of action are not the same. Software based on conven-

The Paradox Explosion

The barber shaves every man who does not shave himself. Who shaves the barber? With these simple sentences Bertrand Russell demolished the apparent perfection of predicate logic and mathematical set theory. The barber paradox (or, for the set-theoretically inclined, the set of all sets that do not contain themselves as members) illustrates the pitfalls of allowing statements to refer to the truth of other statements.

Artificial intelligence researchers care about this ancient chestnut because most of their programs are firmly based on logical inference. The laws of logic provide that it is possible to deduce anything from a contradiction, and so the possibility of paradoxes spells disaster for AI.

The incantation runs as follows: if "A" is true, then obviously "A or B" is true. Meanwhile, if "A" is false and "A or B" is true, then "B" must be true. But a contradiction says that "A" is true and "A" is false, and so "B" must be true whatever it is.

There are two paths out of the quandary, according to Stuart C. Shapiro of the State University of New York at Buffalo. One is the version that mathematics chose: first-order predicate logic, which does not permit statements that refer to one another's truth value. No paradoxes, but the system is incomplete, as Kurt Godel showed in the generation after Russell. A statement may be manifestly true but impossible to prove in formal terms.

The other path is to impose (logically speaking) arbitrary constraints on relevance, so that deductions cannot simply run away into infinity, Shapiro says. Knowing that the moon both is and is not made of green cheese, for example, might lead an automated reasoner to bizarre conclusions about lunar soil samples, but it should not permit the machine to decide that the color of the sky determines shipping tariffs for Tibetan yak fat.

Ironically, Shapiro explains, the structure of simple automated deduction systems prevents such blowups automatically. These programs never try to reason from items that are not directly linked to one another. Only the most sophisticated and complete programs, after long and arduous work by their authors, can haul together completely unrelated facts and explode when confronted by a contradiction.

24

tional logic, such as the PROLOG programming language, typically forms a belief only once, even if it is used as a logical step in several different chains of reasoning. Applying the same economy measure to actions would lead you to conclude, for example, that you had no need to walk in order to get out your front door in the morning, because you had already walked when you went to the bathroom to brush your teeth.

Logical representations capable of dealing with both actions and beliefs are essential to intelligent agents. Not only will such machines be required to reason about the consequences of their own actions, Shapiro points out, they also will have to represent and understand the beliefs and actions of people and other intelligent agents. For example: "John believes that Mary believes that the program is faulty, but John believes that the program is not faulty, and yet Mary is right."

Such logical regressions are reflected in the work of Thomas Dean of Brown University, among others. Dean's systems not only plan out their actions, they even plan how much planning to do. Those who think about these "anytime algorithms" are painfully aware that computing resources are not infinite—like people, machines can exercise only what Herbert A. Simon calls "bounded rationality."

PROBLEM SOLVERS Allen Newell of Carnegie Mellon University (*left*) and John E. Laird of the University of Michigan are two chief architects of SOAR, an integrated reasoning system that is now nearing its 10th birthday. Variants of the system can

Chunks of Knowledge

Reasoning techniques that produce exact answers eventually are no good to machines that must operate in "real time." (The precise definition of real time is also a matter of controversy, but Mitchell offers the handy rule of thumb that the time spent thinking generally should not be longer than the action being thought about.) Indeed, reasoning techniques that quickly produce approximate answers may not be any good either if they cannot be interrupted partway through. As anyone knows who has watched a treasured possession heading for the floor, virtually any action may be better than inaction.

Advocates of such meta-planning tend to use algorithms based on "iterative refinement." Run them once and they produce an answer; again and they produce a better answer, and so on. Given information about how answers improve with time and about the cost of delay, a system can decide how long it should think before acting. There is, Dean admits, yet another level of regression: optimizing the amount of time to spend planning is itself an unsolvable problem, so the meta-planner must either make approximations or else plan how much time to spend planning how much time to spend planning... ad infinitum.

Dean has proved mathematically that the logical regression "eventually bottoms out," but he acknowledges that intelligent machines based on his algorithms are still in their formative years, "just at the stage where they're robust enough to walk down the hall without leaving huge gouges in the plaster." He predicts that it will probably be the end of the decade before most planning and reactive systems stop using what amounts to ad hoc methods to allocate computational resources.

One such relatively simple logic scheme is the brains of SOAR, a logic system that Newell's group has been running in various incarnations for nearly nine years. It is based on two techniques, search and "chunking."

When the system is confronted with a problem, it checks whether any of the rules in its knowledge base apply and fires off a train of deductions until it can go no further. If the problem is solved, fine. If not, SOAR (which once upon a time stood for State, Operator And Result) sets up a "problem space" to deal with the impasse. Within this problem space, it searches for all possible solutions, using special-purpose techniques if they are available and brute force if not. Once SOAR has found an answer, it generalizes the technique it used in the solution and compiles the technique into a "chunk"—a rule that will be activated the next time a similar question comes up and so eliminate the burden of searching.

SOAR's module for natural-language understanding (NL-SOAR) is a perfect example of this technique, Newell says. It simply applies the operator "Understand this word in context" to each new word of input, rattling off deductions, and chunking with abandon. According to John E. Laird of the University of Michigan, who developed SOAR along with Newell and others, NL-SOAR starts with about 900 rules and expands its knowledge base to more than 1,500 after comprehending only a few hundred words. Newell claims that the system is, in principle, capable of learning new words and rules indefinitely. He speaks of running SOAR for a month straight, continually "adding knowledge in interesting ways."

Then he backpedals. As the system stands now, "that's not to say that it won't seize up in a couple of hours," he says. The flaw resides somewhere in the initial rules that humans have hand coded into the system. Somehow

control mobile robots, understand plain English and perform a variety of other more or less practical tasks.

the structure of these chunks makes for bad learning. Oddly enough, Newell says, the fewer rules people put in and the more SOAR has to learn by itself, the better it works.

In addition to the natural-language system, SOAR also runs a couple of robotic applications: RoboSOAR, a robot arm, and HeroSOAR, a mobile robot. RoboSOAR, which depends on a separate vision module to determine where small objects reside in its work space, can perform simple tasks such as picking up and stacking blocks. HeroSOAR investigates its environment with ultrasonic sensors, paying special attention to things that look like garbage cans.

Ultimately, Laird says, all the different modules of SOAR will be incorporated into one. That will enable a robot to accept plain English commands and answer back in kind as it performs its tasks. But first the researchers have to solve some problems with the vision system and other modules.

The vision software analyzes scenes in a form that does not always match the information that SOAR's problem solver might want. Furthermore, the camera is mounted directly above the robot's work space so that RoboSOAR can't actually see what it is doing. The system must analyze a scene, move its gripper and attempt to pick up objects, then move the gripper out of the way and find out whether it has been successful. It's a little bit like a human being wearing heavy mittens trying to play chess by taking a snapshot of the board, deciding what move to make and then, eyes closed, reaching out for the piece.

Like SOAR, Mitchell's THEO system uses a conceptually similar "plan then compile" architecture. Mitchell contends that integration is not nearly enough; organization is the key. Without some way of focusing on the facts relevant to the case at hand, a program will bog down. "We quickly learned that if a system has five rules it does okay, but 5,000 rules will slow it down hopelessly," he says. (Newell reports the same experience with SOAR. Adding a single "expensive chunk" to the knowledge base can set off a logical chain reaction that reduces the program's speed by a factor of four.)

Besides cluttering up memory space, much of what a system learns by looking at examples, though true, may be irrelevant or even misleading. Mitchell likens the problem to an automated version of the "cargo cults" that led Pacific Islanders to build airstrips and wooden aircraft to entice Western goods and know-how from the sky.

Chunking, Mitchell explains, may speed up the execution of plans, but it doesn't make them any more correct. If the plan is based on an erroneous understanding of the world, it will sometimes fail. Ideally, an intelligent system would learn from its failures, but getting it to learn the right things is a problem. "There is no perfect model for the effects of some actions," Mitchell says, so the best course may in fact be to let THEO fail every now and then and try to recover from its failures, rather than try to anticipate beforehand everything that could go wrong.

AI Meets Virtual Reality

One requirement that Mitchell emphasizes when he talks of letting THEO and other intelligent agents "experiment" with the world around them is that no harm must come to the robot or its subjects. So a number of intelligent-agent researchers have created computer simulations to exercise the programs that will be the brains of their autonomous machines.

One of the simulacrum dwellers that other researchers find most impressive is Homer. Built by Steven A. Vere of the Lockheed Artificial Intelligence Center in Palo Alto, Calif., Homer is a virtual submarine that lives in an aquatic world containing ships, fish, whales, other submarines, mines, buoys, birds, islands, docks and people.

Homer understands about 800 words and can carry out multistep tasks—for example, "Bring the red buoy to the dock before noon and then meet Roger at the breakwater tomorrow morning." It knows how to ask people for information when it doesn't know where something is, and it continually tracks events in its environment.

In fact, a printout of a session with Homer bears a strong resemblance to a transcript of a day spent with a precocious three-year-old:

"I see a brown bird on the gray log."

"There are two sailboats by the island."

"The *Smirnov* just passed me on the right again."

"I have reached the dry dock."

"Homer, bring me Fred's buoy."

"Now I'm going to Fred."

"Fred, where is your buoy?"

Vere admits that having a simulated world to work in has simplified much of his work, but he insists that most of Homer's capabilities could be transferred to a real minisub. In particular, he says, Homer's uncannily acute simulated vision is not particularly farfetched. Because the marine environment contains a set of objects so small, a fairly simple perception system would be able to distinguish between fish, whale, ship, sailboat, log, bird and landmass. Homer would be even more competent if it were transplanted to a conceptually similar (although geographically disparate) artificial environment such as a space station, where every object could be marked for unambiguous identification.

At first glance, Homer appears to be a success story for integrated intelligent architectures. It can plan its actions, change plans in response to new circumstances and interact with others. Under the surface, though, the sailing is not so smooth, Vere says. Homer is slow, and its memory is inefficient. The longer it lives, the slower it runs. Just a few scenarios strung together will bring the program to a near halt. "It's like a house of cards," Vere laments. "When you add a new capability, you discover that old scenarios don't work any more. You have to go back and retune it."

Homer's life is also threatened by corporate budgetary woes and the gradual extinction of its research team. Vere's colleague Timothy W. Bickmore left Lockheed a year ago, and the company now permits Vere to spend only half of his time on the project. Although new computer hardware has

helped improve Homer's reaction time from miserable to tolerable (30 seconds to plan a simple course of action, down from several minutes), the dream of transplanting it to a real minisub is on indefinite hold.

Just down the road from Vere, at the National Aeronautics and Space Administration Ames Research Center, another simulated agent has a somewhat firmer footing. Mark E. Drummond's Entropy Reduction Engine (ERE) is a conceptual model for the kind of robot that might one day help maintain structures in orbit or on the moon.

For now, ERE lives in Tile World, a gridded Flatland that contains any number of polygonal tiles. Tile World is beset by winds that can move tiles from their places, and the robot's attempts to move or clutch them in one or more of its four grippers may fail. Such a simple environment provides more than enough diversity to work out most of the important issues in planning and control, Drummond says.

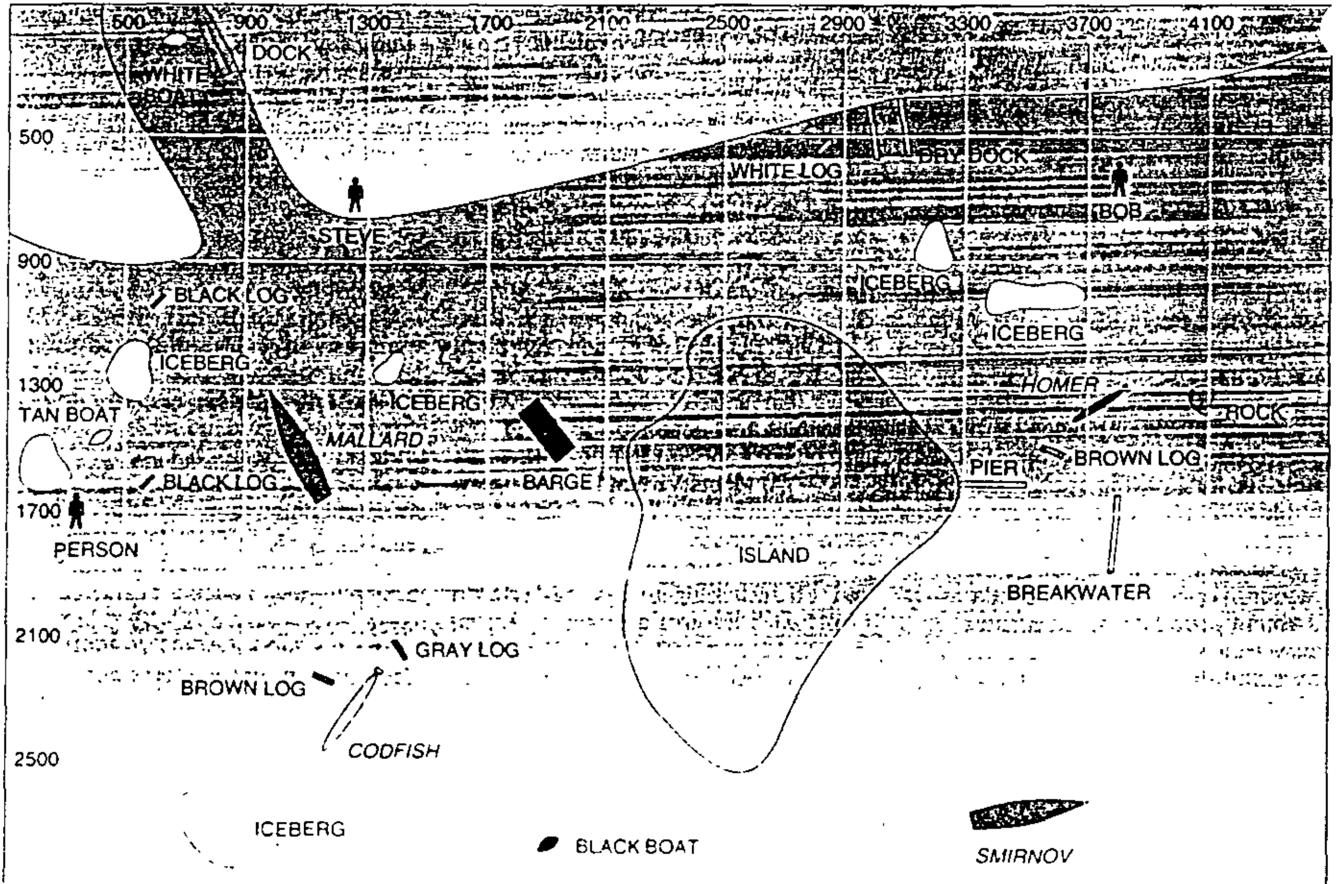
ERE consists of components that decompose problems into simpler form, evaluate possible solutions and compile rules that put those solutions into effect. It is designed not only for "achievement goals," such as assembling tiles into a particular configuration, but also for what Drummond calls behavioral constraints—for example, making sure that a particular tile structure remains intact even under the influence of winds. (Eventually, of course, Drummond would like to be able to generalize such behavioral constraints to build the kind of robot that could be told: "Keep the space station in working order. This is a list of what can go wrong. Here are your tools and supplies.")

Drummond's interest in Tile World, however, goes beyond his own research. He hopes to promote the simple simulated environment as a standard within which the designers of different intelligent agents can compare their performance on various standardized tasks and thereby understand the strengths and weaknesses of various architectures. NASA Ames, along with Rosenschein's Teleos, has won a research contract from the Defense Advanced Research Projects Agency to define a set of benchmarks for integrated intelligent systems.

At one end of the spectrum of intelligence and action scuttle Brooks's insectoid robots. In the middle are the dozens of systems, such as those built by Vere and Drummond, possessing varying levels of knowledge and capability. At the other end majestically sits Douglas B. Lenat's Cyc, which knows almost everything—1.43 million interconnected facts at last count—but does almost nothing at all. Indeed, many would question Cyc's inclusion in a list of integrated intelligent systems because it is essentially an automated encyclopedia (hence the name).

One problem with conceiving of Cyc as an entity (even though, like SOAR, it is nearing the end of its first decade on this earth) is that Cyc has no real sense of itself, says Patrick Hayes of Stanford. Hayes, currently president of the American Association for Artificial Intelligence, was briefly head of Cyc's West Coast office until conceptual differences with Lenat made him decide to be a consultant to the project instead. "Cyc knows that there is this thing called Cyc and that Cyc is a computer program," Hayes says, "but it has no idea that *it* is Cyc."

Even so, the program is remarkable. MCC, the consortium of 56 high-technology computer companies that em-



SIMULATED SEA WORLD is the domain of Homer, an intelligent agent built by Steven Vere of the Lockheed Artificial Intelligence Center. Homer understands simple English commands and can make plans to carry them out.

SPANISH	FRENCH	GERMAN	ITALIAN	JAPANESE	Mandarin CHINESE	RUSSIAN	Brazilian PORTUGUESE
30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus	30 Cassettes + Triple Bonus
$265.00	$265.00	$265.00	$265.00	$285.00	$285.00	$285.00	$265.00

Learn Foreign Languages... Incredibly Fast!

Conversing in a foreign language is a major social and business asset... and brings new life to the worlds of travel, entertainment and relationships. The technique of *accelerated learning*, as conveyed by these proven foreign language courses, allows anyone to comfortably converse in a new language within 30 days.

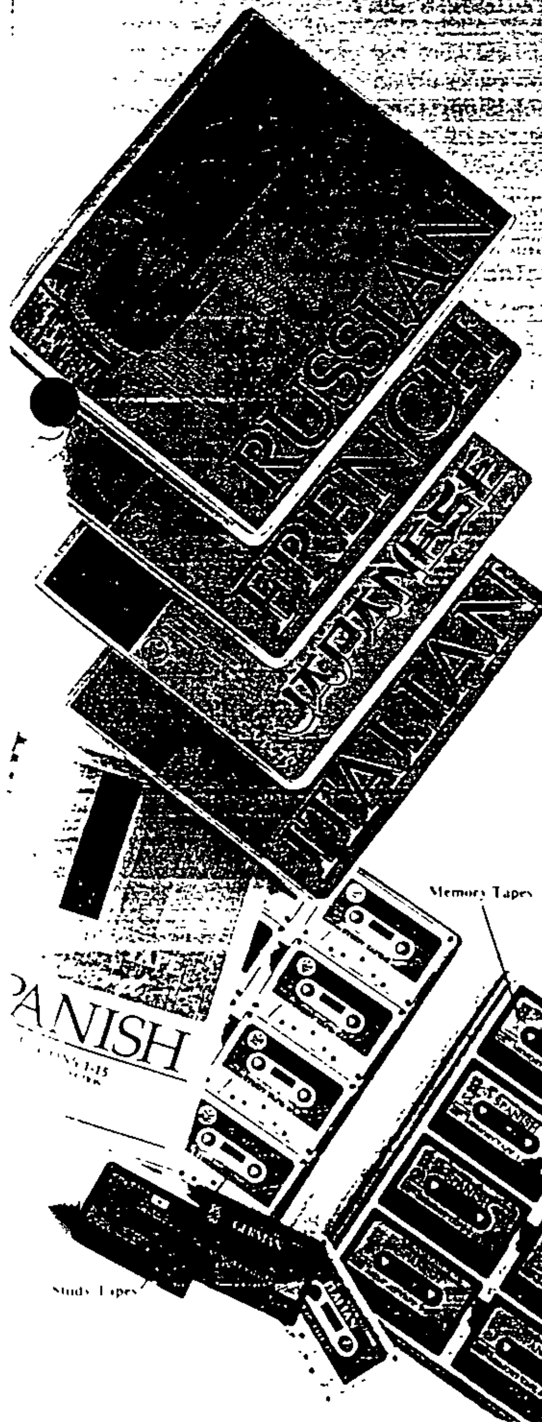
Accelerated learning, developed by famed learning expert **Dr. Georgi Lozanov**, is based on the premise of involving both hemispheres of the brain in the education process. The analytical or logical left side of the brain, when properly activated with the musical or artistic right side of the brain, both increases the speed and heightens the retention of learning. Utilizing these untapped mental capacities of your learning ability is the basis of this unique, highly effective course.

You *will learn* the language as stresslessly as a child does, by hearing new vocabulary and phrases in alternately loud, whispered, and emphatic intonations, all accompanied by slow rhythmic music in digital stereo. This perfect combination of music and words allow the two halves of the brain to work together to dramatically facilitate your assimilation of the new language.

The first 15 (memory) tapes of this 30-tape package help activate the learning capacities of the brain. The second 15 (study) tapes are the very same tried and proven tapes used by the **Foreign Service Institute** to train career diplomats. This marriage of two concepts literally gives you two courses in one, providing the best of both worlds in language instruction.

Best Value! With a total of 32 cassettes plus study materials, this program represents the best value available today in language instruction. Compared to other programs, the **Accelerated Learning Series** outperforms them with twice the audio and 20 times the study material.

To correctly converse in a foreign language, you must understand the meanings and intent of the native speaker. If, after 30 days of listening to the study and memory tapes, you are not *comfortably understanding and conversing in your new language, return them for a full refund.*



"American Managers with Language Skills Open More Doors"
—Wall Street Journal

"Company and marketing executives will find after 1992 that it is a handicap not to be fairly conversant with at least one other major European language – and preferably two or three..."
—The London Times

Triple Bonus !!
You'll also receive:
- Two 90-minute Vocabulary Tapes
- The 100-page *How To Learn A Foreign Language*
- The *American Express International Traveler's Dictionary*

TO ORDER: Phone or send your check, money order or Inst. P.O.

TOLL-FREE 24 HRS: VISA · M/C
1•800•85•AUDIO

Rush Orders PHONE 9-5 PDT:
1•818•799•9000

You may FAX your credit card order or company P.O. to:
1•818•792•7815

• INTERNATIONAL ORDERING INFORMATION •

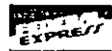
New! Now, for your ordering convenience, you may call our order desk toll-free 24 hours a day from any of the following countries via AT&T's International 800 Service.

BELGIUM	11-6599	NETHERLAND	06-022-4612
DENMARK	8001-0578	SINGAPORE	800-1625
FRANCE	05-90-1368	SPAIN	900-98-1120
GERMANY	0130-81-1139	SWEDEN	020-793-626
ITALY	1678-70-179	SWITZ	046-05-9632
JAPAN	0031-11-1907	UK	0800-89-7452

- ☐ FRENCH $265.00
- ☐ SPANISH $265.00
- ☐ GERMAN $265.00
- ☐ ITALIAN $265.00
- ☐ PORTUGUESE (Brazilian) $265.00
- ☐ JAPANESE $285.00
- ☐ RUSSIAN $285.00
- ☐ CHINESE (Mandarin) $285.00

Name________
Address________
City____ State____ Zip____
Credit Card No. Exp.____

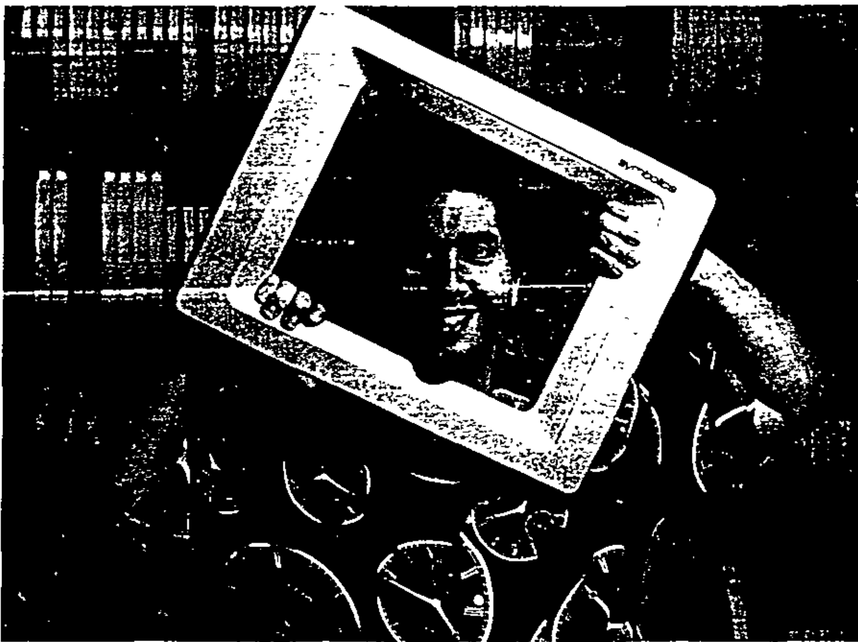
Signature (Card Orders Only)
☐ VISA ☐ MASTERCARD
Need It Tomorrow? Ask Operator for Express Service



Or Write To:
PROFESSIONAL CASSETTE CENTER
350 WEST COLORADO BOULEVARD
DEPARTMENT SCI
PASADENA, CALIFORNIA 91105 U.S.A.

Please add $11.00 shipping & handling
California residents add 8¼% sales tax
All Funds Payable in U.S. Dollars

¡Tambien tenemos cursos para aprender ingles!



WUNDERKIND Douglas B. Lenat of MCC is the instigator of Cyc, a decade-long project for encoding common sense. At its birth seven years ago, Cyc was considered near lunatic by many artificial intelligencers; now it is part of the mainstream.

ploys Lenat has invested half a programmer-millennium of effort into Cyc. There is probably an equal amount of work again remaining before—sometime in 1994 or 1995 by Lenat's hopeful reckoning—Cyc reaches the break-even level of about 10 million facts. At that point, it will be able to pick up new knowledge more readily by reading than by having knowledge engineers spoon-feed it. A wider information base may also save Cyc from such gaffes as concluding (from everybody it knew about) that all humans in the world are friends of Doug Lenat.

No Royal Road

The program is designed to have the kind of knowledge that an intelligent agent might need to perform its tasks: what people and trees and omnibuses are and how they interact, how objects fall and break, or how a light bulb works. Lenat envisions a day when Cyc will field requests for commonsense knowledge from programs for natural-language understanding, from expert systems in search of sensible behavior near the edges of their expertise. Cyc might even help people decide what kind of automobile to purchase. Almost anything might be relevant, he notes, from changing gender roles in the modern family to the relative traffic citation rates of cars painted navy blue or arrest me red.

For all its size, Lenat says, Cyc is "kitchen sink engineering": everything it knows is asserted in two different forms: first in clean and elegant "epistemological language" and again at the heuristic level in "a grab bag of different representations" that are designed to make inference faster for a particular class of facts. Thus far the system contains 27 different special-purpose inference engines, and Lenat intends to add more as they are needed.

Cyc thus avoids the trade-off between expressiveness and efficiency, Lenat contends. For example, by employing "a tool kit of partial solutions," Cyc has solved the problems of dealing with time, space, causality, belief and intention that frustrate other artificial intelligences. "Most of the power—positive or negative—of integrated intelligent architectures will come from the content, not the architecture," Lenat asserts.

Moreover, Lenat believes the idea that one particular structure of algorithms will make intelligent behavior possible is wishful thinking. "Intelligence is 10 million rules. If you have a halfway decent knowledge representation and a quarter-way decent architecture, they won't get in the way," he says. Researchers who think that a single elegant theory can solve all the problems of knowledge representation and inference suffer from "physics envy," Lenat contends. "They want a theory that's small, elegant, powerful and correct, and so they try one free-lunch tactic after another."

All the philosophical debates and engineering talent may seem a little much for picking polystyrene cups off the laboratory floor—assuming, that is, the containers are not glued down. When will the life span of integrated intelligent systems break the 24-hour barrier, and how?

It is tempting to believe that the natural progress of semiconductor wizardry will let people solve problems of perception and reasoning by brute force. After all, as Newell says, "if not for the fact that workstations have doubled in power every year, we'd be dead already." Faster, cheaper chips have already put mobile, camera-bearing robots within reach of most research groups.

In reality, however, Newell admits that integrated-circuit advances have at best kept big projects like SOAR a few paces ahead of the knackers. He is not happy thinking about the performance of systems containing perhaps 200,000 rules instead of the paltry few thousand that his machines run today. Mitchell concurs: "We'd like to think that silicon will save us, but it won't."

The answer instead may be nothing more complex than time and hard work. By their very nature, Newell points out, integrated intelligent systems are jacks of all trades—not on the forefront of any particular field of artificial intelligence. And now that planning, natural-language understanding and other disciplines have matured, he says, "you have to walk a long way to get to the frontier." But if artificial intelligencers can muster the determination to continue long beyond the span of any single graduate student or research grant, their silicon babies may yet grow to inhabit working bodies.

FURTHER READING

REFERENCE FRAMES FOR ANIMATE VISION. Dana H. Ballard in *Eleventh International Joint Conference on Artificial Intelligence Proceedings*. Morgan Kauffmann Publishers, 1989.

DESIGNING AUTONOMOUS AGENTS: THEORY AND PRACTICE FROM BIOLOGY TO ENGINEERING AND BACK. Edited by Pattie Maes. MIT/Elsevier, 1990.

UNIFIED THEORIES OF COGNITION. Allen Newell. Harvard University Press, 1990.

INTELLIGENCE WITHOUT REASON. Rodney A. Brooks in *Twelfth International Joint Conference on Artificial Intelligence Proceedings*. Morgan Kauffmann Publishers, 1991.

SPECIAL SECTION ON INTEGRATED COGNITIVE ARCHITECTURE (Proceedings of the AAAI Spring Symposium on Integrated Intelligent Architectures) in *ACM SIGART Bulletin*, Vol. 2, No. 4, pages 2-184; August 1991.

Understanding Ultrasonic Ranging Sensors

How the popular Polaroid sonar unit works and considerations for use on mobile robots for obstacle avoidance and navigation.

by H.R. Everett
Adapted from the book Sensors for Mobile Robots: Theory and Application, *published by AK Peters, Ltd, Wellesley, MA*
ISBN 1-56881-048-2

The Polaroid ranging module is an active time of flight (TOF) device developed for automatic camera focusing. It determines the range to target by measuring elapsed time between transmission of an ultrasonic waveform and the detected echo (Biber, et al., 1980). Probably the single most significant sensor development from the standpoint of its catalytic influence on the robotics research community, this system is widely found in the literature (Koenigsburg, 1982; Moravec & Elfes, 1985; Everett, 1985; Kim, 1986; Arkin, 1989; Borenstein & Koren, 1990). Representative of the general characteristics of a number of such ranging devices, the Polaroid unit soared in popularity as a direct consequence of its extremely low cost (the transducer and ranging module circuit board are available for less than $50), made possible by high-volume usage in its original application as a camera autofocus sensor.

Introduction to Sonar Sensors

All sensors, whether active or passive, perform their function by detecting (and in most cases quantifying) the change in some specific property (or properties) of energy. Active sensors emit energy that travels away from the sensor and interacts with the object of interest, after which part of the energy is returned to the sensor. For passive sensors, the source of the monitored energy is the object itself and/or the surrounding environment. In the case of acoustical systems, it must be recognized that the medium of propagation can sometimes have significant influence, and such effects must be taken into account.

Sound is a vibratory mechanical perturbation that travels through an elastic medium as a longitudinal wave. For gases and liquids the velocity of wave propagation is given by (Pallas-Areny & Webster, 1992):

$$s = \sqrt{\frac{K_m}{\rho}}$$

where:
s = speed of propagation
K_m = bulk modulus of elasticity
ρ = density of medium.

Since the introduction of sonar in 1918, acoustic waves have been successfully used to determine the position, velocity, and orientation of underwater objects in both commercial and military applications (Ulrich, 1983). It therefore seems only logical we should be able to take advantage of this well-developed technology for deployment on mobile robotic vehicles. This seemingly natural carry-over from underwater scenarios has been somewhat lacking, however, for a number of reasons. The speed of sound in air (assume sea level and 25°C) is 1138 feet/second, while under the same conditions in sea-water sound travels 5,034 feet/second (Bolz & Tuve, 1979). The wavelength of acoustical energy is directly proportional to the speed of propagation as shown below:

$$\lambda = \frac{s}{f}$$

where:
λ = wavelength
s = speed of sound
f = operating frequency.

This relationship means the wavelength for an underwater sonar operating at 200 KHz would be approximately 0.30 inches, while that associated with operation in air at the same frequency is in contrast only 0.07 inches. As we shall see later, the shorter the wavelength, the higher the achievable resolution. So in theory, better resolution should be obtainable with sonar in air than that associated with operation in water. In practice, however, the performance of sonar operating in air seems poor indeed in comparison to the success of underwater implementations, for several reasons.

For starters, water (being basically incompressible) is a much better conductive medium (for sound) than is air. In fact, sound waves originating from sources thousands of miles away are routinely detected in oceanography and military applications. One such example involves monitoring global warming (as manifested in long-term variations in average sea-water temperature) by measuring the associated change in the speed of wave propagation over a transoceanic path

Secondly, the mismatch in acoustical impedance between the transducer and the conducting medium is much larger for air than water, resulting in reduced coupling efficiency. The high acoustic impedance of water allows for conversion efficiencies from 50 to 80 percent, depending on the desired bandwidth (Bartram, et al., 1989). In addition, underwater systems are generally looking for fairly large discrete targets in relatively non-interfering surroundings, with the added benefit of intensely powerful pulse emissions.

And finally, one should keep in mind that untold billions of defense dollars have been invested over many decades in the research and development of sophisticated underwater systems that individually cost millions of dollars to procure, operate, and maintain. In contrast, most robotic designers begin to balk when the price of any sensor subsystem begins to exceed a few thousand dollars.

The range of frequencies generally associated with human hearing runs from about 20 Hz to somewhere around 20 KHz. Although sonar systems have been developed that operate (in air) within this audible range, ultrasonic frequencies (typically between 20 KHz and 200 KHz) are by far the most widely applied. It is interesting to note, however, ultrasonic frequencies as high as 600 MHz can be produced using piezoelectric quartz crystals, with an associated wavelength in air of 500 nanometers (Halliday & Resnick, 1974). (This wavelength is comparable to electromagnetic propagation in the visible light region of the energy spectrum.) Certain piezoelectric films can be made to vibrate in the gigahertz range (Campbell, 1986).

Acoustical ranging can be implemented using triangulation, time of flight (Frederiksen & Howard, 1974, Biber, et al., 1980), frequency modulation (Mitome, et al., 1984), phase-shift measurement (Fox, et al., 1983, Figueroa & Barbieri, 1991), or some combination of these techniques (Figueroa & Lamancusa, 1992). Triangulation and time-of-flight methods typically transmit discrete short-duration pulses and are effective for in-close collision avoidance needs, and at longer distances for navigational referencing. Frequency-modulation and phase-shift ranging techniques involving the transmission of a continuous sound wave are better suited for short-range situations where a single dominant target is present.

Ultrasonic-based measurement systems have found broad appeal throughout the industrial community for a wide variety of purposes such as non-destructive testing (Campbell, 1986), industrial process control (Asher, 1983), stock measurement (Shirley, 1991), liquid level measurement (Shirley, 1989), safety interlocks around dangerous machinery (Irwin & Caughman, 1985), and even intrusion detection in security scenarios (Smurlo & Everett, 1993). In the recreational electronics industry, major usage is seen in underwater sonar for depth and fish finding (Frederikson & Howard, 1974), and automatic camera focusing (Biber, et al., 1980). Typical robotic applications include collision avoidance (Everett, 1985), position location (Dunkin, 1985, Figueroa & Mahajan, 1994), and Doppler velocity measurements (Milner, 1990).

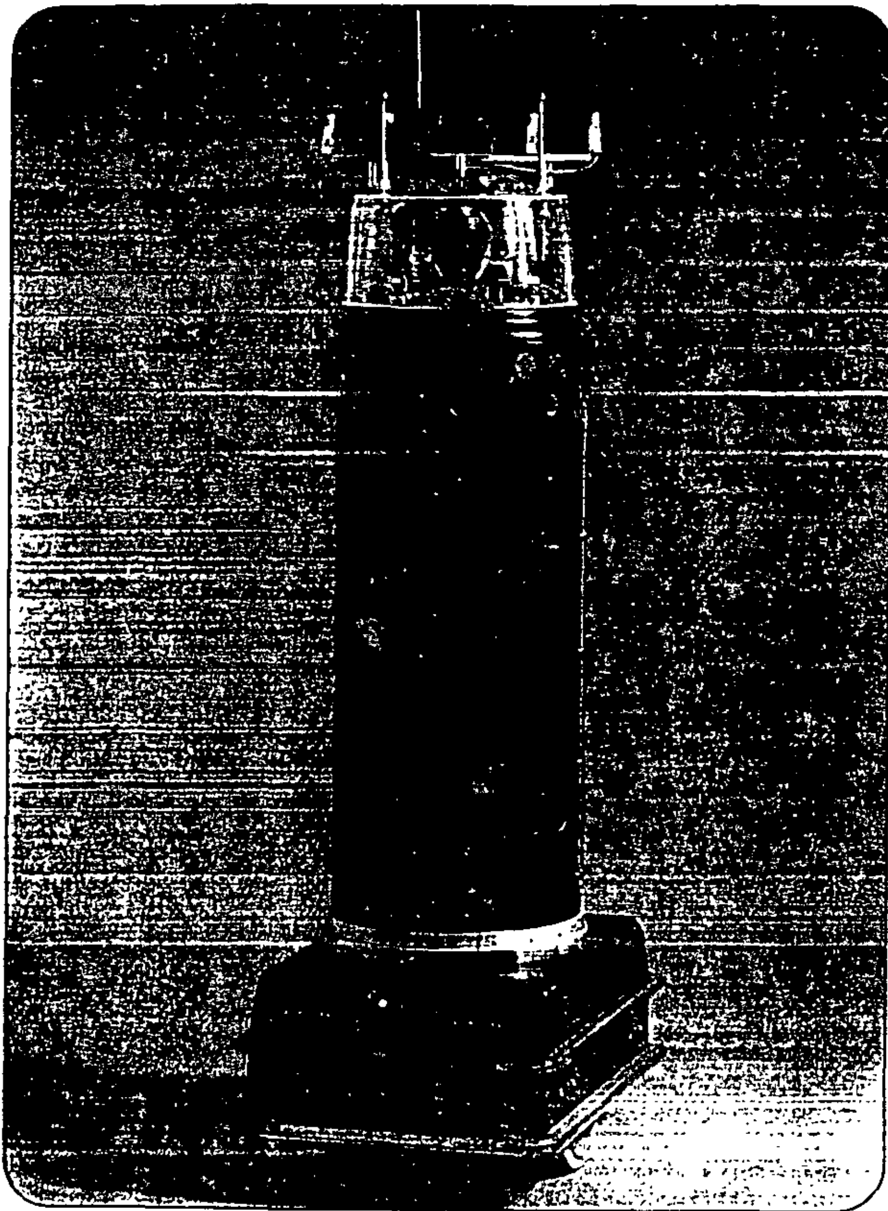


Figure 1: *ROBART II, an autonomous security robotic testbed, employs a total of 36 Polaroid electrostatic transducers for collision avoidance, navigational referencing, and intrusion detection.*

Polaroid Ultrasonic Ranging Module

The most basic configuration of the Polaroid ranging module consists of two fundamental components: 1) the ultrasonic transducer and 2) the ranging module electronics. A choice of transducer types is now available. In the original instrument-grade electrostatic version, a very thin metalized diaphragm mounted on a machined backplate forms a capacitive transducer (Polaroid, 1981). A smaller electrostatic transducer (7000-Series) has also been made available, developed for the Polaroid Spectra camera (Polaroid, 1987). A ruggedized piezoelectric (9000-Series) environmental transducer introduced for applications that may be exposed to rain, heat, cold, salt spray, and vibration is able to meet or exceed guidelines set forth in SAE J1455 January 1988 specification for heavy-duty trucks.

The original Polaroid ranging module (607089) functioned by transmitting a chirp of four discrete frequencies in the neighborhood of 50 KHz. The SN28827 module was later developed with a reduced parts count, lower power consumption, and simplified computer interface requirements. This second-generation board transmits only a single frequency at 49.1 KHz. A third-generation board (6500 series) introduced in 1990 provided yet a further reduction in interface circuitry, with the ability to detect and report multiple echoes (Polaroid, 1990). An Ultrasonic Ranging Developer's Kit based on the Intel 80C196 microprocessor is now available that allows software control of transmit frequency, pulse width, blanking time, amplifier gain, and achieved range measurements from 1 inch to 50 feet (Polaroid, 1993).

The ultrasonic ranging capability of ROBART II (Figure 1) is based entirely on the Polaroid system (three SN28827 ranging modules each multiplexed to 12 electrostatic transducers). For obstacle avoidance purposes, a fixed array of 11 transducers is installed on the front of the body trunk (as shown in the photo) to provide distance information to objects in the path of the robot. A ring of 24 additional ranging sensors (15 degrees apart) is mounted just below the robot's head and used to gather range information for position estimation. A final ranging unit is located on the rotating head assembly, allowing for distance measurements to be made in various directions. Reliability of the Polaroid components has been exceptional, with no failures or degraded performance of any type in over eight years of extended operation.

A typical operating cycle is as follows.

- The control circuitry fires the transducer and waits for an indication that transmission has begun.
- The receiver is blanked for a short period of time to prevent false detection due to residual transmit signal ringing in the transducer.
- The received signals are amplified with increased gain over time to compensate for the decrease in sound intensity with distance.
- Returning echoes that exceed a fixed-threshold value are recorded and the associated distances calculated from elapsed time.

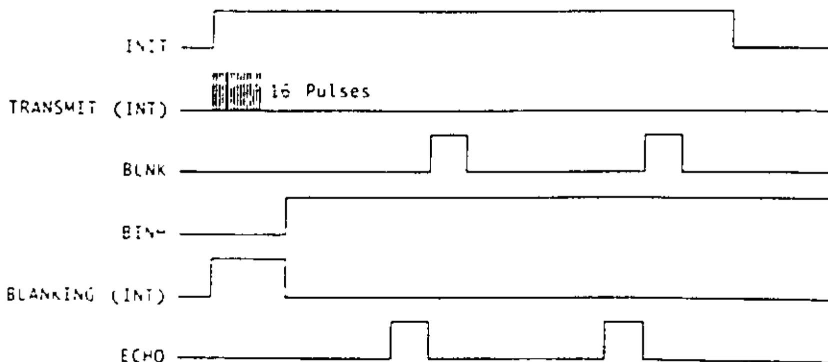


Figure 2: *Timing diagrams for the 6500-Series Sonar Ranging Module executing a multiple-echo-mode cycle with blanking input (courtesy Polaroid Corp).*

In the single-echo mode of operation for the 6500-series module, the blank (BLNK) and blank-inhibit (BINH) lines are held low as the initiate (INIT) line goes high to trigger the outgoing pulse train. The internal blanking (BLANKING) signal automatically goes high for 2.38 milliseconds to prevent transducer ringing from being misinterpreted as a returned echo. Once a valid return is received, the echo (ECHO) output will latch high until reset by a high-to-low transition on INIT. For multiple-echo processing, the blank (BLNK) input must be toggled high for at least 0.44 milliseconds after detection of the first return signal to reset the echo output for the next return as shown in Figure 2 (Polaroid, 1990).

Performance Factors

There are three basic types of ultrasonic transducers: 1) magnetostrictive, 2) piezoelectric, and 3) electrostatic. The first of these categories, magnetostrictive, is primarily used in high-power sonar and ultrasonic cleaning applications (Campbell, 1986), and is of limited utility from a mobile robotics perspective. Piezoelectric transducers (sometimes referred to as piezoceramic) are electrically similar to quartz crystals and resonant at only two frequencies: the resonant and antiresonant frequencies (Pletta, et al., 1992). Transmission is most efficient at the resonant frequency while optimum receiving sensitivity occurs at the antiresonant frequency (National, 1989). In bistatic (dual transducer) systems, the resonant frequency of the transmitting transducer is matched to the antiresonant frequency of the receiver for optimal performance.

Piezoelectric crystals change dimension under the influence of an external electrical potential and will begin to vibrate if the applied potential is made to oscillate at the crystal's resonant frequency. While the force generated can be significant, the displacement of the oscillations is typically very small, and so piezoelectric transducers tend to couple well to solids and liquids but rather poorly to low-density compressible

media such as air (Campbell, 1986). Fox, et al. (1983) report using a quarter-wavelength silicon-rubber matching layer on the front face of the transducer in an attempt to achieve better coupling into air at operating frequencies of 1 to 2 MHz. There is also a mechanical inertia associated with the vibrating piezoelectric crystal. As a consequence, such transducers will display some latency (typically several cycles) in reaching full power, and tend to "ring down" longer as well when the excitation voltage is removed.

Electrostatic transducers, on the other hand, generate small forces but have a fairly large displacement amplitude, and therefore couple more efficiently to a compressible medium such as air than do piezoelectric devices. The low-inertia foil membrane allows for quicker turn-on and turn-off in comparison to the slow response of piezoelectrics, facilitating unambiguous short-duration pulses for improved timing accuracy (Campbell, 1986). Since effective operation is not limited to a unique resonance frequency, electrostatic transducers are much more broadband, but with an upper limit of several hundred kilohertz in contrast to megahertz for the piezoelectric variety.

In addition to transducer design considerations, the performance of ultrasonic ranging systems is significantly affected by target characteristics (i.e., absorption, reflectivity, directivity) and environmental phenomena, as will be discussed below.

Atmospheric Attenuation. As an acoustical wave travels away from its source, the signal power decreases according to the inverse square law as illustrated in Figure 3, dropping 6 dB as the distance from the source is doubled (Ma & Ma, 1984). The following describes the effects of spherical divergence on signal intensity:

$$I = \frac{I_o}{4\pi R^2}$$

where:

I = intensity (power per unit area) at distance R

I_o = maximum (initial) intensity

R = range.

There is also an exponential loss associated with molecular absorption of sound energy by the medium itself (Pallas-Areny & Webster, 1992):

$$I = I_o\, e^{-2\alpha R}$$

where:

α = attenuation coefficient for medium.

The value of α varies slightly with the humidity and dust content of the air and is a function of the operating frequency as well (i.e., higher frequency transmissions attenuate at a faster rate). The maximum detection range for an ultrasonic sensor is thus dependent on both the emitted power and frequency of operation: the lower the frequency, the longer the range.

The maximum theoretical attenuation for ultrasonic energy can be approximated by (Shirley, 1989):

$$a_{max} = \frac{f}{100}$$

where:

a_{max} = maximum attenuation in dB/foot

f = operating frequency in KHz.

For a 20-KHz transmission, a typical absorption factor in air is approximately 0.02 dB/foot, while at 40 KHz losses run between 0.06 and 0.09 dB/foot (Ma & Ma, 1984).

Combining the above spherical-divergence and molecular-absorption attenuation factors results in the following governing equation for intensity as a function of distance R from the source:

$$I = \frac{I_o\, e^{-2\alpha R}}{4\pi R^2}$$

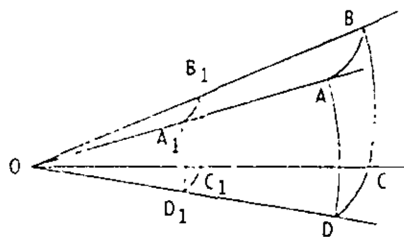


Figure 3: *Neglecting atmospheric attenuation, the total energy flowing within the cone OABCD is independent of the distance at which it is measured, whereas the intensity per unit area falls off with the square of R (adapted from Feynman, et al., 1963)*

Note that in this expression, which does not yet take into consideration any interaction with the target surface, intensity falls off with the square of the distance.

Target Reflectivity. The totality of all energy incident upon a target object is either reflected or absorbed, be it acoustical, optical, or RF in nature. The directivity of the target surface determines how much of the reflected energy is directed back towards the transducer. Since most objects scatter the signal in an isotropic fashion, the returning echo again dissipates in accordance with the inverse square law (Biber, et al., 1980), introducing an additional $4\pi R^2$ term in the denominator of the previous equation for intensity. In addition, a new factor K_r must be introduced in the numerator to account for reflectivity of the target:

$$I = \frac{K_r\, I_o\, e^{-2\alpha R}}{16\pi^2 R^4}$$

where:

K_r = coefficient of reflection.

This coefficient of reflection for a planar wave arriving normal to a planar object surface is given by (Pallas-Areny & Webster, 1992)

$$K_r = \frac{I_r}{I_i} = \left(\frac{Z_a - Z_o}{Z_a + Z_o}\right)^2$$

where:

I_r = reflected intensity

I_i = incident intensity

Z_a = acoustic impedance for air

Z_o = acoustic impedance for the target object

The bigger the impedance mismatch between the two media, the more energy will be reflected back to the source. In industrial applications, this phenomenon allows tank level measurement to be accomplished using an ultrasonic transducer in air looking down on the liquid surface, or alternatively an immersed transducer looking upward at the fluid/air interface.

Figure 4: *A small ball impacting a sawtooth surface as shown will generally bounce back towards its origin, whereas a ball much larger in diameter than the sawtooth dimension* d *will bounce away in specular fasion.*

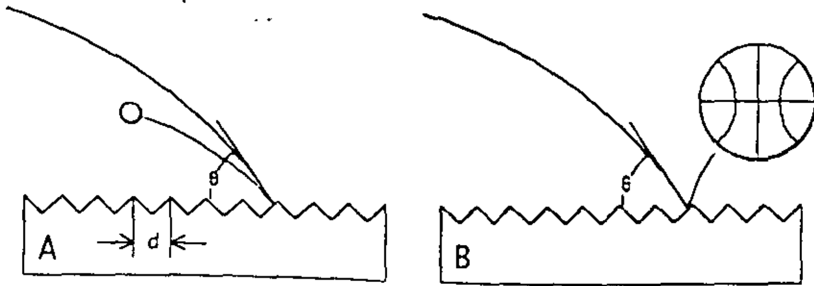
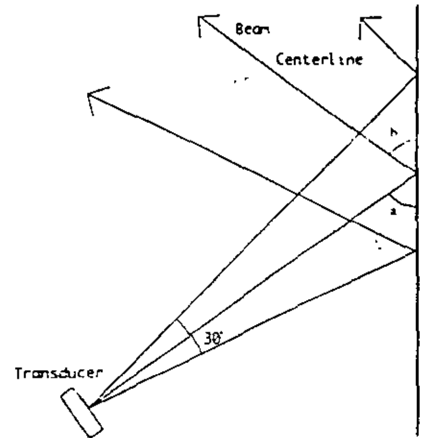


Figure 5: *As the angle of incidence decreases below a certain critical angle, reflected energy no longer returns to the transducer.*



Most targets are specular in nature with respect to the relatively long wavelength (roughly 0.25 inch at 50 KHz) of ultrasonic energy, as opposed to being diffuse. In the case of specular reflection, the angle of reflection is equal to the angle of incidence, whereas for diffuse reflection energy is scattered in various directions by surface irregularities equal to or larger than the wavelength of incident radiation. Lambertian surfaces are ideal diffuse reflectors that in theory scatter energy with equal probability in all directions (Jarvis, 1983).

To develop a more or less intuitive appreciation for this relationship to wavelength, it is perhaps helpful to consider the analogy of a pair of rubber balls impacting a hypothetical surface with the sawtooth profile shown in Figure 4. Assume one ball is approximately an inch in diameter, while the other is a much larger basketball. If the sawtooth dimension d is in the same neighborhood as the diameter of the smaller ball, then there is a good chance this ball when approaching the surface at some angle of incidence θ will bounce back towards its origin as shown in Figure 4A. This is because on the scale of the smaller ball, the surface has a significant normal component. On the other hand, when the basketball impacts this sawtooth surface with the same angle of incidence, the surface irregularities are too small with respect to the ball diameter to be effective. The basketball therefore deflects in a specular manner as shown in Figure 4B.

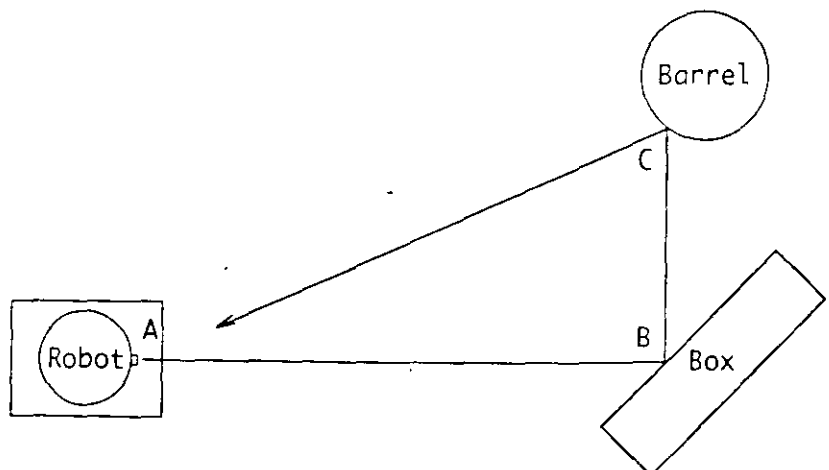
A familiar example of this effect at optical wavelengths can be seen when the beam of an ordinary flashlight is pointed towards a wall mirror at roughly a 45-degree angle. The footprint of illumination on the mirror surface is not visible, because all the light energy is deflected away in a specular fashion. In other words, you can't see the flashlight spot on the mirror itself. Now suppose the flashlight is redirected slightly towards the wall adjacent to the mirror. The spot of light shows up clearly on the wall surface, which is Lambertian in nature with respect to the wavelength of light. The wall is thus a diffuse reflector as opposed to a specular reflector for optical energy.

When the angle of incidence of a sonar beam decreases below a certain critical value, the reflected energy does not return to strike the transducer. The obvious reason for this effect is the normal component falls off as the angle of incidence becomes more shallow, as illustrated in Figure 5.

This critical angle is a function of the operating frequency chosen and the topographical characteristics of the target surface.

For the Polaroid electrostatic transducers this angle turns out to be approximately 65 degrees (i.e., 25 degrees off normal) for a flat target surface made up of unfinished plywood. Transducer offset from the normal will result in either a false echo as deflected energy returns to the detector over an elongated path, or no echo as the deflected beam dissipates. In Figure 5 above, the ranging system would not see the wall and indicate instead maximum range, whereas in Figure 6 the range reported would reflect the total round trip

Figure 6: *Due to specular reflection, the measured range would represent the round trip distance through points A, B, and C as opposed to the actual distance between A and B (adapted from Everett, 1985).*



through points A, B, and C as opposed to just A and B.

When the first prototype of the Mobile Detection Assessment Response System (MDARS) interior robot (Figure 7) was delivered to another government laboratory for formal Technical Feasibility Testing in early 1991 (Holland, et al., 1995; Laird, et al., 1995), the narrow-beam collision avoidance sonar array experienced significant problems in the form of false echo detections. These erroneous sonar readings were quickly seen to correlate with the presence of expansion joints in the concrete floor surface of the test facility. The transducers in the forward-looking array were purposely installed with a 7-degree down angle to increase the probability of detection for low-lying obstructions. This approach had worked very well in our building over months of extended operations, because the smooth floors were very specular targets with no significant discontinuities. An overnight field change realigning the sonar beams to a horizontal orientation was required to resolve the problem (Everett, et al., 1994).

Any significant absorption can result in a reduction of the reflected wave intensity with an adverse impact on system performance. For example, the Polaroid ultrasonic system has an advertised range of 35 feet. In testing the security module on the MDARS robot, however, we found it was difficult to pick up an average size person standing upright much beyond a distance of 23 feet. Harder targets of smaller cross-sectional area, on the other hand, could be seen out to the maximum limit.

The amount of energy coupled into the target surface (i.e., absorbed) versus that reflected is basically determined by the difference in acoustic impedance Z between the propagation medium (air) and the target object itself. Typical values for Z are listed in Table 1. Maximum transmission of energy occurs in the case of a fully homogeneous medium where Z is uniform throughout. For non-homogeneous situations involving an interface between two dissimilar media, effective coupling falls off (and, reflectivity subsequently goes up) as the differential in Z increases. The coefficient of transmission for a planar wave incident upon a planar target in a direction normal to the target surface is given by (Pallas-Areny & Webster, 1992):

$$K_t = \frac{I_t}{I_i} = \frac{4 Z_a Z_o}{\left(Z_a + Z_o\right)^2}$$

where:

K_t = coefficient of transmission (absorption)
I_t = transmitted intensity
I_i = incident intensity
Z_a = acoustic impedance for air
Z_o = acoustic impedance for the target object.

The original Polaroid ranging module transmitted a 1-millisecond chirp consisting of four discrete frequencies: 8 cycles at 60 KHz, 8 cycles at 56 KHz, 16 cycles at 52.5 KHz, and 24 cycles at 49.41 KHz (Biber, et al., 1980). This technique was employed to increase the probability of signal reflection from the target, since certain surface characteristics could theoretically absorb a single-frequency waveform, preventing detection. In actual practice such frequency-dependent effects rarely arose, suggesting this aspect of the absorption problem

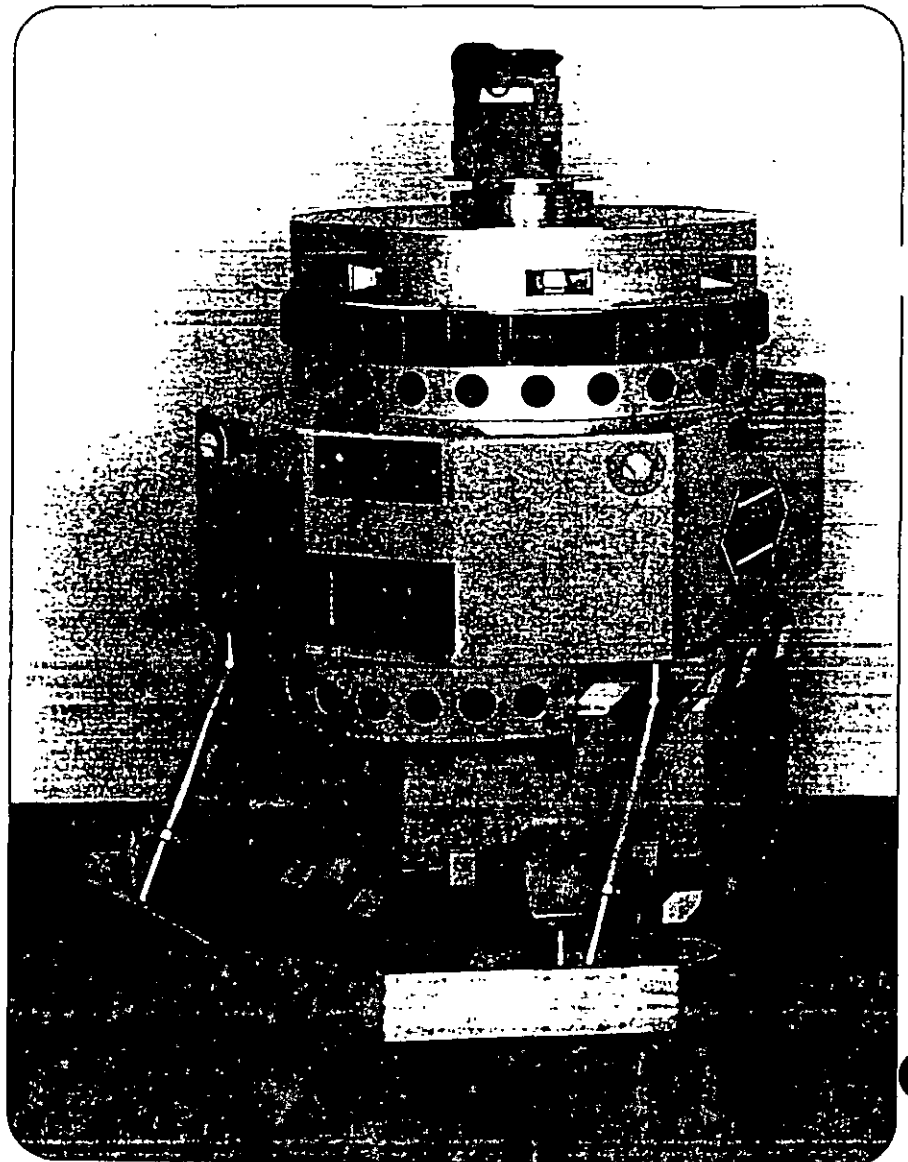


Figure 7: *The early feasibility prototype of the MDARS interior robot employed nine Polaroid electrostic sensors in a forward-looking collision avoidance array, as well as 24 additional sensors in a 360-degree array for ultrasonic motion detection (courtesy Naval Command Control and Ocean Surveillance Center)*

had been somewhat overestimated. In fact, Polaroid's improved version of the ranging module circuit board, the SN28827, operated at a single frequency of 49.1 KHz.

My daughter Rebecca compiled a significant amount of empirical data in 1993 as part of her high school science fair project entitled Determining the Accuracy of an Ultrasonic Ranging Sensor (Figure 8). One of her tests investigated the reflective properties of various target surfaces measuring 16 by 24 inches. The targets were maintained normal to a temperature-compensated Polaroid sensor (a Digitape ultrasonic tape measuring unit distributed by Houseworks) mounted 14 inches above a smooth concrete floor. Starting at a point beyond the maximum range of detection, the distance between the sensor and target was decreased in 1-foot increments until a valid range reading was obtained. The table (Table 2) is reproduced here with her permission.

Table 1: *Typical values of acoustical impedance (Z) for various conducting media (adapted with permission from Bolz & Tuve, 1979, © CRC Press, Boca Raton, FL; and Pallas-Areny & Webster, 1992).*

Medium	Z	Units
Air	4.3×10^{-4}	million Pascal-seconds/meter
Cork	1.0	million Pascal-seconds/meter
Water	1.5	million Pascal-seconds/meter
Human tissue	1.6	million Pascal-seconds/meter
Rubber	3.0	million Pascal-seconds/meter
Glass	13	million Pascal-seconds/meter
Aluminum	17	million Pascal-seconds/meter
Steel	45	million Pascal-seconds/meter
Gold	62.5	million Pascal-seconds/meter

Air Turbulence. Turbulence due to wind and temperature variations can cause bending or distortion of acoustical energy traveling through air (Shirley, 1989). Wind direction and velocity can have a noticeable push or delay effect on the wave propagation velocity, more relevant in the case of outdoor vehicles. Consideration of wind-effect errors must also treat crosswind components in addition to those which travel on a parallel path either with or against the wavefront. Crosswind effects can cause the beam center to be offset from its targeted direction, diminish the intensity of returned echoes, and result in a slightly longer beam path due to deflection.

In general, little effort is made in the case of mobile robotic applications to correct for such errors. This is probably due to the fact that ultrasonic ranging is most widely employed in indoor scenarios where the effects of air turbulence are minimal, unless extreme measurement accuracy is desired. In addition, there is really no practical way to reliably measure the phenomena responsible for the interference, and so compensation is generally limited to averaging over multiple readings. This approach introduces a coordinate transformation requirement in the case of a moving platform, since the slow speed of sound limits effective update rates to roughly 2 Hz (i.e., single transducer, assuming 28 feet maximum range). Faster updates are possible if the system is range-gated to some lesser distance (Gilbreath & Everett, 1988).

Temperature. Recall the earlier expression for wave propagation speed (s) in a gas, as a function of density ρ and bulk modulus of elasticity K_m:

$$s = \sqrt{\frac{K_m}{\rho}}$$

Since both of these parameters change with temperature, the speed of sound in air is also temperature dependent (Pallas-Areny & Webster, 1992), and in fact directly proportional to the square root of temperature in

Figure 8: *Rebecca Everett checked the accuracy of the Polaroid sensor in measuring the distance to a variety of surfaces for her science project*

Table 2: *Maximum detection ranges for standardized 16- by 24-inch cross-sections of various materials.*

Surface	Distance	Reading	Units
Plywood	24	24.2	feet
Towel	22	22.3	feet
Underside of rug	16	16.3	feet
Foam	13	13.3	feet
Pillow	9	9.4	feet
Blanket	8	8.4	feet
Top of rug	3	3.6	feet

degrees Rankine (Everett, 1985):

$$s = \sqrt{g k R T}$$

where:
s = speed of sound
g = gravitational constant
k = ratio of specific heats
R = gas constant
T = temperature in degrees Rankine (F + 460).

For temperature variations typically encountered in indoor robotic ranging applications, this dependence results in a significant effect even considering the short distances involved. A temperature change over the not unrealistic span of 60° to 90°F can produce a range error as large as 12 inches at a distance of 35 feet. Fortunately, this situation can be remedied through the use of a correction factor based upon the actual ambient temperature, available from an external sensor mounted on the robot. The formula is simply:

$$R_a = R_m \sqrt{\frac{T_a}{T_c}}$$

where:
R_a = actual range
R_m = measured range
T_a = actual temperature in degrees Rankine
T_c = calibration temperature in degrees Rankine

The possibility does still exist, however, for temperature gradients between the sensor and the target to introduce range errors, since the correction factor is based on the actual temperature in the immediate vicinity of the sensor only. As in the case of air turbulence, there is generally little recourse other than averaging multiple readings. (Some industrial applications provide a temperature-stabilized column of air using a small blower or fan.)

Beam Geometry. Still another factor to consider is the beamwidth of the selected transducer, defined as the angle between the points at which the sound power has been reduced to half (-3 dB) its peak value (Shirley, 1989). This formal definition does not always map directly into any useful parameter in real-world usage, however. What is generally of more concern can be better described as the effective beamwidth, or the beam geometry constraints within which objects are reliably detected. (Reliable detection, of course, is also very much dependent on the size and shape of the object.) The width of the beam is determined by the transducer diameter and the operating frequency. The higher the frequency of the emitted energy, the narrower and more directional the beam, and hence the greater the angular resolution. Recall, however, an increase in frequency causes a corresponding increase in signal attenuation in air and decreases the maximum range of the system.

The wavelength of acoustical energy is inversely proportional to frequency as shown below:

$$\lambda = \frac{s}{f}$$

where:
λ = wavelength
s = speed of sound
f = operating frequency.

The beam-dispersion angle is directly proportional to this transmission wavelength (Brown, 1985).

$$\theta = 1.22 \frac{\lambda}{d}$$

where:
θ = desired dispersion angle
λ = acoustic wavelength
d = transducer diameter

The above relationship can be intuitively visualized by considering the limiting case where d approaches zero. Such a hypothetical device would theoretically function as a point source, emitting energy of equal magnitude in all directions. As d is increased, the device can be considered a planar array of point sources clustered together in circular fashion. For this configuration, the emitted energy will be in phase and at maximum intensity only along a surface normal. Destructive interference from adjacent point sources causes the beam intensity to fall off rapidly to either side, up to some local minimum value as shown in Figure 9. Constructive interference then occurs past this minimum point, resulting in the presence of side lobes.

Shirley (1989) defines the spot diameter that is insonified by the ultrasonic beam (*i.e.*, footprint of the incident beam at the target surface) in terms of this beam-dispersion angle θ (Figure 10):

$$D = 2 R \tan \frac{\theta}{2}$$

where:
D = spot diameter
R = target range.

Best results are obtained when the beam centerline is maintained normal to the target surface. As the angle of incidence varies from the perpendicular, note the range actually being measured does not always correspond to that associated with the beam centerline (Figure 11). The beam is

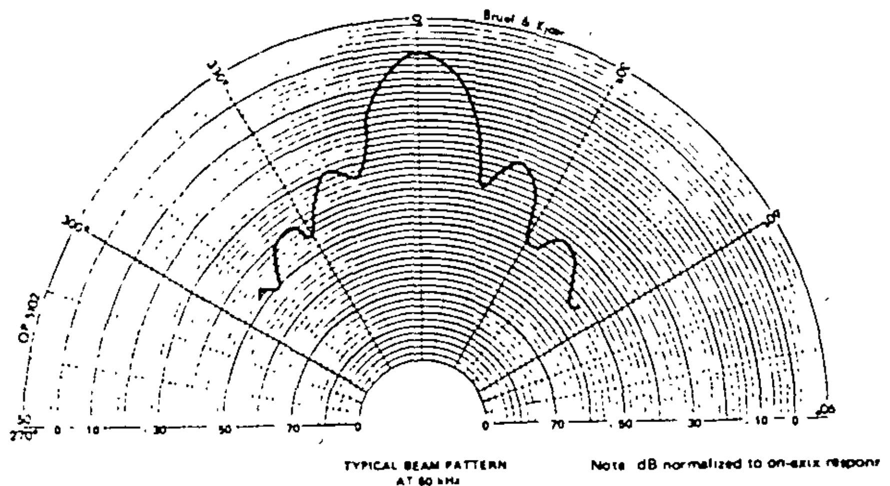


Figure 9: *Constructive interference results in maximum power in the main lobe along the beam center axis (courtesy of Polaroid Corp.)*

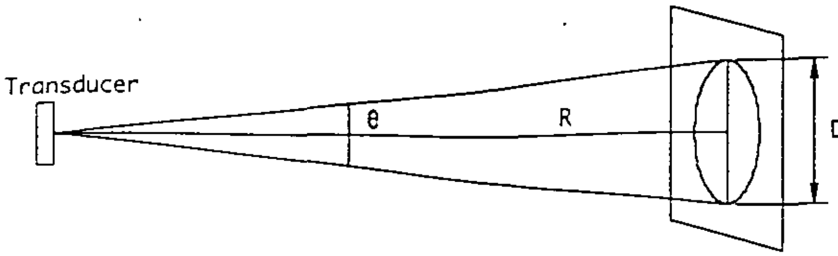


Figure 10: *The diameter of the insonified footprint at the target surface, assuming normal incidence.*

reflected first from the portion of the target closest to the sensor. For a 30-degree beam-dispersion angle at a distance of 15 feet from a flat target, with an angle of incidence of 70 degrees with respect to normal, the theoretical error could be as much as 10 inches. The actual line of measurement intersects the target surface at point B as opposed to point A.

Effective beamwidth introduces some uncertainty in the perceived distance to an object from the sensor but an even greater uncertainty in the angular resolution of the object's position. A very narrow target, such as a vertical pole, would have a relatively large associated region of floor space that would essentially appear to the sensor to be obstructed. Worse yet, a 3-foot doorway may not be discernible at all when only 6 feet away, simply because at that distance the beam is wider than the door opening.

Improved angular resolution can sometimes be obtained through beam splitting, a technique that involves the use of two or more transducers with partially overlapping beam patterns. Figure 12 shows how for the simplest case of two transducers, twice the angular resolution can be obtained along with a 50-percent increase in coverage area. If the target is detected by both sensors A and B, then it (or at least a portion of it) must lie in the region of overlap shown by the shaded area. If detected by A but not B, then it lies in the region at the top of the figure, and so on. Increasing the number of sensors with overlapping beam patterns decreases the size of the respective regions, and thus increases the angular resolution.

It should be noted, however, that this increase in angular resolution is limited to the case of a discrete target in relatively uncluttered surroundings, such as a metal pole supporting an overhead load or a lone box in the middle of the floor. No improvement is seen for the case of an opening smaller than an individual beamwidth, such as a doorway. The entire beam from at least one sensor must pass through the opening without striking either side in order for the opening to be detected, and the only way to improve resolution otherwise is to decrease the individual beamwidths by increasing the operating frequency, changing transducers, or through acoustical focusing. Some designs achieve this effect through use of an attachable horn that concentrates the energy into a tighter, more powerful beam (Shirley, 1989).

A number of factors must be considered when choosing the optimal beamwidth for a particular application. A narrow beamwidth will not detect unwanted objects to either side, is less susceptible to background noise, and can achieve greater ranges since the energy is more concentrated (Shirley, 1989). On the other hand, for collision avoidance applications it is often desirable to detect any and all objects in front of the robot, and since extremely long ranges are not usually required, a wide-angle transducer may be a more optimal choice (Hammond, 1993). When comparing a single transducer of each type, the use of a wide beamwidth will improve chances of target detection due to the greater likelihood of some portion of the beam encountering a surface normal condition as seen in Figure 13. Admittedly this observation is a bit like saying the wider the beam, the more chance of hitting a target. Taken to the extreme, a hypothetical 360-degree field-of-view transducer is clearly of rather limited utility due to the total lack of azimuthal information regarding the target's whereabouts.

Alternatively, an equivalent surface-normal condition can be realized using a cylindrical array of narrow-

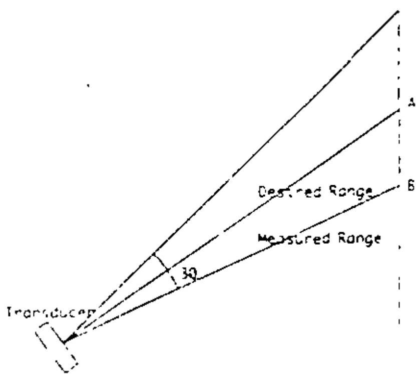


Figure 11: *Ultrasonic ranging error due to beam divergence results in a shorter range measurement to the target surface at B instead of the desired reading to point A.*

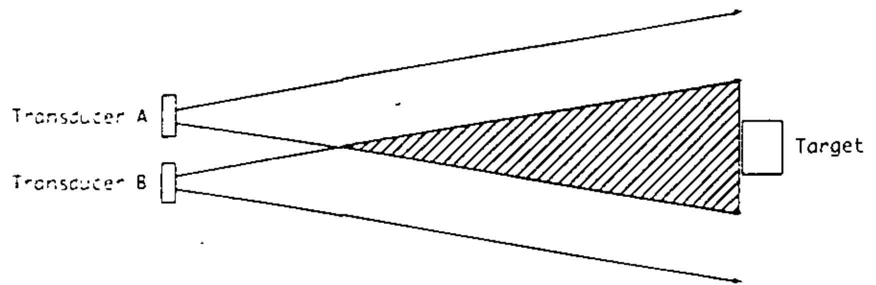


Figure 12: *Beam-splitting techniques using two or more sensors can improve angular resolution for discrete targets (adapted from Everett, 1985).*

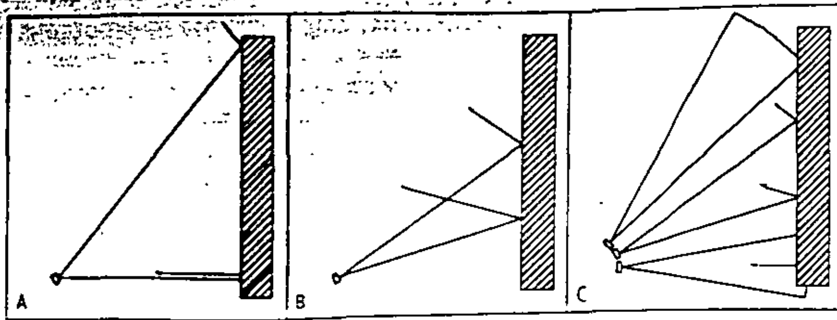


Figure 13: *A wide-angle transducer (A) has a greater chance of encountering a surface normal condition than a single narrow-beam transducer (B), but at the expense of reduced angular resolution and effective range. A more optimal configuration is presented in (C), at a slight cost to system update rate.*

beam transducers to achieve the same volumetric coverage as illustrated in Figure 13C. This approach offers the added advantage of significantly improved angular resolution but at the expense of a slower overall update rate. The MDARS Interior robot uses a combination of wide-angle piezo-electric sonars operating at a frequency of 75 KHz for timely obstacle detection coverage, and a nine-element array of narrow-beam Polaroid electrostatic transducers operating at 49.4 KHz to support intelligent obstacle avoidance. Detection of any potential obstructions by either type of sonar causes the platform to slow to a speed commensurate with the narrow-beam update rate, whereupon the high-resolution Polaroid data is used to formulate an appropriate avoidance maneuver.

Noise. Borenstein & Koren (1992) of the University of Michigan Mobile Robotics Lab define three types of noise affecting the performance of ultrasonic sensors

- *Environmental noise* resulting from the presence of external sources operating in the same space. Typical examples in industrial settings include high-pressure air blasts and harmonics from electrical arc welders.
- *Crosstalk* resulting from the proximity of other sensors in the group, which can be especially troublesome when operating in confined areas
- *Self noise* generated by the sensor itself

A noise-rejection measure for each of the components was developed and integrated into a single algorithm (Michigan, 1991), which was in turn combined with a fast sensor-firing algorithm. This software has been implemented and tested on a mobile platform that was able to traverse an obstacle course of densely packed 8-millimeter-diameter poles at a maximum velocity of 1 meter/second

System-Specific Anomalies. A final source of error to be considered stems from case-specific peculiarities associated with the actual hardware employed We shall again refer to the Polaroid system, in light of its widespread usage, as an illustrative example in the ensuing discussion.

Pulse Width. The 1-millisecond length of the original four-frequency Polaroid chirp was a potential source of range measurement error since sound travels roughly 1100 feet/second at sea level, which equates to about 13 inches/millisecond The uncertainty and hence error arose from not knowing which of the four frequencies making up the chirp actually returned to trigger the receiver, but timing the echo always began at the start of the chirp (Everett, 1985). For the initial application of automatic camera focusing, designers were less concerned about absolute accuracy than missing a target altogether due to surface absorption of the acoustical energy. The depth-of-field of the camera optics would compensate for any small range errors that might be introduced due to this chirp ambiguity.

Even with the more recent SN28827 ranging model operating at a single frequency of 49.1 KHz, the transmission pulse duration is 0.326 milliseconds, giving rise to a maximum theoretical error of approximately 1.7 inches. (This estimate takes into account round-trip distance, and assumes best-case echo detection after just three cycles of returned energy.) The new Polaroid Ultrasonic Ranging Developer's Kit allows for programmable pulse duration to alleviate this limitation in demanding applications (Polaroid, 1993).

Threshold Detection. The specific method for detection of the returned pulse can be a significant source of error in any TOF ranging system (Figueroa & Lamancusa, 1992). Kuc and Siegel (1987) point out that the intensity of a typical pulse transmission peaks in the second cycle (Figure 14), and so simple thresholding of the received signal can cause late detection of weak echoes. Leonard and Durrant-Whyte (1992) discuss further complications in the

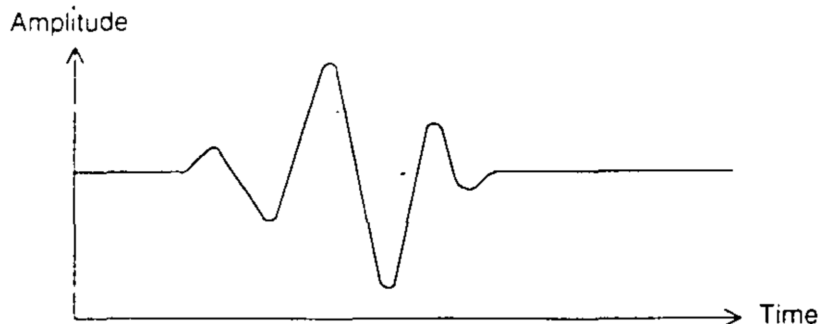


Figure 14: *A typical pulse waveform for an electrostatic transducer can be approximated by a sinusoid that is modulated by a Gaussian envelope, peaking in intensity during the second cycle (Kuc & Siegel, 1987, © IEEE)*

specific case of the integrating capacitive threshold detector employed in the Polaroid ranging module. This integrative approach was incorporated by the designers to discriminate against unwanted noise spikes (Biber, et al., 1980). Compared to strong reflections, valid but weak echo returns can take substantially longer to charge up the capacitor to the threshold level required for the comparator to change state.

The effect of this charging delay is to make those targets associated with weaker returns appear further away. Ignoring the obvious worst-case scenario of a completely missed echo, maximum theoretical error is bounded by the length of the transmitted burst. The obvious question now becomes, which is more preferable: missing target detection altogether, or being alerted to target presence at the expense of range accuracy? The answer of course depends on the particular priorities of the application addressed. If the ranging sensor is being employed as a presence detector for security purposes, precise accuracy is not all that important. On the other hand, if the sensor is being used for navigational referencing, the situation may be somewhat different.

Stepped Gain. Lang, et al. (1989) experimentally confirmed error effects associated with the piecewise 16-step gain ramp employed on the earlier Polaroid 607089 ranging module. In order to precisely counter the effects of signal loss as a function of range-to-target (*i.e.*, due to atmospheric attenuation and spherical divergence), the actual time-dependent gain compensation would be an exponential function inversely related to the spherical divergence equation presented earlier (see the Atmospheric Attenuation section). A rather coarse piecewise approximation to this ideal gain curve (Figure 15) naturally results in a situation where the instantaneous amplifier gain is: 1) correct only for a single point in time over the period represented by a specific step value, 2) excessive prior to this point, and 3) insufficient afterwards. If the gain is too low at the time of reflected pulse train arrival, weak echoes are [illegible] delayed in triggering the [illegible] detector, resulting in an erroneous increase in the perceived range.

Choosing an Operating Frequency. The operating frequency of an ultrasonic ranging system should be selected only after careful consideration of a number of factors, such as the diameter and type of transducer, anticipated target characteristics, sources of possible interference, and most importantly the nature of the intended task, to include desired angular and range resolution. Resolution is dependent on the bandwidth of the transmitted energy, and greater bandwidth can be achieved at higher frequencies but at the expense of maximum effective range. The minimum ranging distance is also a function of bandwidth, and thus higher frequencies are required in close as the distance between the detector and target decreases. Most man-made background noise sources have energy peaks below 50 KHz (Hammond, 1993), however, and so higher-frequency systems are generally preferred in acoustically noisy environments (Shirley, 1989).

About the Author

Commander H. R. (Bart) Everett, USN (Ret.), is the former Director of the Office of Robotics and Autonomous Systems at the Naval Sea Systems Command, Washington, [illegible] Naval C[illegible] Surveillance C[illegible]. Active in the field of [illegible] research for over 20 [illegible] personal involvement in the development of 11 mobile systems, he has published over 70 technical papers and reports and has 16 related patents issued or pending. He serves on the Editorial Board for *Robotics and Autonomous Systems* magazine and on the Board of Directors for the International Service Robot Association, and is a member of Sigma Xi, the Institute of Electrical and Electronics Engineers (IEEE), and the Association for Unmanned Vehicle Systems (AUVS).

References

Asher, R.C., "Ultrasonic Sensors in the Chemical and Process Industries," Journal of Physics E.: Scientific Instruments, Vol. 16, pp. 959-963, 1983.

Bartram, J.F., Ehrlich, S.L., Fredenberg, D.A., Heimann, J.H., Kuzneski, J.A., Skitzki, P., "Underwater Sound Systems," in *Electronic Engineer's Handbook*, D. Christiansen and D. Fink, eds., 3rd edition, McGraw Hill, New York, NY, pp. 25.95-25.133, 1989.

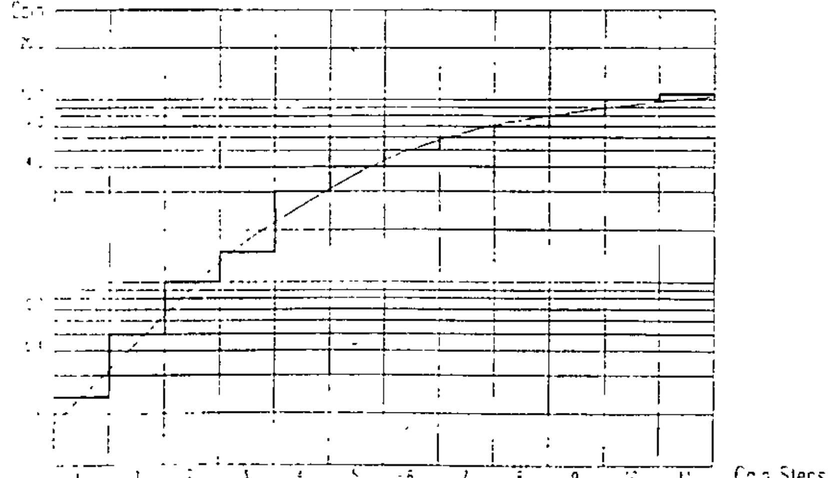


Figure 15: *The 12-step approximation employed in the new 6500-series receiver gain ramp results in a situation where the instantaneous gain is either above or below the ideal value for most of the step duration (adapted from Polaroid, 1993). Note the large jump in gain between steps 3 and 4.*

Biber, C., Ellin, S., Shenk, E., "The Polaroid Ultrasonic Ranging System," Audio Engineering Society, 67th Convention, New York, NY, October-November, 1980.

Bolz, R.E., Tuve, G.L., *CRC Handbook of Tables for Applied Engineering Science*, CRC Press, Boca Raton, FL, 1979.

Borenstein, J, Koren, Y., "Error Eliminating Rapid Ultrasonic Firing for Mobile Robot Obstacle Avoidance," IEEE International Conference on Robotics and Automation, Nice, France, May, 1992.

Brown, M.K., "Locating Object Surfaces with an Ultrasonic Range Sensor," IEEE Conference on Robotics and Automation, St. Louis, MO, pp.110-115, March, 1985.

Campbell, D., "Ultrasonic Noncontact Dimensional Measurement," *Sensors*, pp. 37-43, July, 1986.

Dunkin, W.M., "Ultrasonic Position Reference Systems for an Autonomous Sentry Robot and a Robot Manipulator Arm", Masters Thesis, Naval Postgraduate School, Monterey, CA, March, 1985.

Everett, H.R., "A Multielement Ultrasonic Ranging Array," *Robotics Age*, pp.13-20 July, 1985.

Everett, H.R., Gage, D.W., Gilbreath, G.A., Laird, R.T., Smurlo, R.P., "Real-world Issues in Warehouse Navigation," SPIE Mobile Robots IX, Vol. 2352, Boston, MA, November, 1994.

Feynman, R.P., Leighton, R.B., Sands, M., *The Feynman Lectures on Physics*, Vol. 1, Addison-Wesley, Reading, MA, 1963.

Figueroa, F., Barbieri, E., "Increased Measurement Range Via Frequency Division in Ultrasonic Phase Detection Methods," *Acustica*, Vol. 73, pp 47-49, 1991.

Figueroa, J.F., Lamancusa, J.S., "A Method for Accurate Detection of Time of Arrival: Analysis and Design of an Ultrasonic Ranging System," Journal of the Acoustical Society of America, Vol. 91, No. 1, pp. 486-494, January, 1992.

Figueroa, J.F., Mahajan, A., "A Robust Navigation System for Autonomous Vehicles Using Ultrasonics," *Control Engineering Practice*, Vol. 2, No. 1, pp 49-59, 1994.

Fox, J.D., Khuri-Yakub, B.T., Kino, G.S., "High-Frequency Acoustic Wave Measurements in Air," IEEE Ultrasonics Symposium, pp. 581-584, 1983.

Frederiksen, T.M., Howard, W.M., "A Single-Chip Monolithic Sonar System," *IEEE Journal of Solid State Circuits*, Vol. SC-9, No. 6, December, 1974.

Gilbreath, G.A., Everett, H.R., "Path Planning and Collision Avoidance for an Indoor Security Robot," SPIE Mobile Robots III, Cambridge, MA, pp 19-27, Novemeber, 1988.

Halliday, D., Resnick, R., *Fundamentals of Physics*, John Wiley, New York, NY, 1974.

Hammond, W., "Smart Collision Avoidance Sonar Surpasses Conventional Systems," *Industrial Vehicle Technology '93: Annual Review of Industrial Vehicle Design and Engineering*, UK and International Press, pp. 64-66, 1993.

Holland, J.M., Martin, A., Smurlo, R.P., Everett, H.R., "MDARS Interior Platform," Association of Unmanned Vehicle Systems, 22nd Annual Technical Symposium and Exhibition (AUVS '95), Washington, DC, July, 1995.

Irwin, C.T., Caughman, D.O., "Intelligent Robotic Integrated Ultrasonic System," Proceedings, Robots 9, Society of Manufacturing Engineers, Detroit, MI, Sect 19, pp. 38-47, June, 1985.

Jarvis, R.A., "A Laser Time-of-Flight Range Scanner for Robotic Vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol PAMI-5, No 5, pp 505-512, 1983.

Kuc, R., Siegel, M.W., "Physically Based Simulation Model for Acoustic Sensor Robot Navigation," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-9, No. 6, pp. 766-778, November, 1987.

Laird, R.T., Gilbreath, G.A., Heath-Pastore, T.A., Everett, H.R., Inderieden, R S., Grant, K., "MDARS Multiple Robot Host Architecture," Association of Unmanned Vehicle Systems, 22nd Annual Technical Symposium and Exhibition (AUVS '95), Washington, DC, July, 1995.

Lang, S., Korba, L., Wong, A., "Characterizing and Modeling a Sonar Ring," SPIE Mobile Robots IV, Philadelphia, PA, pp. 291-304, 1989

Leonard, J.J., Durrant-Whyte, H.F., *Directed Sonar Sensing for Mobile Robot Navigation*, Kluwer Academic Publishers, Boston, MA, 1992

Ma, Y L., Ma, C., "An Ultrasonic Scanner System Used on an Intelligent Robot," IEEE IECON '84, Tokyo, Japan, pp 745-748, October, 1984

Michigan, "Mobile Robotics Lab," Brochure, University of Michigan Mobile Robotics Lab, Ann Arbor, MI, 1991.

Milner, R., ""Measuring Speed and Distance with Doppler," *Sensors*, pp 42-44, October, 1990.

Mitome, H., Koda, T., Shibata, S., "Double Doppler Ranging System Using FM Ultrasound," *Ultrasonics*, pp 199-204, September, 1984

Pallas-Areny, R., Webster, J.G., "Ultrasonic Based Sensors," *Sensors*, pp 16-20, June, 1992.

Polaroid, "Polaroid Ultrasonic Ranging Developer's Kit," Publication No PXW6431 6/93, Polaroid Corporation, Cambridge, MA, June, 1993

Shirley, P.A., "An Introduction to Ultrasonic Sensing," *Sensors*, pp 10-17, November, 1989.

Shirley, P.A., "An Ultrasonic Echo-Ranging Sensor for Board Inspection and Selection," *Sensors*, June, 1991.

Smurlo, R.P., Everett, H.R., "Intelligent Sensor Fusion for a Mobile Security Robot," *Sensors*, pp. 18-28, June, 1993

Ulrich, R., *Principles of Underwater Sound for Engineers*, 1983.

TEMA 7:
VEHÍCULOS AUTOGUIADOS.

1 - INTRODUCCIÓN.

1.1 - ESTADO DEL ARTE.

Un Robot Móvil es una máquina autopropulsada y no tripulada, capaz de desplazarse libremente y bajo control con un cierto grado de autonomía en un entorno determinado.

Su precio oscila entre unos pocos cientos de miles de pesetas para pequeños robots educacionales a robots de más de diez millones de pesetas que realizan trabajos muy sofisticados, hablando siempre de robots existentes actualmente en el mercado. Lógicamente los precios de los robots utilizados, por ejemplo, en aplicaciones espaciales o submarinas quedan fuera de esta clasificación.

La definición anterior es muy amplia y engloba desde máquinas parcialmente telecontroladas hasta el más complejo vehículo que puede imaginarse en cualquier novela de ciencia ficción Hay sistemas en los que los desplazamientos están limitados a un movimiento lineal. Por ejemplo, equipos que puedan moverse sobre un carril o vehículos filoguiados. A estos sistemas no se les considera normalmente como vehículos móviles, por el hecho de que su libertad está excesivamente restringida Aunque más adelante hablaremos un poco de este tipo de robots, mientras no se diga lo contrario, a partir de ahora sólo consideraremos robots móviles aquellos que pueden desplazarse, al menos, en un espacio bidimensional.

El mecanismo empleado por un robot móvil para desplazarse depende, lógicamente, del medio en el que se mueve Los sistemas empleados en los robots navales y aeroespaciales no difieren en absoluto de los utilizados por los sistemas equivalentes tripulados (hélices, turbinas, ..) En los robots terrestres se usan también mecanismos tradicionales como ruedas u orugas, sin embargo en aplicaciones en las que se requiere que el robot tenga gran capacidad de maniobra estos mecanismos pueden ser insuficientes. Actualmente se está investigando en otros sistemas de locomoción. El principal esfuerzo de este campo se está realizando en Japón, donde se están desarrollando robots movidos mediante patas articuladas.

Una característica importante de un robot móvil es su grado de autonomía. La autonomía requerida al robot depende de la aplicación que se le vaya a asignar. En muchos casos no es necesario, ni siquiera recomendable, que el robot sea totalmente autónomo, y es preferible que esté parcialmente telecontrolado. Un ejemplo claro lo constituyen los robots diseñados para actuar en caso de catástrofes o para la desactivación de explosivos. En otros casos un alto grado de autonomia es necesario por motivos económicos (un operador permanentemente en contacto con el robot resulta

costoso), estratégicos (por ejemplo en robots militares las comunicaciones pueden ser interferidas) o técnicos. Entre otros motivos técnicos cabe destacar los casos en que las comunicaciones entre el robot y el operador sean casi imposibles, (por ejemplo robots submarinos sin cables de conexión con el exterior) o sufran grandes alteraciones (por ejemplo robots para la explotación de otros planetas, en los que cualquier comunicación sufriría retrasos considerables desde el envío a la recepción).

Para desplazarse, los robots móviles necesitan estar dotados de sensores que les permitan a ellos o al operador del telecontrol tener información sobre el entorno para poder localizar objetos o evitar obstáculos y para informarse sobre la situación del robot en cada instante. Para reconocer el entorno se usan cámaras de TV, sistemas de sonar, infrarrojos etc. Para conocer la situación absoluta se utilizan sistemas de balizamiento, brújulas, inclinómetros, altímetros, odómetros, acelerómetros, etc.

Aunque puede hacerse una división de los robots móviles según la función que realizan (básicamente de transporte y manipulación), la clasificación principal, en nuestra opinión, desde el punto de vista técnico, viene determinada por la tecnología de guiado en que se basa su diseño ,y que podemos resumir en cuatro grandes grupos que se detallan a continuación:

- Robots filoguiados.
- Robots autoguiados.
- Robots guiados por raíles.
- Robots guiados por visión.

De estos cuatro grupos los más utilizados actualmente con gran diferencia son los dos primeros. Los robots autoguiados mejoran en cuanto a versatilidad y flexibilidad a los anteriores, aunque su aplicación industrial es más compleja. Los robots guiados por raíles tienen prestaciones muy parecidas a los filoguiados, pero su coste es sensiblemente superior, razón por la que raramente se acude a esta solución.

En cuanto a los vehículos guiados por visión, se encuentran todavía en estado experimental.

1.2 - ROBOTS FILOGUIADOS.

El principio físico en que se basan los robots filoguiados es muy simple y robusto. Se trata de un cable enterrado que es excitado por un generador de corriente alterna (en el rango de 1KHz a 15KHz). Esta señal produce un campo magnético a lo largo de todo el cable, con una distribución de simetría radial entorno al mismo. Esta simetría es la que permite detectar la posición relativa del robot respecto al cable. Dos bobinas situadas simétricamente respecto al eje longitudinal del vehículo permiten captar la intensidad magnética generada por el cable, de modo que si el vehículo está perfectamente alineado con el cable, la diferencia de intensidad magnética entre ambas bobinas es cero. Cualquier diferencia no nula de intensidad (señal de error) generará por medio de un algoritmo de control una señal de actuación sobre los motores de dirección y guiado del robot.

13

Estos robots disponen de un computador de a bordo que posibilita el control local del vehiculo: cálculo de error de desviación respecto al cable de guiado; algoritmos de control que actúan sobre los motores de dirección del robot; comunicación bidireccional con el computador central o con otros vehículos; chequeo de su estado; detección de obstáculos... Su mayor inconveniente radica en el elevado coste de la instalación de cableado, lo que hace poco viable la remodelación del trazado de trayectorias, con lo que su utilización ofrece un grado de flexibilidad muy pequeño.

1.3 - ROBOTS AUTOGUIADOS.

A diferencia de los anteriores, los robots autoguiados no necesitan de ningún cable o dispositivo para su guiado. Normalmente utilizan el cálculo y control del número de giros de las ruedas. Mediante este control el robot sabe perfectamente la posición absoluta en la que se encuentra, así como sus velocidades lineal y angular, su orientación...

Son mucho más flexibles que los filoguiados, ya que pueden realizar cualquier desplazamiento en dos dimensiones (por lo menos), y quizás lo mas importante, para cambiar la trayectoria a seguir no hará falta la reinstalación de ningún cable. Bastará con indicarle la nueva trayectoria

Las características principales de este tipo de robots son las siguientes:

- Son impulsados por motores eléctricos alimentados por baterías.
- Disponen de un computador de abordo normalmente comunicado con un ordenador central para su control y obtención de información.
- La comunicación con este ordenador central se realiza mediante módem, vía radiofrecuencia (FM o frecuencias extendidas con inmunidad al ruido) o vía infrarrojos.
- Disponen de sensores, de ultrasonidos u ópticos para determinar donde se encuentran y que hay a su alrededor.

1.4 - ROBOTS GUIADOS POR RAÍLES.

Los robots guiados por railes siguen una trayectoria prefijada por la instalación de unos railes semejantes a los de los trenes Sus posibilidades y limitaciones son los mismos que las de los robots filoguiados, por lo que no entraremos mas en ellos. Tan sólo indicar una ventaja: este tipo de vehículos presenta un resultado óptimo a la hora de transportar grandes cargas. Y un inconveniente: el alto precio de la instalación de los railes

1.5 - ROBOTS GUIADOS POR VISIÓN.

En los vehículos de uso comercial la capacidad de visión es mínima (realmente es incorrecto hablar de visión), ya que se reduce a la detección de pintura reflectante sobre el suelo, normalmente por dos células fotoeléctricas y el control de la trayectoria es muy limitado.

Los robots realmente guiados por visión requieren una cámara de video. Se trata de sistemas muy sofisticados y actualmente se encuentran en fase de experimentación. La idea fundamental en la que se basan estos robots es la sustitución del cableado seguido en el caso de los robots filoguiados por señales de navegación que puedan ser reconocidas mediante visión.

El principal inconveniente de este tipo de guiado es la programación de los algoritmos de reconocimiento de formas para el tratamiento de imágenes. En un futuro se especula con que se puedan distinguir formas tridimensionales.

1.6 - APLICACIONES.

Como se ha visto en apartados anteriores, los robots móviles presentan una gran variedad de configuraciones y posibilidades.

En primer lugar existen dispositivos de esta clase capaces de moverse en los distintos medios físicos: gases, líquidos y sólidos. Entre los primeros está el grupo de los denominados robots espaciales, de los que ya se ha citado como característica, cuando trabajan fuera de la órbita terrestre, la dificultad de comunicación con el controlador y, en consecuencia, la necesidad de un alto grado de autonomía. Estos robots realizan tareas en el espacio tales como toma de muestras, mantenimiento, etc. De un uso mas común son los robots que operan sumergidos en líquidos. Las tareas para las que se utilizan son muy variadas, desde mantenimiento de objetos sumergidos, como barcos o plataformas petrolíferas, hasta extracciones de muestras de fondos marino o detección de grietas en grandes recipientes o redes de tuberías.

Los robots móviles terrestres constituyen los de mayor campo de aplicación. Las distintas áreas de actividad en las que se aplican configuran los distintos tipos existentes en la actualidad o en vías de desarrollo. Entre otros se pueden citar los industriales, de protección civil, los agrícolas, de servicio, etc., aparte de las aplicaciones de defensa.

Los robots móviles industriales se están desarrollando como una evolución de los sistemas de transporte basados en AGV (vehículos guiados automáticamente), en un intento de darles mayor flexibilidad y de evitar la obra de infraestructura externa que su instalación exige. Además de estas funciones de transporte pueden realizar otras más complejas, como alimentación de máquinas, montaje, mantenimiento de instalaciones, etc. Este tipo de robots requiere un alto grado de autonomía, ya que la disminución del coste de operario puede ser decisiva para optar por su implantación. Por otra parte,

evoluciona en ambientes conocidos y haciendo funciones en gran parte repetitivas, lo que permite que su estructura sensorial no tenga que ser excesivamente compleja.

Otro tipo de robot móvil terrestre es el de protección civil, dedicado a tareas tales como actuaciones en incendios, en ambientes tóxicos o radiactivos, manipulación de explosivos, etc. Las características de este tipo de robots son distintas de las citadas para los industriales. En primer lugar deberán acceder a recintos no uniformes, lo que exigirá dotarlos de medios mecánicos de movimientos muy sofisticados, tales como de desplazamiento en suelos no asfaltados, blindaje térmico, etc. Sin embargo el grado de autonomía en su control puede ser muy reducido ya que en principio su movimiento será teleoperado, al menos parcialmente. Se puede igualmente pensar en otros muchos campos de aplicación en los que los robots móviles pueden ser utilizados. En cada uno de ellos será necesario un diseño específico y dará lugar a un sistema distinto que quizás, durante bastantes años, no guarde relación con los destinados a otras aplicaciones de distinta índole.

1.7 - VENTAJAS E INCONVENIENTES DE LOS ROBOTS MÓVILES.

Para poder hablar de las ventajas o de los inconvenientes de los robots móviles habrá que centrarse en los robots de uso comercial, ya que no tiene sentido hablar, por ejemplo, de los inconvenientes de un robot experimental que ha sido enviado a un lejano planeta donde no puede llegar un ser humano.

Los robots móviles resultan muy baratos una vez se ha hecho el desembolso inicial (ordenadores para su control, ayudas para su navegación, equipo de mantenimiento, formación técnica del personal encargado del mismo, ...). Un robot móvil no necesita una persona que lo dirija, pero necesitará la instalación de ayudas especiales para la navegación a
lo largo de su ruta. Aunque un robot móvil sea fácil de manejar, un sistema CAD será indispensable para generar y estudiar la gran cantidad de datos geométricos sobre la trayectoria que debe seguir y sobre los obstáculos que se puede encontrar. La instalación de sensores para detectar obstáculos o colisiones lo pueden hacer muy seguro, pero el riesgo de un accidente inesperado está siempre presente cuando el robot trabaja rodeado de gente, especialmente si hay niños o ancianos. Por esta razón, la velocidad de desplazamiento del robot debe ser bastante inferior a la de las personas por razones de seguridad. Necesitará recargar sus baterías de forma periódica, luego no podrá estar trabajando 24 horas al dia Tiene limitaciones físicas que serían insignificantes para un ser humano (escaleras, puertas cerradas, ...). Sólo se pueden usar en recintos cerrados al tráfico (la posibilidad de un robot móvil circulando por nuestras calles de forma independiente está muy lejana todavía). Aunque pueden trabajar en medios peligrosos para las personas (radiactividad, medios tóxicos, desactivación de explosivos), las personas tienen una versatilidad y capacidad de reacción que las hacen insustituibles en algunos casos.

A la vista de esto quedan claras las grandes limitaciones de los robots móviles. Pero estas limitaciones vendrán impuestas por el uso que queramos hacer de ellos. Un robot móvil será un mal vigilante jurado en una empresa, pero puede ser un incansable transportista de materiales entre dos centros de mecanizado.

1.8 - ALGUNOS DATOS SOBRE LOS ROBOTS MÓVILES ACTUALES.

En la tabla siguiente aparecen algunos de los robots móviles o vehículos autoguiados más comunes, así como su fabricante y la utilidad para la que han sido diseñados. La tabla se ofrece a título orientativo, pues los continuos avances tecnológicos hacen que cada vez haya más fabricantes que desarrollen vehiculos de este tipo.

PRINCIPALES VENDEDORES DE ROBOTS MÓVILES

VENDEDOR	ROBOT	ESTADO ACTUAL	PROPÓSITO	APLICACIÓN
Apogee Robotics	Orbitor 750 AGV	Mercado	Transporte material	Industria
Bell & Howell	Mailmovil AGV	Mercado	Transporte correo	Oficinas
Caterpillar	Self Guided Lift Truck with Vision	Mercado	Intercambiador de palés	Industria
Cybermation	Kluge	Mercado	Inspección de edificios	Reactores nucleares y museos
Cybermotion	Spimaster	Mercado	Patrulla de seguridad	Oficinas, factorías
Cyberotics	Lab Rover	Mercado	Transporte material	Hospitales laboratorios
Cyberworks	CyberVac	Mercado	Aspirador	Industria
Cyberworks	CyberGuard ac	Mercado	Patrulla de seguridad	Industria
Eaton-Kenway	Kenway Eagle	Mercado	Manipulación de materiales	Industria
EPO Radamec	RP2 H	Mercado	Grúa cámara TV	Estudios TV
IS Robotics	Attila-II	Educacional	Educación y desarrollo	Univ. centros de investigación
Kent	RoboKent	Mercado	Aspirador	Oficinas supermercados

Odetics	Odex-III	Bajo pedido	Centrales Nucleares	Comisión de Energía Atómica de Francia
Odetics	IRMA	Mercado	Mapa de radiaciones	Reactores Nucleares
Panasonic	Robot Aspirador	Experimental	Aspirador	Hogar
Samsung	Scout-About	Experimental	Patrulla de seguridad	Hogar
Sandia Nacional Laboratories	Fire Ant	Mercado	Vehículo Armado	Campo de batalla
Transitions Research	ScrubMate	Experimental	Limpieza aseos	Hoteles oficinas
Veeco instruments	AGV with robotic manipulator	Bajo pedido	Manipulación y transporte de circuitos integrados	Laboratorios de fabricación de C.I

PRINCIPALES MECANISMOS DE LOCOMOCIÓN

Mecanismo	Suelo	Escaleras	Escombros
Ruedas	SI	NO	NO
Ruedas en trébol	SI	SI	SI
Orugas	SI	SI	SI
Brazos	SI	SI	SI

SENSORES PARA NAVEGACIÓN

Tipo de sensor	Descripción	Función
Bumper	Switch	Detección de colisiones
Guia	Detectores de campos magnéticos, luz ultravioleta, y células fotoeléctricas	Seguimiento de marcas ubicadas en el suelo
Iman	Efecto Hall	Localizar marcas
Reflector	Luz y células fotoeléctricas	Localizar marcas
Código de barras	Scanner y cel. fotoeléctr	Localizar e identificar marcas
Baliza	Cel. fotoelec.	Localizar e identificar marcas
Proximidad	Luz y células fotoeléctricas	Detección de obstáculos Cálculo de distancias
Ultrasonidos	sensores de ultrasonidos	Calculo de distancias a los obstáculos

Heading	Giroscopio inercial o compás de rumbo.	Cálculo de orientación y dirección del movimiento
Odómetro	Rotación de las ruedas, ángulo de dirección	Cálculo de orientación y dirección del movimiento
GPS	GPS, computador de navegación	Indica la posición del robot en cualquier lugar de la tierra con un error de 100m. (10 m los militares)

SENSORES PARA PERCEPCIÓN DE OBJETOS EN EL ESPACIO DE TRABAJO

Tipo de sensor	Descripción	Función
Sonar-Vision	Múltiples sensores de ultrasonidos	Detección, localización e identificación de objetos
Light-stripe vision	Proyector de luz, cámara TV	Detección, localización de objetos
2-D vision	Cámara TV	Detección, localización e identificación de objetos
3-D vision	Cámara TV, Scanner láser, scanner de luz, . .	Detección, localización e identificación de objetos
Tacto	Palpadores	Detección, localización e identificación de objetos

APLICACIONES DE LOS ROBOS MÓVILES.

AGRICULTURA: Arado, plantado, fertilización,...; mantenimiento de campos de golf; control de invernaderos, control de plagas.

LIMPIEZA: Servicios limpieza de hoteles, oficinas, hospitales. Limpieza de ventanas de edificios comerciales. Limpieza y pintado de cascos de embarcaciones. Limpieza el hogar (mini- y microrrobots).

CONSTRUCCIÓN: Transporte de materiales dentro de la zona en construcción, pintado y enlucido de paredes.

RECURSOS DE LA TIERRA: Investigación geológica y recogida de muestras.

ENTRETENIMIENTO: Animación de dinosaurios para museos Parques de atracciones.

TAREAS PELIGROSAS: Regeneración de bases militares contaminadas en Estados Unidos y Europa. Manipulación de residuos químicos o radiactivos.

MANTENIMIENTO: Inspección mediante ultrasonidos de los componentes de aviones (alas, fuselaje,...). Inspección de tanques de gas liquido. Inspección, reparación y pintado de puentes ante la corrosión.

MANIPULACIÓN DE MATERIALES: Carga y descarga de contenedores, barcos, aviones,...

EJÉRCITO: Infiltración y reconocimiento. Detección de minas. Reparación y repostaje de tanques. Evacuación de heridos. Ayuda frente a guerra química o biológica...

REHABILITACIÓN: Sillas de ruedas móviles. Sillas de ruedas con brazos de robot para cuadrapléjicos...

2 - DESCRIPCIÓN DEL EQUIPO.

Para el desarrollo de la aplicación se ha utilizado el vehículo, una tarjeta de visión para PC y un joystick convencional. A continuación se describen someramente el vehículo autoguiado y la tarjeta de visión.

2.1 - DESCRIPCIÓN DEL VEHÍCULO AUTOGUIADO.

El vehiculo autoguiado Robuter II es un producto fabricado por la empresa francesa Robosoft S.A. El vehiculo tiene un peso aproximado de 150 Kg. y la capacidad de carga es de 120 Kg. Consta de una plataforma con unas dimensiones en milímetros de 1.025x680x440 desplazada mediante dos motores de corriente continua de 300W cada uno. El sistema está controlado por una CPU basada en un microprocesador Motorola 68020 a 16 MHz y arquitectura de bus VME. El sistema operativo de esta CPU recibe el nombre de ALBATROS. Se trata de

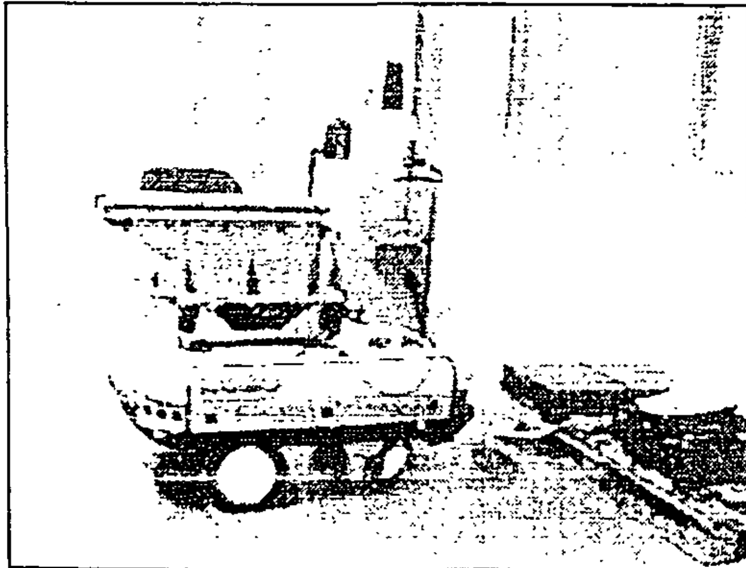


Ilustración 1.- Vista general del vehículo autoguiado Robuter II.

50

un sistema operativo diseñado específicamente para el control en tiempo real de sistemas robotizados de ejes múltiples. Gracias a este sistema operativo el control del vehículo se realiza de una forma mucho más cómoda, ya que se lanzan órdenes de alto nivel y la CPU del Robuter resuelve el problema cinemático directo y actúa en consecuencia para cumplir las órdenes recibidas. Las órdenes son recibidas por la CPU mediante un módem de infrarrojos que se conecta al computador via RS-232.

Como herramientas más importantes, el Robuter II cuenta con un sistema de detección de obstáculos basado en sensores de ultrasonidos, un sistema de seguridad que detiene el vehículo en caso de colisión, un sistema de visión y un sistema de parada de emergencia.

2.2 -DESCRIPCIÓN DE LA TARJETA DE VISIÓN.

Las imágenes captadas por la cámara de visión instalada a bordo del Robuter son transmitidas al dispositivo receptor vía modems de radiofrecuencia, independientes de los modems de infrarrojos utilizados en el control del propio vehículo. De este modo se consigue que la visualización de las imágenes se produzca en tiempo real, condición indispensable para poder realizar la teleoperación, al independizar la transmisión de las señal de video de la transmisión de las órdenes de control. Para la visualización de las imágenes se dispone de una tarjeta integradora de imágenes para PC. Esta tarjeta permite abrir una ventana en el monitor de un PC y utilizar dicha ventana como monitor de video de la señal recibida, a la vez que permite el almacenado en disco de dichas imágenes para un posterior tratamiento de las mismas. La tarjeta, denominada Screen Machine II, permite conectar cualquier señal de vídeo a una de sus entradas.

2.3 - COMUNICACIÓN CON EL VEHÍCULO.

La comunicación con el vehículo se realiza vía transmisión serie mediante modems de infrarrojos. Según las experiencias realizadas, los modems de infrarrojos hacen que el proceso de comunicación sea más lento que con modems de radiofrecuencia pero a cambio el índice de errores en la transmisión es mucho menor. La transmisión se realiza a una velocidad de 9600 baudios, sin paridad y con dos bits de stop. Los modems se rigen por el protocolo de comunicaciones request-to-send clear-to-send.

51

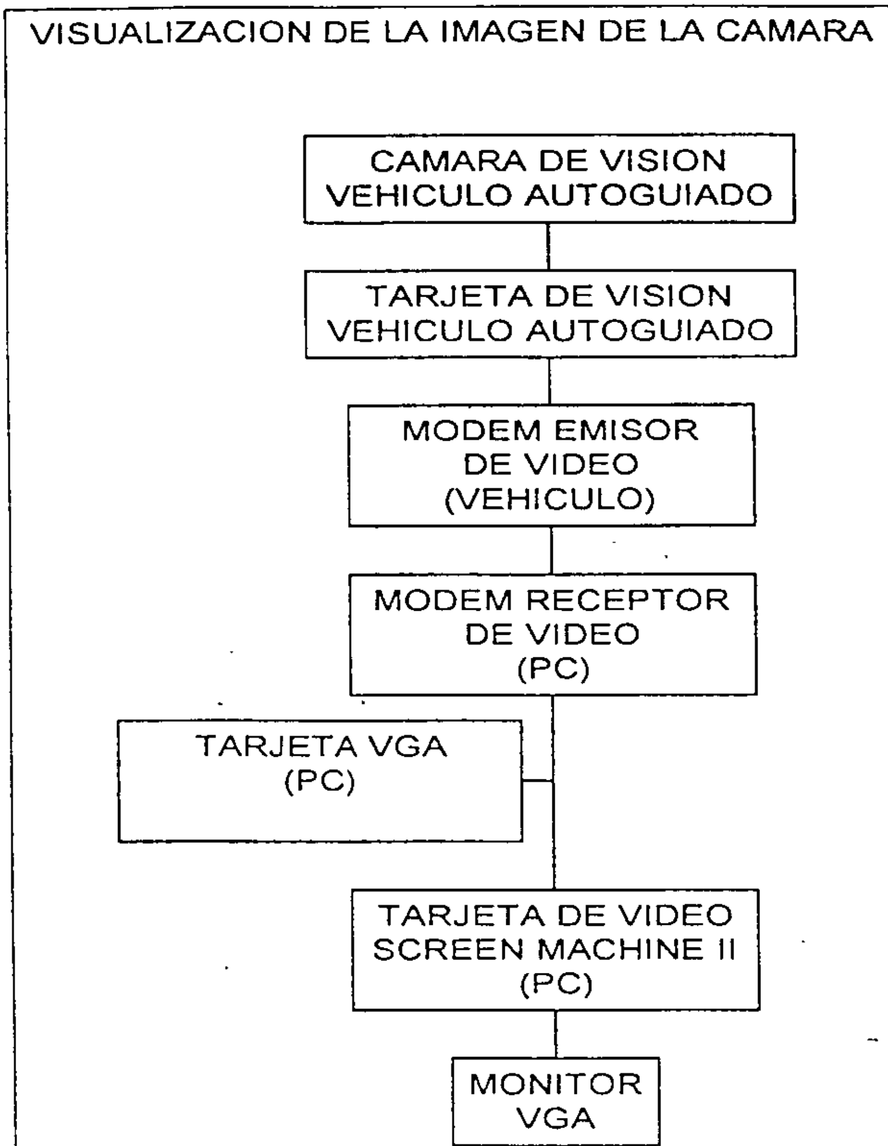


Ilustración 2.- Visualización de la imagen captada por la cámara del vehículo.

Pero lo más característico de la comunicación es el protocolo que siguen los modems para aceptar un carácter como bueno.

A continuación se detallan los pasos seguidos durante el proceso de comunicación:

1) Se escribe un carácter desde el computador en el módem
2) El módem emite el carácter.
3) El otro módem recibe el carácter, lo almacena en memoria y lo reemite
4) El módem emisor recibe el eco y comprueba si coincide con el carácter que envío. Si coincide lo da por bueno y lo vuelve a emitir.

5) El módem receptor vuelve a recibir el carácter y comprueba que es el mismo que recibió antes.
6) El módem pasa el carácter recibido a la CPU

Si en alguno de los pasos no hay coincidencia se establece un error de comunicación y vuelve a empezar el proceso otra vez desde el principio. Esto introduce retardos en la transmisión, por lo que hay que procurar minimizar la transmisión de datos para no formar un cuello de botella.

3 - INTERFAZ DE USUARIO.

El interfaz de usuario consiste en una sola pantalla compuesta de varios elementos con distinto significado. Principalmente podemos distinguir ventanas gráficas, pulsadores y conmutadores.

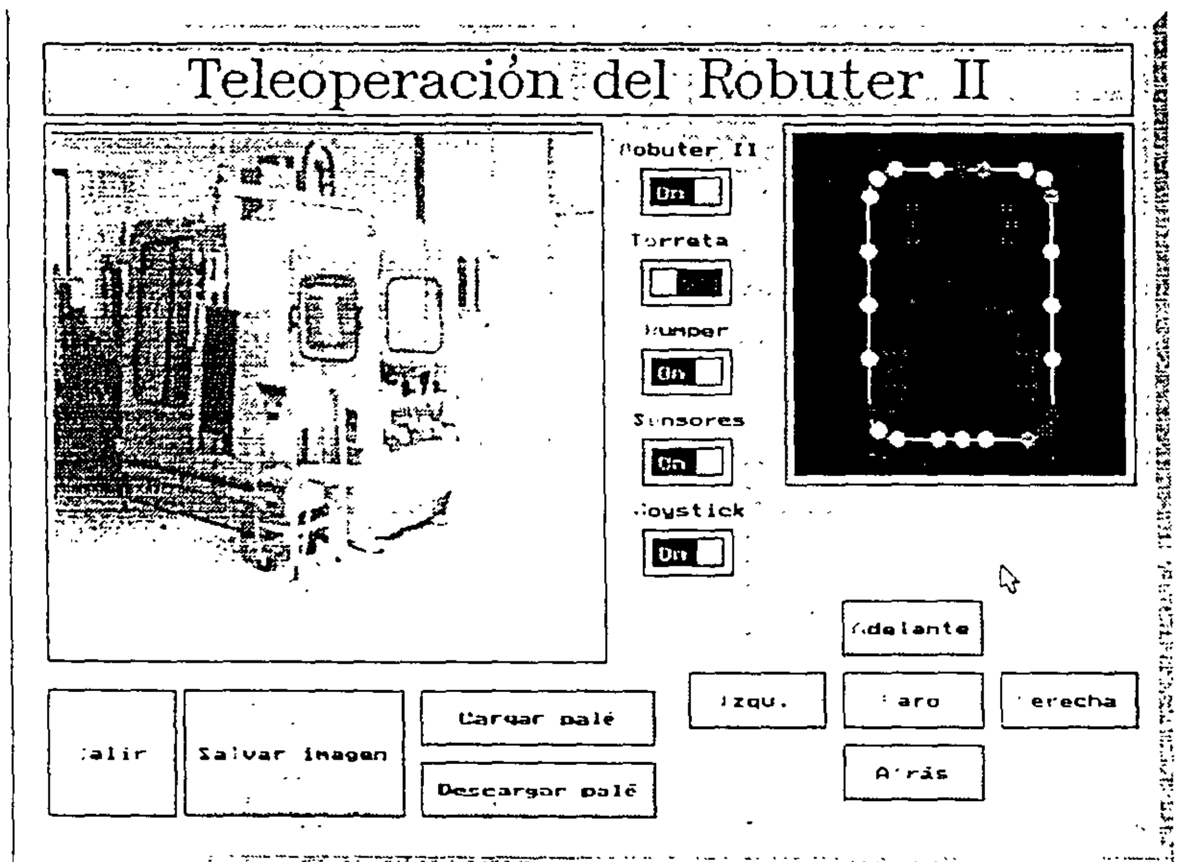


Ilustración 3.- Entorno del usuario al iniciar la aplicación.

53

Cabe distinguir entre dos ventanas gráficas. En la parte superior izquierda de la pantalla hay una ventana gráfica en la que se puede ver la imagen captada por la cámara de visión. Ésta es la ventana principal, puesto que a través de ella el operador establece contacto con el medio en el que se encuentra el vehículo.

En la parte superior derecha de la pantalla aparece otra ventana gráfica en la que podemos apreciar el estado de los sensores del vehículo. Sobre la figura del Robuter aparecen veinticuatro esferas de colores que representan los sensores de ultrasonidos del vehículo, cada uno en su posición. El sensor que aparece sobre el vehículo y no en el perfil representa el sensor de ultrasonidos de la torreta de la cámara de visión.

Cada sensor puede aparecer en los colores azul, verde, amarillo o rojo. El color en que aparezca dibujado el sensor indica la lectura que se realiza del mismo. Los colores a)azul, b)verde, c)amarillo o d)rojo significan:

a) desactivado,
b) que no hay ningún obstáculo que entorpezca el paso del vehículo,
c) vía libre pero con obstáculo próximo, por lo que el vehículo deberá desplazarse a baja velocidad ($1m<d<4m$), y
d) riesgo de colisión ($d<1m$).

Existen nueve pulsadores con las siguientes funciones:

- Pulsador "Salir" sirve para salir del programa y finalizar la sesión
- Pulsador salvar imagen sirve para archivar la imagen captada por la cámara a un fichero en disco (directorio por defecto "films").
- Pulsadores cargar palé y descargar palé.
- Los cinco pulsadores que aparecen en la parte inferior izquierda de la pantalla sirven para desplazar el vehiculo o bien la torreta de visión, según el comando de movimiento que haya sido activado. Si no hay activado ningún comando de movimiento, estos pulsadores se inhabilitan y se activan las funciones del portapalés.

Los conmutadores que aparecen en la parte superior central de la pantalla sirven para habilitar los comandos de movimiento y los dispositivos de seguridad, con las siguientes funciones:

- Conmutador "Robuter II" habilita el movimiento del vehículo.
- Conmutador "Torreta" habilita el movimiento de la torreta de visión.
- Conmutador "Bumper" gestiona la activación de los bumpers.
- Conmutador "Sensores" gestiona la activación de los sensores de ultrasonidos
- Conmutador "Joystick" habilita el dispositivo de maniobra.

Cabe resaltar que para trayectorias en espacios muy reducidos, la lectura de los sensores introduce un tiempo de retardo, por lo que en maniobras de aparcamiento se aconseja desactivar los sensores y dejar activado el bumper, para tener ,mayor control sobre el vehículo.

Por último, hay una función que sólo se gestiona por teclado y no aparece en el menú. Esta función lo que hace es devolver la torreta de visión a la posición de arranque y se ejecuta al pulsar la tecla W estando activado el conmutador de la torreta. Si pulsamos esta tecla en otras condiciones, no sucede nada.

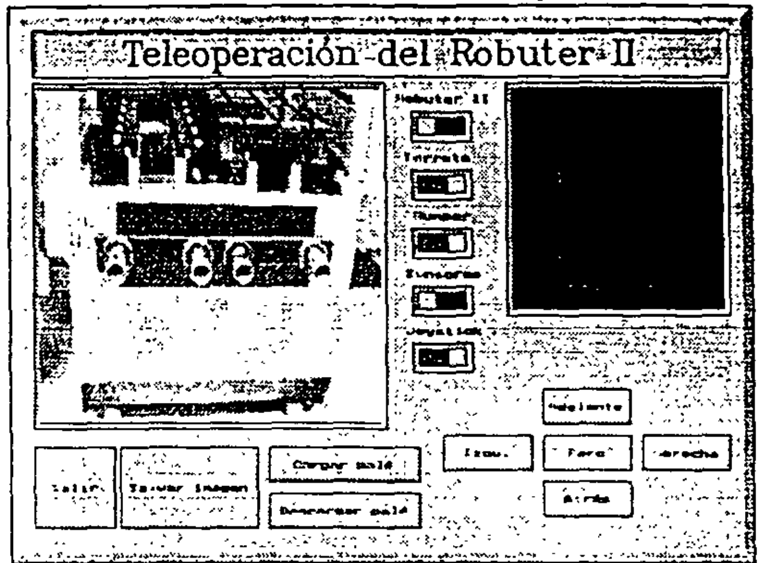


Ilustración 4.- Estado del panel al mover la torreta.

En la ilustración número cuatro se describe la posición de los conmutadores para la maniobra de la torreta de visión.

4.- CARGA Y DESCARGA DE PALÉS

El vehiculo tiene un elemento adicional que la aplicación también se encarga de gestionar. Se trata de un intercambiador de palés para una máquina herramienta. El arrastre del palé se realiza mediante un motor de corriente continua alimentado por las propias baterias del Robuter. La gestión del sistema se realiza mediante un sistema combinacional basado en relés, que utiliza sensores de presencia para detectar la posición del palé y el enclavamiento del vehiculo en la máquina herramienta. El dispositivo de control del vehiculo se basa en seis entradas lógicas que actúan sobre el sistema combinacional. De estas seis entradas, tres corresponden a los sensores de

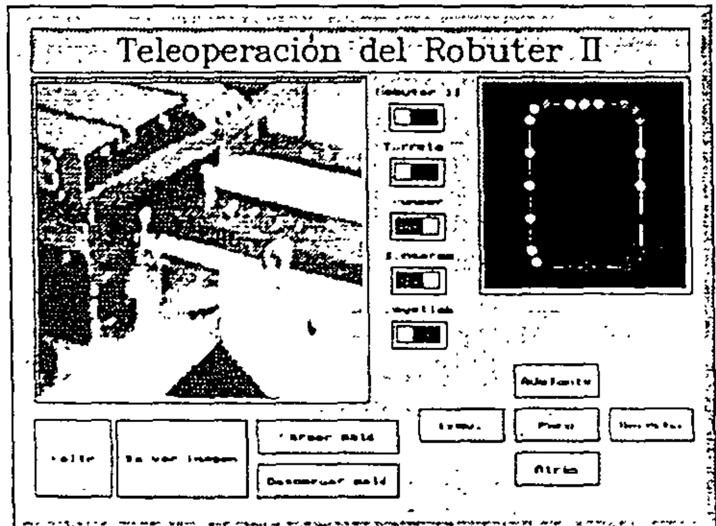


Ilustración 5.- Carga de un palé.

presencia que se encuentran alojados en el portapalés y ya vienen conectados de fábrica. Estos tres sensores detectan el enclavamiento del vehículo a la máquina herramienta y la posición del portapalés. La información que se recibe de estos sensores también se encuentra disponible en tres salidas lógicas por si el usuario desea utilizar dicha información para su programa de gestión. Dichas salidas se conectan a las entradas optoacopladas del vehículo y de tal modo que accediendo a dichas entradas en la CPU del Robuter podemos conocer el estado en que se encuentra el portapalés. Cabe destacar que el sistema sólo actúa cuando el sensor de enclavamiento detecta que el vehículo ha sido anclado en la máquina herramienta. Este modo de funcionamiento es muy seguro, ya que de este modo se evitan posibles incidentes debidos a la mala manipulación del vehículo. En caso de que un palé cayera al suelo, se podrían producir daños graves, con serio peligro para las personas de las inmediaciones. De este modo, se evitan todos estos estados de peligro.

Las funciones de carga y descarga de palés permanecen inhabilitadas hasta que el vehículo se encuentra acoplado a la máquina herramienta en la posición de carga/descarga y que los comandos de movimiento se encuentren desactivados. Un sensor de enclavamiento del portapalés detectará la posición adecuada del vehículo y cerrará el contacto que permite el paso de la corriente al motor del portapalés.

La razón de tener que desactivar los comandos de movimiento del vehículo responde a dos razones principales: evitar que accidentalmente se proceda a desplazar el vehículo durante el proceso de carga o descarga del palé y ofrecer una mayor fijación al suelo.

ARQUITECTURA JERÁRQUICA BASE PARA VEHÍCULOS AUTOGUIADOS (AGV).

Juan Miguel Martínez Rubio
Junio 1995.

DEFINICIÓN.

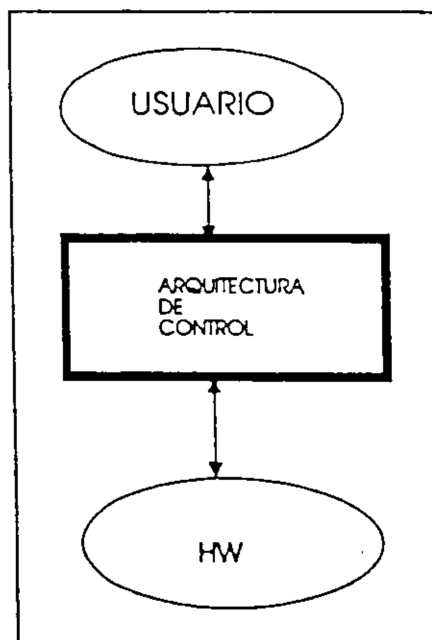
Modo de estructurar el sistema de funcionamiento de un vehículo autoguiado (AGV), en diferentes niveles o capas interconectadas, con el objetivo de conseguir total autonomía y mayor eficiencia.

PUNTO DE PARTIDA.

Para poder desarrollar la arquitectura será necesario:

* Vehículo móvil. *Ej: ROBUTER-II del DISCA.*
  Dotado con sistema de movimiento, percepción, computador de abordo, etc..
* Sistema Operativo básico para comandar el vehículo. *Ej: ALBATROS*
* Computador de control externo para comunicarse con el usuario.

Supuesto en el nivel superior el Usuario y en el inferior el hardware que maneja el vehículo. La arquitectura de control deberá comunicar ambos al objeto de gestionar el movimiento del AGV a partir de los comandos de usuario.

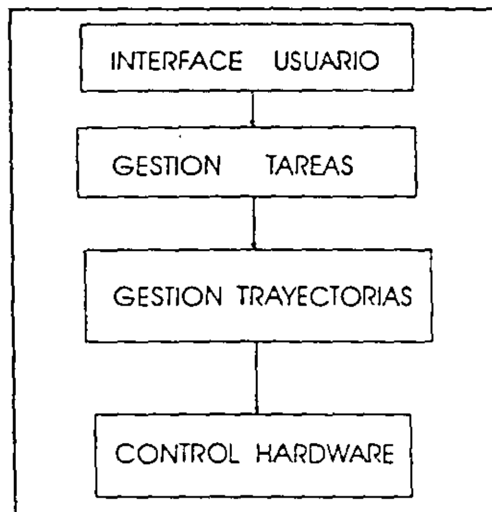


ARQUITECTURA JERÁRQUICA BASE: NIVELES NECESARIOS.

La arquitectura jerárquica realiza una descomposición para el control del AGV en capas o niveles. En su modelo más básico se puede abordar en cuatro capas:

INTERFACE CON EL USUARIO
NIVEL DE GESTIÓN DE TAREAS
NIVEL DE GESTIÓN DE TRAYECTORIAS
NIVEL DE CONTROL DEL HARDWARE

La figura muestra lo que sería los distintos niveles desde un punto de vista de diseño descendente (*top-down*). Se trata de un modelo jerárquico puesto que cada nivel sólo se interconecta con sus inmediatos vecinos y no con los demás.



Seguidamente se describe cada uno de estos Niveles incluyendo su implementación.

Interface con el usuario.

Corresponde con el nivel más alto de la jerarquía y por lo tanto es el que será accesible por el usuario. Debe proporcionar un entorno de trabajo cómodo y agradable al usuario, con el fin de que éste pueda manipular y gobernar el vehículo lo más sencillo posible.

Al usuario se le ofrecerán todas las funciones que el AGV puede realizar, por ejemplo bajo un entorno WINDOWS:

- ON/OFF
- Definición del entorno.
- Definición de la misión.
- Operación manual.
- Ejecución de una misión.
- Identificación del estado del AGV.
- Planificación de una misión.
- Parada de emergencia.
- Etc.

El desarrollo de este nivel debe ser suficiente modular para poder ir incorporando nuevas funciones en la medida de que éstas se necesiten.

Puede observarse que alguna de estas funciones se pueden resolver directamente en este nivel (Ej: *Definición/Modificación del entorno; Visualización del entorno actual; Definición de trayectorias por defecto; etc*).

Sin embargo otras operaciones sí que involucrarán otros niveles, por lo que se deberá comunicar al nivel inferior (Ej: *Inicialización del vehículo; generación manual de trayectorias; planificación de una misión; on/off del agv; ejecución de una misión; etc*)

Entrada en este nivel:

Operación escogida por el usuario en función del menú ofrecido en la pantalla con todas las opciones posibles. En definitiva la entrada que se va a obtener en este nivel es el propio usuario que nos identificará la opción que desea del vehículo.

Salida de este nivel:

La salida que se obtendrá es variable en función de la opción escogida. En el caso más simple, cuando se resuelve completamente en este nivel, únicamente alterará la *base de datos del entorno*, llamada "*modelado del mundo*".

Por modelado del mundo se entiende una estructura de datos que permita conocer a priori cuál es el entorno por el que se moverá el vehículo, su implementación se realizará mediante matrices en las que se puede almacenar información binaria. Por ejemplo "1" para marcar obstáculo y "0" para indicar la no existencia, posteriormente se pueden incluir valores intermedios si se considera necesario. Por ejemplo, la idea para una matriz E que represente un espacio de una habitación "i":

$$E_i = \begin{matrix} 0000000000011 \\ 0011000000011 \\ 0011000000011 \\ 0001100000000 \\ 0000000000000 \\ 0000000000000 \\ 0011000000000 \\ 0011000000011 \end{matrix}$$

Esta base de datos se actualizará dinámicamente de forma que si se detectan nuevos objetos pueden ser incluidos como nuevos. Pueden existir dos mapas implementados ambos mediante matrices. Uno que almacene información de los objetos fijos y otro que almacene información de los nuevos objetos que se vayan detectando durante el funcionamiento. El mapa final para conocer los huecos libres se calculará como la OR de ambos.

Por otra parte, si el usuario da la orden de mover el vehículo a un determinado punto, éste nivel deberá dar las órdenes necesarias para que se ejecute la orden. En este caso la orden que pasará al nivel inferior vendrá dada por un código de orden y un parámetro que indique el destino. Luego en general, si la orden no se ejecuta completamente en este nivel, la información que se pasará al nivel inferior es:

Salida nivel inferior: {código orden; parámetros}

Para el ejemplo que nos ocupa, los parámetros podrían ser:

parámetros: {coordenada_inical coordenada_final}

A partir de esta información comenzará a operar el nivel inferior.

Nivel de gestión de tareas.

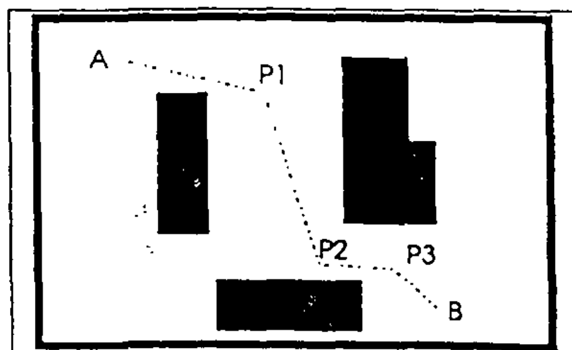
Para poder operar correctamente deberá disponer de la información necesaria almacenada en una base de datos llamada **modelado del mundo**, con la información siguiente:

* Mapa(s) con información de los obstáculos.
* Tipo de terreno.

La misión principal en esta capa es la de planificación al más alto nivel. Por ejemplo, para el caso de movimiento, supongamos que el usuario ha seleccionado llevar el vehículo desde un punto A hasta otro B que se encuentra distante. Para alcanzar esta meta habrá que salvar toda una serie de obstáculos conocidos en este nivel. Por lo tanto se puede descomponer ir desde A hasta B como una serie de puntos (Px) libres de colisiones:

$$T\{A,B\} = T\{A,P1,P2,\ldots,Pn,B\}$$

E x i s t e n



diferentes técnicas que permiten la planificación de las trayectorias, algunas de ellas requieren gran capacidad de almacenamiento y de cálculo. No existe una técnica que resuelva totalmente y de forma eficiente todo el problema:

* *Mapa de carreteras.*
* *Campos Potenciales.*
* *Descomposición celular.*

Entrada en este nivel:

Recibiremos del nivel superior un trabajo a realizar que vendrá codificado, tal y como ya se ha descrito, del modo siguiente:

trabajo: {código; parámetros}

Con esos parámetros de entrada éste nivel debe conocer el trabajo que deberá realizar y que consistirá básicamente en descomponer el trabajo en una serie de tareas que puedan ser llevadas a cabo por un nivel inferior

Salida de este nivel:

Supongamos por ejemplo que se desea mover el vehículo de una coordenada origen hasta otra destino, tras la actuación de éste nivel y teniendo en cuenta la información disponible en la base de datos, se deberá descomponer en una serie de tareas donde cada tarea sea por ejemplo el llevar el vehículo a una coordenada intermedia que en principio sea libre de bloqueos.

Luego la salida que se obtiene a este nivel será:

$$Trabajo = \{tareas_i\ \}$$

Donde las *tareas* pueden ser, por ejemplo, un conjunto de puntos que describen un camino por el que deberá pasar el vehículo para alcanzar la coordenada final desde la inicial.

Nivel de gestión de trayectorias.

Encargado de planificar y monitorizar las trayectorias que se van a ejecutar. Se debe garantizar que el AGV alcance los puntos planificados en el nivel anterior. En este nivel también se recibe información, proveniente del *control del hardware*, referente al sistema de sensorización (Ej: Ultrasonidos, visión, etc.). Por lo tanto se deben realizar dos controles simultáneamente, por un lado la monitorización de que la trayectoria se está efectuando correctamente a partir de la información recibida del sistema de percepción y de otro lado la corrección de posibles desviaciones o la evitación de obstáculos imprevistos.

Dentro de este nivel también se deberá introducir algún sistema de localización absoluta del AGV. A este respecto hay diferentes posibilidades por ejemplo:

* *Recibir información por radiofrecuencia.*
* *Utilizar balizas luminosas que serán identificadas por sensores.*
* *Reconocer marcas en el techo mediante visión artificial.*
* *Etc.*

Entrada a este nivel:

Recibe del nivel superior un **conjunto de tareas** que se deberán realizar, cada tarea consiste por ejemplo en una coordenada por la que tiene que pasar el vehículo. Por otra parte también se recibe información de los sensores que nos indica por dónde se está moviendo.

Si se dispone de algún mecanismo de localización absoluta también deberá ser tenido en cuenta a este nivel con el fin de poder corregir posibles desviaciones respecto de la trayectoria o posición deseada.

Salida de este nivel:

Envía al nivel inferior **comandos de movimiento**. Debido a que la presente arquitectura está pensada para el vehículo comercial *ROBUTER-II*, los comandos que se generará son los específicos para él y son suministrados por el fabricante a nivel del sistema operativo *ALBATROS*.

Algunos de estos comandos devuelven parámetros como confirmación de su ejecución. Antes de enviar una nueva salida se deberá prever que el comando anterior se ejecute correctamente.

Nivel de control del hardware.

Nos encontramos al más bajo nivel dentro de la arquitectura. Las labores encaminadas en esta capa van a ir encaminadas al movimiento del vehículo mediante la **actuación y control de los motores**. También se recogerá información del **sistema de percepción sensorial**, de forma que toda la información recogida por los sensores y sistema de visión se trasladará a niveles superiores para su procesamiento.

También en este módulo se tomarán **acciones de emergencia** para poder evitar situaciones de choque inminentes ante la presencia de objetos imprevistos. Por lo tanto, en este nivel se debe actuar de emergencia en una primera instancia y la decisión final ante esta situación será tomada en niveles superiores que es donde se tiene el "modelo del mundo".

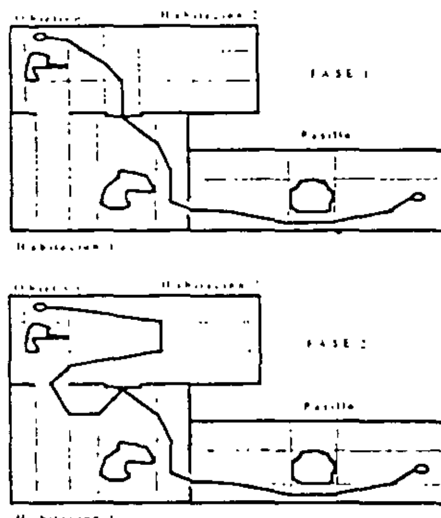
El funcionamiento de este nivel debe ser en tiempo real, ya que es crítico el tiempo de respuesta que se obtenga. Por ello uno de los problemas que se plantean es cómo obtener un código lo más eficiente posible, que pueda manejar y procesar todo el volumen de información en el menor tiempo posible.

CARACTERISTICAS GENERALES DE LAS ARQUITECTURAS CLÁSICAS

- PRESENTAN UN ALTO GRADO DE EXACTITUD EN LA CONSECUCIÓN DEL OBJETIVO.
- SE TRATA, EN SU MAYORÍA DE ACCIONES DE ALTO NIVEL CON UN ALTO COSTE COMPUTACIONAL.
- PRESENTAN UNA DISPOSICIÓN SERIE QUE ORIGINA UN FALLO DE FUNCIONAMIENTO DEL SISTEMA SI UN MÓDULO FALLA.
- NO ES ADECUADO EN ENTORNOS DINÁMICOS, DESCONOCIDOS O EN AQUELLOS CASOS EN LOS QUE LAS MEDIDAS CAPTADAS POR LOS SENSORES NO TIENEN UNA MÍNIMA FIABILIDAD.

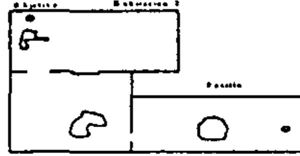
ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

REPLANIFICACIÓN EN LAS ARQUITECTURAS CLÁSICAS

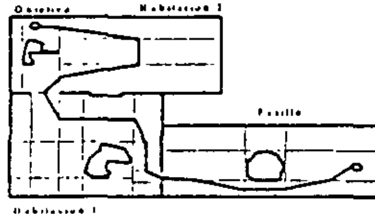


ETAPAS PARA LA CONSECUCIÓN DE UNA MISIÓN

1. CAPTAR LA INFORMACIÓN DEL ENTORNO
2. CREAR UN MAPA DEL ENTORNO, A PARTIR DE ESTA INFORMACIÓN



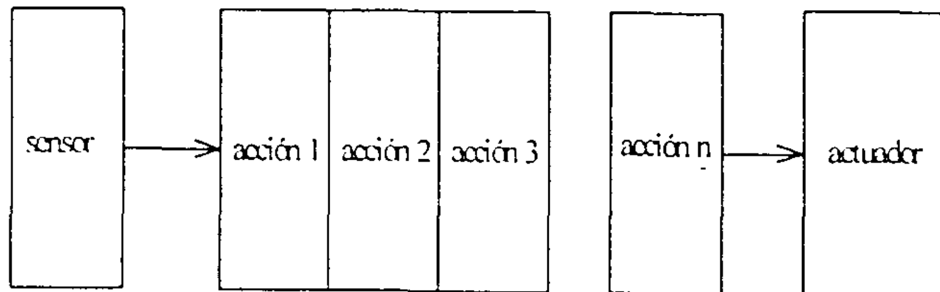
3. PLANIFICAR LAS ACCIONES A EJECUTAR



4. EJECUTAR DICHAS ACCIONES

ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

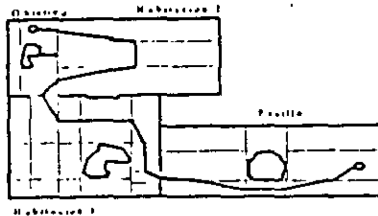
ARQUITECTURAS TRADICIONALES (SMPA)



ETAPAS PARA LA CONSECUCIÓN DE UNA MISIÓN

1. CAPTAR LA INFORMACIÓN DEL ENTORNO
2. CREAR UN MAPA DEL ENTORNO, A PARTIR DE ESTA INFORMACIÓN

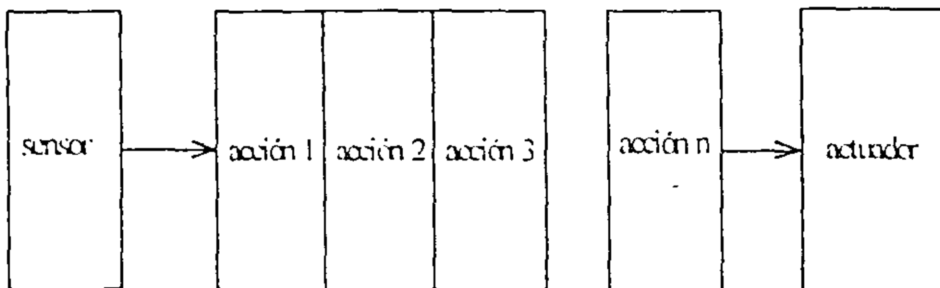
3. PLANIFICAR LAS ACCIONES A EJECUTAR



4. EJECUTAR DICHAS ACCIONES

ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

ARQUITECTURAS TRADICIONALES (SMPA)

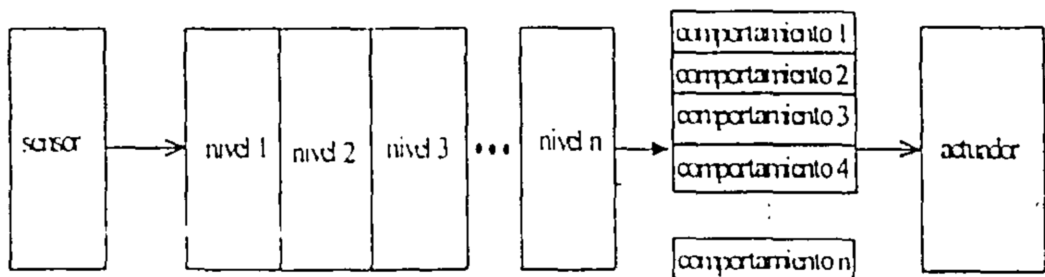


PRINCIPALES CARACTERISTICAS DE LOS COMPORTAMIENTOS REACTIVOS

- LA SALIDA DE CADA COMPORTAMIENTO DEBE SER COMPLETAMENTE DETERMINISTA.
- CADA COMPORTAMIENTO ES INDEPENDIENTE DEL RESTO
- LOS COMPORTAMIENTOS NO POSEEN MEMORIA.
- DAN COMO RESULTADO RESPUESTAS INNATAS A ESTIMULOS INTERIORES Y EXTERIORES.
- EL RESULTADO DEBE SER HOMOGENEO.

ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

ARQUITECTURAS MIXTAS

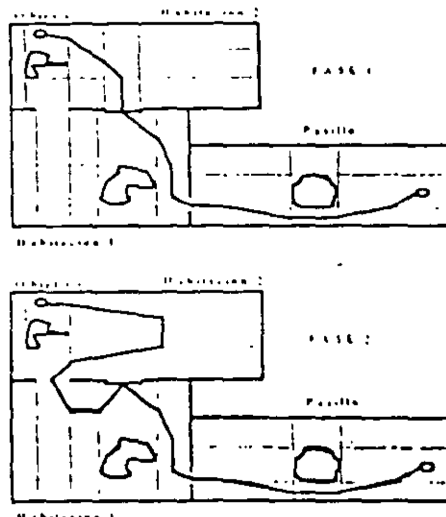


CARACTERISTICAS GENERALES DE LAS ARQUITECTURAS CLÁSICAS

- PRESENTAN UN ALTO GRADO DE EXACTITUD EN LA CONSECUCIÓN DEL OBJETIVO.
- SE TRATA, EN SU MAYORÍA DE ACCIONES DE ALTO NIVEL CON UN ALTO COSTE COMPUTACIONAL.
- PRESENTAN UNA DISPOSICIÓN SERIE QUE ORIGINA UN FALLO DE FUNCIONAMIENTO DEL SISTEMA SI UN MÓDULO FALLA.
- NO ES ADECUADO EN ENTORNOS DINÁMICOS, DESCONOCIDOS O EN AQUELLOS CASOS EN LOS QUE LAS MEDIDAS CAPTADAS POR LOS SENSORES NO TIENEN UNA MÍNIMA FIABILIDAD.

ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

REPLANIFICACIÓN EN LAS ARQUITECTURAS CLÁSICAS



TAREAS A REALIZAR POR UN ROBOT MOVIL

- CAPTAR INFORMACION DEL ENTORNO
- PREPROCESAR ESA INFORMACION
- MODELIZAR EL ENTORNO
- CALCULO DE TRAYECTORIAS
- RESPONDER ANTE UNA ORDEN INMEDIATA
- COMUNICARSE CON EL USUARIO

ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

SISTEMAS INICIALES DE TRANSPORTE DE MATERIALES

- GUIADO POR RAILES
- GUIADO ELECTROMAGNETICO MEDIANTE CABLE ENTERRADO EN EL SUELO.
- GUIADO POR SEGUIMIENTO DE LINEA PINTADA EN EL SUELO.

Entrada a este nivel:

Básicamente será cualquiera de los comandos o primitivas de funcionamiento de las que dispone el vehículo *ROBUTER-II*.

Además también se utilizará información procedente de los sensores para ser retransmitida al nivel superior y tomar decisiones de emergencia en un momento determinado para evitar la colisión inmediata.

Salida de este nivel:

Principalmente consiste en la actuación directa sobre los motores para poder mover el vehículo, tal y como se suministra el vehículo no será necesario desarrollar los reguladores y control de los mismos puesto que ya está implementado.

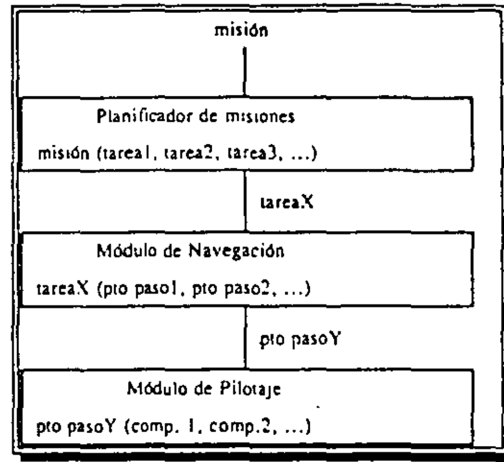
También se darán las órdenes de lectura de los sensores para que se produzca la realimentación necesaria para el funcionamiento.

El *ROBUTER-II* dispone de un sistema operativo en tiempo real llamado *Albatros* que se utiliza para controlar y programar el AGV en este nivel. Adicionalmente se encuentra el entorno *ASDP* (*Albatros Software Development Package*) que es un entorno software para desarrollar y cargar aplicaciones software para el *ROBUTER-II*.

Para tener mejor idea de la forma en la que trabaja el *ROBUTER-II*, seguidamente se presentan algunos de los numerosos comandos disponibles para su funcionamiento:

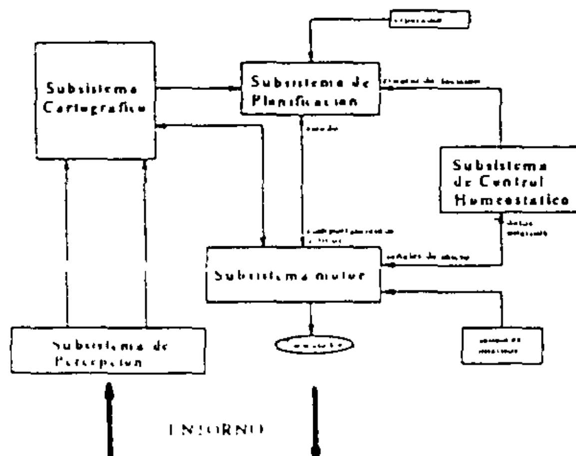
COMANDO	DESCRIPCIÓN
MONT <param>	Comando básico de movimiento.
MOTV <param>	Control velocidad lineal, angular.
MOTP <param>	Ejecuta trayectorias entre puntos especificados.
JSK <param>.	Utiliza y controla el uso del joystick
READ <param>	Leer sensores de ultrasonidos.
SNEW <param>	Añade descriptor sensor a la tabla de configuración.
SSET <param>	Modificar descriptor del sensor en la tabla de config.
BUMP <param>	Activa parada emergencia.

EJEMPLO DE PLANIFICACIÓN DE UNA MISIÓN MEDIANTE UNA ARQUITECTURA MIXTA: ARQUITECTURA AuRA



ARQUITECTURAS DE CONTROL EN EL CAMPO DE LA ROBOTICA MOVIL

ARQUITECTURA AuRA



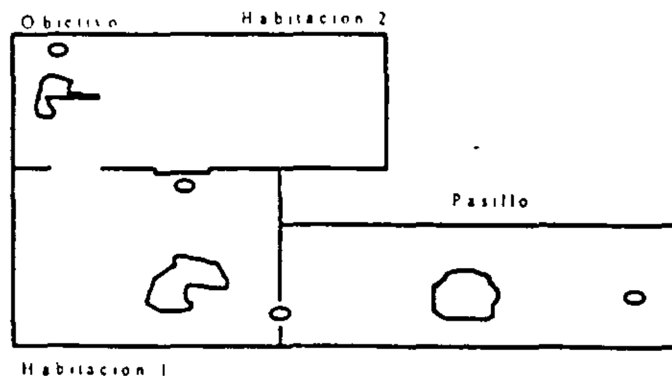
LISTA DE COMPORTAMIENTOS LANZADOS PARA LA EJECUCIÓN DEL ANTERIOR EJEMPLO EN LA ARQUITECTURA AuRA (PRIMER PUNTO DE PASO).

- MOVERSE HACIA LA PUERTA

- MOVERSE HACIA LA PUERTA
- EVITAR UN OBSTÁCULO ESTÁTICO

- MOVERSE HACIA LA PUERTA

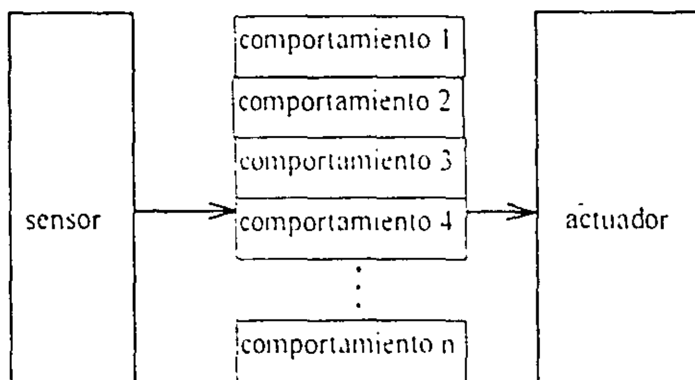
EJEMPLO DE PLANIFICACIÓN MEDIANTE ARQUITECTURA MIXTA: ARQUITECTURA AuRA



PRINCIPALES CARACTERISTICAS DE LAS ARQUITECTURAS REACTIVAS

- SE ESTABLECE UNA DESCOMPOSICIÓN POR COMPORTAMIENTOS.
- NO ES NECESARIO UN MODELO GLOBAL DEL ENTORNO
- NO SE NECESITA CONOCIMIENTO, A PRIORI, DEL ENTORNO.
- NO PRECISA PLANIFICACIÓN.
- ES ADECUADO PARA ENTORNOS DINÁMICOS Y CON INCERTIDUMBRE SENSORIAL.

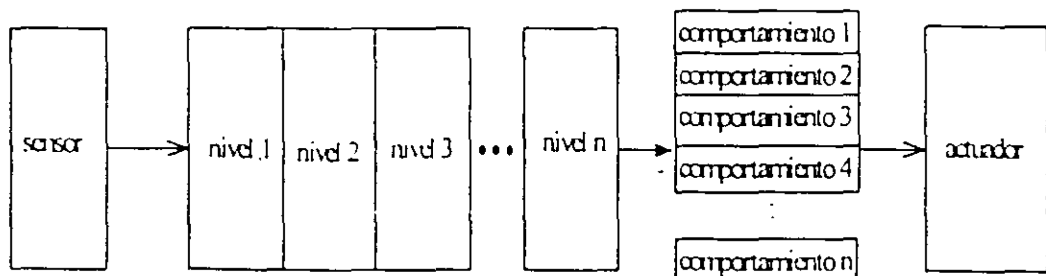
ARQUITECTURAS REACTIVAS



PRINCIPALES CARACTERISTICAS DE LOS COMPORTAMIENTOS REACTIVOS

- LA SALIDA DE CADA COMPORTAMIENTO DEBE SER COMPLETAMENTE DETERMINISTA.
- CADA COMPORTAMIENTO ES INDEPENDIENTE DEL RESTO
- LOS COMPORTAMIENTOS NO POSEEN MEMORIA.
- DAN COMO RESULTADO RESPUESTAS INNATAS A ESTIMULOS INTERIORES Y EXTERIORES.
- EL RESULTADO DEBE SER HOMOGENEO.

ARQUITECTURAS MIXTAS



LISTA DE COMPORTAMIENTOS LANZADOS PARA LA EJECUCIÓN DEL ANTERIOR EJEMPLO EN LA ARQUITECTURA AuRA (OBJETIVO FINAL).

- MOVERSE HACIA EL OBJETIVO

- MOVERSE HACIA EL OBJETIVO
- EVITAR UN OBSTÁCULO ESTÁTICO

- MOVERSE HACIA EL OBJETIVO

LISTA DE COMPORTAMIENTOS LANZADOS PARA LA EJECUCIÓN DEL ANTERIOR EJEMPLO EN LA ARQUITECTURA AuRA (SEGUNDO PUNTO DE PASO).

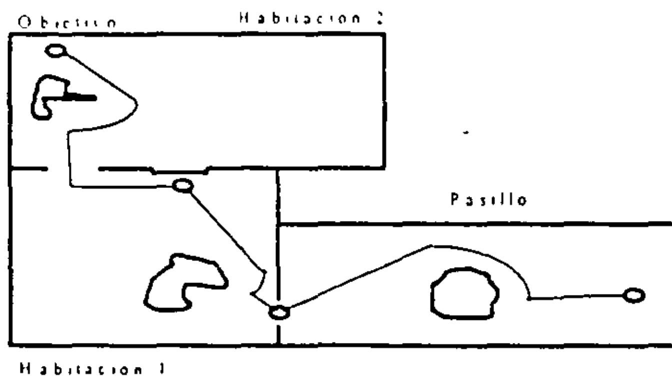
- MOVERSE HACIA LA PUERTA

- MOVERSE HACIA LA PUERTA
- EVITAR UN OBSTÁCULO ESTÁTICO

- MOVERSE HACIA LA PUERTA

- SEGUIMIENTO DE UN MURO

EJEMPLO DE PLANIFICACIÓN MEDIANTE ARQUITECTURA MIXTA: ARQUITECTURA AuRA. TRAJECTORIA FINAL



8

FUZZY ASSOCIATIVE MEMORIES

FUZZY SYSTEMS AS BETWEEN-CUBE MAPPINGS

Chapter 7 introduced multivalued or fuzzy sets as points in the unit hypercube $I^n = [0, 1]^n$. Within the cube we were interested in the distance between points. This led to measures of the size and fuzziness of a fuzzy set and, more fundamentally, to a measure of how much one fuzzy set is a subset of another fuzzy set. This *within-cube* theory directly extends to the continuous case where the space X is a subset of R^n or, in general, where X is a subset of products of real or complex spaces.

The next step considers mappings *between* fuzzy cubes. This level of abstraction provides a surprising and fruitful alternative to the propositional and predicate-calculus reasoning techniques used in artificial-intelligence (AI) expert systems. It allows us to reason with sets instead of propositions.

The fuzzy-set framework is numerical and multidimensional. The AI framework is symbolic and one-dimensional, with usually only bivalent expert "rules" or propositions allowed. Both frameworks can encode structured knowledge in linguistic form. But the fuzzy approach translates the structured knowledge into a

flexible *numerical* framework and processes it in a manner that resembles neural-network processing. The numerical framework also allows us to adaptively infer and modify fuzzy systems, perhaps with neural or statistical techniques, directly from problem-domain sample data.

Between-cube theory is fuzzy-systems theory. A fuzzy set defines a point in a cube. A fuzzy system defines a mapping between cubes. A fuzzy system S maps fuzzy sets to fuzzy sets. Thus a **fuzzy system** S is a transformation $S\colon I^n \rightarrow I^p$. The n-dimensional unit hypercube I^n houses all the fuzzy subsets of the domain space, or input *universe of discourse*, $X = \{x_1, \ldots, x_n\}$. I^p houses all the fuzzy subsets of the range space, or output universe of discourse, $Y = \{y_1, \ldots, y_p\}$. X and Y can also denote subsets of R^n and R^p. Then the fuzzy power sets $F(2^X)$ and $F(2^Y)$ replace I^n and I^p.

In general a fuzzy system S maps families of fuzzy sets to families of fuzzy sets, thus $S\colon I^{n_1} \times \cdots \times I^{n_r} \rightarrow I^{p_1} \times \cdots \times I^{p_s}$. Here too we can extend the definition of a fuzzy system to allow arbitrary products of arbitrary mathematical spaces to serve as the domain or range spaces of the fuzzy sets.

(A technical comment is in order for sake of historical clarification. A tenet, perhaps the defining tenet, of the classical theory [Dubois, 1980] of fuzzy sets as functions concerns the fuzzy extension of any mathematical function. This tenet holds that any function $f\colon X \rightarrow Y$ that maps points in X to points in Y extends to map the fuzzy subsets of X to the fuzzy subsets of Y. The so-called *extension principle* defines the set-function $f\colon F(2^X) \rightarrow F(2^Y)$, where $F(2^X)$ denotes the fuzzy power set of X, the set of all fuzzy subsets of X. The formal definition of the extension principle is complicated. The key idea is a supremum of pairwise minima. Unfortunately, the extension principle achieves generality at the price of triviality. In general [Kosko, 1986a, 1987] the extension principle extends functions to fuzzy sets by stripping the fuzzy sets of their fuzziness, mapping the fuzzy sets into bit vectors of nearly all 1s. This shortcoming, combined with the tendency of the extension-principle framework to push fuzzy theory into largely inaccessible regions of abstract mathematics, led in part to the development of the alternative sets-as-points geometric framework of fuzzy theory.)

We shall focus on fuzzy systems $S\colon I^n \rightarrow I^p$ that map *balls* of fuzzy sets in I^n to balls of fuzzy sets in I^p. These continuous fuzzy systems behave as associative memories. They map close inputs to close outputs. We shall refer to them as **fuzzy associative memories**, or FAMs.

The simplest FAM encodes the **FAM rule** or association (A_i, B_i), which associates the p-dimensional fuzzy set B_i with the n-dimensional fuzzy set A_i. These minimal FAMs essentially map one ball in I^n to one ball in I^p. They are comparable to simple neural networks. But we need not adaptively train the minimal FAMs. As discussed below, we can directly encode structured knowledge of the form "If traffic is heavy in this direction, then keep the stop light green longer" in a Hebbian-style FAM correlation matrix. In practice we sidestep this large numerical matrix with a virtual representation scheme. In place of the matrix the user

encodes the fuzzy-set association (HEAVY, LONGER) as a single linguistic entry in a FAM-bank linguistic matrix.

In general a **FAM system** $F: I^n \rightarrow I^p$ encodes and processes in parallel a **FAM bank** of m FAM rules $(A_1, B_1), \ldots, (A_m, B_m)$. Each input A to the FAM system activates each stored FAM rule to different degree. The minimal FAM that stores (A_i, B_i) maps input A to B'_i, a partially activated version of B_i. The more A resembles A_i, the more B'_i resembles B_i. The corresponding output fuzzy set B combines these partially activated fuzzy sets $B'_1, \ldots, B'_m$. B equals a weighted average of the partially activated sets:

$$B = w_1 B'_1 + \cdots + w_m B'_m$$

where w_i reflects the credibility, frequency, or strength of the fuzzy association (A_i, B_i). In practice we usually "defuzzify" the output waveform B to a single numerical value y_j in Y by computing the fuzzy centroid of B with respect to the output universe of discourse Y.

More general still, a FAM system encodes a bank of compound FAM rules that associate multiple output or consequent fuzzy sets $B_i^1, \ldots, B_i^s$ with multiple input or antecedent fuzzy sets $A_i^1, \ldots, A_i^r$. We can treat compound FAM rules as compound linguistic conditionals. This allows us to naturally, and in many cases easily, obtain structural knowledge. We combine antecedent and consequent sets with logical conjunction, disjunction, or negation. For instance, we would interpret the compound association $(A^1, A^2; B)$ linguistically as the compound conditional "IF X^1 is A^1 AND X^2 is A^2, THEN Y is B" if the comma in the fuzzy association $(A^1, A^2; B)$ denotes conjunction instead of, say, disjunction.

We specify in advance the numerical universes of discourse for fuzzy variables X^1, X^2, and Y. For each universe of discourse or fuzzy variable X, we specify an appropriate *library* of fuzzy-set values, $A_1^r, \ldots, A_k^r$. Contiguous fuzzy sets in a library overlap. In principle a neural network can estimate these libraries of fuzzy sets. In practice this is usually unnecessary. The library sets represent a weighted, though overlapping, quantization of the input space X. They represent the fuzzy-set *values* assumed by a fuzzy *variable*. A different library of fuzzy sets similarly quantizes the output space Y. Once we define the library of fuzzy sets, we construct the FAM by choosing appropriate combinations of input and output fuzzy sets. Adaptive techniques can make, assist, or modify these choices.

An **adaptive FAM** (AFAM) is a *time-varying* FAM system. System parameters gradually change as the FAM system samples and processes data. Below we discuss how neural network algorithms can adaptively infer FAM rules from training data. In principle, learning can modify other FAM system components, such as the libraries of fuzzy sets or the FAM-rule weights w_i.

Below we propose and illustrate an unsupervised adaptive clustering scheme, based on competitive learning, to "blindly" generate and refine the bank of FAM rules. In some cases we can use supervised learning techniques if we have additional information to accurately generate error estimates.

78

FUZZY AND NEURAL FUNCTION ESTIMATORS

Neural and fuzzy systems estimate sampled functions and behave as associative memories. They share a key advantage over traditional statistical-estimation and adaptive-control approaches to function estimation. They are *model-free* estimators. Neural and fuzzy systems estimate a function without requiring a mathematical description of how the output functionally depends on the input. They "learn from example." More precisely, they learn from samples.

Both approaches are numerical, can be partially described with theorems, and admit an algorithmic characterization that favors silicon and optical implementation. These properties distinguish neural and fuzzy approaches from the symbolic processing approaches of artificial intelligence.

Neural and fuzzy systems differ in how they estimate sampled functions. They differ in the kind of samples used, how they represent and store those samples, and how they associatively "inference" or map inputs to outputs.

These differences appear during system construction. The neural approach requires the specification of a nonlinear dynamical system, usually feedforward, the acquisition of a sufficiently representative set of numerical training samples, and the encoding of those training samples in the dynamical system by repeated learning cycles. The fuzzy system requires only that we partially fill in a linguistic "rule matrix." This task is markedly simpler than designing and training a neural network. Once we construct the systems, we can present the same numerical inputs to either system. The outputs will reside in the same numerical space of alternatives. So both systems define a surface or manifold in the input-output product space $X \times Y$. We present examples of these surfaces in Chapters 9, 10, and 11.

Which system, neural or fuzzy, is more appropriate for a particular problem depends on the nature of the problem and the availability of numerical and structured data. To date engineers have applied fuzzy techniques largely to control problems. These problems often permit comparison with standard control-theoretic and expert-system approaches. Neural networks so far seem best applied to ill-defined two-class pattern-recognition problems (defective or nondefective, bomb or not, etc.).

Fuzzy systems estimate functions with *fuzzy-set* samples (A_i, B_i). Neural systems use *numerical-point* samples (x_i, y_i). Both kinds of samples reside in the input-output product space $X \times Y$.

Figure 8.1 illustrates the geometry of fuzzy-set and numerical-point samples taken from the function $f: X \rightarrow Y$.

Engineers sometimes call the fuzzy-set association (A_i, B_i) a "rule." This is misleading, since reasoning with sets is not the same as reasoning with propositions. Reasoning with sets is harder. Sets are multidimensional, and matrices, not propositional conditionals, house associations. We must take care how we define each term and operation. We shall refer to the antecedent term A_i in the fuzzy association (A_i, B_i) as the **input associant** and the consequent term B_i as the **output associant**.

The fuzzy-set sample (A_i, B_i) encodes *structure*. It represents a mapping, a minimal *fuzzy association* of part of the output space with part of the input space.

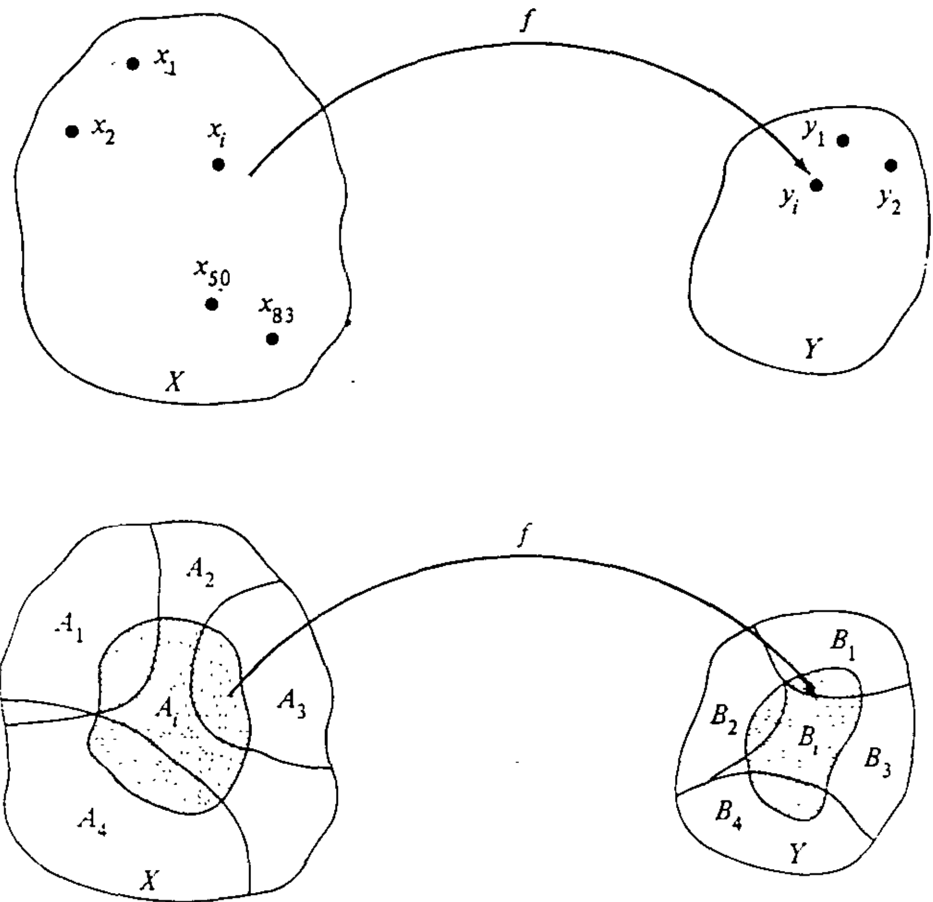


FIGURE 8.1 Function f maps domain X to range Y. In the first illustration we use several numerical-point samples (x_i, y_i) to estimate $f: X \to Y$. In the second case we use only a few fuzzy subsets A_i of X and B_i of Y. The fuzzy association (A_i, B_i) represents system structure, as an adaptive clustering algorithm might infer or as an expert might articulate. In practice there are usually fewer different output associants or "rule" consequents B_i than input associants or antecedents A_i.

In practice this resembles a meta-rule—IF A_i, THEN B_i—the type of structured linguistic rule an expert might articulate to build an expert-system "knowledge base." The association might also represent the result of an adaptive clustering algorithm.

Consider a fuzzy association for the intelligent control of a traffic light: "If the traffic is heavy in this direction, then keep the light green longer." The fuzzy association is (HEAVY, LONGER). The input fuzzy variable *traffic density* assumes the fuzzy-set value HEAVY. The output fuzzy variable *green light duration* assumes the fuzzy-set value LONGER. Another fuzzy association might be (LIGHT, SHORTER). The fuzzy system encodes each linguistic association or "rule" in a numerical *fuzzy associative memory* (FAM) mapping. The FAM then numerically processes numerical input data. A measured description of traffic density (e.g., 150 cars per unit road surface area) then corresponds to a unique numerical output (e.g., 3 seconds), the "recalled" output.

The degree to which a particular measurement of traffic density is heavy depends on how we define the fuzzy set of heavy traffic. The definition may arise from statistical or neural clustering of historical data or from pooling the responses of experts. In practice the fuzzy engineer and the problem-domain expert agree on one of many possible libraries of fuzzy-set values for the fuzzy variables.

The degree to which the traffic light stays green longer depends on the degree to which the measured traffic density is heavy. In the simplest case the two degrees are the same. In general the two degrees differ. In actual fuzzy systems the output-control variables—in this case the single variable green-light duration—depend on many FAM-rule antecedents or associants activated to different degrees by incoming data.

Neural vs. Fuzzy Representation of Structured Knowledge

The distinction between fuzzy and neural systems begins with how they represent structured knowledge. How would a neural network encode the same associative information? How would a neural network encode the structured knowledge "If the traffic is heavy in this direction, then keep the light green longer"?

The simplest method encodes two associated numerical vectors. One vector represents the input associant HEAVY. The other represents the output associant LONGER. But this is too simple. For the neural network's fault tolerance now works to its disadvantage. The network tends to reconstruct partial inputs to complete sample inputs. It erases the desired partial degrees of activation. If an input is close to A_i, the output will tend to be B_i. If the output is distant from A_i, the output will tend to be some other sampled output vector or a spurious output altogether.

A better neural approach encodes a mapping from the heavy-traffic subspace to the longer-time subspace. Then the neural network needs a representative sample set to capture this structure. Statistical networks, such as adaptive vector quantizers, may need thousands of statistically representative samples. Feedforward multilayer neural networks trained with the backpropagation algorithm in Chapter 5 may need hundreds of representative numerical input-output pairs and may need to recycle these samples tens of thousands of times in the learning process.

The neural approach suffers a deeper problem than just the computational burden of training. *What* does it encode? How do we know the network encodes the original structure? What does it recall? There is no natural inferential audit trail. System nonlinearities wash it away. Unlike an expert system, we do not know which inferential paths the network uses to reach a given output or even which inferential paths exist. There is only a large system of synchronous or asynchronous nonlinear functions. Unlike, say, the adaptive Kalman filter, we cannot appeal to a postulated mathematical model of how the output state depends on the input state. Model-free estimation is, after all, the central computational advantage of neural networks. The cost is system inscrutability.

We are left with an unstructured computational black box. We do not know

what the neural network encoded during training or what it will encode or forget in further training. (For competitive adaptive vector quantizers we do know that synaptic vectors asymptotically estimate sample-space centroids and perhaps higher-order moments.) We can characterize the neural network's behavior only by exhaustively passing all inputs through the black box and recording the recalled outputs. The characterization may use a summary scalar like mean-squared error.

This black-box characterization of the network's behavior involves a computational *dilemma*. On the one hand, for most problems the number of input-output cases we need to check is computationally prohibitive. On the other, when the number of input-output cases is tractable, we may as well store these pairs and appeal to them directly, and without error, as a look-up table. In the first case the neural network is unreliable. In the second case it is unnecessary.

A further problem is sample generation. Where did the original numerical point samples come from? Did we ask an expert to give numbers? How reliable are such numerical vectors, especially when the expert feels most comfortable giving the original linguistic data? This procedure seems at most as reliable as the expert-system method of asking an expert to give condition-action rules with numerical uncertainty weights.

Statistical neural estimators require a "statistically representative" sample set. We may need to randomly "create" these samples from an initial small sample set by bootstrap techniques or by random-number generation of points clustered near the original samples. Both sample-augmentation procedures assume that the initial sample set sufficiently represents the underlying probability distribution. The problem of where the original sample set comes from remains. The fuzziness of the notion "statistically representative" compounds the problem. In general we do not know in advance how well a given sample set reflects an unknown underlying distribution of points. Indeed when the network adapts on-line, we know only past samples. The remainder of the sample set resides in the unsampled future.

In contrast, fuzzy systems directly encode the linguistic sample (HEAVY, LONGER) in a dedicated numerical matrix, perhaps of infinite dimensions. The default encoding technique is the fuzzy Hebb procedure discussed below. For practical problems we need not store this large, perhaps infinite, numerical matrix. Instead we use a virtual representation scheme. Numerical point inputs permit this simplification. Mathematically we implicitly pass large unit bit vectors, or delta pulses in the continuous case, through the FAM-rule matrix. In general we describe inputs by an uncertainty distribution, probabilistic or fuzzy. Then we must use the entire matrix or reduce the input to a scalar by averaging.

For instance, if the *heavy traffic* input is 150 cars, we can omit the FAM matrix. Below we refer to these systems as binary input-output FAMs, or BIOFAMs. But if the input is a Gaussian curve with mean 150, then in principle we must process the vector input with a FAM matrix. (In practice we might use only the mean.) The dimensions of the *linguistic* FAM-bank matrix are usually small. The dimensions reflect the quantization levels of the input and output spaces, the number of fuzzy-set values assumed by the fuzzy variables.

82

The fuzzy approach combines the purely numerical approaches of neural networks and mathematical modeling with the symbolic, structure-rich approaches of artificial intelligence. We acquire knowledge symbolically—or numerically if we use adaptive techniques—but represent it numerically. We also process data numerically. Adaptive FAM rules correspond to common-sense, often nonarticulated, behavioral rules that improve with experience.

This approach does not abandon neural-network techniques. Instead, it limits them to *unstructured* parameter and state estimation, pattern recognition, and cluster formation. The system *architecture* remains fuzzy.

FAMs as Mappings

Fuzzy associative memories (FAMs) are transformations. *FAMs map fuzzy sets to fuzzy sets.* They map unit cubes to unit cubes, as in Figure 8.1. In the simplest case the FAM system consists of a single association, such as (HEAVY, LONGER). In general the FAM system consists of a bank of different FAM associations. Each association corresponds to a different numerical FAM matrix, or a different entry in a linguistic FAM-bank matrix. We do not combine these matrices as we combine or superimpose neural-network associative-memory (outer-product) matrices. (A exception is the *fuzzy cognitive map* [Kosko, 1988; Taber, 1987, 1991].) We store the matrices separately and access them in parallel. This avoids crosstalk. Since we use a virtual (BIOFAM) representation scheme, the computational burden of the parallel access is light.

We begin with single-association FAMs. For concreteness let the fuzzy-set pair (A, B) encode the traffic-control association (HEAVY, LIGHT). We quantize the domain of traffic density to the n numerical variables $x_1, x_2, \ldots, x_n$. We quantize the range of green-light duration to the p variables $y_1, y_2, \ldots, y_p$. The elements x_i and y_j belong respectively to the ground sets $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_p\}$. x_1 might represent zero traffic density. y_p might represent 10 seconds.

The fuzzy sets A and B are multivalued or fuzzy subsets of X and Y. So A defines a point in the n-dimensional unit hypercube $I^n = [0, 1]^n$, and B defines a point in the p-dimensional fuzzy cube I^p. Equivalently, A and B define the membership functions m_A and m_B that map the elements x_i of X and y_j of Y to degrees of membership in [0, 1]. The membership values, or *fit* (fuzzy unit) values, indicate how much x_i belongs to or fits in subset A, and how much y_j belongs to B. We describe this with the abstract functions $m_A: X \longrightarrow [0, 1]$ and $m_B: Y \longrightarrow [0, 1]$. We shall freely view sets both as functions and as points in fuzzy power sets.

The geometric *sets-as-points* interpretation of finite fuzzy sets A and B as points in unit cubes allows a natural vector representation. We represent A and B by the numerical *fit vectors* $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_p)$, where $a_i = m_A(x_i)$, and $b_j = m_B(y_j)$. We can interpret the identifications A = HEAVY and B = LONGER to suit the problem at hand. Intuitively the a_i values should increase

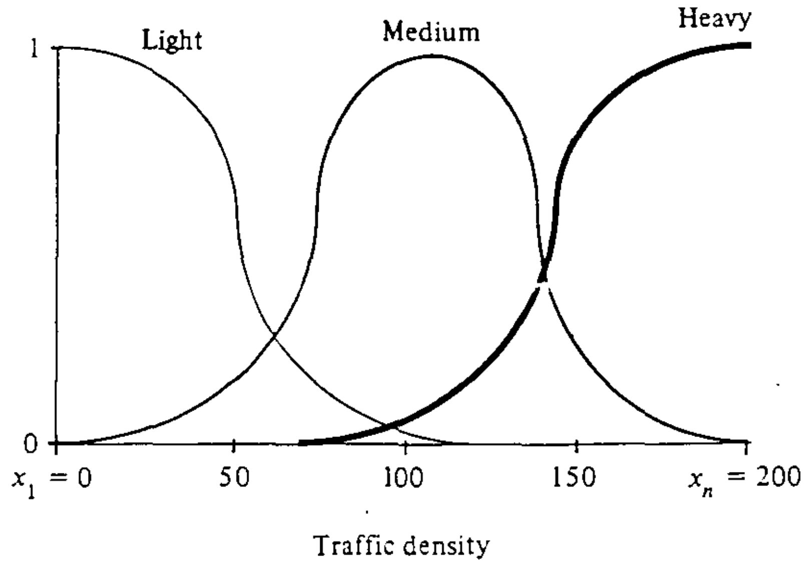


FIGURE 8.2 Three possible fuzzy subsets of traffic-density space X. Each fuzzy sample corresponds to such a subset. We draw the fuzzy sets as continuous membership functions. In practice membership values are sampled or quantized. So the sets are points in some unit hypercube I^n.

as the index i increases, perhaps approximating a sigmoid membership function. Figure 8.2 illustrates three possible fuzzy subsets of the universe of discourse X.

Fuzzy Vector-Matrix Multiplication: Max-Min Composition

Fuzzy vector-matrix multiplication resembles classical vector-matrix multiplication. We replace pairwise multiplications with pairwise minima. We replace column (row) sums with column (row) maxima. We denote this **fuzzy vector-matrix composition** relation, or the **max-min composition** relation [Klir, 1988], by the composition operator "$\circ$". For row fit vectors A and B and fuzzy n-by-p matrix M (a point in $I^{n \times p}$):

$$A \circ M = B \tag{8-1}$$

where we compute the "recalled" component b_j by taking the fuzzy inner product of fit vector A with the jth column of M:

$$b_j = \max_{1 \le i \le n} \min(a_i, m_{ij}) \tag{8-2}$$

Suppose we compose the fit vector $A = (.3 \ .4 \ .8 \ 1)$ with the fuzzy matrix M given by

$$M = \begin{pmatrix} .2 & .8 & .7 \\ .7 & .6 & .6 \\ .8 & .1 & .5 \\ 0 & .2 & .3 \end{pmatrix}$$

Then we compute the "recalled" fit vector $B = A \circ M$ componentwise as

$$\begin{aligned} b_1 &= \max\{\min(.3, .2), \min(.4, .7), \min(.8, .8), \min(1, 0)\} \\ &= \max(.2, .4, .8, 0) \\ &= .8 \\ b_2 &= \max(.3, .4, .1, .2) \\ &= .4 \\ b_3 &= \max(.3, .4, .5, .3) \\ &= .5 \end{aligned}$$

So $B = (.8\ \ .4\ \ .5)$. If we somehow encoded (A, B) in the FAM matrix M, we would say that the FAM system exhibits *perfect recall* in the forward direction: $A \circ M = B$.

The neural interpretation of max-min composition is that each neuron in field F_Y (or field F_B) generates its signal/activation value by fuzzy linear composition. Passing information back through M^T allows us to interpret the fuzzy system as a bidirectional associative memory (BAM). The bidirectional FAM theorems below characterize successful BAM recall for fuzzy correlation or Hebbian learning.

For completeness we also mention the **max-product composition** operator, which replaces minimum with product in (8-2):

$$b_j = \max_{1 \le i \le n} a_i m_{ij}$$

Fuzzy literature often confuses this composition operator with the fuzzy correlation encoding scheme discussed below. Max-product composition is a method for "multiplying" fuzzy matrices or vectors. Fuzzy correlation, which also uses pairwise products of fit values, constructs fuzzy matrices. In practice, and in the following discussion, we use only max-min composition.

FUZZY HEBB FAMS

Most fuzzy systems found in applications are fuzzy Hebb FAMs [Kosko, 1986b]. They are fuzzy systems $S\colon I^n \to I^p$ constructed in a simple neurallike manner. As discussed in Chapter 4, in neural-network theory we interpret the classical Hebbian hypothesis of correlation synaptic learning [Hebb, 1949] as unsupervised learning with the signal product $S_i S_j$:

$$\dot{m}_{ij} = -m_{ij} + S_i(x_i)S_j(y_j) \tag{8-3}$$

For a given pair of bipolar row vectors (X, Y), the neural interpretation gives the *outer-product* correlation matrix

$$M = X^T Y \tag{8-4}$$

85

We define pointwise the **fuzzy Hebb matrix** by the minimum of the "signals" a_i and b_j, an encoding scheme we shall call **correlation-minimum encoding**:

$$m_{ij} \quad = \quad \min(a_i, b_j) \tag{8-5}$$

given in matrix notation as the *fuzzy outer-product*

$$M \quad = \quad A^T \circ B \tag{8-6}$$

Mamdani [1977] and Togai [1986] independently arrived at the fuzzy Hebbian prescription (8-5) as a multivalued logical-implication operator: truth $(a_i \rightarrow b_j) = \min(a_i, b_j)$. The min operator, though, is a symmetric truth operator. So it does not properly generalize the classical implication $P \rightarrow Q$, which is false if and only if the antecedent P is true and the consequent Q is false, $t(P) = 1$ and $t(Q) = 0$. In contrast, a like desire to define a "conditional-possibility" matrix pointwise with continuous implication values led Zadeh [1983] to choose the Lukasiewicz implication operator: $m_{ij} = \text{truth}\ (a_i \rightarrow b_j) = \min(1, 1 - a_i + b_j)$. Unfortunately the Lukasiewicz operator usually equals or approximates unity, for $\min(1, 1 - a_i - b_j) < 1$ iff $a_i > b_j$. Most entries of the resulting matrix M are unity or near unity. This ignores the information in the association (A, B). So $A' \circ M$ tends to equal the largest fit value a'_i for any system input A'.

We construct an *autoassociative* fuzzy Hebb FAM matrix by encoding the redundant pair (A, A) in (8-6) as the fuzzy autocorrelation matrix:

$$M = A^T \circ A \tag{8-7}$$

In the previous example the matrix M was such that the input $A = (.3\ \ .4\ \ .8\ \ 1)$ recalled fit vector $B = (.8\ \ .4\ \ .5)$ upon max-min composition: $A \circ M = B$. Will A still recall B if we replace the original matrix M with the fuzzy Hebb matrix found with (8-6)? Substituting A and B in (8-6) gives

$$M \quad = \quad A^T \circ B \quad = \quad \begin{pmatrix} .3 \\ .4 \\ .8 \\ 1 \end{pmatrix} \circ (.8\ \ .4\ \ .5) \quad = \quad \begin{pmatrix} .3 & .3 & .3 \\ .4 & .4 & .4 \\ .8 & .4 & .5 \\ .8 & .4 & .5 \end{pmatrix}$$

This fuzzy Hebb matrix M illustrates two key properties. First, the ith row of M equals the pairwise minimum of a_i and the output associant B. Symmetrically, the jth column of M equals the pairwise minimum of b_j and the input associant A:

$$M \quad = \quad \begin{bmatrix} a_1 \wedge B \\ \hline \vdots \\ \hline a_n \wedge B \end{bmatrix} \tag{8-8}$$

$$= \quad [b_1 \wedge A^T | \dots | b_p \wedge A^T] \tag{8-9}$$

where the cap operator denotes pairwise minimum: $a_i \wedge b_j = \min(a_i, b_j)$. The term $a_i \wedge B$ indicates componentwise minimum:

$$a_i \wedge B \quad = \quad (a_i \wedge b_1, \dots, a_i \wedge b_p) \tag{8-10}$$

Hence if some $a_k = 1$, then the kth row of M equals B. If some $b_l = 1$, the lth column of M equals A. More generally, if some a_k is at least as large as every b_j, then the kth row of the fuzzy Hebb matrix M equals B.

Second, the third and fourth rows of M equal the fit vector B. Yet no column equals A. This allows perfect recall in the forward direction, $A \circ M = B$, but not in the backward direction, $B \circ M^T \neq A$:

$$A \circ M = (.8 \ .4 \ .5) = B$$

$$B \circ M^T = (.3 \ .4 \ .8 \ .8) = A' \subset A$$

A' is a proper subset of A : $A' \neq A$ and $S(A', A) = 1$, where S measures the degree of subsethood of A' in A, as discussed in Chapter 7. In other words, $a'_i \leq a_i$ for each i and $a'_k < a_k$ for at least one k. The bidirectional FAM theorems below show that this holds in general: If $B' = A \circ M$ differs from B, then B' is a proper subset of B. Hence fuzzy subsets map to fuzzy subsets.

The Bidirectional FAM Theorem for Correlation-Minimum Encoding

Analysis of FAM recall uses the traditional [Klir, 1988] fuzzy-set notions of the *height* and the *normality* of fuzzy sets. The **height** $H(A)$ of fuzzy set A is the maximum fit value of A:

$$H(A) = \max_{1 \leq i \leq n} a_i$$

A fuzzy set is **normal** if $H(A) = 1$, if at least one fit value a_k is maximal: $a_k = 1$. In practice fuzzy sets are usually normal. We can extend a nonnormal fuzzy set to a normal fuzzy set by adding a dummy dimension with corresponding fit value $a_{n+1} = 1$.

Recall accuracy in fuzzy Hebb FAMs constructed with correlation-minimum encoding depends on the heights $H(A)$ and $H(B)$. Normal fuzzy sets exhibit perfect recall. Indeed (A, B) is a bidirectional fixed point—$A \circ M = B$, and $B \circ M^T = A$—if and only if $H(A) = H(B)$, which always holds if A and B are normal. This is a corollary of the bidirectional FAM theorem [Kosko, 1986a] for correlation-minimum encoding. Below we present a similar theorem for correlation-product encoding.

Correlation-minimum bidirectional FAM theorem. If $M = A^T \circ B$, then

$$\begin{array}{lll} \text{(i)} & A \circ M = B & \text{iff } H(A) \geq H(B) \\ \text{(ii)} & B \circ M^T = A & \text{iff } H(B) \geq H(A) \\ \text{(iii)} & A' \circ M \subset B & \text{for any } A' \\ \text{(iv)} & B' \circ M^T \subset A & \text{for any } B' \end{array}$$

Proof. Observe that the height $H(A)$ equals the *fuzzy norm* of A:

$$A \circ A^T = \max_i a_i \wedge a_i = \max_i a_i = H(A)$$

Then

$$\begin{aligned} A \circ M &= A \circ (A^T \circ B) \\ &= (A \circ A^T) \circ B \\ &= H(A) \circ B \\ &= H(A) \wedge B \end{aligned}$$

So $H(A) \wedge B = B$ iff $H(A) \geq H(B)$, establishing (i). Now suppose A' is an arbitrary fit vector in I^n. Then

$$\begin{aligned} A' \circ M &= (A' \circ A^T) \circ B \\ &= (A' \circ A^T) \wedge B \end{aligned}$$

which establishes (iii) since $A' \circ A^T \leq H(A)$. A similar argument using $M^T = B^T \circ A$ establishes (ii) and (iv). **Q.E.D.**

The equality $A \circ A^T = H(A)$ implies an immediate corollary of the bidirectional FAM theorem. Supersets $A' \supset A$ behave the same as the encoded input associant A: $A' \circ M = B$ if $A \circ M = B$. Fuzzy Hebb FAMs ignore the information in the difference $A' - A$, when $A \subset A'$.

Correlation-Product Encoding

Correlation-product encoding provides an alternative fuzzy Hebbian encoding scheme. The standard mathematical outer product of the fit vectors A and B forms the FAM matrix M. Then

$$m_{ij} = a_i b_j \tag{8-11}$$

and in matrix notation,

$$M = A^T B \tag{8-12}$$

So the ith row of M equals the fit-scaled fuzzy set $a_i B$, and the jth column of M equals $b_j A^T$:

$$M = \begin{bmatrix} a_1 B \\ \hline \vdots \\ \hline a_n B \end{bmatrix} \tag{8-13}$$

$$= [b_1 A^T | \ldots | b_m A^T] \tag{8-14}$$

If $A = (.3\ \ .4\ \ .8\ \ 1)$ and $B = (.8\ \ .4\ \ .5)$ as above, we encode the FAM rule (A, B) with correlation product in the following matrix M:

$$M = \begin{pmatrix} .24 & .12 & .15 \\ .32 & .16 & .2 \\ .64 & .32 & .4 \\ .8 & .4 & .5 \end{pmatrix}$$

Note that if $A' = (0\ \ 0\ \ 0\ \ 1)$, then $A' \circ M = B$. The FAM system recalls output associant B to maximal degree. If $A' = (1\ \ 0\ \ 0\ \ 0)$, then $A' \circ M = (.24\ \ .12\ \ .15)$. The FAM system recalls output B only to degree .3.

Correlation-minimum encoding produces a matrix of clipped B sets, while correlation-product encoding produces a matrix of scaled B sets. In membership-function plots, the scaled fuzzy sets $a_i B$ all have the same shape as B. The clipped fuzzy sets $a_i \wedge B$ are flat at or above the a_i value. In this sense correlation-product encoding preserves more information than correlation-minimum encoding, an important point in fuzzy applications when we add output fuzzy sets together as in Equation (8-17) below. In the fuzzy-applications literature this often leads to the selection of correlation-product encoding.

Unfortunately, the fuzzy literature invariably confuses the correlation-product *encoding* scheme with the max-product composition method of recall or *inference*, as mentioned above. This widespread confusion warrants formal clarification.

In practice, and in the fuzzy applications developed in the next chapters, the input fuzzy set A' is a binary vector with one 1 and all other elements 0—a row of the n-by-n identity matrix (or a delta pulse in the continuous case). A' represents the occurrence of the crisp measurement datum x_i, such as a traffic density value of 30. When applied to the encoded FAM rule (A, B), the measurement value x_i activates A to degree a_i. This is part of the max-min composition recall process, for $A' \circ M = (A' \circ A^T) \circ B = a_i \wedge B$ or $a_i B$ depending on whether we encoded (A, B) in M with correlation-minimum or correlation-product encoding. We activate or "fire" the output associant B of the "rule" to degree a_i.

Since the values a'_i are binary, $a'_i m_{ij} = a'_i \wedge m_{ij}$. So the max-min and max-product composition operators coincide. We avoid this confusion by referring to both the recall process and the correlation encoding scheme as **correlation-minimum inference** when we combine correlation-minimum encoding with max-min composition, and as **correlation-product inference** when we combine correlation-product encoding with max-min composition.

We now prove the correlation-product version of the bidirectional FAM theorem.

Correlation-product bidirectional FAM theorem. If $M = A^T B$ and A and B are nonnull fit vectors, then

$$\begin{array}{lll} \text{(i)} & A \circ M = B & \text{iff } H(A) = 1 \\ \text{(ii)} & B \circ M^T = A & \text{iff } H(B) = 1 \\ \text{(iii)} & A' \circ M \subset B & \text{for any } A' \\ \text{(iv)} & B' \circ M^T \subset A & \text{for any } B' \end{array}$$

Proof.

$$\begin{aligned} A \circ M &= A \circ (A^T B) \\ &= (A \circ A^T) B \\ &= H(A) B \end{aligned}$$

Since B is not the empty set, $H(A)B = B$ iff $H(A) = 1$, establishing (i). ($A \circ M = B$ holds trivially if B is the empty set.) For an arbitrary fit vector A' in I^n:

$$\begin{aligned} A' \circ M &= (A' \circ A^T) B \\ &\subset H(A) B \\ &\subset B \end{aligned}$$

since $A' \circ A \leq H(A)$, establishing (iii). (ii) and (iv) follow similarly using $M^T = B^T A$. **Q.E.D.**

Superimposing FAM Rules

Now suppose we have m FAM rules or associations $(A_1, B_1), \ldots, (A_m, B_m)$. The fuzzy Hebb encoding scheme (8-6) leads to m FAM matrices $M_1, \ldots, M_m$ to encode the associations. The natural neural-network temptation is to add, or in this case maximum, the m matrices pointwise to distributively encode the associations in a single matrix M:

$$M = \max_{1 \leq k \leq m} M_k \tag{8-15}$$

This superimposition scheme fails for fuzzy Hebbian encoding. The superimposed result tends to be the matrix $A^T \circ B$, where A and B denote the pointwise maximum of the respective m fit vectors A_k and B_k. We can see this from the pointwise inequality

$$\max_{1 \leq k \leq m} \min(a_i^k, b_j^k) \leq \min(\max_{1 \leq k \leq m} a_i^k, \max_{1 \leq k \leq m} b_j^k) \tag{8-16}$$

Inequality (8-16) tends to hold with equality as m increases, since all maximum terms approach unity [Kosko, 1986a]. We lose the information in the m association (A_k, B_k).

The fuzzy approach to the superimposition problem *additively superimposes the m recalled vectors B'_k* instead of the fuzzy Hebb matrices M_k. B'_k and M_k correspond to

$$\begin{aligned} A \circ M_k &= A \circ (A_k^T \circ B_k) \\ &= B'_k \end{aligned}$$

for any fit-vector input A applied in parallel to the bank of FAM rules (A_k, B_k). This requires separately storing the m associations (A_k, B_k), as if each association in the FAM bank represents a separate feedforward neural network.

Separate storage of FAM associations consumes space but provides an "audit trail" of the FAM inference procedure and avoids crosstalk. The user can directly determine which FAM rules contributed how much membership activation to a "concluded" output. Separate storage also provides knowledge-base modularity. The user can add or delete FAM-structured knowledge without disturbing stored knowledge. Both of these benefits are advantages over a pure neural-network architecture for encoding the same associations (A_k, B_k). Of course we can use neural networks exogenously to estimate, or even individually house, the associations (A_k, B_k).

Separate storage of FAM rules brings out another distinction between FAM systems and neural networks. A fit-vector input A activates all the FAM rules (A_k, B_k) in parallel but to different degrees. If A only partially "satisfies" the antecedent associant A_k, the consequent associant B_k only partially activates. If A does not satisfy A_k at all, B_k does not activate at all. B'_k equals the null vector.

Neural networks behave differently. They try to reconstruct the entire association (A_k, B_k) when stimulated with A. If A and A_k mismatch severely, a neural network will tend to emit a nonnull output B'_k, perhaps the result of the network dynamical system falling into a "spurious" attractor in the state space. We may desire this for metrical classification problems, but not for inferential problems and, arguably, for associative-memory problems. When we ask an expert a question outside his field of knowledge, it may be more prudent if he gives no response than if he gives an educated guess.

Recalled Outputs and "Defuzzification"

The recalled fit-vector output B equals a weighted sum of the individual recalled vectors B'_k:

$$B = \sum_{k=1}^{m} w_k B'_k \tag{8-17}$$

where the nonnegative weight w_k summarizes the credibility or strength of the kth FAM rule (A_k, B_k). The credibility weights w_k are immediate candidates for adaptive modification. In practice we choose $w_1 = \ldots = w_m = 1$ as a default.

In principle, though not in practice, the recalled fit-vector output equals a normalized sum of the B'_k fit vectors. This keeps the components of B unit-interval valued. We do not use normalization in practice because we invariably "defuzzify" the output distribution B to produce a single numerical output, a single value in the output universe of discourse $Y = \{y_1, \ldots, y_p\}$. The information in the output waveform B resides largely in the relative values of the membership degrees.

The simplest defuzzification scheme chooses that element y_{max} that has maximal membership in the output fuzzy set B:

$$m_B(y_{max}) = \max_{1 \leq j \leq k} m_B(y_j) \tag{8-18}$$

The popular probabilistic methods of maximum-likelihood and maximum-a-posteriori parameter estimation motivate this **maximum-membership defuzzification** scheme.

The maximum-membership defuzzification scheme has two fundamental problems. First, the mode of the B distribution is not unique. This problem affects correlation-minimum encoding, as the representation (8-8) shows, more than it affects correlation-product encoding. Since the minimum operator clips off the top of the B'_k fit vectors, the additively combined output fit vector B tends to be flat over many regions of universe of discourse Y. For continuous membership functions this leads to infinitely many modes. Even for quantized fuzzy sets, there may be many modes.

In practice we can average multiple modes. For large FAM banks of "independent" FAM rules, some form of the central limit theorem (whose proof ultimately depends on Fourier transformability, not probability) tends to hold. The waveform B tends to resemble a Gaussian membership function. So a unique mode tends to emerge. It tends to emerge with fewer samples if we use correlation-product encoding.

Second, the maximum-membership scheme ignores the information in much of the waveform B. Again correlation-minimum encoding compounds the problem. In practice B is often highly asymmetric, even if it is unimodal. Infinitely many output distributions can share the same mode.

The natural alternative is the **fuzzy centroid defuzzification** scheme. We directly compute the real-valued output as a (normalized) convex combination of fit values, the *fuzzy centroid* $\bar{B}$ of fit-vector B with respect to output space Y:

$$\bar{B} = \frac{\sum_{j=1}^{p} y_j m_B(y_j)}{\sum_{j=1}^{p} m_B(y_j)} \tag{8-19}$$

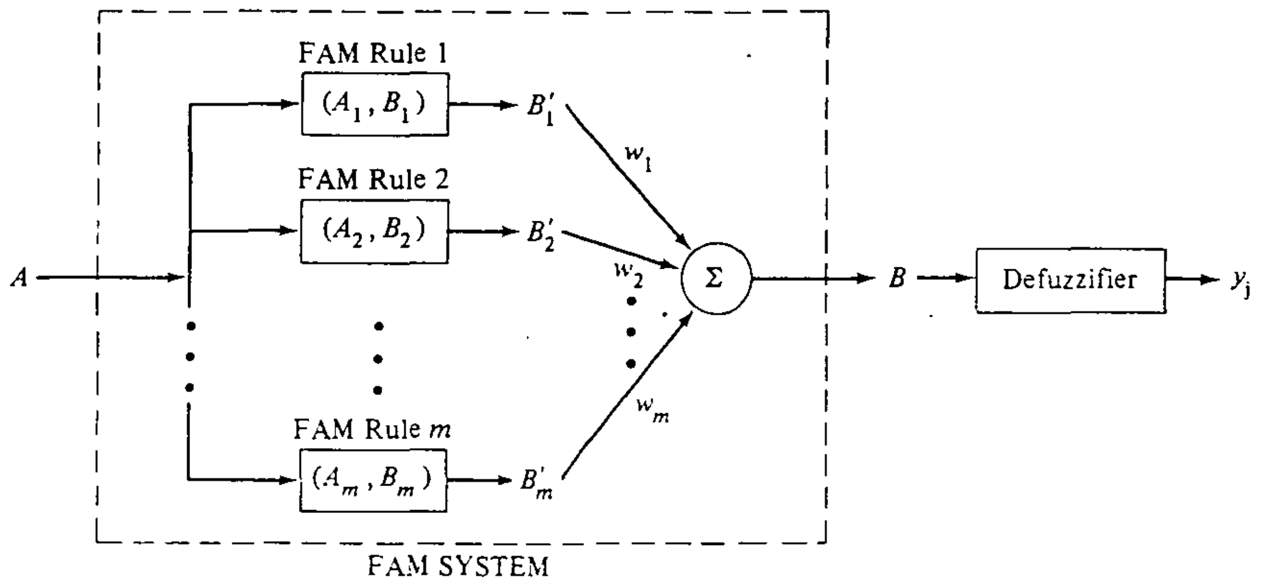


FIGURE 8.3 FAM system architecture. The FAM system F maps fuzzy sets in the unit cube I^n to fuzzy sets in the unit cube I^p. Binary input sets model exact input data. In general only an uncertainty estimate of the system state confronts the FAM system. So A is a proper fuzzy set. The user can defuzzify output fuzzy set B to yield exact output data, reducing the FAM system to a mapping between Boolean cubes.

The fuzzy centroid is unique and uses all the information in the output distribution B. For symmetric unimodal distributions the mode and fuzzy centroid coincide. In many cases we must replace the discrete sums in (8-19) with integrals over continuously infinite spaces. We show in Chapter 11, though, that for libraries of trapezoidal fuzzy-set values we can replace such a ratio of integrals with a ratio of simple discrete sums.

Computing the centroid (8-19) is the only step in the FAM inference procedure that requires division. All other operations are inner products, pairwise minima, and additions. This promises realization in a fuzzy optical processor. Already some form of this FAM-inference scheme has led to digital [Togai, 1986] and analog [Yamakawa, 1987, 1988] VLSI circuitry.

FAM System Architecture

Figure 8.3 schematizes the architecture of the nonlinear FAM system F. Note that F maps fuzzy sets to fuzzy sets: $F(A) = B$. So F defines a fuzzy-system transformation $F: I^n \to I^p$. In practice A equals a bit vector with one unity value, $a_i = 1$, and all other fit values zero, $a_j = 0$, or a delta pulse.

We defuzzify the output fuzzy set B with the centroid technique to produce an exact element y_j in the output universe of discourse Y. In effect defuzzification produces an output binary vector O, again with one element 1 and the rest 0s. At this level the FAM system F maps sets to sets, reducing the fuzzy

system F to a mapping between Boolean cubes, F: $\{0, 1\}^n \rightarrow \{0, 1\}^p$. In many applications we model X and Y as continuous universes of discourse. So n and p are quite large. We shall call such systems binary input-output FAMs.

Binary Input-Output FAMs: Inverted-Pendulum Example

Binary input-output FAMs (BIOFAMs) are the most popular fuzzy systems for applications. BIOFAMs map system state-variable data to control, classification, or other output data. In the case of traffic control, a BIOFAM maps traffic densities to green (and red) light durations.

BIOFAMs easily extend to multiple FAM-rule antecedents, to mappings from product cubes to product cubes. There has been little theoretical justification for this extension, aside from Mamdani's [1977] original suggestion to multiply relational matrices. In the next section we present a general method for dealing with multiantecedent FAM rules. First, though, we present the BIOFAM algorithm by illustrating it, and the FAM construction procedure, on a standard control problem.

Consider an inverted pendulum. We wish to adjust a motor to balance an inverted pendulum in two dimensions. The inverted pendulum is a classical control problem and admits a math-model control solution. This provides a formal benchmark for BIOFAM pendulum controllers.

There are two state fuzzy variables and one control fuzzy variable. The first state fuzzy variable is the *angle* θ that the pendulum shaft makes with the vertical. Zero angle corresponds to the vertical position. Positive angles are to the right of the vertical, negative angles to the left.

The second state fuzzy variable is the *angular velocity* $\Delta\theta$. In practice we approximate the instantaneous angular velocity $\Delta\theta$ as the difference between the present angle measurement θ_t and the previous angle measurement θ_{t-1}:

$$\Delta\theta_t \quad = \quad \theta_t - \theta_{t-1}$$

The control fuzzy variable is the *motor current* or angular velocity v_t. The velocity can be positive or negative. We expect that if the pendulum falls to the right, the motor velocity should be negative to compensate. If the pendulum falls to the left, the motor velocity should be positive. If the pendulum successfully balances at the vertical, the motor velocity should be zero.

The real line R is the universe of discourse of the three fuzzy variables. In practice we restrict each universe of discourse to a comparatively small interval, such as $[-90, 90]$ for the pendulum angle, centered about zero.

We can quantize each universe of discourse into five overlapping fuzzy-set values. We know that the fuzzy variables can be positive, zero, or negative. We can quantize the magnitudes of the fuzzy variables finely or coarsely. Suppose we

quantize the magnitudes as small, medium, and large. This leads to seven fuzzy-set values:

NL: Negative Large
NM: Negative Medium
NS: Negative Small
ZE: Zero
PS: Positive Small
PM: Positive Medium
PL: Positive Large

For example, θ is a fuzzy *variable* that takes NL as a fuzzy-set *value*. Different fuzzy quantizations of the angle universe of discourse allow the fuzzy variable θ to assume different fuzzy-set values. The expressive power of the FAM approach stems from these fuzzy-set quantizations. In one stroke we reduce system dimensions, and we describe a nonlinear numerical process with linguistic commonsense terms.

We are not concerned with the exact shape of the fuzzy sets defined on each of the three universes of discourse. In practice the quantizing fuzzy sets are usually symmetric triangles or trapezoids centered about representive values. (We can think of such sets as *fuzzy numbers*.) The set ZE may define a Gaussian curve for the pendulum angle θ, a triangle for the angular velocity $\Delta\theta$, and a trapezoid for the motor current v. But all the ZE fuzzy sets center about the numerical value zero, which will have maximum membership in the set of zero values.

How much should contiguous fuzzy sets overlap? This design issue depends on the problem at hand. Too much overlap blurs the distinction between the fuzzy-set values. Too little overlap tends to resemble bivalent control, producing excessive overshoot and undershoot. In Chapter 11 we determine experimentally the following default heuristic for ideal overlap: *Contiguous fuzzy sets in a library should overlap approximately 25 percent.*

Inverted-pendulum FAM rules are triples, such as (NM, ZE; PM). They describe how to modify the control variable for observed values of the pendulum state variables. A FAM rule associates a motor-velocity fuzzy-set value with a pendulum-angle fuzzy-set value and an angular-velocity fuzzy-set value. So we can interpret the triple (NM, ZE; PM) as the set-level implication

IF the pendulum angle θ is negative but medium
AND the angular velocity $\Delta\theta$ is about zero,
THEN the motor velocity should be positive but medium

These commonsensical FAM rules are comparatively easy to articulate in natural language. Consider a terser linguistic version of the same two-antecedent FAM rule:

$$\begin{array}{ll} \text{IF} & \theta = \text{NM} \text{ AND } \Delta\theta = \text{ZE} \\ \text{THEN} & v = \text{PM} \end{array}$$

Even this mild level of formalism may inhibit the knowledge-acquisition process.

On the other hand, the still terser FAM triple (NM, ZE; PM) allows knowledge to be acquired simply by filling in a few entries in a linguistic FAM-bank matrix. In practice this often allows us to develop a working system in minutes.

We specify the pendulum FAM system when we choose a *FAM bank* of two-antecedent FAM rules. Perhaps the first FAM rule to choose is the *steady-state FAM rule*:(ZE, ZE; ZE). The steady-state FAM rule describes what to do in equilibrium. For the inverted pendulum we should do nothing.

Many control problems require nulling a scalar error measure. We can control many multivariable problems by nulling the norms of the system error vector and error-velocity vectors, or, better, by directly nulling the individual scalar variables. (Chapter 11 shows how error nulling can control a real-time target tracking system.) Adaptive error-nulling extends the FAM methodology to nonlinear estimation, control, and decision problems of high dimension.

The pendulum FAM bank is a 7-by-7 matrix with linguistic fuzzy-set entries. We index the columns by the seven fuzzy sets that quantize the angle θ universe of discourse. We index the rows by the seven fuzzy sets that quantize the angular velocity $\Delta\theta$ universe of discourse.

Each matrix entry can equal one of seven motor-current fuzzy-set values or equal no fuzzy set at all. Since a FAM rule is a mapping or function, there is exactly one output motor-current value for every pair of angle and angular-velocity values. So the 49 entries in the FAM bank matrix represent a subset of the 343 (7^3) possible two-antecedent FAM rules. In practice most of the entries are blank. In the adaptive FAM case discussed below, we adaptively generate the entries from process sample data.

Common sense and engineering judgment dictate the entries in the pendulum FAM-bank matrix. Suppose the pendulum does not move. So $\Delta\theta$ = ZE. If the pendulum tilts to the right of vertical, the motor velocity should be negative to compensate. The farther the pendulum tilts to the right, the larger the negative motor velocity should be. The motor velocity should be positive if the pendulum tilts to the left. So the fourth row of the FAM bank matrix, which corresponds to $\Delta\theta$ = ZE, should equal the ordinal inverse of the θ row values. This assignment includes the steady-state FAM rule (ZE, ZE; ZE).

Now suppose the angle θ is zero but the pendulum moves. If the angular velocity is negative, the pendulum will overshoot to the left. So the motor velocity should be positive to compensate. If the angular velocity is positive, the motor velocity should be negative. The greater the angular velocity is in magnitude, the greater the motor velocity should be in magnitude. So the fourth column of the FAM-bank matrix, which corresponds to θ = ZE, should equal the ordinal inverse of the $\Delta\theta$ column values. This assignment also includes the steady-state FAM rule.

Positive θ values with negative $\Delta\theta$ values should produce negative motor-current values, since the pendulum heads toward the vertical. So (PS, NS; NS) is a candidate FAM rule. Symmetrically, negative θ values with positive $\Delta\theta$ values should produce positive motor-current values. So (NS, PS; PS) is another candidate FAM rule.

This gives 15 FAM rules altogether. In practice these rules can successfully balance an inverted pendulum. Different, and smaller, subsets of FAM rules can also balance the pendulum. The software problems at the end of the chapter explore these cases.

We can represent the bank of 15 FAM rules as the 7-by-7 linguistic matrix

$\Delta\theta$ \ θ	NL	NM	NS	ZE	PS	PM	PL
NL				PL			
NM				PM			
NS				PS	NS		
ZE	PL	PM	PS	ZE	NS	NM	NL
PS			PS	NS			
PM				NM			
PL				NL			

The BIOFAM system F admits a geometric interpretation. The set of all possible input-outpairs $(\theta, \Delta\theta, F(\theta, \Delta\theta))$ defines a *FAM surface* in the input-output product space, in this case in R^3. We plot examples of these control surfaces in Chapters 9, 10 and 11.

The BIOFAM *inference procedure* activates in parallel the antecedents of all 15 FAM rules. The binary or pulse nature of inputs picks off single fit values from the quantizing fuzzy-set values of the fuzzy variables. We can use either the correlation-minimum or correlation-product inferencing technique. For simplicity we shall illustrate the procedure with correlation-minimum inferencing.

Suppose the current pendulum angle θ equals 15 degrees and the angular velocity $\Delta\theta$ equals -10. This amounts to passing two bit vectors of one 1 and all else 0 through the BIOFAM system. What is the corresponding motor-current value $v = F(15, -10)$?

Consider first how the input data pair $(15, -10)$ activates the steady-state FAM rule (ZE, ZE; ZE). Suppose we define the antecedent and consequent fuzzy sets for ZE with the triangular fuzzy-set membership functions in Figure 8.4. Then the angle datum 15 defines a zero angle value to degree .2: $m^{\theta}_{\text{ZE}}(15) = .2$. The angular-velocity datum -10 defines a zero angular-velocity value to degree .5 : $m^{\Delta\theta}_{\text{ZE}}(-10) = .5$.

We combine the antecedent fit values with minimum or maximum depending on whether we combine the antecedent fuzzy sets with the conjunctive AND or the disjunctive OR. Intuitively, it should be at least as difficult to satisfy both antecedent conditions as to satisfy either one separately.

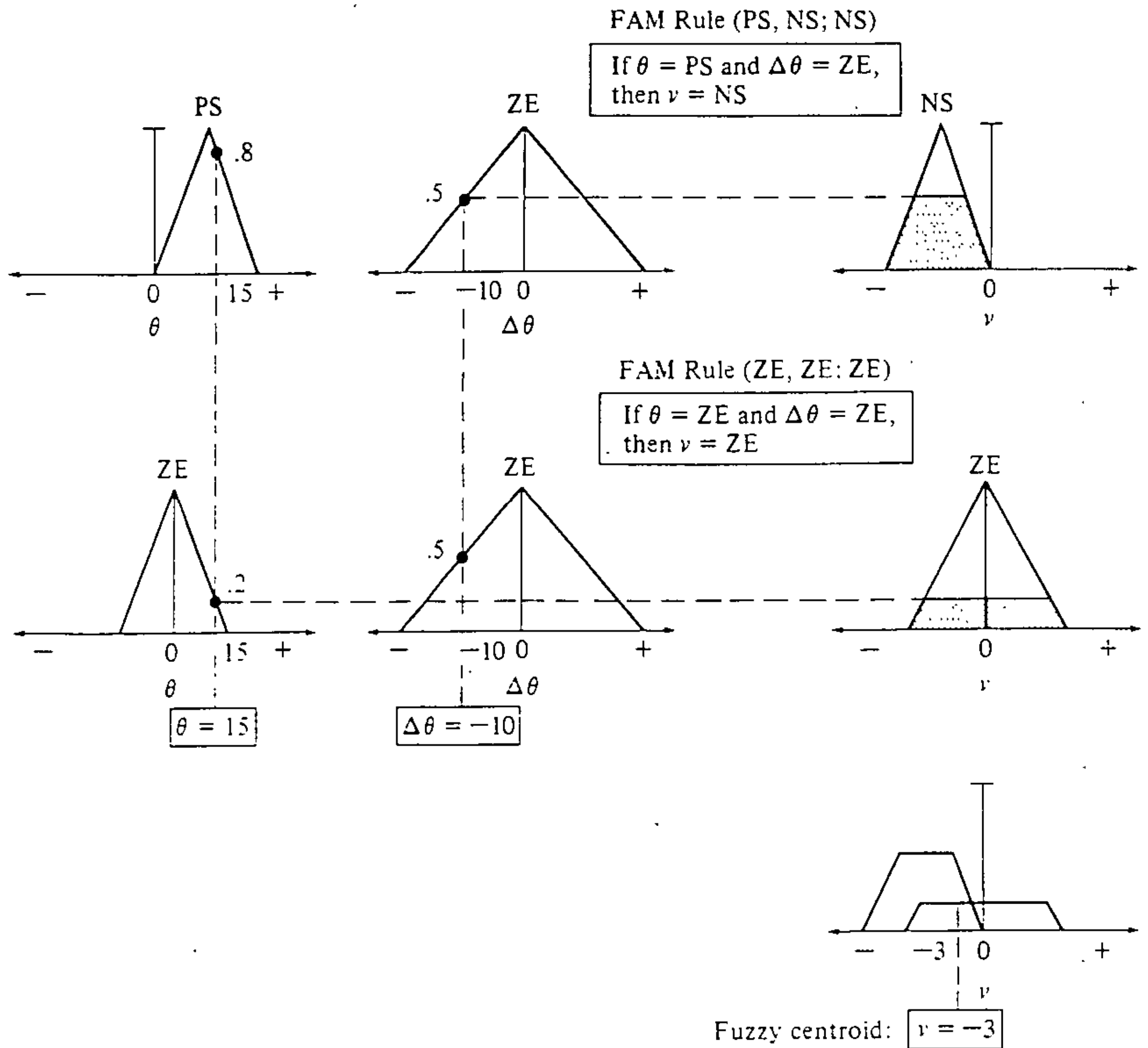


FIGURE 8.4 FAM correlation-minimum inference procedure. The FAM system consists of the two two-antecedent FAM rules (PS. ZE: NS) and (ZE. ZE: ZE) The input angle datum equals 15 and is more a small but positive angle value than a zero angle value. The input angular-velocity datum equals −10. and is a zero angular-velocity value only to degree .5. The system combines antecedent fit values with minimum, since the conjunction AND combines the antecedent terms. The combined fit value then scales the consequent fuzzy set with pairwise minimum. The system adds the minimum-scaled output fuzzy sets and computes the fuzzy centroid of this output waveform This yields the system output-current value −3.

The FAM-rule notation (ZE. ZE: ZE) implicitly assumes that we combine antecedent fuzzy sets conjunctively with AND. So the data satisfy the compound antecedent of the FAM rule $(ZE, ZE; ZE)$ to degree

$$\begin{aligned}\min(m^{\theta}_{ZE}(15),\ m^{\Delta\theta}_{ZE}(-10)) &= \min(.2,\ .5)\\ &= .2\end{aligned}$$

This methodology extends to any number of antecedent terms connected with arbitrary logical (set-theoretical) connectives.

The system should now activate the consequent fuzzy set of zero-motor-current values to degree .2. This differs from activating the ZE motor-current fuzzy set 100 percent with probability .2, and certainly differs from Prob $\{v = 0\} = .2$. Instead a deterministic 20 percent of ZE should result and, according to the additive combination formula (8-17), we should add this truncated fuzzy set to the final output fuzzy set.

The correlation-minimum inference procedure activates the angular-velocity fuzzy set ZE to degree .2 by taking the pairwise minimum of .2 and the ZE fuzzy set m^v_{ZE}:

$$\min(m^{\theta}_{ZE}(15), m^{\Delta\theta}_{ZE}(-10)) \wedge m^v_{ZE}(v) \quad = \quad .2 \wedge m^v_{ZE}(v)$$

for all velocity values v. The correlation-product inference procedure would multiply the zero-angular-velocity fuzzy set by .2: $.2m^v_{ZE}(v)$ for all v.

The data similarly activate the FAM rule (PS, ZE; NS) depicted in Figure 8.4. The angle datum 15 is a small but positive angle value to degree .8. The angular-velocity datum -10 is a zero-angular-velocity value to degree .5. So we scale the output motor-velocity fuzzy set of small but negative motor-velocity values by .5, the lesser of the two antecedent fit values:

$$\min(m^{\theta}_{PS}(15), m^{\Delta\theta}_{ZE}(-10)) \wedge m^v_{NS}(v) \quad = \quad .5 \wedge m^v_{NS}(v)$$

for all velocity values v. So the data activate the FAM rule (PS. ZE: NS) to greater degree than it activates the steady-state FAM rule (ZE, ZE; ZE) since in this example an angle value of 15 degrees is more a small but positive angle value than a zero angle value.

The data similarly activate the other 13 FAM rules. We combine the resulting minimum-scaled consequent fuzzy sets according to (8-17) by summing pointwise. We can then compute the fuzzy centroid with Equation (8-19), with perhaps integrals replacing the discrete sums, to determine the specific output motor velocity v. In Chapter 11 we show that, for symmetric fuzzy-set values of fuzzy variables, we can always compute the centroid exactly with simple discrete sums even if the fuzzy sets are continuous. In many real-time applications we must repeat this entire FAM inference procedure hundreds, perhaps thousands, of times per second. This may require fuzzy VLSI or optical processors.

Figure 8.4 illustrates the equal-weight additive combination procedure for just the FAM rules (ZE. ZE: ZE) and (PS. ZE: NS). In this case the fuzzy-centroidal motor-velocity value equals -3.

Multiantecedent FAM Rules: Decompositional Inference

BIOFAM inference treats antecedent fuzzy sets as propositions with fuzzy truth values. This holds because fuzzy logic corresponds to one-dimensional fuzzy-set

theory and because we use binary (or delta-pulse) or exact inputs. We now formally develop the connection between BIOFAMs and the FAM theory presented earlier.

Consider the compound FAM rule "IF X is A AND Y is B, THEN C is Z," or $(A, B; C)$ for short. Let the universes of discourse X, Y, and Z have dimensions n, p, and q: $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_p\}$, and $Z = \{z_1, \ldots, z_q\}$. We can directly extend this framework to multiple antecedent and consequent terms. Continuously infinite universes of discourse require a delta-pulse formulation.

In our notation X, Y, and Z double as universes of discourse and fuzzy variables. The fuzzy *variable* X can assume the fuzzy-set *values* $A_1, A_2, \ldots$, and similarly for the fuzzy variables Y and Z. When controlling an inverted pendulum, the identification "X is A" might represent the natural-language description "The pendulum angle is positive but small."

What matrix represents the FAM rule $(A, B; C)$? The question is nontrivial, since A, B, and C are fuzzy subsets of different universes of discourse, points in different unit cubes (fuzzy power sets). Their dimensions and interpretations differ. Mamdani [1977] and others have suggested representing such rules as fuzzy multidimensional relations or arrays. Then the FAM rule $(A, B; C)$ would define a fuzzy subset of the product space $X \times Y \times Z$. Engineers do not use this representation in practice, since actual systems present only exact inputs or measurements to FAM systems, and the BIOFAM procedure applies. If we presented the system with a genuine fuzzy-set input, we could preprocess the fuzzy set with a centroidal or maximum-fit-value technique, so we could still apply the BIOFAM inference procedure.

We present an alternative representation that decomposes, then recomposes, the FAM rule $(A, B; C)$ in accord with the FAM inference procedure. This representation allows neural networks to adaptively estimate, store, and modify the decomposed FAM rules. The representation requires far less storage than the multidimensional-array representation.

Let the fuzzy Hebb matrices M_{AC} and M_{BC} store the simple FAM associations (A, C) and (B, C):

$$M_{AC} = A^T \circ C \qquad (8\text{-}20)$$

$$M_{BC} = B^T \circ C \qquad (8\text{-}21)$$

The fuzzy Hebb matrices M_{AC} and M_{BC} *split* the compound FAM rule $(A, B; C)$. We can construct the *splitting matrices* with correlation-product encoding.

Let $I_X^i = (0 \ \ldots \ 0 \ 1 \ 0 \ \ldots \ 0)$ denote an n-dimensional bit vector with ith element 1 and all other elements 0. (In the continuous case I_i^j denotes a delta pulse in a convolution integral.) I_X^i equals the ith row of the n-by-n identity matrix. Similarly, I_Y^j and I_Z^k equal the respective jth and kth rows of the p-by-p and q-by-q identity matrices. The bit vector I_X^i represents the occurrence of the exact input x_i.

We will call the proposed FAM representation scheme **FAM decompositional inference**, in the spirit of the max-min compositional inference scheme dis-

100

cussed above. FAM decompositional inference *decomposes* the compound FAM rule $(A, B; C)$ into the component rules (A, C) and (B, C). The BIOFAM processes the simpler component rules in parallel. New fuzzy-set inputs A' and B' pass through the FAM matrices M_{AC} and M_{BC}. Max-min composition then gives the recalled fuzzy sets $C_{A'}$ and $C_{B'}$:

$$C_{A'} = A' \circ M_{AC} \tag{8-22}$$

$$C_{B'} = B' \circ M_{BC} \tag{8-23}$$

The trick is to *recompose* the fuzzy sets $C_{A'}$ and $C_{B'}$ with intersection or union depending on whether we combine the antecedent terms "X is A" and "Y is B" with AND or OR. The negated antecedent term "X is NOT A" requires forming the set complement $C^c_{A'}$ for input fuzzy set A'.

Suppose we present the new inputs A' and B' to the single-FAM-rule system F that stores the FAM rule $(A, B; C)$. Then the recalled output fuzzy set C' equals the intersection of $C_{A'}$ and $C_{B'}$:

$$\begin{aligned} F(A', B') &= [A' \circ M_{AC}] \cap [B' \circ M_{BC}] \\ &= C_{A'} \cap C_{B'} \\ &= C' \end{aligned} \tag{8-24}$$

We can then defuzzify C' to yield the exact output I_Z^k.

Logical connectives apply to antecedent terms of different dimension and meaning. Decompositional inference applies the set-theoretic analogues of the logical connectives to subsets of Z. All subsets C' of Z have the same dimension and meaning.

We now prove that decompositional inference generalizes BIOFAM inference. This generalization is not simply formal. It opens an immediate path to adaptation with arbitrary neural-network techniques.

Suppose we present the exact inputs x_i and y_j to the single-FAM-rule system F that stores $(A, B; C)$. So we present the unit bit vectors I_X^i and I_Y^j to F as nonfuzzy set inputs. Then

$$F(x_i, y_j) = F(I_X^i, I_Y^j) = [I_X^i \circ M_{AC}] \cap [I_Y^j \circ M_{BC}]$$

$$= a_i \wedge C \cap b_j \wedge C \tag{8-25}$$

$$= \min(a_i, b_j) \wedge C \tag{8-26}$$

Equation (8-25) follows from (8-8). Representing C with its membership function m_C, (8-26) corresponds to the BIOFAM prescription

$$\min(a_i, b_j) \wedge m_C(z) \tag{8-27}$$

for all z in Z.

If we encode the simple FAM rules (A, C) and (B, C) with correlation-product encoding, decompositional inference gives the BIOFAM version of correlation-product inference:

$$
\begin{aligned}
F(I_X^i, I_Y^j) &= [I_X^i \circ A^T C] \cap [I_Y^j \circ B^T C] \\
&= a_i C \cap b_j C && \text{(8-28)} \\
&= \min(a_i, b_j) C && \text{(8-29)} \\
&= \min(a_i, b_j) m_C(z) && \text{(8-30)}
\end{aligned}
$$

for all z in Z. Equation (8-13) implies (8-28). $\min(a_i c_k, b_j c_k) = \min(a_i, b_j) c_k$ implies (8-29).

Decompositional inference allows arbitrary fuzzy sets, waveforms, or distributions A' and B' to pass through a FAM system. The FAM system can house an arbitrary FAM bank of compound FAM rules. If we use the FAM system to control a process, the input fuzzy sets A' and B' can equal the output of an independent state-*estimation* system, such as a Kalman filter. A' and B' might then represent probability distributions on the exact input spaces X and Y. The filter-controller cascade is a common engineering architecture.

We can split compound consequents as desired. We can split the compound FAM rule "IF X is A AND Y is B, THEN Z is C OR W is D," or $(A, B; C, D)$, into the FAM rules $(A, B; C)$ and $(A, B; D)$. We can use the same split for the consequent logical-connective AND.

We can give a propositional-calculus justification for the decompositional inference technique. Let A, B, and C denote bivalent *propositions* with truth values $t(A)$, $t(B)$, and $t(C)$ in $\{0, 1\}$. Then truth tables prove the two consequent-splitting tautologies used in decompositional inference:

$$[A \rightarrow (B \text{ OR } C)] \rightarrow [(A \rightarrow B) \text{ OR } (A \rightarrow C)] \qquad \text{(8-31)}$$

$$[A \rightarrow (B \text{ AND } C)] \rightarrow [(A \rightarrow B) \text{ AND } (A \rightarrow C)] \qquad \text{(8-32)}$$

where the arrow denotes logical implication.

In bivalent logic, the implication $A \rightarrow B$ is false iff the antecedent A is true and the consequent B is false. Equivalently, $t(A \rightarrow B) = 1$ iff $t(A) = 1$ and $t(B) = 0$. This allows a "brief" truth-table check for validity. We implicitly chose truth values for the terms in the consequent of the overall implication (8-31) or (8-32) to make the consequent false. Given those restrictions, if we cannot find truth values to make the antecedent true, the statement is a tautology. In Equation (8-31), if $t((A \rightarrow B) \text{ OR } (A \rightarrow C)) = 0$, then $t(A) = 1$ and $t(B) = t(C) = 0$, since a disjunction is false iff both disjuncts are false. This forces the antecedent $A \rightarrow (B \text{ OR } C)$ to be false. So Equation (8-31) is a **tautology**: a statement is true in all cases.

A propositional tautology also justifies splitting the compound FAM rule "IF X is A OR Y is B, THEN Z is C" into the disjunction (union) of the two simple FAM rules "IF X is A, THEN Z is C" and "IF Y is B, THEN Z is C":

$$[(A \text{ OR } B) \longrightarrow C] \quad \longrightarrow \quad [(A \longrightarrow C) \text{ OR } (B \longrightarrow C)] \tag{8-33}$$

Now consider splitting the original compound FAM rule "IF X is A AND Y is B, THEN Z is C" into the conjunction (intersection) of the two simple FAM rules "IF X is A, THEN Z is C" and "IF Y is B, THEN Z is C." A problem arises in the truth table of the corresponding proposition

$$[(A \text{ AND } B) \longrightarrow C] \quad \longrightarrow \quad [(A \longrightarrow C) \text{ AND } (B \longrightarrow C)] \tag{8-34}$$

Proposition (8-34) is not always true, and hence not a tautology. The implication is false if A is true and if B and C are false, or if A and C are false and if B is true. But the implication (8-34) is valid if *both* antecedent terms A and B are true. So if $t(A) = t(B) = 1$, the compound conditional $(A \text{ AND } B) \rightarrow C$ implies both $A \rightarrow C$ and $B \rightarrow C$.

The simultaneous occurrence of the data values x_i and y_j satisfies this condition. Recall that logic is one-dimensional set theory. The condition $t(A) = t(B) = 1$ arises from the 1 in I_X^i and the 1 in I_Y^j. We can interpret the unit bit vectors I_X^i and I_Y^j as the (true) bivalent propositions "X is x_i" and "Y is y_j." Propositional logic applies coordinatewise. A similar argument holds for the converse of (8-33)

For general fuzzy-set inputs A' and B' the argument still holds in the sense of continuous-valued logic. But the truth values of the logical implications may be less than unity while greater than zero. If A' is a null vector and B' is not, or vice versa, the implication (8-34) is false coordinatewise, at least if one coordinate of the nonnull vector equals unity. But in this case the decompositional inference scheme yields an output null vector C'. In effect the FAM system indicates the propositional falsehood.

Adaptive Decompositional Inference

The decompositional-inference scheme allows arbitrary splitting matrices M_{AC} and M_{BC}. Indeed it allows us to eliminate them altogether.

Let $N_X: I^n \rightarrow I^q$ define an arbitrary *neural-network* system that maps fuzzy subsets A' of X to fuzzy subsets C' of Z. $N_Y: I^p \rightarrow I^q$ can define a different neural network. In general N_X and N_Y chage with time.

The adaptive decompositional inference (ADI) scheme allows neural networks to adaptively split, store, and modify compound FAM rules. We can split the compound FAM rule "IF X is A AND Y is B, THEN Z is C," or $(A, B; C)$, with N_X and N_Y. N_X can house the simple FAM association (A, C). N_Y can house (B, C). Then for arbitrary fuzzy-set inputs A' and B', ADI proceeds as before for

an adaptive FAM system F: $I^n \times I^p \rightarrow I^q$ that houses the FAM rule $(A, B; C)$ or a bank of such FAM rules:

$$\begin{aligned} F(A', B') &= N_X(A') \cap N_Y(B') \\ &= C_{A'} \cap C_{B'} \\ &= C' \end{aligned} \tag{8-35}$$

Any neural network can define the system operation. The backpropagation algorithm, discussed in Chapter 5, provides a reasonable candidate for many unstructured problems. The primary concerns are memory space and training time. We can often train several small neural networks in parallel faster, and more accurately, than we can train a single large neural network.

The ADI approach illustrates one way neural algorithms embed in a FAM architecture. Below we introduce a more practical way that uses unsupervised clustering algorithms.

ADAPTIVE FAMS: PRODUCT-SPACE CLUSTERING IN FAM CELLS

An **adaptive FAM (AFAM)** is a time-varying mapping between fuzzy cubes. In principle the adaptive decompositional-inference technique generates AFAMs. But we shall reserve the label AFAM for systems that generate FAM rules from training data but that do not require splitting and recombining FAM data.

We propose a geometric AFAM procedure. The procedure adaptively clusters training samples in the FAM system *input-output product space*. FAM mappings define balls or clusters in the input-output product space. These clusters correspond to fuzzy Hebb matrices, which define fuzzy Cartesian products. The procedure "blindly" generates weighted FAM rules from training data. Further training modifies the weighted set of FAM rules. We call this unsupervised procedure **product-space clustering**.

Consider first a discrete one-dimensional FAM system S: $I^n \rightarrow I^p$. Then a FAM rule has the form "IF X is A_i, THEN Y is B_i," or (A_i, B_i). The input-output product space is $I^n \times I^p$.

What does the FAM rule (A_i, B_i) look like in the product space $I^n \times I^p$? It looks like a cluster of points centered at the numerical point (A_i, B_i). The FAM system maps points A near A_i to points B near B_i. The closer A is to A_i, the closer the point (A, B) is to the point (A_i, B_i) in the product space $I^n \times I^p$. In this sense FAMs map balls in I^n to balls in I^p. The notation is ambiguous, since (A_i, B_i) stands for both the FAM-rule mapping, or fuzzy subset (Cartesian product) of $I^n \times I^p$, and the numerical fit-vector point in $I^n \times I^p$.

10C

Adaptive clustering algorithms can estimate the unknown FAM rule (A_i, B_i) from training samples of the form (A, B). In general there are m unknown FAM rules $(A_1, B_1), \ldots, (A_m, B_m)$, and we do not know m. The user may select m arbitrarily in many applications.

Competitive *adaptive vector-quantization* (AVQ) algorithms can adaptively estimate both the unknown FAM rules (A_i, B_i) and the unknown number m of FAM rules from FAM system input-output data. The AVQ algorithms do not require fuzzy-set data. Scalar BIOFAM data suffices, as we illustrate below for adaptive estimation of inverted-pendulum control FAM rules.

Suppose the r fuzzy sets $A_1, \ldots, A_r$ quantize the input universe of discourse X. The s fuzzy sets $B_1, \ldots, B_s$ quantize the output universe of discourse Y. If we double up on notation and view X and Y as fuzzy variables as well, then fuzzy variable X assumes fuzzy-set values A_i, and fuzzy variable Y assumes fuzzy-set values B_j. In general r and s do not relate to each other or to the number m of FAM rules (A_i, B_i). The user must specify r and s and the shape of the fuzzy sets A_i and B_i. In practice this is not difficult. Quantizing fuzzy sets are usually trapezoidal or triangular, and r and s are less than 10.

The quantizing collections $\{A_i\}$ and $\{B_j\}$ define rs **FAM cells** F_{ij} in the input-output product space $I^n \times I^p$. The FAM cells F_{ij} overlap, since contiguous quantizing fuzzy sets A_i and A_{i-1}, and B_j and B_{j+1}, overlap. So the FAM cell collection $\{F_{ij}\}$ does not partition the product space $I^n \times I^p$. The union of all FAM cells also does not equal $I^n \times I^p$, since the patches F_{ij} are fuzzy subsets of $I^n \times I^p$. The union provides only a fuzzy "cover" for $I^n \times I^p$.

The *fuzzy Cartesian product* $A_i \times B_i$ defines the FAM cell F_{ij}. $A_i \times B_i$ equals the fuzzy outer product $A_i^T \circ B_i$ in (8-6) or the correlation product $A_i^T B_i$ in (8-12). A FAM cell F_{ij} corresponds to the fuzzy correlation-minimum or correlation-product matrix M_{ij} : $F_{ij} = M_{ij}$. Thus we connect fuzzy geometry with fuzzy algebra and arrive at the following equalities: product-space cluster = FAM cell = $A_i \times B_j$ = M_{ij} = FAM rule (A_i, B_j).

Adaptive FAM-Rule Generation

Let $\mathbf{m}_1, \ldots, \mathbf{m}_k$ denote k quantization vectors in the input-output product space $I^n \times I^p$ or, equivalently, in I^{n+p}. $\mathbf{m}_j$ defines the jth column of the synaptic connection matrix $\mathbf{M}$. $\mathbf{M}$ has $n + p$ rows and k columns.

Suppose, for instance, $\mathbf{m}_j$ changes in time according to the differential competitive learning (DCL) AVQ algorithm discussed in Chapters 4 and 6, and in Chapter 1 of the companion volume [Kosko, 1991]. The competitive system samples concatenated fuzzy-set samples of the form $[A \mid B]$. The augmented fuzzy set $[A \mid B]$ is a point in the unit hypercube I^{n+p}.

The synaptic vectors $\mathbf{m}_j$ converge to FAM-matrix centroids in $I^n \times I^p$. More generally, they estimate the density or distribution of the FAM rules in $I^n \times I^p$. The quantizing synaptic vectors naturally weight an estimated FAM rule. The more

synaptic vectors clustered about a centroidal FAM rule, the greater its weight w_i in (8-17).

Suppose there are 15 FAM-rule centroids in $I^n \times I^p$ and $k > 15$. Suppose k_i synaptic vectors $\mathbf{m}_j$ cluster around the ith centroid. So $k_1 + \cdots + k_{15} = k$. Suppose the *cluster counts* k_i obey

$$k_1 \geq k_2 \geq \cdots \geq k_{15} \tag{8-36}$$

The first centroidal FAM rule is at least as frequent as the second centroidal FAM rule, and so on. This gives the adaptive FAM-rule weighting scheme

$$w_i = \frac{k_i}{k} \tag{8-37}$$

The FAM rule weights w_i evolve in time as the FAM system samples new augmented fuzzy sets $[A \mid B]$. In practice we may want only the 15 most-frequent FAM rules or only the FAM rules with at least some minimum frequency $w_{\min}$. Then (8-37) provides a quantitative solution.

We count the number k_{ij} of quantizing vectors in each FAM cell F_{ij}. We can define FAM-cell boundaries in advance or even designate some FAM cells as always sufficiently occupied. High-count FAM cells outrank low-count FAM cells. Most FAM cells contain zero or few synaptic vectors.

Product-space clustering extends to compound FAM rules and product spaces. The FAM rule "IF X is A AND Y is B, THEN Z is C," or $(A, B; C)$, defines a point in $I^n \times I^p \times I^q$. The t fuzzy sets $C_1, \ldots, C_t$ quantize the output space Z. There are rst FAM cells F_{ijk}. Equations (8-36) and (8-37) extend similarly and X, Y, and Z can be continuous. The adaptive clustering procedure extends to any number of FAM-rule antecedent terms.

Adaptive BIOFAM Clustering

BIOFAM data clusters more efficiently than fuzzy-set FAM data. We can more easily obtain and process paired numbers than we can obtain and process paired fit vectors. This allows system input-output data to directly generate FAM systems.

In control applications, human or automatic controllers generate streams of "well-controlled" system input-output data. Adaptive BIOFAM clustering converts this data to weighted FAM rules. The adaptive system transduces behavioral data to behavioral rules. The fuzzy system learns causal patterns. It learns which control inputs cause which control outputs. The system approximates these causal patterns when it acts as the controller.

Adaptive BIOFAMs cluster in the input-output product space $X \times Y$. The product space $X \times Y$ is vastly smaller than the power-set product space $I^n \times I^p$ used above. The adaptive synaptic vectors $\mathbf{m}_j$ are now two-dimensional instead of $(n+p)$-dimensional. On the other hand, competitive BIOFAM clustering requires many more input-output data pairs $(x_i, y_i) \in R^2$ than augmented fuzzy-set samples $[A \mid B] \in I^{n+p}$.

Again we double up on notation. We now use x_i as the numerical sample from X at sample time i. Earlier x_i denoted the ith ordered element in the finite nonfuzzy set $X = \{x_1, \ldots, x_n\}$. So now X can denote a continuous set, say R^n, usually R.

BIOFAM clustering counts synaptic quantization vectors in FAM cells. The system samples the nonfuzzy input-output stream $(x_1, y_1), (x_2, y_2), \ldots$. Unsupervised competitive learning distributes the k synaptic quantization vectors $\mathbf{m}_1, \ldots, \mathbf{m}_k$ in $X \times Y$. Learning distributes them to different FAM cells F_{ij}. The FAM cells F_{ij} overlap but are nonfuzzy subcubes of $X \times Y$. The BIOFAM FAM cells F_{ij} cover $X \times Y$. The key idea is *cluster equals rule.*

F_{ij} contains k_{ij} quantization vectors at each sample time. The cell counts k_{ij} define a frequency *histogram*, since all k_{ij} sum to k. So $w_{ij} = k_{ij}/k$ weights the FAM rule "IF X is A_i, THEN Y is B_j."

Suppose the overlapping fuzzy sets NL, NM, NS, ZE, PS, PM, PL quantize the input space X. Suppose seven similar fuzzy sets quantize the output space Y. We can define the fuzzy sets arbitrarily. In practice they are symmetric trapezoids or triangles. (The boundary fuzzy sets NL and PL are ramp functions or clipped trapezoids.) X and Y may each equal the real line. A typical FAM rule is "IF X is NL, THEN Y is PS," or (NL; PS).

Input datum x_i is nonfuzzy. When $X = x_i$ holds, the relations $X =$ NL, ..., $X =$ PL hold to different degrees. Most hold to zero degree. $X =$ NM holds to degree $m_{NM}(x_i)$. Input datum x_i partially activates the FAM rule "IF X is NM, THEN Y is ZE" or, equivalently, (NM; ZE). Since the FAM rules have single antecedents, x_i activates the consequent fuzzy set ZE to degree $m_{NM}(x_i)$ as well. Multiantecedent FAM rules activate output consequent sets according to a logic-based function of antecedent term membership values, as discussed above on BIOFAM inference.

Suppose that Figure 8.5 represents the input-output data stream (x_1, y_1), $(x_2, y_2), \ldots$ in the planar product space $X \times Y$, and that the sample data in Figure 8.5 trains a DCL system. Also suppose competitive learning distributes ten two-dimensional synaptic vectors $\mathbf{m}_1, \ldots, \mathbf{m}_{10}$ as in Figure 8.6. (We can use other types of learning as well.)

FAM cells do not overlap in Figure 8.5 and Figure 8.6 for convenience's sake. The corresponding quantizing fuzzy sets touch but do not overlap. The FAM cells partition the input-output product space.

Figure 8.5 reveals six sample-data clusters. The six quantization-vector clusters in Figure 8.6 estimate the six sample-data clusters. The single synaptic vector in FAM cell (PM; NS) indicates a smaller cluster. Since $k = 10$, the number of quantization vectors in each FAM cell measures the percentage or frequency weight w_{ij} of each possible FAM rule.

In general the additive combination rule (8-17) does not require normalizing the quantization-vector count k_{ij}. $w_{ij} = k_{ij}$ is acceptable. This holds for both maximum-membership defuzzification (8-18) and fuzzy-centroid defuzzification (8-19).

The ten quantization vectors in Figure 8.6 estimate at most six FAM rules.

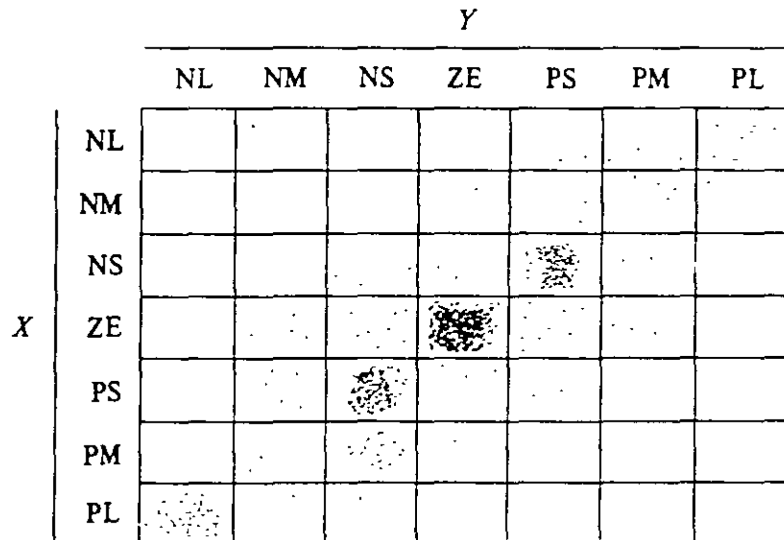


FIGURE 8.5 Distribution of input-output data (x_i, y_i) in the input-output product space $X \times Y$. Data clusters reflect FAM rules, such as the steady-state FAM rule "IF X is ZE, THEN Y is ZE."

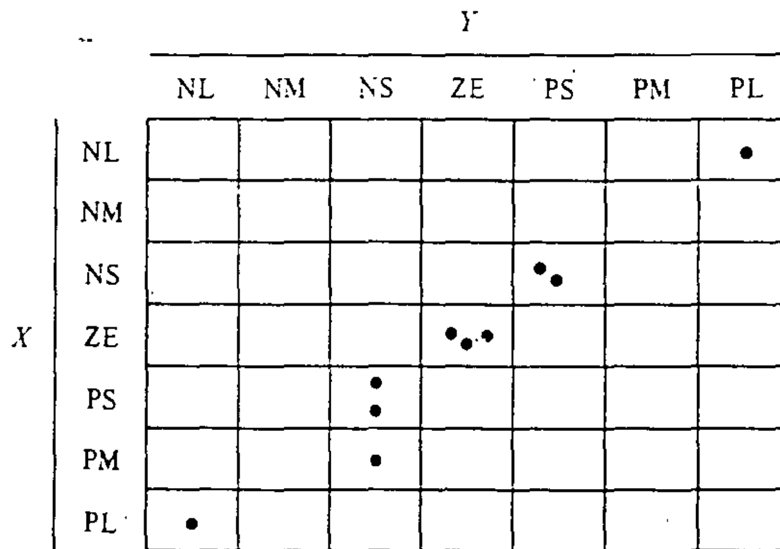


FIGURE 8.6 Distribution of ten two-dimensional synaptic quantization vectors $\mathbf{m}_1, \ldots, \mathbf{m}_{10}$ in the input-output product space $X \times Y$. As the FAM system samples nonfuzzy data (x_i, y_i), competitive learning distributes the synaptic vectors in $X \times Y$. The synaptic vectors estimate the frequency distribution of the sampled input-output data, and thus estimate FAM rules. Clusters define rules.

From most to least frequent or "important," the FAM rules are (ZE: ZE), (PS: NS), (NS: PS), (PM: NS), (PL: NL), and (NL: PL). These FAM rules suggest that fuzzy variable X behaves as an error variable or an error-velocity variable, since the steady-state FAM rule (ZE: ZE) is most important. If we sample a system only in steady-state equilibrium, we will estimate only the steady-state FAM rule. We can accurately estimate the FAM system's global behavior only if we representatively sample the system's input-output behavior. Experts must exhibit their expertise in problem cases.

The "corner" FAM rules (PL; NL) and (NL; PL) may be more important than their frequencies suggest. The boundary sets Negative Large (NL) and Positive Large (PL) usually define ramp functions as negatively and positively sloped lines. NL and PL alone cover the important end-point regions of the universe of discourse X. They give $m_{NL}(x) = m_{PL}(x) = 1$ only if x falls at or near the end point of X, since NL and PL are ramp functions, not trapezoids. NL and PL cover these end-point regions "briefly." Their corresponding FAM cells tend to be smaller than the other FAM cells. The end-point regions must be covered in most control problems, especially error-nulling problems like stabilizing an inverted pendulum. The user can weight these FAM-cell counts more highly, for instance $w_{ij} = ck_{ij}$ for scaling constant $c > 0$. Or the user can simply include these end-point FAM rules in every operative FAM bank.

Most FAM cells do not generate FAM rules, for we estimate every possible FAM rule but usually with zero or near-zero frequency weight w_{ij}. For large numbers of multiple FAM-rule antecedents, system input-output data streams through comparatively few FAM cells. Structured trajectories in $X \times Y$ may be few.

A FAM-rule's mapping structure also limits the number of estimated FAM rules. A FAM rule maps fuzzy sets in I^n or $F(2^X)$ to fuzzy sets in I^p or $F(2^Y)$. A fuzzy associative memory maps every domain fuzzy set A to a unique range fuzzy set B. Fuzzy set A cannot map to multiple fuzzy sets B, B', B'', and so on. We write the FAM rule as $(A; B)$ not $(A; B$ or B' or B'' or $\ldots)$. So we estimate *at most* one rule per FAM-cell row in Figure 8.6.

If two FAM cells in a row are equally and highly frequent, we can pick arbitrarily either FAM rule to include in the FAM bank. This occurs infrequently but can occur. In principle we could estimate the FAM rule as a compound FAM rule with a disjunctive consequent. The simplest strategy picks only the highest-frequency FAM cell per row.

The user can estimate FAM rules without counting the quantization vectors in each FAM cell. There may be too many FAM cells to search at each estimation iteration. The user never need examine FAM cells. Instead the user checks the synaptic-vector components m_{ij}. The user defines in advance fuzzy-set intervals, such as $[l_{NL}, u_{NL}]$ for NL. If $l_{NL} \leq m_{ij} \leq u_{NL}$, then the FAM-antecedent reads "IF X is NL."

Suppose the input and output spaces X and Y are the same, the real interval $[-35, 35]$. Suppose we partition X and Y into the same seven disjoint fuzzy sets:

$$
\begin{aligned}
\text{NL} &= [-35, -25] \\
\text{NM} &= [-25, -15] \\
\text{NS} &= [-15, -5] \\
\text{ZE} &= [-5, 5] \\
\text{PS} &= [5, 15] \\
\text{PM} &= [15, 25] \\
\text{PL} &= [25, 35]
\end{aligned}
$$

104

$\Delta\theta$ \ θ	NM	NS	Z	PS	PM
NM			PM		
NS			PS	Z	
Z	PM	PS	Z	NS	NM
PS		Z	NS		
PM			NM		

FIGURE 8.7 Inverted-pendulum FAM bank used in simulation. This BIOFAM generated 1000 sample vectors of the form $(\theta, \Delta\theta, v)$.

Then the observed synaptic vector $\mathbf{m}_j = [9, -10]$ increases the count of FAM cell PS $\times$ NS and increases the weight of FAM rule " IF X is PS, THEN Y is NS."

This amounts to nearest-neighbor classification of synaptic quantization vectors. We assign quantization vector $\mathbf{m}_k$ to FAM cell F_{ij} iff $\mathbf{m}_k$ is closer to the centroid of F_{ij} than to all other FAM-cell centroids. We break ties arbitrarily. Centroid classification allows the FAM cells to overlap.

Adaptive BIOFAM Example: Inverted Pendulum

We used DCL to train an AFAM to control the inverted pendulum discussed above. We used the accompanying C-software to generate 1000 pendulum trajectory data. These product-space training vectors $(\theta, \Delta\theta, v)$ were points in R^3. Pendulum angle θ data ranged between -90 and 90. Pendulum angular-velocity $\Delta\theta$ data ranged from -150 to 150.

We defined FAM cells by uniformly partitioning the product-space "cube" in R^3. Fuzzy variables could assume only the five fuzzy-set values NM, NS, ZE, PS, and PM. So there were 125 possible FAM rules. For instance, the steady-state FAM rule took the form (ZE, ZE; ZE) or, more completely, "IF θ = ZE AND $\Delta\theta$ = ZE, THEN v = ZE."

A BIOFAM controlled the inverted pendulum. The BIOFAM restored the pendulum to equilibrium as we knocked it over to the right and to the left. (Function keys F9 and F10 knock the pendulum over to the left and to the right. Input-output sample data reads automatically to a training data file.) Eleven FAM rules described the BIOFAM controller. Figure 8.7 displays this FAM bank. The zero (ZE) row and column are ordinal inverses of the respective row and column indices.

We trained 125 three-dimensional synaptic quantization vectors with differential competitive learning, as discussed in Chapters 4 and 6, and in Chapter 1 of the companion volume [Kosko, 1991]. In principle the 125 synaptic vectors could describe a uniform distribution of product-space trajectory data. Then the 125 FAM

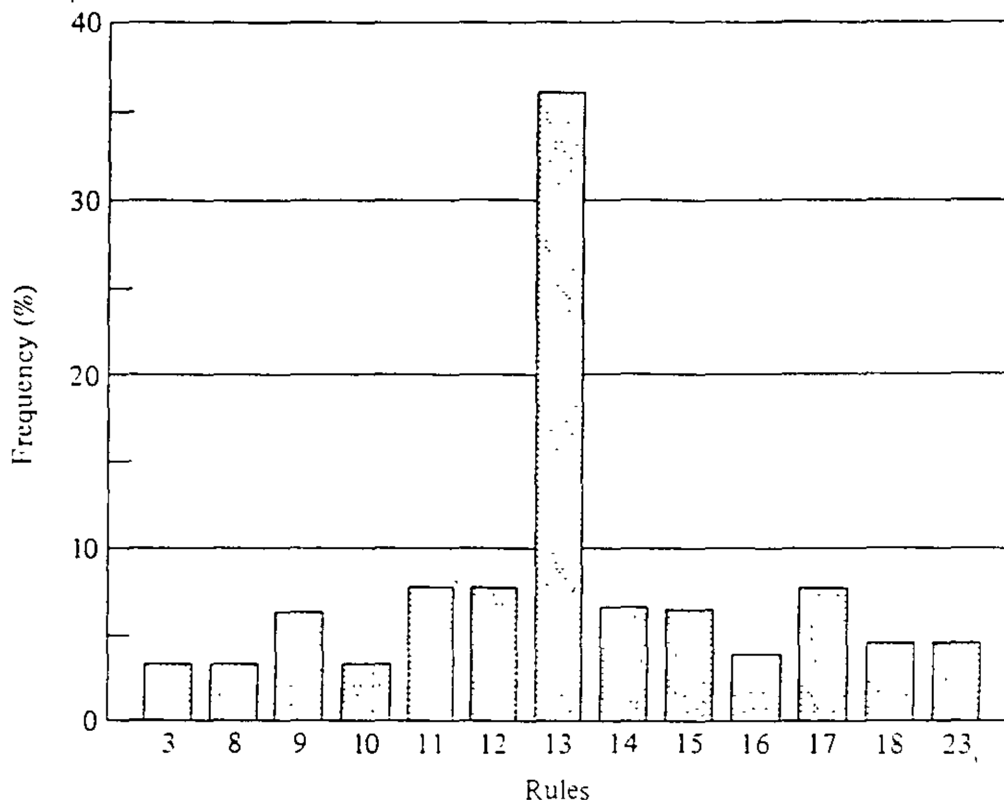


FIGURE 8.8 Synaptic-vector histogram. Differential competitive learning allocated 125 three-dimensional synaptic vectors to the 125 FAM cells. Here the adaptive system has sampled 1000 representative pendulum-control data. DCL allocates the synaptic vectors to only 13 FAM cells. The steady-state FAM cell (ZE, ZE; ZE) is most frequent.

cells would each contain one synaptic vector. Alternatively, if we used a vertically stabilized pendulum to generate the 1000 training vectors, all 125 synaptic vectors would concentrate in the (ZE, ZE; ZE) FAM cell. This would still be true if we perturbed the pendulum only mildly from vertical equilibrium.

DCL distributed the 125 synaptic vectors to 13 FAM cells. So we estimated 13 FAM rules. Some FAM cells contained more synaptic vectors than others. Figure 8.8 displays the synaptic-vector histogram after the DCL system samples the 1000 samples. Actually Figure 8.8 displays a truncated histogram. The horizontal axis should list all 125 FAM cells, all 125 FAM-rule weights w_k in (8-17). The missing 112 entries have zero synaptic-vector frequency.

Figure 8.8 gives a snapshot of the adaptive process. Successive data gradually modify the histogram. "Good" training samples should include a significant number of equilibrium samples. In Figure 8.8 the steady-state FAM cell (ZE, ZE; ZE) is clearly the most frequent.

Figure 8.9 displays the DCL-estimated FAM bank. The product-space clustering method rapidly recovered the 11 original FAM rules. It also estimated the two additional FAM rules (PS, NM; ZE) and (NS, PM; ZE), which did not affect the BIOFAM system's performance. The estimated FAM bank defined a BIOFAM,

Δθ \ θ	NM	NS	Z	PS	PM
NM			PM	Z	
NS			PS	Z	
Z	PM	PS	Z	NS	NM
PS		Z	NS		
PM		Z	NM		

FIGURE 8.9 DCL-estimated FAM bank. Product-space clustering recovered the original 11 FAM rules and estimated two new FAM rules. The new and original BIOFAM systems controlled the inverted pendulum equally well.

with all 13 FAM-rule weights set w_k equal to unity, that controlled the pendulum as well as the original BIOFAM controlled it.

In non-real-time applications we can in principle omit the adaptive step altogether. We can directly compute the FAM-cell histogram if we count all sampled data. Then the (growing) number of synaptic vectors equals the number of training samples. This procedure equally weights all samples, and so tends not to "track" an evolving process. Competitive learning weights more recent samples more heavily. Competitive learning's metrical-classification step also helps filter noise from the stream of sample data.

REFERENCES

Dubois, D., and Prade, H., *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, Orlando, FL, 1980.

Hebb, D., *The Organization of Behavior*, Wiley, New York, 1949.

Klir, G. J., and Foger, T. A., *Fuzzy Sets, Uncertainty, and Information*, Prentice Hall, Englewood Cliffs, NJ, 1988.

Kosko, B., "Fuzzy Knowledge Combination," *International Journal of Intelligent Systems*, vol. 1, 293-320, 1986.

Kosko, B., "Fuzzy Associative Memory Systems," *Fuzzy Expert Systems*, A. Kandel (Ed.), Addison-Wesley, Reading, MA, in press, December 1986.

Kosko, B., "Fuzzy Entropy and Conditioning," *Information Sciences*, vol. 40, 165-174, 1986.

Kosko, B., *Foundations of Fuzzy Estimation Theory*, Ph.D dissertation, Department of Electrical Engineering, University of California at Irvine, June 1987; Order Number 8801936, University Microfilms International, 300 N. Zeeb Road, Ann Arbor, MI, 48106.

Kosko, B., "Hidden Patterns in Combined and Adaptive Knowledge Networks," *International Journal of Approximate Reasoning*, vol. 2, no. 4, 377-393, October 1988.

Kosko, B., *Neural Networks for Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Mamdani, E. H., "Application of Fuzzy Logic to Approximate Reasoning Using Linguistic Synthesis," *IEEE Transactions on Computers*, vol. C-26, no. 12, 1182-1191, December 1977.

Taber, W. R., and Siegel, M. A., "Estimation of Expert Weights Using Fuzzy Cognitive Maps," *Proceedings of the IEEE 1st International Conference on Neural Networks (ICNN-87)*, vol. II, 319-325, June 1987.

Taber, W. R., "Knowledge Processing with Fuzzy Cognitive Maps," *Expert Systems with Applications*, vol. 2, no. 1, 82-87, February 1991.

Togai, M., and Watanabe, H., "Expert System on a Chip: An Engine for Realtime Approximate Reasoning, *IEEE Expert*, vol. 1, no. 3, 1986.

Yamakawa, T., "A Simple Fuzzy Computer Hardware System Employing MIN & MAX Operations," *Proceedings of the Second International Fuzzy Systems Association (IFSA)*, Tokyo, 827-830, July 1987.

Yamakawa, T., "Fuzzy Microprocessors—Rule Chip and Defuzzification Chip," *Proceedings of the International Workshop on Fuzzy Systems Applications*, Iizuka-88, Kyushu Institute of Technology, 51-52, August 1988.

Zadeh, L. A., "A Computational Approach to Fuzzy Quantifiers in Natural Languages," *Computers and Mathematics*, vol. 9, no. 1, 149-184, 1983.

PROBLEMS

8.1. Use correlation-minimum encoding to construct the FAM matrix M from the fit-vector pair (A, B) if $A = (.6\ 1\ .2\ .9)$ and $B = (.8\ .3\ 1)$. Is (A, B) a bidirectional fixed point? Pass $A' = (.2\ .9\ .3\ .2)$ through M, and $B' = (.9\ .5\ 1)$ through M^T. Do the recalled fuzzy sets differ from B and A?

8.2. Repeat Problem 8.1 using correlation-product encoding.

8.3. Compute the fuzzy entropy $E(M)$ of M in Problems 8.1 and 8.2.

8.4. If $M = A^T \circ B$ in Problem 8.1, find a different FAM matrix M' with greater fuzzy entropy, $E(M') > E(M)$, but that still gives perfect recall: $A \circ M' = B$. Find the *maximum-entropy fuzzy associative memory* (MEFAM) matrix $\mathbf{M}^*$ such that $A \circ \mathbf{M}^* = B$.

8.5. Prove: If $M = A^T \circ B$ or $M = A^T B$, $A \circ M = B$, and $A \subset A'$, then $A' \circ M = B$.

8.6. Prove: $\max_{1 \le k \le p} \min(a_k, b_k) \le \min(\max_{1 \le k \le p} a_k, \max_{1 \le k \le p} b_k)$.

8.7. Use truth tables to prove the two-valued propositional tautologies:

(a) [A → (B OR C)] → [(A → B) OR (A → C)].

(b) [A → (B AND C)] → [(A → B) AND (A → C)].

(c) [(A OR B) → C] → [(A → C) OR (B → C)].

(d) $[(A \longrightarrow C) \text{ AND } (B \longrightarrow C)] \longrightarrow [(A \text{ AND } B) \longrightarrow C]$.

Is the converse of (c) a tautology? Explain whether this affects BIOFAM inference.

8.8. *(BIOFAM inference.)* Suppose the input spaces X and Y both equal $[-10, 10]$, and the output space Z equals $[-100, 100]$. Define five trapezoidal fuzzy sets—NL, NS, ZE, PS, PL—on X, Y, and Z. Suppose the underlying (unknown) system transfer function is $z = x^2 - y^2$. State at least five FAM rules that accurately describe the system's behavior. Use $z = x^2 - y^2$ to generate streams of sample data. Use BIOFAM inference and fuzzy-centroid defuzzification to map input pairs (x, y) to output data z. Plot the BIOFAM outputs and the desired outputs z. What is the arithmetic average of the squared errors $(F(x, y) - x^2 + y^2)^2$? Divide the product space $X \times Y \times Z$ into 125 overlapping FAM cells. Estimate FAM rules from clustered system data (x, y, z). Use these FAM rules to control the system. Evaluate the performance.

SOFTWARE PROBLEMS

The following problems use the accompanying FAM software for controlling an inverted pendulum.

1. Explain why the pendulum stabilizes in the diagonal position if the pendulum bob mass increases to maximum and the motor current decreases slightly. The pendulum stabilizes in the vertical position if you remove which FAM rules?
2. Oscillation results if you remove which FAM rules? The pendulum sticks in a horizontal equilibrium if you remove which FAM rules?
3. Use DCL to train a new FAM system. Use the F3 and F4 function keys to gradually generate interesting control trajectories. Try first to recover the original FAM rules. Try next to recover only half of the original FAM rules. Does the FAM system still stabilize the inverted pendulum?

9

COMPARISON OF FUZZY AND NEURAL TRUCK BACKER-UPPER CONTROL SYSTEMS

Seong-Gon Kong and Bart Kosko

FUZZY AND NEURAL CONTROL SYSTEMS

In this chapter we develop fuzzy and neural systems to back up a simulated truck, and truck-and-trailer, to a loading dock in a planar parking lot. We use differential competitive learning and the product-space clustering technique, discussed in Chapter 8, to adaptively generate fuzzy-associative-memory (FAM) rules from training data taken from the fuzzy and neural simulations.

We developed the neural truck systems on the design recently proposed by Nguyen and Widrow [1989]. We trained the neural truck systems with the backpropagation learning algorithm, discussed in Chapter 5. In principle product-space clustering can convert any neural black-box system into a representative set of FAM rules.

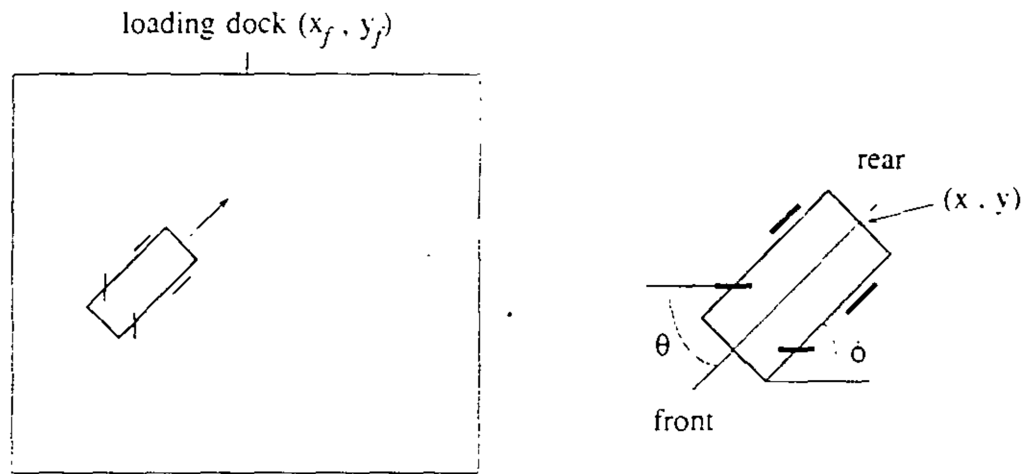


FIGURE 9.1 Diagram of simulated truck and loading zone.

BACKING UP A TRUCK

Figure 9.1 shows the simulated truck and loading zone. The truck corresponds to the cab part of the neural truck in the Nguyen-Widrow neural truck backer-upper system. The three state variables ϕ, x, and y exactly determine the truck position. ϕ specifies the angle of the truck with the horizontal. The coordinate pair (x, y) specifies the position of the rear center of the truck in the plane.

The goal was to make the truck arrive at the loading dock at a right angle ($\phi_f = 90°$) and to align the position (x, y) of the truck with the desired loading dock (x_f, y_f). We considered only backing up. The truck moved backward by some fixed distance at every stage. The loading zone corresponded to the plane $[0, 100] \times [0, 100]$, and (x_f, y_f) equaled (50, 100).

At every stage the fuzzy and neural controllers should produce the steering angle θ that backs up the truck to the loading dock from any initial position and from any angle in the loading zone.

Fuzzy Truck Backer-Upper System

We first specified each controller's input and output variables. The input variables were the truck angle ϕ and the x-position coordinate x. The output variable was the steering-angle signal θ. We assumed enough clearance between the truck and the loading dock so we could ignore the y-position coordinate. The variable ranges were as follows:

$$0 \leq x \leq 100$$

$$-90 \leq \phi \leq 270$$

$$-30 \leq \theta \leq 30$$

Positive values of θ represented clockwise rotations of the steering wheel. Negative values represented counterclockwise rotations. We discretized all values to reduce

computation. The resolution of ϕ and θ was one degree each. The resolution of x was 0.1.

Next we specified the fuzzy-set values of the input and output fuzzy variables. The fuzzy sets numerically represented linguistic terms, the sort of linguistic terms an expert might use to describe the control system's behavior. We chose the fuzzy-set values of the fuzzy variables as follows:

Angle ϕ		x-position x		Steering-angle signal θ	
RB:	Right Below	LE:	Left	NB:	Negative Big
RU:	Right Upper	LC:	Left Center	NM:	Negative Medium
RV:	Right Vertical	CE:	Center	NS:	Negative Small
VE:	Vertical	RC:	Right Center	ZE:	Zero
LV:	Left Vertical	RI:	Right	PS:	Positive Small
LU:	Left Upper			PM:	Positive Medium
LB:	Left Below			PB:	Positive Big

Fuzzy subsets contain elements with degrees of membership. A fuzzy membership function $m_A: Z \longrightarrow [0, 1]$ assigns a real number between 0 and 1 to every element z in the universe of discourse Z. This number $m_A(z)$ indicates the degree to which the object or data z belongs to the fuzzy set A. Equivalently, $m_A(z)$ defines the *fit* (fuzzy unit) value [Kosko, 1986] of element z in A.

Fuzzy membership functions can have different shapes depending on the designer's preference or experience. In practice fuzzy engineers have found triangular and trapezoidal shapes help capture the modeler's sense of fuzzy numbers and simplify computation. Figure 9.2 shows membership-function graphs of the fuzzy subsets above. In the third graph, for example, $\theta = 20^\circ$ is Positive Medium to degree 0.5, but only Positive Big to degree 0.3.

In Figure 9.2 the fuzzy sets CE, VE, and ZE are narrower than the other fuzzy sets. These narrow fuzzy sets permit fine control near the loading dock. We used wider fuzzy sets to describe the endpoints of the range of the fuzzy variables ϕ, x, and θ. The wider fuzzy sets permitted rough control far from the loading dock.

Next we specified the fuzzy "rulebase" or bank of *fuzzy associative memory* (FAM) rules. Fuzzy associations or "rules" (A, B) associate output fuzzy sets B of control values with input fuzzy sets A of input-variable values. We can write fuzzy associations as antecedent-consequent pairs or IF-THEN statements.

In the truck backer-upper case, the FAM bank contained the 35 FAM rules in Figure 9.3. For example, the FAM rule of the left upper block (FAM rule 1) corresponds to the following fuzzy association:

$$\text{IF } x = \text{LE AND } \phi = \text{RB, THEN } \theta = \text{PS}$$

FAM rule 18 indicates that if the truck is in near the equilibrium position, then the controller should not produce a positive or negative steering-angle signal. The FAM rules in the FAM-bank matrix reflect the symmetry of the controlled system.

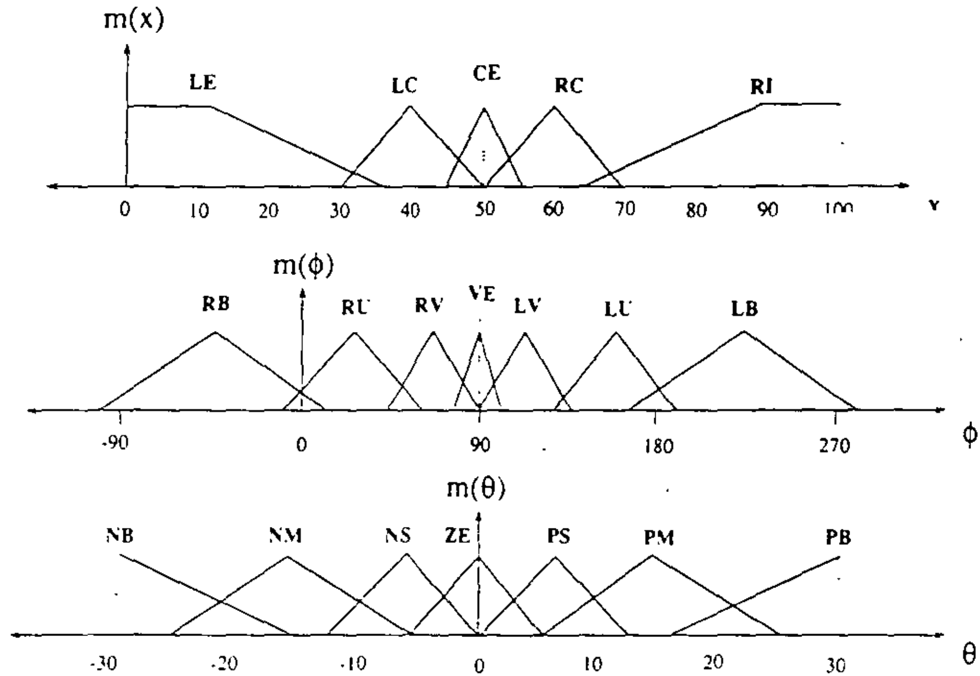


FIGURE 9.2 Fuzzy membership functions for each linguistic fuzzy-set value. To allow finer control, the fuzzy sets that correspond to near the loading dock are narrower than the fuzzy sets that correspond to far from the loading dock.

φ \ x	LE	LC	CE	RC	RI
RB	1 PS	2 PM	3 PM	4 PB	5 PB
RU	6 NS	7 PS	PM	PB	PB
RV	NM	NS	PS	PM	PB
VE	NM	NM	18 ZE	PM	PM
LV	NB	NM	NS	PS	PM
LU	NB	NB	NM	NS	PS
LB	NB	NB	NM	NM	35 NS

FIGURE 9.3 FAM-bank matrix for the fuzzy truck backer-upper controller.

For the initial condition $x = 50$ and $\phi = 270$, the fuzzy truck did not perform well. The symmetry of the FAM rules and the fuzzy sets cancelled the fuzzy controller output in a rare saddle point. For this initial condition, the neural controller (and truck-and-trailer below) also performed poorly. Any perturbation breaks the symmetry. For example, the rule (IF $x = 50$ AND $\phi = 270$, THEN $\theta = 5$) corrected the problem.

The three-dimensional control surfaces in Figure 9.4 show steering-angle signal outputs θ that correspond to all combinations of values of the two input state

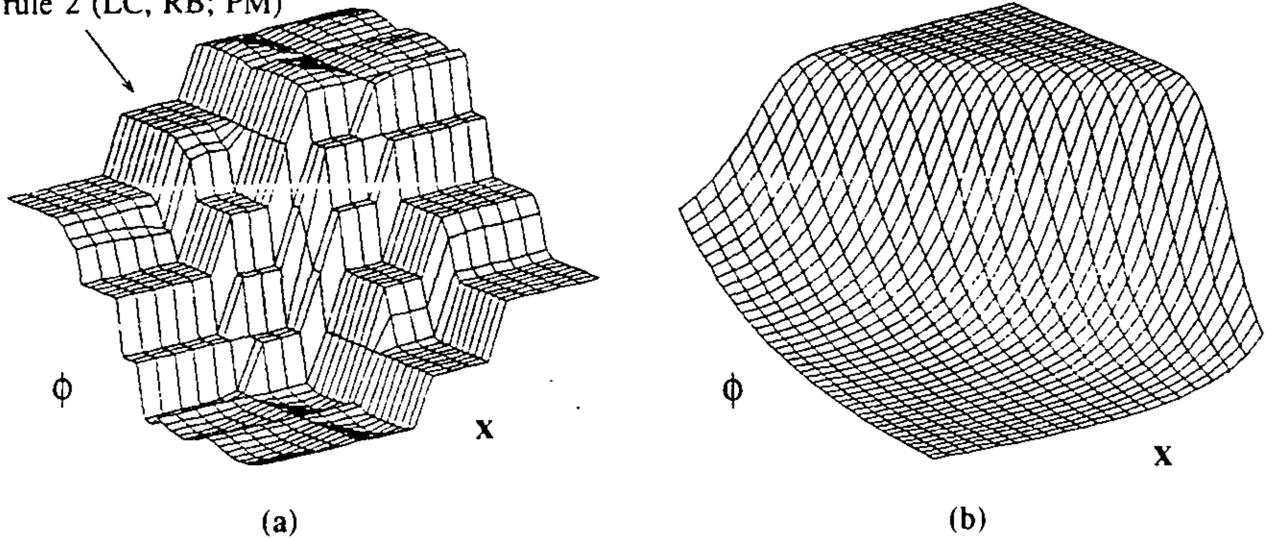


FIGURE 9.4 (a) Control surface of the fuzzy controller. Fuzzy-set values determined the input and output combination corresponding to FAM rule 2 (IF x = LC AND ϕ = RB, THEN θ = PM). (b) Corresponding control surface of the neural controller for constant value $y = 20$.

variables ϕ and x. The control surface defines the fuzzy controller. In this simulation the correlation-minimum FAM inference procedure, discussed in Chapter 8, determined the fuzzy control surface. If the control surface changes with sampled variable values, the system behaves as an *adaptive* fuzzy controller. Below we demonstrate unsupervised adaptive control of the truck and the truck-and-trailer systems.

Finally, we determined the output action given the input conditions. We used the correlation-minimum inference method illustrated in Figure 9.5. Each FAM rule produced the output fuzzy set clipped at the degree of membership determined by the input conditions and the FAM rule. Alternatively, correlation-product inference would combine FAM rules multiplicatively. Each FAM rule emitted a fit-weighted output fuzzy set O_i at each iteration. The total output O added these weighted outputs:

$$O = \sum_i O_i \tag{9-1}$$

$$= \sum_i \min(f_i, S_i) \tag{9-2}$$

where f_i denotes the antecedent fit value and S_i represents the consequent fuzzy set of steering-angle values in the ith FAM rule. Earlier fuzzy systems combined the output sets O_i with pairwise maxima. But this tends to produce a uniform output set O as the number of FAM rules increases. Adding the output sets O_i invokes the fuzzy version of the central limit theorem. This tends to produce a symmetric, unimodal output fuzzy set O of steering-angle values.

Fuzzy systems map fuzzy sets to fuzzy sets. The fuzzy control system's

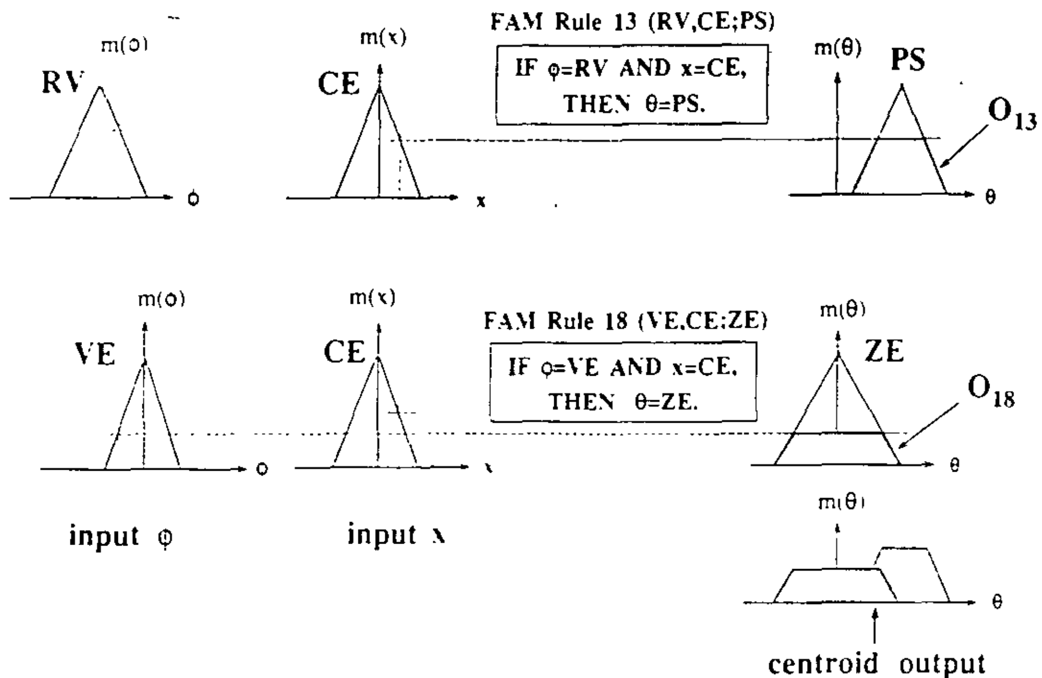


FIGURE 9.5 Correlation-minimum inference with centroid defuzzification method. Then FAM-rule antecedents combined with AND use the *minimum* fit value to activate consequents. Those combined with OR would use the *maximum* fit value.

output defines the fuzzy set O of steering-angle values at each iteration. We must "defuzzify" the fuzzy set O to produce a numerical (point-estimate) steering-angle output value θ.

As discussed in Chapter 8, the simplest defuzzification scheme selects the value corresponding to the *maximum fit* value in the fuzzy set. This mode-selection approach ignores most of the information in the output fuzzy set and requires an additional decision algorithm when multiple modes occur.

Centroid defuzzification provides a more effective procedure. This method uses the *fuzzy centroid* $\bar{\theta}$ as output:

$$\bar{\theta} = \frac{\sum_{j=1}^{p} \theta_j m_O(\theta_j)}{\sum_{j=1}^{p} m_O(\theta_j)} \tag{9-3}$$

where O defines a fuzzy subset of the steering-angle universe of discourse $\Theta = \{\theta_1, \ldots, \theta_p\}$. The central-limit-theorem effect produced by adding output fuzzy set O_i benefits both max-mode and centroid defuzzification. Figure 9.5 shows the correlation-minimum inference and centroid defuzzification applied to FAM rules 13 and 18. We used centroid defuzzification in all simulations.

With 35 FAM rules, the fuzzy truck controller produced successful truck backing-up trajectories starting from any initial position. Figure 9.6 shows typical

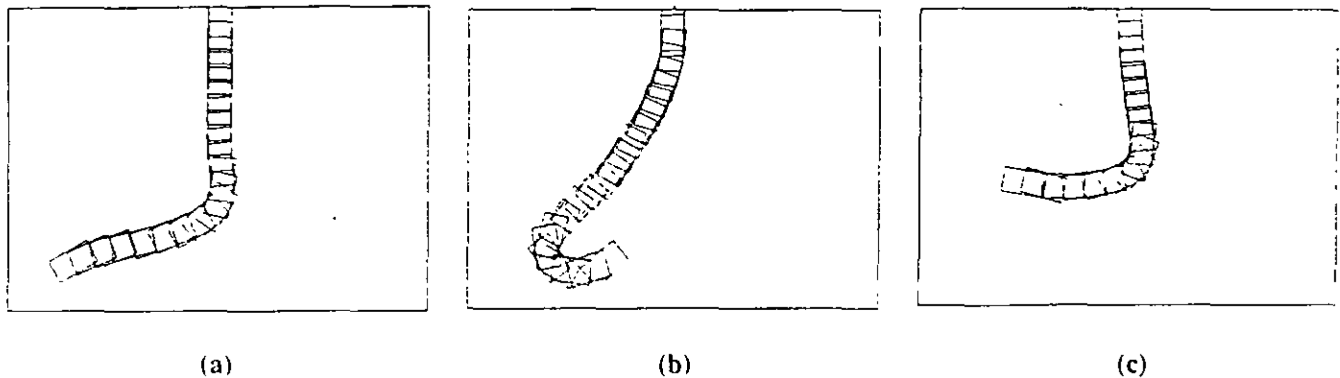


FIGURE 9.6 Sample truck trajectories of the fuzzy controller for initial positions (x, y, ϕ): (a) (20, 20, 30), (b) (30, 10, 220), and (c) (30, 40, −10).

examples of the fuzzy-controlled truck trajectories from different initial positions. The fuzzy control system did not use ("fire") all FAM rules at each iteration. Equivalently most output consequent sets are empty. In most cases the system used only one or two FAM rules at each iteration. The system used at most 4 FAM rules at once.

Neural Truck Backer-Upper System

The neural truck backer-upper of Nguyen and Widrow [1989] consisted of multilayer feedforward neural networks trained with the backpropagation gradient-descent (stochastic-approximation) algorithm. The *neural control system* consisted of two neural networks: the controller network and the truck emulator network. The *controller network* produced an appropriate steering-angle signal output given any parking-lot coordinates (x, y), and the angle ϕ. The *emulator network* computed the next position of the truck. The emulator network took as input the previous truck position and the current steering-angle output computed by the controller network.

We did not train the emulator network since we could not obtain "universal" synaptic connection weights for the truck-emulator network. The backpropagation learning algorithm did not converge for some sets of training samples. The number of training samples for the emulator network might exceed 3000. For example, the combinations of training samples of a given angle ϕ, x-position, y-position, and steering-angle signal θ might correspond to 3150 (18 × 5 × 5 × 7) samples, depending on the division of the input-output product space. Moreover, the training samples were numerically similar, since the neuronal signals assumed scaled values in [0, 1] or [−1, 1]. For example, we treated close values, such as 0.40 and 0.41, as distinct sample values.

Simple kinematic equations replaced the truck-emulator network. If the truck moved backward from (x, y) to (x', y') at an iteration, then

$$x' = x + r\cos(\phi') \quad (9\text{-}4)$$

$$y' = y + r\sin(\phi') \quad (9\text{-}5)$$

$$\phi' = \phi - \theta \quad (9\text{-}6)$$

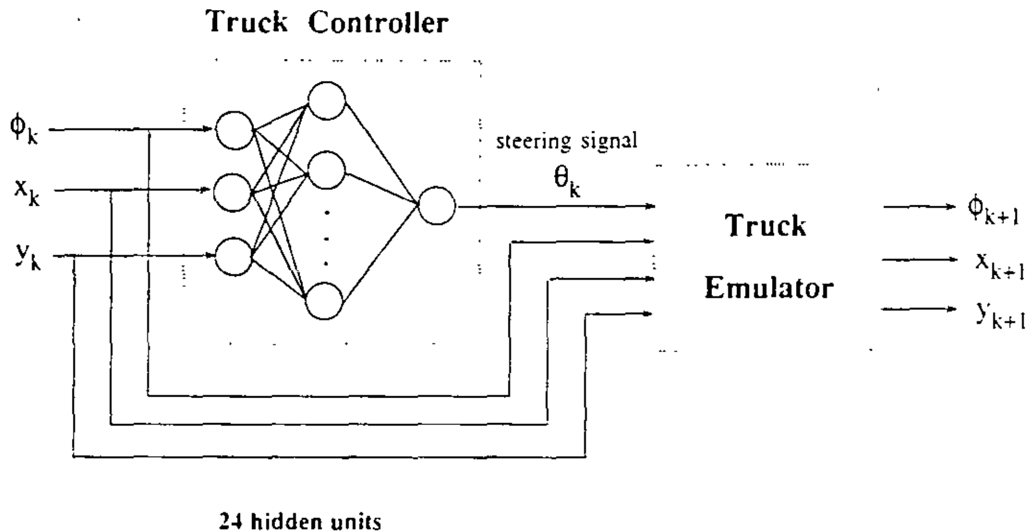


FIGURE 9.7 Topology of our neural control system.

r denotes the fixed driving distance of the truck for all backing movements. We used Equations (9-4)–(9-6) instead of the emulator network. This did not affect the posttraining performance of the neural truck backer-upper, since the truck emulator network backpropagated only errors.

We trained only the controller network with backpropagation. The controller network used 24 "hidden" neurons with logistic sigmoid functions. In the training of the truck controller, we estimated the ideal steering-angle signal at each stage before we trained the controller network. In the simulation, we used the arc-shaped truck trajectory produced by the fuzzy controller as the ideal trajectory. The fuzzy controller generated each training sample (x, y, ϕ, θ) at each iteration of the backing-up process. We used 35 training-sample vectors and needed more than 100,000 iterations to train the controller network.

Figure 9.4(b) shows the resulting neural control surface for $y = 20$. The neural control surface shows less structure than the corresponding fuzzy control surface. This reflects the unstructured nature of black-box supervised learning. Figure 9.7 shows the network connection topology for our neural truck backer-upper control system.

Figure 9.8 shows typical examples of the neural-controlled truck trajectories from several initial positions. Even though we trained the neural network to follow the smooth arc-shaped path, some learned truck trajectories were nonoptimal.

Comparison of Fuzzy and Neural Systems

As shown in Figures 9.6 and 9.8, the fuzzy controller always smoothly backed up the truck, but the neural controller did not. The neural-controlled truck sometimes followed an irregular path

Training the neural control system was time-consuming. The backpropagation

123

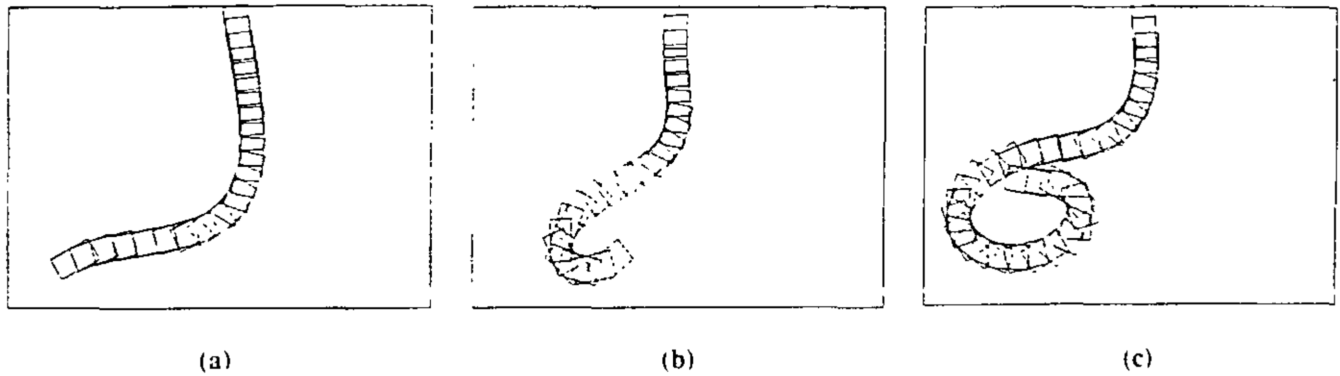


FIGURE 9.8 Sample truck trajectories of the neural controller for initial positions (x, y, ϕ): (a) (20, 20, 30), (b) (30, 10, 220), and (c) (30, 40, −10).

algorithm required thousands of back-ups to train the controller network. In some cases, the learning algorithm did not converge.

We "trained" the fuzzy controller by encoding our own common-sense FAM rules. Once we develop the FAM-rule bank, we can compute control outputs from the resulting FAM-bank matrix or control surface. The fuzzy controller did not need a truck emulator and did not require a math model of how outputs depended on inputs.

The fuzzy controller was computationally lighter than the neural controller. Most computation operations in the neural controller involved the multiplication, addition, or logarithm of two real numbers. In the fuzzy controller, most computational operations involved comparing and adding two real numbers.

Sensitivity Analysis

We studied the sensitivity of the fuzzy controller in two ways. We replaced the FAM rules with destructive or "sabotage" FAM rules, and we randomly removed FAM rules. We deliberately chose sabotage FAM rules to confound the system. Figure 9.9 shows the trajectory when two sabotage FAM rules replaced the important steady-state FAM rule—FAM rule 18: the fuzzy controller should produce zero output when the truck is nearly in the correct parking position. Figure 9.10 shows the truck trajectory after we removed four randomly chosen FAM rules (7, 13, 18, and 23). These perturbations did not significantly affect the fuzzy controller's performance.

We studied robustness of each controller by examining failure rates. For the fuzzy controller we removed fixed percentages of randomly selected FAM rules from the system. For the neural controller we removed training data. Figure 9.11 shows performance errors averaged over ten typical back-ups with missing FAM rules for the fuzzy controller and missing training data for the neural controller. The missing FAM rules and training data ranged from 0 to 100 percent of the total. In Figure 9.11(a), the docking error equaled the Euclidean distance from the actual

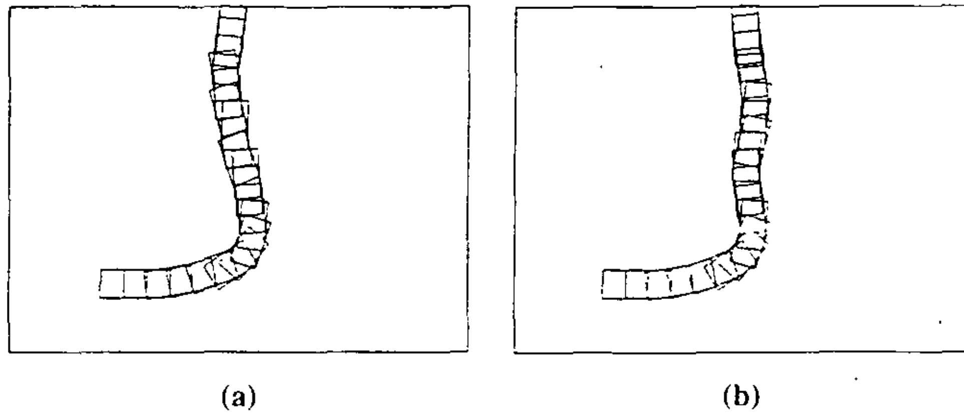


FIGURE 9.9 The fuzzy truck trajectory after we replaced the key steady-state FAM rule 18 by the two worst rules: (a) IF x = CE AND ϕ = VE, THEN θ = PB; (b) IF x = CE AND ϕ = VE, THEN θ = NB.

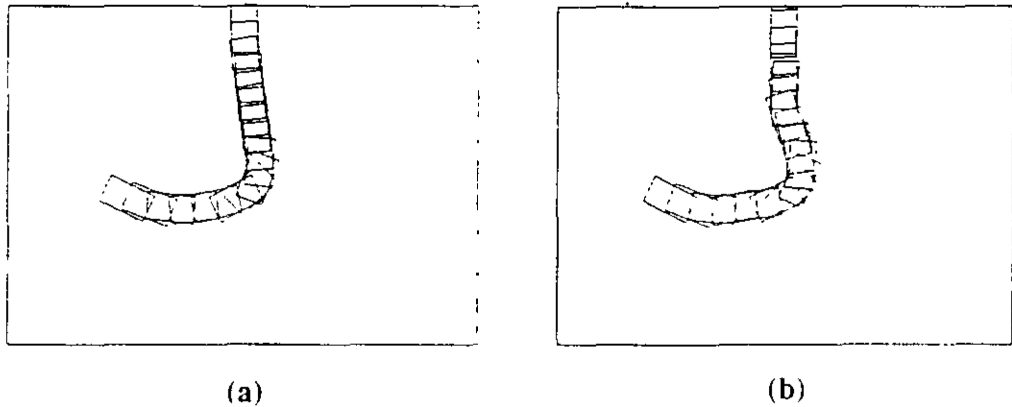


FIGURE 9.10 Fuzzy truck trajectory when (a) no FAM rules are removed and (b) FAM rules 7, 13, 18, and 23 are removed.

final position (ϕ, x, y) to the desired final position (ϕ_f, x_f, y_f):

$$\text{Docking error} \quad = \quad \sqrt{(\phi_f - \phi)^2 + (x_f - x)^2 + (y_f - y)^2} \tag{9-7}$$

In Figure 9.11(b), the trajectory error equaled the ratio of the actual trajectory length of the truck divided by the straight-line distance to the loading dock:

$$\text{Trajectory error} \quad = \quad \frac{\text{length of truck trajectory}}{\text{distance(initial position, desired final position)}} \tag{9-8}$$

Adaptive Fuzzy Truck Backer-Upper

Adaptive FAM (AFAM) systems generate FAM rules directly from training data. A one-dimensional FAM system, $S: I^n \longrightarrow I^p$, defines a FAM rule, a single association of the form (A_i, B_i). In this case the input-output product space equals $I^n \times I^p$. As discussed in Chapter 8, a FAM rule (A_i, B_i) defines a cluster or ball

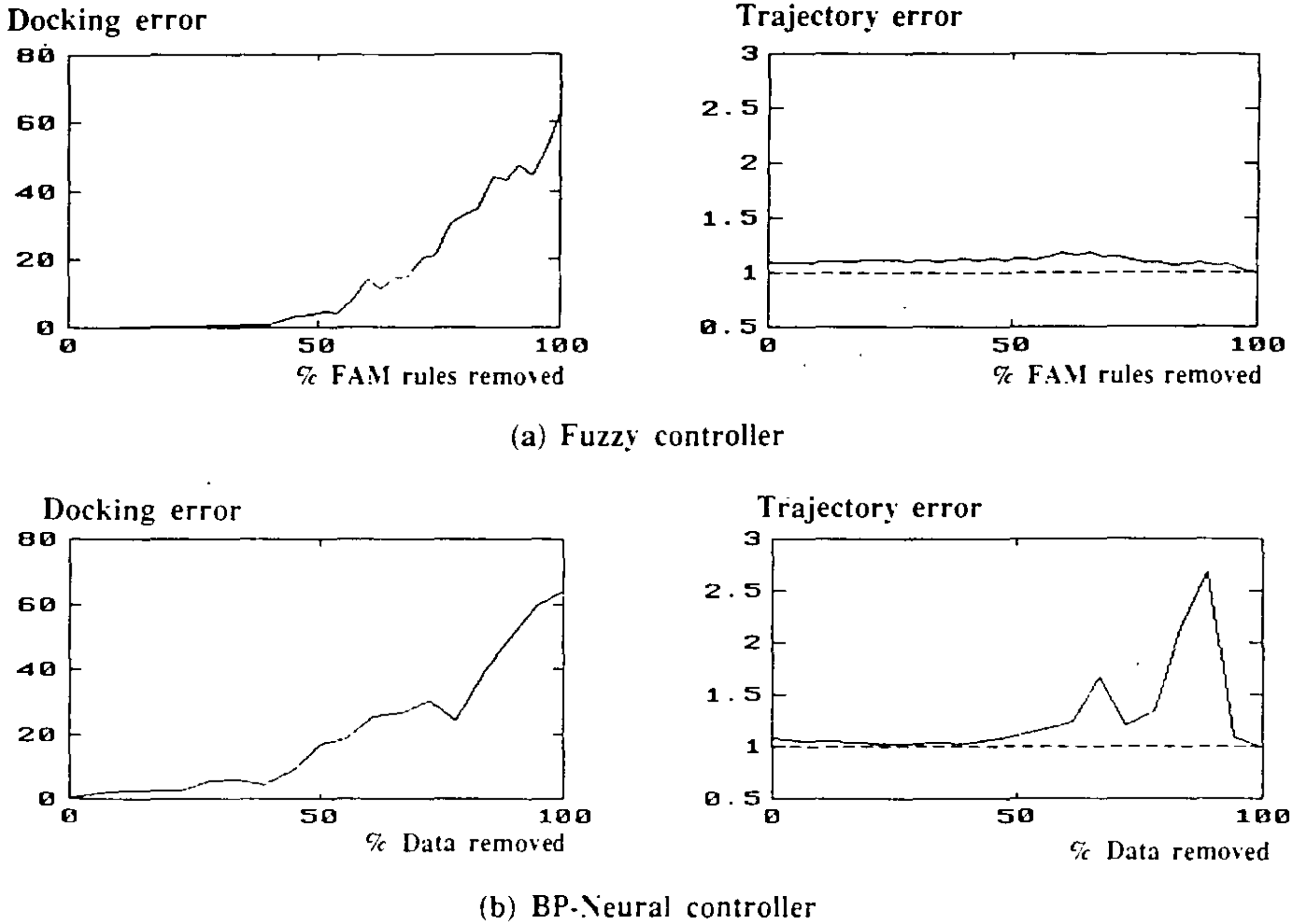


FIGURE 9.11 Comparison of robustness of the controllers: (a) Docking and trajectory error of the fuzzy controller; (b) Docking and trajectory error of the neural controller.

of points in the product-space cube $I^n \times I^p$ centered at the point (A_i, B_i). Adaptive clustering algorithms can estimate the unknown FAM rule (A_i, B_i) from training samples in R^2. We used differential competitive learning (DCL) to recover the bank of FAM rules that generated the truck-training data.

We generated 2230 truck samples from seven different initial positions and varying angles. We chose the initial positions (20, 20), (30, 20), (45, 20), (50, 20), (55, 20), (70, 20), and (80, 20). We changed the angle from $-60°$ to $240°$ at each initial position. At each step, the fuzzy controller produced output steering angle θ. The training vectors (x, ϕ, θ) defined points in a three-dimensional product-space. x had five fuzzy-set values: LE, LC, CE, RC, and RI. ϕ had seven fuzzy-set values: RB, RU, RV, VE, LV, LU, and LB. θ had seven fuzzy-set values: NB, NM, NS, ZE, PS, PM, and PB. So there were 245 ($5 \times 7 \times 7$) possible FAM cells.

We defined FAM cells by partitioning the effective product space as follows. We divided the space $0 \leq x \leq 100$ into five nonuniform intervals [0, 32.5], [32.5, 47.5], [47.5, 52.5], [52.5, 67.5], and [67.5, 100]. Each interval represented the five fuzzy-set values LE, LC, CE, RC, and RI. This choice corresponded to the nonoverlapping intervals of the fuzzy membership function graphs $m(x)$

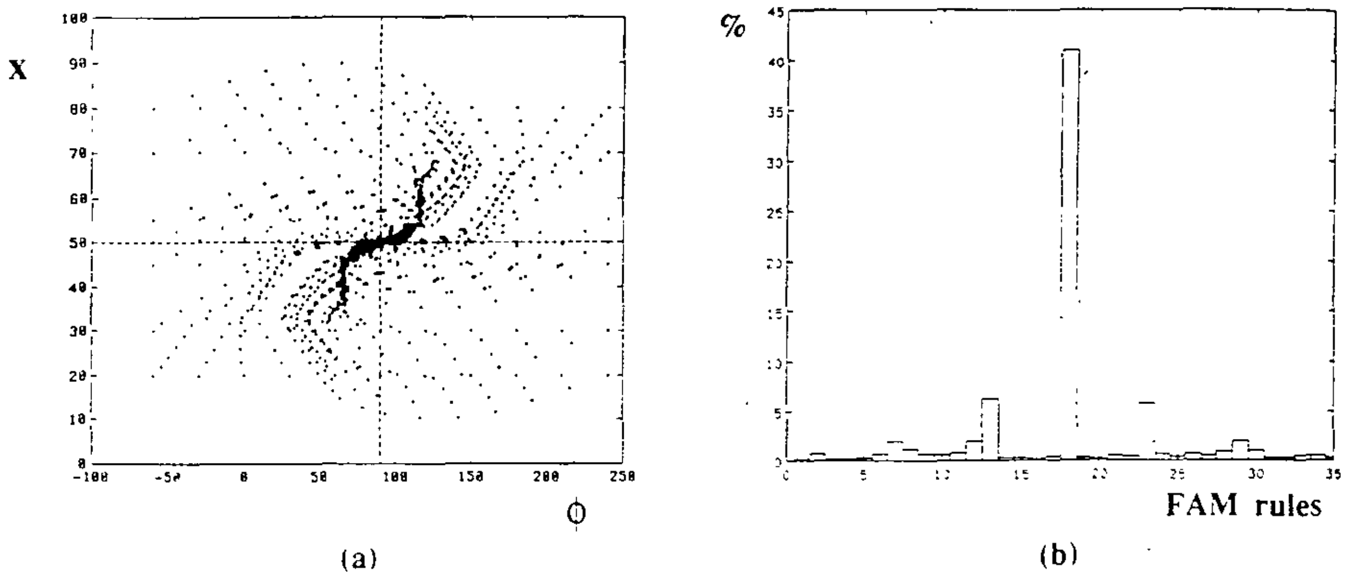


FIGURE 9.12 (a) Input data distribution. (b) Synaptic-vector histogram. Differential competitive learning allocated synaptic quantization vectors to FAM cells. The steady-state FAM cell (CE, VE; ZE) contained the most synaptic vectors.

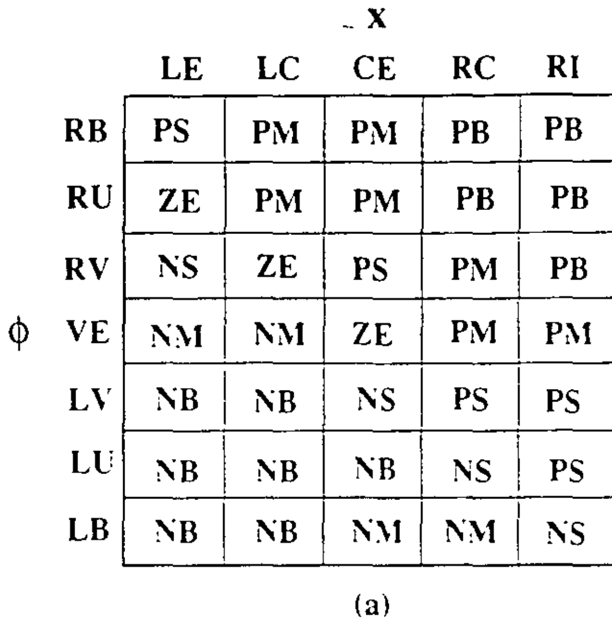
in Figure 9.2. Similarly, we divided the space $-90 \le \phi \le 270$ into seven nonuniform intervals $[-90, 0]$, $[0, 66.5]$, $[66.5, 86]$, $[86, 94]$, $[94, 113.5]$, $[113.5, 182.5]$, and $[182.5, 270]$, which corresponded respectively to RB, RU, RV, VE, LV, LU, and LB. We divided the space $-30 \le \theta \le 30$ into seven nonuniform intervals $[-30, -20]$, $[-20, -7.5]$, $[-7.5, -2.5]$, $[-2.5, 2.5]$, $[2.5, 7.5]$, $[7.5, 20]$, and $[20, 30]$, which corresponded to NB, NM, NS, ZE, PS, PM, and PB. FAM cells near the center were smaller than outer FAM cells because we chose narrow membership functions near the steady-state FAM cell. Uniform partitions of the product space produced poor estimates of the original FAM rules. As in Figure 9.2, this reflected the need to judiciously define the fuzzy-set values of the system fuzzy variables.

We performed product-space clustering with the version of DCL discussed in Chapter 1 of the companion volume [Kosko, 1991]. If a FAM cell contained at least one of the 245 synaptic quantization vectors, we entered the corresponding FAM rule in the FAM matrix. In case of ties we chose the FAM cell with the most densely clustered data.

Figure 9.12(a) shows the input sample distribution of (x, ϕ). We did not include the variable θ in the figure. Training data clustered near the steady-state position ($x = 50$ and $\phi = 90°$). Figure 9.12(b) displays the synaptic-vector histogram after DCL classified 2230 training vectors for 35 FAM rules. Since successful FAM system generated the training samples, most training samples, and thus most synaptic vectors, clustered in the steady-state FAM cell.

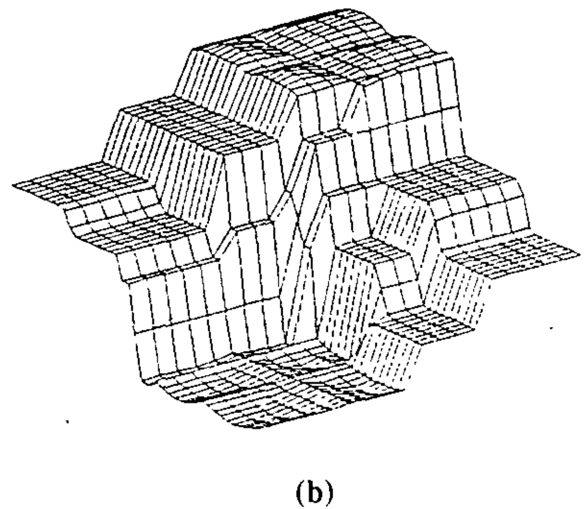
DCL product-space clustering estimated 35 new FAM rules. Figure 9.13 shows the DCL-estimated FAM bank and the corresponding control surface. The DCL-estimated control surface visually resembles the underlying unknown control surface in Figure 9.4(a). The two systems produce nearly equivalent truck-backing behavior. This suggests adaptive product-space clustering can estimate the FAM

127



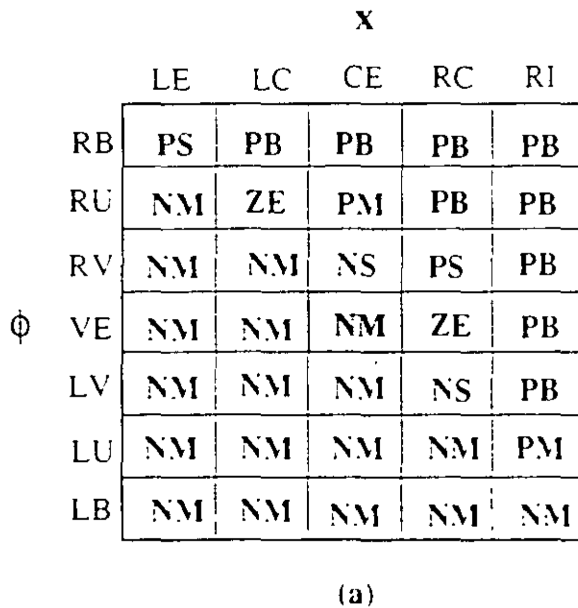
ϕ \ x	LE	LC	CE	RC	RI
RB	PS	PM	PM	PB	PB
RU	ZE	PM	PM	PB	PB
RV	NS	ZE	PS	PM	PB
VE	NM	NM	ZE	PM	PM
LV	NB	NB	NS	PS	PS
LU	NB	NB	NB	NS	PS
LB	NB	NB	NM	NM	NS

(a)



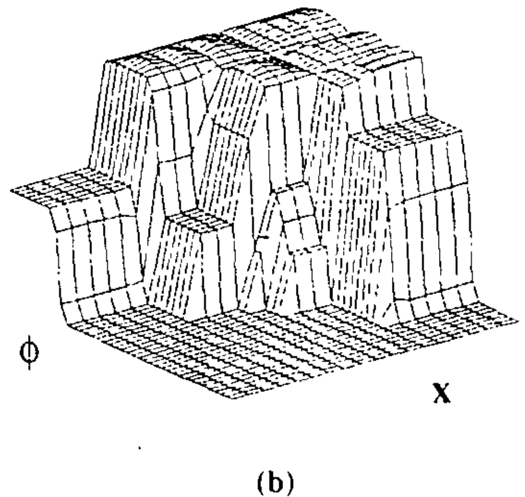
(b)

FIGURE 9.13 (a) DCL-estimated FAM bank. (b) Corresponding control surface



ϕ \ x	LE	LC	CE	RC	RI
RB	PS	PB	PB	PB	PB
RU	NM	ZE	PM	PB	PB
RV	NM	NM	NS	PS	PB
VE	NM	NM	NM	ZE	PB
LV	NM	NM	NM	NS	PB
LU	NM	NM	NM	NM	PM
LB	NM	NM	NM	NM	NM

(a)



(b)

FIGURE 9.14 (a) FAM bank generated by the neural control surface in Figure 9.4b (b) Control surface of the new BP-AFAM system in (a).

rules underlying expert behavior in many cases, even when the expert or fuzzy engineer cannot articulate the FAM rules.

We also used the neural control surface in Figure 9.4(b) to estimate FAM rules. We divided the rectangle $[0, 100] \times [-90, 270]$ into 35 nonuniform squares with the same divisions as in the fuzzy control case. Then we added and averaged the control-surface values in the square. We added a FAM rule to the FAM bank if the averaged value corresponded to one of the seven FAM cells. Figure 9.14 shows the resulting FAM bank and corresponding control surface generated by the neural

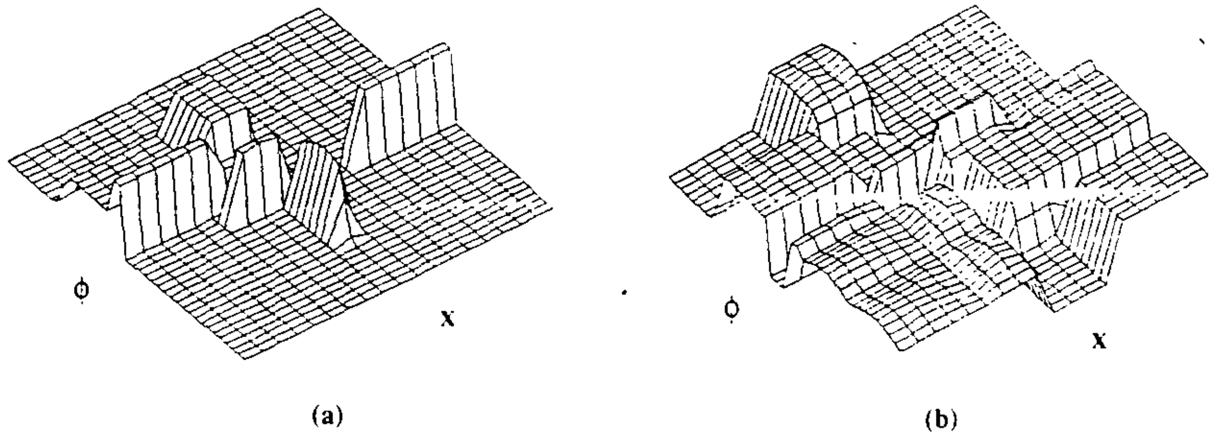


FIGURE 9.15 (a) Absolute difference of the FAM surface in Figure 9.4(a) and the DCL-estimated FAM surface in Figure 9.13(b). (b) Absolute difference of the FAM surface in Figure 9.4(a) and the neural-estimated FAM surface in Figure 9.14(b).

control surface in Figure 9.4(b). This new control surface resembles the original fuzzy control surface in Figure 9.4(a) more than it resembles the neural control surface in Figure 9.4(b). Note the absence of a steady-state FAM rule in the FAM matrix in Figure 9.14(a).

Figure 9.15 compares the DCL-AFAM and BP-AFAM control surfaces with the fuzzy control surface in Figure 9.4(a). Figure 9.15 shows the absolute difference of the control surfaces. As expected, the DCL-AFAM system produced less absolute error than the BP-AFAM system produced.

Figure 9.16 shows the docking and trajectory errors of the two AFAM control systems. The DCL-AFAM system produced less docking error than the BP-AFAM system produced for 100 arbitrary backing-up trials. The two AFAM systems generated similar backing-up trajectories. This suggests that black-box neural estimators can define the front end of FAM-structured systems. In principle we can use this technique to generate structured FAM rules for *any* neural application. We can then inspect and refine these rules and perhaps replace the original neural system with the tuned FAM system.

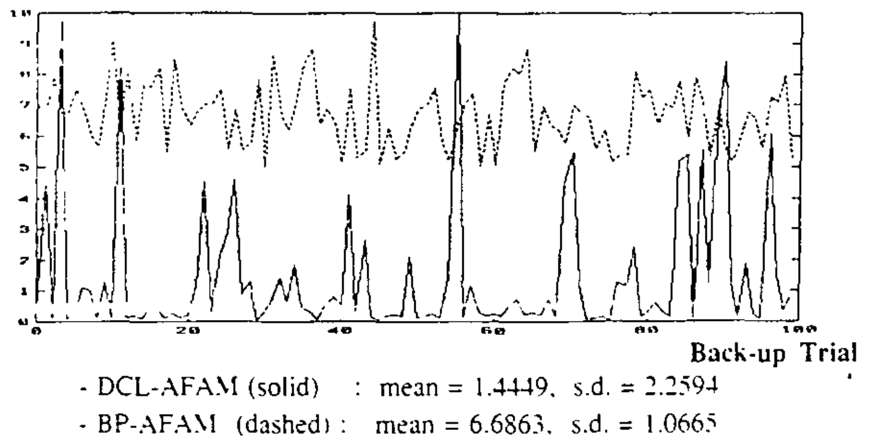
Fuzzy Truck-and-Trailer Controller

We added a trailer to the truck system, as in the original Nguyen-Widrow model. Figure 9.17 shows the simulated truck-and-trailer system. We added one more variable (cab angle, ϕ_c) to the three state variables of the trailerless truck. In this case a FAM rule takes the form

$$\text{IF } x = \text{LE AND } \phi_t = \text{RB AND } \phi_c = \text{PO}, \text{ THEN } \beta = \text{NS}$$

The four state variables x, y, ϕ_t, and ϕ_c determined the position of the truck-and-trailer system in the plane. Fuzzy variable ϕ_t corresponded to ϕ for the trailerless truck. Fuzzy variable ϕ_c specified the relative cab angle with respect to the center

(a) Docking Error



(b) Trajectory Error

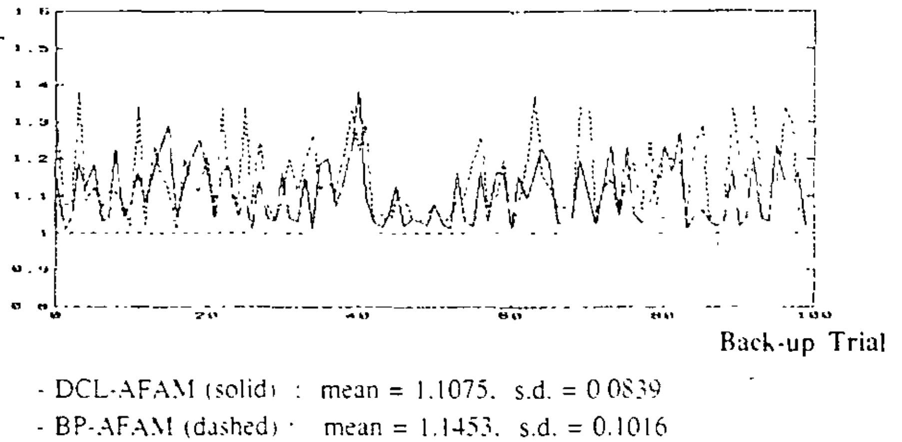
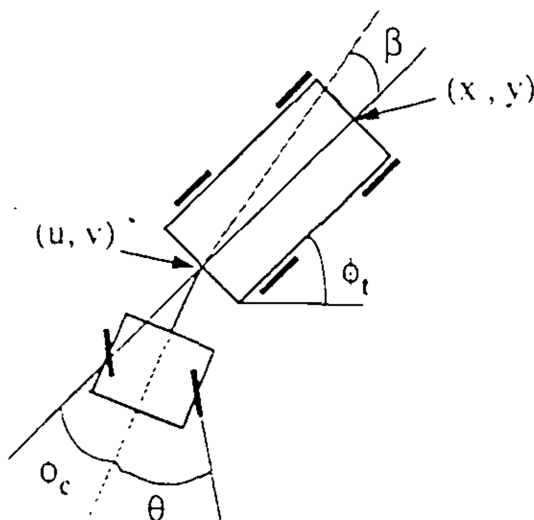


FIGURE 9.16 (a) Docking errors and (b) trajectory errors of the DCL-AFAM and BP-AFAM control systems



(x , y) : Cartesian coordinate of the rear end, [0,100].

(u , v) : Cartesian coordinate of the joint.

ϕ_t : Angle of the trailer with horizontal, [-90,270].

ϕ_c : Relative angle of the cab with trailer, [-90,90].

θ : Steering angle, [-30,30].

β : Angle of the trailer updated at each step, [-30,30].

FIGURE 9.17 Diagram of the simulated truck-and-trailer system.

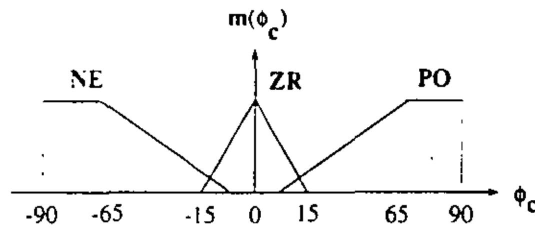


FIGURE 9.18 Membership graphs of the three fuzzy-set values of fuzzy variable ϕ_c.

line along the trailer. ϕ_c ranged from $-90°$ to $90°$. The extreme cab angles $90°$ and $-90°$ corresponded to two "jackknife" positions of the cab with respect to the trailer. Positive ϕ_c value indicated that the cab resided on the left-hand side of the trailer. Negative value indicated that it resided on the right-hand side. Figure 9.17 shows a positive angle value of ϕ_c.

Fuzzy variables x, ϕ_t, and ϕ_c defined the input variables. Fuzzy variable β defined the output variable. β measured the angle that we needed to update the trailer at each iteration. We computed the steering-angle output θ with the following geometric relationship. With the output β value computed, the trailer position (x, y) moved to the new position (x', y'):

$$x' = x + r\cos(\phi_t + \beta) \tag{9-9}$$

$$y' = y + r\sin(\phi_t + \beta) \tag{9-10}$$

where r denotes a fixed backing distance. Then the joint of the cab and the trailer (u, v) moved to the new position (u', v'):

$$u' = x' - l\cos(\phi_t + \beta) \tag{9-11}$$

$$v' = y' - l\sin(\phi_t + \beta) \tag{9-12}$$

where l denotes the trailer length. We updated the directional vector (dir$U$, dir$V$), which defined the cab angle, by

$$\text{dir}U' = \text{dir}U + \Delta u \tag{9-13}$$

$$\text{dir}V' = \text{dir}V + \Delta v \tag{9-14}$$

where $\Delta u = u' - u$, and $\Delta v = v' - v$. The new directional vector (dirU', dirV') defines the new cab angle ϕ'_c. Then we obtain the steering-angle value as $\theta = \phi'_{c,h} - \phi_{c,h}$, where $\phi_{c,h}$ denotes the cab angle with the horizontal. We chose the same fuzzy-set values and membership functions for β as we chose for θ. β ranged from $-30°$ to $30°$. We chose the fuzzy-set values of ϕ_c as NE, ZR and PO as in Figure 9.18.

Figure 9.19 displays the five FAM-rule matrices in the FAM bank of the fuzzy truck-and-trailer system. In Figure 9.19 we fixed the fuzzy variable x as LE, LC, CE, RC, and RI. There were 735 $(7 \times 5 \times 7 \times 3)$ possible FAM rules and only 105 actual FAM rules.

x = LE (ϕ_c rows, ϕ_t columns)

	RB	RU	RV	VE	LV	LU	LB
NE	PS	PS	PS	PS	PS	PB	PB
ZR	PS	NS	NS	NS	NM	NM	NM
PO	NS	NS	NM	NM	NB	NB	NB

x = LC

	RB	RU	RV	VE	LV	LU	LB
NE	PM	PS	PS	PS	PS	PS	PS
ZR	PM	ZE	NS	NM	NM	NB	NB
PO	NS	NS	NS	NM	NM	NB	NB

x = CE

	RB	RU	RV	VE	LV	LU	LB
NE	PM	PM	PS	ZE	PS	PS	PS
ZR	PM	PM	PS	ZE	NS	NM	NM
PO	NS	NS	NS	ZE	NS	NM	NM

x = RC

	RB	RU	RV	VE	LV	LU	LB
NE	PB	PB	PM	PM	PS	PS	PS
ZR	PB	PB	PM	PM	PS	ZE	NM
PO	NS	NS	NS	NS	NS	NS	NM

x = RI

	RB	RU	RV	VE	LV	LU	LB
NE	PB	PB	PB	PM	PM	PS	PS
ZR	PM	PM	PM	PS	PS	PS	NS
PO	NB	NB	NS	NS	NS	NS	NS

FIGURE 9.19 FAM bank of the fuzzy truck-and-trailer control system

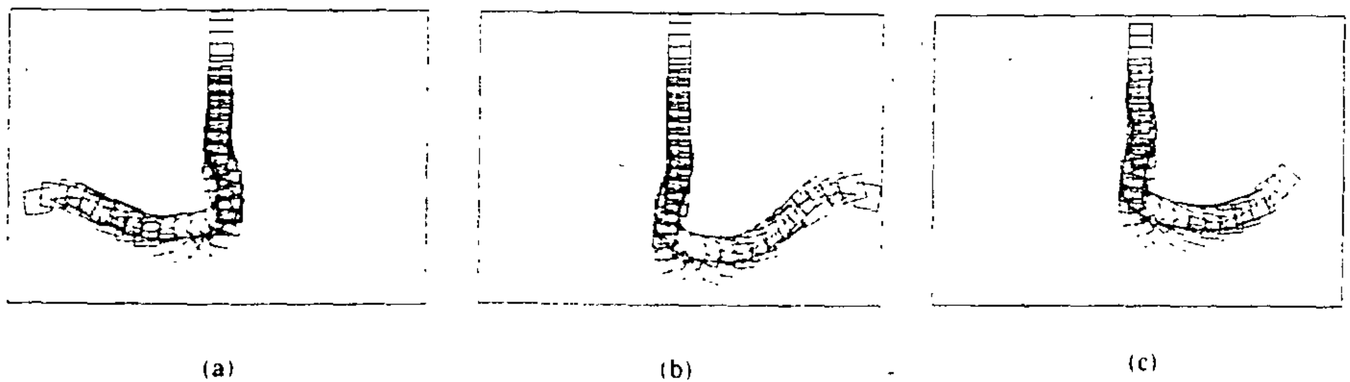


FIGURE 9.20 Sample truck-and-trailer trajectories from the fuzzy controller for initial positions (x, y, ϕ_t, ϕ_c). (a) (25, 30, −20, 30), (b) (80, 30, 210, −40), and (c) (70, 30, 200, 30).

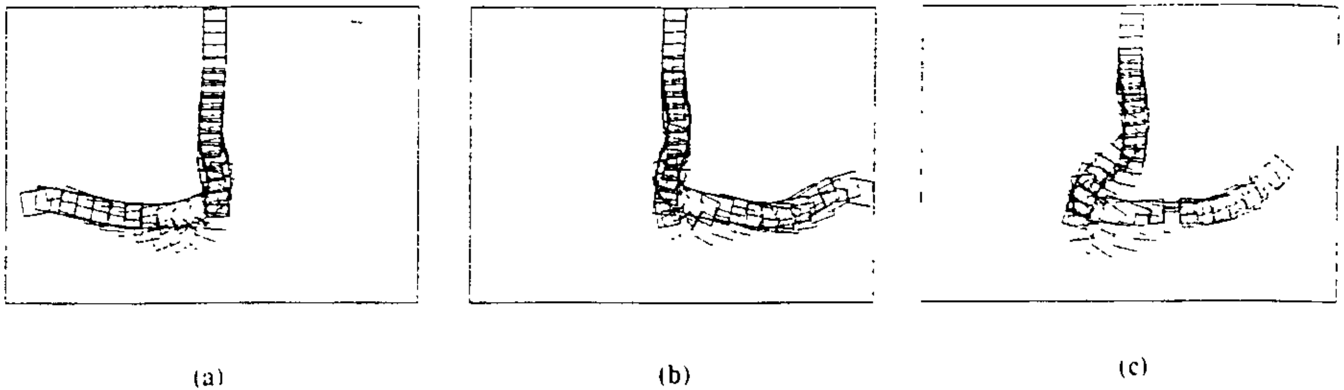


FIGURE 9.21 Sample truck-and-trailer trajectories of the BP-trained controller for initial positions (x, y, ϕ_t, ϕ_c): (a) (25, 30, −20, 30), (b) (80, 30, 210, −40), and (c) (70, 30, 200, 30).

Figure 9.20 shows typical backing-up trajectories of the fuzzy truck-and-trailer control system from different initial positions. The truck-and-trailer backed up in different directions, depending on the relative position of the cab with respect to the trailer. The fuzzy control systems successfully controlled the truck-and-trailer in jackknife positions.

BP Truck-and-Trailer Control Systems

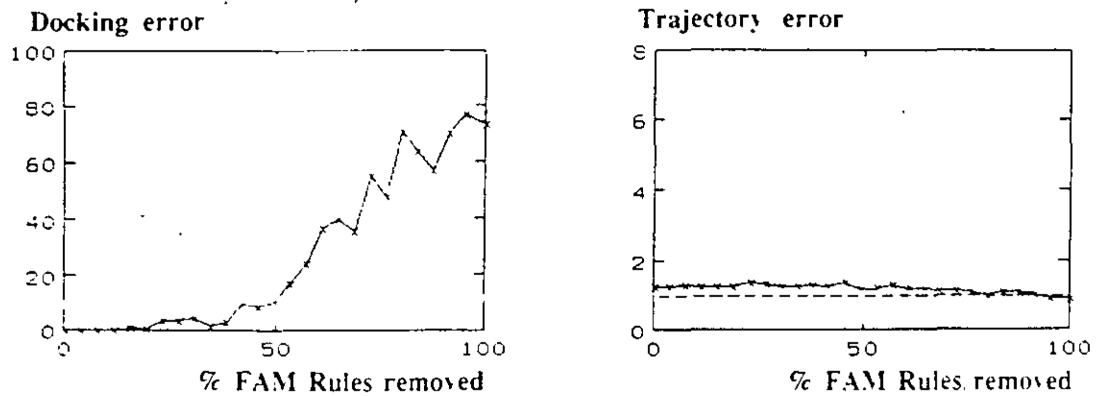
We added the cab-angle variable ϕ_c to the backpropagation-trained neural truck controller as an input. The controller network contained 24 hidden neurons with output variable β. The training samples consisted of five-dimensional space of the form (x, y, ϕ_t, ϕ_c, β). We trained the controller network with 52 training samples from the fuzzy controller: 26 samples for the left half of the plane, 26 samples for the right half of the plane. We used equations (9-9)–(9-14) instead of the emulator network. Training required more than 200,000 iterations. Some training sequences did not converge. The BP-trained controller performed well except in a few cases. Figure 9.21 shows typical backing-up trajectories of the BP truck-and-trailer control system from the same initial positions used in Figure 9.20.

We performed the same robustness tests for the fuzzy and BP-trained truck-and-trailer controllers as in the trailerless truck case. Figure 9.22 shows performance errors averaged over ten typical back-ups from ten different initial positions. These performance graphs resemble closely the performance graphs for the trailerless truck systems in Figure 9.11.

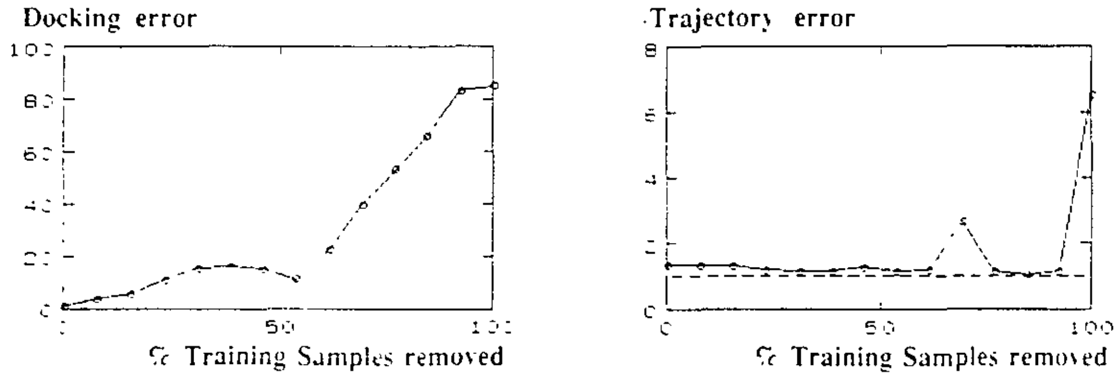
AFAM Truck-and-Trailer Control Systems

We generated 6250 truck-and-trailer data using the original FAM system in Figure 9.19. We backed up the truck-and-trailer from the same initial positions as in the trailerless-truck case. The trailer angle ϕ_t ranged from −60° to 240°, and the cab angle ϕ_c assumed only the three values −45°, 0°, and 45°. The training vectors

133



(a) Fuzzy truck-and-trailer



(b) BP-Neural truck-and-trailer

FIGURE 9.22 Comparison of robustness of the two truck-and-trailer controllers. (a) Docking and trajectory error of the fuzzy controller. (b) Docking and trajectory error of the BP controller

(x, ϕ_t, ϕ_c, β) defined points in the four-dimensional input-output product space. We nonuniformly partitioned the product space into FAM cells to allow narrower fuzzy-set values near the steady-state FAM cell.

We used DCL to train the AFAM truck-and-trailer controller. The total number of FAM cells equaled 735 ($7 \times 5 \times 7 \times 3$). We used 735 synaptic quantization vectors. The DCL algorithm classified the 6250 data into 105 FAM cells. We treated the trailerless-truck fuzzy variables as before. For the cab angle ϕ_c, we divided the space $-90 \leq \phi_c \leq 90$ into three intervals $[-90, -12.5]$, $[-12.5, 12.5]$, and $[12.5, 90]$, which corresponded to NE, ZR, and PO. There were 735 FAM cells, and 735 possible FAM rules, of the form (x, ϕ_t, ϕ_c; β). Figure 9.23 shows the synaptic-vector histogram corresponding to the 105 FAM rules. Figure 9.24 shows the estimated FAM bank by the DCL algorithm. Figure 9.25 shows the original and DCL-estimated control surfaces for the fuzzy truck-and-trailer systems.

Figure 9.26 shows the trajectories of the original FAM and the DCL-estimated AFAM truck-and-trailer controllers. Figures 9.26(a) and (b) show the two trajectories from the initial position (x, y, ϕ_t, ϕ_c) = (30, 30, 10, 45). Figures 9.26(c) and (d) show the trajectories from initial position (60, 30, 210, -60). The original FAM and DCL-estimated AFAM systems exhibited comparable truck-and-trailer control

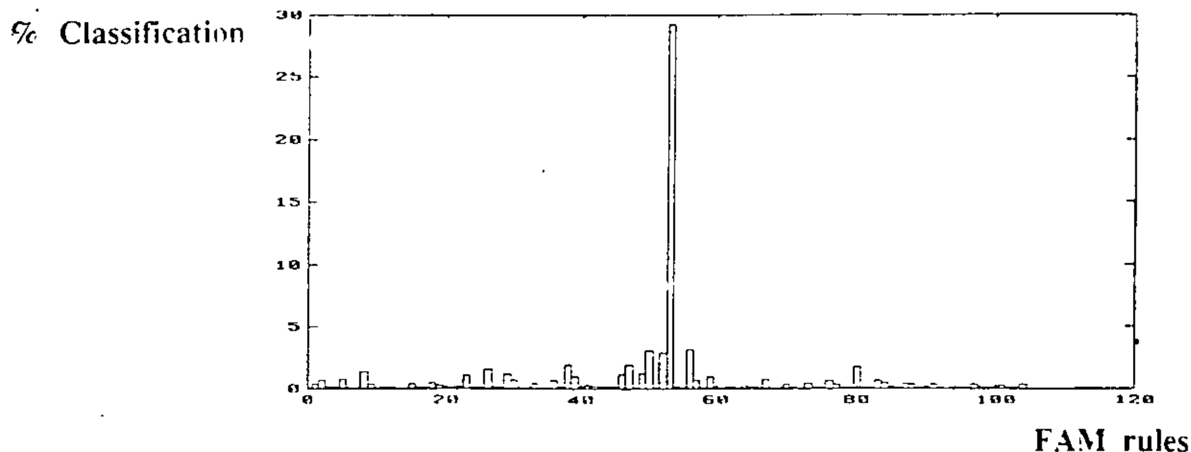


FIGURE 9.23 Synaptic-vector histogram for the AFAM truck-and-trailer system.

ϕ_t (columns), ϕ_c (rows)

x = LE

	RB	RU	RV	VE	LV	LU	LB
NE	PS	PS	PS	PS	PB	PB	PB
ZR	PS	ZE	NS	NS	NM	NB	NB
PO	ZE	NS	NM	NM	NB	NB	NB

x = LC

	RB	RU	RV	VE	LV	LU	LB
NE	PM	PS	PS	PS	PB	PS	PS
ZR	PM	PS	ZE	NM	NS	NM	NM
PO	NS	NS	NS	NM	NB	NB	NB

x = CE

	RB	RU	RV	VE	LV	LU	LB
NE	PM	PM	PM	PS	PS	PS	PS
ZR	PM	PM	PS	ZE	NS	NM	NM
PO	NS	NS	NS	NM	NM	NB	NM

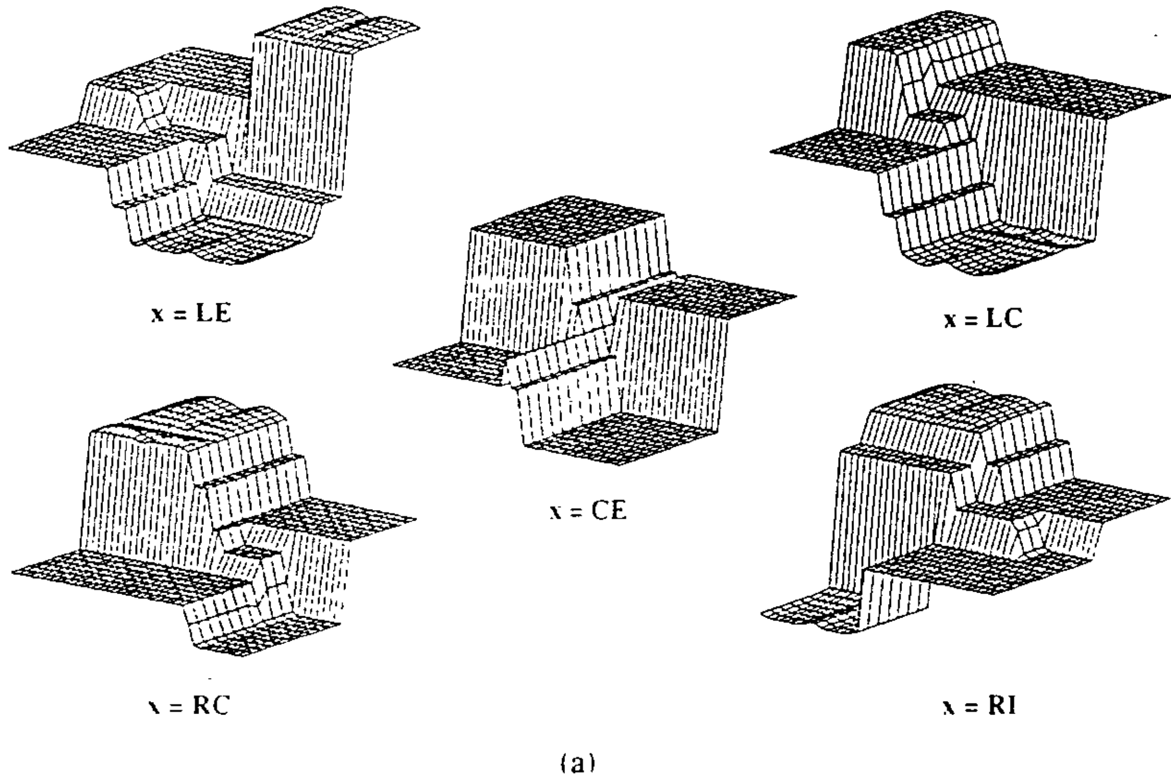
x = RC

	RB	RU	RV	VE	LV	LU	LB
NE	PB	PB	PM	PM	PS	PS	PS
ZR	PB	PB	PB	PS	PS	ZE	NM
PO	NB	NB	NB	NS	NS	NS	NM

x = RI

	RB	RU	RV	VE	LV	LU	LB
NE	PB	PB	PB	PM	PS	PS	ZE
ZR	PB	PB	PM	PS	PS	PS	ZE
PO	NB	NB	NB	NS	NS	NS	NS

FIGURE 9.24 DCL-estimated FAM bank for the AFAM truck-and-trailer system.



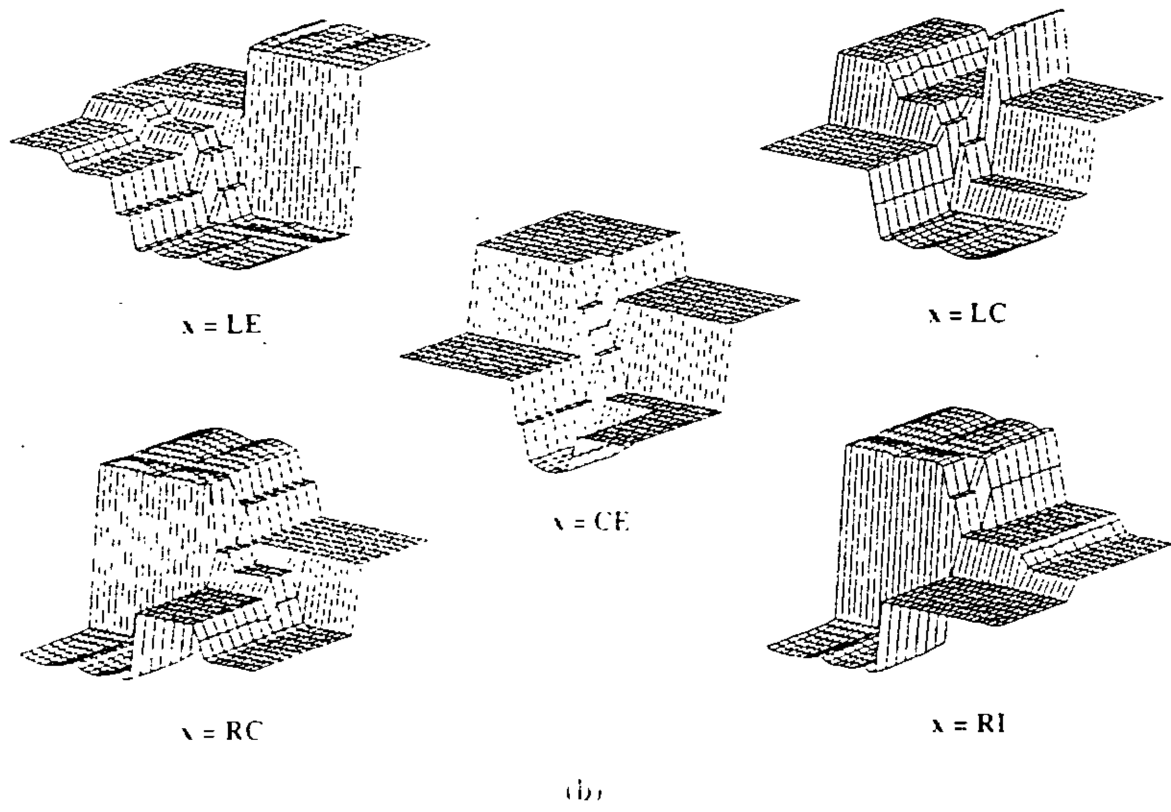
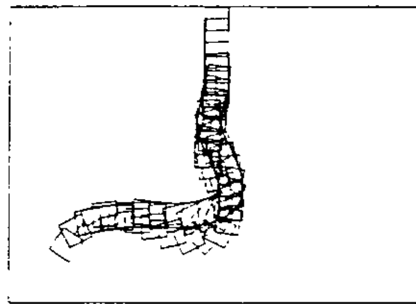
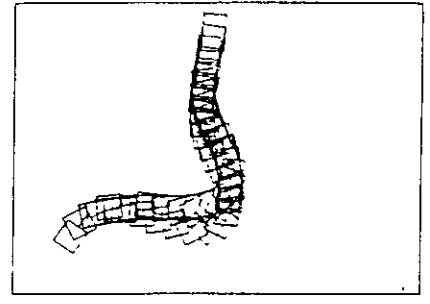


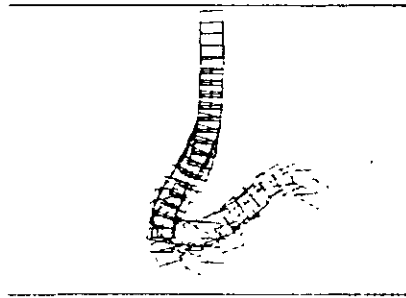
FIGURE 9.25 (a) Original control surface. (b) DCL-estimated control surface.



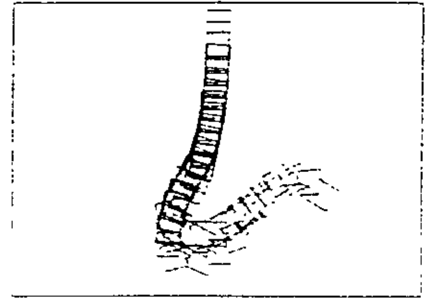
(a) Original FAM



(b) DCL-estimated FAM



(c) Original FAM



(d) DCL-estimated FAM

FIGURE 9.26 Sample truck-and-trailer trajectories from the original and the DCL-estimated FAM systems starting at initial positions (x, y, ϕ_t, ϕ_c) = (30, 30, 10, 45) and (60, 30, 210, −60).

performance except in a few cases, where the DCL-estimated AFAM trajectories were irregular.

Conclusion

We quickly engineered fuzzy systems to successfully back up a truck and truck-and-trailer system in a parking lot. We used only common sense and error-nulling intuitions to generate sufficient banks of FAM rules. These systems performed well until we removed over 50 percent of the FAM rules. This extreme robustness suggests that, for many estimation and control problems, different fuzzy engineers can rapidly develop prototype fuzzy systems that perform similarly and well.

The speed with which the DCL clustering technique recovers the underlying FAM bank further suggests that we can likewise construct fuzzy systems for more complex, higher-dimensional problems. For these problems we may have access to only incomplete numerical input-output data. Pure neural-network or statistical-process-control approaches may generate systems with comparable performance. But these systems will involve far greater computational effort, will be more difficult to modify, and will not provide a structured representation of the systen throughput.

Our neural experiments suggests that whenever we model a system with a

neural network, for little extra computational cost we can generate a set of structured FAM rules that approximate the neural system's behavior. We can then tune the fuzzy system by refining the FAM-rule bank with fuzzy-engineering rules of thumb and with further training data.

REFERENCES

Kosko, B., "Fuzzy Entropy and Conditioning," *Information Sciences*, vol. 40, 165-174, 1986.

Kosko, B., *Neural Networks for Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Nguyen, D., and Widrow, B., "The Truck Backer-Upper: An Example of Self-Learning in Neural Networks," *Proceedings of International Joint Conference on Neural Networks (IJCNN-89)*, vol. II, 357-363, June 1989.

the trajectory of computer programs produced by genetic programming are typical of the early and intermediate results produced by genetic programming in three important ways.

First, the admittedly poor performance in the actual problem domain of early programs produced by genetic programming is better than the completely nonworking early programs produced by the human.

Second, the performance of the best-of-generation program from generation 3 is slightly better in the actual problem domain than the performance of the best-of-generation program from generation 1. It, in turn, was better than the best of generation 0. There was no great leap in performance in genetic programming as there was between the human programmer's second-to-last program and his final program. Instead, genetic programming generally works by making small evolutionary changes that produce relatively small incremental improvements when measured via the natural metric of the space in which the results are stated.

Third, while the control strategy from generation 3 is admittedly suboptimal, one might conceivably use it to center a cart. The intermediate results produced by genetic programming do not work for all combinations of inputs and are certainly not time-optimal; however, they are often somewhat good at the task at hand. Similarly, the intermediate results of the evolutionary process in nature all possess the minimal level of performance required for survival, even if some fitter organism is yet to evolve. The intermediate results produced by human programmers are usually not usable at all.

Human programmers employ an entirely different style of programming (arising from their use of human intelligence and their knowledge of the problem) and make entirely different kinds of mistakes. When considering programs that do not work the first time, the trajectory of programs produced by a human programmer through the space of computer programs is very different from the trajectory produced by genetic programming.

The cart centering problem was previously studied in the field of genetic algorithms in connection with classifier systems by Goldberg (1983) and in the field of genetic programming by Koza and Keane (1990a).

7.2 Artificial Ant

As a second illustration of genetic programming, consider the task of navigating an artificial ant attempting to find all the food lying along an irregular trail as described in subsection 3.3.2 (Jefferson et al. 1991; Collins and Jefferson 1991a, 1991b). The problem involves primitive operations enabling the ant to move forward, turn right, turn left, and sense food along the irregular Santa Fe trail (figure 3.6).

When Jefferson, Collins, et al. used the conventional genetic algorithm operating on strings to find the finite-state automaton to solve this problem, it was first necessary to develop a representation scheme that converted the potential automaton into binary strings of length 453. In genetic program-

ming, the problem can be approached and solved in a far more direct way using the natural terminology of the problem.

The first major step in preparing to use genetic programming is to identify the set of terminals; the second is to identify the set of functions.

In the cart centering problem, the computer program processed information about the current state of the system in order to generate a control variable to drive the future state of the system toward a specified target state. In this problem, we are not primarily concerned with the values of the three overt state variables of the ant (i.e., the numerical values, between 1 and 32, of the vertical and horizontal position of the ant on the grid and the direction the ant is facing). Instead, we are primarily concerned with finding food. And to find food, we must make use of the very limited amount of information about food coming from the ant's sensor.

In this problem, the information we want to process is the information coming in from the outside world via the ant's very limited sensor. Thus, one reasonable approach to this problem is to place the conditional branching operator `IF-FOOD-AHEAD` into the function set. The `IF-FOOD-AHEAD` conditional branching operator takes two arguments and executes the first argument if (and only if) the ant senses food directly in front of it, but executes the second argument if (and only if) the ant does not sense any food directly in front of it. The `IF-FOOD-AHEAD` conditional branching operator is implemented as a macro as described in subsection 6.1.1.

If the function set for this problem contains an operator that processes information, the terminal set for this problem should then contain the actions which the ant should execute based on the outcome of this information processing. Thus, the terminal set for this problem is

```
T = {(MOVE), (RIGHT), (LEFT)}.
```

These three terminals correspond directly to the three primitive functions defined and used by Jefferson, Collins, et al. to change the state of the ant. Since these three terminals are actually functions taking no arguments, their names are enclosed in parentheses. These three primitive functions operate via their side effects on the ant's state (i.e., the ant's horizontal and vertical position on the grid and the ant's facing direction). These three terminals evaluate to 1; however, their numeric return values are not relevant for this problem.

Recall that in the state-transition diagram for the finite-state automaton (figure 3.7), there were two lines emanating from each circle. The two lines represented the two alternative state transitions associated with the two possible sensor inputs of the ant. The `IF-FOOD-AHEAD` conditional branching operator implements these same two alternatives here. Recall also that there was one unconditional state transition in the state-transition diagram of the finite-state automaton. The Common LISP connective `PROGN` provides a connective glue for implementing such an unconditional sequence of steps. For example, the two-argument `PROGN` connective (also often called `PROGN2` in this book) in the S-expression

```
(PROGN (RIGHT) (LEFT))
```

causes the ant to unconditionally perform the sequence of turning to the right and then turning to the left.

Therefore, the function set for this problem is

```
F = {IF-FOOD-AHEAD, PROGN2, PROGN3},
```

taking two, two and three arguments, respectively. Note that we include the PROGN connective in the function set twice (once for two arguments and once for three arguments)

The third major step in preparing to use genetic programming is to identify the fitness measure The natural measure of the fitness of a given computer program in this problem is the amount of food eaten within some reasonable amount of time by an ant executing the given program. Each move operation and each turn operation takes one step. In our version of this problem, we limited the ant to 400 time steps. This time-out limit is sufficiently small in relation to 1,024 to prevent a random walk or a tesselating movement from covering all 1,024 squares of the grid before timing out

Thus, the raw fitness of a computer program for this problem is the amount of food (ranging from 0 to 89) that the ant has eaten within the maximum allowed amount of time. If a program times out, its raw fitness is the amount of food eaten up to that time.

Time was computed here in the same way as in the work of Jefferson, Collins, et al. That is, the three primitive functions RIGHT, LEFT, and MOVE each take one time step to execute, whereas the IF-FOOD-AHEAD conditional branching operator and the unconditional connectives PROGN2 and PROGN3 each take no time steps to execute

For the cart centering problem, a smaller raw fitness (time) was better. For this problem, a bigger raw fitness (food eaten) is better Standardized fitness is a measure of fitness for which a smaller value is better than a larger value Thus for this problem the standardized fitness is the maximum attainable value of raw fitness (i.e., 89) minus the actual raw fitness A standardized fitness of 0 corresponds to a perfect solution for this problem (and many problems) For the cart centering problem standardized fitness was identical to raw fitness

For this problem, the auxiliary hits measure was defined to be the same as raw fitness. The hits measure was then used as part of the termination criterion A run of this problem is terminated if any S-expression attains 89 hits or when the maximum allowed number of generations ($G = 51$) have been run

Potentially, the fitness cases for this problem consist of all the possible combinations of initial conditions for the ant (i.e. the initial starting positions and the initial facing directions) along with all reasonable generalizations of the Santa Fe trail (i.e. trails with single gaps double gaps, single gaps at corners double gaps at corners, and triple gaps at corners appearing in any order) Note however that we do not explicitly create a multiplicity of fitness cases for this problem as we did for the cart centering problem Instead, we have just one fitness case wherein the ant starts at position (0, 0) while facing east and tries to navigate just one trail We rely on the various states

141

of the ant that actually arise along the ant's actual trajectory to be sufficiently representative of the generalized trail following problem. As we will see, this one fitness case is sufficiently representative for this particular problem to allow the ant to learn to navigate this trail and reasonable generalizations of this trail.

It should be emphasized that genetic programming genetically breeds computer programs that have high fitness in grappling with the environment (i.e., they score a high fitness for the explicit fitness cases on which they are run). The programs produced by genetic programming will generalize in the sense that they are useful in solving other problems which a human, in his mind, may envision only if the fitness cases that are chosen are sufficiently representative of the generalization envisioned by the human.

Table 7.3 summarizes the key features of the artificial ant problem for the Santa Fe trail.

The run starts with the generation of 500 random computer programs recursively composed from the available functions and terminals. Predictably, this initial population of random computer programs includes a wide variety of highly unfit computer programs. The random computer programs in generation 0 of this problem (as well as the random programs in generation 0 of the preceding cart centering problem and all later problems in this book) correspond to the computer programs that might be typed out at random by the proverbial monkeys. These random computer programs provide a baseline for comparing the more satisfactory performance achieved by genetic programming in later generations against random performance.

The most common type of individual in the initial random population for this problem fails to move at all. For example, the computer program

Table 7.3 Tableau for the artificial ant problem for the Santa Fe trail

Objective	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe trail
Terminal set:	(LEFT), (RIGHT), (MOVE).
Function set:	IF-FOOD-AHEAD, PROGN2, PROGN3.
Fitness cases	One fitness case
Raw fitness	Number of pieces of food picked up before the ant times out with 400 operations.
Standardized fitness	Total number of pieces of food (i.e., 89) minus raw fitness
Hits	Same as raw fitness for this problem
Wrapper	None
Parameters	$M = 500$. $G = 51$
Success predicate	An S-expression scores 89 hits

142

```
(PROGN2 (RIGHT) (LEFT))
```

turns without looking. It unconditionally turns the ant right and left while not moving the ant anywhere.

Similarly, the program

```
(IF-FOOD-AHEAD (RIGHT) (LEFT))
```

looks without moving. It examines the outside world and then turns the ant different ways on the basis of what it saw, however, it does not move the ant anywhere.

Neither of these highly unfit individuals eats any of the 89 pieces of food. They are mercifully terminated by the expiration of the maximum allowed time.

Some randomly generated computer programs move without turning. For example, the program

```
(PROGN2 (MOVE) (MOVE))
```

shoots across the grid from west to east without either looking or turning This vigorous undirected behavior accidently finds the three pieces of food located on the top row of the grid

One randomly generated computer program (which will be called the "quilter" because it traces a quilt-like tessellating pattern across the toroidal grid) moves and turns without looking. It consists of nine points

```
(PROGN3 (RIGHT)
        (PROGN3 (MOVE) (MOVE) (MOVE))
        (PROGN2 (LEFT) (MOVE))).
```

Note that, in this problem, the entire S-expression is executed as fully as possible and then re-executed until the maximum allowed amount of time is consumed

Figure 7 12 shows the first part of the quilter's path This part of the quilter's path is marked by X s The quilter accidentally finds four pieces of food in the portion of its path shown

One randomly generated computer program (the "looper") finds the first 11 pieces of food on the trail and then goes into an infinite loop when it encounters the first gap in the trail In figure 7 13, the looper's path is marked by X's The raw fitness of the looper is 11

One randomly generated computer program (the "avoider") actually correctly takes note of the portion of food along the trail before finding the first gap in the trail, then actively avoids this food by carefully moving around it until it returns to its starting point It continues with this unrewarding behavior until the time runs out and never eats any food

The S-expression for the avoider has seven points

```
(IF-FOOD-AHEAD (RIGHT)
               (IF-FOOD-AHEAD (RIGHT)
                              (PROGN2 (MOVE) (LEFT)))).
```

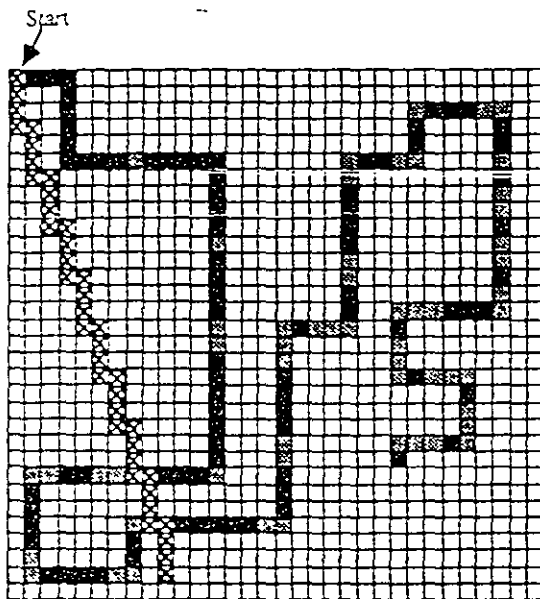


Figure 7.12 Path of the quilter from generation 0 of the artificial ant problem

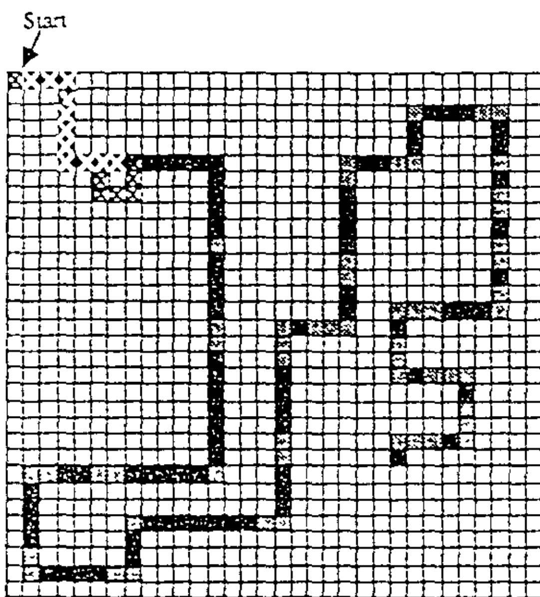


Figure 7.13 Path of the looper from generation 0 of the artificial ant problem

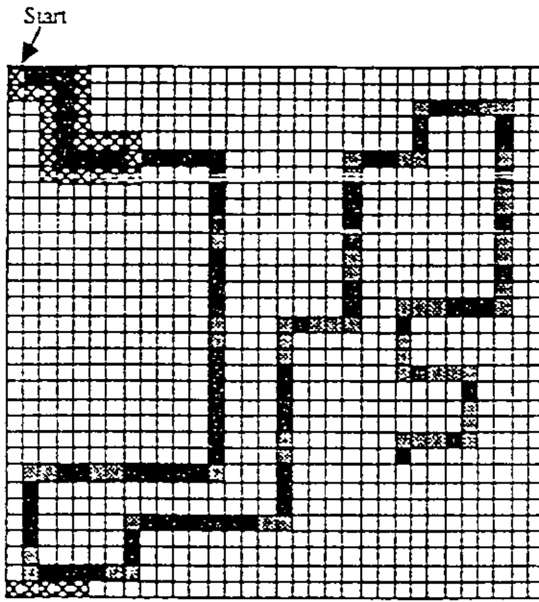


Figure 7.14 Path of the avoider from generation 0 of the artificial ant problem

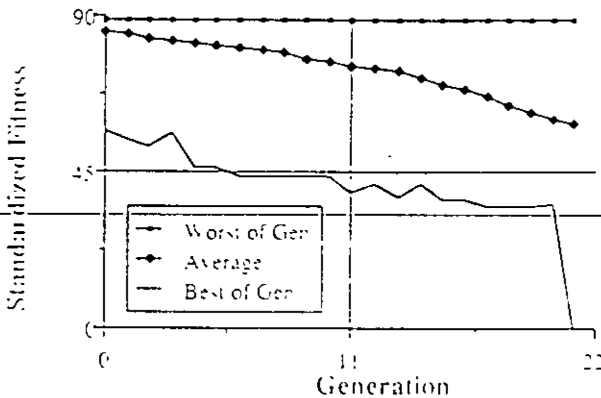


Figure 7.15 Fitness curves for the artificial ant problem

In figure 7.14, the avoider's path is marked by Xs.

In one run, the average amount of food found by the 500 individuals in the initial random population was about 3.5 pieces. The best-of-generation individual in generation 0 was able to find 32 of the 89 pieces of food; the worst individuals in the population found no food.

The Darwinian reproduction operation and the genetic crossover operation were then applied to parents selected from the current population with probabilities proportionate to fitness to breed a new population of offspring computer programs.

Figure 7.15 shows, by generation, the standardized fitness of the best-of-generation individual and the worst-of-generation individual for one run of the artificial ant problem. It also shows the average value of standardized fitness

IF-FOOD-AHEAD test. The ant first turns LEFT. Then, a two-step PROGN2 sequence begins with the test IF-FOOD-AHEAD. If food is present, the ant MOVES forward. If not, the ant turns RIGHT. Then, the ant turns RIGHT again. Then, the ant pointlessly turns LEFT and RIGHT in another two-step PROGN2 sequence. The net effect is that the ant is now facing right relative to its original facing direction (i.e., its direction at the beginning of the execution of this S-expression). The ant next executes the final two-step PROGN2 subtree at the far right of the figure. If the ant now senses food via the IF-FOOD-AHEAD test, it MOVES forward. Otherwise, it turns LEFT. The ant has now returned to its original facing direction. The ant now executes an unconditional MOVE, thereby advancing forward in its original facing direction if it has not found any food to the immediate right or left.

For any given evaluation of the S-expression, only those subtrees that are accessible by virtue of satisfaction of the conditional part of the IF-FOOD-AHEAD test are actually evaluated (i.e., executed). After the S-expression is evaluated, if there is additional time available, the S-expression is evaluated anew. Because the state of the ant changes over time as the ant moves and eats food, different parts of the S-expression are often executed on each evaluation. The repeated application of the above 100%-correct program allows the ant to negotiate all the gaps and irregularities of the trail and to eat all the food in the allotted time.

Note that there is no testing of the backward directions. See also Koza 1990c.

The pointless two-step PROGN2 subtree at the bottom of figure 7.16, where the ant unconditionally turns LEFT and RIGHT, does not harm the ant's performance. the ant is able to find 100% of the available food within the allowed maximum amount of time. Fitness in this problem was defined to be the amount of food eaten within the allowed time. The best-of-run individual from generation 21 was genetically bred with this fitness measure as the driving force. This fitness measure did not incorporate anything about minimizing the size of the S-expression or minimizing the total number of steps, except in the indirect sense that a highly inefficient individual could not find 100% of the food within the available time. Among individuals that can find 100% of the food within the available time, there is no selective pressure whatsoever in favor of efficiency or parsimony.

Humans prefer to organize their conscious thinking in a parsimonious way, however, fitness, not parsimony, is the dominant factor in natural evolution. For example, only about 1% of the sequences of nucleotide bases that occur naturally along DNA are actually expressed into the sequences of amino acids that make up the proteins that perform the work of living organisms. In addition, after a sequence of DNA is actually expressed (via messenger ribonucleic acid, mRNA) into a string of amino acids, the resulting protein structure is rarely maximally parsimonious. The human hemoglobin molecule, for example, weighs about 60,000 daltons (i.e., equivalent hydrogen atoms), yet its primary role is to transport only eight oxygen molecules from the lungs to the body cells. There is almost certainly some variation on the design of this

molecule that is at least slightly smaller than 60,000 daltons. However, if this molecule successfully performs its task, there may no fitness advantage, and hence no selective pressure, in favor of attaining the most parsimonious possible design

Secondary factors, such as efficiency and parsimony, can be incorporated into fitness measures (sections 18.1 and 25.14), but this was not done here.

Note again that in applying genetic programming to this problem we made no assumption in advance about the size, the shape, or the structural complexity of the eventual solution. The solution found above in generation 21 had 18 points. We did not specify that the solution would have 18 points, nor did we specify the shape or the contents of this 18-point S-expression. The size, shape, and contents of the 100%-correct S-expression for this problem evolved in response to the selective pressure provided by the fitness measure (i.e., the amount of food eaten). The required structure emerged from a process driven by the selective pressure exerted by the fitness measure. For this problem, structure flowed from fitness, just as it does in nature.

It is also interesting to consider the artificial ant problem with a more difficult trail.

The new "Los Altos Hills" trail begins with the same irregularities (i.e., single gaps, double gaps, single gaps at corners, double gaps at corners, and triple gaps at corners), in the same order, as the Santa Fe trail. However, the new trail has two new kinds of irregularity, which appear toward its end. Because of these added features, this new trail is embedded in a larger 100 × 100 grid and spacing has been added between the branches of the trail.

Figure 7.17 shows the Los Altos Hills trail for the artificial ant problem situated in the upper left 50 × 70 portion of the 100 × 100 grid. In this new trail, food pellet 105 corresponds to food pellet 89 (i.e., the end) of the Santa Fe trail.

The simpler of the two new irregularities in the Los Altos Hills trail requires a search of locations two steps to the left or two steps to the right of an existing piece of food. This first new irregularity appears for the first time at food pellet 116 in figure 7.17. The previously evolved program that successfully navigates the Santa Fe trail cannot handle this irregularity, since it does not regard a location that is two steps off the trail as being part of the trail. If the artificial ant masters this new irregularity, it can find 136 pieces of food.

The more difficult of the two new irregularities in the Los Altos Hills trail requires moving one step ahead and then searching locations two steps to the left or two steps to the right of an existing piece of food. The second new irregularity appears for the first time at food pellet 136 in the figure. If the artificial ant masters both of these two new irregularities it can find 157 pieces of food.

We approach this upwardly scaled version of the problem in the same way as we approached the simpler version. In particular, we use the same terminal set, the same basic function set, and the same fitness measure. We increase the available time steps to 3,000. This number is sufficiently small in relation to 10,000 to prevent a random walk or any simple tesselating movement from

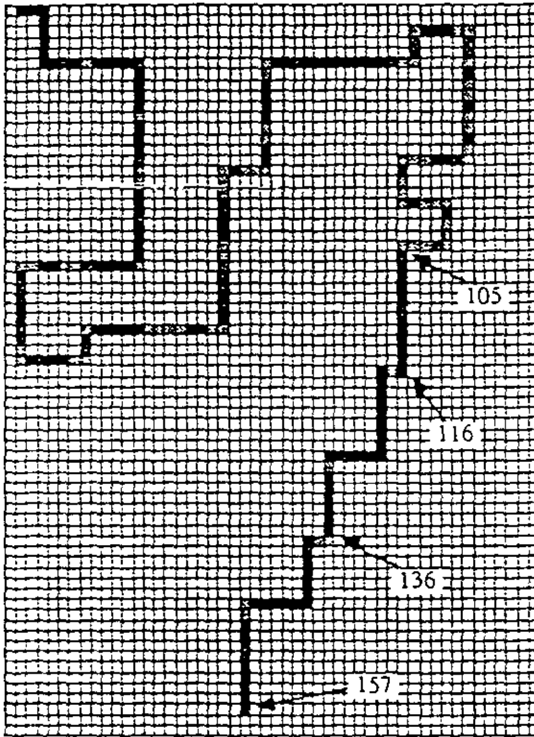


Figure 7.17 The Los Altos Hills trail for the artificial ant

finding all the food merely by visiting all 10,000 squares of the grid. We added PROGN4 to the basic function set. We increased the population size to 2,000.

Figure 7.18 shows, by generation, the average of the standardized fitness for the population as a whole and the standardized fitness of the best-of-generation and worst-of-generation individuals for one run of the artificial ant problem with the Los Altos Hills trail.

In one run (in fact, our first run of this scaled-up version of the problem), the following S-expression was obtained on generation 19. This best-of-run individual is capable of finding all 157 pieces of food on this new Los Altos Hills trail within 1,808 time steps.

```
(PROGN4 (IF-FOOD-AHEAD (PROGN2 (PROGN3 (MOVE) (PROGN2
(MOVE) (MOVE)) (RIGHT)) (IF-FOOD-AHEAD (MOVE)
(IF-FOOD-AHEAD (IF-FOOD-AHEAD (LEFT) (LEFT))
(PROGN4 (PROGN2 (IF-FOOD-AHEAD (MOVE) (RIGHT))
(MOVE)) (RIGHT) (MOVE) (MOVE))))) (PROGN4 (PROGN2
(IF-FOOD-AHEAD (MOVE) (RIGHT)) (MOVE)) (RIGHT)
(MOVE) (MOVE)) (IF-FOOD-AHEAD (IF-FOOD-AHEAD (MOVE)
(IF-FOOD-AHEAD (IF-FOOD-AHEAD (MOVE) (LEFT))
(IF-FOOD-AHEAD (LEFT) (RIGHT)))) (IF-FOOD-AHEAD
(LEFT) (RIGHT))) (PROGN2 (PROGN3 (MOVE) (MOVE)
(RIGHT)) (IF-FOOD-AHEAD (PROGN2 (PROGN3 (MOVE)
```

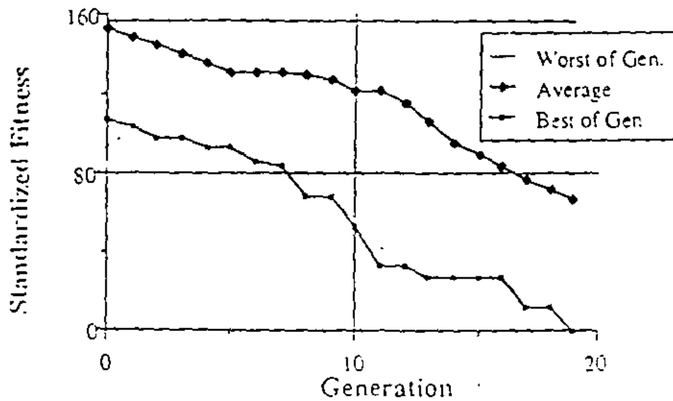


Figure 7.16 Fitness curves for the artificial ant problem with the Los Altos Hills trail

```
(PROGN2 (MOVE) (MOVE)) (RIGHT)) (IF-FOOD-AHEAD
(IF-FOOD-AHEAD (MOVE) (MOVE)) (MOVE))) (MOVE))) (MOVE)).
```

The above best-of-run S-expression contains 66 points. Not surprisingly, solving the more difficult Los Altos Hills trail required an S-expression with more internal and external points than the solution for the original Santa Fe trail. This individual can be simplified to the following

```
(PROGN7 (IF-FOOD-AHEAD
          (PROGN5 (MOVE) (MOVE) (MOVE) (RIGHT)
                  (IF-FOOD-AHEAD
                    (MOVE)
                    (PROGN5 (RIGHT) (MOVE)
                            (RIGHT)
                            (MOVE) (MOVE))))
          (PROGN5 (RIGHT) (MOVE) (RIGHT)
                  (MOVE) (MOVE)))
        (IF-FOOD-AHEAD (MOVE) (RIGHT))
        (MOVE)
        (MOVE)
        (RIGHT)
        (IF-FOOD-AHEAD
          (PROGN5 (MOVE) (MOVE) (MOVE) (RIGHT)
                  (MOVE))
          (MOVE))
        (MOVE))
```

Whenever this best-of-run individual encounters any irregularity in the trail (and occasionally when it does not), this S-expression causes the ant to make a loop that is three squares wide and two squares long. This looping action allows the ant to successfully navigate the two new kinds of irregularities. This looping action is somewhat wasteful and inefficient; however, it works. That is, this S-expression finds 100% of the food within the allowed amount of time and therefore has maximal fitness given the fitness measure we are using.

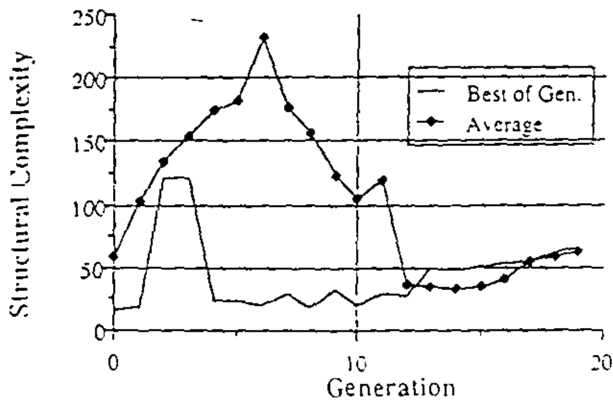


Figure 7.19 Structural complexity curves for the artificial ant problem with the Los Altos Hills trail

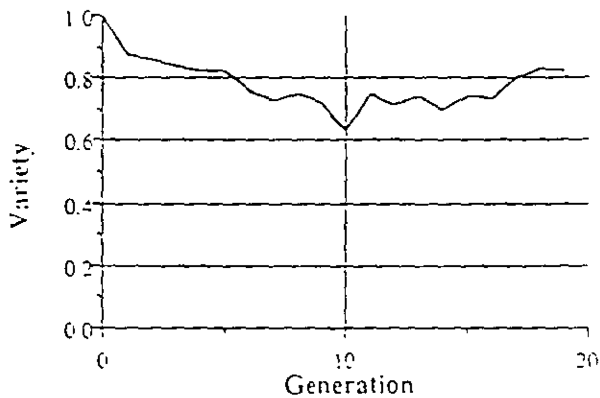


Figure 7.20 Variety curve for the artificial ant with the Los Altos Hills trail

This problem is typical of most problems in that the structural complexity (i.e. total number of function points and terminal points) of the average S-expression in the population increases in later generations of a given run.

Figure 7.19 contains the *structural complexity curves* for this problem. It is one of 13 similar curves found in this book. It shows, by generation, the average of the structural complexity of the population as a whole and the structural complexity of the best-of-generation individual for one run of the artificial ant problem with the Los Altos Hills trail with the primitive function RIGHT deleted. As can be seen, the total number of function points and terminal points of the best-of-generation individual starts at 16 for generation 0 and rises to 66 for generation 19.

Size, of course, is not the only contributor to the complexity of an organism; however, studying gross size and complexity is a first step in studying the evolution of complex structures (Bonner 1988).

Figure 7.20 is the *variety curve* showing the variety of the population, by generation, during one run of the artificial ant problem with the Los Altos Hills trail. This variety curve is one of nine similar curves found in this book. For this

problem, variety starts at 100% at generation 0 because duplicate checking is done when the initial random population is created. It then fluctuates around 80% for most of this particular run. The operation of fitness-proportionate reproduction is alone responsible for reducing variety after generation 0 by the probability p_r of reproduction (10% here). It is common for variety to dip for one generation whenever a small number of individuals have distinctly better fitness than the remainder of the population. Such a dip occurs at generation 10 of this run.

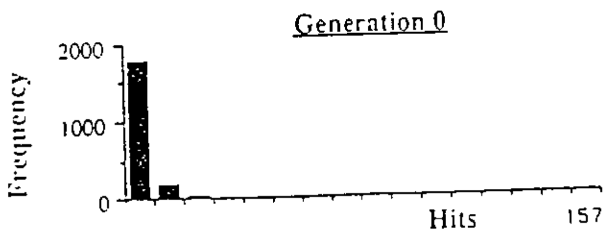
The *hits histogram* is a useful monitoring tool for the population as a whole for a particular generation. The horizontal axis of the hits histogram is the number of hits, the vertical axis is the number of individuals in the population scoring that number of hits. There are 10 sets of similar histograms throughout this book.

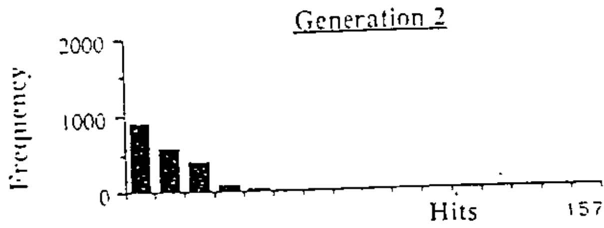
Figure 7.21 shows the hits histograms for five selected generations of this run. The first 15 ticks in the horizontal axis of the histogram represent a range of 150 levels of fitness between 0 and 149, the last tick represents the eight levels of fitness between 150 and 157. Note, in the progression from generation to generation, the left-to-right undulating movement of both the high point and the center of mass of the histogram. This "slinky" movement reflects the improvement of the population as a whole. The arrow marks the barely visible occurrence of one 100%-correct individual scoring 157 on generation 19.

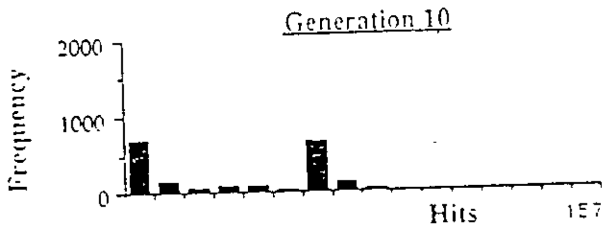
The selection of the terminal set and the selection of the function set are important steps in genetic programming because these sets provide the ingredients from which genetic programming attempts to build a solution. In general, the selection of these sets affects the appearance of the results, the ease of finding a solution, and, indeed, whether a solution can be found at all. For example, in the discussion above, we used both the RIGHT and LEFT primitive functions because that is how this problem was originally defined by Jefferson, Collins, et al. Since both the RIGHT and LEFT operations are obviously not needed, it is interesting to consider the artificial ant problem with the primitive function LEFT deleted. When the problem was rerun with this smaller set of primitive functions using the Santa Fe trail, genetic programming found the following solution in generation 19 of one run:

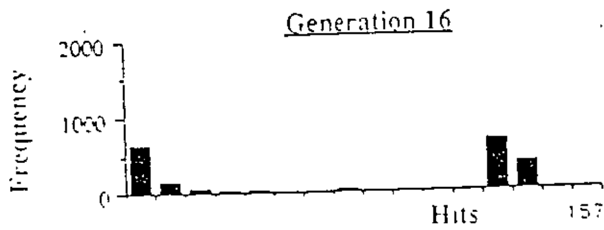
```
(PROGN2 (PROGN2 (IF-FOOD-AHEAD (MOVE) (RIGHT))
                (PROGN2 (MOVE) (RIGHT)))
        (PROGN2 (IF-FOOD-AHEAD
                  (PROGN3 (MOVE)
                          (PROGN2 (MOVE) (RIGHT))
                          (RIGHT))
                  (RIGHT))
                (IF-FOOD-AHEAD (RIGHT)
                               (RIGHT))))
```

This S-expression has 20 points and is a 100%-correct solution to the problem. For this particular problem, the removal of one superfluous primitive









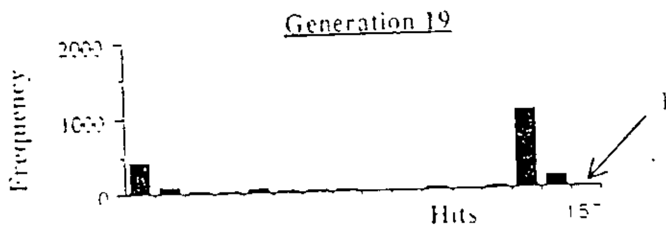


Figure 7.21 Hits histograms for generations 0, 2, 10, 16, and 19 for artificial ant problem with the Los Altos Hills trail

function does not substantially affect the performance of genetic programming (subsection 24.3.3).

7.3 SIMPLE SYMBOLIC REGRESSION

As a third illustration of genetic programming, consider a simple form of the problem of symbolic regression (symbolic function identification).

In linear regression, one is given a set of values of various independent variable(s) and the corresponding values for the dependent variable(s). The goal is to discover a set of numerical coefficients for a linear combination of the independent variable(s) that minimizes some measure of error (such as the square root of the sum of the squares of the differences) between the given values and computed values of the dependent variable(s). Similarly, in quadratic regression the goal is to discover a set of numerical coefficients for a quadratic expression that minimizes error. In Fourier "regression," the goal is to discover a set of numerical coefficients for various harmonics of the sine and cosine functions that minimizes error.

Of course, it is left to the researcher to decide whether to do a linear regression, a quadratic regression, a higher-order polynomial regression, or whether to try to fit the data points to some non-polynomial family of functions. But often, *the* issue is deciding what type of function most appropriately fits the data, not merely computing the numerical coefficients after the type of function for the model has already been chosen. In other words, the real problem is often *both* the discovery of the correct functional form that fits the data and the discovery of the appropriate numeric coefficients that go with that functional form. We call the problem of finding a function, in symbolic form, that fits a given finite sample of data *symbolic regression*. It is "data-to-function" regression. The desirability of doing regression without specifying in advance the functional form of the eventual solution was recognized by Dallemand (1958), Westervelt (1960), and Collins (1968).

For example, suppose we are given a sampling of the numerical values from a target curve over 20 points in some domain, such as the real interval [−1.0, +1.0]. That is, we are given a sample of data in the form of 20 pairs (x_i, y_i), where x_i is a value of the independent variable in the interval [−1.0, +1.0] and y_i is the associated value of the dependent variable. The 20 values of x_i were chosen at random in the interval [−1.0, +1.0]. For example, these 20 pairs (x_i, y_i) might include pairs such as (−0.40, −0.2784), (+0.25, +0.3320), ..., and (+0.50, +0.9375).

These 20 pairs (x_i, y_i) are the fitness cases that will be used to evaluate the fitness of any proposed S-expression.

The goal is to find a function, in symbolic form, that is a good or a perfect fit to the 20 pairs of numerical data points. The solution to this problem of finding a function in symbolic form that fits a given sample of data can be viewed as a search for a mathematical expression (S-expression) from a space of possible S-expressions that can be composed from a set of available functions and terminals.

LOCALIZATION OF UNKNOWN SOURCE OF SEISMIC VIBRATIONS

Vlastimir D. Pavlović[1] Zoran S. Veličković[2]
[1]Faculty of Electronic Engineering, Niš, Yugoslavia
[2]EI Profesionalna elektronika, Niš, Yugoslavia
VPAVLOVIC@EFNIS.ELFAK.NI.AC.YU

Abstract - *In this work, an effective technique for localizing the sources of seismic vibrations using the signals observed by the sensor array, is described. Evaluation of the source location is performed on the basis of the calculated time delays between the first arrival of vibrations, detected by the referent sensor, and the following arrivals, detected by the remaining sensors in the active sensor unit. This technique is a based on a circle method for determination of the source coordinates, and provides unique solution for the actual source coordinates in explicit form.*

1. INTRODUCTION

This work deals with an original approach to the problem of open area protection, which is shifted into the seismology domain. An unwanted object (man, group of people, vehicle, animal), moving in the area covered by seismic transducers, generates vibrations that are propagated through the ground and can be detected by seismic transducers. Seismic transducer converts seismic vibrations into the electric signals, which are processed using PC in order to extract information about the actual origin of vibrations. The paper presents the simulation results, obtained using the prescribed coordinates of the unknown source of vibrations, which are in close agreement with the previously adopted initial values.

2. SYSTEM DESCRIPTION

A source of vibrations generates seismic waves that are propagated through the ground. In Fig. 1, a basic unit of sensor array, consisting of three geophones placed at the apexes of equilateral triangle, is shown. The velocity of seismic wave propagation is assumed to be constant. Seismic waves are supposed to propagate radially without distortion. A time interval t required for the seismic wave to reach the sensors and to activate them after the moment when vibrations are generated, is calculated according to the following expression.

$$t = \frac{S}{v} \tag{1}$$

In the given formula, parameter v is a group velocity of the seismic waves that represents an intrinsic property of the ground and can be calculated for a particular type of the ground, and S is the distance between the actual source of vibrations and the sensor.

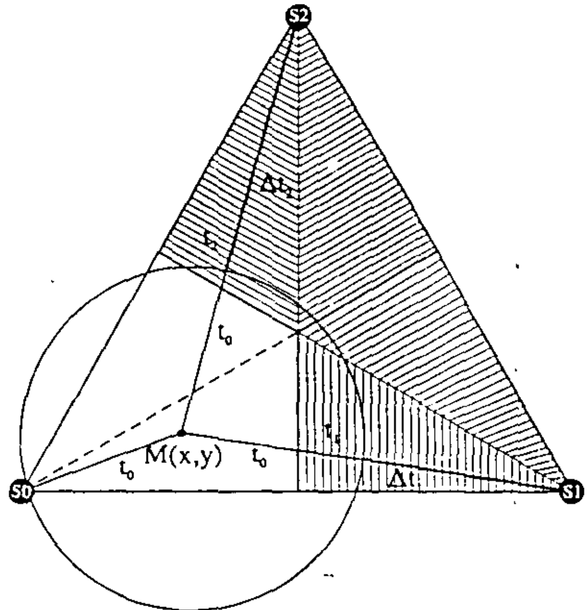


Figure 1. Triangle constellation of the sensor unit with M(x,y) as an active point

Figure 1 illustrates the sensor constellation. If the source of vibrations is located at an arbitrary point inside the triangle, propagation time intervals t_0, t_1 and t_2 are required for signals to reach the sensors S_0, S_1 and S_2, respectively. Figure 2 shows the waveforms of the real seismic signals obtained by the action of a man. The signals are detected by the sensors S_0, S_1, and S_2, respectively, and digitized using the 14-bit AD converter.

In this paper, in order to determine the location of a source of vibrations, the relative time delays are used instead of the total propagation time intervals t_0, t_1 and t_2. The only relevant information, obtained by choosing the closest sensor to be the dominant one, are the time delays between the first

and second arrival of signals s_0 and s_1, Δt_1, and the first and the third arrival of signals s_0 and s_2, Δt_2.

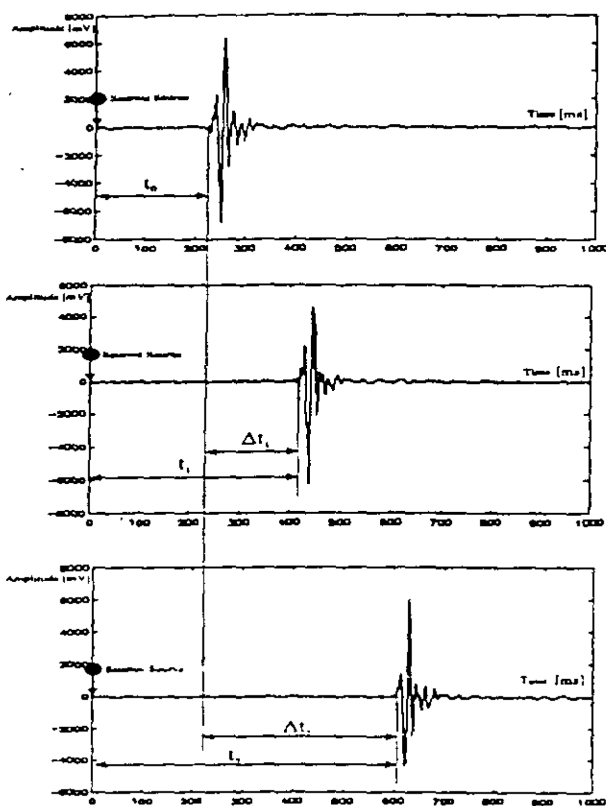


Figure 2. Waveforms of real seismic signals detected by sensors S_0, S_1 and S_2. Sampling frequenc Fs=1 kHz and N=1024.

For digital signals, cross-correlation function can be calculated directly according to the following expressions:

$$R_{0r}(k) = \sum_{n=-N}^{N} s_0(n) s_r(n+k),\ r = 1,2 \quad (2)$$
$$k = 1,2,\ldots,N$$

where S_r represent acquired sequences of the signals obtained by a single seismic exitacion. Since the ground, which represents a transmission medium for the seismic waves, is generally a nonlinear and dispersive medium, the signals detected by the seismic sensors differ in their waveforms. In this paper, the cross-correlation function, which has its maximum value at the moment equal to the required time delay, is used for time delay calculations

$$\Delta t_r = f\left\{\max\left[R_{0r}(k)\right]\right\},\ r = 1,2 \quad (3)$$

In our example, cross-correlation functions $R_{01}(k)$ and $R_{02}(k)$ are calculated according to the cross-correlation theorem for the discrete sequences using the inverse Fourier transform, which is given in the complex form:

$$R_{0r}(k) = IFFT\left[S_0(\omega) S_r^*(\omega)\right],\ r = 1,2 \quad (4)$$

The spectra $S_0(\omega)$, $S_1(\omega)$ and $S_2(\omega)$ are calculated using the Fast Fourier Transform (FFT):

$$S_j(\omega) = FFT\left[s_j(k)\right],\ j = 0,1,2 \quad (5)$$

The location of the unknown source of seismic vibrations can be evaluated using the relative time delays obtained in this way.

3. EVALUATION OF THE COORDINATES OF UNKNOWN SOURCE OF VIBRATIONS

Figure 3. shows the local coordinate system and the basic sensor unit. A triangle constellation is represented using the normalized coordinates. In reality, the distance between the sensors equals 50m.

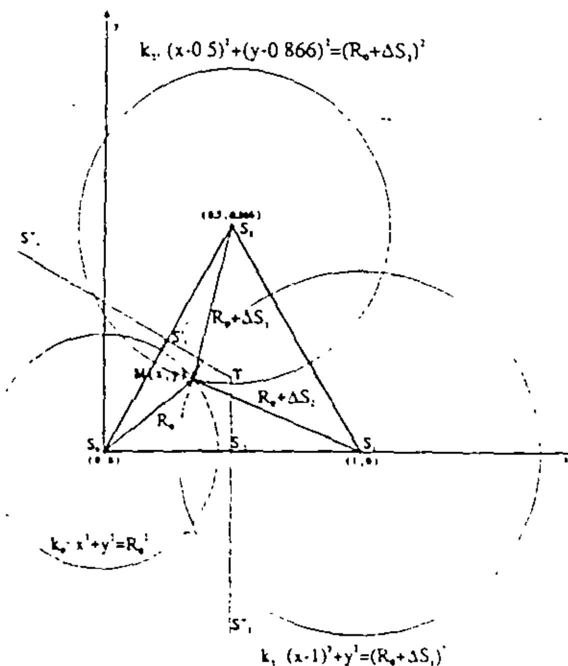


Figure 3. Graphical representation of the circles constructed around the triangle apexes with the radii determined by the location of the active point M(x,y)

The source of vibrations is located in the active point M(x,y), in the active deltoid (S_0, S_2', T, S_1'), and relative to this point, the circles k_0, k_1, and k_2 are constructed around the triangle apexes so that, they include the active point $M(x,y)$. The radii of the circles k_0, k_1, and k_2 are denoted by R_0, R_1, and R_2, respectively. The values of ΔS_1 and ΔS_2 can be calculated using the values of Δt_1 and Δt_2. Since the value of the velocity v, which can be evaluated for a particular type of the ground, is known in advance, the difference distances ΔS_1 and ΔS_2 between the dominant sensor and the remaining ones, are calculated according to the relations:

$$\Delta r_i = \Delta t_i v, \; i = 1,2 \tag{6}$$

In figure 2. the relative time delays between the responses of the sensors S_0 - S_1 and S_0 - S_2 are denoted by Δt_1 and Δt_2, respectively.

In the case in which the active point $M(x,y)$ is outside the active deltoid, by applying the appropriate transformation to the coordinate system (rotation), it is possible to include the active point into the transformed active deltoid. After determination of the active point coordinates in the transformed coordinate system, the coordinates of the active point $M(x,y)$ in the original system can be easily calculated. The active deltoid is chosen with respect to the referent dominant sensor, i.e. the first excited sensor in the sensor unit.

Analytical expressions for the circles k_0, k_1, and k_2, whose centers are placed at the triangle apexes, and the radii are determined by the location of the actual source of seismic vibrations, $M(x,y)$, are given by:

$$K_0: \; x^2 + y^2 = R_0^2 \tag{7}$$

$$K_1: \; (x-1)^2 + y^2 = (R_0 + \Delta S_1)^2 \tag{8}$$

$$K_2: \; \left(x - \frac{1}{2}\right)^2 + \left(y - \frac{\sqrt{3}}{2}\right)^2 = (R_0 + \Delta S_2)^2 \tag{9}$$

The given system of equations has been solved in order to determine the values of the unknown radius R_0, and the coordinates of the actual source of vibrations, $M(x,y)$ The solutions for R_0, x and y in the explicit (closed) form are given by:

$$x_{1,2} = \frac{1}{2}\left(\left(1 - \Delta S_1^2\right) - 2R_{01,2}\Delta S_1\right) \tag{10}$$

$$y_{1,2} = \frac{1}{2\sqrt{3}}\Big[\, 2\left(1-\Delta S_2^2\right) - \left(1 - \Delta S_1^2\right) + 2R_{01,2}\Delta S_1^2 - 4R_{01,2}\Delta S_2 \,\Big] \tag{11}$$

and the values of $R_{01,2}$, A, B, C is given by:

$$R_{01,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{12}$$

$$A = 4\left(4\Delta S_1^2 + 4\Delta S_2^2 - 4\Delta S_1 \Delta S_2 - 3\right) \tag{13}$$

$$B = 4\Big[\left(2\left(1-\Delta S_2^2\right) - \left(1-\Delta S_1^2\right)\right)\Delta S_1 \cdot -2\left(2\left(1-\Delta S_2^2\right) - \left(1-\Delta S_1^2\right)\right)\Delta S_2 - 3\left(1-\Delta S_1^2\right)\Delta S_1 \Big] \tag{14}$$

$$C = 3\left(1-\Delta S_1^2\right)^2 - \left(2\left(1-\Delta S_2^2\right) - \left(1-\Delta S_1^2\right)\right)^2 \tag{15}$$

The system of equations (7,8 and 9) is of the order two, therefore there exist the pairs of the solutions. The solutions with the negative value of R_0 can be discarded.

In general, the described method for determination of the source location can provide the solutions which are outside the deltoid (S_0, S_2', T, S_1').

4. VERIFICATION OF THE RESULTS BY SIMULATION

A software module for simulation of the proposed method is designed in order to test and to provide an efficient field control of a new system. The simulation presumes the choice of the active point $M(x,y)$, i.e. the coordinates of the excitation are known in advance. On the basis of these data, the following lengths are calculated:

$$\overline{MS_0} = \sqrt{x^2 + y^2} \tag{16}$$

$$\overline{MS_1} = \sqrt{\left(1 - x^2\right) + y^2} \tag{17}$$

$$\overline{MS_2} = \sqrt{\left(1/2 - x\right)^2 + \left(\sqrt{3}/2 - x\right)^2} \tag{18}$$

The time intervals required for the seismic wavefronts to propagate from the given excitation point $M(x,y)$ to the sensors S_0, S_1 and S_2, are given by:

$$t_i = \frac{\overline{MS_i}}{v}, \ i = 0,1,2 \qquad (19)$$

Time delays of the seismic wavefronts with respect to the dominant sensor S_0 are given by:

$$\Delta t_r = t_0 - t_r, \ r = 1,2 \qquad (20)$$

Figure 4.a and 4.b show the results of simulations obtained for the cases in which the calculated values of Δt_1 and Δt_2 are the only input parameters to the described algorithm. The obtained trajectory confirms that the previously described solution is general and valid both inside and outside the active sensor unit. By choosing the appropriate coordinate step, various traces of trajectory can be easily obtained.

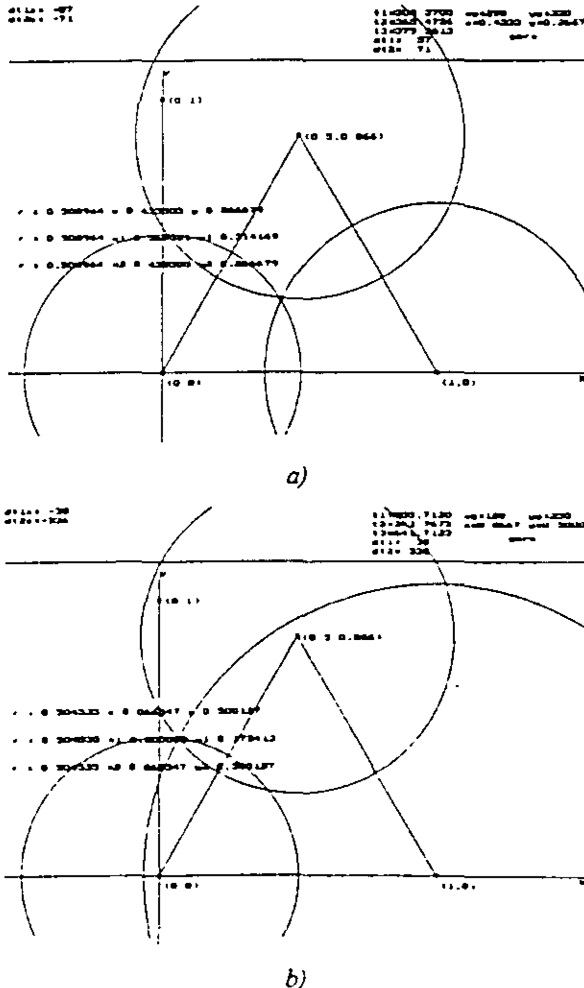


Figure 4 Examples of excitation locations
a) Excitation inside the triangle
b) Excitation outside the triangle

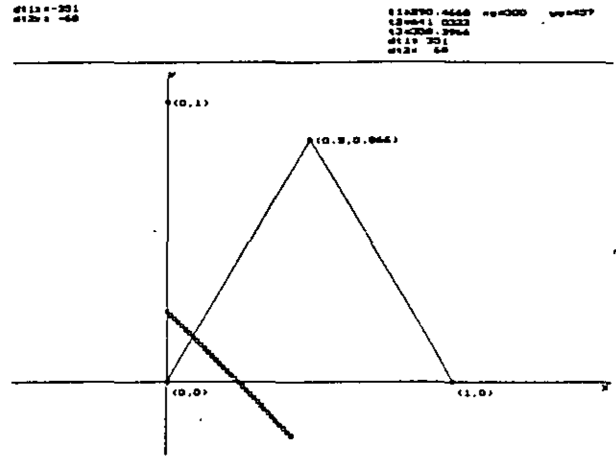


Figure 5 Simulation of the straight-line trajectory

The described software module is designed using the BORLAND C++ 3.1.

5. CONCLUSION

In this paper, an effective explicit technique for localization of the unknown sources of seismic vibrations using the observed signals only, is proposed. This technique is based on a circle method for determination of the source coordinates, and provides unique solution for the actual source coordinates in a closed form.

Because of its simplicity, the proposed technique is appropriate for real-time applications and can be applied to an open area protection. In the proposed method, the distances between the active point and the sensors in the active sensor unit are calculated using the prescribed coordinates of the source of vibration and the previously evaluated propagation velocity.

The actual coordinates of the source of vibrations are calculated using the time delays determined with respect to the referent sensor. The calculated values of coordinates are in close agreement with the prescribed initial values for the program testing. In other words, by choosing the arbitrary point inside the sensor unit, we can verify the accuracy of both the determined location and the propagation velocity of seismic wavefront for the actual type of the ground. In general, the proposed method can be applied in the cases in which the location of the source of vibrations is outside the triangle.

The described protection system, based on the seismic vibrations generated by the moving object, is efficient and reliable for open area protection. Since the sensors are dug in the soil,

concrete, or asphalt, the sensors are invisible for eventual excitations above the ground. Figure 5 demonstrates the results of real data processing with man-made seismic signals.

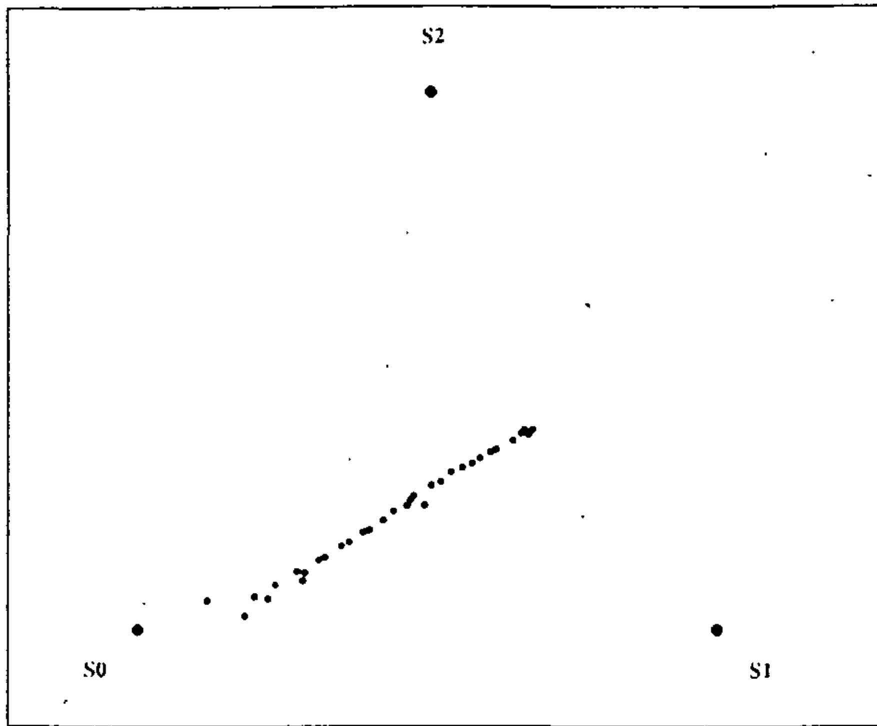


Figure 5. Experimental trajectory on real geological medium. Moving seismic source is a man.

$$\overline{S0S1} = \overline{S0S2} = \overline{S1S21} = 50\,m \quad , F_s=1\ kHz,\ N=1024$$

LITERATURE

[1] I. Gurvich: *Seismic prospecting*, Mir publishers, Moscow, 1972.

[2] А.А. Тресков, М.Б Вертлиб, *Объективное определение эпицентров близких землетрясений*", АКАДЕМИЯ НАУК СССР, СИБИРСКОЕ ОДЕЛЕНИЕ, Институт земной коры, Наука, 1973.

[3] G. Papadopoulos, K. Efstathion , Yiqi Li, A. Delis: Implementation of an Intelligent Instrument of Passive Recognition and Two - Dimensional Location Estimation of Acoustic Targets, IEEE Trans. on Instrumentation and Measurement, Vol. 41, No. 6, December, 1992.

[4] Cindy L. Mason, NASA Ames Research Center, *An Intelligent Assistant for Nuclear Test Ban Treaty Verification*, IEEE Expert, Environmental Applications of AI. December 1995

[5] A. G. Piersol, *Time Delay Estimation Using Phase Data*, IEEE Trans, Accoust. Speech, Signal Processing, vol. ASSP-29, pp. 471-477, 1981.

[6] C.H. Knapp and G.C. Carter, *The generalized correlation method for estimation of time delay*, IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-24, pp. 320-327, 1976.

[7] Denis H. McCabe, Richard L. Moose, *Passive Source Tracking Using Sonar Delay Data*, IEEE Transactions on acoustics, speech, and signal processing, Vol. ASSP-29, No. 3, June 1981, Special issue on Time Delay Estimation.

[8] William Sweet, *Better Networks for Test Ban Monitoring* , IEEE Spectrum, Vol. 33, No 2, February 1996.

6

Kalman Filter Theory and Its Application to Adaptive Transversal Filters

6.1 INTRODUCTION

A limitation of the LMS algorithm is that it does not make full use of all the information available to it at the time of adaptation, with the result that its rate of convergence is relatively slow. One method of overcoming this limitation is to use *Kalman filter theory*, which provides the solution to a class of recursive minimum mean-square estimation problems (Kalman, 1960). The Kalman filter is formulated using the *state-space approach*, in which a dynamical system is described by a set of variables called the *state*. The state contains all the necessary information about the behavior of the system such that, given the present and future values of the input, we may compute the future state and output of the system (Kailath, 1980). The significance of Kalman filter theory will be fully appreciated when we apply it to formulate adaptive filtering algorithms for stationary or nonstationary environments later in the chapter

From Chapter 3, we recall that Wiener filter theory requires the inversion of the correlation matrix of the tap-input vector of a transversal filter for its solution. On the other hand, we find that the application of Kalman filter theory results in

a set of difference equations, the solutions of which can be obtained *recursively*. In particular, we find that each updated estimate can be computed from the previous estimate and the new input data, so only the previous estimate must be stored. In addition to eliminating the need for storing the entire past observed data, the Kalman filtering algorithm is computationally more efficient than computing the estimates directly from the entire past observed data at each step. The Kalman filter is thus ideally suited for implementation on a digital computer.

We begin the study of recursive minimum mean-square estimation by considering the simple case of scalar random variables. For this development, we use the *innovations approach* (Kailath, 1968, 1970), which utilizes the correlation properties of the *innovations process*. The idea of innovations was perhaps first used by Kolmogorov (1939).

6.2 RECURSIVE MINIMUM MEAN-SQUARE ESTIMATION FOR SCALAR RANDOM VARIABLES

Let us assume that, based on a complete set of observed random variables $y(1)$, $y(2), \ldots, y(n-1)$, starting with the first observation at time 1 and extending up to and including time $n-1$, we have found the minimum mean-square estimate $\hat{x}(n-1|\mathcal{Y}_{n-1})$ of a related random variable $x(n-1)$. We are assuming that the observation at (or before) $n = 0$ is zero. The space spanned by the observations $y(1), \ldots, y(n-1)$ is denoted by $\mathcal{Y}_{n-1}$. Suppose we now have an additional observation $y(n)$ at time n, and the requirement is to compute an *updated* estimate $\hat{x}(n|\mathcal{Y}_n)$ of the related random variable $x(n)$, where $\mathcal{Y}_n$ denotes the space spanned by $y(1), \ldots, y(n)$. We may do this computation by storing the *past* observations, $y(1), y(2), \ldots, y(n-1)$, and then redoing the whole problem with the available data $y(1), y(2), \ldots, y(n-1), y(n)$, including the new observation. Computationally, however, it is much more efficient to use a *recursive estimation procedure*. In this procedure we *store* the previous estimate $\hat{x}(n-1|\mathcal{Y}_{n-1})$ and exploit it to compute the updated estimate $\hat{x}(n|\mathcal{Y}_n)$ in the light of the new observation $y(n)$. There are several ways of developing the algorithm to do this recursive estimation. We will use the notion of *innovations* (Kailath, 1968, 1970).

Define the forward prediction error

$$f_{n-1}(n) = y(n) - \hat{y}(n|\mathcal{Y}_{n-1}) \qquad n = 1, 2, \ldots \tag{6.1}$$

where $\hat{y}(n|\mathcal{Y}_{n-1})$ is the *one-step prediction* of the observed random variable $y(n)$ at time n, using *all* past observations available up to and including time $n-1$. The past observations used in this estimation are $y(1), y(2), \ldots, y(n-1)$, so the order of the prediction equals $n-1$. We may view $f_{n-1}(n)$ as the output of a forward prediction-error filter of order $n-1$, and with the filter input fed by the time series $y(1), y(2), \ldots, y(n)$. Note that the *prediction order $n-1$ increases linearly with n*. According to the principle of orthogonality, the prediction error $f_{n-1}(n)$ is orthogonal to all past observations $y(1), y(2), \ldots, y(n-1)$ and may

therefore be regarded as a *measure* of the new information in the random variable $y(n)$ observed at time n; hence, the name "innovation." The fact is that the observation $y(n)$ does not itself convey completely new information, since the predictable part, $\hat{y}(n|\mathcal{Y}_{n-1})$, is already completely determined by the past observations $y(1), y(2), \ldots, y(n-1)$. Rather, the part of the observation $y(n)$ that is really new is contained in the forward prediction error $f_{n-1}(n)$. We may therefore refer to this prediction error as the innovation, and for simplicity of notation write

$$\alpha(n) = f_{n-1}(n), \qquad n = 1, 2, \ldots \tag{6.2}$$

The innovation $\alpha(n)$ has several important properties, as follows:

Property 1. *The innovation $\alpha(n)$, associated with the observed random variable $y(n)$, is orthogonal to the past observations $y(1), y(2), \ldots, y(n-1)$, as shown by*

$$E[\alpha(n)y^*(k)] = 0, \qquad 1 \le k \le n-1 \tag{6.3}$$

This is simply a restatement of the principle of orthogonality.

Property 2. *The innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$ are orthogonal to each other, as shown by*

$$E[\alpha(n)\alpha^*(k)] = 0, \qquad 1 \le k \le n-1 \tag{6.4}$$

This is a restatement of the fact that [see Eq. (4.137)]

$$E[f_{n-1}(n)f_{k-1}^*(k)] = 0, \qquad 1 \le k \le n-1$$

Property 3. *There is a one-to-one correspondence between the observed data $\{y(1), y(2), \ldots, y(n)\}$ and the innovations $\{\alpha(1), \alpha(2), \ldots, \alpha(n)\}$ in that the one sequence may be obtained from the other without any loss of information. We may thus write*

$$\{y(1), y(2), \ldots, y(n)\} \rightleftharpoons \{\alpha(1), \alpha(2), \ldots, \alpha(n)\} \tag{6.5}$$

To prove this property, we use a form of the Gram–Schmidt orthogonalization procedure (described in Chapter 4). The procedure assumes that the observations $y(1), y(2), \ldots, y(n)$ are linearly independent in an algebraic sense. We first put

$$\alpha(1) = y(1) \tag{6.6}$$

where it is assumed that $\hat{y}(1|\mathcal{Y}_0)$ is zero. Next, we put

$$\alpha(2) = y(2) - a_{1,1}y(1) \tag{6.7}$$

The coefficient $a_{1,1}$ is chosen such that the innovations $\alpha(1)$ and $\alpha(2)$ are orthogonal, as shown by

$$E[\alpha(2)\alpha^*(1)] = 0 \tag{6.8}$$

161

This requirement is satisfied by choosing

$$a_{1,1} = -\frac{E[y(2)y^*(1)]}{E[y(1)y^*(1)]} \tag{6.9}$$

Hence, except for a minus sign, $a_{1,1}$ is a partial correlation coefficient in that it equals the cross-correlation between the observations $y(2)$ and $y(1)$, normalized with respect to the mean-square value of $y(1)$.

Next we put

$$\alpha(3) = y(3) + a_{2,1}y(2) + a_{2,2}y(1) \tag{6.10}$$

where the coefficients $a_{2,1}$ and $a_{2,2}$ are chosen such that $\alpha(3)$ is orthogonal to both $\alpha(1)$ and $\alpha(2)$, and so on. Thus, in general, we may express the transformation of the observed data $y(1), y(2), \ldots, y(n)$ into the innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$ by writing

$$\begin{bmatrix} \alpha(1) \\ \alpha(2) \\ \vdots \\ \alpha(n) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ a_{1,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,n-1} & a_{n-1,n-2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(n) \end{bmatrix} \tag{6.11}$$

The nonzero elements of row k of the *lower triangular transformation matrix* on the right side of Eq. (6.11) equal $a_{k-1,k-1}, a_{k-1,k-2}, \ldots, 1$, where $k = 1, 2, \ldots, n$. These elements represent the coefficients of a *backward* prediction-error filter of order $k - 1$. Note that $a_{k,0} = 1$ for all k. Accordingly, given the observed data $y(1), y(2), \ldots, y(n)$, we may compute the innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$. There is no loss of information in the course of this transformation, since we may recover the original observed data $y(1), y(2), \ldots, y(n)$ from the innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$. This we do by premultiplying both sides of Eq. (6.11) by the inverse of the lower triangular transformation matrix. This matrix is nonsingular since its determinant equals 1 for all n. The transformation is therefore reversible.

Using Eq. (6.5), we may thus write.

$\hat{x}(n|\mathcal{Y}_n)$ = minimum mean-square estimate of $x(n)$ given the observed data $y(1), y(2), \ldots, y(n)$

or, equivalently,

$\hat{x}(n|\mathcal{Y}_n)$ = minimum mean-square estimate of $x(n)$ given the innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$

Define the estimate $\hat{x}(n|\mathcal{Y}_n)$ as a linear combinaton of the innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$:

$$\hat{x}(n|\mathcal{Y}_n) = \sum_{k=1}^{n} b_k \alpha(k) \tag{6.12}$$

where the b_k are to be determined. With the innovations $\alpha(1), \alpha(2), \ldots, \alpha(n)$ orthogonal to each other, and the b_k chosen to minimize the mean-squared value of the estimation error $x(n) - \hat{x}(n|\mathcal{Y}_n)$, we find that

$$b_k = \frac{E[x(n)\alpha^*(k)]}{E[\alpha(k)\alpha^*(k)]}, \qquad 1 \le k \le n \tag{6.13}$$

We rewrite Eq. (6.12) in the form

$$\hat{x}(n|\mathcal{Y}_n) = \sum_{k=0}^{n-1} b_k\alpha(k) + b_n\alpha(n) \tag{6.14}$$

where

$$b_n = \frac{E[x(n)\alpha^*(n)]}{E[\alpha(n)\alpha^*(n)]} \tag{6.15}$$

However, by definition, the summation term on the right side of Eq. (6.14) equals the previous estimate $\hat{x}(n-1|\mathcal{Y}_{n-1})$. We may thus express the desired recursive estimation algorithm as

$$\hat{x}(n|\mathcal{Y}_n) = \hat{x}(n-1|\mathcal{Y}_{n-1}) + b_n\alpha(n) \tag{6.16}$$

where b_n is defined by Eq. (6.15). Thus, by adding a *correction term* $b_n\alpha(n)$ to the previous estimate $\hat{x}(n-1|\mathcal{Y}_{n-1})$, with the correction proportional to the innovation $\alpha(n)$, we get the updated estimate $\hat{x}(n|\mathcal{Y}_n)$.

The simple formulas of Eqs. (6.12) and (6.16) are the basis of all recursive estimation schemes. Equipped with these simple and yet basic ideas, we are ready to study the more general Kalman filtering problem.

6.3 STATEMENT OF THE KALMAN FILTERING PROBLEM

Let an M-dimensional parameter vector $\mathbf{x}(n)$ denote the *state* of a discrete-time, linear, dynamical system, and let an N-dimensional vector $\mathbf{y}(n)$ denote the *observed data* of the system, both measured at time n. In general, the vectors $\mathbf{x}(n)$ and $\mathbf{y}(n)$ consist of vector random variables. Thus, the system *model* is described by two equations represented in the form of a signal-flow graph, as in Fig. 6.1. They are as follows:

1. A *process equation*

$$\mathbf{x}(n+1) = \mathbf{\Phi}(n+1, n)\mathbf{x}(n) + \mathbf{v}_1(n) \tag{6.17}$$

where $\mathbf{\Phi}(n+1, n)$ is a known M-by-M *state transition matrix* relating the states of the system at times $n+1$ and n. The M-by-1 vector $\mathbf{v}_1(n)$ represents *process noise*. The vector $\mathbf{v}_1(n)$ is modeled as zero-mean, white-noise proc-

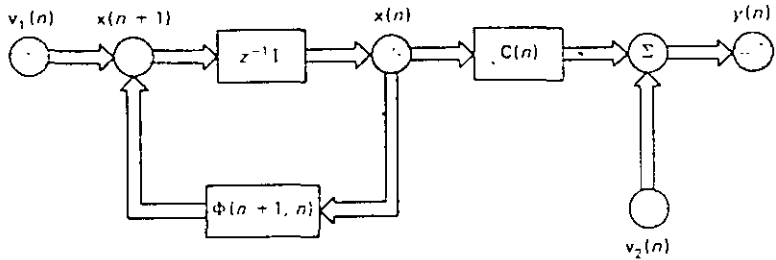


Figure 6.1 Signal-flow graph representation of discrete-time, linear, dynamical system.

esses whose correlation matrix is defined by

$$E[\mathbf{v}_1(n)\mathbf{v}_1^H(k)] = \begin{cases} \mathbf{Q}_1(n), & n = k \\ \mathbf{0}, & n \neq k \end{cases} \tag{6.18}$$

2. A *measurement equation*, describing the observation vector as follows:

$$\mathbf{y}(n) = \mathbf{C}(n)\mathbf{x}(n) + \mathbf{v}_2(n) \tag{6.19}$$

where $\mathbf{C}(n)$ is a known N-by-M *measurement matrix*. The N-by-1 vector $\mathbf{v}_2(n)$ is called *measurement noise*. It is modeled as zero-mean, white-noise processes whose correlation matrix is

$$E[\mathbf{v}_2(n)\mathbf{v}_2^H(k)] = \begin{cases} \mathbf{Q}_2(n), & n = k \\ \mathbf{0}, & n \neq k \end{cases} \tag{6.20}$$

The noise vectors $\mathbf{v}_1(n)$ and $\mathbf{v}_2(n)$ are statistically independent, so we may write

$$E[\mathbf{v}_1(n)\mathbf{v}_2^H(k)] = \mathbf{0}, \qquad \text{for all } n \text{ and } k \tag{6.21}$$

Note that the state transition matrix $\Phi(n+1, n)$ and the measurement matrix $\mathbf{C}(n)$ are both assumed *known*. The problem is to use the observed data, consisting of the vectors $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n)$, to find for each $n \geq 1$ the minimum mean-square estimates of the components of the state $\mathbf{x}(i)$. The problem is called the *filtering* problem if $i = n$, the *prediction* problem if $i > n$, and the *smoothing* problem if $1 \leq i < n$. We will only be concerned with the filtering and prediction problems, which are closely related.

As remarked earlier in the introduction, we will solve the Kalman filtering problem by using the innovations approach (Kailath, 1968, 1970, 1981; Tretter, 1976).

6.4 THE INNOVATIONS PROCESS

Let the vector $\hat{\mathbf{y}}(n|\mathcal{Y}_{n-1})$ denote the minimum mean-square estimate of the observed data $\mathbf{y}(n)$ at time n, given all the past values of the observed data starting at time $n = 1$ and extending up to and including time $n - 1$. These past values are represented by the vectors $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n-1)$, which span the vector space $\mathcal{Y}_{n-1}$. We define the *innovations process* associated with $\mathbf{y}(n)$ as

$$\boldsymbol{\alpha}(n) = \mathbf{y}(n) - \hat{\mathbf{y}}(n|\mathcal{Y}_{n-1}), \qquad n = 1, 2, \ldots \tag{6.22}$$

The M-by-1 vector $\boldsymbol{\alpha}(n)$ represents the new information in the observed data $\mathbf{y}(n)$.

Generalizing the results of Eqs. (6.3), (6.4) and (6.5), we find that the innovations process $\boldsymbol{\alpha}(n)$ has the following properties:

1. The innovations process $\boldsymbol{\alpha}(n)$, associated with the observed data $\mathbf{y}(n)$ at time n, is orthogonal to all past observations $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n-1)$ as shown by

$$E[\boldsymbol{\alpha}(n)\mathbf{y}^H(k)] = \mathbf{0}, \qquad 1 \le k \le n-1 \tag{6.23}$$

2. The innovations process consists of a sequence of vector random variables that are orthogonal to each other, as shown by

$$E[\boldsymbol{\alpha}(n)\boldsymbol{\alpha}^H(k)] = \mathbf{0}, \qquad 1 \le k \le n-1 \tag{6.24}$$

3. There is a one-to-one correspondence between the sequence of vector random variables $\{\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n)\}$ representing the observed data and the sequence of vector random variables $\{\boldsymbol{\alpha}(1), \boldsymbol{\alpha}(2), \ldots, \boldsymbol{\alpha}(n)\}$ representing the innovations process, in that the one sequence may be obtained from the other by means of linear stable operators without loss of information. Thus, we may state that

$$\{\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n)\} \rightleftharpoons \{\boldsymbol{\alpha}(1), \boldsymbol{\alpha}(2), \ldots, \boldsymbol{\alpha}(n)\} \tag{6.25}$$

To form the sequence of vector random variables defining the innovations process, we may use a Gram–Schmidt orthogonalization procedure similar to that described in Section 6.2, except that the procedure is now formulated in terms of vectors and matrices (see Problem 1)

Correlation Matrix of the Innovations Process

To determine the correlation matrix of the innovations process $\boldsymbol{\alpha}(n)$, we first solve the state equation (6.17) recursively to obtain

$$\mathbf{x}(k) = \boldsymbol{\Phi}(k, 0)\mathbf{x}(0) + \sum_{i=1}^{k-1} \boldsymbol{\Phi}(k, i+1)\mathbf{v}_1(i) \tag{6.26}$$

165

where we have made use of the following assumptions and properties:

1. The initial value of the state vector equals $\mathbf{x}(0)$.
2. As previously assumed, the observed data [and therefore the noise vector $\mathbf{v}_1(n)$] are zero for $n \le 0$.
3. The state transition matrix has the properties

$$\Phi(k, k-1)\Phi(k-1, k-2)\ldots\Phi(i+1, i) = \Phi(k, i)$$

and

$$\Phi(k, k) = \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix.

Equation (6.26) shows that $\mathbf{x}(k)$ is a linear combination of $\mathbf{x}(0)$ and $\mathbf{v}_1(1), \mathbf{v}_1(2), \ldots, \mathbf{v}_1(k-1)$.

By hypothesis, the measurement noise vector $\mathbf{v}_2(n)$ is uncorrelated with both the initial state vector $\mathbf{x}(0)$ and the process noise vector $\mathbf{v}_1(n)$. Accordingly, premultiplying both sides of Eq. (6.26) by $\mathbf{v}_2^H(n)$, and taking expectations, we deduce that

$$E[\mathbf{x}(k)\mathbf{v}_2^H(n)] = \mathbf{0}, \qquad k, n \ge 0 \tag{6.27}$$

Correspondingly, we deduce from the measurement equation (6.19) that

$$E[\mathbf{y}(k)\mathbf{v}_2^H(n)] = \mathbf{0}, \qquad 0 \le k \le n-1 \tag{6.28}$$

Moreover, we may write

$$E[\mathbf{y}(k)\mathbf{v}_1^H(n)] = \mathbf{0}, \qquad 0 \le k \le n \tag{6.29}$$

Given the past observations $\mathbf{y}(1), \ldots, \mathbf{y}(n-1)$ that span the space $\mathcal{Y}_{n-1}$, we also find from the measurement equation (6.19) that the minimum mean-square estimate of the present value $\mathbf{y}(n)$ of the observation vector equals

$$\hat{\mathbf{y}}(n|\mathcal{Y}_{n-1}) = \mathbf{C}(n)\,\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}) + \hat{\mathbf{v}}_2(n|\mathcal{Y}_{n-1})$$

However, the estimate $\hat{\mathbf{v}}_2(n|\mathcal{Y}_{n-1})$ of the measurement noise vector is zero since it is orthogonal to the past observations $\mathbf{y}(1), \ldots, \mathbf{y}(n-1)$, see Eq. (6.28). Hence, we may simply write

$$\hat{\mathbf{y}}(n|\mathcal{Y}_{n-1}) = \mathbf{C}(n)\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}) \tag{6.30}$$

Therefore, using Eqs. (6.22) and (6.30), we may express the innovations process in the form

$$\boldsymbol{\alpha}(n) = \mathbf{y}(n) - \mathbf{C}(n)\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}) \tag{6.31}$$

Substituting the measurement equation (6.19) in (6.31), we get

$$\boldsymbol{\alpha}(n) = \mathbf{C}(n)\boldsymbol{\varepsilon}(n, n-1) + \mathbf{v}_2(n) \tag{6.32}$$

where $\boldsymbol{\varepsilon}(n, n-1)$ is the *predicted state-error vector* at time n, using data up to time $n-1$. That is, $\boldsymbol{\varepsilon}(n, n-1)$ is the difference between the state vector $\mathbf{x}(n)$ and the one-step prediction vector $\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})$, as shown by

$$\boldsymbol{\varepsilon}(n, n-1) = \mathbf{x}(n) - \hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}) \tag{6.33}$$

Note that the predicted state-error vector is orthogonal to both the process noise vector $\mathbf{v}_1(n)$ and the measurement noise vector $\mathbf{v}_2(n)$; see Problem 2.

The correlation matrix of the innovations process $\boldsymbol{\alpha}(n)$ is defined by

$$\boldsymbol{\Sigma}(n) = E[\boldsymbol{\alpha}(n)\boldsymbol{\alpha}^H(n)] \tag{6.34}$$

Therefore, substituting Eq. (6.32) in (6.34), expanding the pertinent terms, and then using the fact that the vectors $\boldsymbol{\varepsilon}(n, n-1)$ and $\mathbf{v}_2(n)$ are orthogonal, we obtain the desired result:

$$\boldsymbol{\Sigma}(n) = \mathbf{C}(n)\mathbf{K}(n, n-1)\mathbf{C}^H(n) + \mathbf{Q}_2(n) \tag{6.35}$$

where $\mathbf{Q}_2(n)$ is the correlation matrix of the noise vector $\mathbf{v}_2(n)$. The M-by-M matrix $\mathbf{K}(n, n-1)$ is called the *predicted state-error correlation matrix*, defined by

$$\mathbf{K}(n, n-1) = E[\boldsymbol{\varepsilon}(n, n-1)\boldsymbol{\varepsilon}^H(n, n-1)] \tag{6.36}$$

where $\boldsymbol{\varepsilon}(n, n-1)$ is the predicted state-error vector.

6.5 ESTIMATION OF THE STATE USING THE INNOVATIONS PROCESS

Consider next the problem of deriving the minimum mean-square estimate of the state $\mathbf{x}(i)$ from the innovations process. From the discussion presented in Section 6.2, we deduce that this estimate may be expressed as a linear combination of the sequence of innovations processes $\boldsymbol{\alpha}(1), \boldsymbol{\alpha}(2), \ldots, \boldsymbol{\alpha}(n)$ [see Eq. (6.12)]:

$$\hat{\mathbf{x}}(i|\mathcal{Y}_n) = \sum_{k=1}^{n} \mathbf{B}_i(k)\boldsymbol{\alpha}(k) \tag{6.37}$$

where $\{\mathbf{B}_i(k)\}$ is a set of M-by-N matrices to be determined. According to the principle of orthogonality, the predicted state-error vector is orthogonal to the innovation process, as shown by

$$\begin{aligned} E[\boldsymbol{\varepsilon}(i, n)\boldsymbol{\alpha}^H(m)] &= E[\{\mathbf{x}(i) - \hat{\mathbf{x}}(i|\mathcal{Y}_n)\}\boldsymbol{\alpha}^H(m)] \\ &= \mathbf{0}, \qquad m = 1, 2, \ldots, n \end{aligned} \tag{6.38}$$

Substituting Eq. (6.37) in (6.38) and using the orthogonality property of the innovations process, namely Eq. (6.24), we get

$$\begin{aligned} E[\mathbf{x}(i)\boldsymbol{\alpha}^H(m)] &= \mathbf{B}_i(m)E[\boldsymbol{\alpha}(m)\boldsymbol{\alpha}^H(m)] \\ &= \mathbf{B}_i(m)\boldsymbol{\Sigma}(m) \end{aligned} \tag{6.39}$$

Hence, postmultiplying both sides of Eq. (6.39) by $\Sigma^{-1}(m)$, we find that $B_i(m)$ is given by

$$B_i(m) = E[x(i)\alpha^H(m)]\Sigma^{-1}(m) \tag{6.40}$$

Finally, substituting Eq. (6.40) in (6.37), we get the desired value for the minimum mean-squares estimate $\hat{x}(i|\mathcal{Y}_n)$ as follows:

$$\begin{aligned}\hat{x}(i|\mathcal{Y}_n) &= \sum_{k=1}^{n} E[x(i)\alpha^H(k)]\,\Sigma^{-1}(k)\,\alpha(k)\\ &= \sum_{k=1}^{n-1} E[x(i)\alpha^H(k)]\,\Sigma^{-1}(k)\,\alpha(k)\\ &\quad + E[x(i)\alpha^H(n)]\,\Sigma^{-1}(n)\alpha(n)\end{aligned}$$

For $i = n + 1$, we may therefore write

$$\begin{aligned}\hat{x}(n+1|\mathcal{Y}_n) &= \sum_{k=1}^{n-1} E[x(n+1)\,\alpha^H(k)]\,\Sigma^{-1}(k)\alpha(k)\\ &\quad + E[x(n+1)\,\alpha^H(n)]\,\Sigma^{-1}(n)\alpha(n)\end{aligned} \tag{6.41}$$

However, the state $x(n+1)$ at time $n+1$ is related to the state $x(n)$ at time n by Eq. (6.17). Therefore, using this relation, we may write for $0 \le k \le n$:

$$\begin{aligned}E[x(n+1)\alpha^H(k)] &= E\{[\Phi(n+1, n)x(n) + v_1(n)]\alpha^H(k)\}\\ &= \Phi(n+1, n)E[x(n)\alpha^H(k)]\end{aligned} \tag{6.42}$$

where we have made use of the fact that $\alpha(k)$ depends only on the observed data $y(1), \ldots, y(k)$, and therefore from Eq. (6.29) we see that $v_1(n)$ and $\alpha(k)$ are orthogonal for $0 \le k \le n$. We may thus rewrite the summation term on the right side of Eq. (6.41) as follows:

$$\begin{aligned}&\sum_{k=1}^{n-1} E[x(n+1)\alpha^H(k)]\Sigma^{-1}(k)\alpha(k)\\ &\qquad = \Phi(n+1, n)\sum_{k=1}^{n-1} E[x(n)\alpha^H(k)]\,\Sigma^{-1}(k)\alpha(k)\\ &\qquad = \Phi(n+1, n)\hat{x}(n|\mathcal{Y}_{n-1})\end{aligned} \tag{6.43}$$

Next, define the M-by-N matrix:

$$G(n) = E[x(n+1)\alpha^H(n)]\Sigma^{-1}(n) \tag{6.44}$$

Then, using this definition and the result of Eq. (6.43), we may rewrite Eq. (6.41) as follows

$$\hat{x}(n+1|\mathcal{Y}_n) = \Phi(n+1, n)\,\hat{x}(n|\mathcal{Y}_{n-1}) + G(n)\alpha(n) \tag{6.45}$$

Equation (6.45) shows that we may compute the minimum mean-square estimate $\hat{\mathbf{x}}(n+1|\mathcal{Y}_n)$ of the state of a linear dynamical system by adding to the previous estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})$ that is premultiplied by the state transition matrix $\mathbf{\Phi}(n+1, n)$ a correction term equal to $\mathbf{G}(n)\,\boldsymbol{\alpha}(n)$. The correction term equals the innovations process $\boldsymbol{\alpha}(n)$ premultiplied by the matrix $\mathbf{G}(n)$. Accordingly, and in recognition of the pioneering work by Kalman, the matrix $\mathbf{G}(n)$ is called the *Kalman gain*.

There now remains only the problem of expressing the Kalman gain $\mathbf{G}(n)$ in a form convenient for computation. To do this, we first use Eqs. (6.32) and (6.42) to express the expectation of the product of $\mathbf{x}(n+1)$ and $\boldsymbol{\alpha}^H(n)$ as follows:

$$\begin{aligned} E[\mathbf{x}(n+1)\boldsymbol{\alpha}^H(n)] &= \mathbf{\Phi}(n+1, n)E[\mathbf{x}(n)\boldsymbol{\alpha}^H(n)] \\ &= \mathbf{\Phi}(n+1, n)E[\mathbf{x}(n)\left(\mathbf{C}(n)\boldsymbol{\varepsilon}(n, n-1) + \mathbf{v}_2(n)\right)^H] \\ &= \mathbf{\Phi}(n+1, n)E[\mathbf{x}(n)\boldsymbol{\varepsilon}^H(n, n-1)]\mathbf{C}^H(n) \end{aligned} \tag{6.46}$$

where we have used the fact that the state $\mathbf{x}(n)$ and noise vector $\mathbf{v}_2(n)$ are uncorrelated; see Eq. (6.27). We further note that the predicted state-error vector $\boldsymbol{\varepsilon}(n, n-1)$ is orthogonal to the estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})$. Therefore, the expectation of the product of $\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})$ and $\boldsymbol{\varepsilon}^H(n, n-1)$ is zero, and so we may rewrite Eq. (6.46) by replacing the multiplying factor $\mathbf{x}(n)$ by the predicted state-error vector $\boldsymbol{\varepsilon}(n, n-1)$ as follows:

$$E[\mathbf{x}(n+1)\boldsymbol{\alpha}^H(n)] = \mathbf{\Phi}(n+1, n)E[\boldsymbol{\varepsilon}(n, n-1)\boldsymbol{\varepsilon}^H(n, n-1)]\mathbf{C}^H(n) \tag{6.47}$$

From Eq. (6.36), we see that the expectation on the right side of Eq. (6.47) equals the predicted state-error correlation matrix. Hence, we may rewrite Eq. (6.47) as follows:

$$E[\mathbf{x}(n+1)\boldsymbol{\alpha}^H(n)] = \mathbf{\Phi}(n+1, n)\,\mathbf{K}(n, n-1)\,\mathbf{C}^H(n) \tag{6.48}$$

We may now redefine the Kalman gain. In particular, substituting Eq. (6.48) in (6.44), we get

$$\mathbf{G}(n) = \mathbf{\Phi}(n+1, n)\mathbf{K}(n, n-1)\mathbf{C}^H(n)\mathbf{\Sigma}^{-1}(n) \tag{6.49}$$

where the correlation matrix $\mathbf{\Sigma}(n)$ itself is defined in Eq. (6.35).

As it stands, the formula of Eq. (6.49) is not particularly useful for computing the Kalman gain $\mathbf{G}(n)$, since it requires that the predicted state-error correlation matrix $\mathbf{K}(n, n-1)$ be known. To overcome this difficulty, we derive a formula for the recursive computation of $\mathbf{K}(n, n-1)$.

The predicted state-error vector $\boldsymbol{\varepsilon}(n+1, n)$ equals the difference between the state $\mathbf{x}(n+1)$ and the one-step prediction $\hat{\mathbf{x}}(n+1|\mathcal{Y}_n)$ [see Eq. (6.33)]:

$$\boldsymbol{\varepsilon}(n+1, n) = \mathbf{x}(n+1) - \hat{\mathbf{x}}(n+1|\mathcal{Y}_n) \tag{6.50}$$

Substituting Eqs. (6.17) and (6.45) in (6.50), and using Eq. (6.31) for the inno-

vations process $\boldsymbol{\alpha}(n)$, we get

$$\begin{aligned}\boldsymbol{\varepsilon}(n+1,n) = \boldsymbol{\Phi}(n+1,n)[\mathbf{x}(n) - \hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})] \\ - \mathbf{G}(n)\,[\mathbf{y}(n) - \mathbf{C}(n)\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})] + \mathbf{v}_1(n)\end{aligned} \tag{6.51}$$

Next, using the measurement equation (6.19) to eliminate $\mathbf{y}(n)$ in Eq. (6.51), we get the following difference equation for recursive computation of the predicted state-error vector:

$$\begin{aligned}\boldsymbol{\varepsilon}(n+1,n) = [\boldsymbol{\Phi}(n+1,n) - \mathbf{G}(n)\,\mathbf{C}(n)]\,\boldsymbol{\varepsilon}(n,n-1) \\ + \mathbf{v}_1(n) - \mathbf{G}(n)\,\mathbf{v}_2(n)\end{aligned} \tag{6.52}$$

The correlation matrix of the predicted state-error vector $\boldsymbol{\varepsilon}(n+1,n)$ equals [see Eq. (6.36)]

$$\mathbf{K}(n+1,n) = E[\boldsymbol{\varepsilon}(n+1,n)\,\boldsymbol{\varepsilon}^H(n+1,n)] \tag{6.53}$$

Substituting Eq. (6.52) in (6.53), and recognizing that the error vector $\boldsymbol{\varepsilon}(n,n-1)$ and the noise vectors $\mathbf{v}_1(n)$ and $\mathbf{v}_2(n)$ are mutually uncorrelated, we may express the predicted state-error correlation matrix as follows:

$$\begin{aligned}\mathbf{K}(n+1,n) = [\boldsymbol{\Phi}(n+1,n) - \mathbf{G}(n)\mathbf{C}(n)]\,\mathbf{K}(n,n-1) \\ \cdot [\boldsymbol{\Phi}(n+1,n) - \mathbf{G}(n)\mathbf{C}(n)]^H \\ + \mathbf{Q}_1(n) + \mathbf{G}(n)\,\mathbf{Q}_2(n)\,\mathbf{G}^H(n)\end{aligned} \tag{6.54}$$

where $\mathbf{Q}_1(n)$ and $\mathbf{Q}_2(n)$ are the correlation matrices of $\mathbf{v}_1(n)$ and $\mathbf{v}_2(n)$, respectively. By expanding the right side of Eq. (6.54), and then using Eqs. (6.49) and (6.35) for the Kalman gain, we get the *Riccati difference equation* for the recursive computation of the predicted state-error correlation matrix:

$$\mathbf{K}(n+1,n) = \boldsymbol{\Phi}(n+1,n)\,\mathbf{K}(n)\boldsymbol{\Phi}^H(n+1,n) + \mathbf{Q}_1(n) \tag{6.55}$$

The M-by-M matrix $\mathbf{K}(n)$ is described by the recursion

$$\mathbf{K}(n) = \mathbf{K}(n,n-1) - \boldsymbol{\Phi}(n,n+1)\,\mathbf{G}(n)\,\mathbf{C}(n)\,\mathbf{K}(n,n-1) \tag{6.56}$$

Here we have used the fact that

$$\boldsymbol{\Phi}(n+1,n)\,\boldsymbol{\Phi}(n,n+1) = \mathbf{I} \tag{6.57}$$

where $\mathbf{I}$ is the identity matrix. This property follows from the definition of the transition matrix. The mathematical significance of the matrix $\mathbf{K}(n)$ in Eq. (6.56) will be explained in Section 6.6.

Equations (6.49), (6.31), (6.45), (6.56), and (6.55), in that order, define Kalman's one-step prediction algorithm.

Comments

The process applied to the input of the Kalman filter consists of the observed data $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n)$ that span the space $\mathcal{Y}_n$. The resulting filter output equals the predicted state vector $\hat{\mathbf{x}}(n+1|\mathcal{Y}_n)$. Given that the matrices $\mathbf{\Phi}(n+1, n)$, $\mathbf{C}(n)$, $\mathbf{Q}_1(n)$, and $\mathbf{Q}_2(n)$ are all known quantities, we find from Eqs. (6.44), (6.55) and (6.56) that the predicted state-error correlation matrix $\mathbf{K}(n+1, n)$ is actually independent of the input $\mathbf{y}(n)$: see Problem 2. Therefore, no one set of measurements helps more than any other to eliminate some uncertainty about the state vector $\mathbf{x}(n)$ (Anderson and Moore, 1979). The Kalman gain $\mathbf{G}(n)$ is also independent of the input $\mathbf{y}(n)$. Consequently, the predicted state-error correlation matrix $\mathbf{K}(n+1, n)$ and the Kalman gain $\mathbf{G}(n)$ may be computed before the Kalman filter is actually put into operation.

As already mentioned, the Kalman filter theory assumes knowledge of the matrices $\mathbf{\Phi}(n+1, n)$, $\mathbf{C}(n)$, $\mathbf{Q}_1(n)$ and $\mathbf{Q}_2(n)$. However, the theory may be *generalized* to include a situation where one or more of these matrices may assume values that depend on the input $\mathbf{y}(n)$. In such a situation, we find that although $\hat{\mathbf{x}}(n+1|\mathcal{Y}_n)$ and $\mathbf{K}(n+1, n)$ are still given by Eqs (6.45) and (6.55), respectively, the Kalman gain $\mathbf{G}(n)$ and the predicted state-error correlation matrix $\mathbf{K}(n+1, n)$ are *not* precomputable (Anderson and Moore, 1979). Rather, they both now depend on the input $\mathbf{y}(n)$. This means that $\mathbf{K}(n+1, n)$ is a *conditional* error correlation matrix, conditional on the input $\mathbf{y}(n)$.

Initial Conditions

To operate the one-step prediction algorithm described, we need to know the pertinent set of *initial conditions*. We now address this issue.

We first note that the state $\mathbf{x}(n)$ and the noise vector $\mathbf{v}_1(n)$ are independent of each other. Hence, from the state equation (6.17) we find that the minimum mean-square estimate of the state $\mathbf{x}(n+1)$ at time $n+1$, given the observed data up to and including time n [i.e., given $\mathbf{y}(1), \ldots, \mathbf{y}(n)$], equals

$$\hat{\mathbf{x}}(n+1|\mathcal{Y}_n) = \mathbf{\Phi}(n+1, n)\,\hat{\mathbf{x}}(n|\mathcal{Y}_n) + \hat{\mathbf{v}}_1(n|\mathcal{Y}_n) \tag{6.58}$$

Since the noise vector $\mathbf{v}_1(n)$ is independent of the observed data $\mathbf{y}(1), \ldots, \mathbf{y}(n)$, it follows that the corresponding minimum mean-square estimate $\hat{\mathbf{v}}_1(n|\mathcal{Y}_n)$ is zero. Accordingly, Eq. (6.58) simplifies as

$$\hat{\mathbf{x}}(n+1|\mathcal{Y}_n) = \mathbf{\Phi}(n+1, n)\,\hat{\mathbf{x}}(n|\mathcal{Y}_n) \tag{6.59}$$

Based on this relation, we are ready to define the required initial conditions. Putting $n = 0$ in Eq. (6.59), we get

$$\hat{\mathbf{x}}(1|\mathcal{Y}_0) = \mathbf{\Phi}(1, 0)\,\hat{\mathbf{x}}(0|\mathcal{Y}_0)$$

However, $\hat{x}(0|\mathcal{Y}_0)$ is the estimate of $x(0)$ given no observed data, since the observed data are assumed to start at time $n = 1$. Hence,

$$\hat{x}(1|\mathcal{Y}_0) = \mathbf{0} \tag{6.60}$$

Correspondingly, putting $n = 0$ in Eq. (6.53), we find that the initial value of the predicted state-error correlation matrix equals

$$\begin{aligned} \mathbf{K}(1, 0) &= E[\boldsymbol{\varepsilon}(1, 0)\, \boldsymbol{\varepsilon}^H(1, 0)] \\ &= E[(x(1) - \hat{x}(1|\mathcal{Y}_0))\, (x(1) - \hat{x}(1|\mathcal{Y}_0))^H] \\ &= E[x(1)\, x^H(1)] \end{aligned} \tag{6.61}$$

Equations (6.60) and (6.61) define the required initial conditions.

6.6 FILTERING

The next signal-processing operation we wish to consider is that of filtering. In particular, we wish to compute the filtered estimate $\hat{x}(n|\mathcal{Y}_n)$ by using the one-step prediction algorithm described previously.

To find this estimate, we premultiply both sides of Eq. (6.59) by the inverse of the transition matrix $\Phi(n + 1, n)$, and thus write

$$\hat{x}(n|\mathcal{Y}_n) = \Phi^{-1}(n + 1, n)\, \hat{x}(n + 1|\mathcal{Y}_n) \tag{6.62}$$

Using the property of the state transition matrix given in Eq. (6.57), we have

$$\Phi^{-1}(n + 1, n) = \Phi(n, n + 1) \tag{6.63}$$

We may therefore rewrite Eq. (6.62) in the form

$$\hat{x}(n|\mathcal{Y}_n) = \Phi(n, n + 1)\, \hat{x}(n + 1|\mathcal{Y}_n) \tag{6.64}$$

This shows that knowing the solution to the one-step prediction problem, that is, the minimum mean-square estimate $\hat{x}(n + 1|\mathcal{Y}_n)$, we may determine the corresponding filtered estimate $\hat{x}(n|\mathcal{Y}_n)$ simply by multiplying $\hat{x}(n + 1|\mathcal{Y}_n)$ by the state transition matrix $\Phi(n, n + 1)$.

Thus, we may reformulate Eq. (6.31) to express the innovations process $\alpha(n)$ in terms of the filtered estimate of the state vector at time $n - 1$ as follows:

$$\alpha(n) = y(n) - C(n)\, \Phi(n, n - 1)\, \hat{x}(n - 1|\mathcal{Y}_{n-1}) \tag{6.65}$$

Similarly, we may reformulate Eq. (6.45) to express the recursion for time-updating the filtered estimate of the state vector as follows:

$$\hat{x}(n|\mathcal{Y}_n) = \Phi(n, n - 1)\, \hat{x}(n - 1|\mathcal{Y}_{n-1}) + \Phi(n, n + 1)\, G(n)\, \alpha(n) \tag{6.66}$$

Equations (6.49), (6.65), (6.66), (6.56), and (6.55) collectively and in that order represent another way of describing the Kalman filter, based on the filtered estimate of the state vector.

Filtered State-error Correlation Matrix

Earlier we introduced the M-by-M matrix $\mathbf{K}(n)$ in the formulation of the Riccati difference equation (6.55). We conclude our present discussion of the Kalman filter theory by showing that this matrix equals the correlation matrix of the error inherent in the filtered estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$.

Define the *filtered state-error vector* $\boldsymbol{\varepsilon}(n)$ as the difference between the state $\mathbf{x}(n)$ and the filtered estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$, as shown by

$$\boldsymbol{\varepsilon}(n) = \mathbf{x}(n) - \hat{\mathbf{x}}(n|\mathcal{Y}_n) \tag{6.67}$$

Substituting Eqs. (6.45) and (6.64) in (6.67), and recognizing that the product of $\boldsymbol{\Phi}(n, n+1)$ and $\boldsymbol{\Phi}(n+1, n)$ equals the identity matrix, we get

$$\begin{aligned}\boldsymbol{\varepsilon}(n) &= \mathbf{x}(n) - \hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}) - \boldsymbol{\Phi}(n, n+1)\,\mathbf{G}(n)\,\boldsymbol{\alpha}(n) \\ &= \boldsymbol{\varepsilon}(n, n-1) - \boldsymbol{\Phi}(n, n+1)\,\mathbf{G}(n)\,\boldsymbol{\alpha}(n)\end{aligned} \tag{6.68}$$

where $\boldsymbol{\varepsilon}(n, n-1)$ is the predicted state-error vector at time n, using data up to time $n-1$, and $\boldsymbol{\alpha}(n)$ is the innovations process.

By definition, the correlation matrix of the filtered state-error vector $\boldsymbol{\varepsilon}(n)$ equals the expectation $E[\boldsymbol{\varepsilon}(n)\boldsymbol{\varepsilon}^H(n)]$. Hence, using Eq. (6.68), we may express this expectation as follows:

$$\begin{aligned}E[\boldsymbol{\varepsilon}(n)\boldsymbol{\varepsilon}^H(n)] &= E[\boldsymbol{\varepsilon}(n, n-1)\,\boldsymbol{\varepsilon}^H(n, n-1)] \\ &\quad + \boldsymbol{\Phi}(n, n+1)\,\mathbf{G}(n)\,E[\boldsymbol{\alpha}(n)\boldsymbol{\alpha}^H(n)]\,\mathbf{G}^H(n)\,\boldsymbol{\Phi}^H(n, n+1) \\ &\quad - 2E[\boldsymbol{\varepsilon}(n, n-1)\boldsymbol{\alpha}^H(n)]\,\mathbf{G}^H(n)\,\boldsymbol{\Phi}^H(n, n+1)\end{aligned} \tag{6.69}$$

Examining the right side of Eq. (6.69), we find that the three expectations contained in it may be interpreted individually as follows:

1. The first expectation equals the predicted state-error correlation matrix:

$$\mathbf{K}(n, n-1) = E[\boldsymbol{\varepsilon}(n, n-1)\,\boldsymbol{\varepsilon}^H(n, n-1)]$$

2. The expectation in the second term equals the correlation matrix of the innovations process $\boldsymbol{\alpha}(n)$:

$$\boldsymbol{\Sigma}(n) = E[\boldsymbol{\alpha}(n)\boldsymbol{\alpha}^H(n)]$$

3. The expectation in the third term may be expressed as follows:

$$\begin{aligned}E[\boldsymbol{\varepsilon}(n, n-1)\,\boldsymbol{\alpha}^H(n)] &= E[(\mathbf{x}(n) - \hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}))\boldsymbol{\alpha}^H(n)] \\ &= E[\mathbf{x}(n)\,\boldsymbol{\alpha}^H(n)]\end{aligned}$$

where, in the last line, we have used the fact that the estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})$ is orthogonal to the innovations process $\boldsymbol{\alpha}(n)$, acting as input. Next, from Eq. (6.42) we see, by putting $k = n$ and premultiplying both sides by the

inverse matrix $\Phi^{-1}(n+1, n) = \Phi(n, n+1)$, that

$$\begin{aligned} E[\mathbf{x}(n)\,\alpha^H(n)] &= \Phi(n, n+1)\,E[\mathbf{x}(n+1)\alpha^H(n)] \\ &= \Phi(n, n+1)\,\mathbf{G}(n)\,\Sigma(n) \end{aligned}$$

where, in the last line, we have made use of Eq. (6.44). Hence,

$$E[\varepsilon(n, n-1)\,\alpha^H(n)] = \Phi(n, n+1)\,\mathbf{G}(n)\Sigma(n)$$

We may now use these results in Eq. (6.69), and so obtain

$$\begin{aligned} E[\varepsilon(n)\varepsilon^H(n)] = {} & \mathbf{K}(n, n-1) \\ & - \Phi(n, n+1)\,\mathbf{G}(n)\,\Sigma(n)\mathbf{G}^H(n)\,\Phi^H(n, n+1) \end{aligned} \tag{6.70}$$

We may further simplify this result by noting that [see Eq. (6.49)]

$$\mathbf{G}(n)\,\Sigma(n) = \Phi(n+1, n)\,\mathbf{K}(n, n-1)\,\mathbf{C}^H(n) \tag{6.71}$$

Accordingly, using Eqs. (6.70) and (6.71), and recognizing that the product of $\Phi(n, n+1)$ and $\Phi(n+1, n)$ equals the identity matrix, we get the desired result for the filtered state-error correlation matrix:

$$\begin{aligned} E[\varepsilon(n)\varepsilon^H(n)] = {} & \mathbf{K}(n, n-1) \\ & - \mathbf{K}(n, n-1)\mathbf{C}^H(n)\,\mathbf{G}^H(n)\,\Phi^H(n, n+1) \end{aligned} \tag{6.72}$$

Equivalently, using the Hermitian property of $E[\varepsilon(n)\varepsilon^H(n)]$ and that of $\mathbf{K}(n, n-1)$, we may write

$$E[\varepsilon(n)\varepsilon^H(n)] = \mathbf{K}(n, n-1) - \Phi(n, n+1)\,\mathbf{G}(n)\,\mathbf{C}(n)\,\mathbf{K}(n, n-1) \tag{6.73}$$

Comparing Eq. (6.73) with (6.56), we see that

$$E[\varepsilon(n)\,\varepsilon^H(n)] = \mathbf{K}(n)$$

This shows that the matrix $\mathbf{K}(n)$ used in the Riccati difference equation is in fact the filtered state-error correlation matrix.

Initial Conditions

From Eqs. (6.60) and (6.64), we deduce that the initial value of the filtered estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$ equals

$$\begin{aligned} \hat{\mathbf{x}}(0|\mathcal{Y}_0) &= \Phi(0, 1)\,\hat{\mathbf{x}}(1|\mathcal{Y}_0) \\ &= \mathbf{0} \end{aligned}$$

As before, for the initial condition of the predicted state-error correlation matrix $\alpha(n-1, n)$, we use the value defined by Eq. (6.61)

6.7 SUMMARY AND DISCUSSION

The Kalman filter is a *linear, discrete-time, finite-dimensional system*, the implementation of which is well suited for a digital computer. The input of the filter is the vector process $\{\mathbf{y}(n)\}$, represented by the vector space $\mathcal{Y}_n$, and the output is the filtered estimate $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$ of the state vector. In Table 6.1, we present a summary of the Kalman filter (including initial conditions) based on the one-step prediction algorithm.

A key property of the Kalman filter is that it leads to minimization of the trace of the filtered state-error correlation matrix $\mathbf{K}(n)$. This means that the Kalman filter is the *linear minimum variance estimator* of the state vector $\mathbf{x}(n)$ (Goodwin and Sin, 1984; Anderson and Moore, 1979)

The input–output relation of the Kalman filter is depicted in Fig. 6.2(a), representing a signal-flow graph interpretation of Eqs. (6.65) and (6.66) and the fact that $\hat{\mathbf{x}}(n-1|\mathcal{Y}_{n-1})$ is the delayed version of $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$. In this model, $\mathbf{y}(n)$ is the input and $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$ is the output

We may rearrange Eq. (6.65) in the form

$$\mathbf{y}(n) = \boldsymbol{\alpha}(n) + \mathbf{C}(n)\,\boldsymbol{\Phi}(n, n-1)\,\hat{\mathbf{x}}(n-1|\mathcal{Y}_{n-1}) \tag{6.74}$$

Accordingly, we may represent Eqs. (6.66) and (6.74) by the signal-flow graph shown in Fig. 6.2(b). Here again we have included a branch in the graph to represent the fact that $\hat{\mathbf{x}}(n-1|\mathcal{Y}_{n-1})$ is the delayed version of $\hat{\mathbf{x}}(n|\mathcal{Y}_n)$. We may view Fig. 6.2(b) as a model for generating the process $\{\mathbf{y}(n)\}$ by driving it with the innovations process $\{\boldsymbol{\alpha}(n)\}$. The model of Fig. 6.2(b) is known as the *inverse*

TABLE 6.1 SUMMARY OF THE KALMAN FILTER BASED ON ONE-STEP PREDICTION

Input Vector Process

Observations = $\{\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(n)\}$

Known Parameters

State transition matrix = $\boldsymbol{\Phi}(n+1, n)$

Measurement matrix = $\mathbf{C}(n)$

Correlation matrix of process noise vector = $\mathbf{Q}_1(n)$

Correlation matrix of measurement noise vector = $\mathbf{Q}_2(n)$

Computation: $n = 1, 2, 3, \ldots$

$\mathbf{G}(n) = \boldsymbol{\Phi}(n+1, n)\mathbf{K}(n, n-1)\mathbf{C}^H(n)[\mathbf{C}(n)\mathbf{K}(n, n-1)\mathbf{C}^H(n) + \mathbf{Q}_2(n)]^{-1}$

$\boldsymbol{\alpha}(n) = \mathbf{y}(n) - \mathbf{C}(n)\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1})$

$\hat{\mathbf{x}}(n+1|\mathcal{Y}_n) = \boldsymbol{\Phi}(n+1, n)\hat{\mathbf{x}}(n|\mathcal{Y}_{n-1}) + \mathbf{G}(n)\boldsymbol{\alpha}(n)$

$\hat{\mathbf{x}}(n|\mathcal{Y}_n) = \boldsymbol{\Phi}(n, n+1)\hat{\mathbf{x}}(n+1|\mathcal{Y}_n)$

$\mathbf{K}(n) = \mathbf{K}(n, n-1) - \boldsymbol{\Phi}(n, n+1)\mathbf{G}(n)\mathbf{C}(n)\mathbf{K}(n, n-1)$

$\mathbf{K}(n+1, n) = \boldsymbol{\Phi}(n+1, n)\mathbf{K}(n)\boldsymbol{\Phi}^H(n+1, n) + \mathbf{Q}_1(n)$

Initial Conditions

$\hat{\mathbf{x}}(1|\mathcal{Y}_0) = \mathbf{0}$

$\mathbf{K}(1, 0) = E[\mathbf{x}(1)\mathbf{x}^H(1)]$

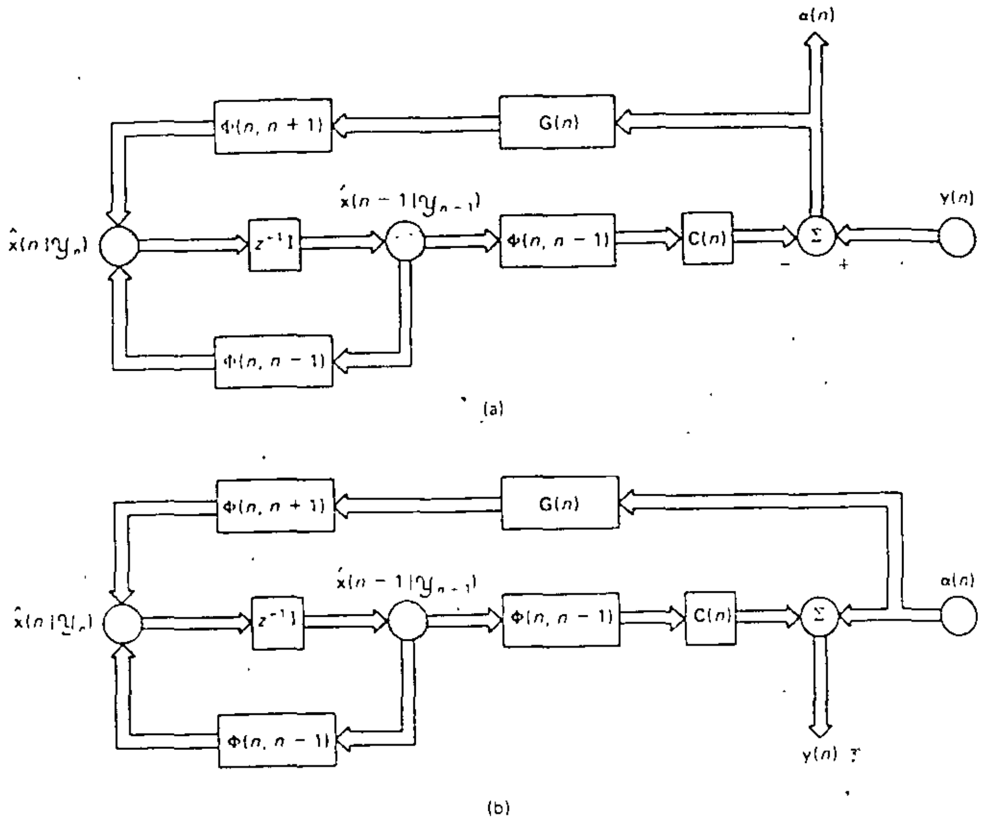


Figure 6.2 Signal-flow graph representations of (a) the Kalman filter, and (b) the inverse model

model In this model, the innovations process $\{\alpha(n)\}$ acts as the input, and the process $\{y(n)\}$ is the output of interest.

We may develop a third model by viewing the observations process $\{y(n)\}$ as the input and the innovations process $\{\alpha(n)\}$ as the output. The signal-flow graph representation of this model is the same as that shown in Fig. 6.2(a), except that now $\alpha(n)$ represents the desired output at time n. The resulting model is known as the *whitening filter*.

We conclude therefore that the Kalman filter, the inverse model, and the whitening filter merely represent three different, and yet equivalent, ways of viewing the optimum linear filter

The Kalman filter has been successfully applied to solve many real-world problems as can be seen in the literature on control systems. Invariably, assumptions are made so as to manipulate the problem of interest into a form amenable to the application of the Kalman filter theory, the aim being to produce a near-optimum and yet workable solution.

Our interest in the Kalman filter theory in this book is that it provides a general framework for the development of various adaptive filtering algorithms with a fast rate of convergence. The application of Kalman filter theory to adaptive filtering was apparently first considered by Lawrence and Kaufman (1971). This was followed by Godard (1974), who used a different approach. In particular, Godard formulated the adaptive filtering problem (using a transversal structure) as the estimation of a state vector in Gaussian noise, a classical Kalman filtering problem. The important feature of the adaptive filtering algorithm derived by Godard is that, when all the random variables pertaining to the problem have joint Gaussian distributions, the algorithm yields the fastest possible rate of convergence. Although these assumptions are not always satisfied, nevertheless, in the majority of environments encountered in practice, we find that the algorithm derived by Godard offers a much faster rate of convergence than is attainable by the simple LMS algorithm. The price that has to be paid for this improvement, however, is increased algorithm complexity.

In the next two sections we follow Godard's approach to derive two adaptive transversal filtering algorithms as special cases of Kalman filter theory. The first algorithm, presented in Section 6.8, assumes a stationary environment. The second algorithm, presented in Section 6.9, assumes a nonstationary environment.

6.8 THE KALMAN ALGORITHM APPLIED TO ADAPTIVE TRANSVERSAL FILTERS WITH STATIONARY INPUTS

Consider a linear transversal filter operating in a stationary environment. Suppose that the M-by-1 tap-weight vector of the filter at time n is set equal to the optimum Wiener value $\mathbf{w}_o(n)$, determined in accordance with the normal equation (see Chapter 2). In a stationary environment, the error-performance surface of the transversal filter has fixed shape and fixed orientation, with the result that the optimum tap-weight vector $\mathbf{w}_o(n)$ has a constant value for all time. Under this condition, we may write

$$\mathbf{w}_o^*(n + 1) = \mathbf{w}_o^*(n) \tag{6.75}$$

Let $\mathbf{u}(n)$ denote the M-by-1 *tap-input vector* applied to this filter at time n. The resulting response of the filter equals the inner product $\mathbf{u}^T(n)\,\mathbf{w}_o^*(n)$. Let $e_o(n)$ denote the optimum value of the *estimation error*, measured with respect to the *desired response* $d(n)$. We may thus model the desired response as

$$d(n) = \mathbf{u}^T(n)\,\mathbf{w}_o^*(n) + e_o(n) \tag{6.76}$$

Equations (6.75) and (6.76) describe the optimum Wiener condition of the transversal filter when operating in a stationary environment. The tap-weight vector on the right sides of these two equations appears in exactly the same form; hence, the use of complex conjugation in (6.75).

In the context of Kalman filter theory, we may view Eq. (6.75) as the process

Avoidance System for Moving Obstacles

James Gil de Lamadrid

Computer Science Department
136 Lind Hall
University of Minnesota
207 Church St. S.E.
Minneapolis, Minnesota 55455

Abstract.

This paper presents a system which moves a robot through Cartesian space in the presence of objects which are also moving in predictable trajectories. The method presented calculates a piecewise linear path with piecewise constant velocities. This is done by assuming a straight-line path from the start to the goal and recursively planning smaller subpaths if collisions are detected.

Two methods for generating subgoals are presented and compared. The first is a blind method employing no information on the location of the objects at collision time. The second is a method using Khatib's artificial potential functions for locating the subgoal.

The algorithm currently is implemented in two dimensions, and represents an arbitrary number of objects by their bounding circles. The robot is represented by a point. The algorithm guarantees that all movement of the robot will take place in a specified workspace. In addition the robot is required to stay on a specified time schedule within a certain tolerance. Obstacle trajectories are constrained to be represented as [illegible] parametric equations in time.

The paper points out some obvious extensions to this work. These include extensions to three dimensions, [illegible] motion planning with complex shapes, and use of the system with objects following unknown trajectories.

Introduction.

This paper describes the method used by the Avoid system to do obstacle avoidance. Avoid is a system which does obstacle avoidance for obstacles moving through Cartesian space. The current version operates in two dimensions, and this is the version described.

The ability of a robot to move through moving obstacles is fundamental to an operator independent system. This ability is important in systems which do navigation in a changing environment, such as a robot automobile driving down a highway. It is also important in robot assembly tasks, where repairs must be made to machines which have moving parts, and cannot be completely halted during repairs.

Related work.

Several authors have described methods for planning paths through static objects. Lozano-Perez [1,2] describes methods for path planning in configuration space using projection to reduce dimensionality. Paul [3] and Taylor [4] discuss Cartesian and straight line trajectories which are used by Avoid in terms of [illegible] issues. Brady [5] points out that straight line trajectories minimize inertial effects and produce shorter paths.

[illegible] approach to obstacle avoidance [illegible] to describe free space. The path used is described by the axes of [illegible] approaches [illegible] representation [illegible] metric shapes.

The method used by Avoid [illegible] was suggested by Salter [7] who described a system which plans [illegible] events.

Cameron [8] discussed three collision detection methods dealing with moving objects. The first is a reduction of dimensionality, such as Lozano-Perez [1,2] has done. The second uses snapshots of the world taken at small time intervals, and the third method is intersection calculation. This latter is the method used [illegible] and the Avoid [illegible].

System overview.

Avoid represents the robot as a three dimensional point, with x and y coordinates and a time coordinate. Objects in the workspace are represented as circles, described by the x and y coordinates of the center, a radius r, and a time coordinate. The system calculates a path through an arbitrary number of objects, consisting of several connected line segments. The robot can then move along these line segments at a constant velocity without fear of collision.

The system [illegible] be broken into two main modules, which will be called the collision detector [illegible] the replanner in this [illegible]. These two pieces interact to develop the collision free path. The replanner is capable of using two methods for replanning [illegible] the "[illegible]" and "tangent" methods. Both of these methods will be described.

The path which Avoid produces is constrained by the workspace. The workspace is defined by a region called the scheduling bubble. The scheduling bubble is described, and it is shown that the robot remains inside of the bubble throughout the [illegible].

The collision detector.

It is the job of the collision detector to find the chronologically first collision point of the robot with any object. The collision detector is given a three coordinate initial position and a three coordinate goal position. It is possible the robot attempts to reach [illegible] position from the start position with a straight line constant velocity trajectory. This may not be possible because of a collision with one of the obstacles. It is the collision detector which determines the plausibility of a straight line path.

The collision detector is given a start configuration and [illegible] configuration which define a constant velocity line of movement from the two configurations as it derives the equations of motion of the robot as

$$x_r = [illegible] t + [illegible] \qquad (1)$$

$$y_r = [illegible] t + [illegible] \qquad (2)$$

[illegible]

The system maintains a list of objects. Each object is described by its radius and a list of coefficients which describe the motion of the object. The equations of motion are restricted to second degree polynomials of the following form

$$x_i = a_0 + a_1 t + a_2 t^2 \qquad (3)$$

$$y_i = b_0 + b_1 t + b_2 t^2 \qquad (4)$$

[illegible] the collision detector must solve the equation

$$(x_r - x_i)^2 + (y_r - y_i)^2 = r_i^2 \qquad (5)$$

That is it must find the point at which the distance of the robot to the center of the object squared is the same as the radius of the object. Substituting in the above equation and simplifying one gets the following forth degree polynomial in t

$$[illegible] \qquad (7)$$

where

$$[illegible] \qquad (8)$$

$$[illegible] \qquad (9)$$

$$[illegible] \qquad (10)$$

$$[illegible] \qquad (11)$$

$$[illegible] \qquad (12)$$

$$[illegible] \qquad (13)$$

[illegible] then be found using [illegible] and factoring [illegible] the others.

[illegible]

The scheduling bubble

The scheduling bubble is an imaginary circle of a certain radius. It moves in a straight line at [illegible] velocity from the initial start point to the initial goal point. The robot is required to stay inside the scheduling bubble throughout its movement. A proof is [illegible]

[illegible] the scheduling bubble defines the workspace [illegible]. This workspace is an oval whose width is the same size as the diameter of the scheduling bubble and having as its [illegible] the line segment defined by the start and goal point. The oval extends [illegible] the start and goal [illegible] the radius of [illegible] (Fig. [illegible]).

In addition to defining a workspace, the scheduling bubble defines a time schedule for the robot. The robot must always be within [illegible] distance given by the radius of the bubble [illegible] the "beeline point". The beeline point is the point where the robot would be [illegible] a straight line at constant velocity from the initial start to the initial goal. This point is the center of the scheduling bubble at a given time (Fig. [illegible]).

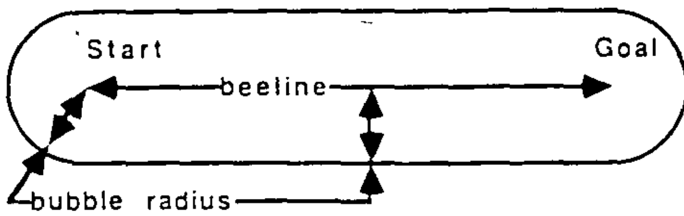


Fig. 1a. An example Workspace defined by the start and goal position, and the scheduling bubble radius

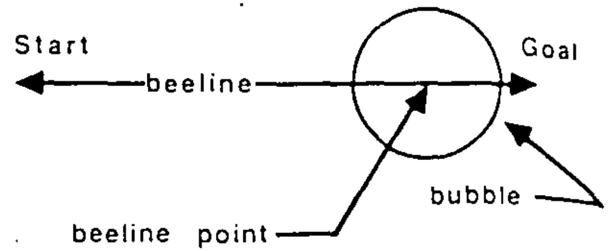


Fig 1b. The workspace, defined by the scheduling bubble at time 8. In this example the robot must leave the Start at time 0 and arrive at the Goal at time 10. At time 8 the robot must be somewhere within the bubble.

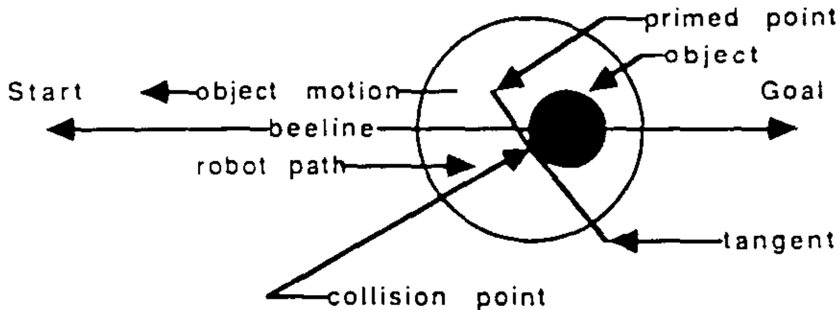


Fig 2. Priming the robot at collision. The robot is relocated along the tangent to the object, at the collision point. In this case it is moved along the tangent half the distance to the scheduling bubble.

The replanner.

The replanner is used by Avoid to replan a path from an initial point to a goal point once it has been determined that a straight line path results in a collision. Its job is to produce a subgoal point. An attempt will then be made to plan a straight line path from the start point to the subgoal and a straight line path from the subgoal to the original goal.

A "snapshot" is taken at the point in time where the collision occurred. A snapshot is simply a representation of the objects and robot's positions at the time of collision, which has been determined by the collision detector. In addition the snapshot contains the location of the scheduling bubble and the location of the goal which the robot was attempting to reach when the collision occurred. This latter piece of information is used by the field method but not by the tangent method.

The tangent method.

We begin the actual discussion of the replanner with the tangent method because it is simpler. The replanner can operate in either tangent or field mode. When in tangent mode the replanner uses less of the information available to it but dramatically saves computing time.

[illegible] It then finds a new position at the collision time in free space. This new [illegible] subgoal.

[illegible] at the collision point. The subgoal is the [illegible] at the head of the vector. The task of defining this vector is broken down into two parts: finding the direction, and finding the magnitude. The direction of the vector is taken to be the tangent of one of the objects with which the robot has collided (Fig. 2). The robot may be touching several objects but it suffices to find only one of them. That is the direction of the replanning vector is

$$r_1 = y_r - y_o \quad (4)$$

$$r_2 = x_o - x_r \quad (5)$$

or

$$r_1 = y_o - y_r \quad (6)$$

$$r_2 = x_r - x_o \quad (7)$$

where (x_r, y_r) is the location of the robot and (x_o, y_o) is the location of the object which touches the robot.

190

The tangent to the object is used for the following reason. If the robot collides with only one object, then space is divided into two half-planes by the tangent to the object at the collision point. If the robot moves in a direction into the half-plane containing the object, it enters the object initially. If the robot travels into the other half-plane it may be traveling in free space, depending on the location of the other objects. The robot may collide with at most two objects simultaneously. This is due to the property of circles that no more than two circles can intersect at exactly one point. If the robot collides with two objects then traveling into either of the half-planes results in entering an object. However traveling in either of the two directions defined by the tangent is guaranteed initially to place the robot in free space. This is why the subgoal vector follows the tangent.

The magnitude of the replanning vector is based on three calculations. The first calculation is the minimum, over all objects not touching the robot, of the distance from the robot to the circular obstacle boundary. The distance is a directed distance along the replanning vector. The second calculation is the distance of the robot to the scheduling bubble boundary. This is again a directed distance.

Let d_o be the distance to the closest object, r_i be the radius of one of the objects the robot is touching, and d_b be the distance to the bubble. Then

$$d^* = \min\left(\frac{d_o}{2}, \frac{d_b}{2}, r_i\right) \tag{18}$$

is the magnitude of the replanning vector.

The replanner must still choose which direction along the tangent to take. This choice is based on the magnitude calculated along each direction. The direction with the greatest magnitude is chosen. This is done to keep the robot from getting caught in a corner. The replanning vector constructed in this way places the robot at a point in free space, and inside the scheduling bubble. The proof of this is that he has moved at most half the distance to the boundary of the scheduling bubble, and at most half the distance to the nearest object.

From the above formula, the robot moves at most the radius of one of the objects it is touching. This is a purely arbitrary step size.

The field method is now considered. This method builds on the tangent method by adding more information into the subgoal choice.

The field method.

The field method is initially very similar to the tangent method. It uses the tangent method to calculate a subgoal point. This operation is called "priming". It then uses Khatib's [14] artificial fields to move the point to an hopefully more effective point. It uses all the information used by the tangent method, and in addition it needs the location of the subgoal which the detector was trying to reach at the time of collision.

The replanner first sets up an artificial potential field around various items in the workspace. These fields are of two types, repulsive and attractive. Repulsive fields are placed around objects, and attractive fields are placed around the beeline point and the subgoal. So the robot is being pushed away from objects, and pulled toward the straight line constant velocity path and the current subgoal.

The repulsive field for the ith object is calculated as

$$u_i = \eta_i \left(\frac{1}{\rho_i} - \frac{1}{\rho_0}\right)^2 \tag{19}$$

where ρ_i is the minimum distance from the robot to object i and η_i the gain. On the boundary of the object these fields are infinite, becoming 0 at a distance of ρ_0 from the object. Attractive fields are of the form

$$u = \eta\left(\frac{1}{\rho - \epsilon} - \frac{1}{\epsilon}\right)^2 \tag{20}$$

where ρ is the distance from the [illegible] to the [illegible] center, and η the gain. These fields are placed around points, and so are 0 at the point and infinite at a distance of ϵ from the point. The total field felt by the robot is then

$$U = \sum_i u_i + u_b + u_s \tag{21}$$

where u_b is the attractive field around the beeline point, and u_s is the attractive field around the subgoal point.

U is a function of x and y. The replanner finds a local minimum of U using a variant of Newton's method (see Gill [15]). The initial point used in the algorithm is the primed point from the tangent method (Fig. 3). To minimize the function U, the following equation is solved:

$$\nabla U(x, y) = 0$$

The method also checks for saddle points, and uses a steepest descent method to get off the saddle point. The local minimum of U is the new subgoal generated by the replanner.

The field extent used for the beeline [illegible] is the radius of the scheduling bubble. This assures that [illegible] will stay within the scheduling bubble. The extent of the field around the current subgoal [illegible]

$$[illegible] = \left| \vec{T}_s - \vec{T}_c \right| + r \qquad (2)$$

where $\left| \vec{T}_s - \vec{T}_c \right|$ is the distance from the subgoal to [illegible] point. This assures that the whole scheduling bubble is contained in the subgoal field.

The overall planner.

Let us now take a look at the overall strategy of the planner. The problem to solve is to get the robot from some start position to a goal position. First the collision detector is used to find a collision. If no collisions occur, the problem is solved. If a collision occurs, the replanner produces a subgoal. The planner then calls itself recursively on the two subproblems generated by the subgoal. The first subproblem is to get from the initial point to the subgoal without collision. The second subproblem is to get from the subgoal to the goal without collision (Fig. 4).

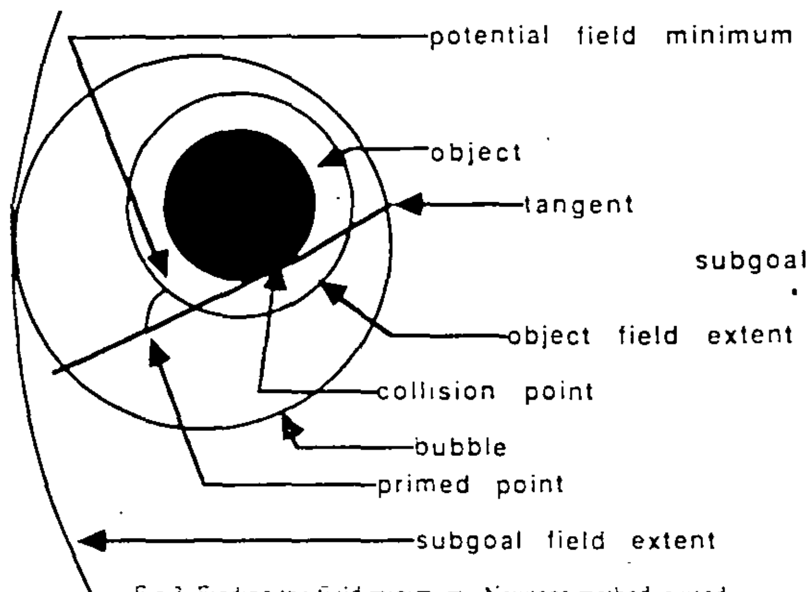


Fig 3. Finding the field minimum. Newtons method is used to find the total field minimum, and relocate the robot, after priming. The subgoal field and the beeline field are attractive, and the object fields are repulsive.

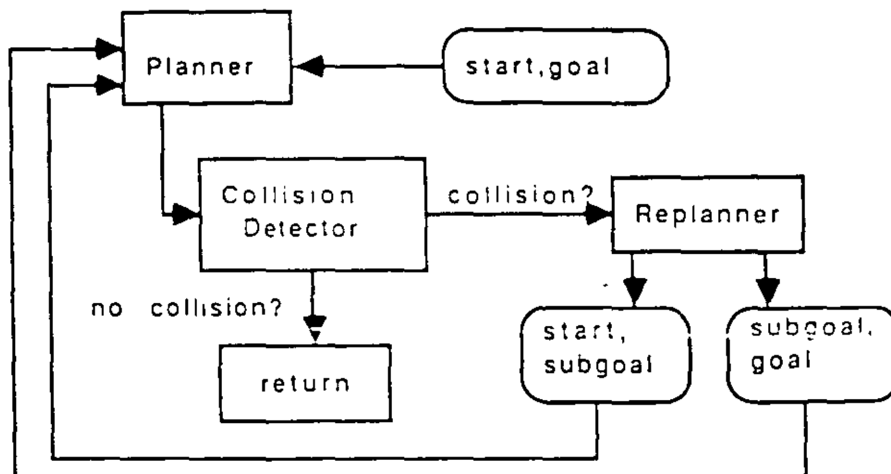


Fig. 4. Chart of the Planner Algorithm. The input to the Planner is a pair representing the current start position, and the current goal position.

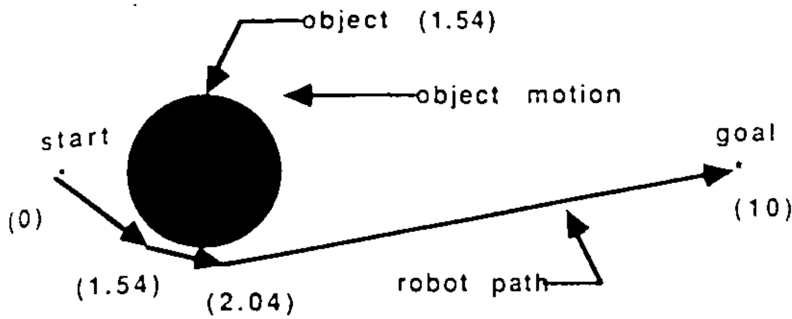


Fig. 5a. Example results. An object of size 1 is moving directly towards the robot, at a uniform rate, starting 3 units away. The robot must reach the goal 10 units away, directly behind the object. The solution shown is produced by the tangent method. The numbers indicate the time of the position.

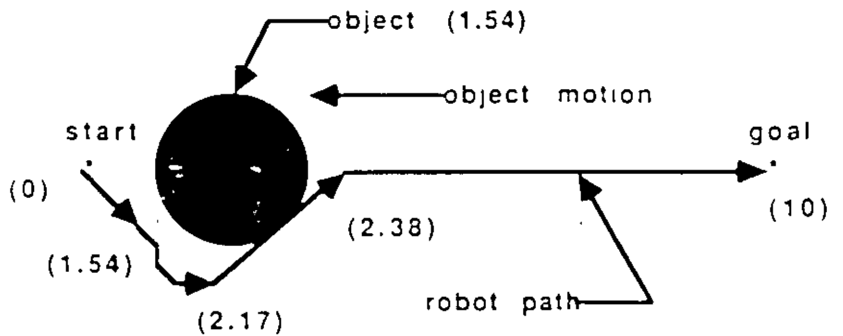


Fig. 5b. Same example as Fig. 5a, except that the field method was used to produce the results.

Conclusion.

An example run of the system is presented here. Both the tangent and field methods are presented (Fig. 5). The example seen has the robot trying to arrive at a point 10 units away in ten time units. The object is moving directly towards the robot at a constant speed which will bring it from a point 3 units away from the robot to the starting position of the robot in 10 time units.

Now various extensions to the system are discussed. One of the problems with the system is that the tangent method cannot handle certain pathological cases, although it is much faster than the field method. This problem could be overcome by combining the two methods, using the field method only in those pathological cases which are incorrectly handled by the tangent method.

A version of the system in three dimensions is currently under development. This extension uses the natural extensions to the theory built up for two dimensions. The complexity of object shapes is limited by their representation as spheres. It would be difficult, for example, to represent [illegible] as [illegible] radius zero. One possible solution to this problem is to represent objects as a set of spheres moving together in some relationship.

Another extension to the system would involve unknown object paths. A vision system could be used to locate objects and determine their local velocities. From this the system would project the path [illegible] used [illegible] plan a small motion, execute it, and take another [illegible].

Proof of bubble containment

This is the proof that the robot is always inside the safe [illegible] bubble.

Theorem

Let $\overline{T}_1(t)$ and $\overline{T}_2(t)$ be two points moving [illegible] between two endpoints t_1 and t_2 in straight lines. Further assume that $\nabla\overline{T}_1$ and $\nabla\overline{T}_2$ are both constant vectors. If

$$\left|\overline{T}_1(t_1)-\overline{T}_2(t_1)\right|<s \tag{21}$$

and

$$\left|\overline{T}_1(t_2)-\overline{T}_2(t_2)\right|<s$$

then for any $t_1 \le t \le t_2$

$$\left|\overline{T}_1(t)-\overline{T}_2(t)\right|<s \tag{25}$$

Proof

To prove this theorem we assume that

$$t_1=0 \qquad (27)$$

$$\vec{r}_a(t_2)=(0,0) \qquad (28)$$

The results of this case are easily extended to the general case. The proof starts with the proof of a lemma.

Lemma

For any two vectors such that

$$\left|\vec{a}\right|<\epsilon \qquad (29)$$

$$\left|\vec{b}\right|<\epsilon \qquad (30)$$

then

$$\left|\vec{a}(1-\lambda)+\vec{b}(\lambda)\right|<\epsilon \qquad (31)$$

where $0\le\lambda\le 1$

Proof

We know the following inequalities

$$\left|\vec{a}(1-\lambda)+\vec{b}(\lambda)\right|<\left|\vec{a}(1-\lambda)\right|+\left|\vec{b}(\lambda)\right|<\epsilon(1-\lambda)+\epsilon(\lambda)=\epsilon \qquad (32)$$

This proves the lemma.

To prove the theorem we note that the points have constant velocities, hence

$$\vec{r}_a(t)=\vec{m}_a t \qquad (33)$$

$$\vec{r}_b(t)=\vec{m}_b t+\vec{r}_b(0) \qquad (34)$$

where

$$\vec{m}_a=\frac{\vec{r}_a(t_2)}{t_2} \qquad (35)$$

$$\vec{m}_b=\frac{\vec{r}_b(t_2)-\vec{r}_b(0)}{t_2} \qquad (36)$$

Then the distance between points at time t is

$$\left|\vec{r}_a-\vec{r}_b\right|=\left|\vec{r}_b(0)\left(1-\frac{t}{t_2}\right)+\left(\vec{r}_b(t_2)-\vec{r}_a(t_2)\right)\frac{t}{t_2}\right| \qquad (37)$$

But

$$0\le\frac{t}{t_2}\le 1 \qquad (38)$$

so by the lemma just proven

$$\left|\vec{r}_a(t)-\vec{r}_b(t)\right|<\rho \qquad (39)$$

which was to be proven.

Bibliography

1. Lozano-Perez, Tomas & Mason, Matthew T. & Taylor, Russell H. Automatic synthesis of fine-motion strategies for robots. The International Journal of Robotics Research, Vol. 3, No. 1, Spring 1984.

2. Lozano-Perez, Tomas. Automatic planning of manipulator transfer movements. Robot Motion, Planning and Control, Brady, M. & Hollerbach, J. M. & Johnson, T. L. & Lozano-Perez, T. & Mason, M. T. The MIT Press, Cambridge, Massachusetts, 1982, pp. 499 - 520.

3. Paul, Richard P. C. Manipulator Cartesian path control. Robot Motion, Planning and Control, Brady, M. & Hollerbach, J. M. & Johnson, T. L. & Lozano-Perez, T. & Mason, M. T. The MIT Press, Cambridge, Massachusetts, 1982, pp. 245 - 264.

4. Taylor, Russell H. Planning and execution of straight-line manipulator trajectories. Robot Motion, Planning and Control, Brady, M. & Hollerbach, J. M. & Johnson, T. L. & Lozano-Perez, T. & Mason, M. T. The MIT Press, Cambridge, Massachusetts, 1982, pp. 265 - 285.

5. Brady Michael. Trajectory planning. Robot Motion Planning and Control. Brady M. & Hollerbach, J. M. & Johnson T. L. & Lozano-Perez T. & Mason M. T. The MIT press. Cambridge, Massachusetts 1982 pp. 221 - 211

6. Brooks Rodney A. Solving the find-path problem by good representation of free space. IEEE Transactions on Systems Man Cybernetics. March/April 1983 Vol. SMC-13 No. 3 pp. 190 - 197

7. Sater Richard M. Planning in a continuous domain - an introduction. Robotica 1983, Vol. 1, pp. 85 - 93

8. Cameron Stephen. A study of the clash detection problem in robotics. IEEE International Conference on Robotics and Automation 1985 pp 488 - 493

9. Atkinson Kendall E. An Introduction to Numerical Analysis. John Wiley and Sons Inc. 1978, pp. 85 - 88

10. Khatib Oussama. Real-time obstacle avoidance for manipulators and mobile robots. IEEE International Conference on Robotics and Automation 1985, pp 500 - 505

11. Gill Philip E. Murray Walter and Wright Margaret. Practical Optimization. Academic Press. 1981 pp. 99 - 115

Handling Real-World Motion Planning: A Hospital Transport Robot

J. Evans, B. Krishnamurthy, B. Barrows, T. Skewis, and V. Lumelsky

The objective of the hospital transport mobile robot, HelpMate, developed recently by Transitions Research Corporation is to carry out such tasks as the delivery of off-schedule meal trays, lab and pharmacy supplies, and patient records. Unlike many existing delivery systems in the industry which operate within a rigid network of wires buried or attached to the floor, a hospital transport robot is expected to be able to navigate much like a human would, including handling uncertainty and unexpected obstacles. The navigation system of HelpMate makes use of the specific structure of a hospital hallway environment and relies on recent results on provable sensor-based motion planning algorithms that specifically address the issue of navigation in an unknown and unstructured environment. Also incorporated in the system are algorithms for handling unmodeled factors such as sensor noise and sensor inaccuracy, errors in position estimation, and moving obstacles (e.g., people).

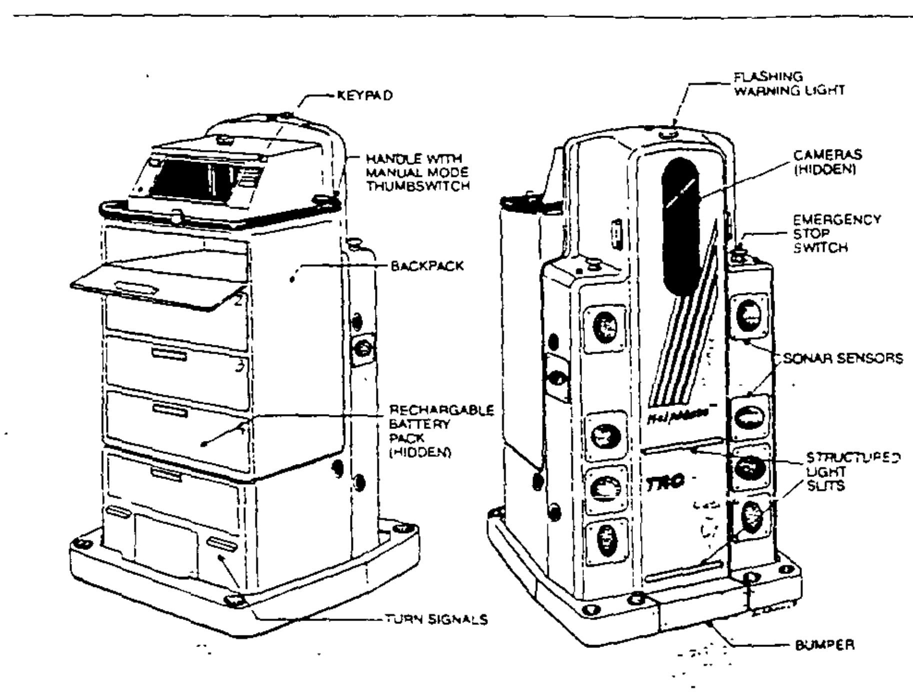


Fig. 1. HelpMate's sensor system.

The HelpMate Project

The hospital transport mobile robot, HelpMate, developed by Transitions Research Corporation is expected to become one answer to the current shortage of help in hospitals. Specifically, it addresses the need for assistance with such tasks as point to point delivery. Examples of delivery tasks include meal trays, pharmacy and lab supplies, patient records, and mail. Given the unstructured environment of a typical hospital, one technical objective of the HelpMate project is to successfully handle the uncertainty present and to navigate based only upon as little prior knowledge as possible. Also, the HelpMate should require little modification to a given hospital and thus make a maximum use of on-board sensing of the natural environment. A more complete requirements analysis of this project can be found in [1]. A discussion of technical problems such as elevator control and communication, cargo security, user interface, etc. can be found in [2]. In this article, only the navigation system of HelpMate is reviewed.

The HelpMate navigation system makes use of the recent work on motion planning with incomplete information [3], [4]. An algorithm developed in [3], called *Bug2*, allows a point automaton with tactile sensing to travel from some start point *S*, to some target point *T* with no prior knowledge of the presence of obstacles in the environment.

Very briefly, the algorithm operates as follows. First, the automaton attempts to proceed toward the target point along a prescribed path. If it contacts an obstacle, [it] moves along its boundary until it crosses t[he] prescribed path at a distance from the targ[et] shorter than that between the contact point a[nd] the target. (The distance is computed along t[he] prescribed path. It is therefore Euclidean d[is]tance only if the prescribed path is a straig[ht] line segment.) Then, the automaton resum[es] its motion along the prescribed path tow[ard] the target and the process repeats until eit[her] the target is reached or a certain conditio[n is] satisfied which indicates that the target can[not] be reached. The path produced by the Bu[g2] algorithm is referred to as the Bug2 path.

It has also been shown that with a pro[per] modification Bug2 can be used for navigat[ing] a disc-shaped finite size mobile robot. In [this] case, the algorithm is applied to the underly[ing] configuration space, rather than to the w[ork]space. The required information inclu[des] "simulation" of tactile sensing in the c[on]figuration space which is accomplished [by] covering the perimeter of the real robot [with] tactile sensors. Better yet, the algorithm ca[n be] further improved by making use of n[...]

A previous version of this paper was presented at the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, April 7-12, 1991. J. Evans, B. Krishnamurthy and B. Barrows are with Transitions Research Corporation, Danbury, CT 06810. T. Skewis was with Yale University, New Haven, CT 06520. V. Lumelsky was with Yale University, New Haven, CT 06520. He is now with the Department of Mechanical Engineering, University of Wisconsin-Madison, 1513 University Ave., Madison, WI 53706.

0272-1708/92/$03.00©1992IEEE

sensing media, such as range sensing or vision. The resulting sensor-based planning algorithm, called *VisBug* [4], was shown to deliver a quite reasonable performance in complex environments. The main idea of the version of VisBug used in this work is to continuously identify the furthest point of the Bug2 path that has a clear path to it and then proceed directly to that point. (For the proof of convergence see [4].) This identified point is referred to as an *intermediate target point*. As the Bug2 path lies along either the prescribed path or an obstacle boundary so will the intermediate target points.

In the version of the VisBug algorithm adopted in the HelpMate navigation system, no prior information about obstacles is assumed. On-board sensor information is used for navigation. To make use of prior information about the robot environment — specifically, of the hospital hallway layout — the algorithm operates within a higher level subsystem which generates a prescribed path exemplified by *landmark* features, such as walls, or landmark objects along the robot path that are "known to be there." In other words, the assumption of incomplete information is used only to maneuver around obstacles, the robot always knows its position relative to the hospital layout. The navigation system continuously identifies intermediate target points along the prescribed path when possible or along an obstacle boundary if necessary. To assure feasibility, intermediate target points always lie within the range of sensors.

To date TRC has installed HelpMates in several hospitals including Danbury Hospital, Danbury, CT, Downey Hospital, Downey, CA, and Mt Sinai Medical Center, New York, NY. In some hospitals HelpMate is in operation 24 hours per day and is being used for the delivery of meal trays, lab supplies, etc. The hospitals are reporting an increase in productivity and efficiency due to the HelpMate and are finding it to be cost effective. The following sections present in more detail the requirements on the robot navigation system stemming from a typical hospital environment, and the HelpMate navigation capabilities.

The Hospital Environment

The HelpMate perceives the hospital environment as a set of predefined stations which are connected by a network of hallways, elevator lobbies, and elevators. For example, for the application of meal tray delivery, the set of stations would be the dietary center and various nursing units. To travel to a specified station the HelpMate may need to ma[illegible] elevator. This task has been simplified by installing an elevator control computer that the HelpMate communicates with through an infrared transceiver. Similarly, robot controlled door openers are installed on doors that the HelpMate may encounter.

Hallways are typically 6 to 8 feet wide with many doors and alcoves along the walls. Some hallways are free of objects but have a constant flow of people in both directions. Other hallways are so cluttered with medical equipment and carts that passage is impossible until the obstacles have been moved. Placing of wires or tape on the floor to guide the HelpMate is not allowed. Within this environment, the HelpMate is expected to navigate along the hallways at a speed of approximately 600 mm/s when the path is free of obstacles and 300 mm/s while maneuvering around obstacles.

HelpMate Drive and Sensor Subsystems

The HelpMate drive subsystem has two independently controlled drive wheels whose common axis is centered on the robot. Thus, the HelpMate is a nonholonomic system without a bounded steering angle. A spring suspension keeps the drive wheels in contact with the floor even when its rough or bumpy and four casters at the corners provide stability. The drive subsystem accepts forward and angular velocity commands from which it calculates the corresponding left and right wheel velocities. It has a maximum speed of 1000 mm/s and a maximum acceleration of 1000 mm/s^2.

The HelpMate is equipped with ultrasound and structured light range sensing. This sensing continuously provides a set of range values, each with an associated origin on the robot and direction of sensing. The ultrasound sensors are Polaroid transducers driven by Texas Instruments sonar modules. The location of the transducers are indicated in Fig. 1. The transducers operate on the principle of timing the return of a transmitted pulse and using the speed of sound to calculate distance. The structured light sensor is a CCD camera positioned above two structured light projectors. The location of the camera and projectors are indicated in Fig. 1. The camera points downward and the projectors are parallel to the floor in a high and low arrangement. Triangulation is used to calculate the range along discrete directions to objects within a field of view of [illegible]. Infrared filters are used on both the camera and the projectors to both reject external light sources and to render the projected light invisible to people. For details on the mathematics of structured light range sensing see [5].

The use of these sensors for both detecting obstacles and identifying walls helps determine their location and orientation on the robot. Sensing obstacles directly in front of the robot is critical and so both ultrasound and structured light sensing are located in the front. Ultrasound sensors are also located on the side of the robot, to provide enough information about an obstacle while avoiding it, and to provide information about walls while traveling parallel to them. As ultrasound sensors are defeated by specular reflection, the orientations of these sensors are chosen so as to cover a wide range of obstacle orientation. Also, since some obstacle surfaces have to be detected at only particular heights (e.g., in case of a table), the sensors are located at various prespecified heights.

To detect obstacles that might go undetected by the range sensors, touch sensitive "bumpers" are mounted on the front and back of the HelpMate. (See Fig. 1.) These sensors are tied directly into the drive subsystem and will stop the robot when triggered, thus overriding any current command issued by the navigation system. An LCD and keypad provide a user interface. Turn signals and a warning light are used to indicate when the HelpMate is deviating from a straight line path such as when turning a corner or avoiding an obstacle. Emergency stop switches and a manual control handle are provided for situations where the hallway must be cleared quickly. The HelpMate has a locked backpack for carrying meal trays, lab supplies, etc.

All motion planning software runs on a Motorola 68000 based single board computer acting as the system master. Several Motorola 68HC11 based computers are used in the drive, ultrasound, and other subsystems. An IBM PC computer and a video digitizing board are used in the vision subsystem. A high speed serial network with a master/slave protocol allows communication between the master and all subsystems.

Hallway Navigation

Geometric and topographical information about the hospital hallways, elevator lobbies, elevators, and stations are generated by the installation engineer via an off-line CAD system tailored for HelpMate applications. Application specific information such as travel restricted hallways, speed restricted hallways, station arrival announcements and any prespecified station lists for rounds are also input with the CAD system. This CAD data is transformed into a data structure referred to as the Topography Knowledge Base (TKB). The TKB is loaded into the HelpMate as part of the installation process.

The HelpMate system architecture is shown in Fig. 2. HelpMate operation begins when a member of the hospital staff enters via a keypad the mission, a sequence of stations to be visited. Stored information about the hospital, the TKB, is used in a one-time planning operation to calculate a route, a sequence of hallways to accomplish the given mission. (No need has been found for "replanning.") The navigation of a hallway is accomplished by switching between the actions of: 1) following a prescribed path along the hallway and 2) avoiding an obstacle. The former uses an estimation of the robot's position and heading within a hallway and the latter uses a local map of obstacles. The algorithms for both actions produce forward and angular velocity commands for the drive subsystem.

Convergence of navigation algorithms is a relatively simple problem for the HelpMate, thanks mainly to the availability of the hallway layout and to the fact that the robot can be safely assumed to be locked inside the hospital. Normally, the robot proceeds by following the prescribed path. When an obstacle prevents following this path, points along the obstacle boundary are identified in order to guide the robot around it.

A hospital hallway with obstacles presents such a simple environment topologically that the obstacle avoidance algorithm works satisfactorily with a simple rule that prescribes the robot to always proceed within an angular range of ±90° about the direction of a hallway. This way, if HelpMate concludes that the current obstacle cannot be avoided — e.g., the hallway is completely blocked — it will simply wait until it has cleared. It will not, for example, follow a wall back down the hallway; unlike the general motion planning problem with incomplete information, there is no need in this application to explore other hallways for a path to the destination. The width of a typical hallway plus the types of objects that are typically encountered in a hospital suggest that the "mazes" the HelpMate will be presented with are relatively simple.

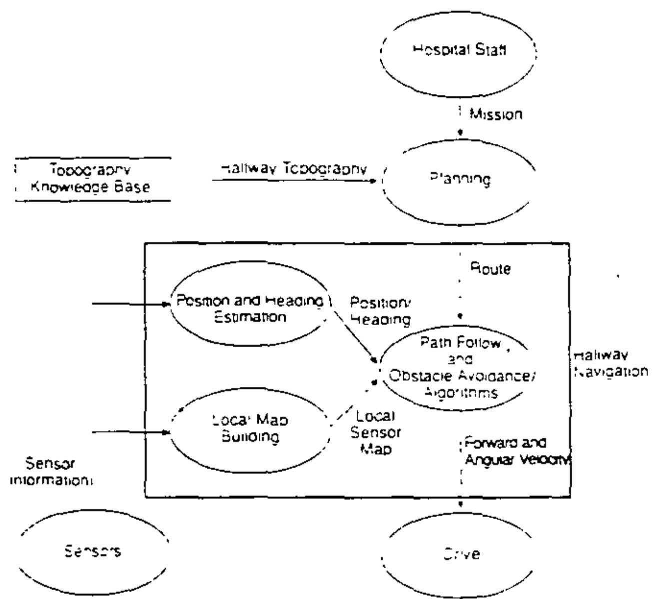


Fig. 2. HelpMate system architecture

Following the Prescribed Path

The route generated to accomplish a given mission is in the form of a program consisting of steps that need to be performed either serially or in parallel. The robot is given a path to traverse along in each hallway and a means of getting from one hallway to another. This prescribed path is a mathematically defined curve specified by a finite sequence of line segments. Each line segment is defined by its length and the angle relative to the previous segment.

Following the prescribed path amounts to more than simply moving point-by-point along the path. The robot need not be on the prescribed path in order for it to be "followed." This fact is important due to the continuous use of position estimation using natural landmarks. At one instant the robot's position may be on the prescribed path and at the next instant it will be significantly off the path. To follow the prescribed path, the navigation algorithm calculates a point along the prescribed path at some distance from the robot and the robot is then directed towards this point [6]. This technique provides the desirable "smooth" performance. If following the path is prevented by an obstacle then the robot will be directed so as to maneuver around the obstacle.

Obstacle Avoidance

Since the obstacles in the hallways are constantly changing, obstacle geometry and positions cannot be prestored in the HelpMate memory. Obstacle avoidance must be an on-line operation with information about the environment continuously obtained from on-board sensors. While traveling along the prescribed path, sensor information about the HelpMate's immediate surroundings is continuously analyzed. If an obstacle is detected obstructing the prescribed path, the robot will leave its path if that is necessary to avoid it. A simple rule is used to handle transient objects: the robot decelerates to within a prespecified distance of the obstacle and pauses for some time before it decides to leave the prescribed path. If the obstacle has moved during this pause then HelpMate will simply continue following the prescribed path as if noth[...] happened. If the obstacle is still present th[...] sensor information is used to decide on [...] most promising direction for maneuver[...] around the obstacle, either right or left.

The navigation algorithm does not use [...]formation directly from the plurality of He[...]Mate sensors for the purpose of obsta[...] avoidance. Instead, all sensor data is poo[...] into a local sensor map (LSM) approxima[...] 4 m². The LSM is an occupancy grid re[...]sentation [7] of the robot's local environm[...] in Cartesian coordinates. Each cell represe[...] sensor derived knowledge manifesting it as a level of certainty as to whether or not [...] cell is occupied. This LSM is a form [...] memory, as the robot moves the cells [...] "scrolled" in order to keep the robot ce[...] within the center cell without deleting [...]viously gathered data about the immed[...] environment. Detection of transient obj[...] and noisy sensor data are overcome by a [...] derived decay which periodically decre[...] the certainty level and eventually erases [...] map resident knowledge.

The robot's obstacle avoidance is c[...]posed of three distinct modules: "path de[...]tion," "path selection," and "path track[...]" Path detection analyses this robo-centric [...] every 150 ms to decide on areas that are [...] of clutter and provides the path selec[...] module with a list of possible paths tha[...] robot may traverse through. Using TKB [...] about the hallway being traversed the se[...] is restricted to the boundaries of the hal[...] allowing for position and orientation u[...]tainty. To simplify the scanning all obst[...] are "grown" and the robot "shrunk" to a [...]

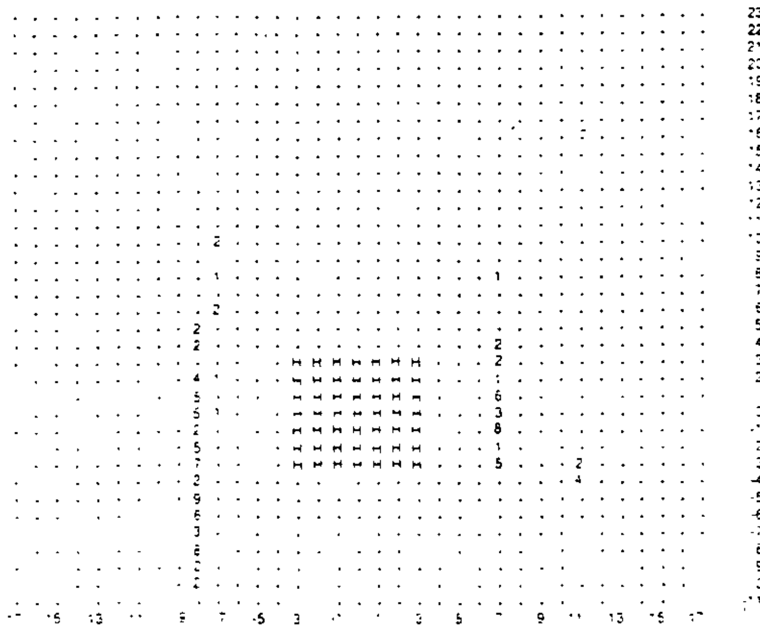


Fig. 3. Local map — robot navigating a hallway.

similar to the configuration space proposed in [8]. The paths thus generated are weighted by the path selection module according to the suitability and depending on the state of the robot. The final path selected is chosen for ease of maneuverability of the robot while it is trying to move towards its final destination. The path tracking module directs the robot along the selected path by sending forward and angular velocity commands to the robot's drive subsystem. Approximately, 15 to 25 ms depending on the scene in front of the robot, are spent in the obstacle avoidance algorithm.

In the obstacle avoidance algorithm, points along the accessible (visible) part of the obstacle boundary are identified. In identifying the obstacle boundary points, only sensor information from the local map is used. The identified points are then used as intermediate target points so that the HelpMate's path around the obstacle is smoother than if it "hugged" the boundary, following point-by-point. Furthermore, the obstacle boundary crossing the prescribed path needs not be identified before points along the prescribed path are identified as intermediate targets. As soon as a clear path to the prescribed path has been identified the robot ceases the obstacle avoidance maneuver. This assures smoother performance and shorter paths.

One interesting problem encountered in our hospital application is that of discouraging people from "playing" with the robot. TRC has found that a navigation algorithm that makes the robot pause before it starts maneuvering around an obstacle and that makes the robot to only proceed in one preferred direction is relatively uninteresting to a lay person. People soon become bored with the robot and move on, allowing it to carry out its task.

Example of Obstacle Avoidance

Figs. 3 and 4 are two snapshots of the LSM during a typical obstacle detection and avoidance maneuver. The LSM is a two dimensional array of 35×35 cells. Cell numbers can be found along the right and the bottom edge of the map. The robot is centered on cell coordinate (0,0). The square represented by H's is the HelpMate robot oriented facing the hallway. The robot's orientation is not represented in the LSM and is only used when detecting, selecting, and tracking along the path. Each dot represents an empty cell. A cell represented by a number rather than a dot indicates that an object has been detected in that cell. The higher the number the higher the certainty that an object is in that cell position. Fig. 3 is a snapshot of the robot along a hallway with no obstacles: note the sensed walls to the left and right of the robot. Fig. 4 shows the obstacles detected in front and to the right of the robot. Subsequently the robot passes the obstacles to the left, the obstacle is still being detected on its right side by the robot's right facing sensors. When the robot is well past the obstacle, the obstacle still appears behind the robot and will get either scrolled out of the map or decayed away.

Position and Heading Estimation

Estimation of the HelpMate's position and heading is accomplished through the sensing of drive wheel positions and environment landmarks and is performed continuously. Dead reckoning is a relatively simple and commonly used technique. Incremental left and right drive wheel angular displacements are used in a discrete integration to calculate the HelpMate's current position and heading. Unfortunately, dead reckoning suffers from various inaccuracies and its errors accumulate over distance. The magnitude of the dead reckoning errors are such that HelpMate cannot

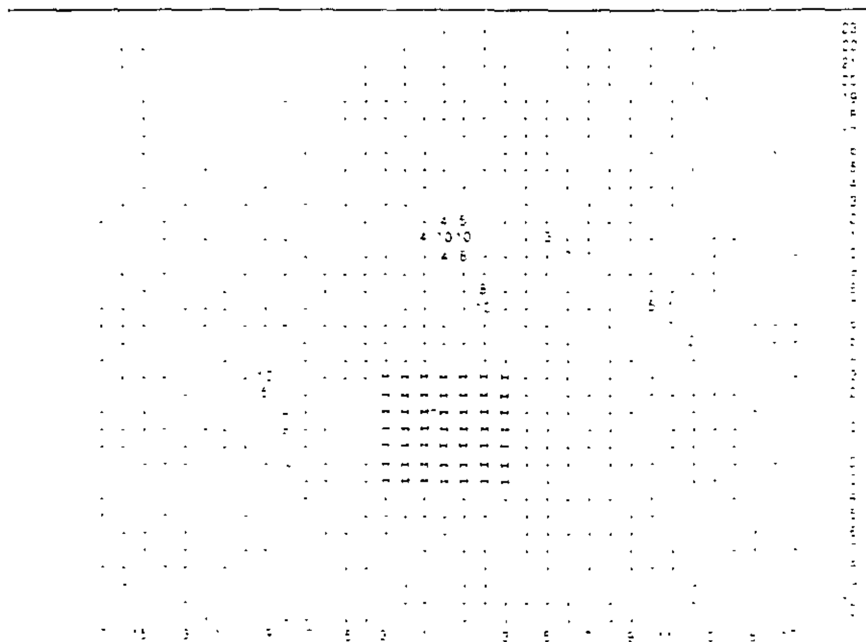


Fig. 4. Local map — robot encountering an obstacle.

navigate successfully by dead reckoning alone.

As the objective of HelpMate is to operate in an environment with a mostly fixed layout, the errors due to dead reckoning are corrected by use of natural landmarks. This process is referred to as *registration*. Information about the natural landmarks is provided in the hospital layout stored in the HelpMate memory. Currently, only one type of natural landmark is used: hallway walls. The walls indicate the lateral position and heading of the robot in a hallway. Doorways, alcoves, objects and people may limit the amount of wall information that the HelpMate has to register with.

Hallway walls are identified through the use of ultrasound and structured light range sensing. As the robot moves along the hallway, range values are obtained and used to calculate point objects which are stored in a list. The pattern recognition technique of an *iterative endpoint fit* is used on a list of points to extract a line segment [9]. A line segment of sufficient length consisting of a sufficient number of points is assumed to be a wall. Then, the HelpMate's lateral position and heading with respect to the wall is calculated.

Further Research

The currently available results of the HelpMate testing indicate that the chosen navigation system provides a performance that is suitable for the chosen environment. This work has also identified areas in need of further research. For example, more alternatives for registration would provide more reliable position and heading estimation. More intelligence in obstacle avoidance would be desired — it would help, for example, if the robot could at a distance distinguish between stationary and moving obstacles.

References

[1] J. Evans, B. Krishnamurthy, W. Pong, T. Skewis, B. Barrows, S. King, C. Weiman and G. Engelberger, "Creating smart robots for hospitals," in *Proc. of Robots [illegible] Conf.* Washington DC, May 1989.

[2] B. Krishnamurthy, B. Barrows, S. King, T. Skewis, W. Pong and C. Weiman, "Helpmate: A mobile robot for transport applications," in *Proc. 1988 SPIE Conf.*, Boston, MA, Nov. 1988.

[3] V. Lumelsky and A. Stepanov, "Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape," *Algorithmica*, vol. 3, no. 4, 1987.

[4] V. Lumelsky and T. Skewis, "Incorporating range sensing in the robot navigation function," *IEEE Trans. Syst. Man Cybern.*, vol. 20, no. 5, pp. 1058-1069, 1990.

[5] S. King and C. Weiman, "Helpmate autonomous mobile robot navigation system," in *Proc. 1990 SPIE Conf.*, Boston, MA, Nov. 1990.

[6] R. Wallace *et al.*, "First results in road following," in *Proc. Joint Int. Conf. Artificial Intell.* Los Angeles, CA, Aug. 1985.

[7] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proc. 1985 IEEE Int. Conf. Robotics and Automation*, St. Louis, MO, Mar. 1985.

[8] T. Lozano-Perez and M. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Communications of the ACM*, vol. 22, no. 10, pp. 560-570, 1979.

[9] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*. New York: Wiley, 1973.

John Evans' accomplishments are in the technical and managerial areas of robotics and automation. He received the B.A. degree from Yale University and the Ph.D. from the University of Colorado, both in physics. In 1970 he joined the staff of the Director of the National Bureau of Standards. In 1972, he started the Automation Technology Program at NBS. This program has achieved recognition as one of the major research efforts in sensor and control technology, particularly in the development of concepts of hierarchical control. Dr. Evans was instrumental in the formation of a new Center for Mechanical Engineering, which combined several programs to provide technical support for industry in automated manufacturing. In 1978, he joined Gerber Scientific where he became Vice President, Engineering and Operations of Gerber Systems Technology, a newly formed subsidiary responsible for a CAD/CAM product line. Dr. Evans formed his own company, Nova Robotics, in 1981. This company developed an industrial robot system targeted at electronic assembly and featuring a sophisticated software system designed for sensory feedback and off-line programming. Subsequently, he accepted the position of President of Transitions Research Corporation to pursue new opportunities in robotics and automation. Dr. Evans has over 30 publications on robotics and automation, has been recognized by several awards, and has been a member of many professional societies and committees.

Bala Krishnamurthy has spent the past twelve years in the design, control and applications of robots in industrial environments. She holds the B.S. degree in mathematics from Rutgers University (1971), the M.S. degree in applied mathematics from Stevens Institute of Technology (New Jersey), and her second M.S. in computer science from Polytechnic Institute of Technology. From 1971 to 1973, Ms. Krishnamurthy worked at Olivetti Corporation of America as a Systems Analyst and designed and implemented accounting packages on all their accounting machines. In 1979 she joined Unimation as a software engineer and was instrumental in productizing the VAL language on the series of Unimation robots. After briefly leading a team of software engineers in productizing the VAL II system, Ms. Krishnamurthy was chosen to lead the software design and development on Unimation's third generation controller developmental effort subsequently known as the UNIVAL system. The Westinghouse special award for technical leadership on the UNIVAL project was awarded to Ms. Krishnamurthy in 1985. Since 1986, she has been responsible for the software/system design and implementation of not only HelpMate, but all other TRC mobile robot projects as well.

B. Barrows, photograph and biography not available at time of publication.

Tim Skewis received the B.S. degree in electric[al] engineering from Lehigh University, Bethlehem, PA, in 1983, and the M.S. and Ph.D. degrees [in] electrical engineering from Yale University, New Haven, CT in 1986 and 1991. From 1984-1987 [he] was on the Technical Staff of Unimation Inc., Danbury, CT. He developed motion planning and trajectory software for the UNVAL robot arm controller. From 1987-1989 he was on the Technical Staff [of] Transitions Research Corp., Danbury, CT, [and] developed navigation software for HelpMate, [a] mobile robot for hospital transport tasks. He [is a] recipient of a Connecticut High Technology Scholarship and is a member of Tau Beta Pi and [Eta] Kappa Nu. His research interests include sensor-based motion, mobile robots, and graphical robot simulation.

V. Lumelsky is currently Consolidated Pap[ers] Professor of Engineering [in] the Department [of] Mechanical Engineering and Department of Electrical and Computer Engineering, University [of] Wisconsin-Madison. [He] received the B.S. and M.[S.] degrees in electrical and computer engineering [from] Leningrad in 1962, and the Ph.D. degree in app[lied] mathematics from the Institute of Control Scie[nces] (ICS), U.S.S.R. National Academy of Scien[ces], Moscow, in 1970. From 1967 to 1975 he [held] academic positions in ICS conducting researc[h in] pattern recognition, cluster analysis, factor anal[ysis] and control systems, and served concurrently [as] Adjunct Professor at Moscow Institut[e of] Radioelectronics and Automation. From 19[7?] [to] 1980 he was on the research staff at Ford M[otor] Company Scientific Laboratories, Dearborn, and from 1980 to 1985 on the research sta[ff of] General Electric Corporate Research C[enter], Schenectady, NY. From 1985 to 1990 he was o[n the] faculty of the Department of Electrical Engine[ering] at Yale University.

A Modular Agent/Deliverable Modality for Mobile Robot Development

Robert W. Albrecht
Department of Electrical Engineering
University of Washington
Seattle, Washington 98195 USA

Abstract

An agent/deliverable modality has been formulated for the purpose of enabling the development of mobile robots in an environment where the personnel consists of a large number of neophytes (undergraduate and beginning graduate students) together with a small number of dedicated but transient researchers (MS and PhD thesis students) and a still smaller number of permanent researchers (faculty). The objective is to create a structured work environment in which projects that are implemented by neophytes can be easily and systematically integrated into an ongoing broader program of mobile robot development. In addition, the environment must provide a learning experience for the neophytes.

1: Modularity

If an educational institution has the ambition to educate students and to use student projects to contribute to the progress of a large and diverse research and development program, it is necessary to pay careful attention to the organization of the work. Without systematic organization it is likely that the student work will be lost in archives and never contribute to the project. With a carefully devised modular structure it is possible that the combined goals of student education and progress in a complex project can be accomplished.

To reach this objective, a structure has been formulated that is used as both a component of the learning environment and as the basis for creating and modifying modules in such a way that neophyte generated projects are suitable for integration into the ongoing research program. This modality uses the concepts of "agent" and "deliverable". An agent performs tasks that result in specific deliverable results that may, in turn, be used by other agents as components of a hierarchical robot control system.

2: Agents

An outline of the agents that are under development in this program [1] and the interactions between agents are shown in Fig. 1. The discussion in this section refers to Fig. 1.

It is assumed that the entire robot task begins with the specification of task descriptions, environments, and certain limitations and constraints. These activities are performed by human agents.

The inputs from the human agents enter a specialized knowledge base either directly or through the action of the computerized agents known as surveyor [2] and cartographer [3]. This speciali[illegible] knowledge is composed for the particular problem [illegible] hand and it is an instance of a more general knowledge base [4].

The specialized knowledge base contains the computerized agents known as planner, navigator, pilot and controller which are shown in a hierarchical configuration [5]. These agents may be supplemented by simulation.

Only the controller agent interacts directly with the actuators and sensors of the robot (or an emulated robot) through an appropriate interface. The robot operates within an environment which may be enhanced with beacons, reflectors, scenes, etc. Of course this could also be an emulated robot within a simulated environment.

3: Deliverables

Each agent is responsible for delivering a certain item that is carefully defined. It is extremely important to be precise when specifying the responsibilities of agents so that development can proceed in a systematic way

The human agent is responsible for delivering tasks or setting goals and schedules for the performance [illegible] tasks, specifying the environment in which the task [illegible]

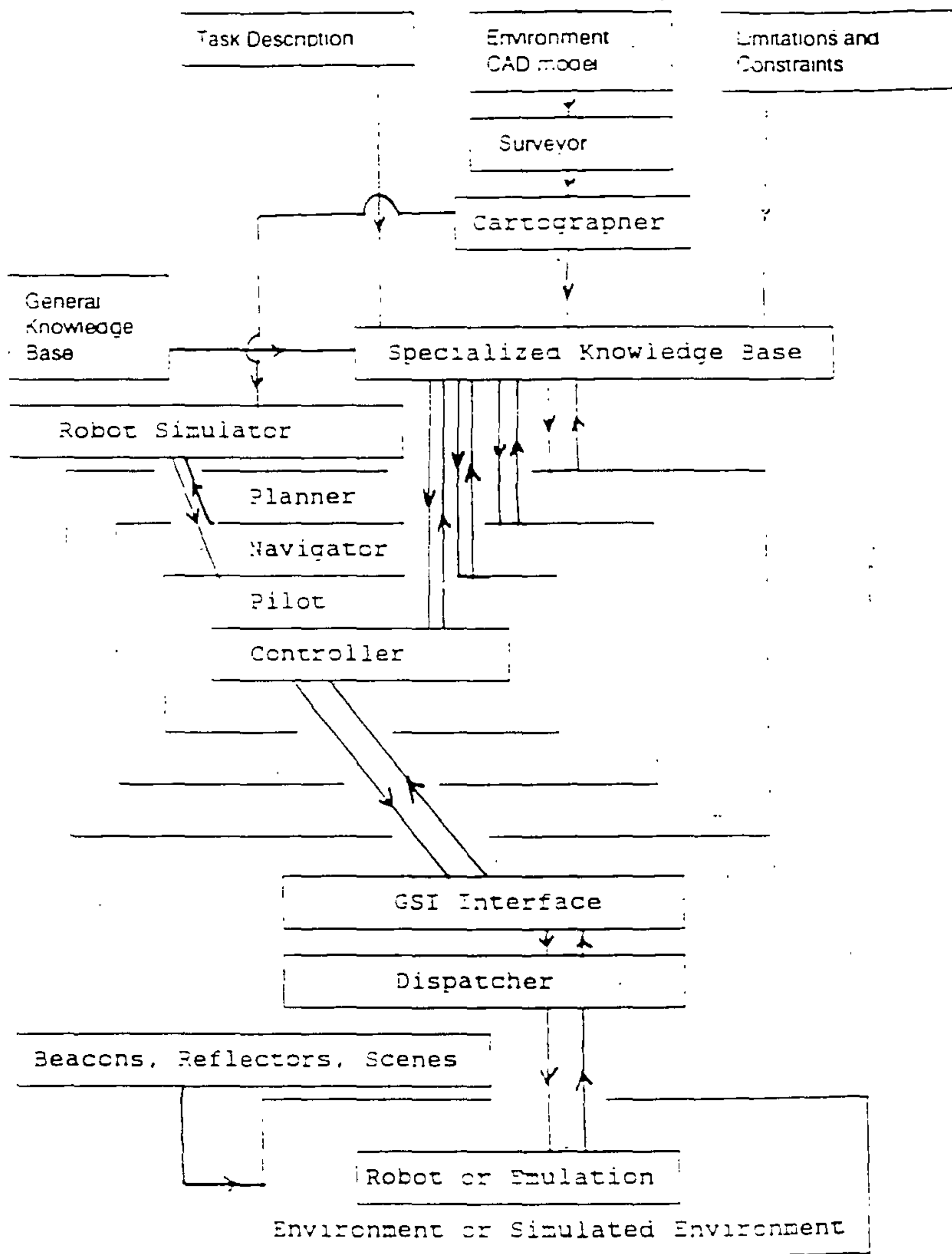


Figure 1. Interacting Agents for Mobile Robots Development

take place, and defining limitations and constraints associated with safety, speed, environmental modifications, etc. The human agent *delivers* these items to the autonomous robot system. We assume that if these items are not delivered to the autonomous robot system then the system will not function.

We do not assume that the robot is sufficiently autonomous to invent its own goals and associated task descriptions. We do not assume that the robot is an explorer that totally defines its own environment through exploration. Of course, we do assume that in addition to the defined environment there may be transient and temporal objects to deal with. We do not assume that the robot can derive its own value system concerning safety, environmental impact, etc.

The surveyor agent produces deliverables that are map-features, connections, relations, and dimensional-references. Map features are objects (in the classic sense of OOP). Connections and relations express interrelationships between objects. Dimensional-references are a special class of objects that refer to coordinate systems and measured quantities such as distances and angles. These deliverables are currently produced either directly by a human agent or by the surveyor from CAD models. Plans are being implemented to produce these deliverables from robot exploration and to use exploration as a means to enhance the knowledge of the environment.

The cartographer agent produces a deliverable that is an atlas. An atlas consists of maps, indices to maps, and useful information about spaces. These maps are in the form of iconized cartograms with geometric and relational information contained in the attributes (symbolic, numeric, and graphical) of objects that reside in a knowledge base suitable for robot consumption.

The planner agent produces deliverables called plans. Plans are lists of symbols and quantities that specify a procedure to change the state of the robot and environment from some given condition to some goal condition. These plans are made on the basis of available information. Of course the plans may not account for information that is unavailable to the planner.

The navigator agent delivers positional awareness in both quantitative and qualitative form. In addition to a positional fix, the navigator agent delivers a safe domain and a local subgoal. That is, the navigator delivers a bounded domain in which the pilot can operate to reach a subgoal from the current positional fix without reference to maps, plans, etc.

The pilot is the agent that receives sensor data from the controller, receives a safe domain, a position fix, and a local subgoal from the navigator, and delivers commands for driving, turning, or executing various combinations of maneuvers and sensing to th controller. These deliverables are in the form of highe. level specifications.

The controller agent receives higher level specifications for maneuvering and sensing from the pilot and composes these into command strings that are delivered to the robot multitasking operating system

The robot (or an emulated robot) is the agent that delivers actual motion and actual sense data. Motion and sensing are delivered by the multitasking operating system scheduling these tasks, informing expert modules to carry out the tasks, monitoring for task completion or interruptions, and reporting results to the pilot agent.

4: Modality

The modality for carrying out mobile robot development consists of a curriculum for neophytes, MS and PhD research activities for transient researchers and integration by permanent researchers.

Curriculum: The curriculum that has been structured for neophytes to make contributions to the overall mobile robot program covers three academic quarters of ten weeks duration each. In the first quarter the neophytes concentrate on learning the programmi environment (object oriented, real-time expert systen learning to interface this programming environment to a real robot, and learning the details of the protocol that is required to ultimately produce a module that can be integrated into the ongoing program. The learning environment is structured by providing neophytes with a proto-agent and a proto-deliverable which have the structure of useable agents and deliverables but have no content. Figure 2 shows the heirarchical structure for a typical proto-agent.

In the process of providing content to the proto-agent and the proto-deliverable the neophytes learn the protocol and structure for all present and future mobile robot agents and deliverables. All research projects that contribute to the development of an agent and/or a deliverable must produce results that are in conformity with the corresponding proto-structures. Introducing the proto-agent and proto-deliverable and enforcing a research and development pattern that produces results that correspond to this pattern are the keys to obtaining project results that can ultimately be integrated. Both the proto-agent and proto-deliverable provide hierarchies of objects, messages, connections, variables, and parameters that are structured to be compatible with any ultimate applications module. Emphasis placed on keeping these structures organized so th

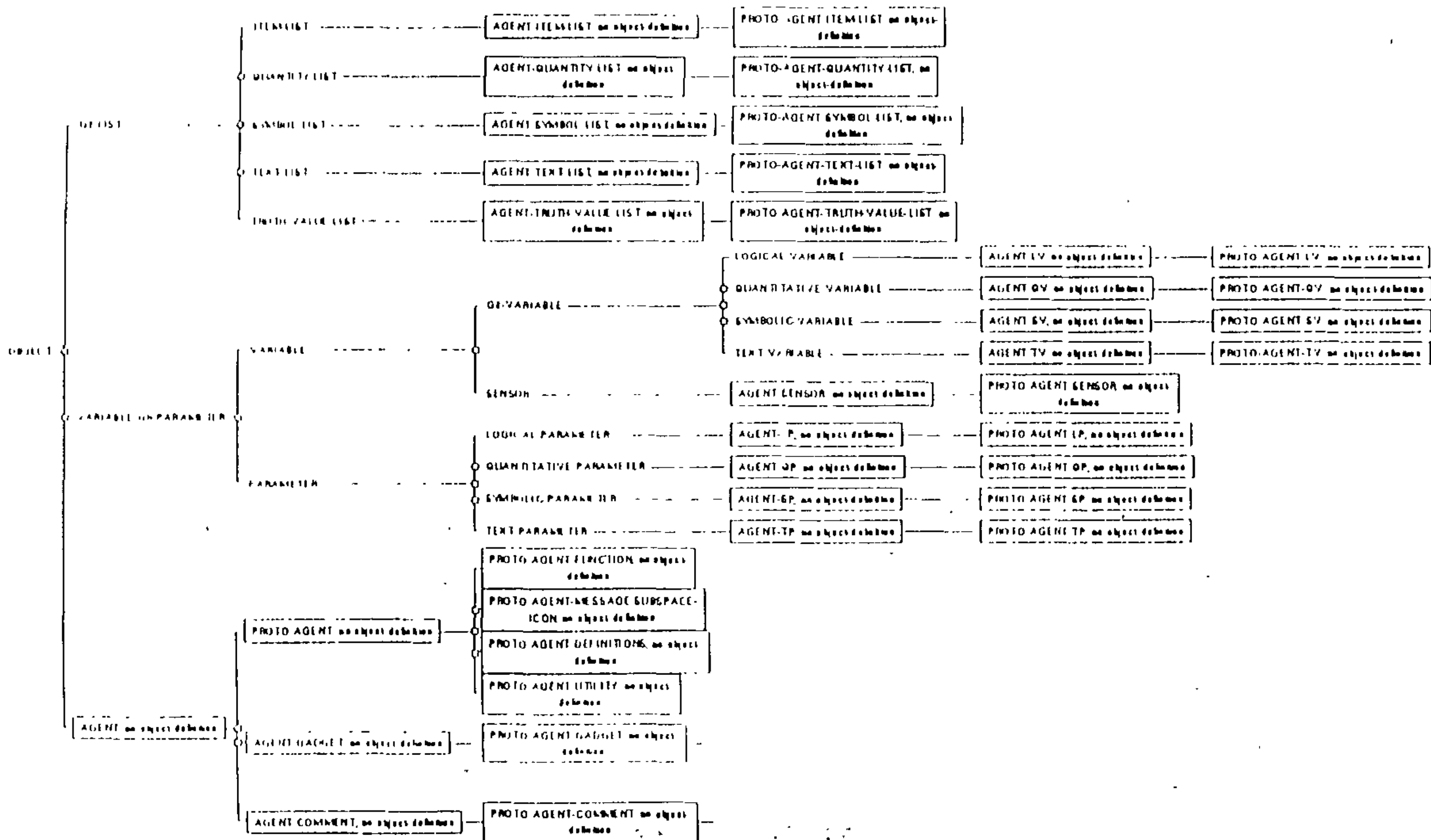


Figure 2. Typical Hierarchy of a Proto-Agent

interfaces to other agents and deliverables are transparent. Note that the proto-agent illustrated in Fig. 2 only shows the definitions of high level proto-objects (definitions of messages, connections, and specialized gadgets are not shown). Instances are not shown. An operating agent or deliverable will include many object definitions that inherit attributes from the superior objects.

Figure 3 shows the object hierarchy of definitions for the cartographer agent. Since the cartographer agent has no lists and does not define any new variables or sensors, objects of these classes do not appear. The cartographer agent appears in the class hierarchy below agent. Other agents will have a similar structure and their definitions will merge seamlessly with the cartographer. The deliverable associated with the cartographer is world. World has many lists and a much richer array of object definitions than cartographer. This is typically the case. Deliverables consist of a large variety of objects and a minimum of procedures and rules. Agents consist of a large number of procedures and rules and a minimum of object classes. Thus the hierarchical structure of object definitions for world retains different portions of the proto-structure than does the hierarchy for cartographer. All deliverables will merge seamlessly with each other and with agents. By enforcing this modularity, one can construct partial or complete robot operating hierarchies from the individual components at will. Figure 4 shows a merge of the cartographer with a particular world named ee-nrb. Note how the hierarchies are separated. This structure is maintained for any number of agents and deliverables that are merged.

In the second quarter, the neophytes are introduced to the details of actual agents and deliverables that have been previously produced. Careful attention is paid to the question of integrating modules into a coherent and cooperating system. During the second quarter the neophytes develop a project description that (when approved) forms the basis for the project that is executed in the third quarter.

In the third quarter, neophytes execute projects that must be demonstrated by integrating the project results into the existing framework for mobile robot behavior and showing that the project module enhances a skill of the robot vehicles.

Typical project modules that have been implemented in the past are:

The raw atlas for a limited world (a deliverable) consisting of maps constructed in the form of iconised cartograms whose attributes represent information concerning the interior environments of two buildings and one outdoor environment between the buildings.

A cartographer (an agent) that uses the raw atlas information to produce a completed atlas consisting of completed maps, indices, and other useful information.

Student research activities: Students who pursue research projects may enter the research program through one of two paths. The first path is to enter as a neophyte and to complete the neophyte curriculum discussed above. The second path is a shorter and more intense route. In this second path, research students must complete a self-paced instruction course that leads them through the modular hierarchies by syllabus and leads them through current agents and deliverables by a combination of syllabus and programmed instruction packages called robotutors. After students complete the programmed instruction phase of the second path, they must participate in a course entitled "research and development in mobile robotics" which introduces them to current research progress and available topics.

Typical research projects that have been completed or are in progress include:

A surveyor agent that provides attributes to the objects in the atlas (map-feature deliverables) translated from CAD models.

A planner agent that provides plans (deliverables) based on information concerning the robot's current position and orientation, its goal position ar orientation, world information, and timi requirements.

A pilot agent that provides local reflexive bahavior (a deliverable). Such behavior includes local obstacle avoidance, door traversal, local goal seeking, etc.

A controller agent that provides performance orders (deliverables) that are sent to the physical or emulated (an agent).

The physical robot is an agent that produces motion (a deliverable) and sensory data (a deliverable). These deliverables are produced in the context of an on-board multitasking operating system named LLAMA that was developed as part of this program. LLAMA has a structure similar to FORTH.

Faculty: Although the modularity of agents and deliverables combined with an organization of neophytes and transient researchers leads to gradual progress in implementing an integrated autonomous mobile robot, there are significant burdens placed upon the faculty in maintaining this organization. One burden is that the faculty must continue to preach the gospel of modularity as neophytes and transient researchers come and go. A second burden is that the faculty must continually police the work of neophytes and transient researchers to be sure that individu modules do not violate the principles of modularity.

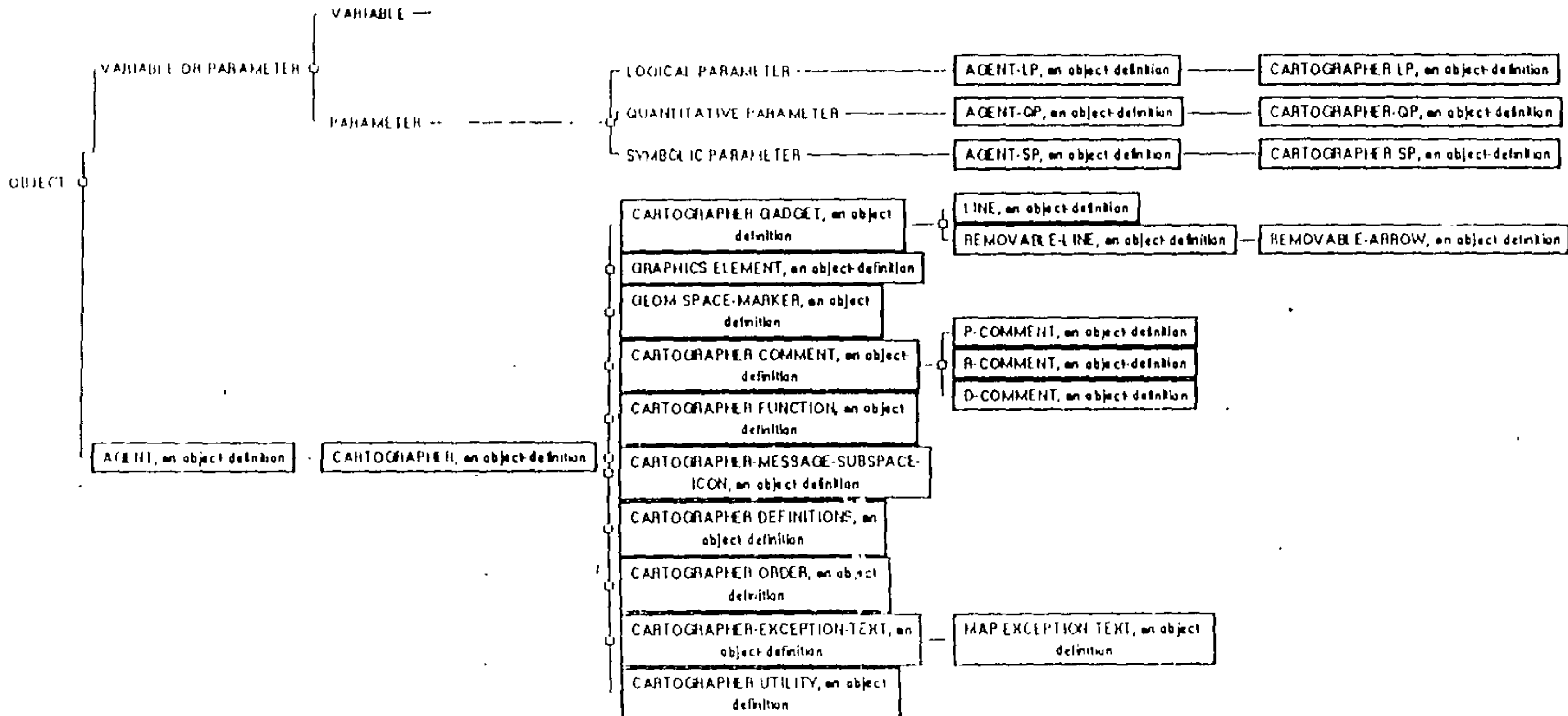


Figure 3. Object Definition Hierarchy of Cartographer Agent

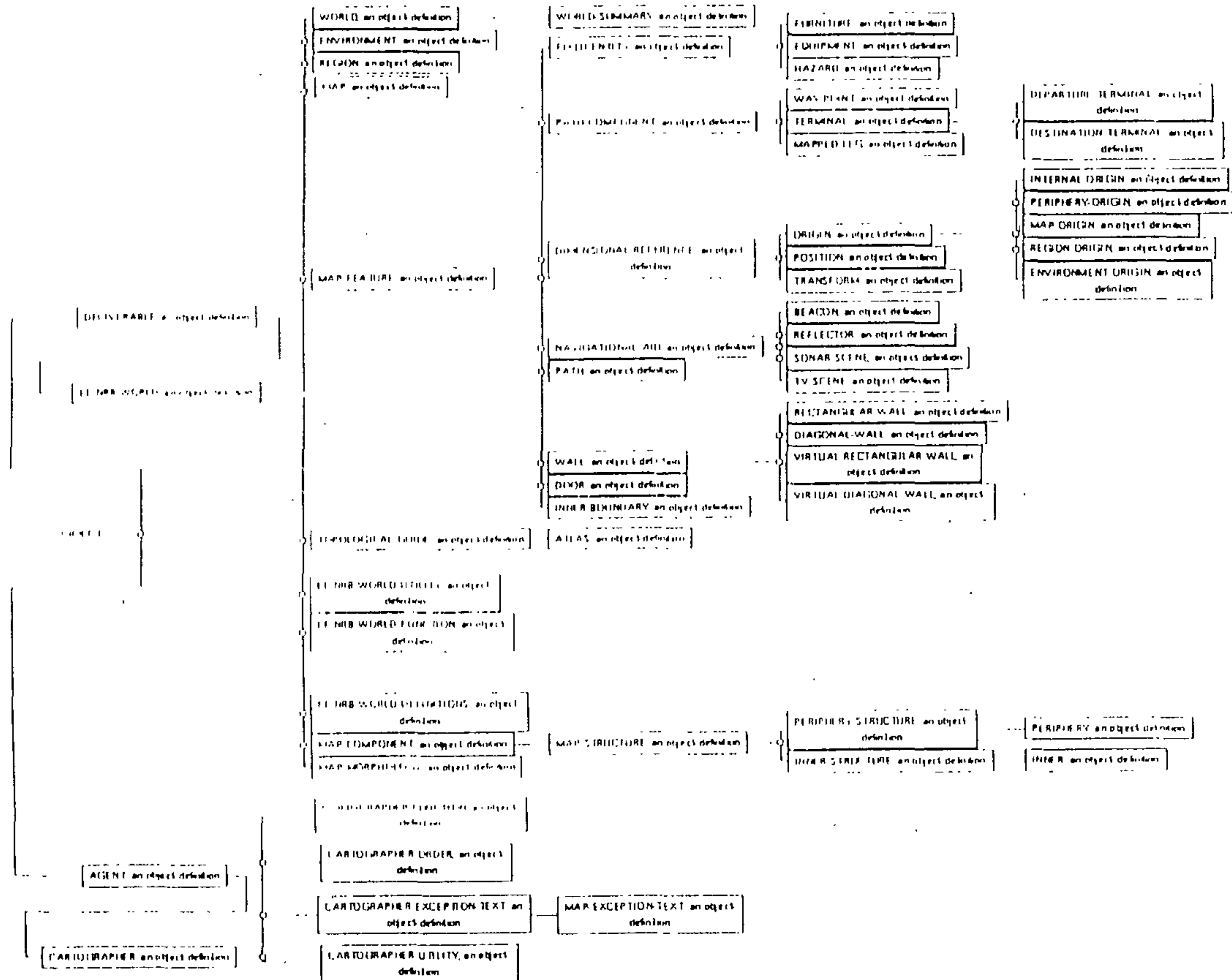


Figure 4. Object Definition Hierarch, . the Merge of Cartographer and World Modules

third burden is that much of the onus of editing syllabi and computerized tutors inevitably falls upon the faculty.

If the faculty does not assume the burdens of preaching, policing, and editing then there is a great risk that the whole edifice that is meant to create a coherent approach to a mobile robot will fall like a house of cards. A chief reason for this is the lack of continuity that could be provided in an industrial setting by intermediate level management personnel who could assume much of the responsibility for maintaining continuity. The symptom of the beginning of the fall of the edifice is when it is discovered that several modules are not capable of being interfaced seamlessly, the individuals responsible for the creation of each of the modules is no longer present, the current neophytes and transient researchers have no stomach for investigating the reasons for this failure, and the faculty is both insufficiently familiar with the details of the work and has insufficient time to repair the rupture in the edifice. The offending modules tend to be ignored by the current neophytes and transient researchers. After being ignored for some time, the functionality of the modules is lost. Sometimes the work is repeated. Often this situation leads to a kind of chaos that ultimately degenerates into the usual pattern of every person for himself and the bold ideas of creating coherence are relegated to the trash can.

5: Mobile Robot Development

An autonomous mobile robot is a complex engineered device. The key to the systematic development of such a device is an organization that facilitates the integration of the contributions from a large number of participants. This kind of integration is especially difficult in an academic environment.

A modality has been formulated that uses an agent/deliverable paradigm for the development of autonomous mobile robots. The development is being implemented by a large group of students over a prolonged period of time.

A principle objective of the modality is to make it possible for neophytes to accomplish tasks within a framework that allows the result of their work to be integrated with the overall goals of the mobile robotics program. This means that the work accomplished by neophytes must be in a modular form with transparent interfaces.

To accomplish this objective, neophytes are introduced to an object-oriented, real-time expert system. They are directed to work through a moderately pedantic syllabus that produces an agent/deliverable module that is in the desired integratable form. They are introduced to modules that have been created by previous neophytes and/or previous research students or faculty. With this background they create new modules (or parts of modules) that can be used to enhance the overall mobile robots program.

The central advantage of this organization is the capability to use neophyte contributions in a cummulative way to reach the goals of a long-term mobile robots developmental program while at the same time providing a meaningful educational experience for neophytes.

The size of the group at any point in time is approximately 30. The time period for the development is anticipated to be 10 years or more with the potential for incremental improvements extending for an indeterminate length of time. The students range from a majority of neophytes (seniors and beginning graduate students in a robots class) to a minority of transient researchers (MS and PhD students) supervised by one or two faculty.

The maintenance of this modality within an educational institution is fragile. Constant vigilance is required to detect and intercept tendencies toward incoherence. Individuality, transient participants and a lack of a strong reward/punishment structure for nonconformity may cause the modality to unravel. Time will tell if this experiment succeeds.

References

[1] R. W. Albrecht, "Mobile Robots: A Paradigm for Complex System Engineering", *Robotics and Computer Integrated Manufacturing*, Vol. 9, No. 1, pp 27-33, 1992.

[2] B. Aldridge, R. W Albrecht, "An Expert System Surveyor for Mobile Robots: Translation of CAD to Knowledge Base Models", Submitted to Conference on Intelligent Control, Metz, France, June, 1992

[3] R. W. Albrecht, "An Expert System Cartographer for mobile Robots", *Proceedings of Thirteenth IASTED International Symposium ROBOTICS AND MANUFACTURING*, 1990

[4] R. W. Albrecht, "Environmental Knowledge Representation for Mobile Robots", *Proceedings of the 1991 International Symposium on Advanced Robot Technology*, 1991

[5] R. W. Albrecht, "Implementation of Semi-Autonomous Mobile Robots in Semi-Structured Environments", *Robotics and Manufacturing*, 3, 941-946, 1990

WHERE AM I?

ROBOT LOGICAL ORGANIZATION

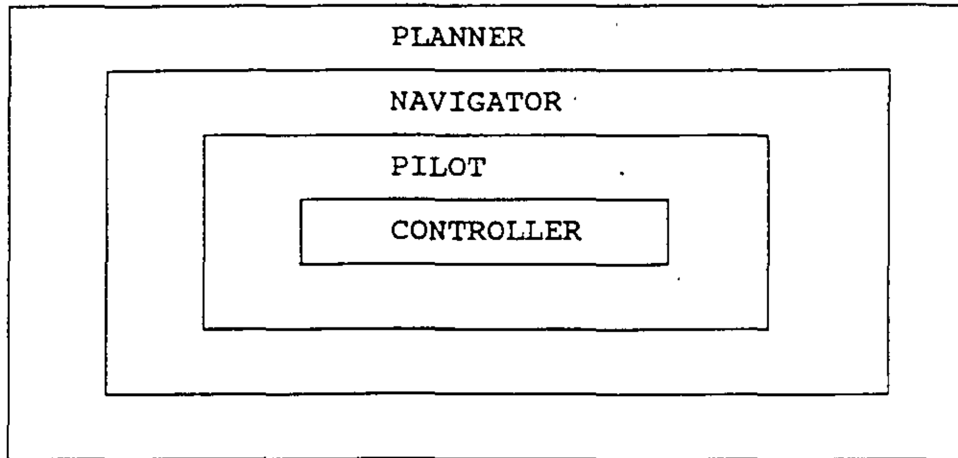


Figure 1: Relationship of Cooperative Agents

INQUIRY DIRECTED TO CONTROLLER

CONTROLLER:

STATUS:

PROPULSION MOTORS NOT TURNING

STEERING MOTORS NOT TURNING

DRIVE MOTOR ENCODER = 0

STEERING MOTOR ENCODER = 0

ANGLE BETWEEN BASE AND WHEELS = 45 DEG. (FIG. 1)

SENSORS NOT ACTIVATED

RESPONSE TO QUESTION:

"NOT A PROPER QUESTION FOR THE CONTROLLER. ASK THE PILOT!"

PILOT:

OPERATION INTERRUPTED (UNKNOWN CAUSE)

NO CURRENT SENSOR DATA (EXCEPT WHEEL ENCODERS)

RESPONSE TO QUESTION:

"NOT A PROPER QUESTION FOR THE PILOT, ASK THE NAVIGATOR!"

NAVIGATOR:

OPERATION INTERRUPTED (UNKNOWN CAUSE)

CURRENT MAP (DATA BASE) IS FOR REGION A, ROBOT PROVING GROUND (RPG)

MOST RECENT MANEUVERS IN REGION A, RPG

NO CURRENT SENSOR DATA

ASK PLANNER FOR INFORMATION CONCERNING "WHERE AM I"

PLANNER:

HISTORY OF PREVIOUS MANEUVERS INDICATES MOTION IN RPG FROM REGION G TO H TO A

GOAL WAS TO OBSERVE BEAM TUBE IN REGION A

NAVIGATOR WAS IN COMMAND WHEN ROBOT PASSED FROM H TO A

PLANNER HAD PASSED RPG-A MAP TO NAVIGATOR

PLANNER RECOGNIZES FOUR POSSIBILITIES:

(1) RPG

(2) RDL

(3) OTHER SPACES (1,2,..N)

(4) LOST

PLANNER "BELIEVES" LOCATION IS REGION RPG-A

PROBABILITIES.

(1) 0.8

(2) 0 0

(3) 0.0

(4) 0 2

TO MODIFY PROBABILITIES, REQUEST NAVIGATOR TO IDENTIFY LOCATION WITH SENSORS

NAVIGATOR:

MAP OF REGION RPG-A SHOWS BEACONS #1, #2, PLUS REFLECTOR (FIG. 2)

NAVIGATOR DIRECTS PILOT TO "LOCATE" BEACONS AND REFLECTOR

PILOT:

FIND BEARING TO BEACON #1, PASS TASK TO CONTROLLER

USE "SCAN 15 DEG, THEN 2 PI"

CONTROLLER:

TURN ON IR BEACON DETECTOR

ROTATE STEERING MOTOR +15 DEG

STOP IF BEACON SENSED

IF NO BEACON, ROTATE STEERING MOTOR -30 DEG

IF NO BEACON, MAKE 360 DEG ROTATION

STOP IF BEACON SENSED

STEERING STOPS

BEARING TO BEACON REPORT:

ANGLE BETWEEN BASE AND WHEELS = 0 DEG (FIG. 3)

REPORT TO PILOT - ALL STOP

PILOT:

PASS BEARING TO NAVIGATOR

FIND BEARING TO BEACON #2, PASS TASK TO CONTROLLER

CONTROLLER:

REPEATS SEARCH SEQUENCE

NAVIGATOR:

DETERMINES CANDIDATE LOCATIONS AND ORIENTATIONS OF ROBOT (FIG 4)

PASSES LOCATION INFORMATION TO PLANNER

PLANNER:

UPGRADES P(RPG-A) = 0.9

CONTROLLER:

REPORTS ANGLE BETWEEN BASE AND WHEELS = 108.43 DEG (FIG 5)

(ALL STOP)

PILOT:

PASS BEARING TO NAVIGATOR

FIND BEARING TO REFLECTOR, PASS TASK TO CONTROLLER

NAVIGATOR:

DETERMINES CANDIDATE LOCATIONS AND ORIENTATION OF ROBOT (FIG. 6)

INFORMATION SUMMARY:

DISTANCE BETWEEN 1 AND 2 = 10.61

ROBOT MUST LIE ON ARC

EQUATION OF CIRCLE IS

$P(x,y) = (x-a)^2 + (y-b)^2 - r^2 = 0$

CONTROLLER:

TURN ON LASER SCANNER

SWEEP + AND - 60 DEG

LOCATE REFLECTOR

BEARING TO REFLECTOR REPORT AT 36.86 DEG

(ALL STOP)

PILOT:

PASS BEARING TO NAVIGATOR

WAIT FOR INSTRUCTIONS

NAVIGATOR:

DETERMINE APPROXIMATE LOCATION AND ORIENTATION OF ROBOT (FIG. 7)

PLANNER:

UPGRADES PROBABILITY TO P(RPG-A) = 1 0

ARE YOU SURE?

NAVIGATOR: (COMMUNICATING WITH PILOT AND CONTROLLER)

CONSIDERS OTHER SENSOR INFORMATION TO VERIFY POSITION

SONAR SCAN (FIG 8)

USE OF BEACON #1 PLUS RANGE TO REFLECTOR (FIG 9)

MOVING TO SELECTED LOCATIONS FOR DIFFERENT ASPECT (FIG. 10)

E. G. EQUIDISTANT FROM BEACONS AND/OR AT INSCRIBED CIRCLE OF BEACONS

CAPTURE A SCENE ON TV

WILL NEED TO MANEUVER TO A POTENTIAL SCENE USING MODEL OF REGION AND CAPTURE A FRAME

PILOT:

GIVES COMMAND TO CONTROLLER TO MOVE FROM (7.5,2.5) TO (5,2.5) ALONG PATH AND TRAJECTORY THAT IS SPECIFIED BY PILOT IN THE FORM "MOVE FROM A GIVEN LOCATION AT A GIVEN ANGLE TO ANOTHER LOCATION AT ANOTHER ANGLE USING THE LEAST TIME TRAJECTORY"

CONTROLLER:

ACTUATES DRIVE AND STEERING MOTORS TO FOLLOW A TRAJECTORY THAT MEETS SPECIFICATIONS

CONTROLLER MAY ACTIVATE SONARS FOR OBSTACLE AVOIDANCE ON THE WAY

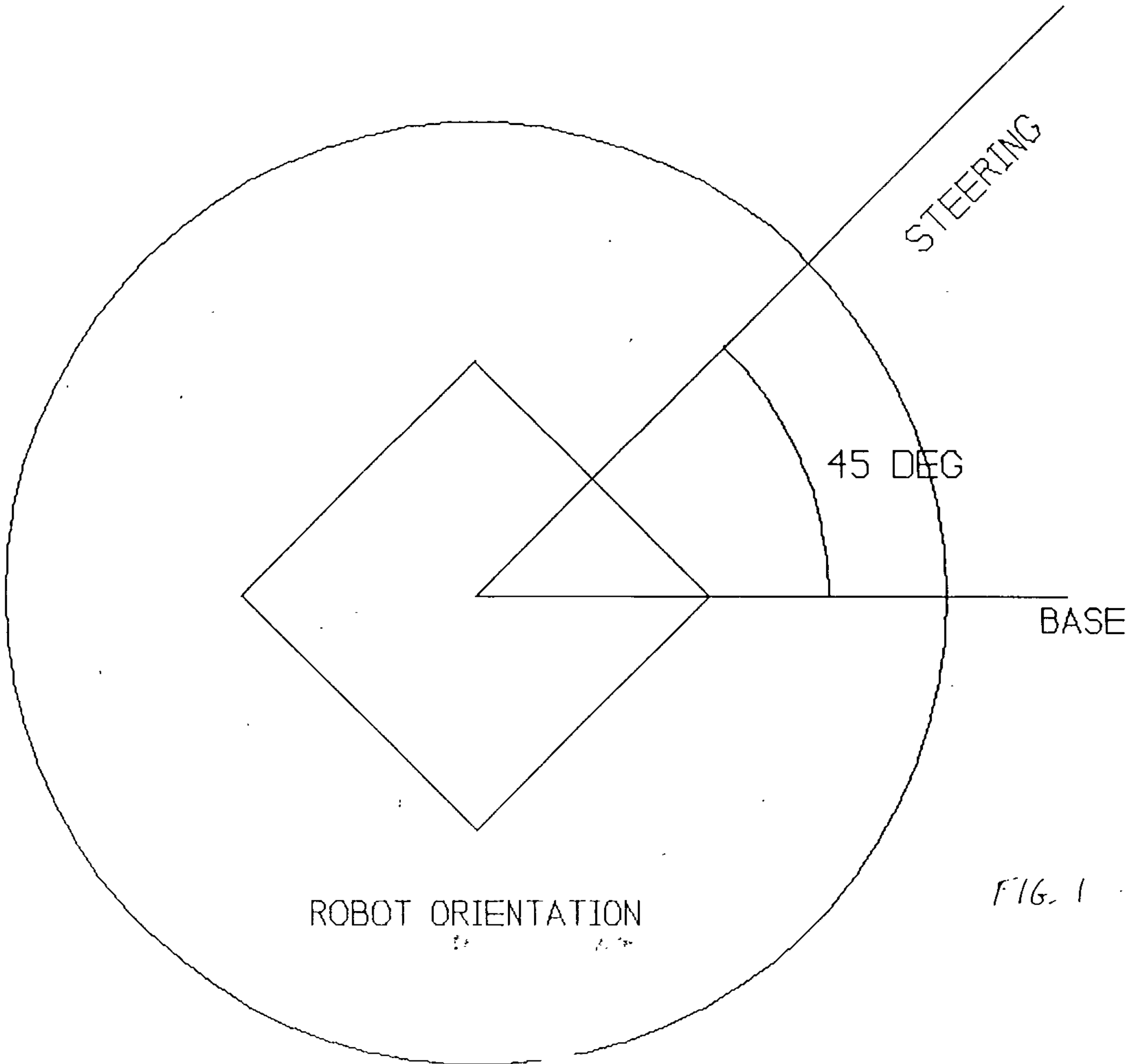


FIG. 1

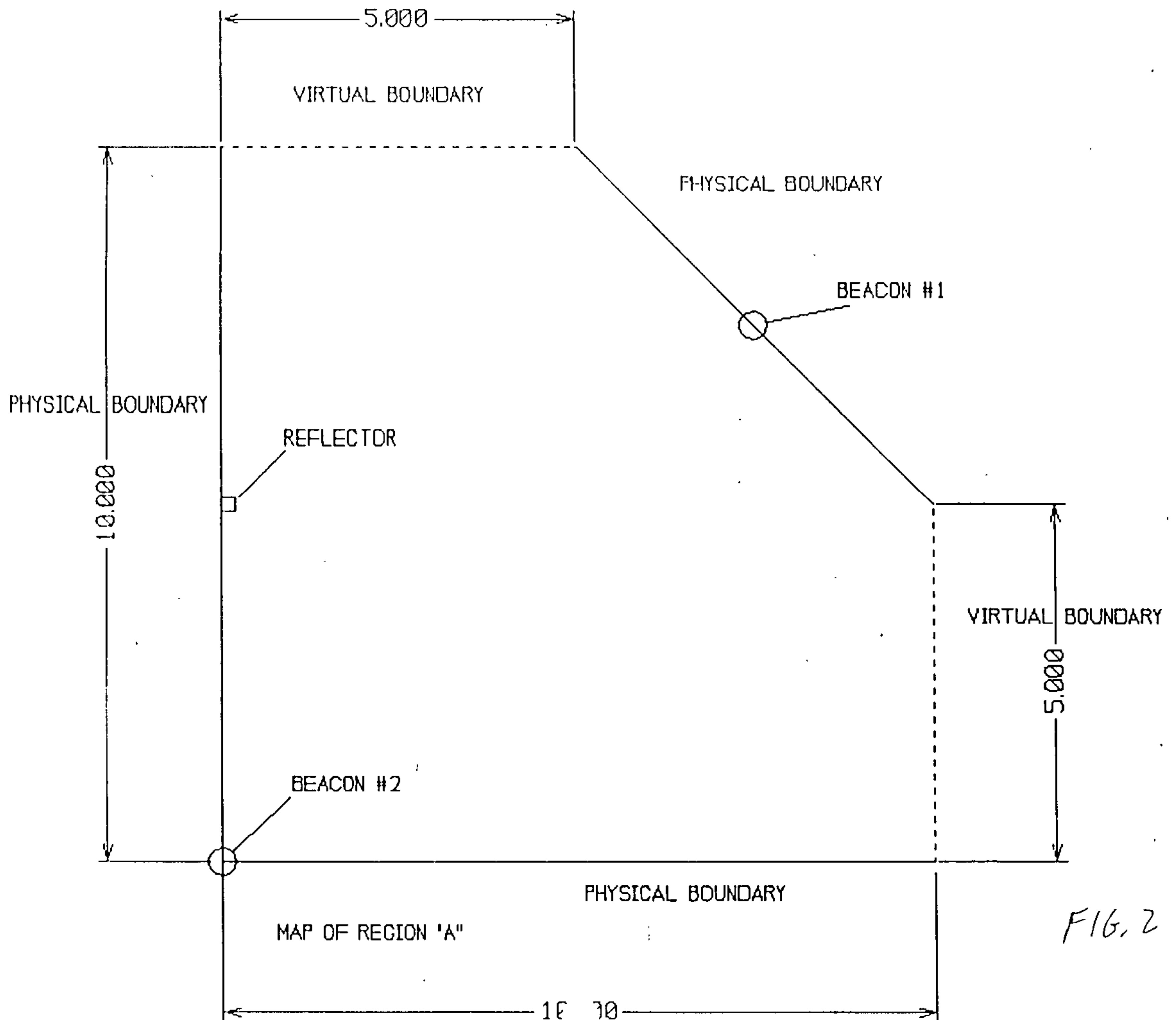


FIG. 2

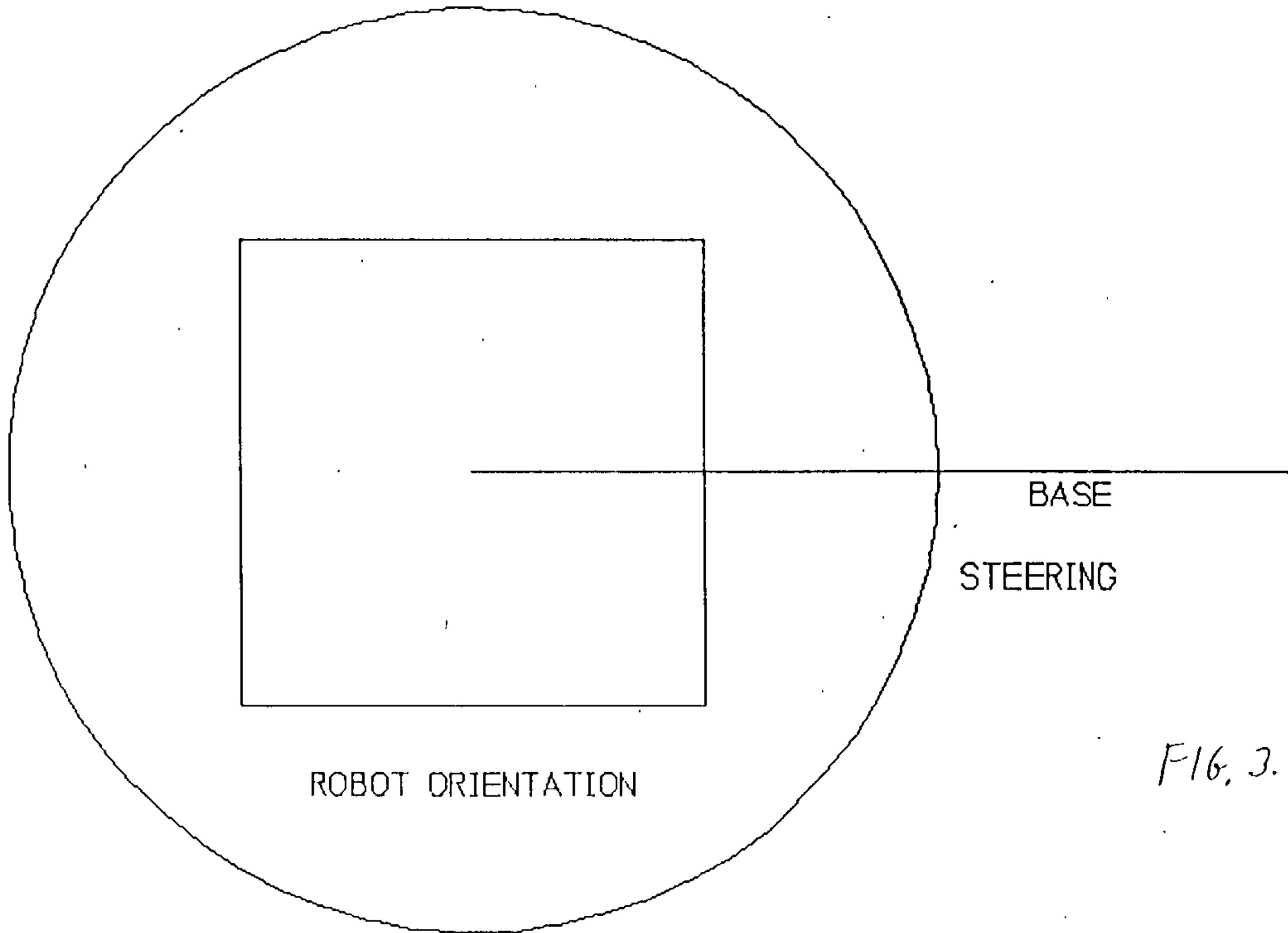


FIG. 3.

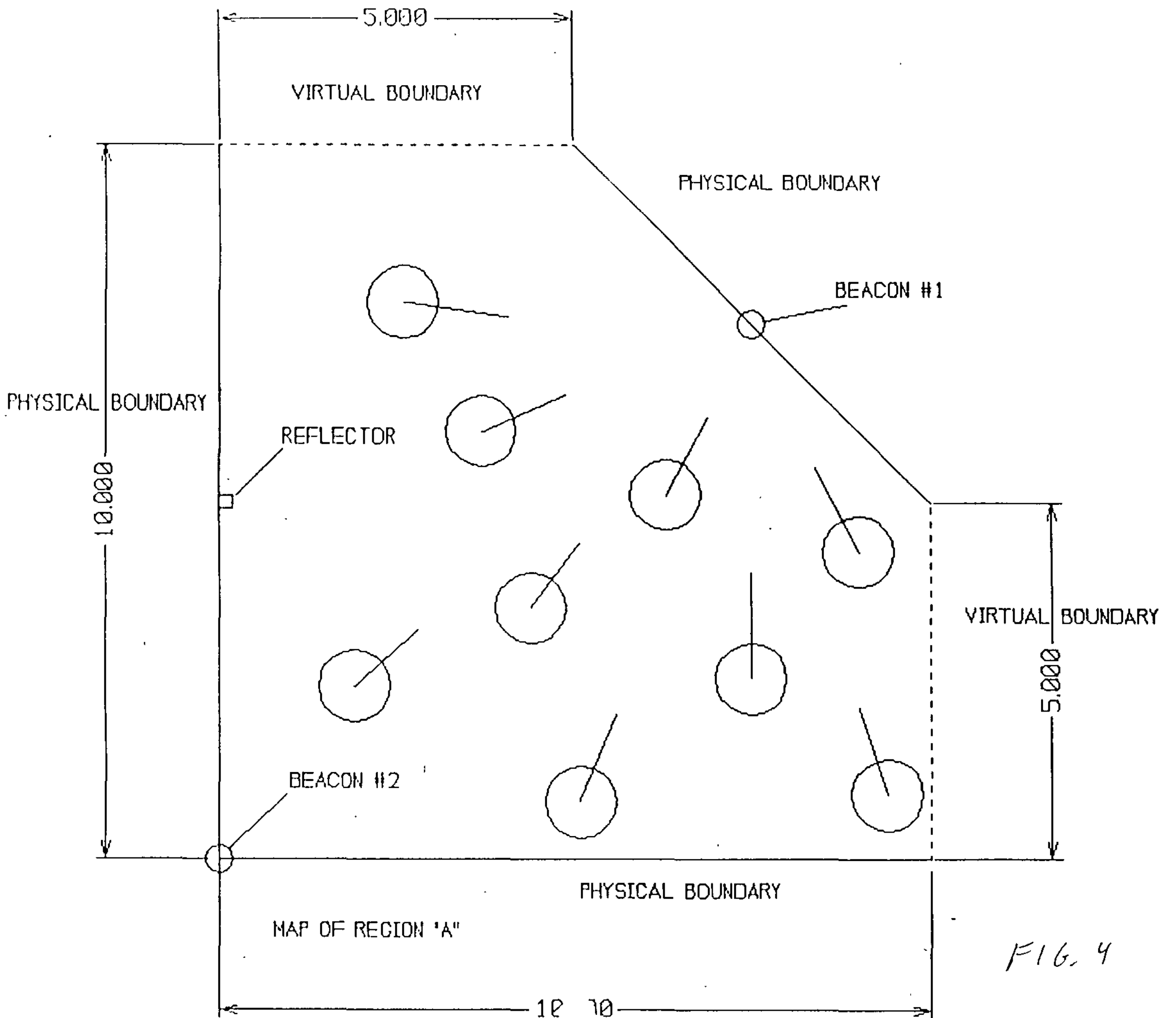


FIG. 4

207

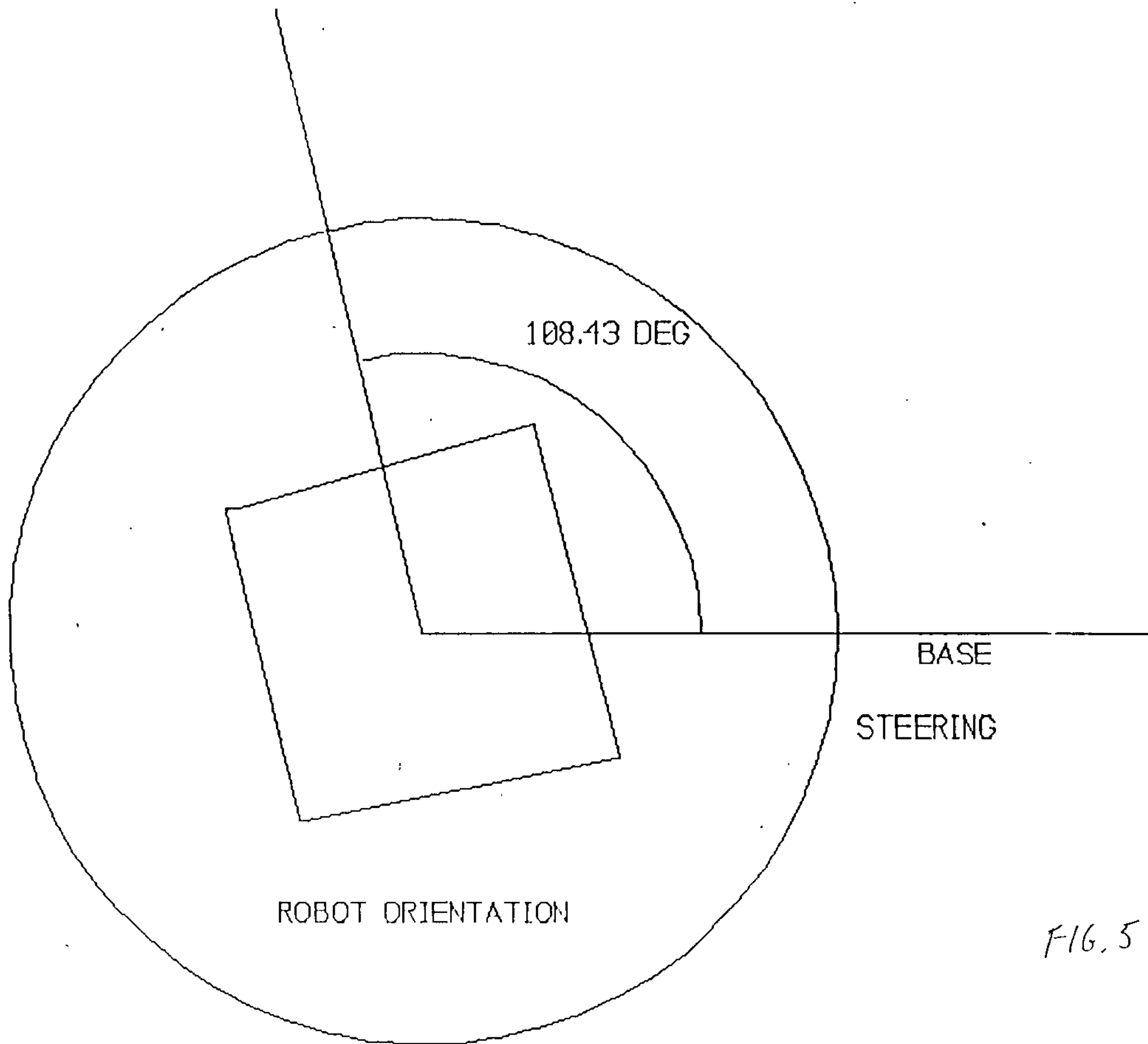


FIG. 5

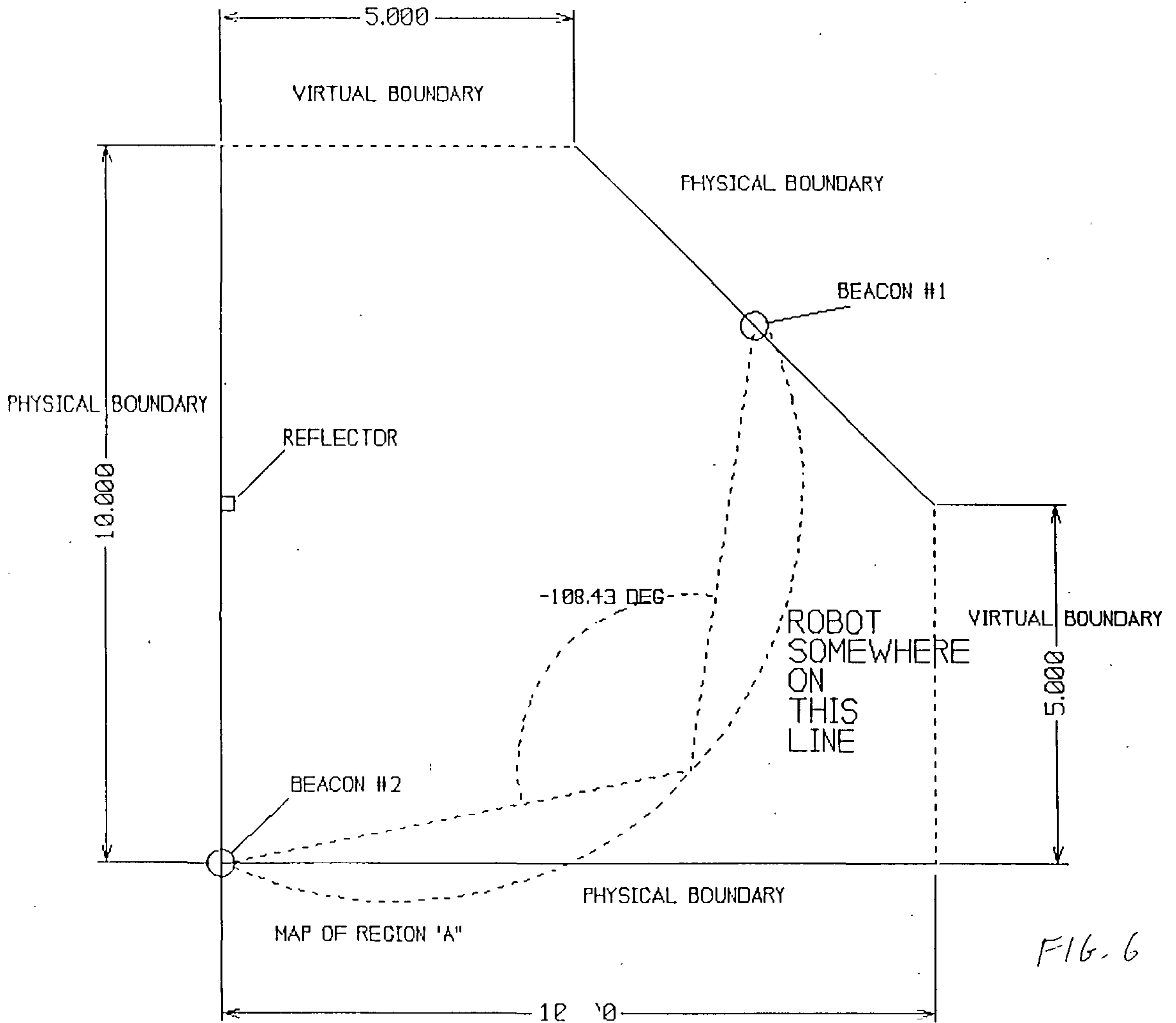


FIG. 6

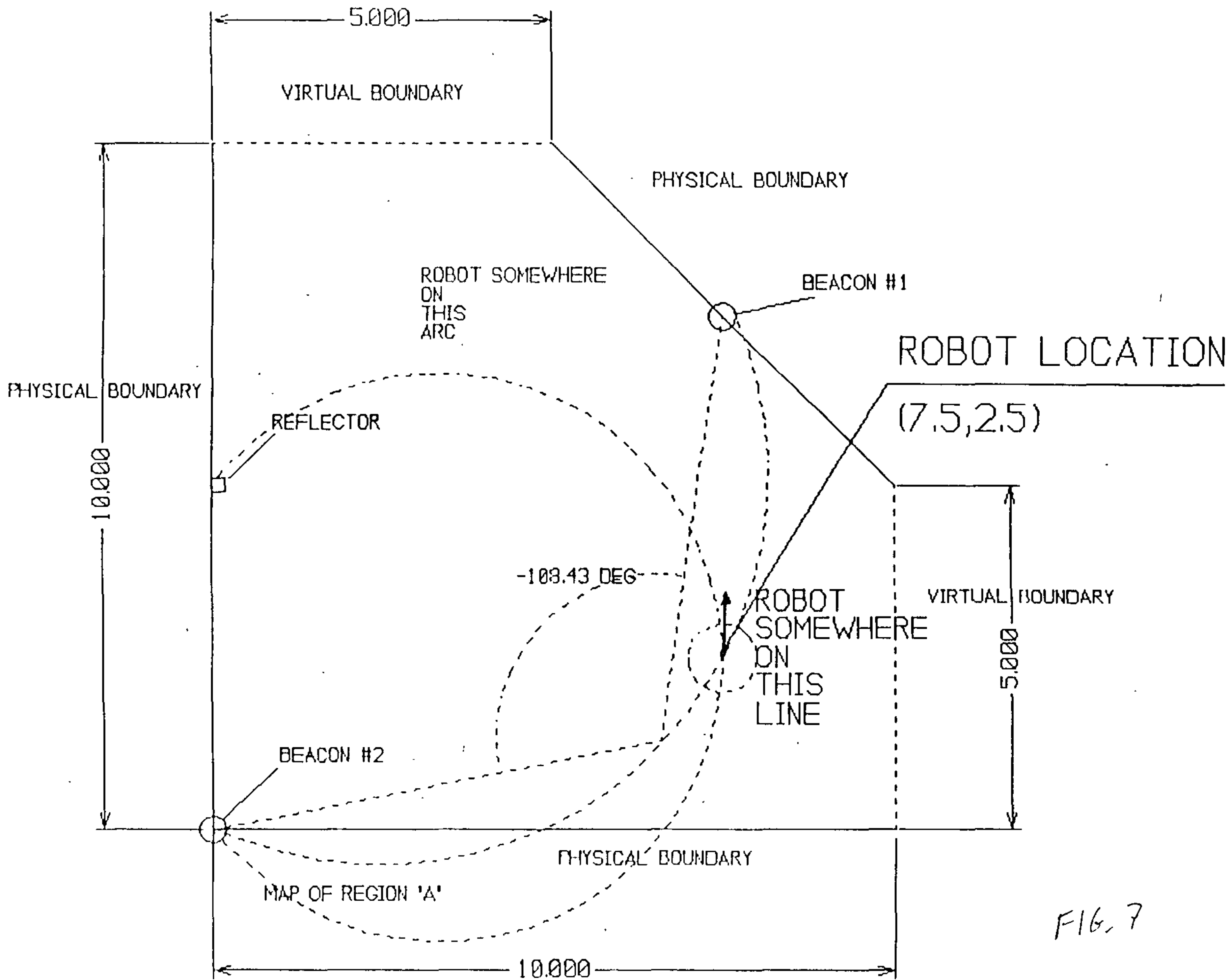


FIG. 7

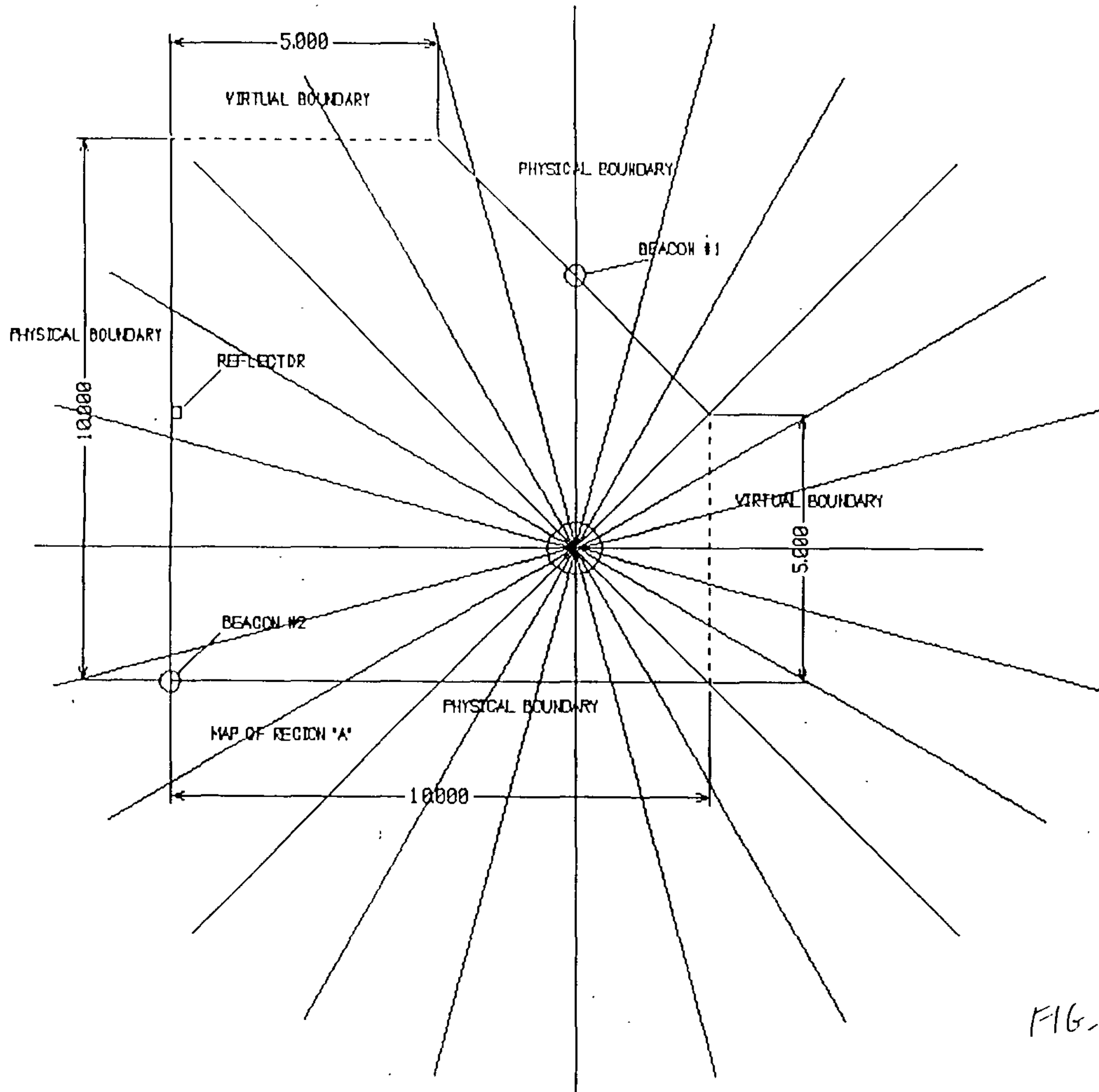


FIG. 8

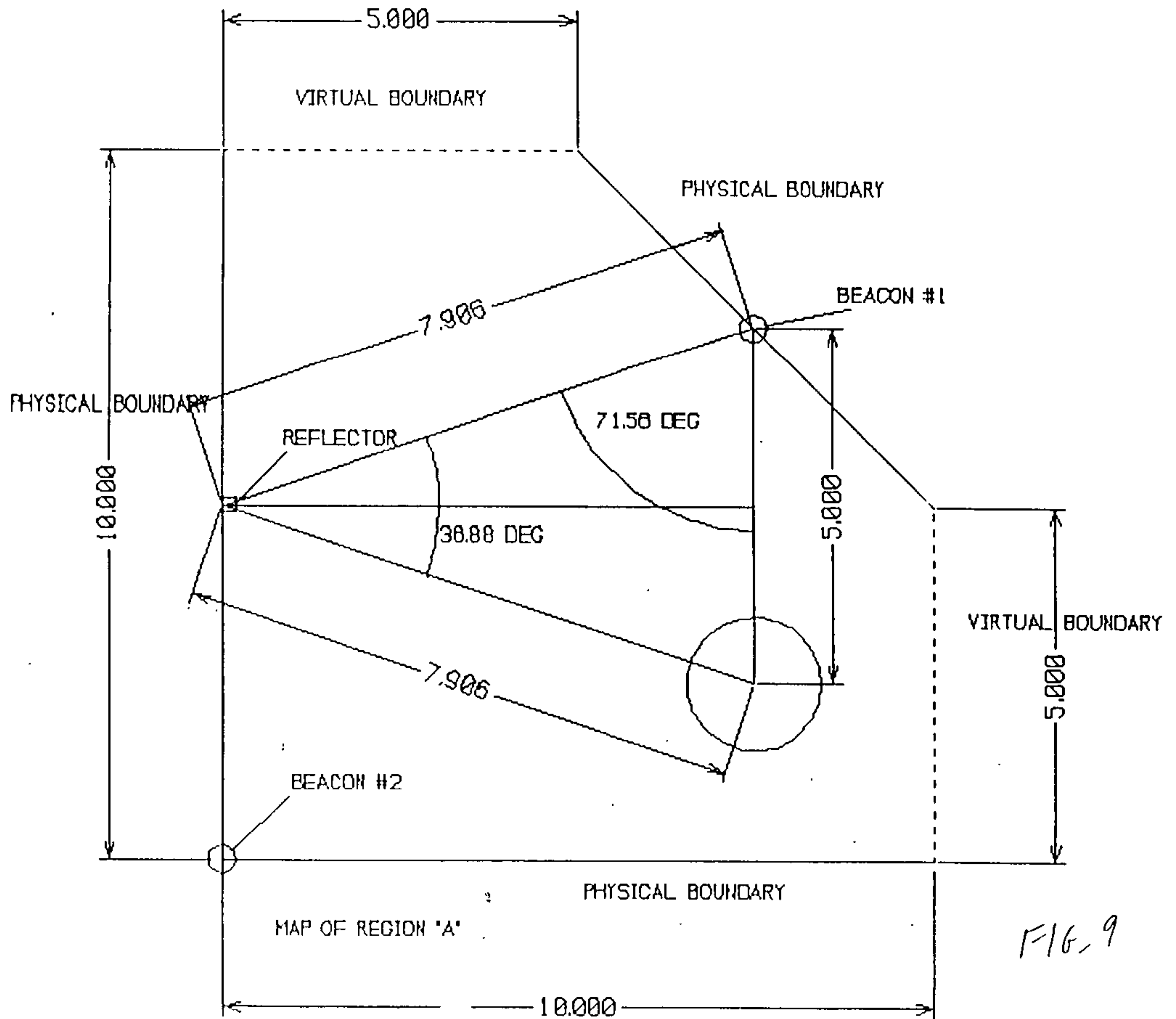


FIG. 9

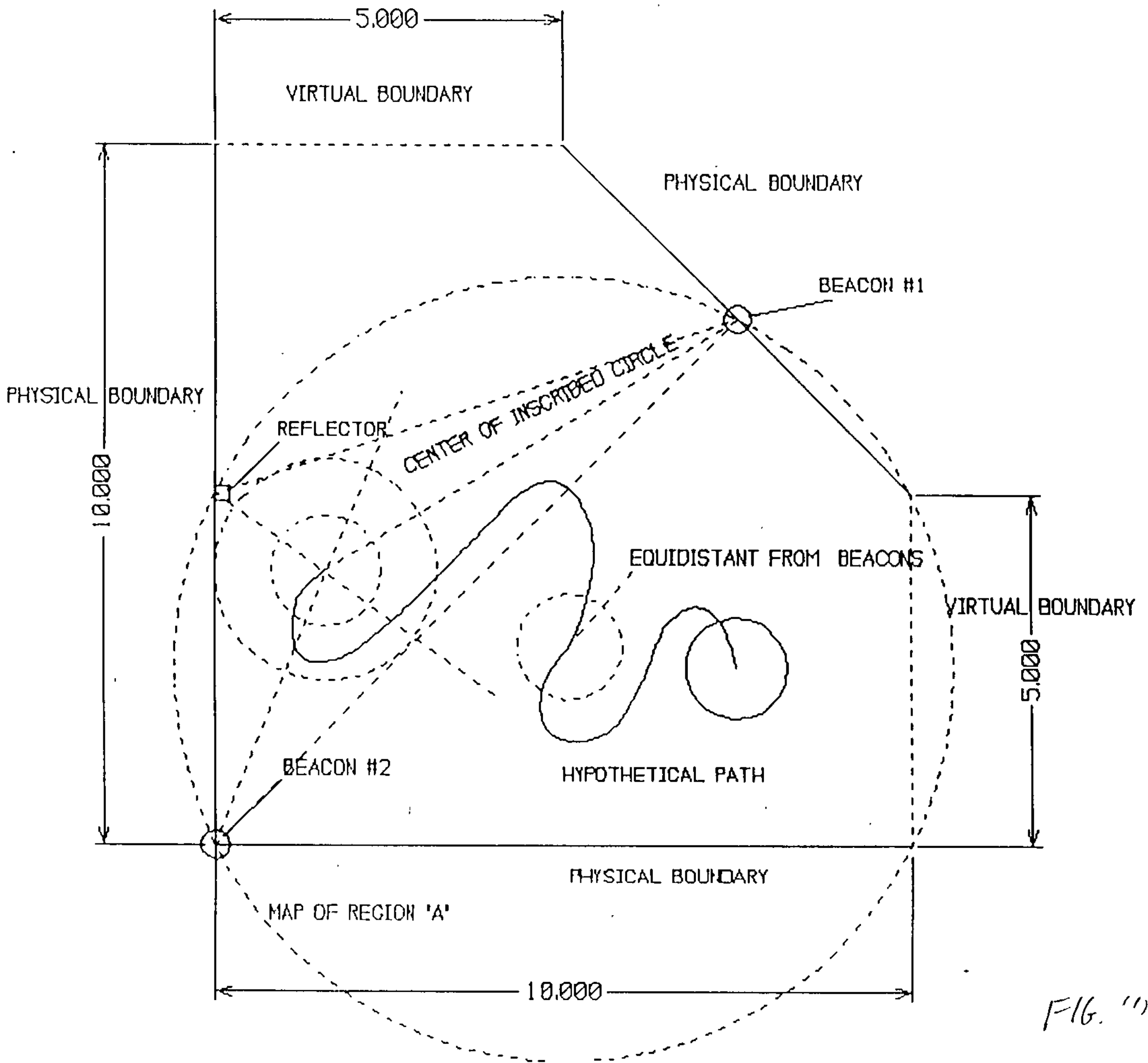


FIG. 11

Computers Math. Applic. Vol. 23, No. 2-5, pp. 51-73, 1992
Printed in Great Britain. All rights Reserved
Copyright©1992 Pergamon Press plc

CONCEPTUAL DEPENDENCY AND ITS DESCENDANTS

STEVEN L. LYTINEN
Artificial Intelligence Laboratory
The University of Michigan
Ann Arbor, MI 48109, U.S.A.

Abstract — This paper surveys representation and processing theories arising out of conceptual dependency theory. One of the primary characteristics of conceptual dependency was the notion of a canonical form, built out of a small number of primitive representations. Although the notion of primitives has largely been lost in subsequent work, many other of the basic notions of CD have remained. In particular, the idea of building representations around inferential capabilities has prevailed in this family of research. The result is a set of representational structures, all of which are highly knowledge-intensive. The use of these structures in various processing theories has led to knowledge-based theories of language understanding, planning, reasoning and other tasks, which have contrasted sharply with the traditional search-oriented approaches used in other systems.

1. INTRODUCTION

This paper will outline a family of representational theories which have developed from the common ancestor of conceptual dependency (CD) theory created by Roger Schank [1,2]. In addition, we will discuss some of the programs that these representations have been used in, as well as the reasons why these representations have proven to be useful for these programs.

The original goal of conceptual dependency work was to develop a "representation of the conceptual base that underlies all natural languages." [1, p. 554] This rather lofty goal resulted in the proposal of a small set of primitive actions (10–12 or so), and a set of dependencies which connected the primitive actions with each other and with their actors, objects, instruments, etc. The claim was that this small set of representational elements could be used to produce a *canonical form* representation for English sentences (and other natural languages).

Almost twenty years later, no one seems to take seriously the notion that such a small set of representational elements covers the conceptual base that underlies all natural languages. Representation theories that have descended from CD, such as scripts [3], have had a virtually unlimited number of vocabulary items. The notion of a "primitive" seems to have disappeared. However, as we shall see, CD has certainly influenced these subsequent theories. One of the questions that this paper will address, then, is: What did we learn from conceptual dependency? What does conceptual dependency theory have in common with the descendant theories? As we examine these descendants and the tasks that they have been used for, we will try to discover the common threads that connect this family of research.

We will begin the paper by briefly reviewing conceptual dependency theory. For a more thorough review, see [2]. Then we will discuss subsequent theories, including scripts, plan/goal representations, Memory Organization Packets (MOPs), and Thematic Organization Packets (TOPs). Finally, we will examine some of the processing theories that have accompanied these representations. In particular, we will focus on the tasks of language understanding and reasoning. As we will see, the basic assumptions behind the representation theories have a large effect on the processing theories that use them.

2. CONCEPTUAL DEPENDENCY

2.1. *A Brief Review of CD Notation*

Conceptual dependency theory was based on two assumptions:

1. If two sentences have the same meaning, they should be represented the same, regardless of the particular words used.
2. Information implicitly stated in the sentence should be represented explicitly. That is, any information which can be inferred from what is explicitly stated should be included in the representation.

These assumptions have many implications for what a representation language should look like. The first assumption implies that representations must be *general*; that is, they must capture the similarities in meanings of synonyms. For example, since "get" and "receive" can be used synonymously in many contexts, their conceptual representations in these contexts ought to reflect this fact by consisting of similar, if not identical, predicates.

The assumption that representations ought to explicitly represent implicit information means that inferences must be made in order to produce complete representations. If a sentence implies information without explicitly stating it, then in order to include that information in the representation, machinery must exist which can infer it from what is explicitly stated. This means that representations must support inference. In order to do this efficiently, they must be *canonical*. Whenever a particular inference can be made, it would be desirable if the same inference rule could always be used to make it. Without a canonical representation, several rules would be required, one for each different possible representation form.

Given these representational requirements, then, the vocabulary for conceptual dependency consisted of the following:

- a set of *primitives*, used to represent actions in the world
- a set of *states*, used to represent preconditions and results of actions
- a set of *dependencies*, or possible conceptual relationships which could exist between primitives, states, and the objects involved.

Representations of English sentences could be constructed by piecing together these building blocks to form a *conceptual dependency graph*.

PTRANS: The transfer of location of an object
ATRANS. The transfer of ownership, possession, or control of an object
MTRANS: The transfer of mental information between agents
MBUILD: The construction of a thought or of new information by an agent
ATTEND: The act of focusing attention of a sense organ toward an object
GRASP: The grasping of an object by an actor so that it may be manipulated
PROPEL. The application of a physical force to an object
MOVE: The movement of a bodypart of an agent by that agent
INGEST: The taking in of an object (food, air, water, etc.) by an animal
EXPEL: The expulsion of an object by an animal
SPEAK: The act of producing sound, including non-communicative sounds

Figure 1. The conceptual dependency primitives.

The set of primitives varied somewhat during the course of conceptual dependency theory, but it consisted of approximately 10-12 predicates, each of which represented a type of [illegible]on. The primitives are shown in Figure 1.

Each primitive had a set of *slots* associated with it, from the set of conceptual dependencies. Associated with each slot were restrictions as to what sorts of objects could appear in that slot. For example, the slots for PTRANS were the following:

ACTOR: a HUMAN (or animate object), that initiates the PTRANS
OBJECT: a PHYSICAL OBJECT, that is PTRANSed (moved)
FROM: a LOCATION, at which the PTRANS begins
TO: a LOCATION, at which the PTRANS ends.

In order to make explicit the information implicitly presented in the text, inference rules were written based on the primitives. For example, from the primitive PTRANS, the inference could be made that the OBJECT which was PTRANSed was initially in the FROM location, and after the PTRANS was in the TO location.

The need to make inferences in order to explicitly represent implicit information suggested a criterion for what made a good primitive: it should support a cluster of reliable inferences. Thus, PTRANS constituted a good primitive because all PTRANS's shared several common inferences: from the representation one could infer the prior location of an object being PTRANSed, the location subsequent to the PTRANS, that the source and destination of the PTRANS must be locations, etc. These same inferences could be made no matter what type of PTRANS: flying, driving, walking, falling, and so on.

Conceptual dependency representations were written graphically as shown in Figure 2. The actor of a primitive action was connected to the primitive using a double arrow; the object appeared to the right with a single arrow connection, and the source and destination (TO and FROM) appeared to the right of the object. Thus, the sentence, "John gave Mary a book" would be represented as shown in Figure 3. John was explicitly represented as both the ACTOR and FROM slots of this action, since it was assumed that John had control/possession of the book originally.

Although individual primitives grouped together a set of similar actions, it was important that distinctions between these actions could also be captured in CD notation. For example, reading and talking are both types of MTRANS's, but there are obvious differences between them: reading involves using one's eyes while talking involves using one's mouth and ears; the source of information in reading is an inanimate object (such as a newspaper) while the source in talking is a human, etc. In order to express differences between similar actions, several primitive actions could be connected to each other, using one or more of the conceptual dependencies. For example, the CD graph in Figure 4 represents the action of reading. The "i" link in this graph stands for an *instrumental* connection between the MTRANS and the MOVE.

Talking, on the other hand, might be represented as shown in Figure 5. Again, an instrumental connection expresses the relationship between the MTRANS and John's MOVEment of his mouth and Mary's ATTENDing with her ears.

Sometimes connections also involved explicit mention of one or more intermediate states or actions. Consider this example, from [3]:

John cried because Mary said she loved Bill.

The causal connection between John crying and Mary loving Bill is not direct: most readers infer that John loves Mary, realizes that Mary does not love him back since she loves Bill, and therefore is sad. This might be represented as shown in Figure 6. In this graph, "I" stands for an *initiate* link, connecting together an event and a mental state; and "I/R" stands for *initiate/reason*, a complex connection between two mental events.

Representations could be arbitrarily complex, sometimes making explicit a whole series of inferences that could be made. These complex sets of primitive actions, states, and conceptual dependencies were called *causal chains*. For example, consider the following sentence:

John went to Sears and found a TV for $100.

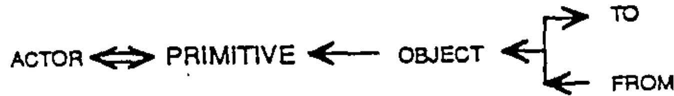


Figure 2. Basic form of a conceptual dependency graph.

Figure 3. Representation of "John gave Mary a book."

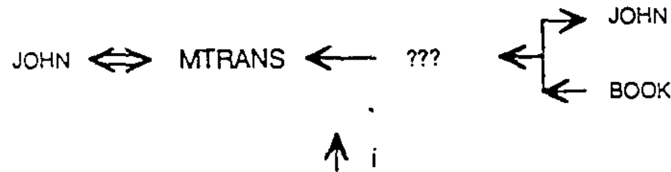


Figure 4. Representation of the act of reading.

Most readers infer that John bought the TV and paid $100 for it. This implies a rather complex chain of events, consisting of John going into the store, looking around, seeing a TV on sale for $100, taking it to the cashier, and paying $100 to the cashier. This causal chain might be represented as shown in Figure 7.

2.2. *Conceptual Dependency and Semantic Nets*

A distinction has been made in the literature between *content* and *structure* theories. In some ways, it is ironic that conceptual dependency and its derivatives are often grouped into the [illegible]ly of semantic net representations, because in general, semantic nets are a structure theory, w[illegible]as CD is a content theory.

The distinction between these two types of theories lies in their emphasis. Semantic net theory is about how knowledge should be organized: there will be nodes, with arcs connecting the nodes together. There is also some general (although, in the case of semantic nets, rarely formalized) notion of the structure's semantics; i.e., how to interpret a particular semantic net structure. Finally, there is usually the general notion of inheritence, etc. This is all structural information. That is, it says nothing about *what* will be represented, it simply says something about the (structural) form that the representation will take. Putting this another way, semantic nets tell us to use nodes and arcs; but they don't tell us what labels to put on the nodes, or what arcs to use and where. It is up to the user to decide these details. Without the details, however, semantic nets do not represent anything.

Conceptual dependency theory, on the other hand, was an attempt to enumerate the types of nodes and arcs which could be used to build representations. Rather than specifying the structure of representations, CD theory specified the *content*. True, the conventions used in drawing conceptual dependency graphs also specified structure, but this was not really the essence of the theory The essence was the primitives, and the *names* of the dependencies which could be used to hook primitives together.

This distinction can be made clearer if one imagines trying to implement conceptual dependencies and semantic nets in first order predicate calculus. In the case of conceptual dependencies, it is not hard to imagine: the primitive acts, dependencies, and states would specify a set of predicates to use when writing predicate calculus statements to represent sentences.[1] We might have

[1] The sort of inferences proposed by those who have used CD are not typically of an exclusively deductive nature, as would be true in the predicate calculus, but this really has more to do with the use of the representations, rather than the representations themselves.

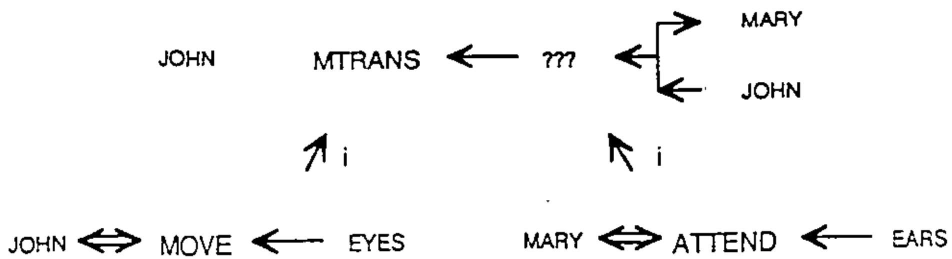


Figure 5. Representation of the act of talking.

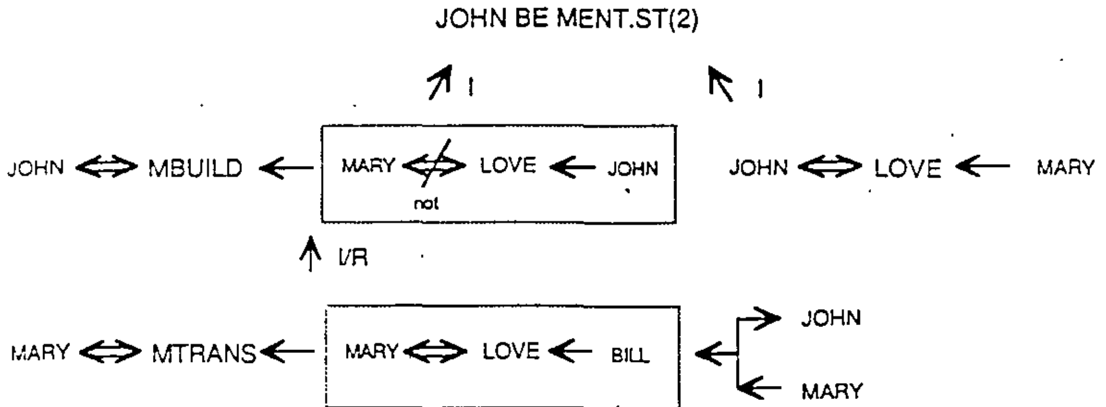


Figure 6. Representation of "John cried because Mary said she loved Bill."

to adopt some translation conventions in order to make all assertions first-order, but these would be quite straightforward. On the other hand, it does not make as much sense to "implement" semantic nets in predicate calculus. The two representations are in competition with each other: each provides a different syntax for distinguishing between predicates, arguments, and relations. There would be nothing left of semantic nets if we translated them to predicate calculus. Putting this another way, semantic nets would not add anything to first-order predicate calculus. This is in contrast to conceptual dependency, which adds the CD primitives as recommended predicates to be used.

2.3. Conceptual Dependency and Inferences

As we stated earlier, one motivation for representing the meaning of a text in a canonical form is to facilitate inferencing. Canonical form allows us to write inference rules as generally as possible: if representations did not always capture similarities in meaning, then rules about what can be inferred from a text would need to be duplicated, one rule required for every form of representation which had the same meaning relevant to the inference.

To illustrate that inferencing was facilitated by conceptual dependency, Rieger wrote the MEMORY program, which made inferences from text [4]. Inferences were used to build a "causal chain" to connect events in a story. For example:

> John hit Mary. Mary's mother took Mary to the hospital. Mary's mother called John's mother. John's mother spanked John.

MEMORY made several inferences from this story, including that John's mother spanked John because she was angry at him for hitting Mary, and that Mary went to the hospital because she was hurt.

Inferences were made in MEMORY by a set of rules, which were organized around inference categories. There were a total of 16 of these categories, some of which were:

1. Specification inferences, which filled in missing "slots" in a CD primitive, such as the ACTOR or INSTRUMENT of an action.

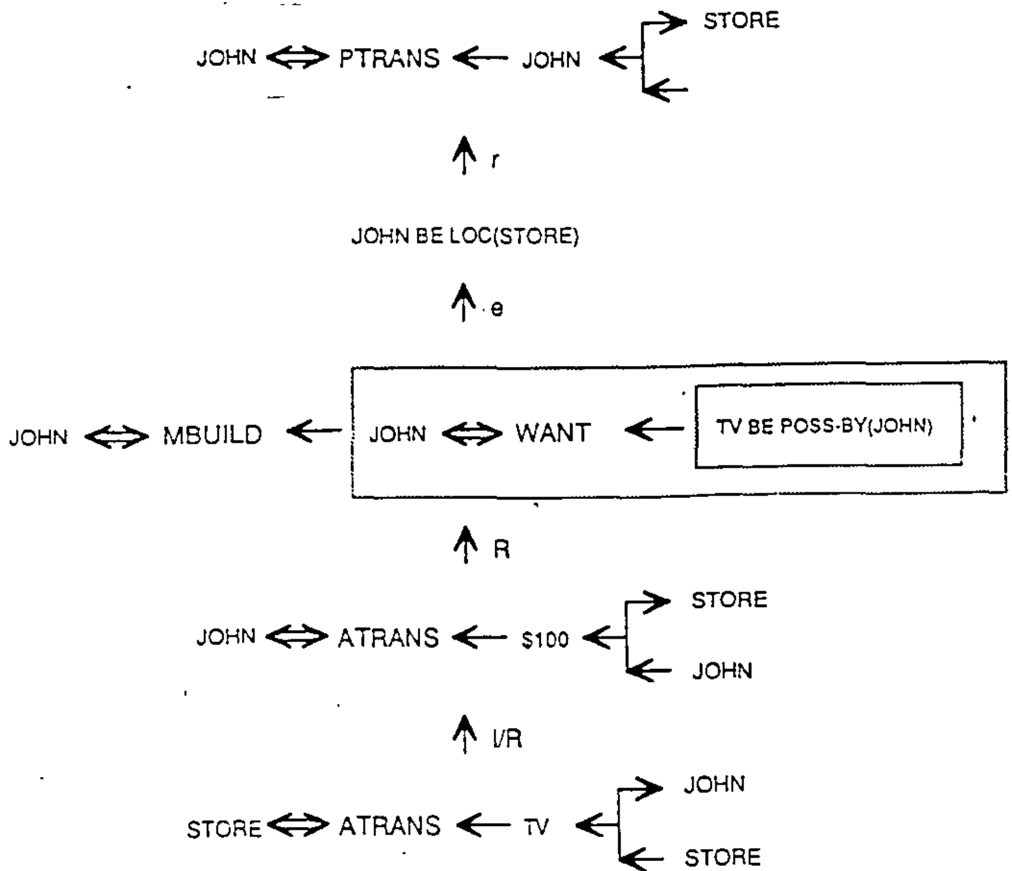


Figure 7. Representation of "John went to Sears and found a TV for $100."

2. Causative inferences, which hypothesized possible causes or preconditions of actions.

3. Resultative inferences, which inferred likely results of actions.

4. Function inferences, which inferred likely functions of objects.

Inference rules were applied in an *undirected* fashion. That is, when MEMORY read a sentence, its inference rules were automatically applied, without any particular goal in mind, such as building a causal chain to connect events together. This bridging often happened, but when it did it was fortuitous in some sense: inferences were applied to a new representation in an undirected fashion, sometimes resulting in the *confirmation* of another representation.

The undirectedness of MEMORY's inferences was meant to reflect the spontaneous nature of inferences that people make. These inferences seem uncontrollable for people: one cannot learn a new fact without inferring things about it. However, this undirected behavior led to problems: when processing a story, MEMORY did not know which inferences were most likely to lead to building a coherent causal chain to represent the story. This led to a combinatorial explosion in the number of inferences that the system had to consider in order to build causal chains. In some sense, Rieger's system was lacking common sense knowledge about what inferences were most likely to be relevant in a situation. For example:

John picked up the menu. He decided on the fish.

There are many conceivable inferences that can be made from someone picking up an object. People pick objects up for many reasons, as is illustrated by the following examples:

Use-as-instrument: John picked up the menu. He swatted a fly.

Subgoal-to-move: John picked up the menu. He found his fork.

Subgoal-to-read: John picked up the menu. He read it.

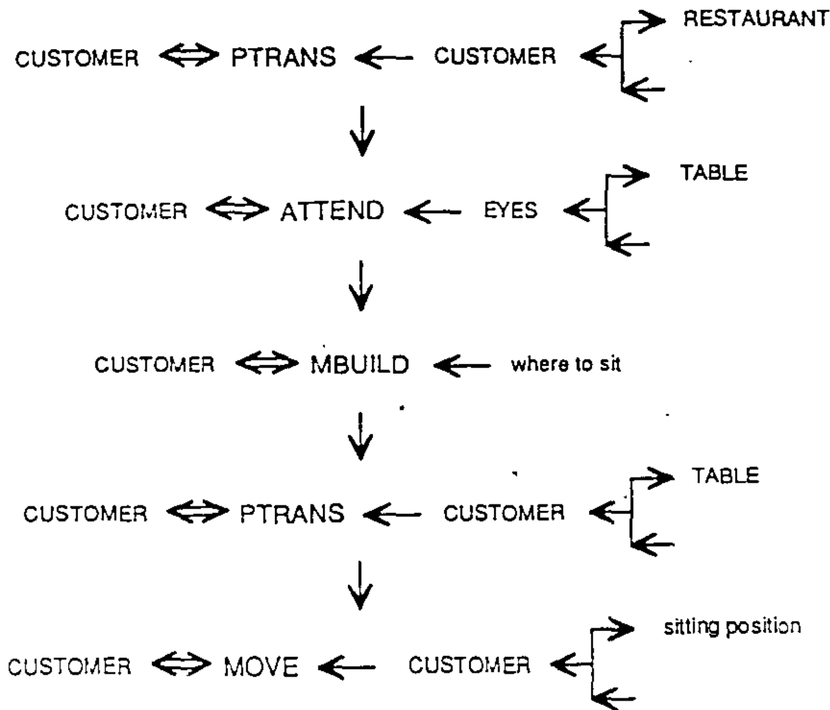


Figure 8. ENTER scene of the $RESTAURANT script.

These are certainly not the only possible reasons why John might pick up a menu. Yet, it is obvious what the most likely reason is why John picked up the menu. Usually people pick up a menu in order to read it, so that they can order food and eat. MEMORY, however, lacked this knowledge: there was no sense in which one inference was more common than others.

3. SCRIPTS AND MOPS

In order to solve the problems with undirected inferencing, larger knowledge structures, called *scripts*, were proposed [3]. Similar to frames [5] and schemas [6,7], scripts "precompiled" sets of likely inferences, packaged together so that they could be searched more efficiently than other, irrelevant inferences. Scripts represented stereotypical sequences of events, such as going to a restaurant. By using the script, the theory was that the understander had quick access to those events which always happen in a stereotypical event sequence, without having to think about other inferences which would most likely be irrelevant.

A script consisted of a set of *roles*, or participants involved in the script as well as common objects used; and *scenes*, each of which described the typical events in one portion of the script. For example, in the $RESTAURANT[2] script, some of the roles were the customer, the waiter/waitress, the restaurant, and the food. Scenes included ENTER, ORDER, EAT, PAY, and LEAVE. The details of each scene were represented as a sequence of conceptual dependency representations. For example, the ENTER scene in $RESTAURANT consisted of the causal chain in Figure 8.

In order to account for the variability of stereotypical events, scripts often contained different *tracks*. Each track reflected one possible alternative sequence that could be followed in the script. For example, the scenes of $RESTAURANT do not always take place in the same order: in a sit-down restaurant, usually one sits at a table, orders, then eats, then pays. However, in fast food restaurants, paying precedes sitting, which is then followed by eating. Other variations take place in other specific restaurants or types of restaurants. The complete $RESTAURANT, then, is shown in Figure 9.

[2]Schank and Abelson preceded script names with a $.

Again, one of the points of proposing scripts was to facilitate common inferences that took place in the events that they represented. Consider our earlier example:

John picked up the menu. He decided on the fish.

If $RESTAURANT is inferred from the first sentence, then it is not difficult to understand how the second sentence connects to it. Making the connecting inferences is simply a matter of matching the second event to some other event in $RESTAURANT, and picking out the precompiled causal chain from the script which connects the two events.

In addition to being a powerful processing structure, scripts were also proposed as a memory structure. Since a script contains all of the standard inferences which are made about a stereotypical event sequence, the theory was that there should be no need to duplicate these inferences in the record of a particular episode. This predicted that various confusions should occur: confusions about which particular scenes in a scriptal episode were mentioned in a story should occur, and memories across episodes using the same script might also get confused.

Bower, Black and Turner [8] conducted a study in which they found that recall confusions did in fact occur. Subjects were shown stories about trips to the doctor's and dentist's office. Scenes, such as waiting in the waiting room, being examined, and checking out, were either included in or omitted from the stories. During subsequent testing, subjects had difficulties recalling which scenes were explicitly mentioned. They also could not recall which particular scenes went together in the same episode. This seemed to provide evidence for scripts or some sort of schema-based memory of the stories.

3.1. Scripts and Generality

Charniak [9] observed that, for efficiency reasons, causal knowledge in scripts or frames should not be duplicated when that knowledge comes from more general causal laws. Thus, in his frame-style representational system, which represented mundane knowledge about painting, he distinguished between two types of frames: *simple events*, which corresponded to common sense causal laws; and *complex events*, which referred to the simple events for their causal explanations. The PAINTING frame was a complex event, which consisted of sequences of actions such as "get paint on the painting instrument," and "bring instrument in contact with object," along with pointers to simple events, like STICK (i.e., "sticks to"), which provided the causal rules explaining why events within the PAINTING frame proceeded in that order. STICK consisted of a causal rule, explaining why bringing the painting instrument in contact with the object to be painted would cause paint to stick on the object. These simple events, like STICK, could be shared between many complex events, like PAINTING, so that knowledge common to more than one situation would not have to be duplicated in the frames used to represent those situations.

For different reasons, a similar sharing of knowledge was proposed in [10], as a modification of script theory. Schank discussed the use of scripts for learning. This new task raised some problems with scripts. In script theory, although similar scenes appeared in different scripts, the representation of these scenes did not capture these similarities. For instance, the scripts $RESTAURANT and $DEPARTMENT-STORE both contained a scene having to do with ordering. In the case of $RESTAURANT, this scene was ORDER-FOOD, and in $DEPARTMENT-STORE, the scene was ORDER-MERCHANDISE. However, these two scenes were totally separate entities, with no representation of the fact that they were similar in very important ways. There was no more general concept ORDER to which they could refer which was an abstraction of the common entities of both scenes. This presented a problem for learning. If a program were to learn scripts like $RESTAURANT and $DEPARTMENT-STORE, knowledge common to the ordering in a restaurant and the ordering in a department store would have to be stored in two different places, since the corresponding scenes in $RESTAURANT and $DEPARTMENT-STORE did not share information. This meant that knowledge learned in one domain could not be accessed in the other domain. So, for example, if one learned that one should be polite in order to get good service in a restaurant, that information could not be accessed by $DEPARTMENT-STORE in order to realize that one should be polite to the catalog clerks in order to get good service at a department store, also.

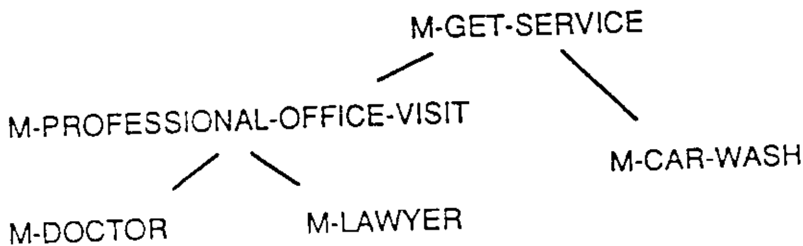
Script representation

$DOCTOR:
HAVE-MEDICAL-PROBLEM + MAKE-APPT + GO +
DOCTOR-WAITING-ROOM + TREATMENT + PAY

$LAWYER:
HAVE-LEGAL-PROBLEM + MAKE-APPT + GO +
LAWYER-WAITING-ROOM + LEGAL-CONSULTATION + PAY

$CAR-WASH:
HAVE-DIRTY-CAR + GO + WAIT-IN-LINE +
GET-CAR-WASHED + PAY

MOP representation



M-GET-SERVICE:
(NEED-SERVICE) + GO + (WAIT) + (GET-SERVICE) + PAY

M-PROFESSIONAL-OFFICE-VISIT:
HAVE-PROBELM + MAKE-APPT + (GO) + WAITING-ROOM +
(GET-SERVICE) + (PAY)

M-CAR-WASH:
HAVE-DIRTY-CAR + (GO) + WAIT-IN-LINE + GET-CAR-WASHED + (PAY)

M-DOCTOR:
HAVE-MEDICAL-PROBLEM + (MAKE-APPT) + (GO) +
DOCTOR-WAITING-ROOM + TREATMENT + (PAY)

M-LAWYER:
HAVE-LEGAL-PROBLEM + (MAKE-APPT) + (GO) +
LAWYER-WAITING-ROOM + LEGAL-CONSULTATION + (PAY)

Figure 10. Script vs. MOP representation of various events.

There was another problem with scripts, etc., which became evident from the Bower *et al.* experiment. In addition to the within-script memory confusions discussed earlier, recognition confusions were also found to occur between stories about visits to the dentist and visits to the doctor. Intuitively, this result was not surprising, since most people have experienced such confusions. But how could they be explained by scripts? Should we posit a "visit to a health care professional" script to explain it? Clearly, this would be beyond the initial conception of what a script was.

To accommodate solutions to these problems, a modification of script theory was proposed in [10] that introduced a new processing structure, called a MOP (Memory Organization Packet). The general idea behind MOPs was to store knowledge which is common to many different situations in only one processing structure, and then to make this processing structure available in all the different situations in which it applies.

To see how MOPs differ from scripts, let us compare the script and MOP representations of several events as presented in Figure 10. In script theory, all the scenes of the doctor, lawyer, or car wash episode were provided by one structure, $DOCTOR, $LAWYER, or $CAR-WASH. However, although similar scripts had many similar scenes, such as HAVE-MEDICAL-PROBLEM and HAVE-LEGAL-PROBLEM in $DOCTOR and $LAWYER, there was no connection between such scenes. In MOP theory, however, all the common elements shared among specific scenes of different contexts are abstracted together into a more general scene. Thus, the common features of doctor visits and lawyer visits are abstracted into a more general structure, M-PROFESSIONAL-OFFICE-VISIT (or M-POV for short). M-POV has scenes, WAITING-ROOM, GET-SERVICE, etc., which are abstractions of the scenes DOCTOR-WAITING-ROOM, LAWYER-WAITING-ROOM, and GET-TREATMENT, LEGAL-CONSULTATION. Then, features unique to doctors' offices are provided by the more specific scene DOCTOR-WAITING-ROOM, but features shared by other professional office visits exist in the generalized scene WAITING-ROOM. Similarly, the sequential information shared by these scripts is abstracted together also into M-POV. Thus, a great deal of information that was in $DOCTOR in script theory is not in M-DOCTOR in MOP theory. Rather, much of this information comes from more general MOPs.

Similarly, information which is shared between professional office visits and other types of getting service, such as getting your car washed, is abstracted even higher, into M-GET-SERVICE. All get-service actions have things in common, such as first having a problem, waiting for the service, and paying. This common information is abstracted to as general a structure as possible, into even more general scenes such as NEED-SERVICE and WAIT.

MOPs and scenes, then, are arranged hierarchically. M-POV is a generalization of the common elements in M-DOCTOR and M-LAWYER, M-GET-SERVICE is a generalization of M-POV and other types of service MOPs, WAITING ROOM is a generalization of DOCTOR-WAITING-ROOM and LAWYER-WAITING-ROOM, WAIT is a generalization of WAITING-ROOM and WAIT-IN-LINE, etc. Knowledge is stored at as general a level as is possible.

4. PLANS, GOALS, AND TOPS

Another line of work which proceeded from conceptual dependency was the notion of high-level, goal-based representations. Since CD primitives were mainly about physical actions, they did not seem to capture well the concepts useful to express knowledge about goal-based behavior. For example:

> Willa was hungry. She picked up the Michelin guide.

The causal chain which connects these events is shown in Figure 11. Given the complexity of this causal chain, again it seems necessary for a program to have some heuristics about how to search the space of possible inferences in order to construct it. However, this time, it does not seem likely that a script would be available to guide the process. Somehow, $LOOKUP-RESTAURANT-IN-GUIDE seems to be too uncommon an event to warrant a script.

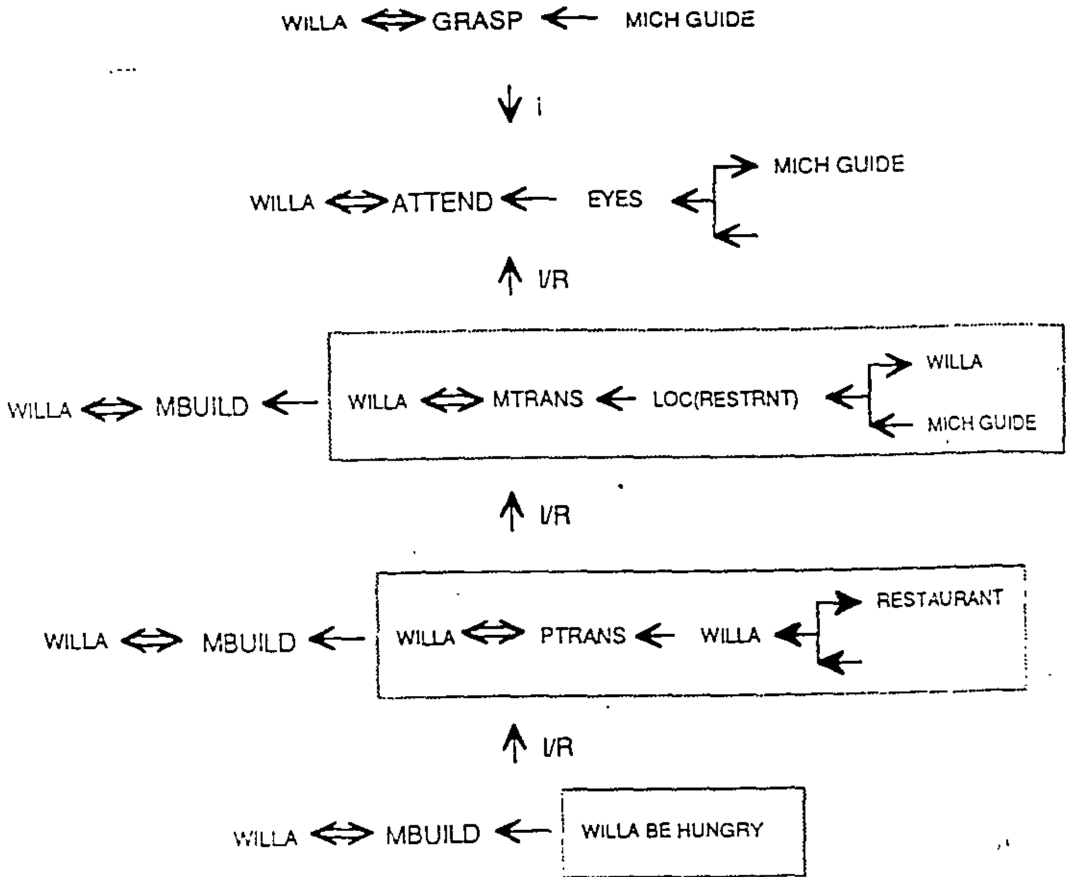


Figure 11. Representation of "Willa was hungry. She picked up the Michelin guide."

To account for the ability to infer complex causal chains which represented plans constructed by an agent in order to solve a problem, Schank and Abelson proposed representations for *goals* and *plans*. Several categories of goals were proposed, including:

S-goals: *Satisfaction* goals which recur regularly, such as S-HUNGER, S-SEX, and S-SLEEP

E-goals: *Enjoyment* goals, such as E-TRAVEL, E-EXERCISE

A-goals: *Achievement* goals, such as A-SKILL or A-POSSESSIONS

I-goals: *Instrumental* goals, or subgoals which arise during pursuit of other goals

D-goals: *Delta* goals, similar to I-goals, but with standard plans associated with them.

Just as with CD primitives, goals were categorized because of inferences that could be made based on the category. For example, D-goals, which included D-KNOW (the goal to know a particular fact), D-CONT (the goal of getting control of an object), and D-PROX (the goal of moving to a location), were associated with a common *planbox*, or set of plans that could be used to achieve them. The D-goal planbox contained plans such as ASK, THREATEN, and INVOKE-THEME (persuading someone to help because of a thematic relationship, such as parent-child). Each plan could be used to satisfy any D-goal in the right circumstances.

Schank and Abelson also proposed *named plans*, which were schematic structures, but at a much more general level than scripts. Named plans consisted of sequences of planning events which led to the satisfaction of a goal. An example is the named plan USE, which consisted of the following steps:

USE(x) = D-KNOW + D-PROX + D-CONT + I-PREP + DO

For example, the goal of using a book would be realized by first knowing where it is, moving to it, achieving control of it (by GRASPing it, for example), preparing to use it in some way (for example, one might have to put on reading glasses), and finally reading the book.

Just as with scripts, this knowledge of common sequential actions could be used to facilitate story understanding. In our earlier example:

> Willa was hungry. She picked up the Michelin guide.

The connection between the events is made as follows: Willa has the goal S-HUNGER. She selects the $RESTAURANT script to satisfy this goal, but before she can execute USE(RESTAURANT), she must KNOW(LOC(RESTAURANT)) (from the named plan). This generates the steps of the plan as subgoals, including the subgoal D-KNOW, which in this case will be achieved by an MTRANS from the Michelin guide. Again, to USE(Michelin-guide), the named plan suggests several steps, one of which is D-CONT, which Willa achieves through a GRASP ("picked up").

4.1. *More Complex Reasoning about Goals*

Schank and Abelson's work provided the vocabulary needed to do elementary reasoning about single-goal situations. Wilensky extended this work to more complex goal situations, involving goal interactions and competitions [11]. Wilensky proposed rules for plan recognition based on goal configurations, as well as key characteristics such as resource limitations, recurrence of goals, etc. For example:

> John wanted to watch the football game. Mary wanted to watch the Bolshoi ballet.
> John got the old black-and-white TV out of the attic.

Wilensky proposed the following rule for understanding this story:

> If a set of characters have goals that compete due to a shortage of a non-consumable functional object that the characters need to use simultaneously, and one character tries to acquire additional functional objects of the same kind, then that action is part of a plan to ease the goal competition. [11, p. 204]

In both Schank and Abelson's work and Wilensky's extensions, goal-level representations were quite separate from the content level, such as CD primitives or scripts. However, it seems that often it is the case that content can have an effect on the goal/plan level. Particular facts about the context in which a goal arises can play a key role in determining the particular plan to select in order to achieve the goal.

Evidence for rather specific structures which combine goals/plans and contextual information can also be seen in remindings which people have about these situations. Schank [10] presents a large number of these remindings. For example:

> X was talking about how there was no marijuana around for a month or two. Then, all of a sudden, everyone was able to get as much as they wanted. But the price had gone up 25 percent. This reminded X of the oil situation that cleared up as soon as the price had risen a significant amount. [10, p. 120]

The two situations in the example have many things in common. First, they are about D-CONT goals (possession of marijuana and gasoline). The D-CONT goals are recurrent, since both desired objects are used up regularly and must be bought over and over. They also involve a situation in which the selected plan is BARGAIN-OBJECT (i.e., exchange money for the desired object), but the object is in short supply. Also, perhaps a common feature is that the desired object is controlled by unethical people (drug pushers, and the OPEC cartel). If we view this as a planning structure, the structure provides a rule something like, "If you want to D-CONT an object using money, the goal recurs regularly, the object is in short supply, and you're dealing

with unethical people, then try waiting a little while, and you'll be able to buy it then, but at a higher price."

Note the difference between this thematic structure (or TOP, as Schank called them) and the goal/planning structures we discussed earlier. The TOP has goal features, such as D-CONT and BARGAIN-OBJECT. There are also very specific contextual features, such as the object being a usable commodity and in short supply; and a very specific suggested plan, to wait a while and buy at a higher price.

Schank gives many examples of TOPs, based on evidence of cross-contextual reminding such as the example above. All of them suggest a combination of goal/plan information and arbitrary sets of conditions, specifying the context in which the structure applies. Each TOP can then provide a specific prediction or set of predictions about what is likely to occur in the situation, or what the planner can do to achieve his/her goal.

Seifert and her colleagues found evidence for thematic remindings in human subjects, providing psychological evidence for thematic structures such as TOPs. In [12], subjects were presented with sets of stories which included TOP-like similarities. Adages were selected as thematic patterns that would be familiar to subjects and culturally shared, minimizing variability in the structures used for encoding by subjects. For example, the adage "closing the barn door after the horse is gone" can be characterized as a planning failure where X knows a plan to prevent goal failure, but delays execution to avoid the cost until the goal is failing; then, the plan is executed, but fails because it is not a recovery plan but a prevention plan [13]. Here are two stories used in the experiments that represent this thematic pattern:

> Story 1: Academia
>
> Dr. Popoff knew that his graduate student Mike was unhappy with the research facilities available in his department. Mike had requested new equipment on several occasions, but Dr. Popoff always denied Mike's requests. One day, Dr. Popoff found out that Mike had been accepted to study at a rival university. Not wanting to lose a good student, Dr. Popoff hurriedly offered Mike lots of new research equipment. But by then, Mike had already decided to transfer.
>
> Story 2. Wedding Bells
>
> Phil was in love with his secretary and was well aware that she wanted to marry him However, Phil was afraid of responsibility, so he kept dating others and made up excuses to postpone the wedding. Finally, his secretary got fed up, began dating, and fell in love with an accountant. When Phil found out, he went to her and proposed marriage, showing her the ring he had bought. But by that time, his secretary was already planning her honeymoon with the accountant.

Two stories were presented in sequence, that either shared the same theme or did not, followed by a test list of items to respond to. Since both stories should be encoded with the same theme (Thematic Abstract Unit, from [13]), understanding the same-theme stories was predicted to result in a connected memory representation. If such a characteristic reminding structure is set up during the understanding process, will reference to the academia story result in activation of the wedding-bells story? To measure activation, subjects responded to a series of test sentences about the two stories, including negatives ("Mike decided to buy his own research equipment") and the two items of interest:

> by then, Mike had already decided to transfer
>
> his secretary fell in love with an accountant

If the two stories are connected in memory, responding to an item from one should speed the time to respond to an item from the other. This same-theme response time was compared to one where the preceding item refers to a story that did not share the same theme, and therefore should have no connection in memory between the two stories. For example, if one story was based on the "pot calling the kettle black" theme, and the other on the barn door theme, no priming of the response would be expected.

The results showed that the expected same theme result did not significantly affect response time to the test items. The shared knowledge structure did not affect the ease of access of the episodes in memory, so the strong hypothesis that connection alone would result in automatic activation of related episodes was not supported. However, if the process is not automatic, it may still occur under strategic conditions, when subjects attempt to utilize effort or take cues from the task specifications.

In a second experiment, subjects were instructed to think about the theme as they read, and to rate the similarity of the story pair after the test list. Otherwise, this experiment was identical to the first. However, the results were different: this time, there was a significant effect of thematic similarity: mean response time for stories sharing the same theme was significantly faster than for different themes. This indicates that responses were faster for test items when the story pair shared the same thematic organizing structure.

The only difference between the two experiments was the instruction to attend to similarity. It seems, then, that accessing the same schema does not automatically connect two episodes in memory; instead, some strategic purpose is necessary to promote activation of prior episodes. Note that the comparison alone is not the cause of the effect, since both conditions required comparing story similarity. When the task includes a strategic purpose for the reminding, such as in a straightforward memory test for access to prior thematically-related exemplars, robust evidence for thematic reminding has been found.

5. PROCESSING THEORIES

We have examined many types of representations that have grown out of conceptual dependency, including scripts, plans, goals, MOPs, and TOPs. The point of all these representations is to facilitate inferencing. Conceptual dependencies captured many inferences that can be made about physical actions. Since not all events are thought of in terms of physical features, other representations seemed to be necessary. Scripts captured knowledge about the most likely inferences to make in stereotypical situations. Plan and goal representations provided inferential capabilities at this level of understanding. TOPs combined goal/plan knowledge with contextual features, to provide more specific predictions useful in planning or plan understanding.

The inferential capabilities provided by these representations has led to a general approach to processing which emphasizes the use of knowledge to guide processing as much as possible. This has been true in many types of processing, such as language understanding, learning, planning, and reasoning. In this section, we will discuss some of the processing theories which have been developed around these representations. In particular, we will focus on two areas: language understanding and reasoning.

5.1. *Language Understanding*

Conceptual dependency was originally developed as a representation theory for language understanding. The processing theory which accompanied it was highly knowledge-based. One of the central points of this theory was that *expectations* could be generated from representations, which would then play a key role in guiding the processing of subsequent input.

This general approach to language understanding was radically different from most other work of its time which focused largely on syntactic processing (e.g., [14,15]). Because of the difference in focus, several issues arose. First, in the expectation-based approach, a key component of the language understander is a conceptual knowledge base. In addition, the system must have linguistic or syntactic knowledge. How should these two types of knowledge be integrated, if at all? Should syntactic knowledge and conceptual knowledge be completely separate, or completely integrated, or somewhere in between? And in processing, should syntactic and semantic analysis proceed in tandem, or should the process be modularized?

Another issue also arose from the different goal of a conceptual analyzer. Since the final product of the parser is not syntactic in nature, what should the role of syntax be in conceptual analysis? Is it necessary to build a syntactic representation of an input text during parsing.

this representation is simply going to be thrown away afterward, leaving the conceptual representation? Or can a conceptual analyzer get away with less syntactic analysis, since this analysis is not the final goal of the parser?

5.1.1. Request-Based Parsing

One approach that has been taken to conceptual analysis has involved the use of ***requests***, or test-action pairs, to encode parsing knowledge. Requests were first used in Riesbeck's analyzer [16]. The requests in this parser were mainly stored in the lexicon.

A request could be in one of two states: active or inactive. A request was activated when the parser encountered a word whose dictionary entry contained that request. Once active, a request stayed active until it *fired*, or was executed; or until it was explicitly deactivated by another request. A request fired if it was in the active state and the conditions of active memory satisfied the test portion of the request's test-action pair.

Requests were largely responsible for building conceptual representations. Thus, common actions performed by requests were building an instantiation of a particular concept, or filling a slot in an already-built concept. For example, the dictionary entry of verb "ate" contained a request to build an instantiation of the concept INGEST (the Conceptual Dependency [1] primitive underlying eating and drinking); and a request to look for a noun group after the verb which referred to a food item, which, if found, was placed in the OBJECT slot of the INGEST.

Some requests in Riesbeck's parser were not stored in the lexicon. For example, at the beginning of a sentence a request was activated which looked for a noun group. When one was found, it was placed in a variable called #SUBJ. Later, when a request from a verb took the noun group from #SUBJ and placed it in the appropriate slot (usually the ACTOR slot), of the verb's representation. However, the number of requests like this was quite small, and thus most of the requests in the system were lexically-based.

The request-based method of parsing has been used in many other parsers since Riesbeck's parser. In the Conceptual Analyzer (CA) [17], requests were used in much the same way as in Riesbeck's analyzer, but with the elimination of many non-lexically-based requests. Thus, instead of a request activated at the beginning of a sentence which looked for a noun group, CA had requests in the dictionary entries of verbs, looking back in the sentence for a noun group which could function as the subject. For instance, the verb "ate" had a request looking for a noun group to its left, which was an ANIMATE. If such a noun group was found, it was placed in the ACTOR slot of the INGEST. Other parsers using request-based knowledge include the Integrated Partial Parser (IPP) [18], the Word Expert Parser (WEP) [19], and BORIS [13].[3]

5.1.2. Integration and Request-Based Parsing

One of the main tenets behind request-based conceptual analysis has been that the traditional separation of text analysis into morphological, syntactic, semantic, and pragmatic phases should be eliminated. This is for two reasons: first, given that the goal of conceptual parsing is to build a meaning representation, and not a syntactic analysis, there is no *a priori* reason for a separate syntactic analysis phase to exist. Second, since semantic and pragmatic information can sometimes help to eliminate syntactic, or even morphological, ambiguities, semantics should be brought into the parsing process as quickly as possible. An example given in [20] involved the following sentence:

> Hunting dogs can be dangerous.

Out of context, this sentence is syntactically (and semantically) ambiguous. However, in the following contexts, the ambiguity is eliminated:

> Do you want to try shooting those dogs that have been pillaging the village?—No, hunting dogs can be dangerous.

[3] The test-action pairs in WEP and BORIS were called *demons*, rather than requests.

Let's take some dogs with us when we go to hunt moose.—No, hunting dogs can be dangerous.

In the context of shooting dogs, in the first example, the words "hunting dogs" make more sense as a gerund and its object, since the semantic interpretation of this syntactic sense fits into the semantic context. However, in the second sentence, since the context is hunting moose, and since we know that dogs are often used to assist in the hunting of other animals, the better interpretation of "hunting dogs" is that "hunting" is an adjective, modifying "dogs."

The desire to do away with the separation of syntax and semantics has been articulated further in [21] in terms of the *Integrated Processing Hypothesis*. Schank and Birnbaum suggest that the integration of syntax and semantics in a parser can be measured in terms of three aspects of the parser: its *control structure*, or the processes which occur during parsing; its *representational structures*, or the representations which it builds during the parsing process; and its *knowledge base*, or the set of rules which the parser draws on and which drive the parse of a text. A parser with an integrated control structure would have no separate syntactic or semantic processes or phases; one with integrated representational structures would not build any separate syntactic structure during the parsing process, but instead only semantic representations of its text; and a parser with an integrated knowledge base would have no rules which contained only syntactic knowledge.

The Integrated Processing Hypothesis states that syntax and semantics should be completely integrated in the control structure and representational structures, and that much of the knowledge base should also be integrated, although some separate syntax will exist in the knowledge base. Thus, it advocates that no separate phases of parsing should exist, that no purely syntactic information should be contained in the representations built during the parsing process, and that only some rules will exist in the parser's knowledge base which are purely syntactic.

In light of the goal of bringing semantic and conceptual information to bear as early as possible in the parsing process, at least some of the motivation behind the Integrated Processing Hypothesis is clear. Since semantics can aid even the earliest stages of the parsing process, it is important that a parser's control structure be highly integrated. One way to ensure that the control structure is integrated is to integrate the semantic and syntactic knowledge in a parser as much as possible. Thus, its knowledge base should also be highly integrated. Finally, with regard to representational structure, the goal of conceptual parsers is to build a conceptual representation of the meaning of a text, not to produce a syntactic parse tree for the text. Therefore, syntactic representations should not be built during parsing unless it is clear that they aid in the process of building the conceptual representation. Given these motivations, then, the Integrated Processing Hypothesis takes almost as strong a stand on the integration of syntax and semantics as can be taken

The result of these principles has been the use of lexically-based requests to encode syntactic knowledge. Lexically-based requests meet, more or less, with the criteria of the Integrated Processing Hypothesis. Certainly syntax and semantics are completely integrated in terms of process. Requests carry on in parallel any syntactic processing with the building of conceptual structures. Requests are also as integrated as possible with respect to the knowledge base, since they usually contain both syntactic and semantic knowledge. For example, the requests discussed earlier for the word "ate" contained the conceptual knowledge that "ate" refers to the concept INGEST, and that the ACTOR of INGEST should be an ANIMATE and the OBJECT of INGEST should be a food item. These same requests also contained the syntactic information that the ACTOR of "ate" occurs before the verb in an active sentence, and the OBJECT after the verb In representational structures, also, the integration is high using requests, since no syntactic representations are explicitly built by the requests.

5.1.9. Semantics-First Parsing

Another system which utilized a more explicit grammar, yet was highly influenced by semantic information was the MOPTRANS system [22,23]. MOPTRANS translated short (1–3 sentences) newspaper stories about terrorism and crime. Source language included English, Spanish, Ger-

man, and Chinese; target languages were English and German. In this system, although knowledge was modularized into relatively separate syntactic and semantic components, the application of knowledge was integrated at processing time.

As MOPTRANS parsed a piece of text, the semantic and syntactic representations that it built were kept in its active memory. During parsing, new constituents were added to active memory as each new word was read. As new constituents were added, semantics was asked if anything in active memory "fit together" well; that is, if there were any semantic attachments that could be made between the elements in active memory. If so, MOPTRANS' syntactic rules were consulted to see if any of these semantic attachments were syntactically legal. If so, the appropriate syntactic rule was run, making syntactic and semantic attachments, and possibly removing some constituents from active memory. In other words, semantics proposed various attachments, and syntax acted as a filter, choosing which of these attachments made sense according to the syntax of the input.

To make this more clear, consider how the following simple sentence was processed by MOPTRANS:

John gave Mary a book.

MOPTRANS' dictionary definitions contained information about what semantic representation the parser should build when it encountered a particular word. Thus, "John" caused the representation PERSON to appear in the parser's active memory. At the same time, since "John" is a proper noun, the syntactic class NP was also activated.

When the word "gave" was processed, MOPTRANS' definition of this word caused the CD representation ATRANS to be placed in active memory. At this point, then, active memory contained the following:

```
"John"                  "gave"
NP                      V
(PERSON NAME JOHN)      (ATRANS)
```

MOPTRANS considered the two semantic representations in active memory, PERSON and ATRANS. The semantic analyzer tried to combine these representations in whatever way it could. It concluded that the PERSON could be either the ACTOR or the RECIPIENT of the ATRANS, since the constraints on these roles are that they must be ANIMATE. It also concluded that the PERSON could be the OBJECT of the ATRANS (that is, the thing whose control or possession is being transferred). However, since this role was expected to be a PHYSICAL-OBJECT, a more general category than ANIMATE, the match was not as good as with the ACTOR or RECIPIENT roles.[4]

This is the point at which the MOPTRANS parser utilized its syntactic rules. Semantics determined that two possible attachments were prefered. Now the parser examined its syntactic rules to see if any of them could yield either of these attachments. Indeed, the parser's Subject Rule would assign the PERSON to be the ACTOR of the ATRANS. The Subject Rule is shown in Figure 12. This rule applied when an NP was followed by a V, and when the NP could fill the ACTOR slot of the semantic representation of the V. The NP would be marked as the SUBJECT of the V, and the V would be marked as having a subject (S). As dictated by the RESULT of the rule, the S would remain in active memory, but the NP would be removed, since its role as subject would prevent many subsequent attachments to it, such as PP attachments. In addition to these syntactic assignments, the semantic representation of the NP "John" would be placed in the ACTOR slot of the ATRANS representing the verb.

After the execution of this rule, then, active memory contained the following:

```
"gave"
S
(ATRANS ACTOR (PERSON NAME JOHN))
```

[4] For details about how the semantic analyzer reached these conclusions, see [22].

```
Subject Rule

Syntactic pattern:        NP, V (active)
Additional restrictions:  NP is not already attached syntactically
Syntactic assignment:     NP is SUBJECT of V, V is indicative (S)
Semantic action:          NP is ACTOR of V (or another slot, if specified
                             by V)
Result:                   S


Dative Movement Rule

Syntactic pattern:        S, NP
Additional restrictions:  S allows dative movement
Syntactic assignment:     NP is (syntactic) INDIRECT OBJECT of S
Semantic action:          NP is (semantic) RECIPIENT of S (or another
                             slot, if specified by S)
Result:                   S, NP


Direct Object Rule

Syntactic pattern:        S, NP
Syntactic assignment:     NP is (syntactic) DIRECT OBJECT of S
Semantic action:          NP is (semantic) OBJECT of S (or another
                             slot, if specified by S)
Result:                   S, NP
```

Figure 12. Some grammar rules used in MOPTRANS.

The rest of the sentence was parsed in a similar fashion. After the word "Mary" was read active memory contained two nodes: the node above representing "gave," and a node for "Mary" which was an NP whose semantic representation was (PERSON NAME MARY). To determine how these nodes should be attached, semantics was asked for its preference. It determined that the RECIPIENT slot of the ATRANS was the best attachment.[5] Syntax was consulted to see if any syntactic rules could make this attachment. This time, the Dative Movement Rule in Figure 12 was found. When applied, this rule assigned "Mary" as the indirect object of "gave," and placed the PERSON concept representing "Mary" into the RECIPIENT slot of the ATRANS, leaving active memory containing the following:

```
"gave"                                 "Mary"
S                                      NP
(ATRANS ACTOR (PERSON NAME JOHN)       (PERSON NAME MARY)
        RECIP (PERSON NAME MARY))
```

The final NP in the sentence, "the book," was attached to "gave" in a similar way. Semantics was asked to determine the best attachment of "book," which was represented as a PHYSICAL-OBJECT, to other concepts in active memory, which at this point contained the nodes for "gave" and "Mary." Semantics determined that the best attachment was to the OBJECT role of the ATRANS, since a book is a type of PHYS-OBJ, which is what ATRANS expects to fill its OBJECT slot. The syntactic rule which could perform this attachment was the Direct Object Rule, also shown in Figure 12. This rule was applied, yielding the final semantic representation:

[5] Just as earlier, "Mary" could either be the ACTOR or RECIPIENT of the ATRANS, but "John" has already been assigned as the ACTOR.

Directed prediction

1. Predict a concept.
2. Retrieve associated patterns which will recognize that concept.
3. Predict patterns' "leading edges" (concepts or lexical items).

Opportunistic recognition

1. Activate a concept.
2. Find all referenced predictions.
3. Refine to more specific predictions.
4. If the predictions' pattern is complete then activate the predicted concept; else predict the pattern's next element.

Figure 13. Components of DMAP.

```
(ATRANS ACTOR  (PERSON NAME JOHN)
        OBJECT (PHYS-OBJ TYPE BOOK)
        RECIP  (PERSON NAME MARY))
```

Note that although the Direct Object Rule could have applied syntactically (and even semantically) when "Mary" was found after the verb, it was never even considered. This is because the semantic analyzer prefered to place "Mary" in the RECIPIENT slot of the ATRANS rather than the OBJECT slot. Since a syntactic rule was found which accomodated this attachment, namely the Dative Movement Rule, the parser never tried to apply the Direct Object Rule.

5.1.4. Direct Memory Access Parsing

One of the common themes of many of the processing structures in the conceptual dependency family is that these structures also serve as memory structures. However, many of the language understanding programs have not utilized this fact. Martin has integrated processing and memory further in the Direct Memory Access Parser (DMAP) [24]. In DMAP, there is no distinction between representations built during parsing and episodic memory. Representations simply consist of activated memory nodes. This enables the system to make inferences based on all information in the memory.

DMAP allows any concept in memory, at any level of generality, to post expectations for other concepts or lexical items. The posting and verification of expectations is broken down into two components: *directed prediction* and *opportunistic recognition* [24, p. 476], shown in Figure 13.

Patterns occur at all levels in memory, and can refer to lexical items or phrases, or schemas such as scripts or MOPs. Thus, inferences and lexical expectations are handled with the same mechanism. Due to this complete integration, changes in DMAP's memory result in automatic changes in pattern-driven expectations.

5.2. Planning and Reasoning

Many of the conceptual structures discussed in this paper suggest a paradigm for planning, problem-solving, and other types of reasoning tasks. The main message is that memory contains many different types of structures, which can make very specific predictions that can be useful in these tasks. Thus, the general approach taken using these structures is that, rather than viewing these tasks as a search problem, much of planning and reasoning is a memory retrieval task.

Reflecting this general approach, the *case-based reasoning* paradigm has developed. We can contrast case-based reasoning with more traditional planning systems. In traditional planning, a goal is pursued by constructing sequences of operators (a plan) which will satisfy it (e.g., [25–29]). Operators have *preconditions* which must be satisfied before they can be applied. Thus, if an operator is selected whose preconditions are not true, then *subgoals* are generated. If several goals are to be pursued at once, then plans are found to satisfy each goal individually, and then interactions between the plans are checked, to make sure that the side-effects from one plan does not destroy the preconditions necessary for other plans.

This approach to planning is very search-intensive. Operators must be selected, from a set of possibly many operators which could apply, then preconditions must be planned for by selecting more operators. If a precondition cannot be satisfied, then the planner must backtrack, selecting another possible operator to satisfy the original goal.

In contrast to this approach, case-based planners search their memory for the closest pla previously constructed, based on a set of possible indices including the goal or goals to be satisfied, and other features of the context which may be relevant. This is very similar in spirit to the notion of TOPs, which are also organized according to a goal configuration as well as specific features of the context. Once a previous plan is selected, it may have to be modified to meet the specifics of the current planning situation. This involves applying plan transformation rules which dictate, according to the differences in the former and current situations, how the plan must change.

The case-based approach was used in the CHEF program [30], which planned Szechuan recipes given a set of constraints, such as ingredients to be used, style, and desired flavor. CHEF found previous recipes in its memory, indexed according to these features, as well as the interactions between features. Then the previous recipe was modified, if it differed from the constraints of the current case. Modification of recipes could involve further goal interaction, which either resulted in further modifications, and/or possible abandonment of one or more goals.

An example of CHEF in operation was its planning of a stir-fry dish with beef and broccoli. In this case, CHEF's goal was to build a recipe satisfying three constraints: the goal to have a stir-fried dish, the goal to include beef, and the goal to include broccoli. Initially, CHEF found a previously constructed plan (recipe) in its memory for beef with green beans, another stir-fry dish. The initial modification was to simply substitute the broccoli for the green beans. A set of *object critics* was responsible for making this substitution, and for identifying differences in the recipe required by the change, such as chopping the broccoli, etc.

The simple substitution created a goal interaction problem, however: the brocolli would become soggy if cooked in the same way as the green beans. Given this failure, CHEF had to construct an explanation of why this happened. After analysis of the failure, CHEF further modified the new plan by suggesting that the broccoli should be added later in the cookir process than the green beans.

The important thing to note in this example is the marked difference in the plan construction process: rather than reasoning from first principles, CHEF created plans by recalling previous plans, then modifying them accordingly. Sometimes this would lead to planning failures, which would then have to be explained, leading to further plan modifications.

Several other case-based systems have been written to perform other planning and reasoning tasks. Simpson's PERSUADER [31] played the role of a mediator to resolve adversarial conflicts. JUDGE [32] reasoned about criminal trials in order to decided upon appropriate sentences. More recent systems have investigated the combination of case-based and rule-based reasoning. These include MEDIATOR [33], which investigated Simpson's domain further using a hybrid approach; and CABARET [34] which operated in the domain of legal reasoning.

Similar work has also been done in an explanation-generating program called ABE (Adaptation-Based Explanation) [35]. ABE tried to explain events in terms of *explanation patterns* [36], or pre-packaged explanations. These patterns were explanations of previously encountered explanation failures. The same basic paradigm was followed: an explanation pattern was retrieved from memory, according to the features of the events being explained. In the case that no exact match was found, the retrieved explanation was modified according to a set of rules so as to fit the current situation. The result was a system which produced explanations without having to generate causal chains from scratch for each explanation.

6. CONCLUSION

The representational structures that have descended from conceptual dependencies are many and varied. They range from low-level descriptions of physical actions, such as CD's themselv to compilations of sequences of these descriptions such as scripts, abstractions of these sequenc

such as MOPs, and goal-oriented representations such as planboxes, named plans, goal categories, and TOPs.

Given the wide range of structures, it might seem that they do not share much in common. However, the motivation behind all of them is the same: to facilitate inferences. As people, we can make inferences at all sorts of levels of generality; the range of proposed structures reflects this.

As these theories have developed, another common theme has been that processing structures and memory structures are the same. That is, not only are these structures useful in processing tasks such as language understanding, planning, and reasoning, but they are also used to store and retrieve episodes. As a result, much of the processing done using these structures is memory storage and retrieval. For example, in planning, rather than constructing a plan from first principles every time a goal arises, the case-based approach suggests that much of what is going on is finding the appropriate previously-built plan in memory, and then adapting it to the current situation.

One might feel that the notion of building structures around inferential capability is a painfully obvious one. However, if we examine related work in language understanding, problem solving, reasoning, and other tasks, we find that this does not seem to be the case. For example, in natural language processing, semantic theories that existed before conceptual dependencies (and even afterward) did not address this issue. Representations such as Katz and Fodor's [37] and Fillmore's case grammar [38] made some attempt to categorize dependencies between constituents in sentences (although many of these dependencies were still highly syntactic in nature), but lexical items (or word senses) were used as the fundamental pieces out of which the representations were built. There was no attempt to capture the similarities in meaning across different lexical items; hence these representations would not have easily facilitated inferential capabilities. Even now, many natural language semantic representations such as SNePS [39] and HPSG [40] do not attempt to capture these meaning similarities in the representational predicates used.

The same can be said of many planning representations. Although operators are usually written so as to be general and facilitate the planning process (a form of inference), in constructive planners no attempt is made to represent knowledge about what sequences of operators are best for particular situations. This information has to be re-derived over and over again. This is a fault of the representation: operators are too general, and do not capture this level of inferential capability.

Thus, perhaps the most important common thread that connects the CD family of research is the assumption that intelligence is knowledge-intensive, rather than processing-intensive. If the representations used to perform a task are rich enough, then processing strategies traditionally associated with AI programs, such as search, become much less important. The result is simpler processing theories which are highly dependent on a very rich, and highly organized, knowledge base.

REFERENCES

1. R. Schank, Conceptual dependency: A theory of natural language understanding, *Cognitive Psychology* **3**, 552–631 (1972).

2. R. Schank, *Conceptual Information Processing*, North-Holland, Amsterdam (1975).

3. R. Schank and R. Abelson, *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum Associates, Hillsdale NJ (1977).

4. C. Rieger, Conceptual memory and inference, In [2].

5. M. Minsky, A framework for representing knowledge, In *The Psychology of Computer Vision*, (Edited by P. Winston), McGraw-Hill, New York (1975).

6. D. Rumelhart, Notes on a schema for stories, In *Representation and Understanding: Studies in Cognitive Science*, (Edited by D. Bobrow and A. Collins), Academic Press, New York (1975).

7. D. Bobrow and D. Norman, Some principles of memory schemata, In *Representation and Understanding: Studies in Cognitive Science*, (Edited by D. Bobrow and A. Collins), Academic Press, New York (1975).

8. G. Bower, J. Black and T. Turner, Scripts in memory for text, *Cognitive Psychology* 11, 177–220 (1979).

9. E. Charniak, A framed PAINTING: The representation of a common sense knowledge fragment, *Cognitive Science* 1, 355–394 (1977).
10. R. Schank, *Dynamic Memory*, Cambridge University Press, Cambridge U.K. (1982).
11. R. Wilensky, Understanding goal-based stories, Ph.D. Thesis, Department of Computer Science, Yale University, Research Report #140 (1978).
12. C. Seifert, G. McKoon, R. Abelson and R. Ratcliff, Memory connections between thematically similar episodes, *Journal of Experimental Psychology: Human Learning and Memory* 12, 220–231 (1986).
13. M. Dyer, *In-depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*, MIT Press, Cambridge, MA (1983).
14. W. Woods, R. Kaplan and B. Nash-Webber, The lunar sciences natural language information system: Final report, Technical Report #2378, Bolt Beranek and Newman, Inc., Cambridge, MA (1972).
15. T. Winograd, *Understanding Natural Language*, Academic Press, New York (1972).
16. C. Riesbeck, Conceptual analysis, In [2].
17. L. Birnbaum and M. Selfridge, Conceptual analysis of natural language, In *Inside Computer Understanding*, (Edited by R. Schank and C. Riesbeck), Lawrence Erlbaum Associates, Hillsdale, NJ (1981).
18. M. Lebowitz, Generalization and memory in an integrated understanding system, Ph.D. Thesis, Department of Computer Science, Yale University, Research Report #186 (1980).
19. S. Small, Word expert parsing: A theory of distributed word-based natural language understanding, Ph.D. Thesis, Department of Computer Science, University of Maryland (1980).
20. C. Riesbeck and R. Schank, Comprehension by computer: Expectation-based analysis of sentences in context Research Report #78, Department of Computer Science, Yale University (1976).
21. R. Schank and L. Birnbaum, Memory, meaning, and syntax, Research report #189, Department of Computer Science, Yale University (1980).
22. S. Lytinen, The representation of knowledge in a multi-lingual, integrated parser, Ph.D. Thesis, Department of Computer Science, Yale University, Research Report #340 (1984).
23. S. Lytinen, Dynamically combining syntax and semantics in natural language processing, In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, 574–578 (1986).
24. C. Martin, Pragmatic interpretation and ambiguity, In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society, Ann Arbor, MI, August*, 474–481 (1989).
25. A. Newell and H. Simon, *Human Problem Solving*, Prentice-Hall, New York (1972).
26. R. Fikes and N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* 2, 189–208 (1971).
27. E. Sacerdote, A structure for plans and behavior, Research Report #109, SRI Artificial Intelligence Centr Palo Alto, CA (1975).
28. G. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier, Artificial Intelligence Series, New York (1975).
29. J. Laird, A. Newell and P. Rosenbloom, Soar: An architecture for general intelligence, *Artificial Intelligence* 33, 1–64 (1987).
30. K. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*, Academic Press, Cambridge, MA (1989).
31. R. Simpson, A computer model of case-based reasoning in problem-solving: An investigation in the domain of dispute mediation, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology (1985).
32. W. Bain, Case-based reasoning: A computer model of subjective assessment, Ph.D. Thesis, Department of Computer Science, Yale University (1986).
33. K. Sycara, Resolving adversarial conflicts: An approach integrating case-based and analytic methods, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology (1987).
34. E. Rissland and D. Skalak, Combining case-based and rule-based reasoning: A heuristic approach, In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit MI, August*, 524–530 (1989).
35. A. Kass, Adaption-based explanation: Extending script/frame theory to handle novel input, In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit MI, August*, 141–147 (1989).
36. R. Schank, *Explanation Patterns: Understanding Mechanically and Creatively*, Lawrence Erlbaum Associates, Hillsdale, NJ (1986).
37. J. Katz and J. Fodor, The structure of a semantic theory, *Language* 39, 170–210 (1963).
38. C. Fillmore, The case for case, In *Universals in Linguistic Theory*, (Edited by E. Bach and R. Harms), pp. 1–88, Holt, Rinehart, and Winston, New York (1968).
39. S. Shapiro, The SNePS semantic network processing system, In *Associated Networks: Representation and Use of Knowledge by Computers*, (Edited by N. Findler), Academic Press, New York (1979).
40. C. Pollard and I. Sag, *Information-Based Syntax and Semantics*, Center for the Study of Language and Information, Menlo Park, CA (1987).

New Approaches to Robotics

RODNEY A. BROOKS

In order to build autonomous robots that can carry out useful work in unstructured environments new approaches have been developed to building intelligent systems. The relationship to traditional academic robotics and traditional artificial intelligence is examined. In the new approaches a tight coupling of sensing to action produces architectures for intelligence that are networks of simple computational elements which are quite broad, but not very deep. Recent work within this approach has demonstrated the use of representations, expectations, plans, goals, and learning, but without resorting to the traditional uses of central, abstractly manipulable or symbolic representations. Perception within these systems is often an active process, and the dynamics of the interactions with the world are extremely important. The question of how to evaluate and compare the new to traditional work still provokes vigorous discussion.

THE FIELD OF ARTIFICIAL INTELLIGENCE (AI) TRIES TO make computers do things that, when done by people, are described as having indicated intelligence. The goal of AI has been characterized as both the construction of useful intelligent systems and the understanding of human intelligence (1). Since AI's earliest days (2) there have been thoughts of building truly intelligent autonomous robots. In academic research circles, work in robotics has influenced work in AI and vice versa (3).

Over the last 7 years a new approach to robotics has been developing in a number of laboratories. Rather than modularize perception, world modeling, planning, and execution, the new approach builds intelligent control systems where many individual modules each directly generate some part of the behavior of the robot. In the purest form of this model each module incorporates its own perceptual, modeling, and planning requirements. An arbitration or mediation scheme, built within the framework of the modules, controls which behavior-producing module has control of which part of the robot at any given time.

The work draws its inspirations from neurobiology, ethology, psychophysics, and sociology. The approach grew out of dissatisfactions with traditional robotics and AI, which seemed unable to deliver real-time performance in a dynamic world. The key idea of the new approach is to advance both robotics and AI by considering the problems of building an autonomous agent that physically is an autonomous mobile robot and that carries out some useful tasks in an environment that has not been specially structured or engineered for it.

There are two subtly different central ideas that are crucial and have led to solutions that use behavior-producing modules:

- *Situatedness*: The robots are situated in the world—they do not deal with abstract descriptions, but with the "here" and "now" of the environment that directly influences the behavior of the system.
- *Embodiment*. The robots have bodies and experience the world directly—their actions are part of a dynamic with the world, and the actions have immediate feedback on the robots' own sensations.

An airline reservation system is situated but it is not embodied—it deals with thousands of request per second, and its responses vary as its database changes, but it interacts with the world only through sending and receiving messages. A current generation industrial spray-painting robot is embodied but it is not situated—it has a physical extent and its servo routines must correct for its interactions with gravity and noise present in the system, but it does not perceive any aspects of the shape of an object presented to it for painting and simply goes through a pre-programmed series of actions.

This new approach to robotics makes claims on how intelligence should be organized that are radically different from the approach assumed by traditional AI.

Traditional Approaches

Although the fields of computer vision, robotics, and AI all have their fairly separate conferences and specialty journals, an implicit intellectual pact between them has developed over the years. None of these fields is experimental science in the sense that chemistry, for example, can be an experimental science. Rather, there are two ways in which the fields proceed. One is through the development and synthesis of models of aspects of perception, intelligence, or action, and the other is through the construction of demonstration systems (4). It is relatively rare for an explicit experiment to be done. Rather, the demonstration systems are used to illustrate a particular model in operation. There is no control experiment to compare against, and very little quantitative data extraction or analysis. The intellectual pact between computer vision, robotics, and AI concerns the assumptions that can be made in building demonstration systems. It establishes conventions for what the components of an eventual fully situated and embodied system can assume about each other. These conventions match those used in two critical projects from 1969 to 1972 which set the tone for the next 20 years of research in computer vision, robotics, and AI.

At the Stanford Research Institute (now SRI International) a mobile robot named Shakey was developed (5). Shakey inhabited a set of specially prepared rooms. It navigated from room to room, trying to satisfy a goal given to it on a teletype. It would, depending on the goal and circumstances, navigate around obstacles consisting of large painted blocks and wedges, push them out of the way, or push them to some desired location. Shakey had an onboard black-and-white television camera as its primary sensor. An offboard

The author is in the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139

computer analyzed the images and merged descriptions of what was seen into an existing symbolic logic model of the world in the form of first order predicate calculus. A planning program, STRIPS, operated on those symbolic descriptions of the world to generate a sequence of actions for Shakey. These plans were translated through a series of refinement into calls to atomic actions in fairly tight feedback loops with atomic sensing operations using Shakey's other sensors, such as a bump bar and odometry.

Shakey only worked because of very careful engineering of the environment. Twenty years later, no mobile robot has been demonstrated matching all aspects of Shakey's performance in a more general environment, such as an office environment. The rooms in which Shakey operated were bare except for the large colored blocks and wedges. This made the class of objects that had to be represented very simple. The walls were of a uniform color and carefully lighted, with dark rubber baseboards, making clear boundaries with the lighter colored floor. This meant that very simple and robust vision of trihedral corners between two walls and the floor could be used for relocalizing the robot in order to correct for drift in the odometric measurements. The blocks and wedges were painted different colors on different planar surfaces. This ensured that it was relatively easy, especially in the good lighting provided, to find edges in the images separating the surfaces and thus to identify the shape of the polyhedron. Blocks and wedges were relatively rare in the environment, eliminating problems due to partial obscurations.

At MIT, a camera system and a robot manipulator arm were programmed to perceive an arrangement of white wooden blocks against a black background and to build a copy of the structure from additional blocks. This was called the *copy-demo* (*6*). The programs to do this were very specific to the world of blocks with rectangular sides and would not have worked in the presence of simple curved objects, rough texture on the blocks, or without carefully controlled lighting. Nevertheless it reinforced the idea that a complete three-dimensional description of the world could be extracted from a visual image. It legitimized the work of others, such as Winograd (*7*), whose programs worked in a make-believe world of blocks—if one program could be built which understood such a world completely and could also manipulate that world, then it seemed that programs which assumed that abstraction could in fact be connected to the real world without great difficulty.

The role of computer vision was "given a two-dimensional image, infer the objects that produced it, including their shapes, positions, colors, and sizes" (*8*). This attitude lead to an emphasis on recovery of three-dimensional shape (*9*), from monocular and stereo images. A number of demonstration recognition and location systems were built, such as those of Brooks (*10*) and Grimson (*11*), although they tended not to rely on using three-dimensional shape recovery.

The role of AI was to take descriptions of the world (though usually not as geometric as vision seemed destined to deliver, or as robotics seemed to need) and manipulate them based on a database of knowledge about how the world works in order to solve problems, make plans, and produce explanations. These high-level aspirations have very rarely been embodied by connection to either computer vision systems or robotics devices.

The role of robotics was to deal with the physical interactions with the world. As robotics adopted the idea of having a complete three-dimensional world model, a number of subproblems became standardized. One was to plan a collision-free path through the world model for a manipulator arm, or for a mobile robot—see the article by Yap (*12*) for a survey of the literature. Another was to understand forward kinematics and dynamics—given a set of joint or wheel torques as functions over time, what path would the robot hand or body follow. A more useful, but harder, problem is inverse kinematics and dynamics—given a desired trajectory as a function of time, for instance one generated by a collision-free path planning algorithm, compute the set of joint or wheel torques that should be applied to follow that path within some prescribed accuracy (*13*).

It became clear after a while that perfect models of the world could not be obtained from sensors, or even CAD databases. Some attempted to model the uncertainty explicitly (*14, 15*) and found strategies that worked in its presence, while others moved away from position-based techniques to force-based planning, at least in the manipulator world (*16*). Ambitious plans were laid for combining many of the pieces of research over the years into a unified planning and execution system for robot manipulators (*17*), but after years of theoretical progress and long-term impressive engineering, the most advanced systems are still far from the ideal (*18*).

These approaches, along with those in the mobile robot domain (*19, 20*), shared the *sense-model-plan-act* framework, where an iteration through the cycle could often take 15 minutes or more (*18, 19*).

The New Approach

Driven by a dissatisfaction with the performance of robots in dealing with the real world, and concerned that the complexity of run-time modeling of the world was getting out of hand, a number of people somewhat independently began around 1984 rethinking the general problem of organizing intelligence. It seemed a reasonable requirement that intelligence be reactive to dynamic aspects of the environment, that a mobile robot operate on time scales similar to those of animals and humans, and that intelligence be able to generate robust behavior in the face of uncertain sensors, an unpredictable environment, and a changing world. Some of the key realizations about the organization of intelligence were as follows:

• Agre and Chapman at MIT claimed that most of what people do in their day-to-day lives is not problem-solving or planning, but rather it is routine activity in a relatively benign, but certainly dynamic, world. Furthermore the representations an agent uses of objects in the world need not rely on naming those objects with symbols that the agent possesses, but rather can be defined through interactions of the agent with the world (*21, 22*).

• Rosenschein and Kaelbling at SRI International (and later at Teleos Research) pointed out that an observer can legitimately talk about an agent's beliefs and goals, even though the agent need not manipulate symbolic data structures at run time. A formal symbolic specification of the agent's design can be compiled away, yielding efficient robot programs (*23, 24*)

• Brooks at MIT argued that in order to really test ideas of intelligence it is important to build complete agents which operate in dynamic environments using real sensors. Internal world models that are complete representations of the external environment, besides being impossible to obtain, are not at all necessary for agents to act in a competent manner. Many of the actions of an agent are quite separable—coherent intelligence can emerge from independent subcomponents interacting in the world (*25-27*)

All three groups produced implementations of these ideas, using as their medium of expression a network of simple computational elements, hardwired together, connecting sensors to actuators, with a small amount of state maintained over clock ticks

Agre and Chapman demonstrated their ideas by building programs for playing video games. The first such program was called Pengi and played a concurrently running video game program, with one protagonist and many opponents which can launch dangerous projectiles (Fig. 1). There are two components to the architecture—a visual routine processor (VRP), which provides input to the system, and a network of standard logic gates, which can be categorized into three components: *aspect detectors, action suggestors,*

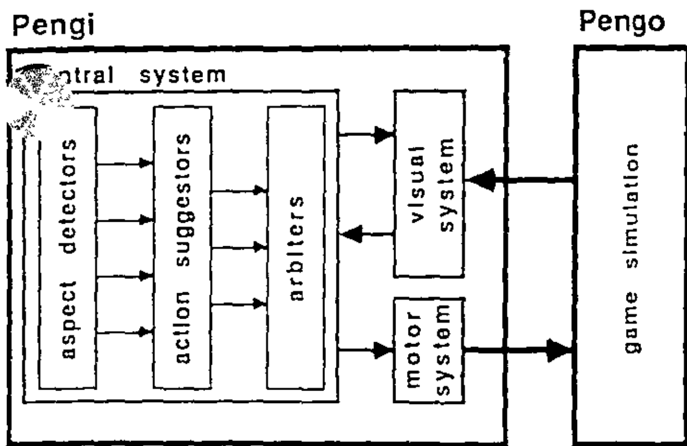


Fig. 1. The Pengi system (*21*) played a video game called Pengo. The control system consisted of a network of logic gates, organized into a visual system, a central system, and a motor system. The only state was within the visual system. The network within the central system was organized into three components: an aspect detector subnetwork, an action suggestor subnetwork, and an arbiter subnetwork.

and *arbiters*. The system plays the game from the same point of view as a human playing a video game, not from the point of view of the protagonist within the game. However, rather than analyze a visual bit map, the Pengi program is presented with an iconic version. The VRP implements a version of Ullman's visual routines theory (*28*), where markers from a set of six are placed on certain icons and follow them. Operators can place a marker on the *nearest opponent*, for example, and it will track that opponent even when it is no longer the nearest. The placement of these markers was the only state in the system. Projection operators let the player predict the consequences of actions, for instance, launching a projectile. The results of the VRP are analyzed by the first part of the central network and describe certain aspects of the world. In the mind of the designer, output signals designate such things as "the protagonist is moving," "a projectile from the north is about to hit the protagonist," and so on. The next part of the network takes Boolean combinations of such signals to suggest actions, and the third stage uses a fixed priority scheme (that is, it never learns) to select the next action. The use of these types of deictic representations was a key move away from the traditional AI approach of dealing only with named individuals in the world (for instance, *opponent-27* rather than the deictic *the-opponent-which-is-closest-to-the-protagonist*, whose objective identity may change over time) and lead to very different requirements on the sort of reasoning that was necessary to perform well in the world.

Rosenschein and Kaelbling used a robot named Flakey, which operated in the regular and unaltered office areas of SRI in the vicinity of the special environment for Shakey that had been built two decades earlier. Their architecture was split into a perception subnetwork and an action subnetwork. The networks were ultimately constructed of standard logic gates and delay elements (with feedback loops these provided the network with state), although the programmer wrote at a much higher level of abstraction—in terms of goals that the robot should try to satisfy. By formally specifying the relationships between sensors and effectors and the world, and by using off-line symbolic computation, Rosenschein and Kaelbling's high-level languages were used to generate provably correct, real-time programs for Flakey. The technique may be limited by the computational complexity of the symbolic compilation process as the programs get larger and by the validity of their models of sensors and actuators.

Brooks developed the subsumption architecture, which deliberately changed the modularity from the traditional AI approach. Figure 2 shows a vertical decomposition into task achieving behaviors rather than information processing modules. This architecture was used on robots which explore, build maps, have an onboard manipulator, walk, interact with people, navigate visually, and learn to coordinate many conflicting internal behaviors. The implementation substrate consists of networks of message-passing augmented finite state machines (AFSMs). The messages are sent over predefined "wires" from a specific transmitting to a specific receiving AFSM. The messages are simple numbers (typically 8 bits) whose meaning depends on the designs of both the transmitter and the receiver. An AFSM has additional registers which hold the most recent incoming message on any particular wire. The registers can have their values fed into a local combinatorial circuit to produce new values for registers or to provide an output message. The network of AFSMs is totally asynchronous, but individual AFSMs can have fixed duration monostables which provide for dealing with the flow of time in the outside world. The behavioral competence of the system is improved by adding more behavior-specific network to the existing network. This process is called layering. This is a simplistic and crude analogy to evolutionary development. As with evolution, at every stage of the development the systems are tested. Each of the layers is a behavior-producing piece of network in its own right, although it may implicitly rely on the presence of earlier pieces of network. For instance, an *explore* layer does not need to explicitly avoid obstacles, as the designer knows that the existing *avoid* layer will take care of it. A fixed priority arbitration scheme is used to handle conflicts.

These architectures were radically different from those in use in the robotics community at the time. There was no central model of the world explicitly represented within the systems. There was no implicit separation of data and computation—they were both distributed over the same network of elements. There were no pointers, and no easy way to implement them, as there is in symbolic programs. Any search space had to be a bounded in size a priori, as search nodes could not be dynamically created and destroyed during a search process. There was no central locus of control. In general, the separation into perceptual system, central system, and actuation system was much less distinct than in previous approaches, and indeed in these systems there was an intimate intertwining of aspects

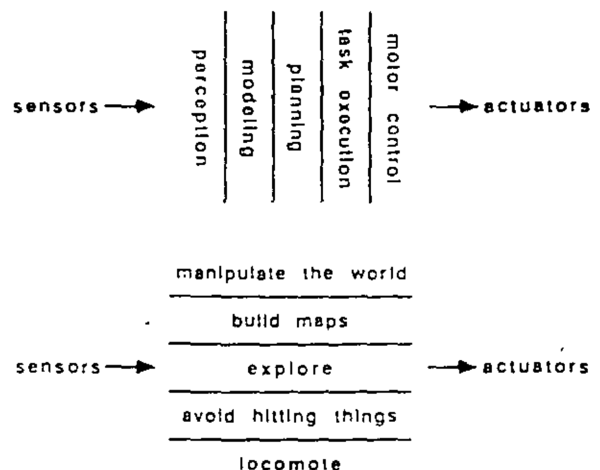


Fig. 2. The traditional decomposition for an intelligent control system within AI is to break processing into a chain of information processing modules (**top**) proceeding from sensing to action. In the new approach (**bottom**) the decomposition is in terms of behavior-generating modules each of which connects sensing to action. Layers are added incrementally, and newer layers may depend on earlier layers operating successfully, but do not call them as explicit subroutines.

of all three of these capabilities. There was no notion of one process calling on another as a subroutine. Rather, the networks were designed so that results of computations would simply be available at the appropriate location when needed. The boundary between computation and the world was harder to draw as the systems relied heavily on the dynamics of their interactions with the world to produce their results. For instance, sometimes a physical action by the robot would trigger a change in the world that would be perceived and cause the next action, in contrast to directly executing the two actions in sequence.

Most of the behavior-based robotics work has been done with implemented physical robots. Some has been done purely in software (*21*), not as a simulation of a physical robot, but rather as a computational experiment in an entirely make-believe domain to explore certain critical aspects of the problem. This contrasts with traditional robotics where many demonstrations are performed only on software simulations of robots.

Areas of Work

Perhaps inspired by this early work and also by Minsky's (*29*) rather more theoretical Society of Mind ideas on how the human mind is organized, various groups around the world have pursued behavior-based approaches to robotics over the last few years. The following is a survey of some of that work and relates it to the key issues and problems for the field.

One of the shortcomings in earlier approaches to robotics and AI was that reasoning was so slow that systems that were built could not respond to a dynamic real world. A key feature of the new approaches to robotics is that the programs are built with short connections between sensors and actuators, making it plausible, in principle at least, to respond quickly to changes in the world.

The first demonstration of the subsumption architecture was on the robot Allen (*25*). The robot was almost entirely reactive, using sonar readings to keep away from moving people and other moving obstacles, while not colliding with static obstacles. It also had a non-reactive higher level layer that would select a goal to head toward, and then proceed in that direction while the lower level reactive layer took care of avoiding obstacles. It thus combines non-reactive capabilities with reactive ones. More importantly, it used exactly the same sorts of computational mechanism to do both. In looking at the network of the combined layers there was no obvious partition into lower and higher level components based on the type of information flowing on the connections, or the finite state machines that were the computational elements. To be sure, there was a difference in function between the two layers, but there was no need to introduce any centralization or explicit representations to achieve a later, higher level process having useful and effective influence over an earlier, lower level.

The subsumption architecture was generalized (*30*) so that some of the connections between processing elements could implement a retina bus, a cable that transmitted partially processed images from one site to another within the system. It applied simple difference operators and region-growing techniques, to segment the visual field into moving and nonmoving parts, and into floor and non-floor parts. Location, but not identity of the segmented regions, was used to implement image-coordinate-based navigation. All the visual techniques were known to be very unreliable on single gray-level images, but by having redundant techniques operating in parallel and rapidly switching between them, robustness was achieved. The robot was able to follow corridors and moving objects in real time, with very little computational resources by modern computer vision standards.

This idea of using redundancy over many images is in contrast to the approach in traditional computer vision research of trying to extract the maximal amount of information from a single image, or pair of images. This lead to trying to get complete depth maps over a full field of view from a single pair of stereo images. Ballard (*31*) points out that humans do not do this, but rather servo their two eyes to verge on a particular point and then extract relative depth information about that point. With this and many other examples he points out that an active vision system, that is, one with control over its cameras, can work naturally in object-centered coordinates, whereas a passive vision system, that is, one which has no control over its cameras, is doomed to work in viewer-centered coordinates. A large effort is under way at Rochester to exploit behavior-based or animate vision. Dickmanns and Graefe (*32*) in Munich have used redundancy from multiple images, and multiple feature windows that track relevant features between images, while virtually ignoring the rest of the image, to control a truck driving on a freeway at over 100 kilometers per hour.

Although predating the emphasis on behavior-based robots, Raibert's hopping robots (*33*) fit their spirit. Traditional walking robots are given a desired trajectory for their body and then appropriate leg motions are computed. In Raibert's one-, two-, and four-legged machines, he decomposed the problem into independently controlling the hopping height of a leg, its forward velocity, and the body attitude. The motion of the robot's body emerges from the interactions of these loops and the world. Using subsumption, Brooks programmed a six-legged robot, Genghis (Fig. 3), to walk over rough terrain (*34*). In this case, layers of behaviors implemented first the ability to stand up, then to walk without feedback, then to adjust for rough terrain and obstacles by means of force feedback, then to modulate for this accommodation based on pitch and roll inclinometers. The trajectory for the body is not specified explicitly, nor is there any hierarchical control. The robot successfully navigates rough terrain with very little computation. Figure 4 shows the wiring diagram of the 57 augmented finite state machines that controlled it.

There have been a number of behavior-based experiments with robot manipulators. Connell (*35*) used a collection of 17 AFSMs to control an arm with two degrees of freedom mounted on a mobile base. When parked in front of a soda can, whether at floor level or on a table top, the arm was able to reliably find it and pick it up, despite other clutter in front of and under the can, using its local

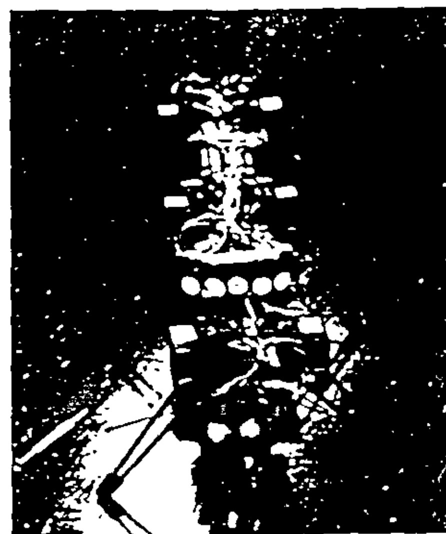


Fig. 3 Genghis is a six-legged robot measuring 35 centimeters in length. Each rigid leg is attached at a shoulder joint with two degrees of rotational freedom, each driven by a model airplane position controllable servo motor. The sensors are pitch and roll inclinometers, two collision-sensitive antennae, six forward-looking passive pyroelectric infrared sensors, and crude force measurements from the servo loops of each motor. There are four onboard eight-bit microprocessors three of which handle motor and sensor signals and one of which runs the subsumption architecture.

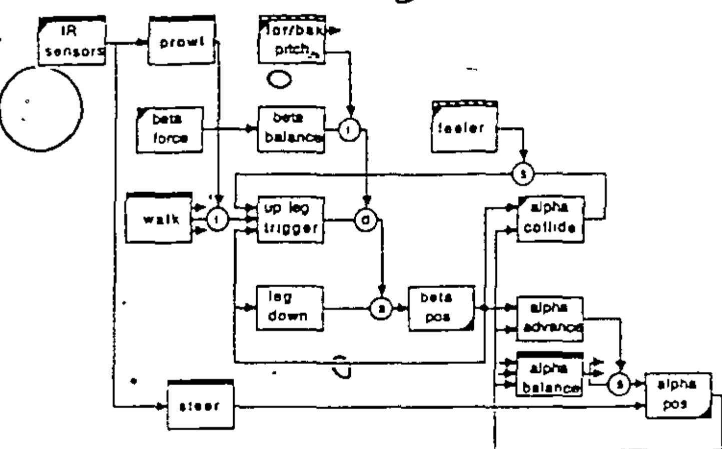


Fig. 4. The subsumption network to control Genghis consists of 57 augmented finite state machines, with "wires" connecting them that pass small integers as messages. The elements without bands on top are repeated six times, once for each leg. The network was built incrementally starting in the lower right corner, and new layers were added, roughly toward the upper left corner, increasing the behavioral repertoire at each stage.

sensors to direct its search. All the AFSMs had sensor values as their only inputs and, as output, actuator commands that then went through a fixed priority arbitration network to control the arm and hand. In this case, there was no communication between the AFSMs, and the system was completely reactive to its environment. Malcolm and Smithers (*36*) at Edinburgh report a hybrid assembly system. A traditional AI planner produces plans for a robot manipulator to assemble the components of some artifact, and a behavior-based system executes the plan steps. The key idea is to give the higher level planner robust primitives which can do more than carry out simple motions, thus making the planning problem easier.

Representation is a cornerstone topic in traditional AI. Mataric at MIT has recently introduced active representations into the subsumption architecture (*37*). Identical subnetworks of AFSMs are the representational units. In experiments with a sonar-based office-environment navigating robot named Toto, landmarks were broadcast to the representational substrate as they were encountered. A previously unallocated subnetwork would become the representation for that landmark and then take care of noting topological neighborhood relationships, setting up expectation as the robot moved through previously encountered space, spreading activation energy for path planning to multiple goals, and directing the robot's motion during goal-seeking behavior when in the vicinity of the landmark. In this approach the representations and the ways in which they are used are inseparable—it all happens in the same computational units within the network. Nehmzow and Smithers (*38*) at Edinburgh have also experimented with including representations of landmarks, but their robots operated in a simpler world of plywood enclosures. They used self-organizing networks to represent knowledge of the world, and appropriate influence on the current action of the robot. Additionally, the Edinburgh group has done a number of experiments with reactivity of robots, and with group dynamics among robots using a Lego-based rapid prototyping system that they have developed.

Many of the early behavior-based approaches used a fixed priority scheme to decide which behavior could control a particular actuator at which time. At Hughes, an alternative voting scheme was produced (*39*) to enable a robot to take advantage of the outputs of many behaviors simultaneously. At Brussels a scheme for selectively activating and de-activating complete behaviors was developed by Maes (*40*), based on spreading activation within the network itself. This scheme was further developed at MIT and used to program Toto amongst other robots. In particular, it was used to provide a learning mechanism on the six-legged robot Genghis, so that it could learn to coordinate its leg lifting behaviors, based on negative feedback from falling down (*41*).

Very recently there has been work at IBM (*42*) and Teleos Research (*43*) using Q-learning (*44*) to modify the behavior of robots. There seem to be drawbacks with the convergence time for these algorithms, but more experimentation on real systems is needed.

A number of researchers from traditional robotics (*45*) and AI (*46, 47*) have adopted the philosophies of the behavior-based approaches as the bottom of two-level systems as shown in Fig. 5. The idea is to let a reactive behavior-based system take care of the real time issues involved with interacting with the world while a more traditional AI system sits on top, making longer term executive decisions that affect the policies executed by the lower level. Others (*48*) argue that purely behavior-based systems are all that are needed.

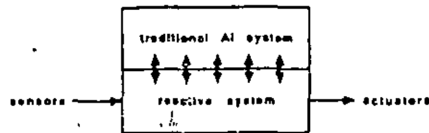
Evaluation

It has been difficult to evaluate work done under the banner of the new approaches to robotics. Its proponents have often argued on the basis of performance of systems built within its style. But performance is hard to evaluate, and there has been much criticism that the approach is both unprincipled and will not scale well. The unprincipled argument comes from comparisons to traditional academic robotics, and the scaling argument comes from traditional AI. Both these disciplines have established but informal criteria for what makes a good and respectable piece of research.

Traditional academic robotics has worked in a somewhat perfect domain. There are CAD-like models of objects and robots, and a modeled physics of how things interact (*16*). Much of the work is in developing algorithms that guarantee certain classes of results in the modeled world. Verifications are occasionally done with real robots (*18*), but typically those trials are nowhere nearly as complicated as the examples that can be handled in simulation. The sticking point seems to be in how well the experimenters are able to coax the physical robots to match the physics of the simulated robots.

For the new approaches to robotics, however, where the emphasis is on understanding and exploiting the dynamics of interactions with the world, it makes sense to measure and analyze the systems as they are situated in the world. In the same way modern ethology has prospered by studying animals in their native habitats, not just in Skinner boxes. For instance, a particular sensor, under ideal experimental conditions, may have a particular resolution. Suppose the sensor is a sonar. Then to measure its resolution an experiment will be set up where a return signal from the test article is sensed, and the resolution will be compared against measurements of distance made with a ruler or some such device. The experiment might be done for a number of different surface types. But when that sensor is installed on a mobile robot, situated in a cluttered, dynamically changing world, the return signals that reach the sensor may come from many possible sources. The object nearest the sensor may not be made of one of the tested materials. It may be at such an angle that the sonar pulse acts as though it were a mirror, and so the sonar sees a secondary reflection. The secondary lobes of the sonar might detect something in a cluttered situation where there was no such interference in the clean experimental situation. One of the main points of the new approaches to robotics is that these effects are extremely important on the overall behavior of a robot. They are also extremely difficult to model. So the traditional robotics approach of proving correctness in an abstract model may be somewhat meaningless in the new approaches. We need to find ways of formalizing

. 5. A number of jects involve combining a reactive system, ing sensors, and actors with a traditional ystem that does sym- ·c reasoning in order to tune the parameters of the situated component.



r understanding the dynamics of interactions with the world so t we can build theoretical tools that will let us make predictions out the performance of our new robots.

In traditional AI there are many classes of research contributions distinct from application deployment). Two of the most popular described here. One is to provide a formalism that is consistent · some level of description of some aspect of the world, for imple, qualitative physics, stereotyped interactions between eakers, or categorizations or taxonomies of animals. This class of ork does not necessarily require any particular results, theorems, or orking programs to be judged adequate; the formalism is the portant contribution. A second class of research takes some input presentation of some aspects of a situation in the world and makes prediction. For example, it might be in the form of a plan to effect me change in the world, in the form of the drawing of an analogy th some schema in a library in order to deduce some non-obvious t, or it might be in the form of providing some expert-level vice. These research contributions do not have to be tested in tuated systems—there is an implicit understanding among researchers about what is reasonable to "tell" the systems in the input ita.

In the new approaches there is a much stronger feeling that the bots must find everything out about their particular world by emselves. This is not to say that a priori knowledge cannot be corporated into a robot, but that it must be non-specific to the articular location in which the robot will be tested. Given the urrent capabilities of computer perception, this forces behavior-ased robots to operate in much more uncertain and much more parsely described worlds than traditional AI systems operating in mulated, imagined worlds. The new systems can therefore seem to ave much more limited abilities. I would argue (*48*), however, that e traditional systems operate in a way that will never be transportble to the real worlds that the situated behavior-based robots lready inhabit.

The new approaches to robotics have garnered a lot of interest, nd many people are starting to work on their various aspects. Some re trying to build systems using only the new approaches, others are ying to integrate them with existing work, and of course there is much work continuing in the traditional style. The community is divided on the appropriate approach, and more work needs to be done in making comparisons in order to understand the issues etter.

REFERENCES AND NOTES

1. P. H Winston, *Artificial Intelligence* (Addison-Wesley, Reading, MA, ed. 2, 1984)
2. A. M Turing, in *Machine Intelligence*, B. Meltzer and D Michie, Eds (American Elsevier, New York, 1970), vol 5, pp 3–23 This paper was originally written in 1948 but was not previously published
3. Applied robotics for industrial automation has not been so closely related to Artificial Intelligence.
4. P. R. Cohen, *AI Magazine* 12, 16 (1991).
5. N. J. Nilsson, Ed., *Technical Note No. 323* (SRI International, Menlo Park, CA, 1984). This is a collection of papers and technical notes, some previously unpublished, from the late 1960s and early 1970s.
6. P. H. Winston, in *Machine Intelligence*, B. Meltzer and D. Michie, Eds. (Wiley, New York, 1972), vol. 7, pp. 431–463.
7. T. Winograd, *Understanding Natural Language* (Academic Press, New York, 1972).
8. E Charniak and D McDermott, *Introduction to Artificial Intelligence* (Addison-Wesley, Reading, MA, 1984)
9. D. Marr, *Vision* (Freeman, San Francisco, 1982)
10. R. A. Brooks, *Model-Based Computer Vision* (UMI Research Press, Ann Arbor, 1984)
11. W. E. L. Grimson, *Object Recognition by Computer: The Role of Geometric Constraints* (MIT Press, Cambridge, MA, 1990).
12. C K. Yap, in *Advances in Robotics*, J. T. Schwartz and C. K. Yap, Eds (Lawrence Earlbaum, 1985), vol. 1.
13. M. Brady, J. Hollerbach, T Johnson, T. Lozano-Pérez, M. Mason, Eds., *Robot Motion: Planning and Control* (MIT Press, Cambridge, MA, 1982).
14. R. A. Brooks, *Int. J. Robotics Res.* 1, 29 (1982).
15. R. Chatila and J.-P Laumond, in *Proceedings of the IEEE Conference on Robotics and Automation*, St. Louis (IEEE Press, New York, 1985), pp. 138–143
16. T. Lozano-Pérez, M. T. Mason, R. H. Taylor, *Int J. Robotics Res* 3, 3 (1984)
17. T. Lozano-Pérez and R. A. Brooks, in *Solid Modeling by Computers*, M. S. Pickett and J. W. Boyse, Eds. (Plenum, New York, 1984), pp. 293–327.
18. T. Lozano-Pérez, J. L. Jones, E. Mazer, P. A. O'Donnell, *Computer* 22, 21 (1989).
19. H. P. Moravec, *Proc. IEEE* 71, 872 (1982).
20. R. Simmons and E. Krotkov, in *Proceedings of the IEEE Robotics and Automation*, Sacramento (IEEE Press, New York, 1991), pp. 2086–2091
21. P. Agre and D. Chapman, in *Proceedings of American Association of Artificial Intelligence*, Seattle (Morgan Kaufmann, Los Altos, CA, 1990), pp 268–272
22. ———, in *Designing Autonomous Agents*, P. Maes, Ed. (MIT Press, Cambridge, MA, 1990), pp 17–34.
23. S J. Rosenschein and L. P. Kaelbling, in *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge*, J. Halpern, Ed. (Morgan Kaufmann, Los Altos, CA, 1986), pp. 83–98.
24. L. P. Kaelbling and S. J. Rosenschein, in *Designing Autonomous Agents*, P. Maes, Ed. (MIT Press, Cambridge, MA, 1990), pp. 35–48.
25. R. A. Brooks, *IEEE J. Robotics Automation* 2, 14 (1986).
26. ———, in *Designing Autonomous Agents*, P. Maes, Ed. (MIT Press, Cambridge, MA, 1990), pp. 3–15.
27. ———, *Artificial Intelligence* 47, 139 (1991).
28. S. Ullman, *Cognition* 18, 97 (1984).
29. M. Minsky, *The Society of Mind* (Simon and Schuster, New York, 1986).
30. I. D. Horswill and R. A. Brooks, in *Proceedings of American Association of Artificial Intelligence*, St. Paul (Morgan Kaufmann, Los Altos, CA, 1988), pp. 796–800.
31. D. H. Ballard, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit (Morgan Kaufmann, Los Altos, CA, 1989), pp. 1635–1641.
32. E. D. Dickmanns and V Graefe, *Machine Vision Appl.* 1, 223 (1988).
33. M. H. Raibert, *Legged Robots that Balance* (MIT Press, Cambridge, MA, 1986).
34. R. A. Brooks, *Neural Computation* 1, 253 (1989).
35. J. H. Connell, *Technical Report No. AIM-TR-1151* (MIT, Cambridge, MA, 1989)
36. C. Malcolm and T. Smithers, in *Designing Autonomous Agents*, P. Maes, Ed. (MIT Press, Cambridge, MA, 1990), pp. 123–144.
37. M. J. Mataric, *Technical Report No. AIM-TR-1228* (MIT, Cambridge, MA, 1990).
38. U Nehmzow and T Smithers, in *From Animals to Animats*, J.-A. Meyer and S. W. Wilson, Eds. (MIT Press, Cambridge, MA, 1990), pp. 152–159
39. D W. Payton, in *Proceedings of the IEEE Conference on Robotics and Automation*, San Francisco (Morgan Kauffman, Los Altos, CA, 1986), pp 1838–1843.
40. P. Maes, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit (Morgan Kauffman, Los Altos, CA, 1989), pp 991–997.
41. P. Maes and R. A. Brooks, in *Proceedings of the American Association of Artificial Intelligence*, Boston (Morgan Kauffman, Los Altos, CA, 1990), pp. 796–802
42. S. Mahadevan and J H Connell, *Automatic Programming of Behavior-based Robots using Reinforcement Learning* (IBM T. J. Watson Research Center, 1990).
43. L Kaelbling, thesis, Stanford University (1990).
44. C Watkins, thesis, Cambridge University (1989)
45. R C Arkin, in *Designing Autonomous Agents*, P. Maes, Ed (MIT Press, Cambridge, MA, 1990), pp. 105–122
46. T M Mitchell, in *Proceedings of the American Association of Artificial Intelligence*, Boston (Morgan Kauffman, Los Altos, CA, 1990), pp. 1051–1058.
47. P K. Maikin and S. Addanki, in *ibid.*, pp. 1045–1050.
48. R. A. Brooks, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Sydney (Morgan Kauffman, Los Altos, CA, 1990), pp. 569–595.

49. Supported in part by the University Research Initiative under Office of Naval Research contract N00014-86-K-0685, in part by the Defense Advanced Research Projects Agency under Office of Naval Research contract N00014-85-K-0124, in part by the Hughes Artificial Intelligence Center, in part by Siemens Corporation, and in part by Mazda Corporation.

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

ROBOTS MÓVILES

1.- MODELOS DE IA APLICADOS A ROBOTS MÓVILES

2.- PLANEACIÓN DE MOVIMIENTOS Y NAVEGACIÓN

EXPOSITORES : Ing. Marco Antonio Morales Aguirre
Ing. Jesús Savage Carmona

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos. 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

Robots Móviles

Marco Antonio Morales Aguirre Jesús Savage Carmona

abril de 1997

Índice General

1 Modelos de IA aplicados a robots móviles **ii**

1.1 La arquitectura de control ii
1.1.1 El enfoque tradicional iii
1.1.2 El enfoque de comportamientos iv
1.2 Búsqueda de soluciones para encontrar un camino al objetivo . . v
1.2.1 Métodos ciegos vii
1.2.2 Métodos heurísticos ix
1.3 Sistemas Expertos xiii
1.3.1 Modelo y componentes xiv
1.3.2 Entornos de desarrollo xvi
1.4 Redes Neuronales Artificiales xvii
1.5 Agentes xix

2 Planeación de movimientos y navegación **xxii**

2.1 El problema básico de planeación de movimientos xxiii
2.2 Espacio de configuraciones xxiv
2.3 Representación de los obstáculos xxv
2.4 Métodos de solución del problema básico de planeación xxvii
2 4.1 Mapas de caminos xxvii
2.4.2 Descomposición de celdas xxviii
2.4.3 Métodos de campos potenciales xxx
2.5 Restricciones cinemáticas xxxi
2.6 Incertidumbres xxxii
2.7 Navegación xxxii
2.7.1 Modelo matemático xxxiii
2.8 Pilotaje xxxvii
2.8.1 Navegación con sensores de tacto xxxvii
2.8.2 El algoritmo Bug2 xxxix
2.8.3 Navegación con sensores de proximidad xl

Capítulo 1

Modelos de IA aplicados a robots móviles

Un robot obtiene datos a través de sus sensores, los interpreta, toma decisiones y envía órdenes a sus actuadores. La "inteligencia" del robot depende de los algoritmos que le permiten interpretar los datos y tomar decisiones, por ésto el controlador del robot debe basarse en técnicas de inteligencia artificial. Desde que nació el campo de la inteligencia artificial ha existido la intención de realizar robots autónomos verdaderamente inteligentes, ésto es, robots que realizen tareas de manera similar a como las realizaría un ser humano.

Un problema que se debe enfrentar es el de la velocidad de los programas que controlan al robot. Contar con algoritmos eficientes y rápidos es prioritario, ya que por el momento, un robot no podría cargar consigo una supercomputadora.

En este capítulo se tratan algunas técnicas de inteligencia artificial que se pueden aplicar a la solución de algunos de los problemas de los robots móviles de manera eficaz.

1.1 La arquitectura de control

El controlador es el encargado de interpretar los datos que un robot recibe mediante sus sensores y de tomar decisiones para que realize los movimientos adecuados con el fin de cumplir con su tarea. La manera en que se distribuyen los componentes del controlador determina el comportamiento que tiene el robot y la velocidad a la que se pueden ejecutar las órdenes.

En los primeros días de la Inteligencia Artificial se propuso una arquitectura de controlador en la que se dividía la carga de trabajo en varios sistemas dispuestos uno tras otro: percepción, modelado del mundo, planeación y ejecución. A fines de los 80's se propuso una nueva arquitectura con varios módulos de comportamiento, cada uno de los cuales tendría sus propios mecanismos de percepción modelado y planeación, y un árbitro que decide qué módulo de comportamiento controla a qué parte del robot en un momento determinado.

1.1.1 El enfoque tradicional

Los primeros robots "inteligentes" que fueron construidos a fines de los 60's y principios de los 70's, marcaron la pauta del desarrollo en robótica al menos por los siguientes 15 años. Shakey, un robot desarrollado en el Instituto de Investigaciones de Stanford (Stanford Research Institute) contaba con un medio ambiente preparado especialmente para él, una cámara de televisión como sensor primario, una computadora externa para analizar las imágenes en base a descripciones hechas en la forma de cálculo de predicados de primer orden, que servían para generar una secuencia de acciones con un planeador que a su vez enviaba comandos a los actuadores que contaban con realimentación en base a otros sensores como el odómetro y una barra sensor de contacto.

En la mayoría de estos sistemas existía la tendencia a utilizar la visión por computadora solamente para reconstruir un mundo tridimensional a partir de imágenes bidimensionales. La inteligencia artificial era utilizada para realizar descripciones del mundo y manipularlas, que casi nunca se convertían en manipulaciones reales de los objetos reales. La robótica sólo tenía como misión la interacción física con el mundo, los problemas principales eran: la planeación de una ruta libre de colisiones; la cinemática y dinámica directa; y la cinemática y dinámica inversa. Como se puede ver en la figura 1.1 la interacción entre los componentes en el enfoque tradicional era:

1. Obtener datos de los sensores.
2. Interpretar esos datos.
3. Realizar un modelo del mundo en función de la respuesta de los sensores.
4. Planear los movimientos necesarios en función de la meta y del modelo del mundo.
5. Enviar instrucciones a los actuadores.

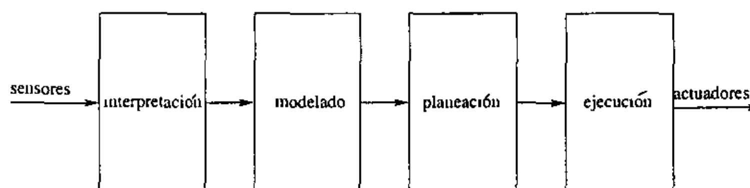


Figura 1.1: Enfoque tradicional

Este enfoque fue aplicado con relativo éxito en muchos casos, pero desgraciadamente en el momento en que se han sacado de los ambientes fabricados expresamente para la operación de esos sistemas, han fallado. Ahora es claro que es muy difícil obtener suficientes datos de los sensores que permitan realizar un modelo exacto del mundo. Además el ciclo sensado, modelado, planeación,

actuación es muy lento y puede hacer que un robot tarde varios minutos en tomar una decisión simple.

1.1.2 El enfoque de comportamientos

Alrededor de 1984, de manera independiente, varias personas comenzaron a reestructurar la organización de la inteligencia en los robots basados en los siguientes argumentos:

1. La mayor parte de las cosas que realiza la gente en su vida cotidiana no es resolver problemas o planear, sino que realizan actividades rutinarias en un mundo dinámico.

2. La representación que un agente tiene del mundo, no necesariamente debe descansar en nombrar esos objetos con símbolos que posee el agente, sino que puede estar definido en términos de las interacciones del agente con el mundo.

3. Un observador puede hablar legítimamente de las creencias y las metas de un agente, e incluso un agente no necesita manipular estructuras de datos simbólicas en el momento de actuar. Dado ésto se puede compilar una especificación simbólica del agente antes de ser usado, llevando a programas eficientes.

4. Modelos internos del mundo que sean representaciones completas de ambientes externos son imposibles de obtener y no son totalmente necesarias para que un agente actúe competentemente.

5. Para poder evaluar la inteligencia de un robot, es necesario construir agentes completos que operen en un mundo dinámico usando sensores reales.

6. La inteligencia puede emerger de componentes independientes que interactúan con el mundo.

Uno de los resultados más importantes de estas apreciaciones fue la arquitectura subsumida (subsumption architecture) que propuso Rodney Brooks [3]. Esta arquitectura está compuesta por una red de máquinas de estado finito aumentadas (AFSM's por sus siglas en inglés) generadoras de comportamientos, cada una de las AFSM's envía y recibe mensajes y almacena información respecto a los mensajes que se recibieron más recientemente. Dependiendo de los datos que cada AFSM recibe, puede enviar mensajes de salida o mensajes a otros AFSM's. Aunque la red es asíncrona, puede existir un AFSM que tenga un reloj interno para poder trabajar en el tiempo en que el mundo real se desplaza. Los mensajes de salida generados por las diversas AFSM's son arbitreados para poder manejar los conflictos que se pueden producir. En la figura 1.2 se puede ver la distribución de los módulos generadores de comportamientos.

En el enfoque de comportamientos, se aprovecha la capacidad de generación de comportamientos de cada módulo, de manera que cuando se cuenta con

varios de éstos surgen comportamientos complejos a partir de otros que son muy simples. logrando así que los sistemas aparenten inteligencia.

A partir de que fue propuesto este modelo se han realizado muchos sistemas que demuestran que es viable construir inteligencia de esta manera. Por otro lado, se ha considerado como una alternativa utilizar una mezcla de los dos enfoques aquí descritos, en éste se contaría con un sistema de alto nivel para planear secuencias de actividades de bajo nivel que a su vez son realizadas por una arquitectura de comportamientos.

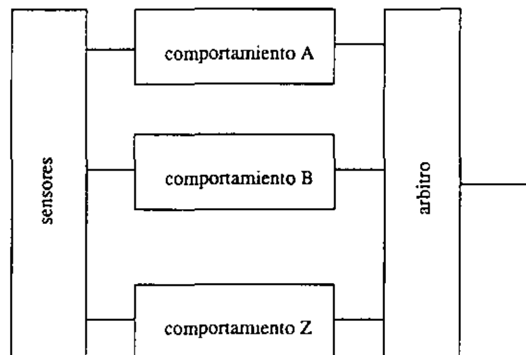


Figura 1.2: Arquitectura de comportamientos

1.2 Búsqueda de soluciones para encontrar un camino al objetivo

Para lograr que un robot móvil se coloque en una configuración predeterminada, es necesario planear la secuencia de configuraciones que debe adquirir con el fin de llegar al objetivo. Si se cuenta con un modelo adecuado del entorno en donde trabaja el robot, esto es posible. El principal componente del planeador es un mecanismo de búsqueda para encontrar un buen camino que el robot pueda seguir.

El camino óptimo es aquel que permite minimizar o maximizar algún factor que se considere importante, aunque no siempre es necesario encontrar el mejor camino, ya que puede ser suficiente encontrar un camino satisfactorio o incluso puede bastar con encontrar al menos uno.

El problema básico consiste en que dado un punto inicial y un punto final, así como un mapa de nodos y conexiones, se debe encontrar algún camino o el mejor camino (quizá el más corto) para llegar del punto inicial al punto final.

Realizar un mapa, como se verá más adelante, es un problema fundamental de la planeación de rutas. En este mapa todos los nodos son puntos en los cuales puede estar el robot sin estar superpuesto con algún objeto de su área de trabajo. Además, tanto el punto inicial como el punto final son considerados nodos del

mapa (si el punto final cumple con la condición de no superponerse con algún objeto), o de lo contrario la búsqueda no se puede realizar. Sólo existe conexión entre dos nodos si no hay ningún objeto que cruze la linea recta que los une, y el peso asociado a la conexión es igual a la distancia entre ellos. Debido a esta característica, a éste mapa se le conoce como mapa de caminos o de carreteras, y de manera más formal se puede decir que es un grafo de visibilidad. En la figura 1.3 se observa un ejemplo de grafo de visibilidad.

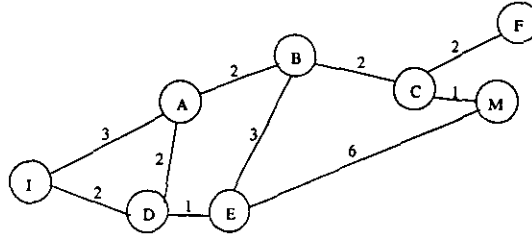


Figura 1.3: Mapa de caminos

El camino encontrado en la búsqueda es una secuencia de nodos a los cuales se puede llegar sin riesgo de tener una colisión con los objetos que se consideraron para determinar los nodos y las conexiones entre ellos. Además se pretende que éste camino sea el camino más corto o al menos uno de los más cortos.

Una forma eficiente de resolver este problema es utilizando un árbol de búsqueda, que es una colección de nodos (que podrían repetirse incluso) y ramas. Los nodos representan trayectorias y las ramas conectan trayectorias a extensiones de trayectorias logradas en un solo paso. Además tiene mecanismos para producir una descripción de trayectoria y para conectar la trayectoria con su correspondiente descripción [4].

Los árboles de búsqueda se utilizan en problemas de naturaleza diversa, en el caso de la generación de rutas para un robot se puede utilizar un árbol para representar al mapa de caminos y proceder a una búsqueda en el mismo, como en la figura 1.4 en la que se vé el árbol generado a partir del grafo de la figura 1.3

Existen diversos métodos para realizar búsqueda en un árbol. Se puede hacer una clasificación de éstos en dos tipos generales: los métodos ciegos y los métodos heurísticos. Los primeros realizan una búsqueda sin tener ninguna clase de información acerca del problema y se conforman con llegar al nodo meta, en tanto que los métodos informados aprovechan las características del problema para hacer eficiente la solución y la búsqueda.

Además de que se debe tratar de encontrar una solución eficiente al problema que se plantea, también se debe cuidar la eficiencia de la búsqueda, ya que si consideramos que si cada nodo del árbol tiene en promedio b ramas y que la profundidad del árbol es d, entonces el número total de trayectorias en él será b^d. El número de trayectorias se incrementa exponencialmente a medida que la profundidad del árbol aumenta, y por lo tanto la potencia de cómputo necesaria

7

para resolver el problema también crece.

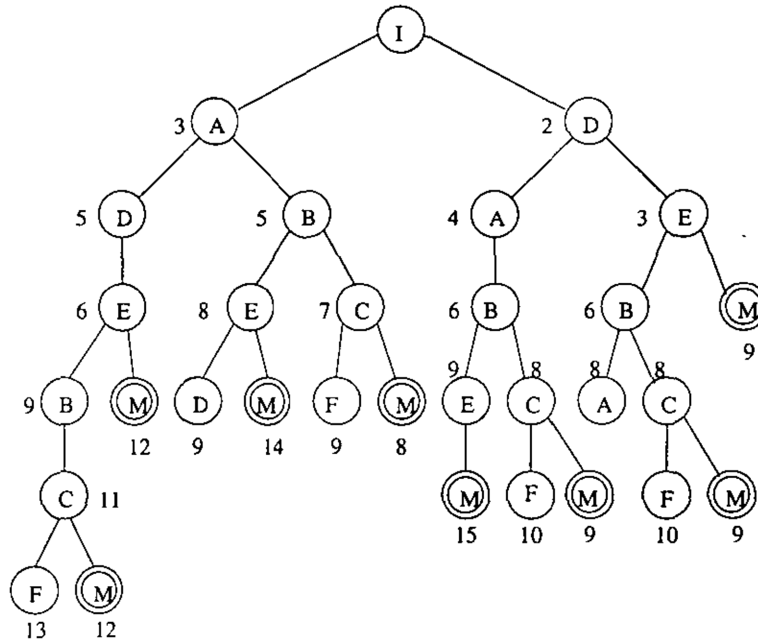


Figura 1.4: Representación del mapa de caminos de la figura 1.3 en un árbol

1.2.1 Métodos ciegos

Existen dos métodos ciegos: el de búsqueda en profundidad y el de búsqueda en amplitud. El problema principal de éstos es que si no existe solución al problema, se tiene que recorrer todo el árbol para darse cuenta, además que aún cuando haya solución la búsqueda es muy ineficiente.

El método de búsqueda en *profundidad* consiste en partir del nodo origen y crear todas las trayectorias que se puedan generar a partir de él, revisar si una de éstas trayectorias lleva al nodo destino, si nó tomar a uno de los hijos generados y repetir la búsqueda de la trayectoria a partir de él, las demás alternativas del mismo nivel se ignoran a menos que se hayan agotado las posibilidades de llegar al nodo destino a partir de la primera elección. En la figura 1.5 se puede observar un ejemplo de búsqueda en profundidad.

El método de búsqueda en amplitud, como se puede ver en la figura *amplitud* consiste en obtener todos los hijos de todos los nodos del mismo nivel, si ninguno de ellos es el nodo destino, se incrementa el nivel y se repite el proceso hasta llegar a la solución.

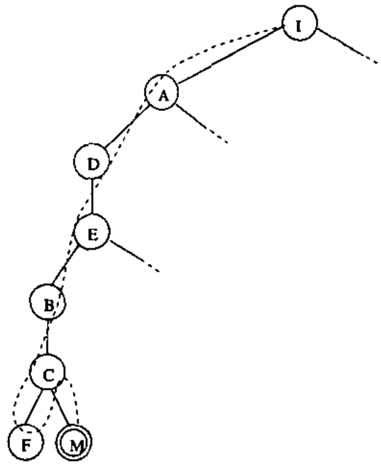


Figura 1.5: Búsqueda en profundidad

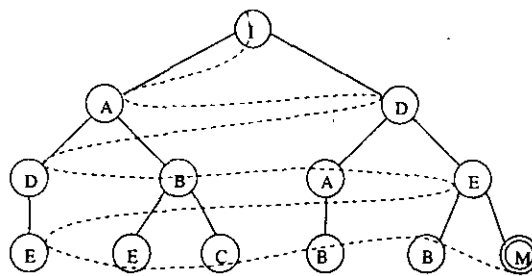


Figura 1.6: Búsqueda en amplitud

1.2.2 Métodos heurísticos

El objetivo primordial de los métodos heurísticos es aumentar la calidad de la búsqueda, utilizando una regla heurística (o de "sentido común") para reducir el número de nodos a través de los cuales ésta se tiene que realizar. Dependiendo de la naturaleza del problema la regla heurística puede variar, en el caso de planeación de rutas de robots se puede considerar la distancia en línea recta del nodo a evaluar al nodo destino. En la figura 1.7 se observa la distancia en línea recta al nodo meta M desde todos los demás.

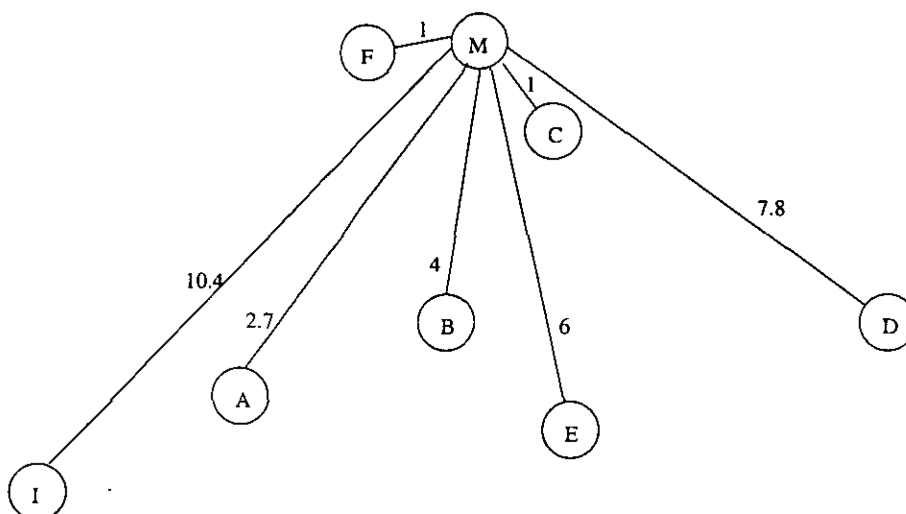


Figura 1.7: Distancias desde el nodo meta a todos los demás

Ya que la regla heurística mide qué tan bueno es un camino, estos métodos dan como resultado un buen o el mejor camino. En primer lugar se muestran los métodos de ascenso de colina, de búsqueda en haz y de búsqueda primero el mejor, los cuales son mas eficientes que los métodos ciegos y localizan un buen camino, probablemente el mejor. En segundo lugar se describe el método de la fuerza bruta, el de ramificación y cota, de ramificación y cota con estimación del límite inferior, de ramificación y cota con programación dinámica y el método A*, que permiten encontrar la mejor solución con alta eficiencia.

El método de ascenso de colina es similar al de búsqueda en profundidad, con la diferencia que para decidir cuál es el nodo que se debe expandir, se extrae de los nodos una medida heurística de la distancia que queda por recorrer hasta la meta y se expande el que proporciona la menor distancia.

El método de búsqueda en haz o búsqueda rayo, es similar a la búsqueda en amplitud, pero en lugar de expandir todos los nodos de un nivel sólo expande los w mejores. Si bien ésto puede reducir el número de nodos no ofrece aún un criterio adecuado que permita suponer que la búsqueda es muy eficiente. En la figura 1.9 se puede ver un ejemplo con $w = 3$.

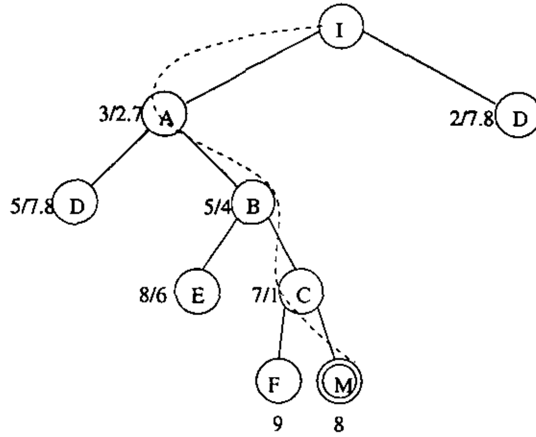


Figura 1.8: Búsqueda en ascenso de colina

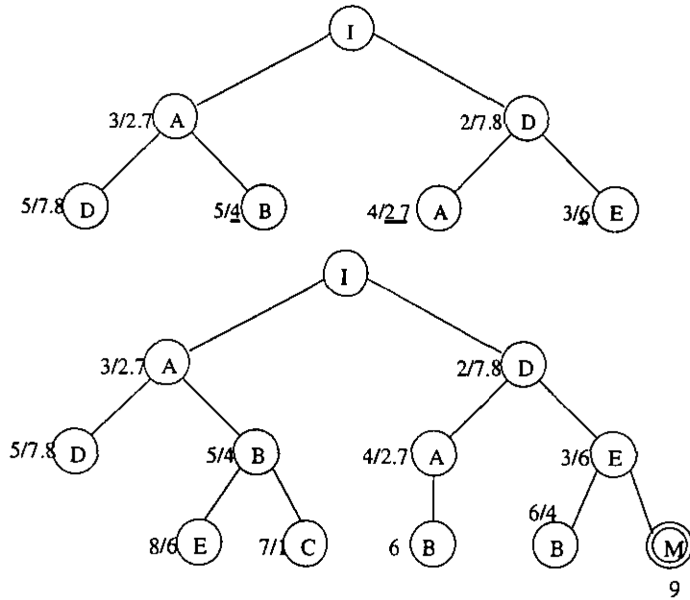


Figura 1.9: Búsqueda en haz con $w = 3$

La búsqueda primero el mejor solo expande el nodo que ofrece la menor distancia sin importar en que nivel se encuentre, ésto mejora enormemente la eficiencia de la búsqueda, e incluso es probable que la ruta hayada sea la mejor, como en el ejemplo de la figura 1.10

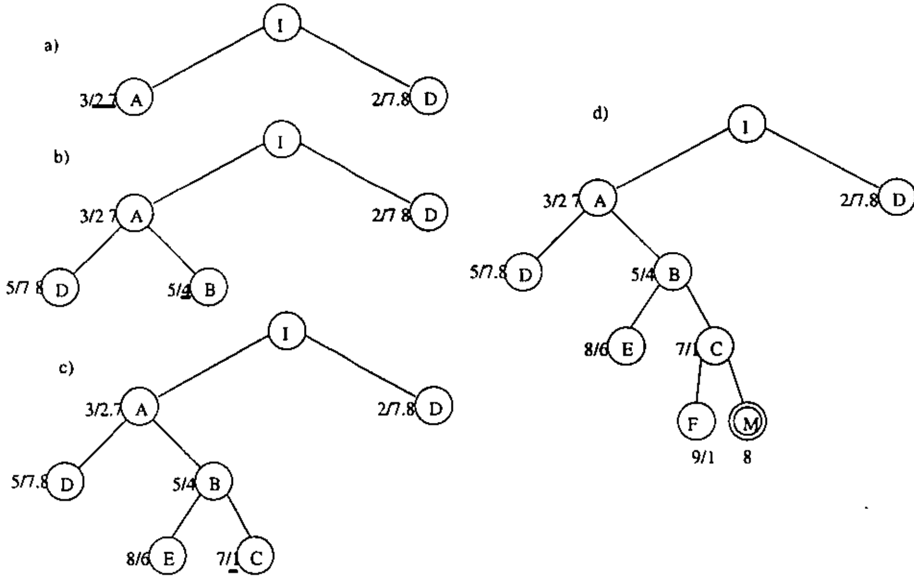


Figura 1.10: Búsqueda primero el mejor

El método de la fuerza bruta consiste en usar búsqueda en amplitud o en profundidad para encontrar todas las posibles soluciones al problema, y seleccionar la mejor de entre todas ellas, pero a un alto costo de velocidad, sobre todo en problemas que tienen una gran cantidad de nodos y alta conectividad entre ellos.

La búsqueda de ramificación y cota realiza una búsqueda similar a primero el mejor, pero no se detiene en cuanto encuentra la primera solución. A partir de encontrar la primera solución continúa expandiendo nodos con el mismo método pero jamás expande un nodo cuya distancia sea mayor o igual a la menor distancia de las soluciones ya encontradas. En la figura 1.11 se observan los primeros pasos de la búsqueda (incisos a al g) y el árbol después de haber finalizado (inciso h).

La búsqueda de ramificación y cota con estimación del límite inferior la fórmula heurística hace una estimación de la distancia total a recorrer de la siguiente manera:

$$e(longitud\,total) = d(recorrida) + e(distancia_restante)$$

Aquí el problema sería que la estimación de la distancia restante fuera mayor a la distancia restante real, por lo que se corre el riesgo de prohibir la expansión a

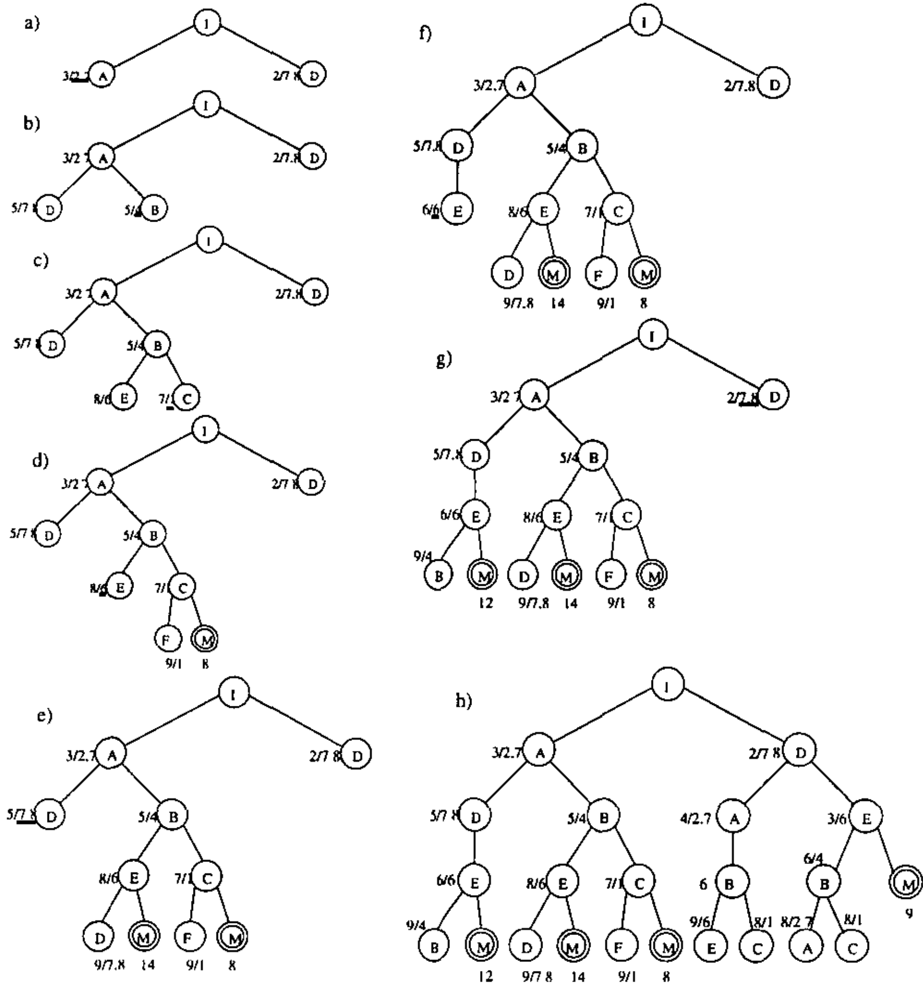


Figura 1.11: Búsqueda de ramificación y cota

nodos que lleven a la distancia óptima. Pero si en vez de hacer una estimación se hace una subestimación que garantize ser menor o igual a la distancia restante, no se corre el riesgo mencionado y sí se aumenta la eficiencia de la búsqueda. En el caso de robots una subestimación confiable es la distancia en línea recta desde el nodo en cuestión al nodo destino.

El método de ramificación y cota con programación dinámica es similar al de ramificación y cota, pero parte del principio que dice que si existen dos formas de llegar al mismo lugar y una de ellas es menos costosa que la otra , no tiene sentido seguir evaluando opciones en la mas costosa. En éste método si un nodo se repite en alguna parte del árbol, no se debe seguir expandiendo aquel que implique un costo mayor. Ésto elimina búsquedas inutiles.

El método de búsqueda $A*$ es una combinación de los métodos de ramificación y cota con estimación de límite inferior y con programación dinámica. Este método es el más eficiente de todos los que aqui se han visto, como se puede ver en la figura 1.12 en la que en sólo cuatro pasos encuentra el mejor camino.

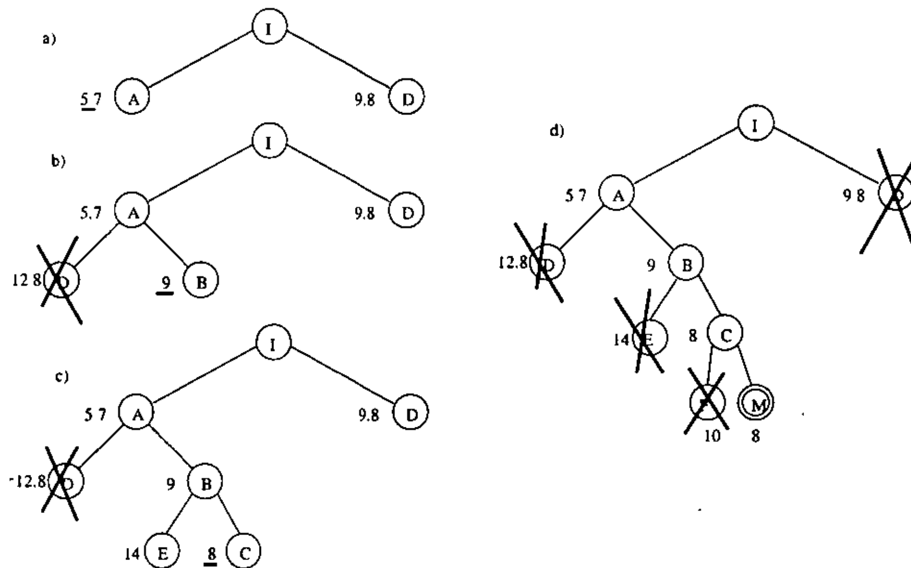


Figura 1.12: Busqueda A*

1.3 Sistemas Expertos

Un sistema experto es "un programa de computadora que utiliza conocimientos y procedimientos de inferencia para resolver problemas que son lo suficientemente difíciles para requerir capacidad experta humana para su solución" [5]. Se han propuesto varias maneras para representar el conocimiento y para implantar los procedimientos de inferencia sobre el mismo, pero el más popular es el modelo

basado en reglas, que aquí se detalla. En este modelo el sistema trata de emular la capacidad de toma de decisiones de los seres humanos utilizando reglas, que forman el conocimiento y están almacenadas en una base de conocimientos.

1.3.1 Modelo y componentes

A partir de que Newell y Simon realizaron su "solucionador general de problemas", se pudo ver que el conocimiento necesario para resolver problemas se puede expresar en gran medida en forma de *reglas de producción*, del tipo si-entonces como en:

```
si
   antecedente 1    y
   antecedente 2    y

   antecedente n
entonces
   consecuente 1
   consecuente 2

   consecuente m
```

donde $n > 0$ y $m \geq 0$.

La idea esencial es que las entradas sensoriales que recibimos, provocan estímulos que activan las reglas apropiadas en nuestra memoria de largo plazo y a partir de ahí se produce la respuesta adecuada a la situación.

Un sistema experto trabaja con hechos que pueden considerarse como parte de una memoria de corto plazo y que se almacenan en la memoria de trabajo. Los antecedentes de las reglas son, generalmente, cuestionamientos acerca de la existencia o la inexistencia de hechos que coinciden con algún patrón predefinido. Los consecuentes son acciones, que pueden ser llamadas a funciones o instrucciones para aseverar o negar hechos. En la figura 1.13 se pueden ver los componentes de un sistema experto.

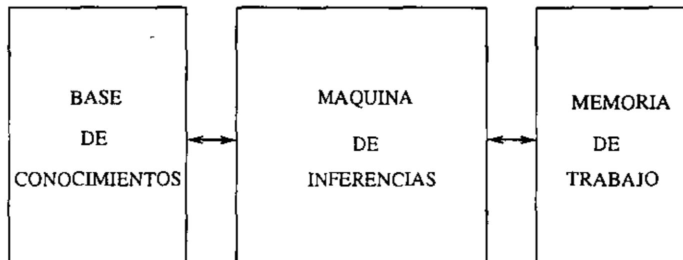


Figura 1.13: Esquema general de un sistema experto

Además de los hechos y las reglas, los sistemas expertos cuentan con un mecanismo para hacer inferencias, que es conocido como máquina de inferencias.

A través de ésta el sistema realiza inferencias tratando de emular el razonamiento que sólo es propio de los seres humanos. El proceso de inferencia consiste básicamente en encontrar las reglas válidas en cierto momento y ejecutar las acciones correspondientes a alguna de ellas.

La máquina de inferencias puede actuar de dos maneras: en la primera busca los hechos que hay en la memoria de trabajo y después verifica qué reglas de la base de conocimientos cumplen con ellos; en la segunda, busca los antecedentes de las reglas y los coteja con los hechos que hay en la memoria de trabajo. Todas las reglas que resultan del proceso anterior, se consideran reglas activas y susceptibles de ser ejecutadas, pero la máquina de inferencias tiene que elegir sólo una de ellas para ejecutarla. La elección se puede realizar tomando en cuenta cualquier criterio y al proceso de ejecución se le llama disparo.

Aunque las maneras de hacer un sistema experto pueden ser diversas, típicamente sus componentes, como se aprecia en la figura 1.14 son los siguientes [7]:

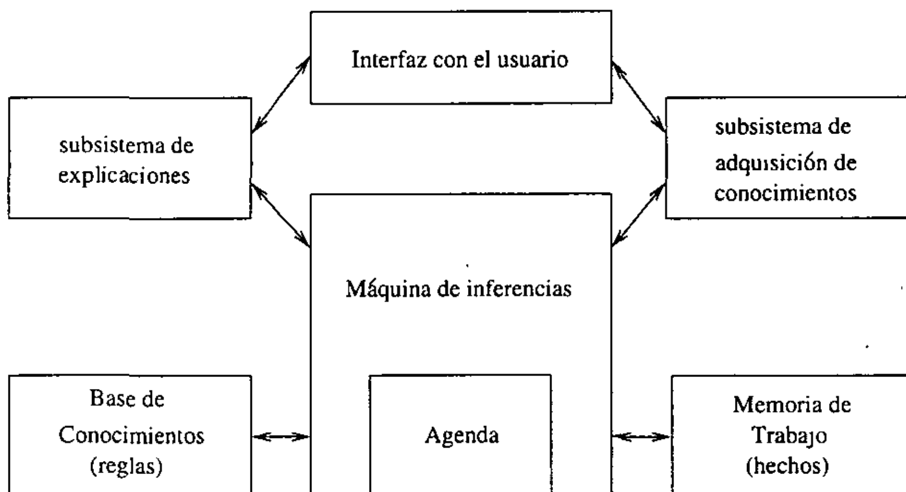


Figura 1.14: Estructura de un sistema experto basado en reglas

Interfaz con el usuario Mecanismo de comunicación entre el usuario y el sistema experto.

Subsistema de explicaciones Le explica al usuario el proceso de inferencias realizado para llegar a una conclusión.

Subsistema de adquisición de conocimientos Permite que el usuario introduzca conocimientos al sistema.

Base de conocimientos Base de datos global que almacena las reglas.

Memoria de trabajo Base de datos global que almacena los hechos usados para activar las reglas.

Máquina de inferencias Realiza inferencias para determinar qué reglas son satisfechas por hechos u objetos, de éstas ejecuta la que tiene mayor prioridad.

Agenda Es una lista priorizada de reglas creada en el proceso de inferencias, cuyos patrones son satisfechos por hechos u objetos en la memoria de trabajo y que por lo tanto son susceptibles de ejecutarse.

1.3.2 Entornos de desarrollo

Existen lenguajes de programación que permiten resolver problemas basados en la programación procedural y otros que facilitan el desarrollo de aplicaciones para inteligencia artificial, pero niguno de ellos proporciona maneras flexibles y robustas para representar el conocimiemto. Debido a ésto, se hace necesario contar con un lenguaje de programación orientado a los sistemas expertos que permita dos niveles de abstracción: la abstracción de datos y la abstracción de conocimiento.

Por otro lado, en los lenguajes procedurales existe una estrecha relación entre los datos y los métodos para manipularlos por lo cual, el programador debe ser muy cuidadoso al describir la secuencia de ejecución, en un lenguaje de sistema experto se requiere un control de ejecución mucho menos rígido debido a que los datos (hechos) y el conocimiento (reglas) están separados explícitamente. Para aplicar los conocimientos a los datos se utiliza una pieza de código totalmente distinto: el motor de inferencias. Esta separación permite un altísimo grado de paralelismo y modularidad.

A partir de los años 70's el crecimiento explosivo en el área de los sistemas expertos, condujo a la existecia de un sinnúmero de herramientas enfocadas a su desarrollo. Desde lenguajes como PROLOG que solo permiten alimentar instrucciones simples, hasta herramientas que incluso le permiten al usuario introducir el conocimiento a través de ejemplos almacenados en tablas u hojas de cálculo y que generan el código por sí mismos. Por otro lado, también se han desarrollado shells que son aplicaciones específicas a las cuales se les extrajo el conocimiento para poder darle una aplicación diferente.

Una herramienta muy útil para el desarrollo de sistemas expertos es CLIPS (C language Integrated Production System) [8]. Esta herramienta fué desarrollada en el Centro Espacial Johnson de la NASA a partir de 1984 para profundizar en el conocimiento de la construcción de herramientas para sistemas expertos. La primera versión que fué puesta a disposición de usuarios fuera de la NASA fué la 3.0 que apareció en el verano de 1986. Actualmente se usa la versión 6.0 que apareció en 1993.

CLIPS es una herramienta que cuenta con su base de conocimientos en la que se introducen los conocimientos en forma de reglas de producción. También tiene una memoria de trabajo en la que se almacenan los hechos. La máquina de inferencias que utiliza utiliza el algoritmo de Rete que hace eficiente la

búsqueda de las reglas activas. Permite introducir segmentos procedurales y de programación orientada a objetos. Cuenta con una interfaz que permite cargar reglas, hechos, funciones, objetos, monitorear el desempeño de los programas, etc. Una ventaja adicional de CLIPS es que es de muy bajo costo.

1.4 Redes Neuronales Artificiales

El principal objetivo de la Inteligencia Artificial es construir máquinas que emulen al ser humano en actividades que requieren inteligencia. Se sabe que el origen de la inteligencia humana es el cerebro y que está compuesto por millones de neuronas que están interconectadas formando una inmensa red, de ahí que resulte natural tratar de reproducirlas para producir inteligencia.

Las redes neuronales artificiales se inspiran en este modelo formando un sistema de cómputo compuesto por un grán número de elementos de proceso simples (neuronas) que están altamente interconectados. Las neuronas se organizan en capas jerárquicas y operan en forma paralela procesando información alimentada mediante entradas externas. En la mayoría de los casos los elementos de proceso son adaptables, o sea que permiten alterar su comportamiento.

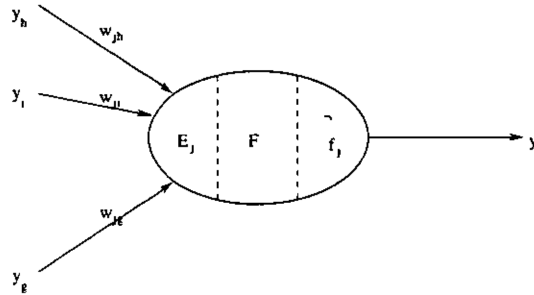


Figura 1.15: Neurona U_j

En la figura 1.15 se observa la estructura de una neurona. Del lado izquierdo se observan las entradas yi, que pueden provenir del exterior de la red o de otra neurona o incluso de su propia salida, cada entrada es ponderada por un peso w_{ji} que representa la ponderación entre la entrada i (proveniente quizá de la neurona i) a la neurona j. Para producir un resultado, lo primero que hace la neurona es sumar sus entradas ponderadas de acuerdo a

$$E_j = \sum_i y_i w_{ij}$$

después se aplica la función de activación $a_j(t+1) = F_j(a_j(t), E_j)$ para determinar el nuevo estado de activación de la neurona $a_j(t+1)$. Finalmente la salida de la neurona la determina la función $f_i(a_i(t)$ que suele ser la misma para todas las neuronas de la red. Típicamente la función de salida o de transferencia puede ser cualquiera de las siguientes:

Función escalón cuya salida es 1 si la suma de las entradas es mayor o igual al umbral de la neurona, y 0 o -1 si la suma es menor al umbral.

Función lineal y mixta en la que la salida es igual a una función lineal de la suma de las entradas si esta se encuentra entre dos límites predefinidos, es 0 o -1 si la suma es menor queel límite inferior y es 1 si la suma es mayor que el límite superior.

Función continua que puede ser cualquier función definida para todo el intervalo de posibles valores de entrada, con incremento monotónico y límites superior e inferiores. La función contínua mas usada es la sigmoidal

$$f(x) = \frac{1}{1 + e^{-\alpha x}}$$

que tiene la ventaja de que su derivada es siempre positiva y cercana a cero en valores grandes, ya sean positivos o negativos.

Función gaussiana que es una campana gaussiana típica con centro y ancho adaptables.

La principal similitud de las redes neuronales artificiales con las neuronas biológicas, es la estructura de conectividad, ya que las neuronas naturales se interconectan mediante sus axones (ramas de salida) que producen un número variable de conexiones (sinápsis) con otras neuronas o con otros componentes del organismo. Las neuronas biológicas no tienen una distribución aleatoria de conexiones, todo indica que los genes determinan que las ramas de las neuronas se dirijan hacia donde serán necesitadas posteriormente. Las redes neuronales artificiales se organizan de manera jerárquica, agrupandose en capas, como se puede ver en el ejemplo de la figura 1.16.

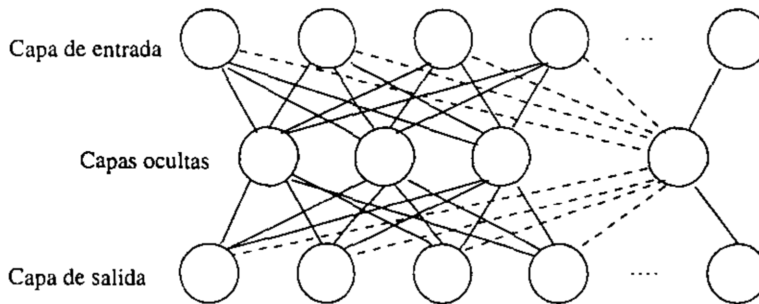


Figura 1.16: Red Neuronal

La programación de las neuronas biológicas está ligada inicialmente a los genes. Conforme se desarrollan, el peso asociado a cada una de las entradas de una neurona puede cambiar al recibir estímulos positivos o negativos cuando genera una respuesta adecuada o inadecuada ante cierta situación. En las

neuronas artificiales se simula este proceso de aprendizaje de manera similar, cambiando los pesos w_{ij} entre la salida de la neurona i y la entrada de la neurona j, siguiendo un algoritmo que deja de hacer cambios cuando la red entrega las respuestas esperadas ante estímulos predefinidos.

Las principales ventajas de las redes neuronales artificiales son:

- Capacidad de aprendizaje adaptable..
- Autoorganización.
- Tolerancia a fallas.
- Operación en tiempo real.

Las redes neuronales *aprenden* a realizar una tarea en base a ejemplos, ésta tarea puede ir desde identificar una figura hasta pronosticar un dato. En éste proceso, los pesos de los enlaces entre las neuronas se ajustan hasta que se obtienen los resultados deseados. De esta manera no es necesario encontrar la solución algorítmica del problema, ya que es la neurona la que genera su propia distribución de pesos para resolverlo.

Las redes neuronales organizan ellas mismas la información que reciben durante el aprendizaje y/o la operación. Ésto lo hacen utilizando su capacidad de aprendizaje adaptable que les permite proporcionar resultados aceptables, aún cuando se les expone a situaciones a las que nunca antes se les había sometido. Por eso, las redes neuronales se desempeñan de manera muy favorable en problemas que no permiten realizar un modelo preciso.

La autoorganización de las redes neuronales también hace que éstas sean tolerantes a fallas. Aún cuando un grupo de neuronas sufra algún daño (error en la memoria cuando están hechas a base de software, o destrucción física cuando lo están en base a hardware), la red sigue siendo funcional, aunque con cierta degradación en sus respuestas. Por otro lado, si la información que reciben está contaminada con ruido, su desempeño también se degrada, pero siguen dando respuestas confiables (en la medida en que el ruido afecte las entradas).

La naturaleza paralela de las redes neuronales hace que presenten resultados con un retardo de tiempo muy pequeño, por lo que se pueden utilizar en aplicaciones que requieren resultados en tiempo real.

Como ya se mencionó, las redes neuronales se pueden implantar utilizando circuitos electrónicos y simulaciones de software. Para contar con una red neuronal que sea verdaderamente paralela, se tiene que implantar en circuitos electrónicos, ya que si se hace con software, está sujeta a la naturaleza secuencial de los microprocesadores.

1.5 Agentes

El concepto de Agente Inteligente Autónomo es muy reciente y da respuesta a un conjunto de problemas planteados por la Inteligencia Artificial Distribuida.

Se puede decir que un agente es un dispositivo, o un programa que cuenta con las siguientes características:

Autonomía No requieren de la intervención directa de una persona o programa externo, controlan sus propias acciones y su estado interno.

Capacidad social Interactúan con otros agentes a través un protocolo de comunicación.

Reactividad Perciben su entorno y responden oportunamente a los cambios que éste presente.

Pro-actividad Tienen un comportamiento dirigido a metas en el que toman la iniciativa para cumplir con ellas.

Existen otras características que algunos autores asocian al concepto de agente, dependiendo del contexto en que sean utilizados. Por ejemplo, la Inteligencia Artificial hace énfasis en conceptos mentales, como el conocimiento o las creencias. Las principales ventajas de los agentes son que pueden planear y ejecutar acciones en base a su estado interno, a sus metas, a sus hipótesis y a sus conocimientos sobre otros agentes, pueden colaborar con otros agentes para tomar decisiones en consenso y en ocasiones permiten crear otros agentes.

El desarrollo de sistemas inteligentes que realizan un modelo del mundo en base a la información que obtienen de sus sensores, hacen planes y los ejecutan, es insuficiente cuando se enfrenta al enorme conjunto de incertidumbres que presenta el mundo real. Es por ésto que se han propuesto soluciones distribuidas que trabajen en forma cooperativa, principalmente en aplicaciones que están separadas geográficamente.

Para apoyar la toma de decisiones en sistemas distribuidos, se necesita un mecanismo de comunicación que permita que todos los componentes cumplan con su trabajo de manera eficiente. Hay dos clases de estructuras de comunicación:

Sistemas de pizarrón La comunicación se facilita aprovechando una estructura de conocimiento llamada pizarrón, en la que todos pueden colocar y leer información.

Sistemas de transferencia de mensajes Cuando un módulo requiere que otro realize o se entere de algo, le manda un mensaje. Todos los módulos tienen previo conocimiento de los nombres de los demás para poder mandar los mensajes al destinatario.

Los sistemas de pizarrón están compuestos por módulos independientes que generan conocimientos, un pizarrón que les sirve para comunicarse y un sistema de control del pizarrón, que determina la secuencia en que los generadores de conocimientos operan en el pizarrón. Este método apoya la estructura modular, pero al centralizar la comunicación con el pizarrón, propicia que el sistema se embotelle.

Los sistemas de transferencia de mensajes pueden basarse en actores o en agentes. Un actor es un objeto activo que procesa mensajes, crea nuevos actores o los activa. Las acciones del actor están definidas por un repertorio que describe su comportamiento, este puede ser: modificación de estado, actualización de lista de destinatarios, envío de mensajes o creación de nuevos actores. La comunicación entre los actores es asíncrona, por lo cual una vez que es enviado un mensaje, se pasa a procesar el siguiente sin esperar respuesta.

Capítulo 2

Planeación de movimientos y navegación

Un robot es un dispositivo que opera en el mundo físico el cual está poblado por objetos que limitan sus posibilidades de movimiento. La secuencia de acciones necesaria para concluir una tarea está determinada por la naturaleza de la misma y por los objetos que "estorban" para realizarla. Este problema es enfrentado con técnicas de planeación movimientos y de navegación. El robot puede tener la organización lógica mostrada en la figura 2.1, en la que cada componente se puede implantar mediante agentes autónomos. La misión de cada componente se lista a continuación.

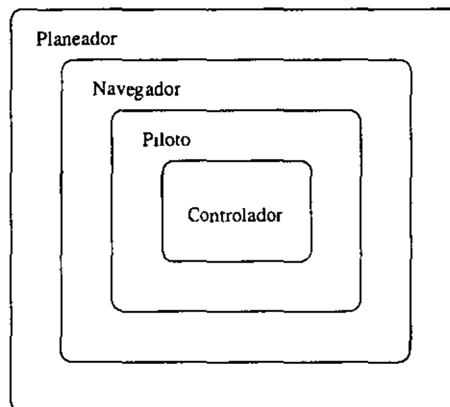


Figura 2.1: Organización lógica de un robot móvil

Planeador Determina la ruta óptima global para que el robot alcanze su configuración meta, utilizando conocimiento *a priori* del entorno del robot y

de las restricciones a las que está sujeto. También puede definir un tiempo máximo para concluir la misión.

Navegador Divide la ruta seleccionada por el planeador en segmentos independientes y encuentra un movimiento contínuo para seguirlas. También se encarga de evitar obstáculos detectados con sus sensores.

Piloto Sigue la ruta definida por el navegador, ligando la variable tiempo a la misma. La combinación de una ruta y "marcas de tiempo sobre la misma da como resultado una trayectoria. Para generar la trayectoria se debe considerar la dinámica del robot.

Controlador Obedece al piloto para que los motores cumplan con los movimientos esperados y manipula a los sensores.

Se puede ver que este esquema cumple con las características del enfoque tradicional del que se habló en el capítulo anterior, pero se puede adaptar para operar en un esquema que aproveche las ventajas del enfoque tradicional y del de comportamientos, como se muestra en este trabajo. En éste capítulo se tratará el problema del planeador y del navegador.

2.1 El problema básico de planeación de movimientos

En el problema básico de la planeación de movimientos de robots se considera que el robot es el único objeto en movimiento y se ignoran sus propiedades dinámicas, con el fin de evitar consideraciones temporales. Se considera que los movimientos que se realizan están libres de contacto, evitando así modelar la interacción mecánica que se podría dar. En resumen, el problema de planear movimientos físicos se transforma en un problema de planeación de rutas puramente geométrico conformado por los siguientes elementos:

- El robot A que se mueve en un espacio Euclidiano W, llamado espacio de trabajo, representado como R^N, donde $N = 2o3$.
- Los obstáculos $B_1, \ldots, B_q$ que son objetos rígidos fijos en W.
- La geometría de $A, B_1, \ldots, B_q$ y la posición de los $B_i's$ en W se conocen con precisión
- No existen restricciones cinemáticas que limiten los movimientos de A (el robot es un objeto libre).

El problema, como dice Jean-Claude Latombe [6], es: Dada un posición y orientación inicial y una posición y orientación meta del robot A en el espacio W, se debe generar una ruta τ que especifique una secuencia continua de posiciones y orientaciones del robot A que eviten el contacto con los obstáculos $B_i's$, partiendo de la posición y orientación iniciales y terminando en la posición y orientación metas. Si no existe ruta se reporta la imposibilidad de resolver el problema

2.2 Espacio de configuraciones

El espacio de configuraciones es un concepto introducido por Lozano Pérez como una herramienta de representación. En ésta el robot es tratado como un punto en un espacio llamado el *espacio de configuraciones del robot*. Los elementos geométricos y conceptos físicos se mapean como construcciones geométricas. Ésto permite formular el problema de planeación de movimientos con precisión y transformarlo en el problema más sencillo de planear los movimientos de un punto y no de un objeto dimensionado.

Tanto el robot A como los obstáculos $B_1, \ldots, B_q$ son subconjuntos cerrados del espacio W. F_A y F_W son planos cartesianos ligados al robot A y a W, F_A es un plano cartesiano que se mueve junto con A, y aunque todos los puntos pertenecientes al robot están fijos con respecto a F_A, su posición en W depende de la posición de F_A con respecto a F_W. Por su parte los $B_i's$ se consideran rígidos y fijos en el espacio W, por lo que tienen una posición fija con respecto a F_W.

La configuración de un objeto es una especificación de la posición de todos sus puntos con respecto a un marco de referencia fijo. De aquí se deduce que una configuración q del robot A es la posición τ y la orientación θ de F_A (el plano ligado al robot) con respecto a F_W (el plano ligado al espacio W). El *espacio de configuraciones* de A es el espacio C de todas las configuraciones que puede adquirir A. Al subconjunto de W ocupado por el robot A en la configuración q se le denota $A(q)$, y al punto a de A en la configuración q se le denota $a(q)$ en el espacio W.

Hay varias maneras de representar una configuración. Por ejemplo, para la posición se puede utilizar el vector de N coordenadas que representa al origen de F_A en F_W y la orientación se puede describir como una matriz de NxN cuyas columnas son las proyecciones de los vectores unitarios de los ejes de F_A en FW. La descripción aquí ejemplificada es redundante y se puede minimizar, pero ésto no siempre es conveniente, ya que aunque el espacio de configuraciones C se parece a un espacio de $\Re^m$, formalmente no lo es, dado que se descompone en una unión de copias ligeramente diferentes de R^m, entre las cuales se va desplazando el robot, dependiendo del tipo de movimiento que realice.

Para establecer los movimientos que debe realizar un robot para trasladarse de una configuración determinada a otra, se necesita seguir una ruta. Una ruta para el robot A desde $q_{inicial}$ hasta q_{meta} es una función contínua

$$\tau : [0, 1] \to C$$

con $\tau(0) = q_{inicial}$ y $\tau(1) = q_{meta}$. $q_{inicial}$ y q_{meta} son, respectivamente, las configuraciones inicial y meta del la ruta. Para que se pueda mover al robot, es necesario que exista una conexión entre al menos dos configuraciones.

En esta representación las fuerzas que ejercen los objetos no juegan ningún papel, ya que el problema es puramente geométrico, aunque un método que se discutirá mas adelante utiliza fuerzas artificiales para guiar la planeación.

2.3 Representación de los obstáculos

Los obstáculos $B_i, i = 1, 2, \ldots, q$ que están en W, se representan como regiones cerradas, aunque no necesariamente limitadas, de $\Re^N$. Estos $B_i's$ denotan tanto a los objetos físicos como a las áreas que los representan. Al mapeo de los obstáculos B_i en el espacio de configuraciones C se le llama *obstáculo-C*, y se define como la región en C que cumples con

$$CB_i = \{q \in C \mid A(q) \cap B_i \neq 0\}$$

Por ejemplo, si se tiene un robot cilíndrico que se mueve en un espacio de 2 dimensiones, y un obstáculo poligonal B_i, CB_i se obtiene haciendo que el área del obstáculo crezca isotrópicamente por el radio de A, como se ejemplifica en la figura 2.2.

Esta representación permite establecer rutas solamente en las partes del espacio W cuya intersección con los *obstáculos-C* es vacía. De la definición de $CB_i's$ se observa que para construirlos se debe considerar la forma tanto del robot A como de los obstáculos B_i.

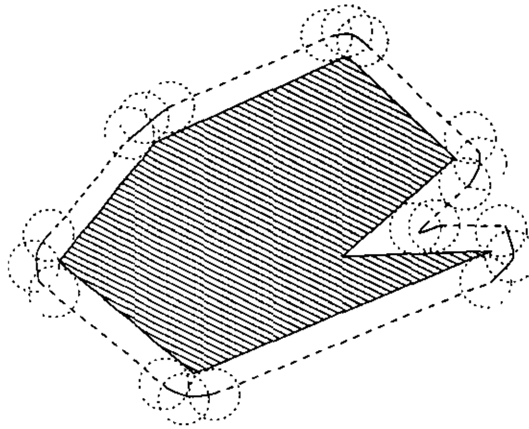


Figura 2.2: Expansión de un obstáculo polígonal para un robot cilíndrico

Primero se considera el caso en que el robot y los obstáculos son poligonales, y que el desplazamiento se realiza en un espacio de dos dimensiones, además el robot solo se puede trasladar, no girar. En este caso, las $CB_i's$ se forman a partir del área de contacto que se forma entre la superficie del robot y la de los polígonos, si se le pone a éste a desplazarse libremente sobre toda la superficie de trabajo sin que se presente alguno de los contactos que se listan a continuación:

Contacto tipo A que se dá cuando los interiores de A y B no se sobreponen.

Contacto tipo B que se dá cuando un vértice del robot está en el borde de de B sin sobreponerse a su interior.

2

Si se le permite girar al robot, se forma un espacio de tres dimensiones formado por planos. Cada plano se forma con una orientación del robot fija, por lo que entre plano y plano solo hay un diferencial de orientación.

Cuando el robot se desplaza en un espacio de tres dimensiones, se representan los obstáculos mediante polihedros. Las regiones $CB_i's$ son de seis dimensiones que están limitadas por superficies de cinco dimensiones generadas por el contacto que se dá entre la superficie del robot. En este caso los contactos pueden ser:

Contacto tipo A que se dá entre una cara del robot y un vértice de B.

Contacto tipo B entre un vértice de A y una cara de B.

Contacto tipo C entre un borde de A y un borde de B.

El mecanismo que se sigue para formar las $CB_i's$ es similar al que se sigue en los espacios de dos dimensiones.

Los casos anteriores son casos particulares que se pueden generalizar si se representan los obstáculos y los objetos como subconjuntos semialgebraicos de $W = \Re^N$. Un conjunto semialgebraico se define mediante sentencias de lógica de primer orden cuyos predicados son $=, \neq, >, <, \geq$ y $\leq$, las variables son números reales y los términos son polinomios multivariables con coeficientes reales [6].

Si el robot y el conjunto finito de obstáculos se representan como subconjuntos semialgebraicos de $\Re^N$, todos los *obstáculos-C*, las áreas libres, las áreas de contacto y las áreas válidas son subconjuntos semialgebraicos de $\Re^{N(N+1)}$. Esta representación es en si misma una herramienta adecuada para operar con los objetos y determinar los caminos viables para un robot de manera eficiente. En la figura 2.3 se observan los elementos del espacio de configuraciones creado para un robot cilíndrico en un entorno poblado por polígonos y que solo puede tener desplazamientos y rotaciones en un plano. Ahí se puede ver como se obtuvieron los *obstáculos-C*, por lo que el robot A ya se puede considerar solamente un punto que se mueve en el espacio C que es el área que está dentro del rectángulo y fuera de los *obstáculos-C*.

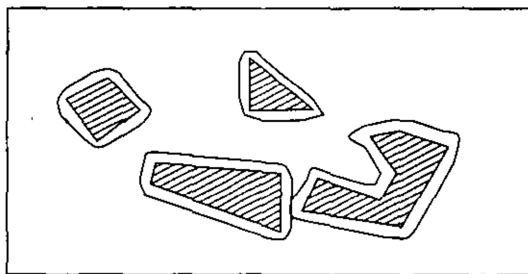


Figura 2.3: Espacio de configuraciones para un robot cilíndrico y obstáculos poligonales

2.4 Métodos de solución del problema básico de planeación

Una vez que se cuenta con una representación confiable del entorno en el que se desempeña el robot, se puede pasar al problema de planeación en si. Como ya se dijo se trata de encontrar una secuencia de rutas libres de colisión para alcanzar la meta. El planeador recibe las configuraciones inicial y meta y el tiempo para ejecutar el traslado, se conocen las restricciones del robot, aceleración y velocidad, se determinan las restricciones de posición y se planea la ruta respetando algún principio de optimización.

Existen varias técnicas para resolver este problema, a continuación se describen los enfoques mas estudiados que son el de mapas de caminos, el de descomposición de celdas y el de campos potenciales.

2.4.1 Mapas de caminos

La idea subyacente en el método de *mapa de caminos* es capturar la conectividad del espacio libre para el robot en forma de una red de curvas de una dimensión, a esta red se le llama también R. Una vez construido, se utiliza como un conjunto de rutas estandarizadas. La planeación de rutas se reduce a conectar las configuraciones inicial y meta al mapa, y buscar en él un camino.

Una forma de construir el mapa de caminos es mediante el grafo de visibilidad. Este método se aplica a espacios de configuraciones de dos dimensiones con obstáculos poligonales. El *grafo de visibilidad* es el mapa de caminos, se construye conectando todos los vértices de los límites del área libre mediante un segmento de recta, siempre que ésta no pase por el interior de algún *obstáculo C*. En la figura 2.4 se observa un ejemplo del uso del grafo de visibilidad.

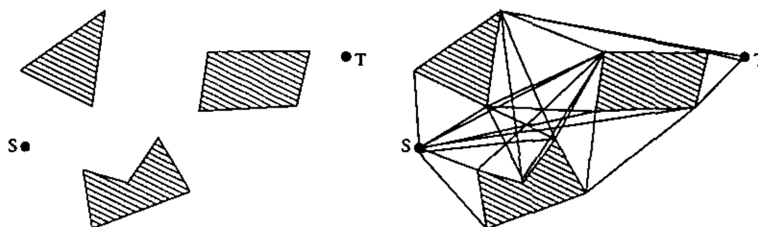


Figura 2.4: Grafo de visibilidad

Otra forma es definir una función continua de el espacio libre para formar R. Cuando el espacio es de dos dimensiones y con *obstáculos-C* poligonales, la función resultante se puede representar, por ejemplo, con el diagrama de Voronoi del espacio libre.

El diagrama de Voronoi (figura 2.5) es un subconjunto unidimensional del espacio libre que maximiza el espacio que queda entre el robot y los obstáculos. Usándolo se logra retraer el área libre hasta obtener solo una pequeña parte de

la misma. Dado ésto, un nombre que se le puede dar a este enfoque es el de enfoque de retracción.

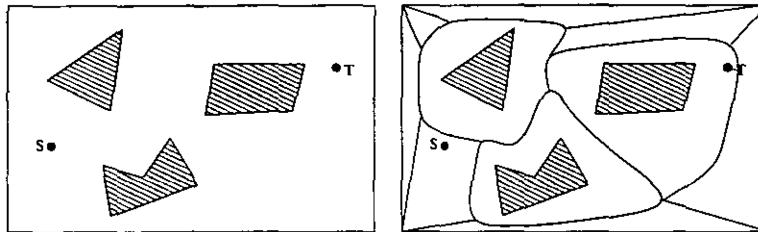


Figura 2.5: Diagrama de Voronoi

Otro método es el de vía rápida o *freeway*, que es similar al de retracción. Se usa en robots poligonales que se trasladan y giran entre obstáculos también poligonales. Se restringe al punto de referencia del robot a estar sobre una red de segmentos de recta que es similar al diagrama de Voronoi.

Un método más, conocido como algoritmo de mapa de carreteras resuelve el problema de planeación en un tiempo exponencial con respecto a la dimensión del espacio de configuraciones, se basa en construir la silueta del espacio libre del robot, añadirle segmentos curvos que ligan "puntos críticos" de la misma con otros segmentos curvos. El mapa de carreteras se forma por la silueta y los segmentos curvos.

2.4.2 Descomposición de celdas

Los métodos de descomposicion de celdas se basan en descomponer el área libre del espacio de configuraciones en regiones. Hay dos formas de obtener la descomposición: la descomposición exacta de celdas y la descomposición aproximada de celdas.

Descomposición exacta de celdas

El enfoque de descomposición exacta de celdas se basa en descomponer el espacio libre en una colección de regiones cuya intersección es vacía y su unión es exactamente el área libre. Para encontrar un camino entre dos puntos se construye el grafo de conectividad que representa la adyacencia entre regiones, y a partir de ahí se realiza la búsqueda.

Si la búsqueda es exitosa, produce un canal formado por una serie de celdas adyacentes que contienen en sus extremos a las celdas con la configuración inicial y final. De esta serie se extrae la ruta que el robot debe seguir.

Un aspecto muy importante de este enfoque es la realización de las celdas, ya que se puede dar el caso de que al definirlas no sean lo mas adecuado para encontrar un camino. Por ejemplo si se decide hacer que el espacio entero del área libre se "descomponga" en una celda simple, no es sencillo realizar la

planeación. Por esto se considera que las celdas generadas deben cumplir al menos con las siguientes características:

- La geometría de cada celda debe ser lo suficientemente simple como para que sea sencillo encontrar una ruta entre dos configuraciones en la misma celda.
- Debe ser fácil evaluar la adyacencia entre cualesquiera par de celdas y encontrar una ruta que cruce la frontera entre dos celdas adyacentes.

Cuando el espacio de configuraciones es bidimensional y los obstáculos son poligonales, se puede descomponer el espacio libre en trapezoides. Otra forma es modelar al robot como un segmento de línea que se traslada y gira entre obstáculos poligonales. En este caso el espacio de configuraciones es de tres dimensiones y los obstáculos son regiones de tres dimensiones.

Un ejemplo de descomposición exacta de celdas se puede ver en la figura 2.6, en el cual el espacio libre se descompone en espacios de los cuales se puede hacer un grafo de adyacencias, y la búsqueda de un camino se reduce a una búsqueda de adyacencias.

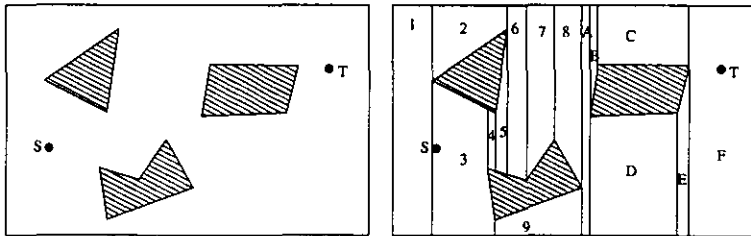


Figura 2.6: Descomposición exacta de celdas

Descomposición aproximada de celdas

La descomposición aproximada de celdas representa el espacio libre en celdas de una geometría rectangular que pueden no representar con exactitud el espacio libre, se realiza una aproximación conservativa del espacio libre. De la misma manera que en la descomposición exacta de celdas se realiza un grafo de conectividad para representar la adyacencia entre regiones y se realiza la planeación de movimientos a partir del grafo.

Las ventajas de este enfoque sobre el de descomposición exacta son: se simplifica la descomposición, ya que se obtiene de iteraciones sobre el mismo algoritmo; y los resultados no son sensibles a aproximaciones numéricas de cómputo. Como consecuencia, la descomposición aproximada es más sencilla de implantar que los métodos exactos. Además, se puede controlar la cantidad de espacio libre alrededor de los objetos fijando la dimensión mínima de las celdas.

Las principales desventajas de no obtener una caracterización exacta del espacio libre son: se puede fallar para encontrar un camino, aún cuando exista; se pierde la caracterización de discontinuidades en las restricciones de movimientos.

El desempeño de este método es muy bueno cuando la dimensión es pequeña, por ejemplo, en el caso de un robot que se mueve sonbre un plano con obstáculos poligonales, pero el grado de complejidad de las búsquedas crece exponencialmente conforme la dimensión del espacio de configuraciones aumenta.

La descomposición se realiza, como se aprecia en la figura 2.7 iterativamente partiendo de una celda cuya área es igual a la del espacio de configuraciones, ésta se divide en partes iguales. Se verifica si las áreas resultantes quedan completamente libres de obstáculos, si no es así, se les aplica el mismo mecanismo. Este proceso se repite hasta que se llege a áreas de una dimensión previamente determinada. Las áreas que quedaron totalmente libres de obstáculos se usan para hacer el grafo de conectividad.

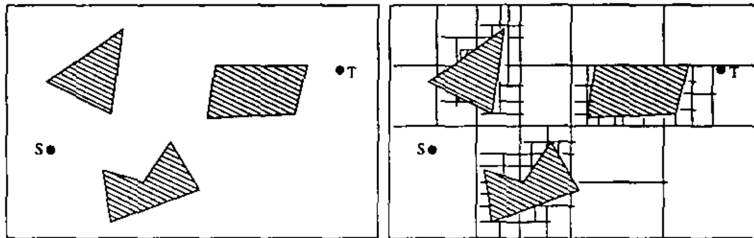


Figura 2.7: Descomposición aproximada de celdas

2.4.3 Métodos de campos potenciales

A diferencia de los métodos descritos con anterioridad, los métodos de campos potenciales no tratan de obtener la conectividad que hay en el espacio libre del robot. La idea es representar al robot como una partícula que sufre la influencia de un campo potencial U cuyas variaciones locales reflejan la estructura del espacio libre.

Típicamente, la función potencial sobre el espacio libre se define como la suma de un potencial de atracción que empuja al robot hacia la configuración meta y de un potencial repulsivo que aleja al robot de los obstáculos. La planeación de movimientos se hace a través de iteraciones, en cada una de las cuales se induce una fuerza artificial

$$\vec{F}(q) = -\vec{\nabla}(q)$$

que se considera como la dirección de movimiento más conveniente, de manera que se genera una ruta en esa dirección con un incremento.

Originalmente, el método de campos potenciales se utilizó con fines de navegación, para evitar obstáculos cuando no se tiene un modelo previo de los

mismos, tratando de generar movimientos de manera eficiente más que llegar a la meta. De hecho y debido al mecanismo de optimización que tienen, es probable caer en un mínimo local distinto a la configuración meta. De cualquier modo, se puede combinar la técnica de campos potenciales con técnicas de búsqueda en grafos para realizar planeación de movimientos.

Para tratar con los mínimos locales se debe intentar definir una función potencial que no los tenga o que tenga muy pocos, además se puede agregar al método técnicas para escapar de ellos.

La función potencial se define de manera que su valor en determinada configuración no dependa de la forma o de la distribución de los obstáculos mas allá de una distancia alrededor de la configuración, por ésto también se les conoce como métodos locales, a pesar de que se considera información que puede ser global, como la configuración meta, para su definición.

La principal desventaja de los métodos de campos potenciales es que pueden fallar y no encontrar una ruta, aún cuando ésta exista, pero por otro lado, son muy rápidos para llegar a una solución. De esta forma es posible construir planeadores eficientes y razonablemente confiables.

2.5 Restricciones cinemáticas

En las secciones anteriores se han descrito métodos de planeación de rutas considerando al robot como un objeto que se puede mover libremente, sin embargo en la realidad los robots están sujetos a restricciones cinemáticas que les impiden trasladarse y rotar con libertad. Por ejemplo, si un robot está compuesto por eslabones articulados, éstos le impiden realizar con libertad algunos movimientos. Otro caso es el de un robot móvil con estructura similar a la de un carro, aunque se puede desplazar y girar, no se puede mover hacia los lados, y los giros que realiza están limitados por el rango de su volante.

Las restricciones que afectan a un robot articulado se pueden representar mediante ecuaciones que relacionan los parámetros de configuración y que se pueden usar para eliminar algunos parámetros y reducir la dimensión del espacio de configuraciones. A estas ecuaciones se les llama *restricciones holonómicas.*

La planeación de rutas cuando el robot está sujeto a restricciones holonómicas es prácticamente igual que la que se realiza en un robot "libre". El efecto de estas restricciones afecta la construcción del espacio de configuraciones, e incluso en ocasiones lo simplifica. Por ejemplo, si se sabe que un robot tendrá orientación constante, no es necesario incluir en el espacio de configuraciones todas las posibles orientaciones para él y el espacio reduce su dimensión en uno. La mayoría de las veces las restricciones holonómicas que se expresan en una ecuación reducen la dimensión del espacio, y las inecuaciones eliminan parámetros.

Las restricciones que afectan los movimientos de un robot tipo coche se pueden convertir en una ecuación y una desigualdad que involucran los parámetros de velocidad del robot, reduciendo la dimensión del espacio de velocidades del mismo. A éstas se les llama *restricciones no holonómicas.*

Una técnica de planeación de movimientos en robots sujetos a restricciones

no holonómicas construye primero un camino libre sin considerarlas y lo transforma en un camino libre factible topológicamente, en la que se incluyen los movimientos necesarios para poder ir adquiriendo las configuraciones parciales. El problema es que se pueden generar demasiados movimientos. Existen otras técnicas que han sido diseñadas específicamente para algún tipo de robot.

2.6 Incertidumbres

En las secciones anteriores se presentaron métodos que se basan en consideraciones que en entornos reales son difíciles de encontrar. Por ejemplo, la mayoría de los objetos con los que interactuamos diariamente no son polihedros, además que identificar las características de un objeto a través de sensores no es sencillo.

Por otro lado se tiene al control del robot. Un robot móvil rara vez se desplaza en pisos lisos y perfectamente planos, por el contrario, sus movimientos se realizan en la mayoría de las veces en pisos rugosos. Para tener información acerca de la posición en la que se encuentra el robot, generalmente se confía en encoders que informan a un sistema sobre el número de veces que ha girado el eje de un motor. Esta información se utiliza para estimar la distancia que el robot se ha desplazado y la dimensión de sus giros, pero si los planes de movimientos se realizan asumiendo un piso liso, la información de los encoders no pasa de ser un referencia insuficiente para determinar en todo momento la posición del robot. Si además se fijan en la planeación las velocidades y aceleraciones de los motores como un aspecto crucial, el control se vuelve aún más complejo.

El problema básico de planeación tampoco involucra la posible presencia de objetos en movimiento. Si éstos son otros robots, se pueden involucrar en la planeación todos los robots que se piense estarán y considerarlos a todos un solo robot articulado, de esta manera los movimientos de todos los robots estarán coordinados. Pero muchas veces no es posible hacer ésto y se deben establecer mecanismos de comunicación. Si los objetos en movimiento no son controlables el problema se complica aún más, por ejemplo si de pronto se cruza enfrente del robot una persona, o se encuentra un objeto que no se había considerado en la planeación. Incluso si se hace una nueva planeación, el sistema se puede volver tan lento al grado de que no interese que un robot realize la tarea.

Dada la gran cantidad de incertidumbres con que tiene que lidiar un robot y el tiempo de procesamiento que se consume en la planeación, se piensa que distribuirla daría mejores resultados que centralizarla.

2.7 Navegación

Como se dijo al principio del capítulo, el navegador se encarga de encontrar un movimiento contínuo que lleve al robot de una posición a otra a través de una ruta geométrica localmente definida, y de evitar los obstáculos localizados con los sensores y que no fueron previstos por el planeador. Además se debe considerar el tiempo necesario para realizar los movimientos y utilizar el menor

posible. Debido a ésto es frecuente que el navegador genere rotaciones angulares para los giros y líneas rectas para las rutas.

2.7.1 Modelo matemático

La forma mas simple de realizar la navegación es dividir la ruta en una secuencia de lineas rectas. Si se piensa en que el robot está fijo con respecto al plano F_A, y que éste a su vez tiene una configuración q_{ini} y se le quiere mover a la configuración q_{meta}, ambas con respecto al espacio W, siguiendo una línea recta, es necesario encontrar la configuración q_{meta} con respecto a q_{ini} para determinar los movimientos necesarios para alcanzar q_{meta}.

Las $q's$ están formadas por una posición y una orientación. Una posición se puede describir mediante el vector que se forma entre el origen del sistema de referencia y el punto del que se quiere describir su orientación:

$$^wP = \begin{pmatrix} ^wP_x \\ ^wP_y \\ ^wP_z \end{pmatrix} \tag{2.1}$$

donde ^wP_i es la proyección del punto P sobre el eje i del sistema de referencia W.

Como las $q's$ representan un sistema de referencia del robot, que está ligado al mismo, su orientación cambia conforme el robot gira. Así que la orientación de q_i con respecto a W se representa de la siguiente manera:

$$^w_{q_i}R = \begin{bmatrix} ^w\hat{x}_{q_i} & ^w\hat{y}_{q_i} & ^w\hat{z}_{q_i} \end{bmatrix} \tag{2.2}$$

o como:

$$^w_{q_i}R = \begin{bmatrix} \hat{x}_{q_i}\hat{x}_w & \hat{y}_{q_i}\hat{x}_w & \hat{z}_{q_i}\hat{x}_w \\ \hat{x}_{q_i}\hat{y}_w & \hat{y}_{q_i}\hat{y}_w & \hat{z}_{q_i}\hat{y}_w \\ \hat{x}_{q_i}\hat{z}_w & \hat{y}_{q_i}\hat{z}_w & \hat{z}_{q_i}\hat{z}_w \end{bmatrix} \tag{2.3}$$

$^w_{q_i}R$ debe ser una matriz ortonormal, por lo que:

$$Det(^w_{q_i}R) = 1$$

y

$$^w_{q_i}R^{-1} = ^w_{q_i} R^T = ^{q_i}_w R$$

Para tener la descripción completa de q_i con respecto a W se necesita la posición del origen de q_i y su orientación con respecto a W:

$$^w\{q_i\} = \{ \; ^w_{q_i}R \quad ^wP_{org(q_i)} \; \} \tag{2.4}$$

Si se conocen las coordenadas de un punto con respecto a una referencia dada y se quieren obtener las coordenadas del mismo punto pero con respecto

a otra referencia, se necesitan transformaciones. Una matriz de transformación se define como una matriz ${}^{A}_{B}T$ tal que

$$^{A}P = ^{A}_{B}T\, ^{B}P$$

Si la transformación se quiere invertir, se hace

$$^{B}P = ^{B}_{A}T\, ^{A}P = ({}^{A}_{B}T)^{-1}\, ^{A}P$$

Por otro lado:

$$^{A}_{B}T = ^{A}_{B}R + ^{A}P_{org(B)} = \begin{bmatrix} ^{A}_{B}R & ^{A}P_{org(B)} \\ 0 & 1 \end{bmatrix}$$

Aplicando la transformación:

$$^{A}P = ^{A}_{B}R\, ^{B}P + ^{A}P_{org(B)}$$

además:

$$^{A}_{B}R\, ^{B}P_{org(A)} + ^{A}P_{org(B)} = 0 \Longrightarrow ^{B}P_{org(A)} = -({}^{A}_{B}R)^{-1}\, ^{A}P_{org(B)}$$

y como ${}^{A}_{B}R$ es ortonormal

$$^{B}P_{org(A)} = -^{A}_{B}R^{T}\, ^{A}P_{org(B)} \tag{2.5}$$

si por otro lado

$$^{B}P = ^{B}_{A}R\, ^{A}P + ^{B}P_{org(A)} \tag{2.6}$$

sustituyendo 2.5 en 2.6

$$^{B}P = ^{B}_{A}R\, ^{A}P - ^{A}_{B}R^{T}\, ^{A}P_{org(B)} = ^{A}_{B}R^{T}\, ^{A}P - ^{A}_{B}R^{T}\, ^{A}P_{org(B)}$$

de lo cual

$$^{B}_{A}T = ^{A}_{B}R^{T} - ^{A}_{B}R^{T}\, ^{A}P_{org(B)} = \begin{bmatrix} ^{A}_{B}R^{T} & -^{A}_{B}R^{T}\, ^{A}P_{org(B)} \\ 0 & 1 \end{bmatrix}$$

$$^{B}_{A}T = ^{B}_{A}R - \, ^{B}P_{org(A)}$$

$$\therefore\ ^{B}_{A}T = ^{A}_{B}T$$

Aplicando transformaciones, si se conoce $q_{inicial}$ y q_{meta} con respecto a W y se quiere conocer q_{meta} con respecto a $q_{inicial}$:

$$^{q_{ini}}_{q_{meta}}T = ^{q_{ini}}_{w}T^{w}_{q_{meta}}T$$

$$ {}^{q_{ini}}_{w}T = \begin{pmatrix} {}^{w}_{q_{ini}}R^T & -{}^{w}_{q_{ini}}R^T\, {}^{w}P_{org(q_{ini})} \\ 0 & 1 \end{pmatrix} $$

$$ {}^{w}_{q_{meta}}T = \begin{pmatrix} {}^{w}_{q_{meta}}R & {}^{w}P_{org(q_{meta})} \\ 0 & 1 \end{pmatrix} $$

Quedando como caso general:

$$ {}^{q_{ini}}_{q_{meta}}T = \left(\begin{array}{c|c} {}^{w}_{q_{ini}}R^T\, {}^{w}_{q_{meta}}R & {}^{w}_{q_{ini}}R^T({}^{w}P_{org(q_{meta})} - {}^{w}P_{org(q_{ini})}) \\ \hline 0 & 1 \end{array} \right) \tag{2.7} $$

Si se resuelve el caso de un robot que se mueve en un espacio de dos dimensiones:

$$ {}^{w}_{q_i}R = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 \\ \sin\theta_i & \cos\theta_i & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

θ_i es el ángulo de rotación alrededor del eje z, ésta es la única rotación posible en dos dimensiones.

En este caso:

$$ {}^{q_{ini}}P_{org(q_{meta})} = \begin{bmatrix} \cos\theta_{ini} & \sin\theta_{ini} & 0 \\ -\sin\theta_{ini} & \cos\theta_{ini} & 0 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{bmatrix} x_{meta} \\ y_{meta} \\ 0 \end{bmatrix} - \begin{bmatrix} x_{ini} \\ y_{ini} \\ 0 \end{bmatrix} \right] \tag{2.8} $$

Por otro lado la orientación se obtiene mediante:

$$ {}^{q_{ini}}_{q_{meta}}R = \begin{bmatrix} \cos\theta_{ini} & \sin\theta_{ini} & 0 \\ -\sin\theta_{ini} & \cos\theta_{ini} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_{meta} & -\sin\theta_{meta} & 0 \\ \sin\theta_{meta} & \cos\theta_{meta} & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{q_{ini}}_{q_{meta}}R = \begin{bmatrix} \cos(\theta_{meta} - \theta_{ini}) & -\sin(\theta_{meta} - \theta_{ini}) & 0 \\ \sin(\theta_{meta} - \theta_{ini}) & \cos(\theta_{meta} - \theta_{ini}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9} $$

que representa una rotación de $(\theta_{meta} - \theta_{ini})$ alrededor del eje z.

2.8 y 2.9 se pueden simplificar en una sola ecuación de la siguiente manera:

$$ \begin{bmatrix} {}^{q_{meta}}x_{q_{ini}} \\ {}^{q_{meta}}y_{q_{ini}} \\ {}^{q_{meta}}\theta_{q_{ini}} \end{bmatrix} = \begin{bmatrix} \cos\theta_{ini} & \sin\theta_{ini} & 0 \\ -\sin\theta_{ini} & \cos\theta_{ini} & 0 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{bmatrix} x_{meta} \\ y_{meta} \\ \theta_{meta} \end{bmatrix} - \begin{bmatrix} x_{ini} \\ y_{ini} \\ \theta_{ini} \end{bmatrix} \right] \tag{2.10} $$

Para calcular los movimientos necesarios para llegar de una configuración a otra, se puede asumir que el robot sólo se puede mover en la dirección del eje

X^+ y que el planeador se asegura de entregar una secuencia de movimientos en la que la orientación es la necesaria para llegar a la posición en cuestión con lo que los movimientos necesarios se pueden calcular con:

$$giro_i = \theta_i - \theta_{i-1} \tag{2.11}$$

y una vez hecho el giro:

$$desplazamiento_i = (x_i - x_{i-1})\cos\theta_i + (y_i - y_{i-1})\sin\theta_i \tag{2.12}$$

Por ejemplo, si se quiere seguir la secuencia de la figura 2.8, se hace lo siguiente:

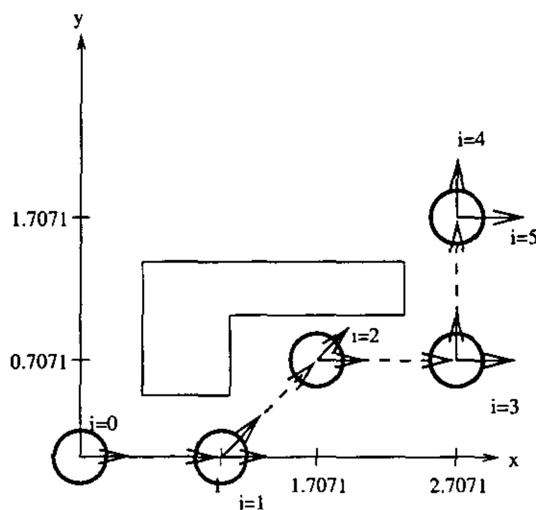


Figura 2.8: Ejemplo de navegación

Si $i = 1$, $x_0 = 0$, $y_0 = 0$, $\theta_0 = 0$, $x_1 = 1$, $y_1 = 0$, $\theta_1 = 0$, entonces:

$$giro_1 = 0 - 0 = 0$$

$$desplazamiento_1 = (1 - 0)\cos 0 + (0 - 0)\sin 0 = 1$$

Si $i = 2$, $x_2 = 1 + \frac{1}{\sqrt{2}}$, $y_2 = \frac{1}{\sqrt{2}}$, $\theta_2 = 45$, entonces:

$$giro_2 = 45 - 0 = 45$$

$$desplazamiento_2 = (1 + \frac{1}{\sqrt{2}} - 1)\cos 45 + (\frac{1}{\sqrt{2}} - 0)\sin 45 = 1$$

Si $i = 3$, $x_3 = 2 + \frac{1}{\sqrt{2}}$, $y_3 = \frac{1}{\sqrt{2}}$, $\theta_3 = 0$, entonces:

$$giro_3 = 0 - 45 = -45$$

$$desplazamiento_3 = (2 + \frac{1}{\sqrt{2}} - 1 - \frac{1}{\sqrt{2}})\cos 0 + (\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}})\sin 0 = 1$$

Si $i = 4$, $x_4 = 2 + \frac{1}{\sqrt{2}}$, $y_4 = 1 + \frac{1}{\sqrt{2}}$, $\theta_4 = 90$, entonces:

$$giro_4 = 90 - 0 = 90$$

$$desplazamiento_4 = (2 + \frac{1}{\sqrt{2}} - 2 - \frac{1}{\sqrt{2}})\cos 90 + (1 + \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}})\sin 90 = 1$$

Si $i = 5$, $x_5 = 2 + \frac{1}{\sqrt{2}}$, $y_5 = 1 + \frac{1}{\sqrt{2}}$, $\theta_5 = 45$, entonces:

$$giro_5 = 45 - 90 = -45$$

$$desplazamiento_5 = (2 + \frac{1}{\sqrt{2}} - 2 - \frac{1}{\sqrt{2}})\cos 45 + (1 + \frac{1}{\sqrt{2}} - 1 - \frac{1}{\sqrt{2}})\sin 45 = 0$$

y se llegó a la meta.

Existen dos modelos básicos de navegación, aquellos que cuentan con información completa y aquellos que no. Las ecuaciones descritas se pueden aplicar en ambos casos, pero cuando no se cuenta con información completa suelen ser insuficientes, así que es necesario realizar la navegación utilizando sensores, aunque esta tarea se puede encomendar al piloto.

2.8 Pilotaje

El piloto es el encargado de seguir las trayectorias que le indica el navegador, en un tiempo previamente establecido. También es el encargado de evadir los objetos que se puedan encontrar y que el planeador no previó para generar la ruta.

Para evadir los obstáculos el piloto requiere de sensores, los más utilizados son los sensores de proximidad y los de tacto. Dependiendo del tipo de sensores es la manera en que se realiza el pilotaje como se puede ver a continuación.

2.8.1 Navegación con sensores de tacto

La forma mas sencilla de sensado en un robot móvil es mediante sensores de tacto. El robot hace el recorrido de su posición inicial S a su posición objetivo T a través de una linea recta hasta que llega a T o interactúa con el *i-esimo* obstáculo en el punto de contacto H_i. Si se dió el contacto el robot sigue el perímetro del obstáculo hasta que llega al punto de salida L_i y continua en línea recta hacia T, como se puede ver en la figura 2.9. Dado que no se conocen las características del objeto con el que se hizo contacto, la dirección en que se le rodea se establece de manera arbitraria, por ejemplo a la izquierda. El problema

de seguir este algoritmo es que es probable que por la forma del obstáculo o por la distribución de éstos en el entorno del robot, nunca se llegue al objetivo a pesar de haber un camino.

Para resolver este problema se han propuesto los algoritmos Bug1 y Bug2 [9] que asumen que el robot puede conocer en cualquier momento sus coordenadas y las de su objetivo.

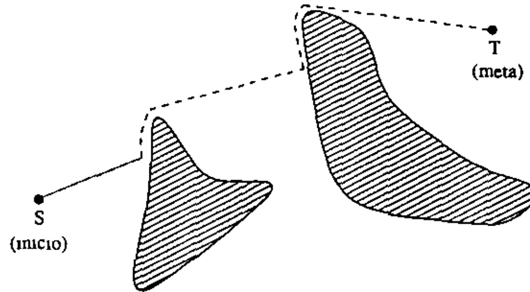


Figura 2.9: Navegación con sensores de tacto

El algoritmo Bug1

El algoritmo Bug1 consta de los siguientes pasos:

1. Del punto L_{i-1} (con $L_0 = S$, mueve al robot hacia T a lo largo de la recta (S, T) hasta que ocurra alguna de las siguientes opciones:
   (a) T es alcanzado, el proceso termina.
   (b) Se encuentra un obstáculo en el punto de contacto H_i; ve al paso 2.
2. Usando la dirección de movimiento predefinida, sigue la orilla del obstáculo. Si se llega a T, para. Mientras se avanza, se almacenan las coordenadas del punto actual en el registro R_1, el punto visitado mas cercano a T se almacena en Q_m, la distancia recorrida desde H_j en el registro R_2 y la distancia recorrida desde Q_m se guardan en el registro R_3. Después de haber rodeado completamente al obstáculo (se llegó de nuevo a H_i) se define el punto de salida $L_i = Q_m$ y se pasa al paso 3.
3. Si hay un segmento de recta entre L_i y T que cruze el obstáculo en el punto L_i, termina el proceso, porque el objetivo no es alcanzable. En otro caso utiliza R_2 y R_3 para determinar el camino más corto a L_i, se sigue el camino elegido y se establece $i = i + 1$ y se regresa a 1.

En la figura 2.10 se observa un ejemplo de seguir este algoritmo.

Figura 2.10: Ejemplo del algoritmo Bug1

2.8.2 El algoritmo Bug2

El algoritmo Bug2 consiste en:

1. Del punto L_{j-1} (con $L_0 = S$), mueve al robot hacia T a lo largo de la recta (S,T) hasta que ocurra una de las siguientes opciones:
   (a) T es alcanzado; termina el proceso.
   (b) Se encuentra un obstáculo en el punto de contacto H_j; se sigue en el paso 2.

2. Usando la dirección de movimiento predefinida, se sigue el contorno del obstáculo hasta que:
   (a) T es alcanzado; termina el proceso.
   (b) La recta (S,T) se encuentra en un punto Q tal que la distancia $D(Q,T) < D(H_j,T)$ y la recta (Q,T) no cruza al obstáculo en el punto Q. Se define el punto de salida $L_j = Q_m$, se establece $j = j+1$ y se va al paso 1.
   (c) El robot recorre totalmente el contorno del obstáculo y regresa a H_j sin haber definido el siguiente punto de contacto H_{j+1}, en este caso termina el proceso, ya que el objetivo está "atrapado" y es inalcanzable.

En la figura 2.11 se observa un ejemplo de la aplicación del algoritmo Bug2.

En la práctica la eficiencia de estos dos algoritmos es variable. Bug1 es más conservador, ya que garantiza que el punto de salida es el más cercano a la meta. Bug2 es más agresivo y muchas veces más eficiente, ya que no requiere rodear todo el obstáculo.

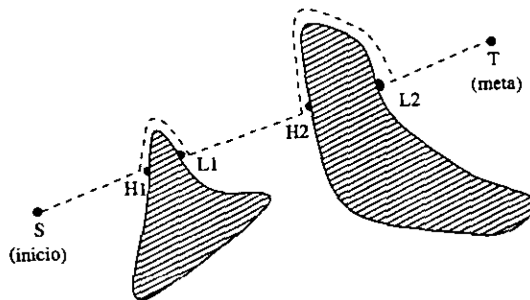


Figura 2.11: Ejemplo del algoritmo Bug2

2.8.3 Navegación con sensores de proximidad

Cuando se cuenta con sensores de proximidad, tales como el sonar, el sensor infrarrojo, sensores capacitivos, inductivos, de laser o visión, se puede mejorar el desempeño del robot aprovechando sus características. Por ejemplo se pueden modificar los algoritmos Bug1 y Bug2 para no necesitar regresar al punto de salida L_i definido, en lugar de ésto se puede "cortar" el camino y dirigirse a la meta en cuanto no haya obstáculo cercano "visible".

También se pueden utilizar otros mecanismos que ya entran en el terreno del pilotaje [10]. Cuando sólo se cuenta con información local y las decisiones de pilotaje se deben realizar dinámicamente, el robot debe ser capaz de tomar decisiones con rapidez. El algoritmo para resolver este problema se puede dividir en dos partes: un algoritmo de selección de submeta (SSA por sus siglas en inglés) y un algoritmo de decisión de dirección (SDA por sus siglas en inglés). En el SSA se genera una submeta que indica una dirección temporal a seguir cuando la meta final no es visible y en el SDA se genera una trayectoria de referencia que dirige al robot hasta que se genera la siguiente submeta.

En el SSA se pueden buscar las esquinas de los objetos que se tienen al alcanze de la vista y elegir como submeta aquella que esté más cerca de la meta. El SDA por su parte solamente genera las instrucciones necesarias para que el robot alcanze la submeta planteada, para esto puede utilizar el modelo matemático anteriormente tratado.

Bibliografía

[1] (Jones 1993) Jones, Joseph L. y Flynn, Anita M., *Mobile robots, inspiration to implementation* A K Peters, Massachusetts, USA, 1993.

[2] (Colombetti 1996) Colombetti, Marco, Marco Dorigo y Giuseppe Borghi, Behavior Analysis and Training - A Methodology for Behavior Engineering, *Transactions on Systems, man and Cybernetics- part B: Cybernetics,* Vol 26, No. 3, June 1996, IEEE.

[3] (Brooks 1991) Rodney A. Brooks, New Approaches to Robotics, *Science,* september 13th, USA.

[4] (Winston 1992) Winston, Patrick H. *Inteligencia Artificial* Version en español de la 3a. edición de "Artificial Intelligence" USA, 1992.

[5] (Feigenbaum 1982) Feigenbaum, Edward A., *Knowledge Engineering in the 1980's*, Depto. de Ciencias de la Computación, Universidad de Stanford, Stanford, California, USA, 1982.

[6] (Latombe 1991) Latombe Jean-Claude, *Robot Motion Planning,* Kluwer Academic Publishers, Massachussets, USA, 1991.

[7] (Giarratano ?) Giarratano Joseph y Riley Gary *Expert Systems, Principles and Programming,*2a edición, PWS Publishing Company, Massachussets, USA, 1994.

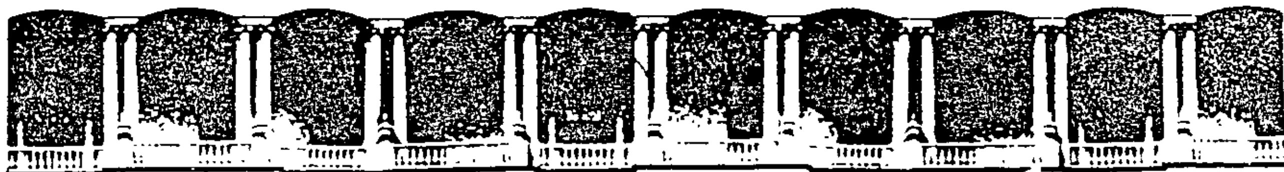
[8] (Giarratano 1994) Giarratano Joseph C., *CLIPS User's Guide*, Software Technology Branch, NASA, E.U. 1994

[9] (Lumelski 1986) Lumelski Vladimir J. y Stepanov Alexander A., "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment", IEEE Transactions on Automatic Control, Vol 31, pp 1058-1063, E.U. 1986.

[10] (Krogh 1989) Krogh B. H. y Feng D. *Dynamic Generation of Subgoals for Autonomous Mobile Robots Using Local Feedback Information*, IEEE Transactions on Automatic Control, Vol 34, pp 483-493, E.U. 1989.

[11] *BeeSoft User's Guide and Software Reference* Real World Interface, New Hampshire, 1997.

[12] (Leighton 1992) Leighton Russell R. *The Aspirin/MIGRAINES Neural Network Software: User's Manual*, MITRE Corporation, USA, 1992.



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS ABIERTOS

ROBOTS MÓVILES

VISIÓN PARA ROBOTS MÓVILES

EXPOSITORES : Ing. Jesús Savage Carmona

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 Mexico, D.F. APDO Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

Path	Frequency	Time	Speed
JFI	11	60.6 s	28.9 cm/s
JFGFI	2	91.5 s	25.7 cm/s
JFGHGFI	1	116.0 s	23.7 cm/s
JFGFGFI	1	133.0 s	22.2 cm/s

Table 5. Experiment 2.

Conclusions

This chapter has presented a navigation architecture that uses partially observable Markov decision process models (POMDPs) for autonomous indoor navigation. The navigation architecture, which is one layer in a larger autonomous mobile robot system, provides for reliable and efficient navigation in office environments.

The implemented POMDP-based navigation architecture has demonstrated its reliability, even in the presence of unreliable actuators and sensors, as well as uncertainty in the metric map information. Robot control is fast and reactive during navigation (Xavier averages fifty centimeters per second), and robust, as demonstrated by experiments that required the robot to navigate over 60 kilometers in total.

We believe that such probabilistic navigation techniques hold great promise for making robots reliable enough to operate unattended for long periods of time in complex and uncertain environments. Applying POMDPs to robot navigation also opens up new application areas for more theoretical results in the area of planning and learning with Markov models.

Acknowledgements

Thanks to Lonnie Chrisman, Richard Goodwin, Karen Haigh, and Joseph O'Sullivan for helping to implement parts of the robot system and for many valuable discussions. Swantje Willms helped to perform some of the experiments with the simulator. This research was supported in part by NASA under contract NAGW-1175 and by the Wright Laboratory and ARPA under grant number F33615-93-1-1330. The views and conclusions contained in this chapter are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations and agencies.

PART TWO

Vision for Mobile Robots

Vision is the single most powerful sense in the human repertoire, yet for most of the history of mobile robotics, vision has avoided in almost phobic proportions. In the early years of the AAAI competitions, the rules committees couldn't offer enough bonus points to get teams to forsake ultrasonic based solutions to navigational tasks. Certainly, there were good reasons for a roboticist to defer on incorporating vision into the robot's sensor suite. The hardware was prohibitive: vision processing required expensive specialized (and difficult to maintain) hardware and cameras which were often too large to physically reside on a mobile robot. The traditional software processing paradigm was worse yet: visual scenes were completely analyzed to reconstruct the outside world. The most fundamental token of information for navigation, whether a region of space in front of the robot was occupied or empty, could take hours to extract from stereo. Vision was so demanding that researchers could justify putting man-months of effort into fusing data from cheap Polaroid ultrasonics or infra-red sensors.

But now almost every mobile robot has a camera or some sort of image processing. Two competitions, MIROSOT and RoboCup, which require significant visual processing have recently begun, and the AAAI competition is hosting events which simply cannot be done without vision. What happened? Certainly the costs of the hardware have dropped significantly. In the early 1990s Horswill published an inexpensive framegrabber that could be built from parts for under US $1000.00. Shortly there after commercial framegrabbers dropped to the same price, while Sparc 10s became a common budget item in grant proposals.

But another more fundamental reason is because the paradigm shift in behavioral organization towards reactivity propagated to vision. In the mid-1980s, researchers such as Arbib (1981) began examining animal models for both motor control *and* perception. They found that a frog uses primarily used simple visual motion detection to catch flying prey. Bees key on specific aspects of the color spectrum for foraging. Indeed, psychophysical studies showed that the human visual system supported similar simple behaviors. The lower-resolution peripheral vision is used to watch for indications of motion (e.g., collisions with looming objects), while the higher resolution fovea is used to gather information for

reasoning about an object. Humans don't take in everything at once in all its color, motion, and temporal dimensions, but rather direct attention to a very narrow portion of their visual field based on the task they're performing.

As a result of these investigations, the paradigm in vision shifted to a philosophy where perception exists to support the behaviors of a robot Furthermore, the vision processing for each behavior might necessitate radically different representations and algorithms The division and encapsulation of vision processing into compact, well-defined routines was also extended to pan-tilt control of ameras (see Combs and Brown, 1991, Huber and Kortenkamp, 1995, and Kroktkov, 1989) Behavioral-based vision has arrived!

However, understanding a paradigm shift and implementing a new organizational style are two different beasts. The chapters in this section address how behavior-based vision is being incorporated into programming robots. They concentrate on the architectural issues of integrating vision into behavioral-based robots, providing a mini-tutorial on how to implement vision in behaviorally-based mobile robots. The chapter by Horswill describes a minimalist system in which simple visual processes produce very sophisticated behavior The chapter by Murphy describes how mobile robots can control multiple sensors in both indoor and outdoor environments. Finally, the chapter by Schiller and colleagues provides a look at how vision can be used on an outdoor mobile robot.

Robin Murphy and David Kortenkamp

CHAPTER FIVE

The Polly System

Ian Horswill

This chapter describes the design and rationale of the Polly system. Although now decommissioned, Polly was unique in the following ways: It performed a complicated task involving navigation in an unmodified environment for extended periods; it used real-time vision as nearly its only sensing modality; and all computation was performed on board by a low-cost off-the-shelf vision system.

Polly was a simple, low-cost robot designed and built between 1992 and 1993 at the MIT Artificial Intelligence Laboratory It was designed to patrol the seventh floor of the laboratory, find visitors, and give them tours. Polly roamed the hallways of the laboratory looking for visitors. When someone approached the robot, it stopped, introduced itself, and offered visitors a tour, asking them to answer by waving their foot around (the robot could only see the person's legs and feet). If the person agreed, the robot led them around the lab, describing places as it recognized them.

Although primitive in many ways, Polly was unusual for its time in that it performed a relatively complicated task over an extended period of time (up to a period of hours), it only used real-time vision, and it only required on-board power and processing. The robot was built using off-the-shelf commercial hardware for less than $20,000 00, and was capable of running approximately six hours off batteries

Polly was intended to explore a number of ideas that were prevalent at the time. First, operation in a real environment. Polly was built for and tested in a real environment with real people going about their daily business. Second, it explored the use of minimalism. The aim of the designers was to build the simplest possible system that solved a given task and use its performance to determine what architectural extensions were necessary. Third, Polly used active, purposeful, and task-based vision (Gibson 1966; Ballard 1991, Ikeuchi and Herbert 1990; Aloimonos 1990). Rather than computing a task-independent model of the environment, vision should compute the specific information needed to solve the agent's tasks. Fourth, Polly explored the use of environmental specialization (Gibson 1966). The robot exploited pre-existing environ[illegible] structure to simplify agent design Finally, Polly tested the limits of rea[illegible] It traded

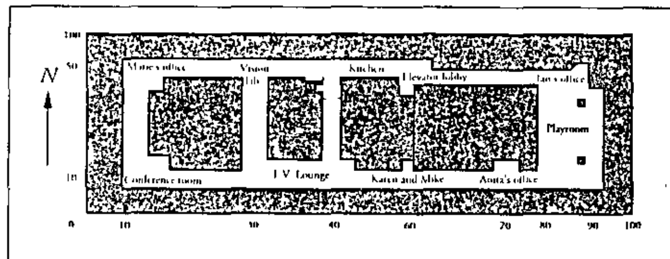


Figure 1 The approximate layout of Polly's environment (not to scale) and its coordinate system

speed of decision making for sophistication of decision making. Although many of these ideas are now widely accepted, Polly was an early test of how far they could be scaled.

Polly was very fast and simple. Its sensing and control systems ran at 15 hertz, so all percepts and motor commands were updated every 66 milliseconds It also used very little hardware (an equivalent robot could be built for approximately $10,000.00 today) and consisted of less than a thousand lines of scheme code (not including device drivers and data structures). All computation was done on-board on a low-cost digital signal processor (a TI C30 with 64K of RAM). Polly was among the best tested mobile robots to date, having seen hundreds of hours of service, and still has one of the larger behavior repertoires.

Design Methodology

Given the desire to build a minimalist system and to use task-based vision, the design methodology needed for Polly is straightforward:

- Choose the tasks to be performed.
- For each task, find a set of actions sufficient to perform the task.
- For each action, find a condition under which each action should be taken.
- For each condition, find a set of perceptual measurements that are sufficient to determine the truth of the condition.
- For each measurement, design a custom visual process to compute it
- If the process is too expensive or unreliable, find a set of environmental (domain) constraints that can be used to simplify it.

The difficulty in building the system comes from the fact that each step underdetermines the next; for example, there are many possible sets of condition and action pairs that can implement a given task. Finding a decomposition that bottoms out in a tractable set of vision primitives is difficult.

Another problem is to find domain constraints that are sufficiently true in practice to yield a robust system. Idealizations such as surface smoothness or the assumption of a slowly changing world are well known to be only approximately true. Oftentimes, they may depart from reality when they are most needed.

Having found a workable decomposition into actions, triggers, and perceptual measurements, we can build a working control program as a network of (simulated) parallel processes in the standard "reactive" or "behavior-based" style (Mataric 1997; Brooks 1986; Rosenschein and Kaelbling 1986). On each clock tick, every perceptual computation recomputes its output, every action trigger recomputes its truth value based on the new perceptual data, and any actions to be taken are fired in parallel. In practice, of course, the system is more complicated than a raw stimulus-response engine: perceptual computations, triggering decisions, and action performance may all require the maintenance of state information. Triggering decisions often share perceptual measurements, and perceptual measurements often share processing steps, so visual processes are only partly specific to actions or tasks.

Computational Properties of Office Environments

Office buildings are actively structured to make navigation easier (Passini 1984). The fact that they are structured as open spaces connected by networks of corridors means that much of the navigation problem can be solved by corridor following. In particular, we can reduce the problem of finding paths in space to finding paths in the graph of corridors. The MIT AI lab is even simpler because the corridor graph has a grid structure, so we can attach coordinates to the verticies of the graph and use difference reduction to get from one pair of coordinates to another.

Determining one's position in the grid is also made easier by special properties of office environments, the lighting of offices is generally controlled; the very narrowness of their corridors constrains the possible viewpoints from which an agent can see a landmark within the corridors. I will refer to this latter property as the *constant viewpoint constraint:* that the configuration space of the robot is restricted so that a landmark can only be viewed from a small number of directions. These properties make the recognition of landmarks in a corridor an easier problem than the fully general recognition problem. Thus very simple mechanisms often suffice.

Another useful property of office buildings is that they are generally carpeted, and their carpets tend to be either regularly textured or not textured at all. The

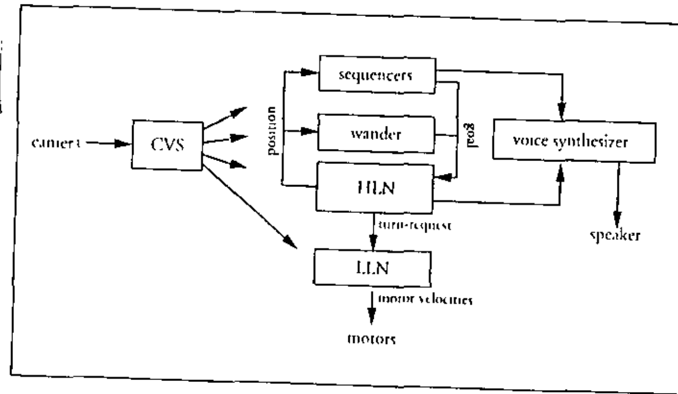


Figure 2. Overall architecture of the control system

predictability of the texturing of the carpet means that any region of the image which isn't textured like the carpet is likely an object resting on the ground (or an object resting on an object resting on the ground). Thus ***obstacle detection*** can be reduced to ***carpet detection,*** which may be a simpler problem admitting simpler solutions. In the case of the MIT AI lab, the carpet has no texture and so a texture detector suffices for finding obstacles. I will refer to this as the ***background-texture constraint*** (see Horswill 1995b for a more detailed discussion).

Finally, office buildings have a useful property in that they have flat floors, so objects that are farther away will appear higher in the image, provided the objects rest on the floor. This provides a very simple way of determining the rough depth of such an object. I will refer to this as the ***ground-plane constraint,*** that all obstacles rest on a flat floor (see Horswill 1995b).

Architecture

The components of Polly's control system are seen grouped in figure 2. The "core vision system," or CVS, computes a set of percepts that are used to guide and trigger various behaviors in the system. The low-level navigation system (LLN) consists of a set of reactive behaviors that collectively attempt to move forward while avoiding obstacles and following any corridors or walls that may be visible. The high-level navigation system (HLN) keeps track of the robot's destination and last-recognized location and periodically overrides the LLN by issuing ballistic turns to redirect the robot from one corridor to another or to reverse directions if it determines that it has overshot. The wander system im-

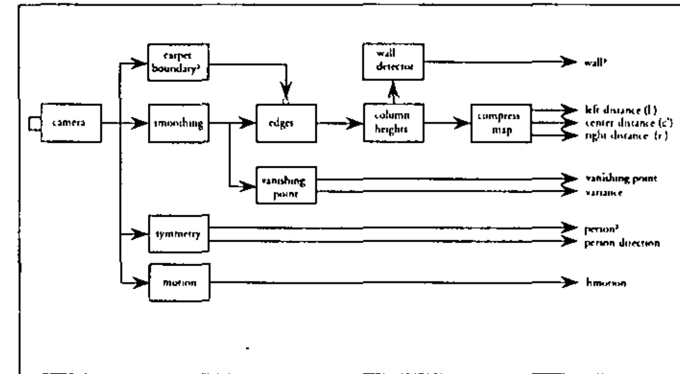


Figure 3. The Core vision system.

ments a patrol pattern by alternately posting opposite corners of the lab as goal locations. Finally, a set of sequencers implements the execution of prespecified action sequences (i.e. plans) that coordinate the overall process of giving a tour or offering a tour.

The actual implementation is a set of scheme procedures, roughly one per parallel process, with variables used to simulate wires. On each clock tick (66 milliseconds), the robot grabs a new image and runs each process in sequence to compute a new motor command that is fed to the base computer.

Physically, the Polly consists of an RWI B12 robot base, which houses the motors and motor control logic, with an added enclosure that houses an extra battery, a VMEbus DSP board (based on the TI TMS320C30 running at 30 MHz), an 8-bit memory mapped VMEbus frame grabber, a 6811 microcontroller for servicing low-speed peripherals, a voice-synthesizer, and an LCD display. All computation is done on board.

Visual System

The visual system processes a 64 x 48 image every 66 milliseconds and generates a large number of "percepts" from it (figure 4). Most of these are related to navigation, although some are devoted to person detection or sanity checking of the image.

The central pipeline in figure 3 ("smoothing" ... "compress map") computes depth information. The major representation used is a *radial depth* —that

open-left?	open-region?	person-direction
open-right?	blind?	wall-ahead?
blocked?	light-floor?	wall-far-ahead?
left-turn?	dark-floor?	vanishing-point
right-turn?	person-ahead?	farthest-direction

Figure 4. Partial list of percepts generated by the visual system

is, a map from direction to distance, similar to the output of a sonar ring. Computing depth is a notoriously difficult problem. As previously discussed, Polly simplified the problem using knowledge of the pre-existing structure of its environment. The fact that obstacles rest on the ground, and satisfy certain technical constraints on their geometry (the "ground-plane constraint"), allows Polly to use the image-plane height of the lowest pixel of an object as a measure of its distance. The fact that the floor has little or no texture (the "background-texture constraint") allows it to perform figure and ground segmentation using an edge detector. The conjunction means that a radial depth map can be computed, modulo monotone deformations, by finding the image plane height of the lowest textured pixel in each image column (see figure 5).

The visual system then compresses the depth map into three values, *left-distance, right-distance,* and *center-distance,* which gives the closest distance on the left side of the image, right side, and the center third, respectively. Other values are then derived from these For example, *open-left?* and *open-right?* are true when the corresponding distance is over a threshold. *Left-turn?* and *right-turn?* are true when the depth map is open on the correct side and the robot is aligned with the corridor. The depth map is also used as input to a simple heuristic wall detector. If all column heights are approximately equal in a wide range of the depth map, the system assumes there is a wall directly ahead

The visual system also computes the vanishing point of the corridor for use in steering. Bellutta, Collini, Verri, and Torre (1989) describe a system that extracts vanishing points by running an edge finder, extracting straight line segments, and performing two-dimensional clustering on the pairwise intersections of the edge segments Polly uses a simplified algorithm based on domain constraints such as knowledge of the location of the horizon and the knowledge that the corridor edges are the predominant edges in the image (for details, see Horswill [1993]); however in practice, the vanishing point algorithm adds little over and above the collision avoidance algorithm

In addition to the depth and vanishing point computations, the CVS implements a gray-level symmetry operator used to find pairs of legs in the image and a very simple motion operator to detect foot gestures

Figure 5. Computing freespace from texture. The robot computed a depth map by first labeling every pixel as being floor-like or not (middle) and then finding the image plane height of the lowest non-floor pixel in each column. The result was a monotonic measure of depth for each column (right)

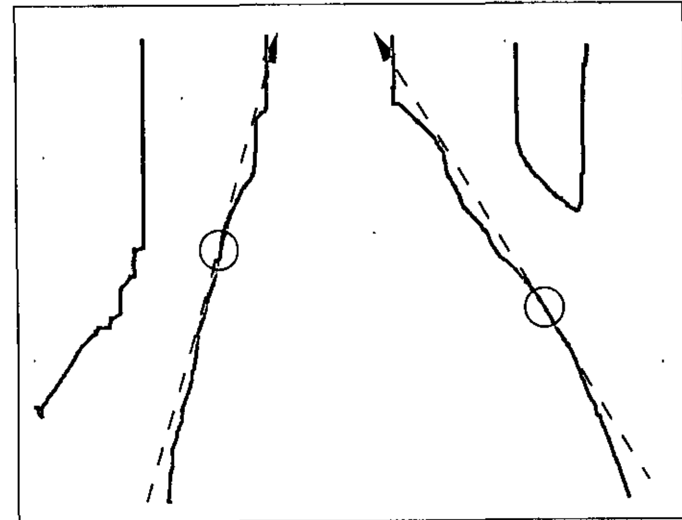


Figure 6 The vanishing point computation Edge fragments at individual pixels (shown in circles) are extended (dashed lines) and intersected with the top of the image to find the horizontal location of their intersections (arrowheads) The mean of the horizontal locations is used as the vanishing point.

Low-Level Navigation

Polly's motion is controlled by three parallel systems. The speed control system drives forward with a velocity of $\alpha(\text{center-distance} - d_{stop})$, where d_{stop} is the threshold distance for braking and α is a gain parameter. This means it backs

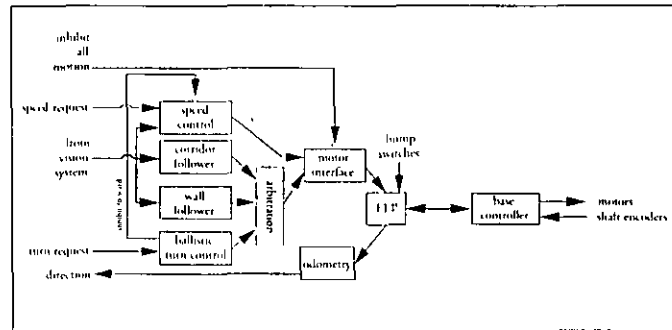


Figure 7. The Low-level navigation system.

up if it overshoots or is herded by an aggressive graduate student.

The corridor follower drives the turning motor to keep the vanishing point in the middle of the screen and keep *left-distance* and *right-distance* equal. An arbiter switches from corridor-following to wall-following mode (keeping the wall at a constant distance) when only one wall is visible. Finally, the turn unit drives the base in open-loop turns when instructed to by higher-level systems. The arbiter overrides both corridor and wall following during open-loop turns. During large turns, the speed control system also inhibits forward motion, which could otherwise lead Polly to suddenly turn into a wall. For a more detailed discussion of the low-level navigation system, see Horswill (1993). Polly's last line of defense against collision is a set of bump switches that immediately limp the motors on contact. The bump switches extend approximately 10 centimeters in front of the robot, so at 1 meter per second, they can predict a collision only 100 milliseconds in advance—an insufficient time for the robot to actively brake. Thus the most the bump switches can do is prevent Polly from continuing to push against the wall once it collides. To insure that the motors are limped as quickly as possible, they are polled not by the DSP (which runs at 15 Hz), but by the front-end processor (FEP), which runs a 1 KHz sense-action loop. The FEP is normally responsible for linking the DSP to the base, control panel, and other low-speed peripherals.

Place Recognition

Polly keeps track of its position by recognizing landmarks and larger-scale "districts," which are given to it in advance. The lab and some of its landmarks are

Kitchen		**Elevator Lobby**	
Position	(50, 40)	Position	(60, 40)
Direction	west	Direction	east
Veer	0	Veer	45
Image	...	Features	right, wall

Figure 8. Example place frames.

shown in figure 1. Since Polly patrols corridors, most landmarks are corridor intersections.

The corridors of the lab provide a great deal of natural constraint on the recognition of landmarks. The corridors form a grid, with each corridor running either north-south (NS) or east-west (EW). The robot's base provides rough rotational odometry which is good enough for the robot to distinguish which of four directions it is moving in and the type of corridor it is in.

Landmarks are represented using a topological map in which nodes are landmarks and arcs are corridors (Kuipers and Byun 1988; Mataric 1992). An unusual feature of the map is that it does not contain explicit representations of edges because the environment is known a-priori to have a grid topology. Instead, landmarks are identified by a pair of "qualitative coordinates" (figure 1) that, although not metrically accurate, preserve the ordering of places along each of the axes. Since the coordinates do not accurately encode real distances, even to within an order of magnitude, they effectively contain only topological information: landmarks are joined by a corridor if they share a coordinate value. Keeping accurate metrical information is unnecessary for this task and would have required more odometric information than was available.

Landmarks are stored in an associative memory that is exhaustively searched on every clock tick (66 milliseconds). The memory consists of a set of framelike structures, one for each possible view of each landmark (figure 8). A frame specifies the expected appearance of a particular landmark view. Frames contain a place name, qualitative coordinates, a compass direction from the viewpoint from which the landmark was viewed, and some specification of the view's appearance, either a 16 x 12 gray-scale image or a set of qualitative features (left-turn, right-turn, wall, dark-floor, light-floor). Frames can also be tagged with a turn (or "veer") to make once a landmark had been reached. This is useful when a corridor jogs to one side or the other. The representation is quite compact; The complete set of frames for the seventh floor require approximately 1 kilobyte of storage.

Matching is performed by finding the landmark view closest to the most recently recognized landmark that is consistent with the current direc[illegible] of travel

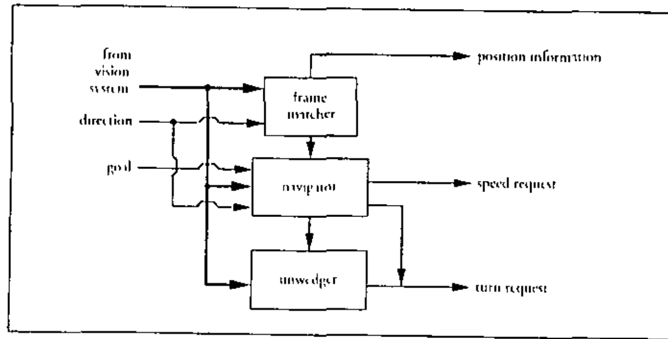


Figure 9 The high-level navigation system

and the observed visual features. For example, a landmark east of the last recognized landmark can't be matched if the robot is presently traveling west, nor can a north-facing view of a landmark.

The system can also recognize large-scale "districts" and correct its position estimate even if it can't determine exactly where it is. For example, when the robot is driving west and sees a left turn, it can only be in the southern EW corridor, so its *y* coordinate has to be 10, regardless of its *x* position. District recognition helps the robot recover more quickly from navigational errors

High-Level Navigation

By default, the corridor follower is in control of the robot at all times. The corridor follower always attempts to go forward and avoid obstacles until it is overridden. Higher-level navigation is implemented by a set of independent processes that are parasitic upon the corridor follower. These processes control the robot by enabling or disabling motion and by forcing open-loop turns.

The navigator chooses corridors by performing difference reduction between the (qualitative) coordinates of the goal and the last recognized location. When there is a positive error in the *y* coordinate, it attempts to drive south, or north for a negative error, and so on This technique is very tolerant of place recognition errors because it effectively replans at every time-step. If a landmark is missed, the system automatically compensated when it relocalizes itself

Because the LLN implements a potential fieldlike algorithm, it is possible for it to get stuck in local minima. The "unwedger" unit detects this condition and forces open-loop turns to get it out of stuck states. The unwedger fires whenever


```
(define-sequencer (offer-tour :import (...))
   (first  (set! global-mode 3)
           (set! inhibit-all-motion? true))
   (when done-talking?
     (new-say  "Hello. I am Polly. Would you like a tour?
               If so, wave your foot around."))

   (sleep 9)
   (then (if blocked?
           ;; The person's still there.
           (if (> hmotion 30)
             (do! give-tour)
             (begin
               (new-say "OK. Have a nice day.")
               (set! global-mode 1)
     (set! inhibit-all-motion? false)))
           ;; The person's gone.
           (begin    (set! global-mode 1)
                     (set! inhibit-all-motion? false)))))
```

Figure 10 Sequencer code for offering a tour

the corridor follower is blocked for a long period of time (2 seconds). One useful feature of the unwedger is its ability to force a turn in the direction of the goal, rather than in a random direction. This helps keep it on track.

Finally, a set of action-sequencers (roughly plan executives for hand-written plans) is used to implement tour-giving and operations such as docking. The sequencers executed a language of high-level commands such as "go to place," which are implemented by sending commands to lower-level modules such as the navigator.

Performance

Polly ran at 1 meter per second. Its speed was generally limited by mechanical factors rather than by computing power The robot ran as fast as 1.5 meters per second, but the base had difficulty accelerating past that point. Even at that speed, the robot became mechanically unstable. If, when running at 1.5 meters per second, it tried to stop in less than a meter or so, it tipped over The update

```
(define-sequencer (give-tour :import (...))
  (first  (new-say "OK  Please stand to one side.")
          (if (= last-y 40)
            (begin  (set! tour-end-x 80)
                    (set! tour-end-y 10))
            (begin  (set! tour-end-x last-x)
                    (set! tour-end-y last-y))))
  (when (not blocked?)
    (new-say "Thank you  Please follow me ")
    (set! inhibit-all-motion? false)
    (set! global-mode 2))
  (wait frame-strobe?)
  (wait (at-place? tour-end-x tour-end-y))
  (sleep 1.0)
  (then
    (new-say  "That's the end of the tour.
              Thank you and have a nice day.")
  (set! global-mode 1)))
```

Figure 11. Sequencer code for giving a tour

rate of the control system was limited by the I/O performance of the frame grabber, not by computing power. More recent implementations of the algorithm run at video rate.

By the standard of research robotics, Polly was very robust, having seen hundreds of hours of service in multiple environments. Polly's successors have been run in many environments, including conferences and elementary schools.

Low-Level Navigation

Most locomotion problems were obstacle detection problems The corridor follower performed well on all floors of the MIT AI lab building on which it was tested (floors 3-9) except for the 9th floor, which has very shiny floors, there the system braked for reflections of the overhead lights in the floor

The system's major failure modes were due to violations of the background-texture constraint. If the environment contained strong illumination gradients, surface markings on the floor, or floor surfaces with high specular reflectivity, the system hallucinated obstacles Thus, for example, sufficiently strong shadows triggered false positives This is less of a problem than one would expect, because office buildings typically have area light sources that generate di[illegible]e

shadows. Another example was the boundary between two distinct carpets, each with a different color, which could be mistaken for obstacles Polly dealt with this problem using a hand-crafted recognizer tailored to the particular carpet boundaries of the lab. A better approach would be to verify the presence of the obstacle using a different algorithm or sensor with statistically independent failure modes.

Polly's other locomotion problems were mechanical. Because of the relatively small support polygon of its base and its relative top-heaviness, it was necessary to limit the maximal acceleration of the base to prevent it from tipping. That, in turn, meant limiting the maximum velocity of the robot to insure that it could stop quickly if someone walked in front of it. Thus, although Polly was capable of running at better than 1.5 meters per second, it was necessary to limit it to 1 meter per second to ensure safety.

Place Recognition

Place recognition was the weakest part of the system. While recognition by matching images is quite general, it is also fragile, and particularly sensitive to changes in the world. If a chair is in view when a landmark template is photographed, then it must continue to be in view, and in the same place and orientation forever. If the chair moves, then the landmark becomes unrecognizable until a new view is taken. Another problem was that the robot's camera was pointed at the floor, and there isn't much of interest to see there. For these reasons, place recognition was restricted to corridor intersections represented by feature frames, because they are more stable over time. The one exception was the kitchen, which was recognized using images. In ten trials, the robot recognized the kitchen eight times while going west, and ten times while going east. Westward recognition of the kitchen failed completely when the water bottles in the kitchen doorway were moved, however.

Both methods consistently missed landmarks when there was a person standing in the way This often led the robot to miss a landmark immediately after picking up a visitor. The methods also failed if the robot was in the process of readjusting its course after driving around an obstacle or if the corridor was very wide and had a large amount of junk in it. Both conditions caused the constant-viewpoint constraint to fail. The former sometimes caused the robot to hallucinate a turn because one of the walls was invisible, although this was rare

Recognition of districts was very reliable, although it could sometimes become confused if the robot was driven in a cluttered open space rather than a corridor.

High-Level Navigation

High-level navigation performance was limited by the accuracy of place recognition In general, the system worked flawlessly unless the robot got l[illegible]r ex-

ample, Polly often ran laps (implemented by alternately giving opposite corners as goals to the navigator) for over an hour without any navigation faults. When the Polly got lost, the navigator generally overshot and turned around If the robot got severely lost, the navigator flailed around until the place recognition system reoriented. The worst failure mode occurred when the place recognition system thought that it was east of its goal when it was actually at the western edge of the building In this case, the navigator unit and the unwedger fought each other, making opposite course corrections. The place recognition system should probably have been modified to notice that it was lost in such situations so that the navigator would stop making course corrections until the place recognition system relocked. This has not yet been implemented, however.

Getting lost is a more serious problem for the action sequencers, since they are equivalent to plans but there is no mechanism for replanning when a plan step fails This absence can be mitigated by using the navigator to execute individual plan steps, which amounts to shifting responsibility from plan-time to run-time.

Polly's Successors

In the last few years, we have built a number of systems based on various aspects of Polly, particularly its collision avoidance system. The Frankie, Wilt, and Gopher systems built at MIT focused on optimizing the hardware designs of the vision system. They are based on a custom-built computer, camera, and active-head system that cost less than $1,000.00 (Horswill and Yamamoto 1994). Because these systems are highly portable, they have been tested in a wide range of environments, including corporate research labs, hotels, grammar school classrooms, and even an auditorium at West Point.

The Pebbles (Lorigo 1996) and Elvis systems, also built at MIT, run on the Cheap Vision Machine, a low-cost, high-performance vision computer designed by Barnhart and Horswill. Lorigo (1996) has extended the collision avoidance system to novel environments, including outdoor environments. Thau (1997) has extended it to fuse multiple views into coherent geometric maps in real-time. I integrated elements of Polly with an implementation of the Ullman visual routine processor theory (Ullman 1984) to build a system that could search for and approach objects in specified configurations on demand (Horswill 1995). This system also ran on the Elvis hardware. At Northwestern, my colleagues and I are developing highly programmable systems that extend this work Kluge is a robot based on the Cheap Vision Machine that combines search and navigation, and manipulation capabilities to perform relatively high-level tasks such as delivering and playing fetch One of Northwestern's principal goals is to build a very robust place recognition system for office environments.

Conclusions

Many vision-based mobile robots have been developed in the past (see for example Kosaka and Kak 1992; Kriegman, Triendl, and Binford 1987; Crisman 1992; and Turk, Morgenthaler, Gremban, and Marra 1987). The unusual aspects of Polly were its relatively large behavioral repertoire, simple design, and principled use of special properties of its environment.

Polly's efficiency and reliability were due to a number of factors Specialization to a task allowed the robot to compute only the information it needed. Specialization to the environment allowed the robot to substitute simple computations for more expensive ones. The use of multiple strategies in parallel reduced the likelihood of catastrophic failure (Horswill and Brooks 1988). Thus, if the vanishing point computation generated bad data, the depth-balancing strategy compensated and the distance control system prevented collisions until the vanishing point was corrected. Finally, the speed of its perception and control loop allowed it to rapidly recover from errors. This relaxed the need for perfect perception and allowed simpler perceptual and control strategies to be used.

The principled use of environmental constraints not only facilitated the design of efficient algorithms, but also helped understand and generalize them. The analysis of the collision avoidance system showed that the background-texture constraint was used to simplify figure and ground separation and nothing else. Thus the system could be ported to environments that satisfied the ground-plane constraint but not the BTC by finding replacement constraints that it did satisfy. As Lorigo (1996) has shown, this can be used to build a relatively general system by adaptively switching between specialized systems.

Specialized systems need not simply be hacks. We can learn things from their design and apply it to the design of other systems.

Polly was an existence proof that a robust system with a large behavioral repertoire could be built using simple components specialized to a task and environment. As with other AI methodologies, it remains to be seen how far this approach can be scaled. Either way, the principled analysis of task requirements and environments will surely play a critical role in any successful methodology for studying action in the world

CHAPTER SIX

Coordination and Control of Sensing for Mobility Using Action-Oriented Perception

Robin R. Murphy

In the reactive renaissance of the 1980s, autonomous mobile robots were constructed with one sensor per each active behavior. As a result, coordination and control of sensing for mobility was not an issue. However, the coordination and control of sensing is becoming increasingly more challenging as mobile robots are applied to more difficult tasks, with larger sets of possible behaviors and a small set of general purpose sensors. Now a designer must decide how to select sensors, design representations and algorithms, and organize and share sensing resources to accomplish demanding tasks.

This chapter presents one philosophy of sensing organization, ***action-oriented perception***, and discusses how it can be applied to mobile robots, with multiple sensors performing locomotive tasks. According to action-oriented perception, all perceptual processes should be examined in terms of their ***perceptual context***: the requirements of the task, the projected condition of the environment for the expected duration of the robot's activities, and the available sensing capabilities. The perceptual context can be viewed as defining the ecological niche of a situated agent. The principles of action-oriented perception have been successfully applied to a number of domains in motor control and sensor fusion for autonomous mobile robots (Arkin 1992; Arkin and Murphy 1990; Murphy and Arkin 1992). This chapter attempts to distill the experiences at the Colorado School of Mines (CSM) in applying this philosophy to one application domain, outdoor road following. The principles of action-oriented perception and the guidelines presented in this article are expected to be relevant for other forms of robotics including industrial manipulators and semi-autonomous teleoperation control systems.

In the next section, I review three current trends in the coordination and control of mobile robots and the associated constraints on the organization of sensing. These architectural paradigms serve as a framework, or "skeleton" of actions, to which sensing, the "muscle," is attached. The skeletal arrangement of the reactive and hybrid styles of architecture is compatible with action-ori perception. Studies from cognitive psychology on the role of sensing in lo

tor activities provide insights in the section on sensing organization in robotic architectures as to how the sensing "muscles" can be analyzed and arranged. Finally, I post a set of guidelines intended to aid the designer in arranging sensing for a specific task. Illustrative examples are taken from the CSM entries in the 1994 and 1995 Association for Unmanned Vehicle Systems (AUVS) International Unmanned Ground Vehicle competitions. The competition and the CSM entries are described in in the section on sensing organization from cognitive psychology. It should be emphasized that this chapter concentrates on the architectural aspect of sensing: *how sensing is interfaced with acting and other sensing demands*. Unfortunately, a discussion of the control and coordination of sensing in general is beyond the scope of a single chapter. Specific aspects, such as multisensor integration and sensing resource allocation, are not addressed. Murphy (1996) provides overview of these issues. The relative merits of different sensors for locomotion (such as laser range finders versus sonar) are also not addressed in this chapter.

Example Domain: UGV Competition

The 1994 and 1995 AUVS International Autonomous Ground Vehicle Competitions serve as motivating examples of realistic outdoor locomotion tasks. The AUVS has sponsored the competitions annually since 1993. The competition is open to teams composed of both graduate and undergraduate students and faculty. The goal of the competition is to have a small autonomous vehicle navigate around an outdoor course in the shortest amount of time. All power, sensing, and computing must be done onboard. In many regards, the competition task is comparable to the road-following demonstrations for the DARPA Autonomous Land Vehicle and Unmanned Ground Vehicle projects.

This section summarizes the competition rules, which delineate the task demands portion of the perceptual context and provides a brief overview of the CSM entries. Each entry was based on the research project for that year's MACS574 graduate course, AI Robotics and Computer Vision. The 1994 team took first place, while the 1995 entry placed fourth. Murphy (1995) and Murphy, Hoff, Blitch, Gough, Hoffman, Hawkins, Krosley, Lyon, Mali, MacMillan, and Warshawsky (1995) provide details of each entry.

Competition

The course boundaries are designated by Department of Transportation white lane lines approximately four inches wide painted on the ground and ten to twelve feet apart. The boundaries were continuous throughout the course for the 1994 competition. In 1995, the majority of the course boundaries were continuously lined; however, one portion was marked with alternating dashed lines. The dashed lines were alternating twenty foot lines with a fifteen foot separation. Obstacles (hay bales covered in plastic) were randomly placed on the course along with a sand pit and, in 1995, a ramp. In practice, the bales never extended more than three feet into the lane, although the rules do permit bales to be placed in the center and almost completely block the course. The teams did not know the metric layout of the course prior to the competition and were expected to compete under late spring weather conditions, such as heavy cloud cover or bright sun. The maximum permissible speed of any entry was five miles per hour. Approximate layouts of the courses are shown in figure 1; both courses were on the order of 800 yards long. The courses were divided into four sections, I-IV, of increasing difficulty (terrain, obstacle density). The course boundaries in 1995 were significantly fainter than in 1994 and offered little contrast with brown grass, confounding the vision processing of almost all entries.

1994 CSM Entry

The 1994 CSM entry used *Omnibot*, a fully autonomous vehicle lent to the team by Omnitech Robotics, Inc., the team's industrial sponsor. The robot is shown in figure 2a. The vehicle base is a Power Wheels battery powered children's jeep. Its maximum drive speed was 1.5 mph. All control was done onboard with a 33 MHz 486 PC using Omnitech CANAMP controllers. The sensor suite consisted of a single panning ultrasonic sensor mounted in front and a video camcorder on a mast near the center. The ultrasonic could sweep 180 degrees and had a reliable range of approximately 10 feet. The video camcorder was a standard consumer electronics model; the output was digitized by a black and white framegrabber board in the PC. All programming was done in C++ with the Lynx UNIX PC operating system.

The pedagogical objectives of the 1994 entry were for the students to become familiar with purely reactive architecture, ecological principles of action-oriented perception, and the use of multiple sensors. The details of sensing and acting will be discussed in later sections, but it is helpful to have an initial overview of the software layout. The design relied on a physical configuration of the robot where both the camera and the ultrasonic sensor were fixed in position, pointing to the right side of the robot (perpendicular to its centerline). The robot viewed only the right course boundary, and the line occupied at least one-fourth of the image area to reduce the signal-to-noise ratio.

The software was divided into two behaviors, *follow-path* and *move-ahead*, coordinated by an *innate releasing mechanism* style of sequencer (Anderson and Donath 1990). The primary behavior was the vision based *follow-path*. *follow-path* exploited the conclusions from an empirical investigation suggesting that the white line was usually represented by the brightest pixels in the image. Highlights on the grass appeared to be randomly distributed. There-

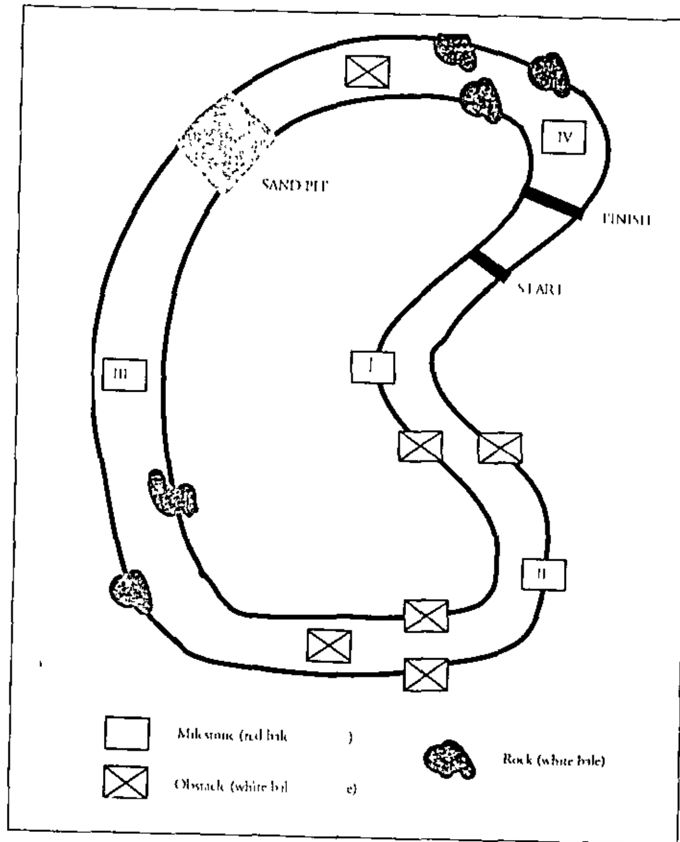


Figure 1. General layout of the competition courses.

fore, the centroid of the brightest pixels in the image generally coincided with the center of the line. Another behavior, *move-ahead*, was needed to navigate past the hay bales on the side lines which confused the vision based *follow-path* behavior. *move-ahead* was activated by the *distraction?* releaser whenever a bale was detected via sonar. *move-ahead* behaved as a no-op, using shaft encoders to maintain the course for the estimated time needed to move out of the viewing range of the bale.

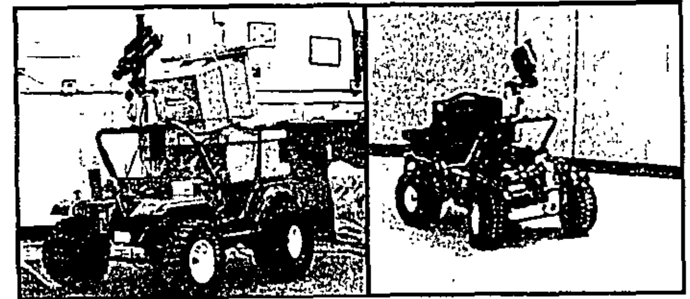


Figure 2. The CSM entries. Left: Omnibot (1994). Right: C2 (1995).

1995 CSM Entry

The 1995 CSM team used *C2* (figure 2). *C2* is the *Omnibot* base and hardware donated to CSM by Omnitech Robotics, Inc. The Lynx-based PC was replaced with a DCI Diamondback field PC using a 100 MHz Pentium Intel microprocessor operating under MS-DOS. The vision system was changed to a ImageNation black and white framegrabber board. The pedagogical objectives of the 1995 entry were for the students to become familiar with the CSM architecture, evaluate existing computer vision based road following techniques, and gain practical experience in using multiple sensing modalities.

The software layout is shown in figure 4. The heart of the entry was the reactive behavior *follow-path*, which used a simplification of the Dickmanns and Graefe vision-based road-following algorithm (Dickmanns and Graefe 1988) called "dynamic vision." Dynamic vision offers several mechanisms for achieving real-time, robust navigation. It interprets only those parts of a road image and image sequence containing features relevant to vehicle navigation. This reduces the amount of computation performed on each image, increasing the update rate of vision to the road model from which the vehicle computes steering commands. This interpretation is facilitated by subdividing the image into regions that can be processed separately in parallel. Again, this increases the update rate of the road model. Finally, the dynamic vision performs error checking by correlating road location information across several images. This prevents the vision system from identifying a shadow or an intersection as an abrupt change in the road.

Dynamic vision requires that the road construction parameters, such as road width, be known in advance in order to focus processing on the relevant regions of the image where the road is likely to be and to provide error checking. The method is particularly attractive in that is does not assume that the road boundaries are homogeneous, that is, that the road and shoulders are uniform color with no shadows, interruptions due to shadows or intersections, etc. One disadvantage

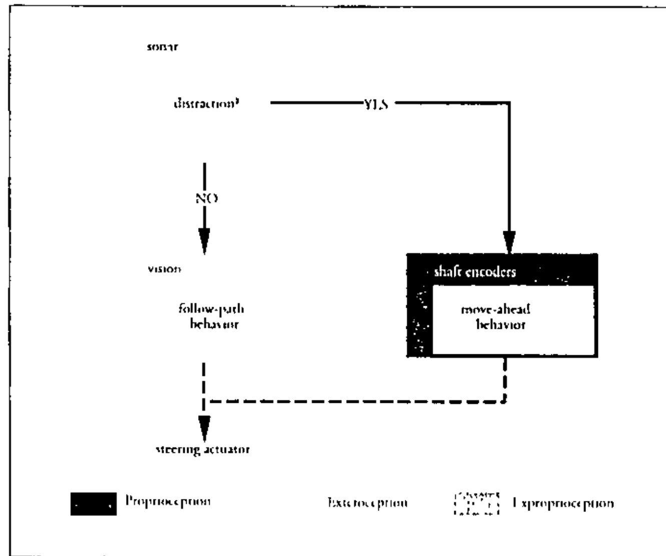


Figure 3. Behavioral control scheme for 1994 with innate releasing mechanism

of dynamic vision for *follow-path* is its underlying assumption that the camera will always be pointed down the road so that road edges will be within specific regions of the image. This means that the vehicle can only slow down or stop in response to obstacles rather than turn the vehicle and risk not facing the road

Dynamic vision processing starts by the acquisition of an image from the camera. Using the road model, six regions of interest representing likely locations of the road boundary (three left and three right) are selected. All operations on a region are handled by a dedicated processor in parallel. First, edges are extracted from the regions of interest. Edges that do not match the expected orientation of the road at that distance are rejected The remaining feasible edges from each region are correlated with each other to refine the road model Once the road model is updated, new steering commands can be generated, which then lead to new predictions for guiding processing of the next image. It should be noted that the correlation algorithm does not require that the edges in all regions agree; some regions may be affected by local phenomena such as shadows, occlusions of the boundaries by obstacles, or puddles It is assumed that these phenomena are local and temporary, that as the vehicle moves forward according to the road model, new information without these distortions

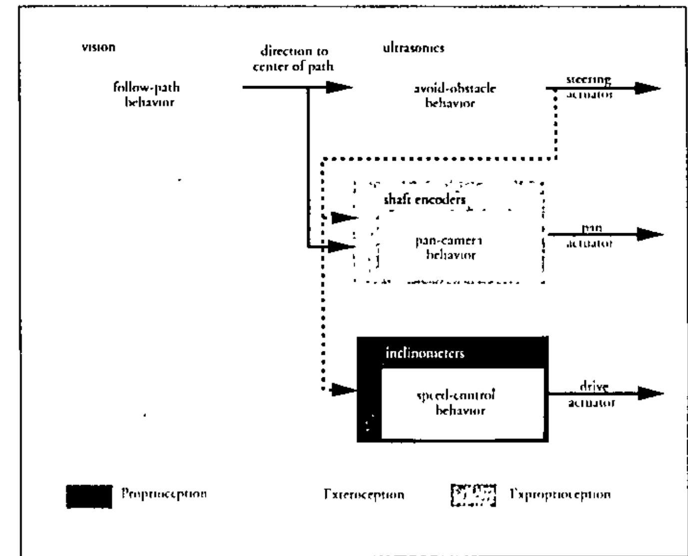


Figure 4. Behavioral control scheme for 1995.

will be acquired and the road model updated. This implies that the update rate must occur fast enough that a few updates can be missed every now and then

The structure of the CSM implementation follows the dynamic vision hierarchy of edge detection, edge filtering, correlation, and updating of the road model. However, the implementation differed in four major ways. First, the CSM implementation has to run on a single processor and relatively slow framegrabber. Second, a Hough transform (Niblack and Petkovic 1988) is used for the edge detection and filtering since the literature did not report on the specific algorithms used by Graefe. Third, the CSM implementation made use of a panning mast to keep both road boundaries in view at all times. This was used because the CSM robot must go around obstacles rather than stop or slow down. Finally, in order to increase processing speed, there is no check for correlations between images

Two advantages of using the more complicated algorithm in 1995 were (1) it was expected to work even in the presence of dashed lines, and (2) it could be reused for other applications, including indoor hall following Essentially, the 1995 approach broadened the ecological niche defined in 1994 from following white lines outdoors to following paths in general. In practice, the simplifications in *follow-path* were too severe, and when combined with other

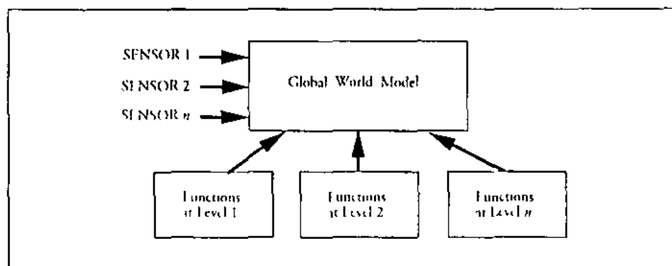


Figure 5 Sensing in the hierarchical paradigm.

hardware and software defects, *C2* placed only fourth in the competition.

The *follow-path* behavior computed the direction to the center of the path. The *avoid-obstacle* behavior was based on the vector field histogram method (Borenstein and Koren 1991) and used ultrasonics to avoid collisions while maintaining progress toward the center of the path. *pan-camera* was intended to keep both course boundaries in view The *speed-control* behavior was not instantiated at the competition because the robot's speed was not upgraded to five mph. However, it was intended to use inclinometer, steering shaft encoder, and sonar data to slow the robot as it went up ramps, made hairpin turns, and approached obstacles.

Sensing Organization in Robotic Architecture

Robotic motor control systems can be loosely grouped into three categories: hierarchical, reactive, and hybrid reactive-deliberative The paradigms focus on structuring motor actions Sensing is largely a secondary issue, although recent efforts have attempted to refine the role of sensing within hybrid architecture. This section describes the architectural paradigms in terms of ramifications for sensing organization and their ability to support action-oriented perception. The overall software design of the 1994 and 1995 entries is also discussed. Both entries used a reactive style of architecture, but with major differences in the perceptual representations at the lowest levels. These entries illustrate the range and flexibility of sensing within an architectural paradigm

Hierarchical Style

Hierarchical robotic motor control typically views a robotic action as the result of a sequence of operations within a strict hierarchy involving functional de-

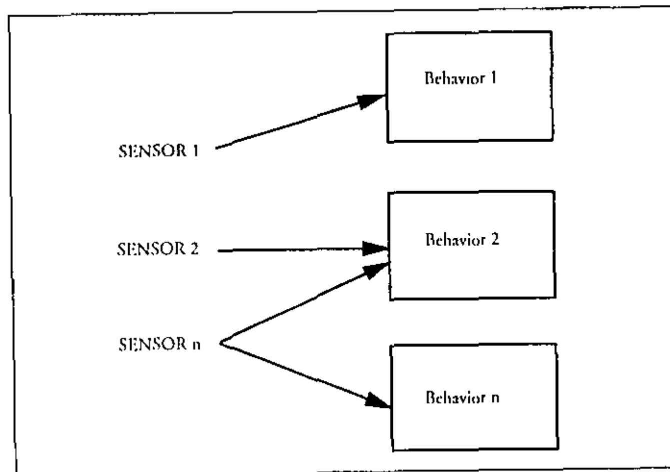


Figure 6 Sensing in the reactive paradigm.

composition, global world models, and fixed levels of sensory processing. Generally speaking, perception exists to update the global world model, and thereby only indirectly influences the robot's actions, as shown in figure 5. These systems can be characterized as following a ***sense, plan, act*** scenario. The real-time control system (RCS) developed at the National Institute for Standards and Technology (Albus 1990) is an example of a hierarchical system.

The hierarchical style is usually not consistent with the action-oriented perception philosophy. The global world model and rigid levels of sensory processing are independent of the task. The levels within the hierarchy may be considered to be an attempt to couple a motor function with the perceptual context, but the coupling is indirect at best.

Reactive Style

As shown in figure 6, reactive style architecture follows a ***sense-act*** arrangement, which lends itself to a reflexive, stimulus-response method of linking perception with actions A behavior can be thought of as a motor action template (act) coupled to dedicated perceptual processing routines (sense). The behavioral paradigm is inherently aligned with the philosophy of action-oriented perception. Behaviors encapsulate the sensing necessary for a particular action in the expected environment.

Reflexive behaviors, popularized by Brooks (1986), exploit the computational simplicity and speed of sensing a "stimulus" which is directly channeled to the motor action "response." More recent implementations of reactive motor control systems do not strictly adhere to a stimulus-response-like relation between sensing and acting. The behavior may use an internal, behavior-specific sensing representation to interpret the sensing data. The distinction between local (internal) representations consistent with a reactive style architecture and the global representations appropriate for a hierarchical system is vague. For the purposes of this chapter, a local sensing representation is defined as being constructed and maintained by only the user behavior. A global world model is intended to be general purpose; it is constructed and maintained by processes not necessarily associated with a behavior (such as map making).

One connotation of the reactive style of architecture is that one behavior is serviced by one sensor. In systems which have more behaviors active at one time than relevant sensors, more than one behavior may take and manipulate a sensor's observations. Since each behavior is acting locally on the sensory output, the other behaviors are unaffected by the concurrent uses of the same data. It is a matter of debate whether a behavior can share or communicate its sensing representation with another behavior in the reactive paradigm. This chapter assumes that if the sensing representation being shared does not require additional processing by functions outside of the behaviors using it, then it is local and the sensing organization is still reactive.

In some situations, one behavior may need to integrate or "fuse" the output of multiple sensors in order to reduce uncertainty or provide complementary information. Murphy and Arkin (1992) describes how evidence from multiple sensors can be fused locally within a reactive behavior (the Sensor Fusion Effects Architecture, SFX), and Murphy and Mali (1996) and Arkin and MacKenzie (1994) describe methods of sequencing perceptual routines.

Another connotation of reactivity is that the perception for a behavior is from the efficacy of early behaviors which were able to rely on stimulus-response methods. A less restrictive interpretation of the reactive paradigm is that the only deciding factor is whether the representation is local or global, not the number of observations or time period for its construction.

Hybrid Style

Hierarchical and reactive motor control, while on the opposite ends of the spectrum, are not necessarily mutually exclusive. Systems such as the Autonomous Robot Architecture (Arkin 1987), Task Control Architecture (Simmons, Lin, and Fedor 1990), and 3T (Bonasso et al. 1997) make use of reactive motor behaviors that are instantiated according to a hierarchical higher cognitive process. This approach can be described as *plan, sense-act*. The hybrid paradigm is also consistent with action-oriented perception, since the reactive

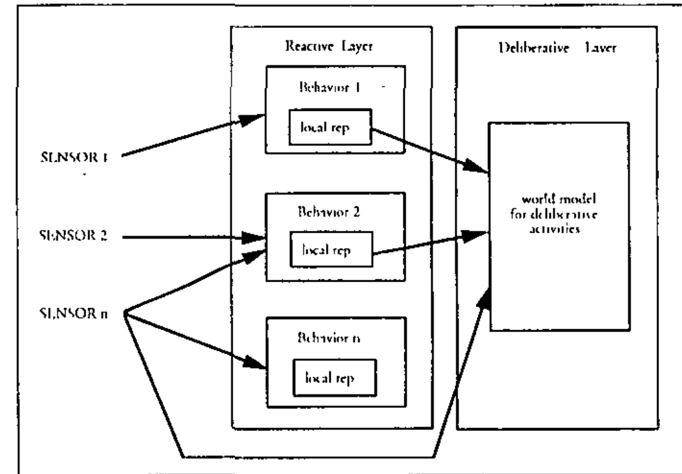


Figure 7. Sensing in the hybrid paradigm

layer tightly couples acting and perceiving for a specific behavior or subtask.

Perception in a hybrid reactive-deliberative system takes different tracks, as shown in figure 7. The output from a sensor may be shared between the two layers: it may be directed to a motor behavior through a behavior-specific perceptual process, while at the same time cognitive functions may integrate the observations into a global model. The output of the behavior-specific representation can itself be integrated into a global model; in essence, the deliberative layer can eavesdrop on the reactive behaviors and expand on their structures to support reasoning, planning, and problem solving.

Architectural Variants Focusing on Sensing

The issue of how to organize sensing has recently been the subject of research efforts. Chen and Trivedi (1995) use a deliberative process to plan about sensing, especially how to take advantage of new information. Noreils and Chatila (1995) use a hybrid style of architecture that detects failures by monitoring the motor behavior from the deliberative layer. SFX (Murphy and Arkin 1992) attempts to keep monitoring and exception handling at the behavioral level, but will fail upwards. None of these three architectures seems to violate the basic association of local representations to behavior, and global models to deliberative

layers. However, none sheds much insight into the process of constructing an adequate sensing hardware and software configuration at the behavioral level.

CSM Uses of Reactive Style

Both the 1994 and 1995 CSM teams used a reactive style architecture since the nature of the single navigation task did not require deliberation (e.g., path planning, map making, or reasoning). However, the entries differed in the use of internal perceptual representations for the behaviors. Although both entries used multiple sensors (i.e., vision, sonar, shaft encoders, inclinometers), the reactive paradigm effectively eliminated the issue of how to integrate such disparate modalities. The output of each sensor was directed to the user behavior, which handled integration locally, bypassing any global model.

The perception for the 1994 entry was composed of two reflexive behaviors. As you may recall, *follow-path* required only thresholding each image on the brightest pixels and taking the centroid in image coordinates. Each observation was independent of previous observations, and no internal representation, per se, was used. The advantage was that the behaviors were extremely fast. However, the overall system was brittle. The robot nearly veered out of bounds when a judge's shoe came into view, and then veered back in bounds following a patch of white dandelion thistles. Also, the path following and move-ahead behaviors were engineered solely for the competition; the software could not be used to guide the robot for any other path.

In contrast, both the *follow-path* and *avoid* behaviors for the 1995 entry used some type of explicit internal representation. The *follow-path* behavior used an internal model of a path. Following Dickmanns and Graefe (1988), the perceptual routine operated on six regions where the path boundaries were expected to appear in the image. The routine identified the best line in each region with a Hough transform and marked it as having either a negative slope (path turning to the left), positive slope (path turning to the right), or none (straight path). The resultant pattern was matched to a steering command via a table. Ambiguous patterns resulted in a "hold your course" action.

The *follow-path* representation was local, it was not used by the *avoid* or pan-camera behaviors. The *avoid* behavior accepted the preferred steering direction of the robot from *follow-path* in the form of a desired sector. It then used its sonar, following VFH (Borenstein and Koren 1991), to attempt to keep the robot in the free space nearest that sector. The *avoid* behavior maintained a local sector map. Similar to the occupancy grid used in VFH, *avoid* incremented or decremented belief in obstacles each sonar sensing cycle. The *pan-camera* behavior was responsible for turning the camera to the preferred steering direction (i.e., the center of the path).

Sensing Organization from Cognitive Psychology

The choice of motor control architecture can be viewed as providing a skeletal framework for the organization of sensing. However, as was seen in the previous section, the framework is very general and does not offer many insights into what types of sensors are appropriate for a task, if multiple sensors must be coordinated, or whether a task can be accomplished without perceptual models. This section summarizes insights from cognitive psychology about sensing and discusses how these insights can be applied to robotics. Cognitive psychology views sensing in terms of the inherent information that must be derived for a task. In particular, it offers a taxonomy of sensing information and some insight into when to use explicit perceptual representations. The performance of the 1995 CSM entry is used to illustrate the consequences of not considering the unique characteristics of each type of sensory information.

Taxonomy of Sensing Information

The choice of sensors and processing algorithms and how they are coordinated depends on the type of information that must be extracted. The traditional taxonomy of sensory information is a function of the mechanisms used for sensing: ***proprioception*** was derived from proprioceptors, which reported body movements relative to an internal frame of reference, while ***exteroception*** reflected how other receptors measured external information (Bruce and Green 1990).

Lee (1978) found this partitioning too restrictive for describing visual sensing activities and proposed an alternative taxonomy based on the content of the sensory information. In his taxonomy, ***exteroception*** supplies information about the layout of the environment and the position of relevant objects relative to the observer. Exteroception for the UGV competitions was used to determine the boundaries of a path visually or the remaining distance to an obstacle from sonar. *Proprioceptive* information concerns the movement of body parts relative to an internal frame of reference. Proprioception was used to determine where the robot has moved in the last few seconds. Finally, ***exproprioception*** reports on the position of the body, or parts of it, relative to the layout of the environment. Exproprioception was used in the 1995 entry to determine which way the camera mounted on the robot was pointing relative to the course boundaries.

This taxonomy can be helpful in integrating sensing and acting for two reasons. It directs the design process to focus on the information content of the sensors, rather than sensing modalities and algorithms. It also provides an opportunity for implicit assumptions to surface.

Figures 3 and 4 show the type of information associated with each behavior. Exteroception dominates because locomotion, especially road-following, is nominally concerned about where the road is. Certainly the *follow-path* in 1994, and

the *follow-path* and *avoid* behaviors in 1995, relied on exteroceptive information. However the CSM entries also depended on proprioception and exproprioception. The experiences of the 1995 team highlight the dangers on concentrating solely on exteroception.

The need for proprioception in the UGV competitions arose in two ways. First, the entries for both years needed proprioception for dead reckoning about the robot's position. In 1994, the robot used the *move-ahead* behavior to continue straight ahead long enough to get past a visual distraction In 1995, *avoid* relied on dead reckoning to update its local map of obstacles by estimating how far, and in what direction, the robot had traveled between readings. Second, proprioception was anticipated to be needed in the 1995 competition to sense inclines. The 1995 competition course included a wooden ramp which the robot could not go up at the maximum speed of five mph. The team created the *speed-control* behavior, which used a set of mercury switches as an inexpensive inclinometer. The *speed-control* behavior was not used in competition because the robot's maximum speed was left at 1 5 miles per hour (from 1994).

It is easy to underestimate the importance of proprioception. *Omnibot* and its successor *C2* had a very loose steering mechanism. The steer motor had shaft encoders, but because of the loose linkages, gearing, and grass conditions, the actual angle the robot turned might vary up to 15 degrees from the steer command. In the 1994 entry, this lack of accurate steering proprioception did not pose a problem. The vision processing for *follow-path* was simple and ran very fast. The robot could adjust its position relative to the course boundaries every 150 milliseconds. The high update frequency compensated for the lack of accurate proprioception about where the robot was actually steering.

However, in the 1995 entry, the algorithm was frequently unable to extract the boundaries because of lightly painted lines with low contrast to dead grass. In ambiguous cases, *follow-path* returned a "hold course" (no-op) command Even though the more advanced vision code ran at the same update rate of 150 milliseconds, the poor visual conditions effectively dropped the update rate to closer to 500 milliseconds If the robot had not turned sufficiently in the previous update cycle to stay on the path, it would continue to drift off course. This exacerbated the problems of detecting the course boundaries in the next vision update and accurately localizing the robot on the obstacle map.

The *pan-camera* behavior in the 1995 entry also relied on exproprioception, but was implemented as if it used exteroception. This led to an overall poor performance For the 1995 competition, the CSM team used a variant of Dickmanns and Graefe (1988) which allowed the camera to pan. The use of pan mechanism was intended to permit the camera to follow the road while the vehicle turned to avoid obstacles. During implementation, the *pan-camera* behavior was given only exteroceptive information; it computed where to position the camera based on the camera's current relationship to the path. If the robot turned before the next image was collected, the camera position was incorrect because the robot body (and the camera) were now in a different position. Because of the substitution of exteroceptive information for the desired exproprioceptive information, the *pan-camera* behavior did not behave correctly on hairpin turns or slopes. The robot would turn in such as way as to cause the incorrectly positioned camera to interpret the right edge as the left boundary. This resulted in a steer command that drove the vehicle out of bounds.

Two Perceptual Systems

As noted earlier, many reactive systems make use of reflexive behaviors, relying on identifying a stimulus-response relationship between the robot's sensing capabilities and its task environment Behaviors which can rely on stimulus-response like relations are generally faster than those which require intermediate processing However, many behaviors make use of a local, internal representation. An important design issue for sensing is to discern when no model is needed. The need for explicit models is also a point of contention between traditional and ecological psychologists (Bruce and Green 1990).

The traditional—or indirect perception—camp maintains that some sort of inference or mediating process is required to extract the desired information from sensing observations. This traditional approach assumes that the mediating process will be operating over some model of the percept. The ecological, or direct perception, camp is based on the work of J. J. Gibson. Gibson argued that the necessary information can be obtained directly from the raw observations without constructing any intermediate representations or models. For instance, the robot does not have to ascertain whether an obstacle is a bush or a bale of hay in order to avoid it In this situation, the time to contact, τ, with an object is a function of the optical flow field associated with the object (essentially, 1/the rate that the observed object region is getting bigger). The object does not have to be identified as a bush or bale. τ is an *affordance* of the environment for detecting collisions that does not require the object to be modeled in any way. Gibson's approach is often referred to as the *ecological* approach because affordances are a function of the organism's ecological niche in its environment.

The two approaches to sensory processing are not necessarily mutually exclusive Neisser (1989) cites physiological evidence that direct perception (i.e , the ecological model) describes the oldest form of perception, supporting stimulus-response types of behaviors A second perceptual system evolved later to handle *recognition* types of activities, i.e., the traditional model-based view. The two perceptual systems dichotomy suggests that the robotic designer must determine whether there is an affordance in the environment that is detectable by the available sensors, or if a recognition style of perception is better suited for the task

Unfortunately, the concept of two different perceptual systems does not resolve the question of what is direct perception versus a recognition-like model. Many animals have evolved specialized receptors; for example, frogs have been

shown to respond visually to large moving objects (predators) and small moving objects (prey) (Cervantes-Perez 1995). In some regards, the use of subregions (i.e., selective perception) and directional filtering could be considered a software approximation of specialized retinal receptors. The issue of execution speed is not a good metric; our experiences suggest that recognition, or model-based, perception can be just as fast as direct perception methods. Both the 1994 and 1995 vision routines updated every 150 milliseconds. Our conclusion is that the real issue for a designer is to determine what information is in the environment and in what form it can be best exploited in a computationally efficient manner.

The 1994 and 1995 entries illustrate both sides of the dichotomy. In 1994, the *follow-path* behavior used a direct perception approach. The brightness of the white line served as the affordance for the boundary. The resulting perception was simple to program and executed rapidly. However, it was brittle. Because the robot was reacting to the centroid of the brightest pixels in the image, it did not exploit any knowledge about lines. The robot veered 30 degrees off course in one update cycle to move toward a judge's white shoes, even though the angular change in the course was much smoother.

In 1995, the *follow-path* behavior used a simple model of the road to attempt to construct a more robust algorithm. The vision processing component used what was known about the width of the course and the maximum rate of curvature to determine six subregions in the image where the boundaries should appear. It also used knowledge about the maximum rate of curvature to filter out edges which were not in the expected direction of the boundary. The best line for each subregion was identified by a Hough transform. The slope of each line in the subregions formed a mask, which served as an index function to a table of steering commands. By operating over subregions instead of the entire image and filtering out unlikely edge candidates, the model-based algorithm executed as quickly as the 1994 direct perception approach. The algorithm was demonstrated to be able to ignore distractions (such aswhite shoes or other white lines going perpendicular to the course) under test conditions. In the field, the algorithm did not perform adequately because of the errors in exproprioception compounded with the difficulty in extracting edges from faint white lines.

Conclusions: Guidelines

In this chapter, I have attempted to give some practical insight into the challenges and issues associated with sensing for mobility. The organization of sensing is left largely to the designer; motor control architectural styles provide a general framework but not much more. In order to aid the software designer, we offer a set of guidelines in keeping with the *action-oriented perception* philosophy.

- *Decompose the task(s) into functional desired activities.* This process will

likely have to be repeated several times as the designer attempts to get at the heart of the task and negotiates for sensing resources. The first iteration should result in the establishment of the style of architecture: hierarchical, reactive, or hybrid. Future iterations can then further partition the activities into behaviors or deliberative functions.
- *Establish the perceptual context.* The perceptual context of the mobile robot (the demands of the activity, the available sensors, and the inherent properties of the target environment) will influence how sensing needs can be satisfied and whether direct perception can be safely used. The 1994 entry used the competition to define a narrow ecological niche, while in 1995 the goal was to be able to follow different paths in both outdoor and indoor conditions.
- *Define the sensing needs for each activity in terms of proprioception, exteroception, and exproprioception.* This will influence the choice of sensors, requisite software processing and indicate the type of coordination necessary between sensors. The 1995 CSM entry typified the types of problems that can result from not adequately considering the task needs in terms of the information taxonomy.
- *Determine whether each sensing objective can be accomplished via direct or model-based methods.* The designer must establish whether there are any affordances that can be exploited by the sensing configuration. However, constraints on how the robot should act in response to unanticipated situations should not be overlooked. If the environment is well engineered and situations like judges wearing white shoes coming into view should not occur, then a direct perception approach such as that used by the 1994 CSM team may suffice. If a more portable behavior is desired, then a model-based approach may be more appropriate, as exemplified by the 1995 entry.

Even with these guidelines, the coordination and control of sensing is still as much of an art as is designing behaviors. Clearly the utility of any system depends on the functionality, simplicity, and reliability with which the components are arranged. I hope that as researchers pursue mechanisms for proving the correctness of configurations of behaviors, they will also find ways of constructing efficient systems which exploit the relationships between sensing and acting.

Acknowledgments

This work has been supported in part by NSF and ARPA under grant IRI-9320318 and the Colorado Space Grant Consortium. I would like to thank Dave Parish and Omnitech Robotics, Inc. for the donation of the *C2* base, the organizers of the AUVS UGV competitions, the many students who have been members of the CSM teams, and Dave Hershberger, who helped with the preparation of this chapter.

CHAPTER SEVEN

Vehicle Guidance Architecture for Combined Lane Tracking and Obstacle Avoidance

Bill Schiller, Yu-feng Du, Don Krantz, Craig Shankwitz, and Max Donath

The National Traffic Safety Board has indicated that in excess of 40 percent of heavy truck accidents are due to driver fatigue. Technologies that provide for collision avoidance and lane following can potentially "reduce urban and rural freeway accidents by a minimum of 30 and 35 percent, respectively. Accident reductions of this magnitude would eliminate approximately 71,000 accidents per year and $700 million in accidents costs" (Preston, Holstein, Orfeson, and Hoffman 1995).

Together with the Minnesota Department of Transportation, we are working on systems capable of taking over control of a semi tractor-trailer when drivers become incapacitated. Such systems must be able to provide for integrated speed and headway regulation, roadway following, obstacle (and other vehicle) detection, and collision avoidance. At the minimum, the control system should be able to drive the vehicle safely over to the shoulder (if and where it exists), slow down, and come to a complete stop.

In the first phase of the SAFETRUCK Program, we have been focusing on subsystems which prevent road departure accidents. The goal for these subsystems is to keep the vehicle in its lane The tractor-trailer experimental testbed uses a new high accuracy, high bandwidth differential global positioning system (DGPS) capable of providing a real-time 5 Hz uninterpolated position of the tractor This system is being integrated with an inertial measurement unit using a Kalman filter (Shankwitz, Donath, Morellas, and Johnson 1995; Bodor, Donath, Morellas, and Johnson 1996) We have experimentally shown that, after a period of continuous satellite lock, it is possible to track the motion of a truck using this DGPS and achieve dynamic accuracies with mean offset errors better than 4 centimeters (Bajikar, Gorjestani, Simpkins, and Donath 1997). Using the new steering system that we designed, we have further demonstrated that our system can take full steering control of a Navistar 9400 tractor trailer and keep it in its lane (Morellas, Morris, Alexander, and Donath 1997). This was accomplished using DGPS and a yaw rate gyro as the primary sensors. In subsequent phases, additional vehicle

guidance systems will be investigated, including, for example, systems based on magnetic striping sensed by vehicle-mounted magnetometers. For safety reasons, it is important to test potentially high-risk vehicle sensing and control strategies on small vehicles before implementation on the semi tractor-trailer. The autonomous land experimental vehicle (ALX) was first designed to provide such a safe platform.

ALX was designed to solve a specific problem: navigating an a priori unknown outdoor road with paint-stripe lane boundaries, under varying sunlight conditions, while avoiding obstacles of unknown size, shape, number, and placement. The intent was to test our concepts for integrating lateral guidance and collision avoidance procedures. We do not expect that all the subsystems described in this chapter will be used on the truck testbed. However, the software and communication architecture described here represents an early prototype that allowed us to experiment with a variety of hardware configurations, software tools, system and sensor data integration issues and interprocessor communication procedures. We have developed two collision avoidance algorithms: one based on path tracking and the other on the virtual bumper. The differences between these algorithms is discussed here as well as our reasoning for selecting the virtual bumper for further development on the tractor-trailer. The vision-based guidance system described in this chapter is a precursor "placeholder" for introducing and evaluating lateral guidance within the overall system. Vision subsystems have limitations vis a vis all-weather operation and are not at present being considered for the truck. In fact, that is why we are currently using a DGPS system for the truck's lateral guidance The ultrasonic range sensors have serious limitations as well, including bandwidth, signal to noise ratio, lateral spatial resolution, and wind effects. We expect that we will be using microwave and millimeter wave-based radar devices for obstacle detection and collision avoidance rather than ultrasonic range sensors.

We also developed a real-time simulated environment to aid in the design process. This simulated environment allowed us to debug the software and perform experiments indoors during the winter months. With the simulated environment, we were able to perform more efficient and controlled system testing as well as reduce our total system development time. The key was our emphasis on maintaining a high fidelity between the simulation and the final real-world implementation. Design could be performed on the controller using the simulation with confidence that the results of each iteration of test and redesign would apply directly to the final vehicle's performance outdoors

During our design process, we studied many of the existing systems as reported in the literature (Du 1995; Shankwitz and Donath 1995). A survey of the literature showed that although there were many systems on robotic vehicles that were capable of detecting obstacles and avoiding collisions automatically, many of them were limited to indoor operation, relied on known environments, or suffered from inadequate lateral control (roadway following) functionality.

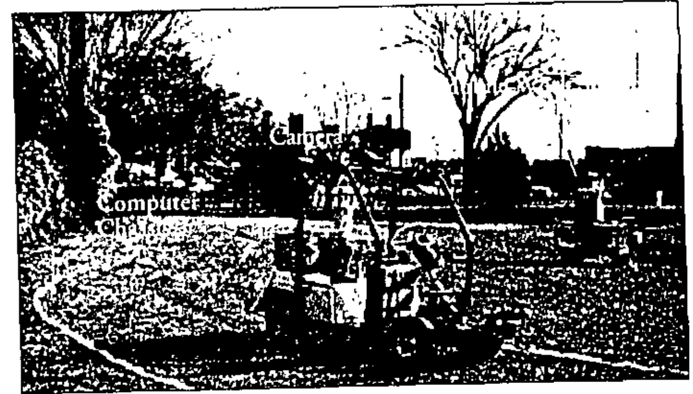


Figure 1 ALX

System Design

ALX is based on an electric golf cart chassis and is equipped with control computers, actuators and feedback sensors, dead-reckoning sensors for position estimation, an array of up to sixteen ultrasonic sonar range sensors with overlapping fields for obstacle detection, and a vision system for roadway sensing.

System Device Overview

ALX (figure 1) was designed around a centralized processor system with several subsystems as shown in figure 2. Some of these subsystems are described briefly below.

Dead-Reckoning System

The dead-reckoning system consists of a set of dead-reckoning sensors, which include one two-axis fluxgate magnetometer, two clinometers, and one odometer (rear wheel encoder). The dead-reckoning system predicts the vehicle's position in Cartesian coordinate space based on position and velocity measurements from the suite of dead-reckoning sensors. Other entities sensed by ALX (roadway edges, obstacles, etc.) are then referenced using ALX's current computed position and orientation, i.e., the results of the dead-reckoning computation.

As with any other dead-reckoning systems, the certainty of ALX's coordinate accuracy decays as the traveled distance increases. This decay is due to slippage between the wheels and the road. The methods described here have demonstrated that successful operation is still possible despite the accumulated error of

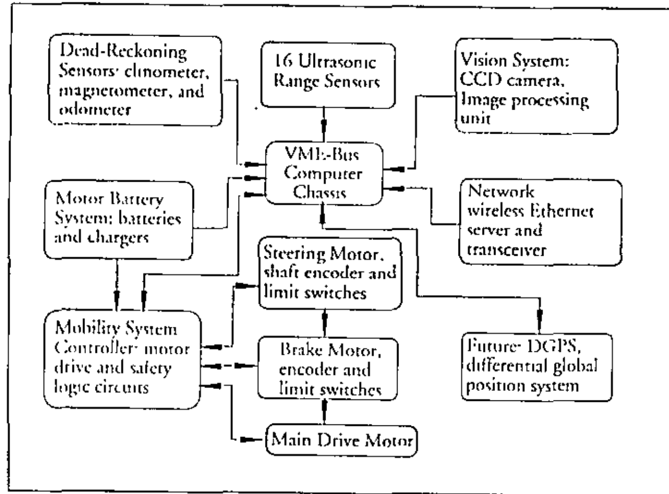


Figure 2. Hardware system layout of ALX

such systems. A system for continuously correcting dead-reckoning results can improve the situation. DGPS is being tested for the truck and may be used later on ALX to correct position estimates.

Ultrasonic Range Sensing System

The ultrasonic range sensor system is used to detect obstacles and to allow ALX to compute and execute a trajectory around these obstacles. The information collected is accumulated and updated to construct a dynamic virtual map (see later) which provides information on the local environment to ALX. The range sensors are arranged on ALX to provide the ability to detect obstacles in front and to the side as shown in figure 3. The current design polls sensors at 10 Hz within each sensor group (1 to 16 sensors may be in each group). Only one sensor group is fired at a time. The firing sequence for the sensor groups is programmed to avoid crosstalk between sensors.

A VME-bus ultrasonic range sensor processing board was designed for ALX. The function of the sensor processing board is to coordinate the firing of the sensors and avoid potential crosstalk interference between them, measure the time-of-flight of the ranging echo pulse, and report range information to the main processor via a serial link which operates up to a maximum rate of 9600 baud.

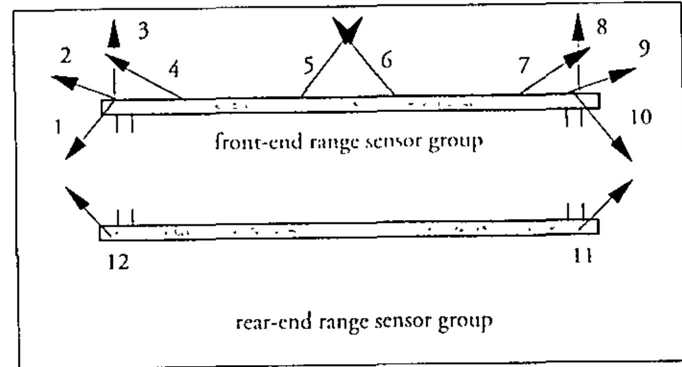


Figure 3. Range sensor layout

Vision System

The vision system provides visual sensing of the roadway for ALX. It first determines the edges of the road from images taken by a CCD camera; then it passes this information to the microprocessor on ALX.

The image processing unit for the vision system is based on a DataCube MaxTD 200 image processor which incorporates a pipeline-processing capability (MaxVideo 200). The communication link between the image processor and the main control computer (a MVME 147SA) on ALX is established via ethernet (figure 4).

Mobility System Controller

The mobility system controller (MSC) contains the high-power control unit for the main drive motor, the steering motor, and the brake motor. The MSC uses pulse-width modulation (PWM) to control the speed of the main drive motor. The PWM frequency and duty cycle are supplied as inputs. The nominal PWM frequency is 10 KHz.

The MSC provides protection against illegal input commands (e.g., both "forward" and "reverse" commands simultaneously being active). It also provides a mechanical relay disconnect for the main drive motor when ALX is not in a safe state. The disconnect must be energized to supply power to the main drive motor. The MSC also detects the limit switch states of the brake and steering motors and cuts off drive power to the appropriate motor when a limit is reached.

Real-time Control Architecture

A real-time control architecture is used in our autonomous vehicle because of

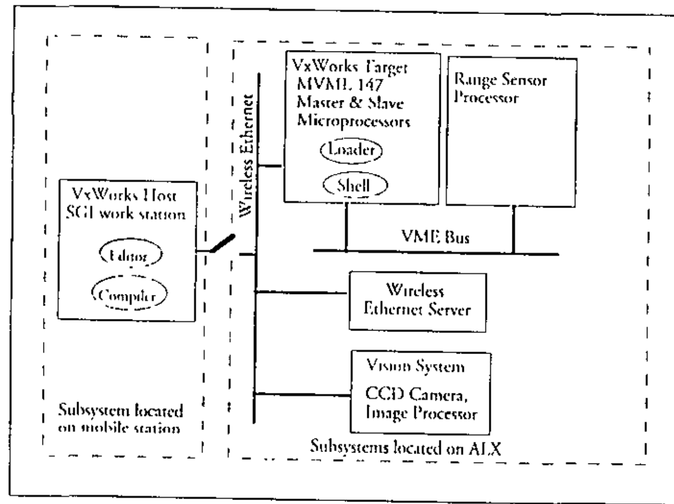


Figure 4. Cross development setup.

the ability to respond to events as they occur in the real world. For instance, consider the case when the vehicle is performing a specified task (i.e. road following) and an event occurs (i.e., obstacle is detected.) A real-time control approach allows for an immediate response to the event and for taking appropriate action (i.e. obstacle avoidance.) A traditional sequential controller on the other hand, will respond to the event based on a predefined order in which events are handled. Responding with actions in this predefined order usually will not satisfy time constraints for successful vehicle control.

By using Wind River's VxWorks real-time operating system to construct our computing architecture, we were able to develop an embedded real-time control system which differs from traditional control systems in that the sequence of actions is determined by real-time events occurring in the outside world

Cross-Development Environment

The hardware for the ALX computer system includes one UNIX host system (an SGI workstation), two Motorola MVME 147SA microprocessors (the VxWorks target) and a DataCube image processor. The SGI workstation is used for software development and also contains an "image" of the VxWorks system, which is used to boot the MV 147SA targets. A Windata wireless Ethernet is used for intercomputer communications. The cross-development setup is illustrated in figure 4.

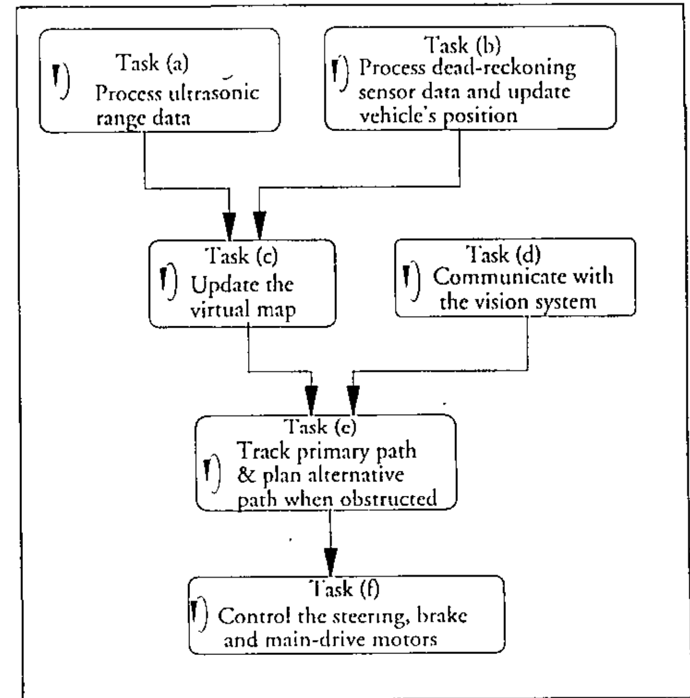


Figure 5. ALX multitasking structure

Software development for ALX takes place on the UNIX host: an SGI workstation which operates off an uninterruptable power supply (UPS) and can therefore be taken outdoors. Program modules are written in C using the VxWorks library are compiled with a C cross-compiler and then downloaded to ALX. ALX then runs independently of the host workstation.

Multitasking Control Structure

The real-time computer control system for ALX is based on the complementary concepts of *multitasking* and intertask communications.

Multitasking provides the fundamental mechanism for control and reaction to multiple, discrete real-world events. The basic multitasking environment is provided by the VxWorks real-time kernel. Multitasking creates the appearance of many programs executing concurrently when, in fact, the kernel interleaves

their execution on the basis of a scheduling algorithm. In the ALX control program, there are several tasks running "concurrently," as shown in figure 5.

The VxWorks multitasking kernel uses an interrupt-driven, priority-based task scheduler. With this preemptive priority-based scheduler, each task is assigned a priority, and the kernel ensures that the CPU will be allocated to the highest priority task that is ready to run.

The primary intertask communication mechanism used in the MVME 147SAs on ALX is message queues. Message queues allow a variable number of messages, each of variable length, to be queued in a first-in-first-out order. Any task can send and receive messages from different message queues, and multiple tasks can send and receive from the same message queue.

Visual Sensing of the Roadway

The goal of the vision system is to determine the location of the edges of the road in screen coordinates. To accomplish this goal, ALX uses a single CCD camera which is mounted six feet above the ground and is pointed forward and down. This camera provides an image that covers a field of view from five to thirty feet ahead of the cart; its average width (trapezoidal mapping) is twenty feet. The position of the lines in these images is determined using a blob analysis (or connectivity) algorithm.

Comparison of Blob Versus Edge Detection

Initially, two vision algorithms were evaluated: blob analysis and edge detection using a Laplace of Gaussian (LoG) operator. The evaluation of these approaches was based on three main criteria: (1) the ability to find the line in the image, (2) the ability to eliminate or ignore noise and reflections (large bright specular regions due to the sun), and (3) the ability to reject obstacles lying on the road that may be confused with the road edge. This evaluation was based on images consisting of white lines spray painted on green grass under varied lighting conditions. The algorithm which proved to be the most complete solution was selected.

Both blob analysis and edge detection find the edges of the road when the images are good. The advantages of blob analysis become apparent when the images have poor lighting and exhibit specular reflections due to the sun. Noise in the image is readily eliminated by discarding the blobs based on pixel count. All blobs with less than a minimum quantity of pixels are ignored. If a reflection creates a blob greater in pixel quantity than the minimum cutoff, the blob is most likely not shaped like a line. Therefore, this reflection can be eliminated based on measures of shape (i.e., perimeter^2/area). Conversely, edge detection

does not handle reflections very well. Although the LoG operator provides smoothing to eliminate noise, it is still very susceptible to noise in the images. This is because reflections of the sun on the grass often exhibit sharper contrast than the edges of the road. When this occurs, the edges of the reflection are incorrectly considered to be lines.

Blob analysis is also superior to edge detection in rejecting the effects of obstacles on the road which appear in the image. The obstacles which ALX encounters are not shaped like a line. Therefore, blobs that are representative of obstacles are rejected as potential line data based on shape measures. Edge detection does not eliminate the obstacles as well as blob analysis. Edge detection finds the edges of the obstacles. If the obstacle is on or near the edge of the road, it is automatically considered part of the line and, as a result, skews the line data.

Due to the simplicity of the approach and the overall ability to eliminate noise and obstacles from the data, we decided to use blob analysis for the vision algorithm. Although many of the shortcomings of edge detection can be eliminated by adding further qualifying criteria for potential lines, we found that blob analysis provides adequate results without adding complexity to the algorithm.

However blob analysis has one major short coming. Blob analysis is only as good as the threshold selected for the image. However, it has been found experimentally that using a histogram-based thresholding method consistently reduced these images to the desired data (lines with some noise and obstacles).

Blob Analysis

Blob analysis is a relatively straightforward algorithm. It involves first thresholding the image (often first smoothed with a Gaussian filter). The resulting binary image is then evaluated for connectivity (connectivity analysis). In other words, neighboring pixels that are "on" or that have a value of one are grouped together. The resulting groups are considered blobs. The final step of blob analysis is gathering data about the blobs, for instance, their shape or size as well as their relationship to one another.

Although the blob analysis is straightforward, it is usually computationally expensive. Connectivity analysis alone has several algorithms designed to enhance computation time. Determining the relationship between blobs can also be costly.

The blob analysis algorithm discussed here addresses the problem of minimizing the computation time as our highest priority. Blobs relationships to one another are not calculated, as this is not a requirement of this system. Information about a blob being a parent or a daughter to another blob is not required for determining the edges of the road. Also the connectivity analysis algorithm has been created to allow for efficient determination of blobs. The details of the algorithms are discussed in the following subsections.

DataCube Image Processing System

The vision algorithm is executed on a MaxTD pipeline-based image processor. The MaxTD is a VMEbus-based computer that contains a Motorola MVME 167 (68040-based) processor and MaxVideo 200 pipeline hardware. This vision algorithm uses both the MV200 pipelines and the MVME 167 to perform the image processing (see figure 6 for flow chart). Similarly, the following subsections describe first the pipelines that are executed and then the processing performed by the Motorola processor.

The blob analysis requires four data pipelines. The first pipeline is used to capture the image. The results are passed to the second pipeline, which collects histogram data and also performs a thresholding operation. Next the third pipeline performs a three x three morphological filter. The third pipeline then uses the filtered image to create a run-length encoded image as well as a chain-coded image. The output of the morphological filter is displayed on the screen by a fourth pipeline with lines fit to appropriate blobs (the line fitting is discussed later). The input/output pipelines are not discussed as they are trivial implementation, but the two main processing pipelines will now be discussed.

Data Pipelines

A pipeline is a software-programmable hardware path on the MaxVideo 200 that performs a desired task (e.g., convolution) on the input data. The pixel values are sent through a pipeline as a stream of data, and the desired operation is performed on the image. All data pipelines are programmed using an application programming interface supplied by DataCube called ImageFlow.

The histogram and thresholding pipeline (Pipe2) is a multidestination pipeline. The destination surfaces are oApViewSurf and oMem4InSurf (figure 7). The DataCube system defines a surface as a block of memory which can be used by a pipeline. The oApViewSurf stores the data from the histogram operation, and the oMem4InSurf stores data from the thresholding operation. The important thing to remember about this pipeline is that having two destination surfaces saves time by performing two processes simultaneously. Of course the processes cannot have conflicting resources (both processes cannot use the same element). The oMem4InSurf stores the thresholded image and is the input for the next pipeline. Note that although the data for the histogram has been extracted by a pipeline, this data is evaluated on the MVME 167 for selecting an appropriate threshold value. In order to be time efficient, threshold selection is performed on the MVME 167 after completion of this pipeline while the next pipeline is executing. The threshold value selected by this method is then used on the following frame.

The other pipeline (Pipe3) is also a multidestination pipeline. The destination surfaces are oMem0InSurf, oMem1InSurf and oMem3InSurf and these

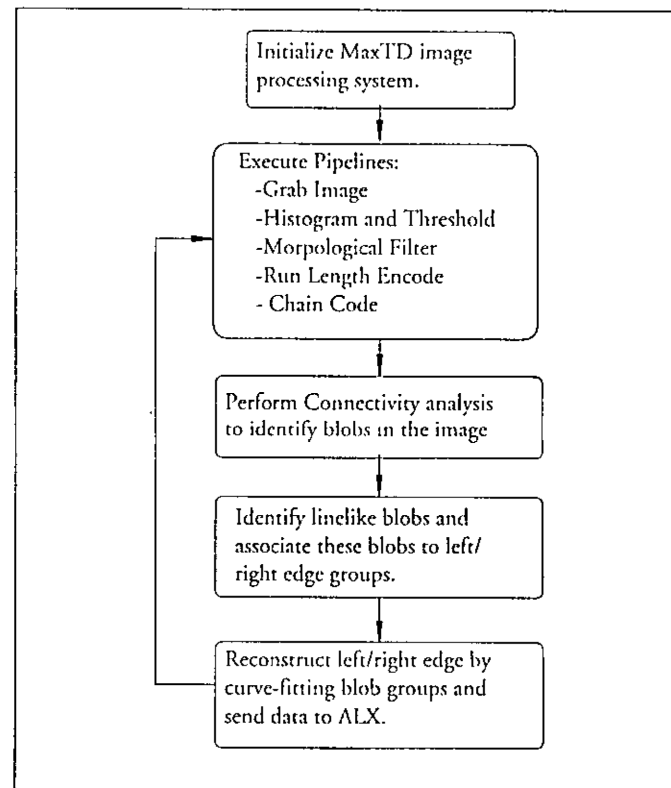


Figure 6. Simplified flow chart of blob analysis vision algorithm

surfaces store data from the morphological filter, the chain-coding process and the run-length encoding process, respectively (figure 8).

The morphological filter is used to improve the quality of the thresholded image. This filter moves across the image looking at a three x three pattern of pixels and replaces the center pixel based on a look-up table (LUT) and the pixel patterns. This filter effectively eliminates isolated points and enhances the connectivity of blobs by filling in gaps in the connected regions. The output of the filter is passed on to the chain-coding and run-length encoding processes and is also stored in oMem0InSurf, which is used to display the results.

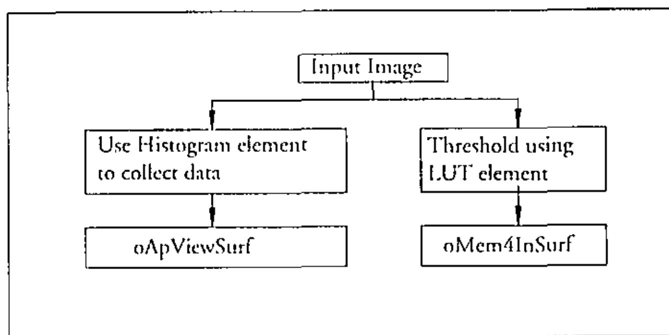


Figure 7. Threshold and histogram data pipeline.

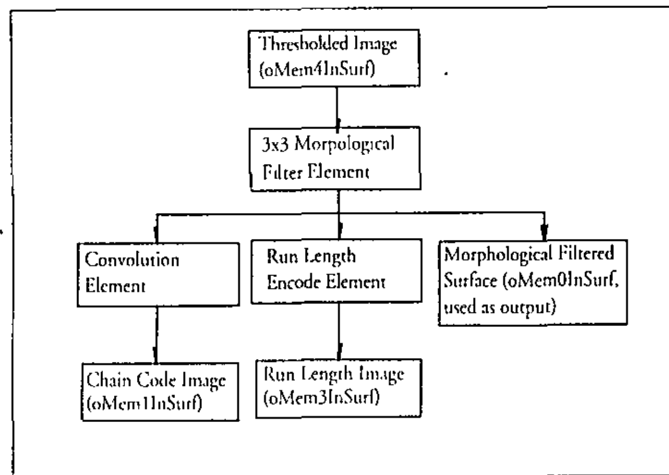


Figure 8. Morphological filter, run-length encode, and chain-code data pipeline.

The run-length encoded image contains information on the position of 0 to 1 and 1 to 0 transitions on each line of the binary image and is used in connectivity analysis (its use is discussed in the connectivity analysis subsection). The run-length encoding process takes as input the least significant bit (the bit representing on or off in the binary image) and provides an eight-bit output. For each pixel value of the output image, the seven least significant bits contain a

A. Binary Image

0	0	0	1	1	1	1	0
0	1	0	1	1	0	0	0
0	0	1	1	1	1	0	0

B. Run Length Encoded Image

1	2	3	129	130	131	132	1
1	129	1	129	130	1	2	3
1	2	129	130	131	132	1	2

Table 1. Example of a binary image(a) and the corresponding run-length encoded image(b).

1	2	4
8	0	16
32	64	128

Table 2. 3 x 3 kernel.

count of the pixels since the last 0 to 1 or 1 to 0 transition in the binary image (counting from left to right). The most significant bit of each pixel value of the output image is the value of the binary image. Note that on the run-length encoded surface a count of 0 represents a distance of 128 times *N* (*N* being any positive integer) pixels from the last transition. Table 1 shows an example binary image and the corresponding run-length encoded image.

The chain-code image is the same as that used in Wensel and Seida (1993). Chain-code images usually contain information regarding the direction and length of a line segment of connected pixels. In this case, information about connected neighbors is only recorded. This is accomplished by convolving the image with the three x three kernel depicted in table 2.

Applying the above kernel to a binary image results in the chain-coded image having values from 0 to 255 depending on the values of the connected neighbors.

At this point the data pipelines used by this algorithm have been defined, but how are they executed? The input and output pipelines are initiated and execute continuously, but Pipe2 and Pipe3 cannot be continuous because of conflicts in

resources. These pipes are programmed such that Pipe3 begins whenever Pipe2 is finished Pipe2 begins when a dummy event is triggered. Therefore, a triggering of the dummy event causes both Pipe2 and Pipe 3 to execute in that order. To allow for efficient operation, the dummy event is triggered after the connectivity analysis has extracted the required data from the run-length encode and chain-coded images. Figure 9 shows the flow chart of the vision algorithm with the pipelines executing in this manner. This allows the pipes to execute while the MVME 167 is processing other data Running the processes in parallel in this fashion accounts for approximately a twenty-five percent reduction in cycle time.

Connectivity Analysis

Once the run-length encoded and the chain-coded image have been generated by the Datacube MaxVideo 200, the next step is to perform the connectivity analysis. The connectivity analysis involves extracting desired information for each blob from the processed images. This process could have been implemented via Datacube pipelines; however, the programming process would have been tedious Since the data to process has already been greatly reduced, a less involved programming task, which is also time efficient, can be implemented on the MVME 167 board. The connectivity analysis and the remainder of the program are executed on the MVME 167.

To extract the connectivity information, two data structures were created. The first is for a point in a blob, BlobPnt. The BlobPnt structure is a statically allocated array and contains information on the screen position of the blob point as well as the index (for the BlobPnt structure) of the next point in the blob. The second structure is for the segments of blobs, RawBlobs. RawBlobs is also a statically allocated structure and contains the indices of the beginning and last point of the blob in the BlobPnt structure. It also contains the pixel count for the blob.

The above structures do not store the blob number for each pixel. The blob number that each pixel is a part of is stored in that pixel's memory location in the run-length and chain-code images (the msb of the blob number is stored in the run-length image, and the lsb of the blob number is stored in the chain-code image, allowing for up to 65536 blobs) This approach quickly allows for checking as to which blob a pixel is a member of. The run-length and chain-code image are already being traversed using pointers. Checking a pixel's blob number involves incrementing the pointers and performing a logical OR. More importantly, this memory mechanism allows checking what blob a pixel belongs to without accessing the BlobPnt structure This reduces the complexity of the connectivity analysis as well as the complexity of the BlobPnt data structure

Data for the RawBlobs and BlobPnt structures are collected using the run-length encoded and chain-coded images (see figure 10 for the flow chart). The run-length encoded image is used to locate all the pixels that are set in the image,

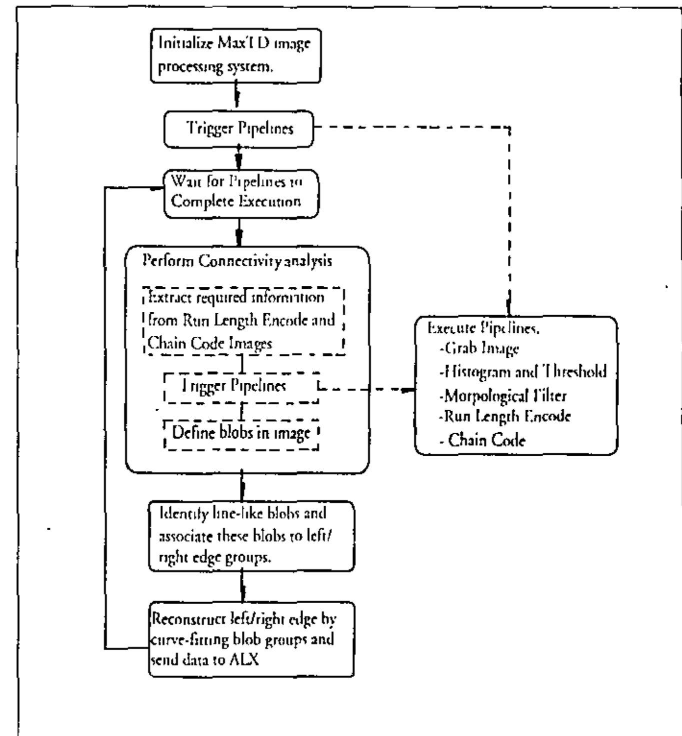


Figure 9. A modified flow chart for the vision algorithm.

and the chain-coded image is used to determine connectivity at these points The image is traversed right-to-left and top-to-bottom by jumping only to set pixels using the information from the run-length encoding. The chain code is now used to determine if the neighbors above or to the right of the run of set pixels are also set (pixels below and to the left have not been visited yet). If there are no set neighbors, all of the points in the run are added to the BlobPnt structure and a new RawBlobs is added. The number of the new blob is now stored in the run-length and chain-coded image in the location of the set pixels for that blob. If there are set neighbors, the blob number of the RawBlobs is determined from the values at that pixel location of the run-length and chain-coded images. The run of set pixels is then added onto the appropriate RawBlobs structure and blob

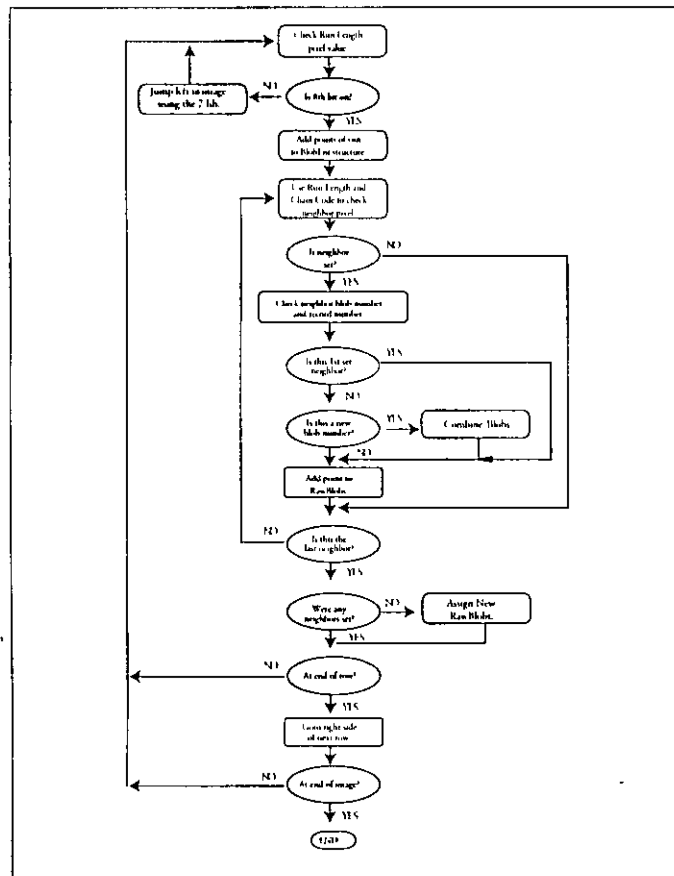


Figure 10. Connectivity analysis flow chart

number is stored for each run point. When one run is connected to more than one RawBlob, all of the blobs are connected into one blob.

Blob Association and Curve Fitting

After the image has been completely traversed and the information is collected for the RawBlobs, the next step is to determine which blobs have acceptable line

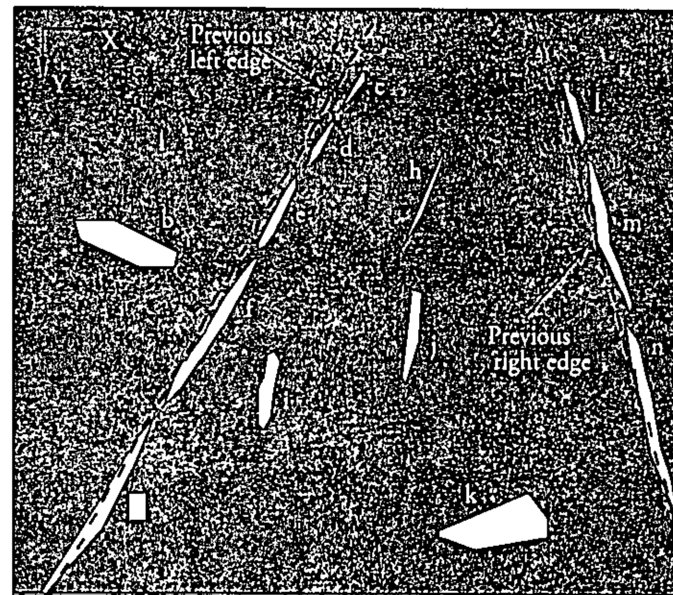


Figure 11. Binary image of the road with previous image lines.

shapes and to save these blobs. There are two criteria that are used for this evaluation: blob area and blob elongation (perimeter^2/area).

Blobs that meet the minimum area and elongation requirements are saved in the GoodBlobs data structure. The data in GoodBlobs are considered as potential candidates for segments of the road lines. These blobs are associated with the left-side edge, the right-side edge, or are discarded; this process is called blob association.

To carry out this function correctly, a memory mechanism was used. The program keeps track of the road edges reconstructed in the previous image frame and then uses them as references for the current image's blob association. This mechanism is based on the assumption that the image-processing speed is reasonable in terms of ALX's operating speed so that the positions of the road edges in two consecutive images do not differ significantly. As shown in figure 11, with the previous edges (dashed lines) as references, some of the linelike blobs are associated together to represent either the left or right roadway edge; and some line-like blobs (for example, blob "*h*," "*j*" and "*i*") are discarded because they cannot be associated with either the previous left edge or the previous right edge.

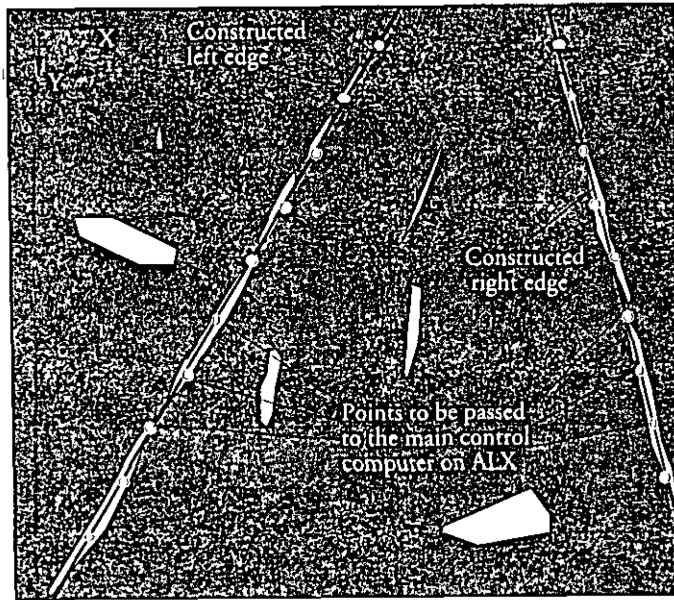


Figure 12 Reconstructed road edges.

After the blob association, a least-square curve fitting approach is used to construct a second-order polynomial curve which represents each of the two roadway edges. To represent the constructed road edges, ten evenly-spaced horizontal lines are scanned across the image, and the intersection points between the horizontal lines (dashed in figure 12) and the left (or right) edges are used to represent the left (or right) edge (figure 12). These intersection points are then transferred to the ALX computer for processing the road data. The results of the morphological filter and of the blob analysis and final road boundaries are displayed by the output pipeline on the MaxVideo 200.

Control Algorithms

We have developed and evaluated two separate control schemes for ALX: Path Tracking and the Virtual Bumper. Both approaches combine road following and obstacle avoidance for control of the vehicle and are discussed in this secti

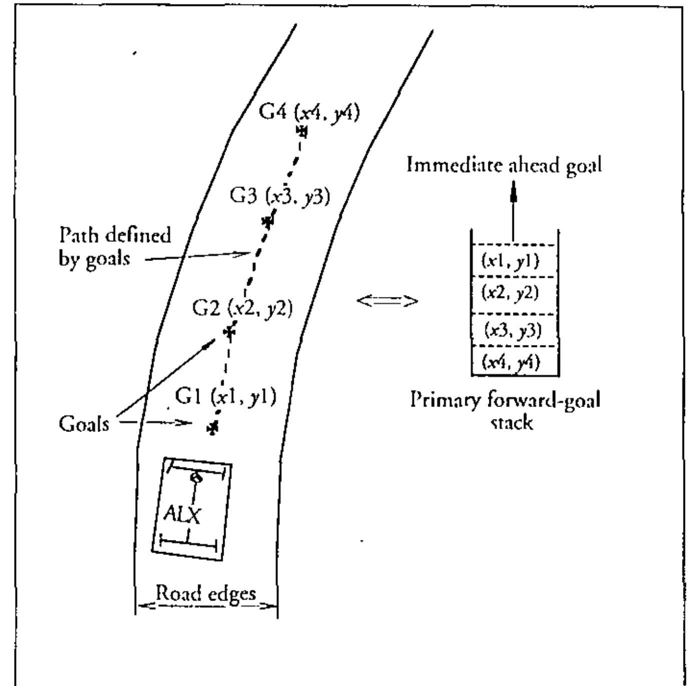


Figure 13 Illustration of the goal stack concept.

Path Tracking

The first approach we developed is Path Tracking. In this method we use the reconstructed road edges and a goal-generation scheme to form a path defined by goal points. At the same time a virtual map of the vehicle's environment is constructed using the range sensor data. When an obstacle is detected, the virtual map is used for planning an alternate collision-free path. The vehicle follows the prescribed path using a pure-pursuit algorithm.

For the path-tracking algorithm, the path, generated from either the visual processing unit or from an alternative path-planning algorithm used when the vehicle is obstructed, is represented by a finite number of goal points on the center-line of this path. The goals are defined in world coordinates and represent the positions which the vehicle is directed to reach. The coordinates of the goals are stored in a *goal stack*. Figure 13 shows an example of a path and

responding goal stack. The purpose of the goal stack is to allow ALX to retrace its steps if necessary to back up and go around an obstacle

Once the immediate-ahead goal is determined, ALX uses a heading-control method which continuously (20 Hz) calculates the heading errors and issues steering commands accordingly.

Range Sensor Processing and Obstacle Avoidance

The on-board ultrasonic range sensors are used to detect obstacles and to map ALX's local environment.

Construction of the Virtual Map

The spatial information processing algorithm of ALX is based on the concept of a *virtual map*, a representation of the local area around ALX (see figures 13 and 14). This virtual map is constructed by a two-dimensional (2D) array and each unit of this array is called a *cell*. Since ALX is currently designed to operate only on a relatively flat surface, all the range sensor data is projected on this 2D Cartesian map, and sensor information is registered into the map cells. The virtual map is designed to be a square area about twice the length of the maximum sensing range along each edge and "scrolls" along under the robot as it moves. As ALX moves along, new "uncharted" areas appear on the leading edge of the map. Traversed areas "fall off" the trailing edge of the map, and all data associated with that part of the world are forgotten. The rationale for this is the short time frame for accurate position data provided by the onboard dead reckoning system. If ALX returns to an area previously visited, the position estimates will probably have accumulated enough error to make the saved data worthless.

The purpose behind constructing the virtual map is to determine the local traversibility; therefore, the map should have accurately and clearly delineated open space and obstacles. However, the use of ultrasonic range sensors can lead to a variety of systematic errors (Brown 1985; Beckerman and Oblow 1990). By systematic errors, we mean those errors that result from deterministic yet incorrect or biased interpretations of the raw data during processing.

Because of the relatively large beamwidth of the range sensor (about 15 degrees), we observed that there are several possible interpretations of the data from a given isolated scan. When combining data from different sensing positions or angles, erroneous initial interpretations will give rise to conflicts. To resolve conflict in the interpretation of the range sensor information, various stochastic methods have been used In these methods, for each sensor scan, the profile of probability of the cells in the sensor cone has to be recalculated, and then every cell has to be updated This is computationally expensive.

To identify and remove systematic errors arising from the processing of sparse sensor data, we have successfully adapted a nonstochastic method based on a multivalue labeling algorithm (Beckerman and Oblow 1990), which is briefly

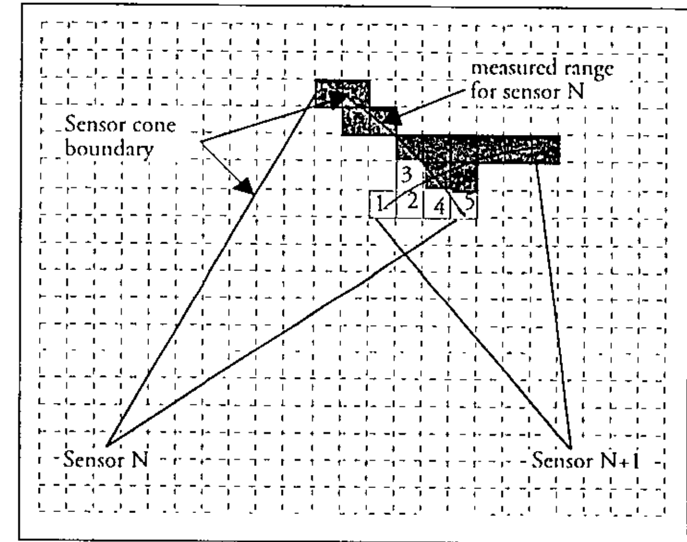


Figure 14. Numbered cells receive conflict label assignments from different sensor scans.

described below. This method is much less computationally intensive compared to the stochastic methods, yet adequately handles systematic errors mainly caused by the range sensor's relatively wide beamwidth.

In order to label the cells, we process the portion of the virtual map within the boundaries of the sensor cone up to the measured range distance. Each cell inside this cone on the virtual map can be given a preliminary label as follows: if the sensor returns a "hit" and the distance from the origin of the cone to the center of the cell under consideration is greater than the sensed range, then the cell is given an assignment of "OCCUPIED." Otherwise, the cell is assigned to be "EMPTY." For example, in figure 14, for sensor $N+1$, both cell 1 and cell 2 are labeled as "OCCUPIED," while both cell 4 and cell 5 (well inside the boundary) are labeled as "EMPTY."

Besides a label, each cell is also assigned an attribute, which is the range of the sensor's actual return. This attribute is used in a relabeling algorithm to determine which sensor was closer to the detected obstacle. The closer the range sensor is to the object, the more confidence we have in the returned range valı (provided that the returned range is greater than the sensor's minimum dᶜ tectable range). Thus, when conflicting assignments arise, the cell label is

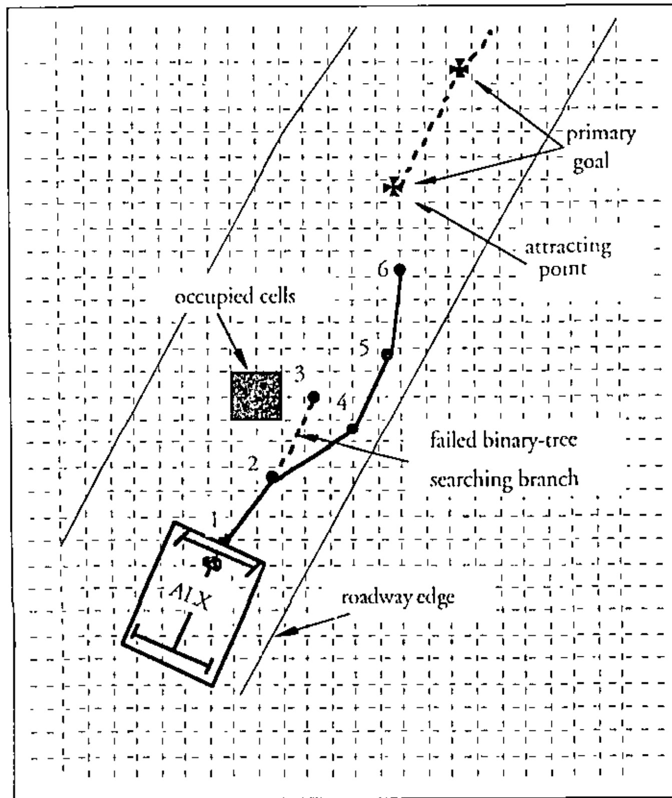


Figure 15. Searching for alternative path

signed based on the sensor which is closest. For example, in figure 14, the cells receiving conflicting label assignments will be finally labeled in favor of sensor $N+1$.

Not all systematic errors can be treated in this way, however. Such a methodology is best suited to the case where the data are sparse, the patterns of conflict are simple, and the corrections are physically unambiguous.

In order to overcome these limitations, we have successfully developed a system that can generate topologically correct maps on line in real-time for ultrasonic range sensors. The methodology, based on self-organizing neural net-

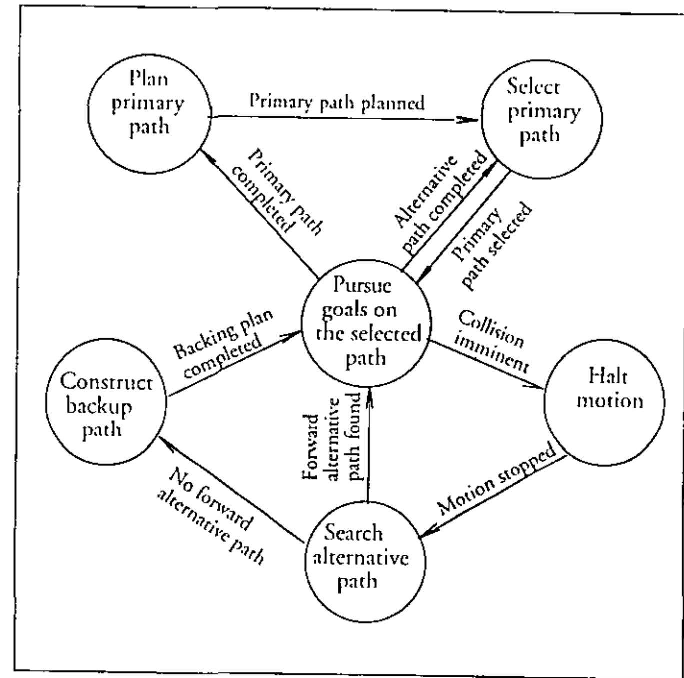


Figure 16. State transition diagram for the overall control strategy

works, is described by Morrellas, Minners, and Donath. (1995) and Garlich-Miller and Donath (1996).

Alternative Path Generation

Alternative path generation is one of the key issues to achieve successful operations of ALX. The central issue for ALX's navigation is how quickly it can generate a safe alternative path when it detects obstacles in its original planned path. A graphic binary-tree search algorithm was developed for ALX's alternative path planning. This method uses the virtual map, which stores the occupancy of ALX's local environment, and an "attracting" location, which lies on the primary path of goal points. ALX tries to reach the attracting location while avoiding obstacles by searching for regions of "empty" cells. The algorithm tests each consecutive goal point by comparing the "footprint" of ALX with the occupancy of the virtual map under the footprint. If occupied, the algori[illegible]ries

a new goal point to the right and then to the left. If these are occupied, ALX backs up by one meter and iterates to find a new path around the occupied region (figure 15). This alternative path-planning method was successfully tested under a number of scenarios, ranging from a single obstacle near the lane boundary (requiring only slight trajectory corrections) to multiple obstacles that completely blocked the marked lane (requiring a reversal of direction to maneuver around the obstacles).

Overall Control Strategy

Figure 16 illustrates the overall control strategy of ALX's autonomous operation. ALX attempts to travel towards the top goal on the primary forward goal stack, which is constructed from the vision information, and then by using the virtual map, constructs a detouring goal stack when an obstacle is detected.

The Virtual Bumper

The second approach that we developed is based on the virtual bumper (Hennessey, Shankwitz, and Donath 1995). For the virtual bumper, we define a personal space around the vehicle (see figure 17). If an obstacle is sensed by the range sensors as having entered the personal space, a virtual force is computed and then applied to the vehicle through the actuators on board ALX. These forces (or torques) in effect "push" ALX away from the obstacle. The magnitude and direction of the virtual force is a function of the encroachment (as measured by the range sensors) into the personal space. This personal space can be characterized as a virtual bumper—a metaphor for its behavior. In effect, the deformation of the virtual bumper, or the rate of change of this deformation (based on the derivative of the range, or the closing rate) will result in an appropriately scaled force vector. The behavior of this virtual bumper can be programmed to mimic the behavior of a combination of linear and/or nonlinear springs and dampers (which can more generally be described as an impedance [Jossi and Donath 1995]). Although not described here, the shape of the virtual bumper can be modified to handle different scenarios (such as traveling through an intersection or traveling at different speeds.) The road, or more accurately the lane boundaries, also applies a virtual force to ALX that "push" it towards the center of the lane. This force can have the profile sketched in figure 17. The forces from the obstacles and the road are then summed together and transformed into speed and steering commands. The resulting control commands maneuver ALX to avoid obstacles while simultaneously attempting to keep ALX in the center of the lane.

Implementation of the virtual bumper requires three major components: generating road forces, generating obstacle forces, and transforming forces into control commands.

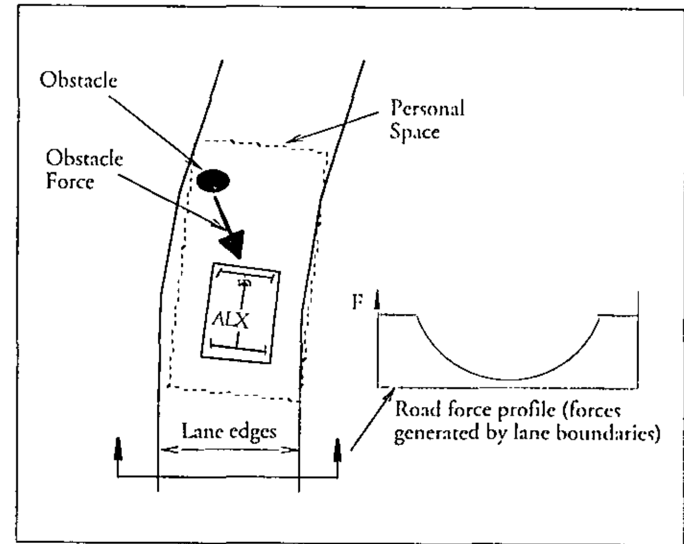


Figure 17. Virtual bumper—overview.

Road Force Generation

The virtual road force is generated as a function of the lateral offset from the center of the lane. The offset is calculated for a point ahead of the vehicle called PointAhead (the determination of this PointAhead is described below.) When there is no offset, there is no force. As the offset increases, so does the virtual force applied to the vehicle. The function used to calculate this road force is defined in equation 1. Figure 18 illustrates how this force is affected by some typical equation parameters.

$$RF = K_{rf} * \left(\frac{1}{\left(Offset_s + \frac{RW}{2} - |y|\right)^2} - \frac{1}{\left(Offset_s + \frac{RW}{2}\right)^2} \right) \quad (1)$$

where,

- *Krf* = Road force stiffness constant
- *y* = Lateral offset at PointAhead from center of road
- *RW* = Road width
- *Offsets* = Sensitivity Offset, needed to prevent force from getting too large as y approaches *RW*/2

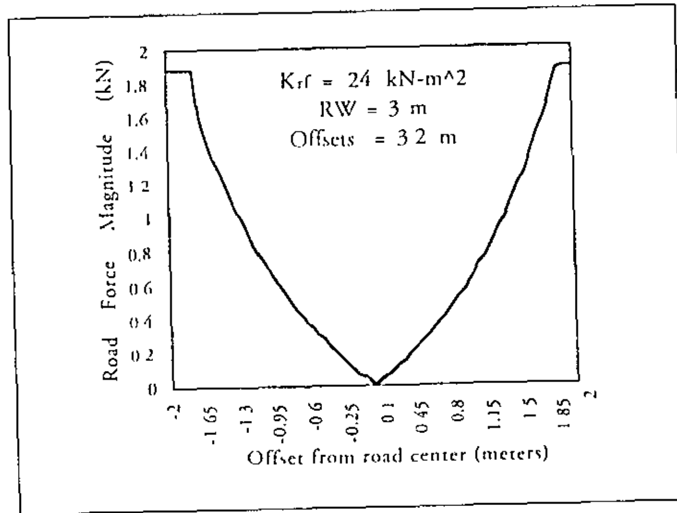


Figure 18 Virtual road force profile

The calculated force is aligned normal to the vehicle's axis of travel and is positive if the offset is to the left of road center and negative if it is to the right of road center. Using a low flat valley at the center of the force profile (as in figure 4-5) results in the vehicle wandering from side to side about the lane center (not unlike human driving). For ALX, we intentionally implemented a discontinuity at the lane center to prevent this type of wandering.

The lateral offset could be measured from ALX's instantaneous position in the lane, but this would not take advantage of our knowledge regarding upcoming turns in the road. Instead, we define a point, called PointAhead, that is projected ahead of the vehicle by a specified distance (this distance is set by the user and is called the LookAheadDistance, equivalent to the preview typically associated with pursuit algorithms.) The lateral offset is then calculated for the position of PointAhead in the lane. This lateral offset is used for calculating the road force. Using this approach allows for improved lane tracking on curved road segments.

Obstacle Force Generation

The obstacle forces are generated by a similar method. First the range sensor readings are used to determine the location of any obstacles with respect to

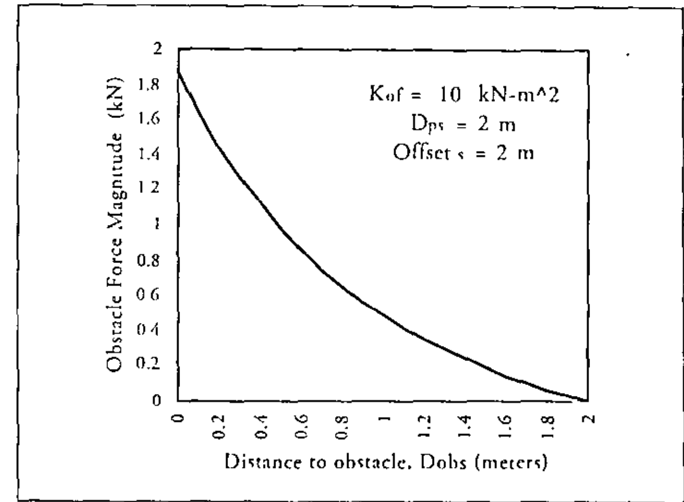


Figure 19 Virtual obstacle force profile

ALX. If an obstacle is within ALX's personal space, the shortest distance from the obstacle to the vehicle is determined and the obstacle force is calculated from equation 2. Figure 19 shows a plot of the calculated forces for some typical equation parameters.

$$OF = K_{of} * \left(\frac{1}{(Offset_s + D_{obs})^2} - \frac{1}{(Offset_s + D_{ps})^2} \right) \tag{2}$$

where:

Kof	= Obstacle force stiffness constant
Dobs	= Distance to obstacle
Dps	= Distance to edge of personal space or virtual bumper
$Offset_s$	= Sensitivity Offset, needed to prevent force from getting too large as D_{obs} approaches zero

If the distance to the obstacle is greater than D_{ps}, the virtual force is set to zero. The obstacle force is also aligned normal to the vehicle's axis of travel and is positive if the obstacle is to the left of vehicle's center and negative if it is to the right of the vehicle's center.

Notice from the figures that the obstacle force and the road force functions have the same general form and magnitude. This is required for both types of forces to have the desired effect on the vehicle's control For insta[illegible] the

magnitude for the obstacle force function is too low, the road force will drive the vehicle into an obstacle. Likewise, if the magnitude for the road force function is too low, an obstacle far from the vehicle can generate a force that will "push" ALX off the road. Therefore, it is important to select functions and function parameters that strike a balance between road following and obstacle avoidance behaviors.

Transforming Forces to Control Commands

Now that the virtual forces are calculated, they must be transformed into vehicle control commands. The virtual forces can have components in both the lateral and longitudinal direction. The lateral components are used for calculating lateral control commands (steering commands) and the longitudinal components are used for calculating velocity control commands. In this scenario, however, we will only calculate the lateral forces. Therefore, the virtual forces are only transformed into lateral control commands. The velocity control commands are determined using a heuristic approach based on the magnitude of the obstacle forces.

In order to better understand how we transform the lateral forces into lateral control commands, we will first discuss the lateral controller. The input to the lateral controller is the desired lateral offset, Y_{lat}. This desired lateral offset is a distance normal to the vehicle's direction of travel. The desired lateral offset and a distance which is set by the user (LookAheadDistance) are then combined to define a point, called SteerPoint, in vehicle coordinates. The angle of SteerPoint with respect to the vehicle's longitudinal axis is used to define a steering command (equation 3).

$$SC = \tan^{-1}\left(\frac{Y_{lat}}{\text{LookAheadDistance}}\right) \tag{3}$$

This steer command turns the front wheels to "aim" at the SteerPoint which is at a distance Y_{lat} to the side and a distance LookAheadDistance ahead of the vehicle.

The next step is to calculate the desired lateral offset from the virtual forces. This lateral offset is calculated using an impedance-based approach. A lateral impedance is first assigned to the vehicle. This impedance is assigned in software and can be representative of a mass, spring, and damper system. The road forces and the obstacle forces are then summed into a resultant. This resultant force is then applied to the system defined by the above specified impedance to determine the desired lateral offset (equation 4).

$$Y_{lat} = \frac{F_{lat}}{Z_{lat}} \tag{4}$$

where

Zlat = Specified lateral impedance

Flat = Resultant lateral force

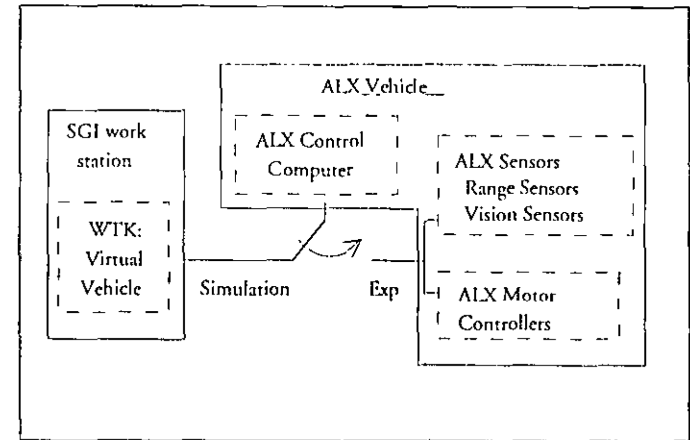


Figure 20. Real-time software is used to control the simulated vehicle as well as ALX

The parameters of the impedance define how the lateral offset will respond to the input forces. These parameters are tuned to provide the desired system response.

The vehicle speed command is set based on the obstacle-generated virtual force. If there is no obstacle force, the speed command is SetSpeed. (SetSpeed is a parameter set by the user and must be set low enough to allow for obstacle detection by ultrasonic range sensors.) If there is a lateral obstacle force and the calculated steer command is within the physical limits of the vehicle, the speed command is set to half of the SetSpeed. The speed is reduced to allow for safer vehicle operation in the presence of obstacles.

One important scenario to note is what happens when the calculated steer command exceeds the steer limits of the vehicle. This occurs when ALX comes too close to an obstacle and would otherwise collide if it were to continue to move in the forward direction. In this case the vehicle is put into a reverse mode. While in this reverse mode, the steer command is set to be at ALX's physical steer limit in the direction opposite to that calculated in equation 3. This causes the vehicle to back up and rotate away from the obstacle. When the steer command calculated from equation 3 falls below the physical steer limits, ALX exits the reverse mode. ALX then starts forward again with a more desirable orientation with respect to the obstacle.

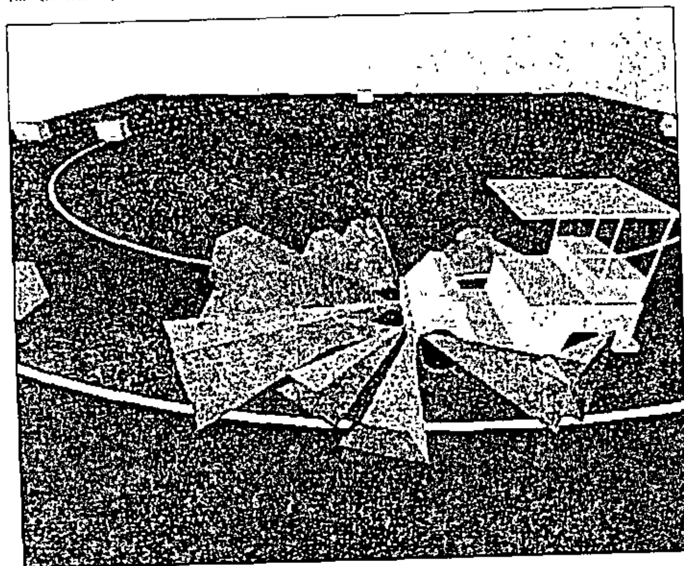


Figure 21. A graphic model of the ALX with multiple range sensors deployed. Each sensor's FOV is shown as a 3D cone.

Simulation Environment

For development of control algorithms for ALX and for our SAFETRUCK testbed, a simulation environment was developed. However, there are many drawbacks in developing control algorithms for a real-time system through simulation. For instance, it is difficult to determine how interprocess communication and the processing rate will affect the controller. Furthermore, once a control law has been determined, it must then be recoded for implementation on the real-time operating system. For these reasons, a nontraditional simulation environment was created.

For the simulation, only the vehicle, sensors, and the environment are simulated, while the vehicle control algorithms are executed in real-time on the ALX computer (figure 20). The vehicle, sensors, and environment are created in a 3D graphics package which runs on a workstation. In the 3D environment, the simulated vehicle can be moved and the simulated sensors (vision and range) calculate the appropriate measurements (see figure 21 for a 3D graphic model of ALX with range sensors). This data is passed to the ALX control computer through a network connection. The ALX computer processes this data as if it

came from ALX's actual vision system and range sensors and determines the appropriate control action. The control action is then used by the ALX computer to calculate position and orientation based on a dynamic model of the vehicle. The new position and orientation is passed through another network connection back to the 3D simulation, and the vehicle is moved and the sensor data recalculated. This type of simulation approach allows for developing control laws while addressing real-time programming issues. Furthermore, the same software developed in simulation can be used on the actual vehicle.

It is important to note that the processing rate of the 3D simulation must be close to the sensor measurement output rate on the actual vehicle. This is because various control algorithms may be adversely affected by different sensor data rates. Therefore, the sensor data may have to be filtered to be used effectively. Keeping the simulation processing rate near the sensor output rate allows for developing appropriate filters for the sensor data through simulation.

Experimental Results

Both the path tracking and the virtual bumper algorithms were evaluated through a series of tests. We observed the ability of each algorithm to perform road following (no obstacles) as well as single- and multiple-obstacle environments. Emphasis was also placed on the ability of the algorithm to react to the vehicle's environment in real-time.

First the control approaches were evaluated based on road-following ability. When no obstacles are present, both algorithms guide the vehicle through similar paths and have similar computational requirements. Therefore, these experiments proved to be more a test of the vision system. The vision system guided ALX over a test track (figure 22) consisting of both straight sections and corners of varying curvatures under bright sunlight, overcast conditions, and low light intensity situations (i.e., dusk and dawn). The vision system proved to provide adequate information for both control algorithms.

Next the ability of the control algorithms to perform obstacle avoidance was tested. Through a series of tests it was determined that the path followed in a static obstacle environment is similar for both approaches. Figure 23 shows an example of ALX following a roadway which contains an obstacle near the middle of the road. The differences between these two algorithms will be described below in terms of the strengths and weaknesses that we observed through these experiments.

The path tracking algorithm has inherent strengths and weaknesses. One of its main weaknesses lies in the fact that it uses a recursive search to find an alternate path. The time to complete the search is significant and is a function of the virtual map cell size. Another weakness is that the virtual map approach does not work in a dynamic environment. Moving obstacles will corrupt the map.

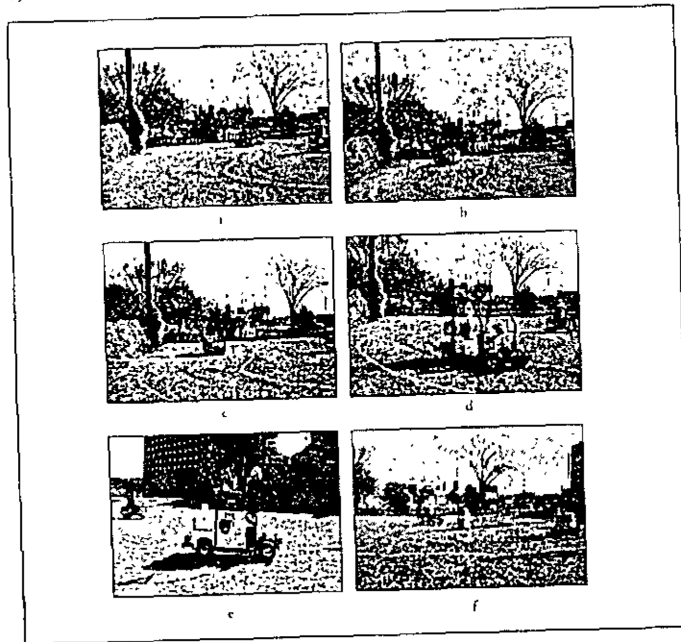


Figure 22. ALX following a test "road."

On the other hand, this approach does have the ability to find a traversible path in a complicated environment. For instance, the alternate path search algorithm can find a path around a dead-end obstacle configuration. The weaknesses noted here led us to investigate the virtual bumper approach.

The virtual bumper algorithm also exhibits problems that need addressing. The primary weakness is the effect of a local minimum in the potential force field. This occurs in a cluttered environment when the forces of the obstacles balance with the forces of the road (i.e., dead end road.) This is an inherent pitfall for potential field approaches. Little work has been done here to alleviate this problem because we do not believe it will have a significant impact on our ultimate goal of vehicle control in a highway environment. For instance, if a control vehicle is on a highway with obstacle vehicles to the front and on both sides, it is in a local minimum. The effect of this local minimum is that the vehicle stays in this relative position until the traffic scenario changes and a path is available. This response is desirable, and is why we believe the mini-

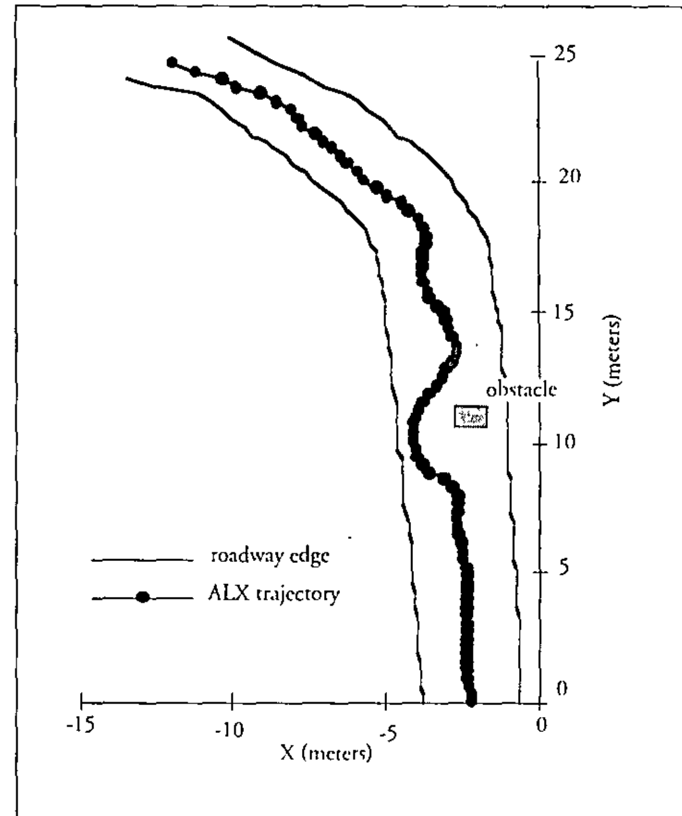


Figure 23. ALX following a roadway which contained an obstacle.

mum is not a significant issue in a structured highway driving environment.

There are also major benefits of the virtual bumper approach. First, this control approach responds quickly to obstacles in its environment without significant computational load and delays—a result of the approach in which the path is determined based on force calculations instead of a binary search. These force calculations require little computation and execution time is approximately constant regardless of obstacle configuration. Second, because of the reflexive nature of this approach, it can respond to stationary or moving obstacle environments

These properties of the virtual bumper make it scalable to control a vehicle on a highway environment where it is absolutely necessary to handle dynamic environments in real-time.

Conclusion

ALX has proven to be a valuable tool for the preliminary investigation of vehicle sensing and control strategies at the system-integration level. To date, it has facilitated the evaluation of multitasking and intertask communication, real-time control protocols, vision-based lateral control, dead-reckoning based vehicle position estimation, watchdog safety subsystems (Krantz, Morris, Donath, and Johnson 1996) and ultrasonic sensor-based collision avoidance strategies. A simulation environment has been developed that has proven to be an effective tool for software and control algorithm development. Given the nature of ALX, vehicle dynamics are not an issue. The path tracking collision avoidance strategies used here might be appropriate for maneuvering in parking lots and in freight loading areas, but they probably have limitations that need to be addressed. They certainly would not apply to highway environments. However, the "virtual bumper" collision avoidance strategy discussed here provides a foundation for developing collision avoidance strategies for any vehicles, and in particular for large trucks, which may have to steer around obstacles rather than brake because of their high inertias. Higher risk strategies can be investigated on ALX without significant concern for human safety and vehicle damage. Other future technologies to be evaluated on ALX include millimeter wave radar (to replace the ultrasonic range sensors) for obstacle sensing and collision avoidance.

Acknowledgments

This project was partially supported by the Minnesota Department of Transportation; the Center for Advanced Manufacturing, Design, and Control (CAMDAC); the Center for Transportation Studies, and the ITS Institute at the University of Minnesota. We would like to acknowledge the assistance of our colleagues Micah Garlich-Miller and Jon Minners of the Robotics Laboratory.

PART THREE

Mobile Robot Architectures

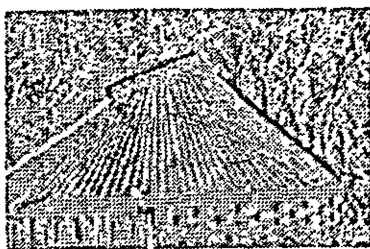
In the context of this book, robot architectures refers to the arrangement of control software for the robot. Further we are concerned here with software architectures designed for use on physical robots from the outset, as opposed to well-known AI architectures which have been adapted to execute on robots as an experiment (e.g., GUARDIAN [Hayes-Roth 1995], and SOAR [Weismeyer 1989]). There has always been some skepticism in the community that there is any generality to be derived from using software architectures with robots, since the endeavor is usually more of an engineering art than a scientific discipline. Nonetheless, as discussed in the introduction to this book, in the past dozen years or so, some unifying principles have emerged among those research groups who are committed to seeing AI paradigms running usefully on mobile robots.

One such principle is that intelligent robots require both continuous and discrete event control capability executing in the same software framework. The control theory community has long known how to implement the former and has a large body of control laws for a variety of mechanisms (see for example, Dorf [1989]). Yet, there are few examples of control theory-based robots that can achieve high level goals such as find and fetch tasks (a notable exception is Saridis [1995]). This is due in large measure to the fact that there is a sharp rise in complexity as we move from from servo loops to interacting conjunctive goals, and for this, control theory has few solutions.

Researchers in AI robotics however, have discovered that continuous control and discrete event processing—that processing required to recognize goal-relevant state occurrences—need not co-exist in a homogeneous whole, but as two heterogeneous layers with some kind of syntactic differential between them. This is the essence of the "three layer" architecture for intelligent control of mobile robots, and Erann Gat's chapter gives a detailed account of the development of what could be considered a consensus architecture. The chapter by Firby and his colleagues describes a specific instance of using the first two layers of this architecture to integrate robot motion and vision control in one framework.

Several groups of researchers have been successful in integrating similar architectures which emphasize different aspects of discrete and continu[illegible] [illegible]ontrol due to past research roots or current endeavors. The chapter by H[illegible] ; Hu

CONVOLUTION


David Young, January 1993, revised January 1994

This teach file is an introduction to *convolution*, an important operation in low-level vision.

Contents

Please read the introduction giving general information about the nature of this document.

- Preliminaries
- Differencing as convolution
- Specifying convolution masks in programs
- Some more convolution masks
  - A small centre-surround mask
  - A smoothing mask
  - The Sobel masks for edge detection
- Why convolution masks are flipped over
- A convolution procedure in Pop-11
- Convolution as template matching
- Summary

Preliminaries

You should have read through TEACH VISION1 and run the examples.

Load the libraries needed to run the examples now:

```
uses popvision          ;;; search vision libraries
uses rci_show           ;;; image display utility
uses showarray          ;;; array printing utility
uses arrayfile          ;;; array storage utility
uses arraysample        ;;; sampling utility
uses float_byte         ;;; type conversion utility
uses float_arrayprocs   ;;; array arithmetic library-
uses convolve_2d        ;;; convolution program
```

We will also use the same image as before, so we might as well read it now. It will speed up the examples if we convert the image array to a packed floating point representation; this makes little difference to how it is used in Pop-11, so you need not worry about this (but see HELP *FLOAT_BYTE if you want more details).

```
vars image;             ;;; declare a permanent variable
```

```
arrayfile(popvision_data dir_>< 'stereo1.pic') -> image;
;;; convert from string array to packed float array
float_byte(image, false, false, 0, 255) -> image;
```

Differencing as convolution

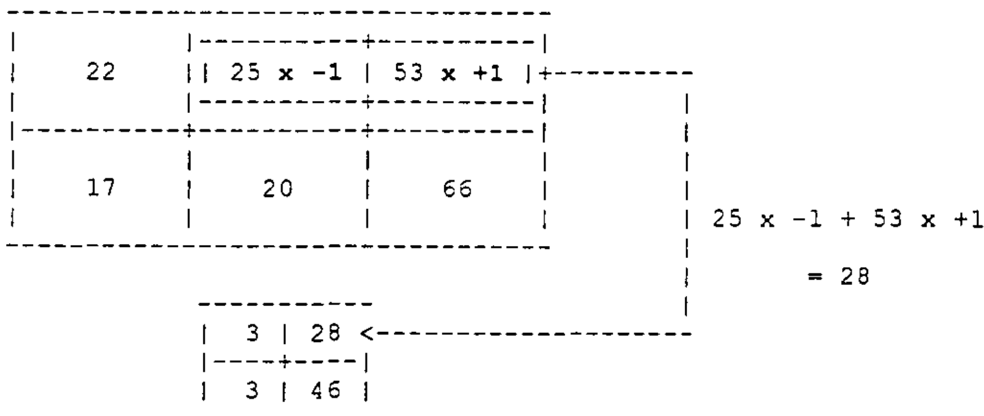
The example at the end of TEACH VISION1 processed an image to highlight vertical boundaries, by taking differences between the values of horizontally adjacent pixels. Each pixel had the value of the pixel to its left subtracted from it. Another way of describing this process begins by making a table or *mask* containing the values -1 and +1, like this:

```
-----------
| -1 | +1 |
-----------
```

We can now do our calculation by overlaying this mask on the array, multiplying each grey-level by the superimposed mask value, and adding the products. This must be done for each mask position in turn For example, suppose we have an array with just 3 columns and 2 rows, containing these grey-level values:

```
----------------
| 22 | 25 | 53 |
|----+----+----|
| 17 | 20 | 66 |
----------------
```

Then the calculation for the top right hand element of the output array looks like:



```
-------------------------------------
|          |----------+----------|
|    22    || 25 x -1 | 53 x +1 |+---------
|          |----------+----------|         |
|----------+----------+----------|         |
|          |          |          |         |
|    17    |    20    |    66    |         |
|          |          |          |         | 25 x -1 + 53 x +1
-------------------------------------      |
                                           |        = 28
          -----------                      |
          |  3 | 28 <-----------------------
          |----+----|
          |  3 | 46 |
          -----------
```

We repeat this calculation for all the other possible positions of the mask (including positions that overlap with each other) to fill in the other squares.

Although the multiplications may seen superfluous if all we want to do is to take differences, the advantage of this way of looking at the process is that it is easy to generalise. For example, we can specify a vertical differencing operation with the mask

```
------
| -1 |
|----|
| +1 |
------
```

and masks for nearest-neighbour diagonal differences look like

```
 -----------      -----------
 | -1 |  0 |      |  0 | -1 |
 |----+----|      |----+----|
 |  0 | +1 |      | +1 |  0 |
 -----------      -----------
```

The table of numbers that specifies a convolution is known as a *mask*, a *kernel*, an *operator* or a *template*, depending on the author, and the numbers in the mask are often called *weights*.

Specifying convolution masks in programs

The natural way to represent a convolution mask in a program is to use an array, just as for an image. However, there is a small added complication: if we lay out the mask as above, then the rows and columns are numbered in the opposite directions to the rows and columns of the image array. Accepting that there is a very good reason for this, which we shall come to later, we can label the rows and columns of the horizontal differencing mask thus:

```
      1    0
   -----------
 0 | -1 | +1 |
   -----------
```

with the column numbers increasing from right to left.

The location of (0, 0) (i.e. column 0, row 0) in the mask is important: during convolution, each result will be placed in the output array at the location corresponding to the pixel lying under the (0, 0) mask element in the input array. Thus in the example above, if the value 53 lies at (3, 1) in the input array, the value 28 will lie at (3, 1) in the output array, because the value 28 is generated when the value 53 lies under the (0, 0) mask element. The labelling of rows and columns for the input and output arrays will therefore be as follows:

```
      1    2    3                         2    3
   ----------------                    -----------
 1 | 22 | 25 | 53 |                  1 |  3 | 28 |
   |----+----+----|        ->          |----+----|
 2 | 17 | 20 | 66 |                  2 |  3 | 46 |
   ----------------                    -----------
```

Now we are in a position to set up a Pop-11 array to act as a convolution mask for horizontal differencing. Referring to the diagram of the mask above, the columns run from 0 to 1 and the rows from 0 to 0, so the boundslist of the array will be [0 1 0 0], and we can set up the mask as follows:

```
vars mask;
newarray([0 1 0 0]) -> mask;     ;;; create a new array
-1 -> mask(1, 0);                ;;; col 1, row 0
 1 -> mask(0, 0);                ;;; col 0, row 0
```

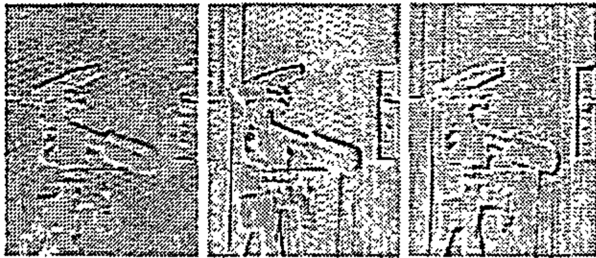
To try this out, we will use a library procedure for two-dimensional convolutions, and apply it to our test image.

```
vars newimage;
convolve_2d(image, mask, false, false) -> newimage;
```

As usual, you can refer to HELP *CONVOLVE_2D if you want more information about the procedure we have just used, but it is not necessary to do so now. The result, of course, should look identical to the last example of TEACH VISION1: to check, first print the output array to see the boundslist (note it starts at column 81 as it ought) and then display it.

```
    newimage =>
prints:
    ** <array [81 176 64 191]>

    rci_show(newimage) -> ;
```

You should now try out the vertical difference mask and the diagonal difference masks in the same way. Do not forget that row numbers in convolution masks increase upwards; if you have difficulty with the boundslists or setting the values in the arrays, try drawing out small diagrams like the ones above. (Continue just to give <false> as the last two arguments to **convolve_2d**.)



You should be able to see clearly how edges of different orientations stand out in the four different results. Remember that you can increase the size of the display windows by changing **rci_show_scale** if you wish.

Some more convolution masks

A small centre-surround mask

It is possible to design a convolution mask that is sensitive to all edges, regardless of orientation. One of the simplest such masks is the small *centre-surround* mask that looks like this:

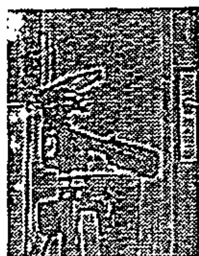
```
 ----------------------
 | -1/8 | -1/8 | -1/8 |
 |------+------+------|
 | -1/8 |  +1  | -1/8 |
 |------+------+------|
 | -1/8 | -1/8 | -1/8 |
 ----------------------
```

This mask is approximately *isotropic* - you can rotate it about its centre and it looks roughly the same -

so that it will respond more or less equally to edges in any direction. If the operator is on a uniform patch f the image, its output is zero because the 8 weights of -1/8 will cancel out with the single +1. If it is close to an edge, so that there is an imbalance, then it will give a positive or negative output, depending on which side of the edge it is

We can set up this mask and try it with the following code. Note that the central +1 is put at the (0, 0) position in the mask, which is the natural choice. This means that the value stored at (X, Y) in the output array is the value at (X, Y) in the input array, minus the average of its neighbours. We use the initialisation argument to **newarray** to set the whole mask to -1/8, then update the central value to +1.

```
newarray([-1 1 -1 1], -0.125) -> mask;
1 -> mask(0, 0);
convolve_2d(image, mask, false, false) -> newimage;
rci_show(newimage) -> ;
```



We will see how to make more effective use of the centre-surround mask later. Although centre-surround operations are important in theories of biological vision, practical computer vision systems tend to use masks which are more closely related to the vertical and horizontal difference operations.

This centre-surround mask implements an approximation to the mathematical operator known as the *Laplacian*; this in turn is closely related to taking the *second derivative* of a function. If you are familiar with these mathematical operations, you may be able to see the link. If you have not met the Laplacian before, but encounter it in a book or paper, it may help to understand it if you think in terms of the convolution mask shown above.

A smoothing mask

We often want to reduce the amount of small-scale detail in an image, either to get rid of irrelevant texture in trying to pick out the main shapes, or to reduce the effects of noise (random variations in the grey-level values) introduced by the camera optics or digitisation process. One way of doing this is to replace each grey-level by a local average of itself and its neighbours. Such averages tend to be done with 3 x 3 or 5 x 5 masks so that they can be symmetrical about their centres. For example, a 3 x 3 mask that implements a straight average, giving all 9 values the same weight, looks like

```
-------------------
| 1/9 | 1/9 | 1/9 |
|-----+-----+-----|
| 1/9 | 1/9 | 1/9 |
|-----+-----+-----|
| 1/9 | 1/9 | 1/9 |
-------------------
```

and is easily tried with

```
newarray([-1 1 -1 1], 1.0/9.0) -> mask;
```

```
convolve_2d(image, mask, false, false) -> newimage;
rci_show(newimage) -> ;
```

You will see that the result looks like a blurred version of the original image (which you should display if you have not already done so). This kind of mask is sometimes called a *blurring* mask as well as a smoothing mask.

To blur the image more heavily, you could use a larger mask, or alternatively, you could repeat the convolution one or more times:

```
convolve_2d(newimage, mask, false, false) -> newimage;
rci_show(newimage) -> ;
```

Note the picture getting increasingly fuzzy each time you execute the two lines of code above.

The Sobel masks for edge detection

It is possible to combine more than one operation in a single mask. A good example is given by the Sobel masks for edge detection, which combine the vertical and horizontal differencing operations with some smoothing to reduce the effects of noise or very local texture.

The masks look like:

```
----------------          ----------------
| -1 |  0 |  1 |          | -1 | -2 | -1 |
|----+----+----|          |----+----+----|
| -2 |  0 |  2 |          |  0 |  0 |  0 |
|----+----+----|          |----+----+----|
| -1 |  0 |  1 |          |  1 |  2 |  1 |
----------------          ----------------
```

The single values used in the simple differencers above are replaced by averages over 3 pixels, weighted towards the centre in each case. In addition, the positive and negative parts are separated by a one-pixel gap; by increasing the baseline for differencing this too has a smoothing effect, and it allows the mask to be more symmetrical, so that the results are "centred" in a way that is not true for the smaller difference operators

You should now easily be able to set up the Sobel masks and look at their effects Compare the outputs from the Sobel masks with the results of smoothing the image and then taking vertical or horizontal differences - the results should be similar, though not identical

Why convolution masks are flipped over

Why should the rows and columns of masks be numbered in the opposite directions to those of images? The reason is that with this convention, the operation of convolution is *associative* - that is, if two masks are convolved one with the other, then the result can be used as a mask that has the same effect as convolving each of the two masks with the image, one after the other. (If you decide to check this statement experimentally, note that you will have to extend one of the initial mask arrays with a border of zeroes to avoid losing any values in the compound mask.) Also, convolution is *commutative* - that is, you can exchange the mask and the image as far as the mathematics goes (extending the mask with a border of zeroes), though in practice this would result in absurdly inefficient programs.

You could skip the rest of this section on a first reading.

There is another way of looking at convolution which makes sense of the mask indices. It can be useful to have this other way of thinking about convolution.

This approach involves shifting copies of the array, multiplying the copies and the original by the weights, and adding them together. Taking horizontal differencing as an example again, we proceed as follows:

1. Make a copy of the image array, but with every value shifted, or offset, one pixel to the right - call this **shift_image**:

```
          image                            shift_image

      1    2    3                      1    2    3    4
    ----------------                 --------------------
  1 | 22 | 25 | 53 |               1 |  - | 22 | 25 | 53 |
    |----+----+----|                 |----+----+----+----|
  2 | 17 | 20 | 66 |               2 |  - | 17 | 20 | 66 |
    ----------------                 --------------------
```

2. Trim the edges of the arrays to retain only the values that are defined in both arrays (i.e. in this case remove columns 1 and 4).

3. Multiply every grey-level in the unshifted array by 1 and every grey-level in the shifted array by -1:

```
        image x 1                shift_image x -1

          2    3                     2     3
        -----------                 -------------
      1 | 25 | 53 |              1 | -22 | -25 |
        |----+----|                |-----+-----|
      2 | 20 | 66 |              2 | -17 | -20 |
        -----------                 -------------
```

4. Add these two arrays, pixel by pixel, giving as the result:

```
         2         3                  2     3
     -----------------              -----------
   1 | 25-22 | 53-25 |           1 |  3 | 28 |
```

```
     |-------+-------|   =    |----+----|
   2 | 20-17 | 66-20 |      2 |  3 | 46 |
     -----------------        -----------
```

Of course, this seems very cumbersome compared to the original procedure, and in fact the method just described would not be used in a practical program, unless one happened to have parallel hardware that was suited to it It should be clear, however, that the results are identical.

In step 3 we multiplied the two arrays by 1 and -1. We can put these two values in a table, where position in the table corresponds to the offset of the array that was multiplied, thus:

```
       0     1
     -----------
   0 | +1 | -1 |
     -----------
```

The value at column 0, row 0 in this table specifies the multiplier for the unshifted array. The value at column 1, row 0 specifies the multiplier for the array shifted one column to the right.

This is, of course, just the same convolution mask as we had before, although it has now been drawn the other way round. However, the correct assignment of row and column indices to the elements of the mask has fallen out naturally in this development.

A convolution procedure in Pop-11

So far we have just used the library procedure **convolve_2d** to carry out the convolutions. How complex is such a procedure? In fact, a Pop-11 version is reasonably simple, and it would be a good exercise to try to write your own before looking at the one given below.

On first reading this teach file, you need not look at the following procedure in detail, and could skip to the next section. However, if you are writing your own vision programs, you should bear this example in mind. In particular, it is a very common mistake for people to write procedures like this with unnecessary assumptions built in - e g. that the array bounds begin at 0 or 1, or that the (0, 0) point of the mask is in its centre. Such assumptions are bad practice (except in the rare cases where they allow a large efficiency gain) - procedures should be as general as possible. This convolution code illustrates how to write a general procedure that handles Pop-11 arrays properly, regardless of their boundslists.

```
define convolve_new(image, mask) -> result;
    ;;; This procedure carries out a convolution. Both the arguments
    ;;; must be 2-D arrays containing numbers. The result is a new
    ;;; 2-D array. The result bounds are chosen so that the whole of
    ;;; the output array can be filled without moving any mask
    ;;; element outside the bounds of image.

    ;;; Declare arguments as lvars for efficiency
    lvars image, mask, result;
    ;;; Get the array bounds
    lvars
        (Icol0, Icol1, Irow0, Irow1) = explode(boundslist(image)),
        (Mcol0, Mcol1, Mrow0, Mrow1) = explode(boundslist(mask));
    ;;; Set up the bounds for the results.
    lvars
        Rcol0 = Icol0 + Mcol1,
        Rcol1 = Icol1 + Mcol0,
        Rrow0 = Irow0 + Mrow1,
```

```
        Rrow1 = Irow1 - Mrow0;
    ;;; Create the output array
    newarray([% Rcol0, Rcol1, Rrow0, Rrow1 %]) -> result;
    ;;; Declare variables needed for the loops
    lvars Rcol, Rrow, sum, Mcol, Mrow;

    ;;; Loop over the output array elements. Can use fast_for for
    ;;; a slight efficiency gain (does same as for)
    fast_for Rrow from Rrow0 to Rrow1 do
        fast_for Rcol from Rcol0 to Rcol1 do
            0 -> sum;           ;;; reset
            ;;; Loop over the mask elements, building up the sum
            fast_for Mrow from Mrow0 to Mrow1 do
                fast_for Mcol from Mcol0 to Mcol1 do
                    ;;; Basic convolution formula
                    sum + mask(Mcol, Mrow) * image(Rcol-Mcol, Rrow-Mrow)
                        -> sum
                endfor
            endfor;
            ;;; Assign the complete sum to the result array
            sum -> result(Rcol, Rrow)
        endfor
    endfor
enddefine;
```

You should be able to see how this procedure implements the calculation described in the first section above. The main things to notice are:

1. the structure of the two sets of nested **for** loops;
2. the way the subtraction of the indices (**Rcol-Mcol**, **Rrow-Mrow**) in the innermost statement follows from the reversal of the mask indices discussed above;
3. the way the bounds for the output array (**Rcol0** etc.) are calculated.

The last of these is the hardest part to get right in writing such a program - but the final formulae are surprisingly simple.

The techniques used in the procedure above are not restricted to Pop-11 - programs for this sort of operation are basically similar in most languages, except for special-purpose languages with built-in array operations or for use with parallel hardware. Pascal and FORTRAN procedures would look quite like the Pop-11 one, whilst C procedures are superficially rather different because C has a rather low-level view of arrays, though the underlying structure would be the same. (The curious can look at LIB *CONVOLVE_2D_F.C to see the C code called by **convolve_2d**.)

You can load the procedure defined above and check that it gives the same results as **convolve_2d** from the library. Assuming that **mask** has been set up as a mask array, you can compare the two displays resulting from

```
rci_show(convolve_2d(image, mask, false, false)) -> ;
rci_show(convolve_new(image, mask)) -> ;
```

which should look identical. However, there is one big difference - the library routine will run a lot faster than **convolve_new**. (Test this with *TIMEDIFF.) This is because **convolve_2d** uses externally loaded code written in C. The combination of code written in C for well-defined number crunching procedures such as convolution with code written in Pop-11 for higher-level processes and iteractive use (as in this teach file) is a powerful one.

The library procedure's last two arguments, which we have not used, provide added flexibility and efficiency for programs which use it. If you want to see what these arguments do, look at HELP *CONVOLVE_2D - you could consider incorporating the same features in any programs you write yourself.

Convolution as template matching

As a final example of a convolution mask, we will look at the possibility of using a mask tailored to a specific feature for which we want to search. It seems intuitively likely that the convolution output will be highest at places where the image structure matches the mask structure, where large image values get multiplied by large mask values. We can try out this idea by picking out part of our image to use as a mask.

The *ARRAYSAMPLE library is convenient for this purpose. The following code copies a small part of the array into a new array - the part to be copied is specified by a boundslist-type argument:

```
    arraysample(image, [98 108 128 138], false, [5 -5 5 -5], "nearest")
        -> mask;
```

The 11 x 11 region we have copied, between columns 98 and 108 and rows 128 and 138, contains the oval image of the end of main pivot of the tripod head, as can be seen by looking at the mask with **rci_show** (increasing **rci_show_scale** temporarily if necessary). The other boundslist-type argument, **[5 -5 5 -5]**, specifies that this region is to be copied onto columns 5 to -5 and rows 5 to -5 of a new array. The unconventional ordering in this list causes the rows and columns to be reversed when the values are copied (top to bottom and left to right) as required for a convolution template.

As it stands, this mask will not pick out any features - as all its values are positive, each output will be a weighted average of an 11 x 11 region of the input array, so the effect will just be smoothing. To make a useful feature detection mask, we need to have values that go negative as well as positive, so that the result will be zero for any uniform region of the input image. The following line of code does this by subtracting out the average of the values in the mask (Refer to HELP *FLOAT_ARRAYPROCS if you want to use the routines yourself.)

```
    float_addconst(-float_arraymean(mask), mask, mask) -> mask;
```

You can see that this has made the values go negative as well as positive with the simple numerical print routine from *SHOWARRAY:

```
    3 -> sa_sigfigs;          ;;; reduce the number of digits printed
    sa_simple_nums(mask);     ;;; print the whole array
```

prints:

```
   -------------------------
   5555544444333332222211111000001111122222333334444455555

 -5>  -40  -40  -40  -40  -27   32   74   68   -2  -40  -25
 -4>  -40  -40  -40  -40   -5   89  103   79   10  -40  -36
 -3>  -40  -40  -40  -38   60  107   94   64   -2  -40  -40
 -2>  -40  -40  -40   -6   66   66   38   28  -12  -40  -40
 -1>  -40  -40  -40    6   44   29   23   22   -4  -40  -40
  0>  -40  -40  -40   -5   50   10    4    4    3  -40  -40
  1>  -40  -40  -40   -2   71   49   69   82   46  -40  -39
  2>  -40  -40  -40   25   99   91   96   92   18  -40  -36
```

```
3>  -40  -40  -40   -1   97  114  104   74  -20  -40  -29
4>  -40  -40  -40  -39   74  131  105   27  -40  -40  -31
5>  -40  -40  -37  -35  -26   -6  -37  -40  -40  -40  -24
```

Now you can display the result of convolving this mask, or template, with the original array, using

```
rci_show(convolve_2d(image, mask, false, false)) -> ;
```

Overall, the result looks pretty messy, because our mask happens to have a centre-surround structure and so is sensitive to much of the grey-level variation in the image. However, there is a definite bright patch at the position of the end of the pivot, and in fact the centre of this patch has the highest value in the output array (at row 103, column 133, as you can verify by writing a small program). In other words, the convolution operation has found the part of the image with the structure most closely corresponding to the template - not surprisingly, this is the part from which the template was originally copied.

Summary

The results of convolution processing are useful - the process is extremely common in both practical and experimental computer vision systems. However, the way in which the output of a convolution is actually handled in such systems has not been demonstrated here - rather convolution has been presented as an image processing operation, and the results presented graphically.

You should now:

- know what the convolution algorithm is, in terms of scanning an image with a mask of weight values;
- know how to set up convolution masks as Pop-11 arrays, and use the library procedure **convolve_2d** to carry out convolutions;
- have some feel for the effects of different convolution masks, in particular differencing, centre-surround, smoothing and edge-detection masks;
- understand how convolution can be used as template-matching;
- know where to find further information about the routines that have been used in the examples;

and possibly:

- understand the description of convolution in terms of weighted sums of offset copies of an array;
- know how to write a general convolution procedure in Pop-11.

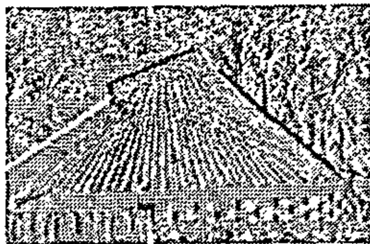
Here are links to:

- The index of teach files

- The next file in the series
- The School of Cognitive and Computing Sciences home page

Copyright University of Sussex 1994. All rights reserved.

GAUSSIAN MASKS, SCALE SPACE AND EDGE DETECTION


David Young, January 1993, revised January 1994

This teach file continues with the theme of convolution, introducing biological models and practical edge detection methods which rely on convolution with *Gaussian* masks. The effects of image analysis at different scales are also illustrated.

Contents

Please read the introduction giving general information about the nature of this document.

- Preliminaries
- The Gaussian convolution mask
- Gaussian smoothing
- The difference of Gaussians mask
- Zero-crossings
- The grey-level gradient
- The Canny edge detector
- Summary

Preliminaries

You should have read TEACH VISION1 and VISION2.

Load the necessary libraries and get hold of an image now:

```
uses popvision          ;;; search vision libraries
uses rci_show           ;;; image display utility
uses arrayfile          ;;; array storage utility
uses float_byte         ;;; type conversion utility
uses float_arrayprocs   ;;; array arithmetic library
uses convolve_gauss_2d  ;;; Gaussian convolution program
uses rc_graphplot       ;;; graph display utility
uses canny              ;;; Canny edge detection

vars image;
arrayfile(popvision_data dir_>< 'stereol.pic') -> image;
float_byte(image, false, false, 0, 255) -> image;
```

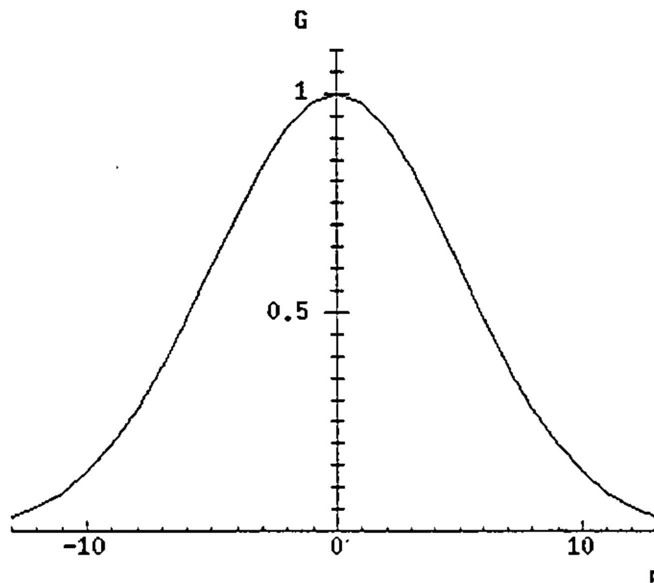
The Gaussian convolution mask

The Gaussian convolution mask is a circularly symmetrical (or isotropic) mask, such that any cross-section through its centre yields a weight profile that has the form of a Gaussian or normal curve. The mathematical formula is in all statistics text books, and almost all computer vision books. A Pop-11 procedure which implements the formula to generate Gaussian weights looks like this.

```
define gauss_weight(r, sigma) -> g;
    lvars r, sigma, g;
    exp( -(r**2) / (2 * sigma**2) ) -> g
enddefine;
```

The **r** argument gives the distance (in pixels) from the centre of the mask, and **sigma** specifies the "width" of the mask. The procedure returns the weight for this distance and width parameter. We can see what the curve looks like by plotting the function with *RC_GRAPHPLOT:

```
rc_new_window(400, 300, 500, 20, false);  ;;; make a new window
rc_graphplot(-13, 1, 13, 'r', gauss_weight(% 5 %), 'G') -> ;
```



If the graph window appears in an inconvenient place on the screen, move and resize it with the mouse, then if necessary re-execute the call to **rc_graphplot**.

To draw the graph, **sigma** was set to 5 by creating a closure (see *CLOSURES) of **gauss_weight**. You can see that the curve drops to about 60% of its central value when **r** is equal to **sigma** - this is what is meant by by saying that **sigma** sets its width. Try changing **sigma** to see the effect.

The mathematical Gaussian function does not fall to zero for any finite value of **r** - the "tails" of the curve go on forever. For a practical mask, it is necessary to *truncate* the function when the weights have become small enough to be insignificant, to keep the mask to a reasonable size. As you can see from the graph, the values are pretty small when **r** is plus or minus 13 for **sigma** = 5. An efficient library procedure to generate arrays containing Gaussian weights is **gaussmask**. This has a sensible criterion for truncation built in (see HELP *GAUSSMASK), and it also *normalises* the weights - that is, it multiplies them by a constant so that the weights in the mask sum to 1, which avoids increasing or decreasing the average grey-level when the mask is used for smoothing. To get a 1-D Gaussian mask, execute:

```
    vars gmask, gsize, sigma;
    5 -> sigma;
    gaussmask(sigma) -> gmask;          ;;; calculate the weights
    gaussmask_limit(sigma) -> gsize;    ;;; get the mask size
```

The bounds of the mask array are given by **[-gsize gsize]**. If you plot the values in **gmask** with **rc_graphplot** (it will accept an array as an argument instead of a function) you will find the curve looks the same as the one we already have, though the actual values are smaller because of normalisation. A fast and simple way to print the values in the **gmask** array is:

```
    gmask =>                        ;;; print the boundslist
prints:
    ** <array [-13 13]>

    arrayvector(gmask) =>           ;;; see HELP *ARRAYS
prints:
    ** <sfloatvec 0.002735 0.00451 0.007144 0.010873 0.015899 0.022337 0.030152
        0.039105 0.048727 0.058337 0.067104 0.074161 0.078747 0.080338 0.078747
        0.074161 0.067104 0.058337 0.048727 0.039105 0.030152 0.022337 0.015899
        0.010873 0.007144 0.00451 0.002735>
```

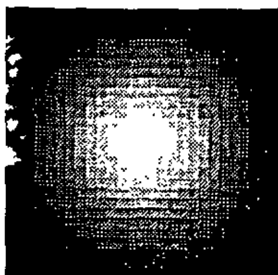
but this method is only really useful for small 1-D arrays.

We can get a 2-D Gaussian mask by multiplying elements of the 1-D mask together. (This is because the 2-D Gaussian mask has a convenient property called *separability*; you would have to go back to the formula and do some algebra to demonstrate this, which we will not do here.) We can therefore generate a 2-D Gaussian mask with a little sleight-of-hand, using a procedure as initialiser for **newarray**:

```
    vars gmask_2d;
    newarray([% -gsize, gsize, -gsize, gsize %],
        procedure(x, y); lvars x, y;
            gmask(x) * gmask(y)
        endprocedure) -> gmask_2d;
```

We can now look at the 2-D gaussian mask with

```
    5 -> rci_show_scale;            ;;; expand the display size
    rci_show(gmask_2d) -> ;
    1 -> rci_show_scale;            ;;; reset the display size
```



Of course, a digital mask like this is only an approximation to the mathematical Gaussian because it has been sampled on a discrete grid and truncated. However, it is good enough for practical use.

Gaussian smoothing

The Gaussian convolution mask is important both in theories of biological vision, such as Marr's, and in

many computer vision systems. As a smoothing mask, it has optimal properties in a particular sense: it removes small-scale texture and noise as effectively as possible for a given spatial extent in the image.

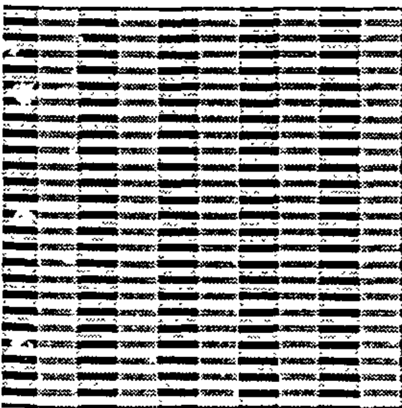
To analyse this property fully requires a mathematical development that is beyond the scope of this course, and requires careful definition of terms like "spatial extent" used loosely above. One key idea, that is useful to know about even without the mathematics, is that of *spatial frequency*. High spatial frequencies correspond to small-scale structure, low frequencies to large scale structure. It is possible to decompose an image into its constituent spatial frequences, just as it is possible to decompose a sound wave into its constituent temporal frequencies (which we hear as different pitches). Spatial frequency analysis is used in some practical methods in image processing, and is an important psychophysical tool in investigations of biological vision.

Since small-scale texture contains a lot of grey-level variation at high spatial frequencies, the aim of a smoothing operation is to remove high spatial frequencies without distorting lower spatial frequencies. It turns out that because the Gaussian mask is itself smooth, it is particularly good at separating high and low spatial frequencies without using information from a larger area of the image than necessary.

We will use the procedure **convolve_gauss_2d** to try out Gaussian smoothing and demonstrate its efficacy. (This procedure is much more efficient than a general procedure like **convolve_2d**, because it makes use of the separability property to greatly reduce the amount of computation.)

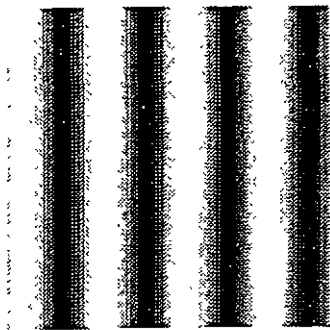
It will be useful to have a test image to look at: we can generate one with a stripy pattern as follows

```
vars testimage;
newsfloatarray([1 100 1 100],        ;;; float array for efficiency
    procedure(x, y) -> value; lvars x, y, value = 0;
        if x mod 20 > 9 then
            20 -> value
        endif;
        if y mod 4 > 1 then
            20 + value -> value
        endif
    endprocedure) -> testimage;

2 -> rci_show_scale;
rci_show(testimage) -> ;
```



This gives an array with a high spatial frequency (i e fine) grid intersecting a low spatial frequency (i.e coarse) grid. Now we examine the effect of Gaussian smoothing with **sigma** = 4.

```
rci_show(convolve_gauss_2d(testimage, 4)) -> ;
```



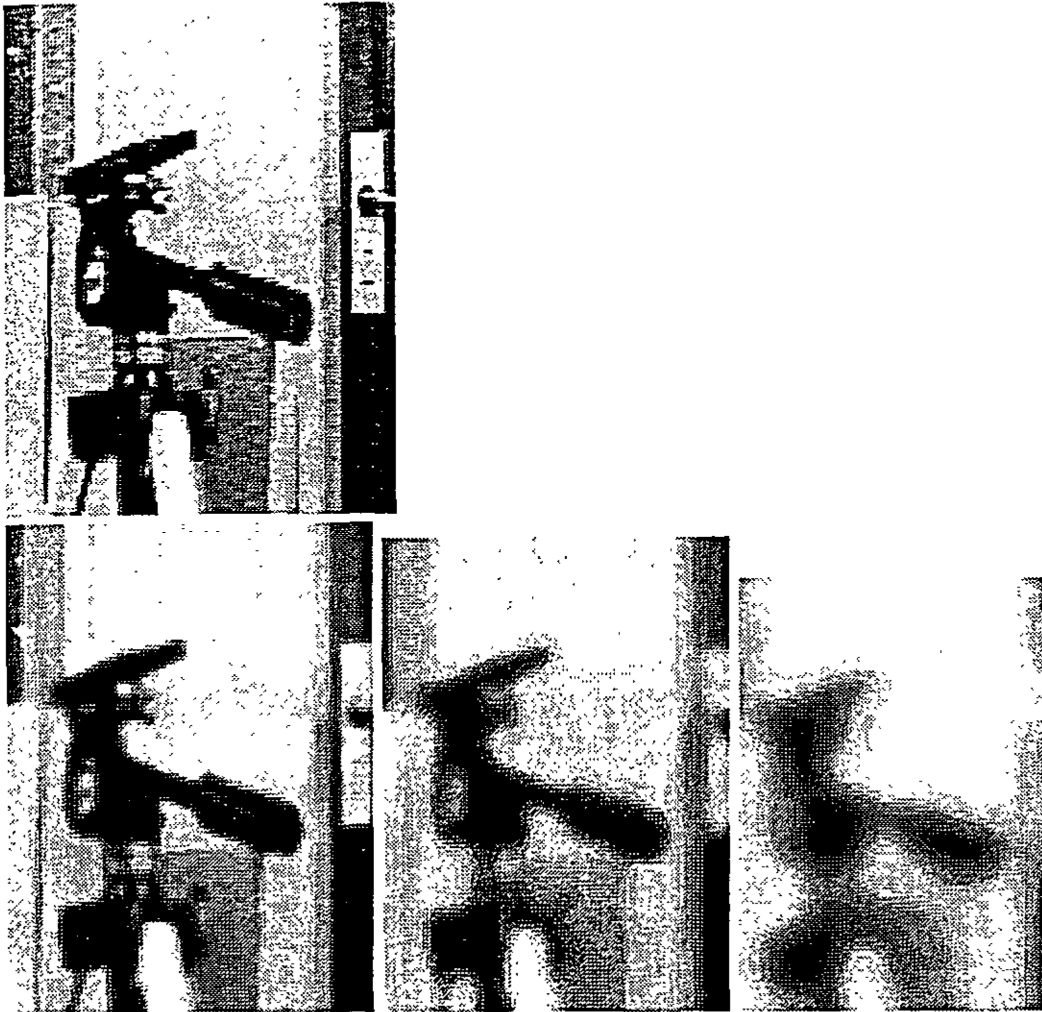
The fine stripes are effectively filtered out, leaving the broad ones. If we try the same thing with a uniform mask, it is impossible to do it as effectively - e.g. a 7 x 7 mask gives these results

```
vars umask, usize;
3 -> usize;          ;;; -3 to +3 gives a 7 x 7 mask
newarray([% -usize, usize, -usize, usize %], 1.0/(usize**2)) -> umask;
rci_show(convolve_2d(testimage, umask, false, false)) -> ;
```

Changing the size of the mask will not improve the results significantly. The reason is that the uniform mask has an abrupt cut-off at its boundaries, so as it passes over the thin stripes and their edges cross the mask boundary, sharp changes in the output values are inevitable. It is this sharp cut-off that the Gaussian mask avoids.

Now look at the effects of Gaussian smoothing on the real image.

```
rci_show(image) -> ;                          ;;; no smooothing
rci_show(convolve_gauss_2d(image, 1)) -> ; ;;; sigma = 1
rci_show(convolve_gauss_2d(image, 2)) -> ; ;;; sigma = 2
rci_show(convolve_gauss_2d(image, 4)) -> ; ;;; sigma = 4
```



and note the increasing reduction in detail and fine texture, and the removal of all but the main shape in the final image.

It is reasonable to think of **sigma** as setting the *scale* at which we preserve information in the convolved image. Structure on a scale small compared with **sigma** will be removed, whilst structure on a larger scale is retained.

Many systems make use of structure at several scales, retaining multiple representations like the ones displayed by the code above. When the image has been smoothed, it is not necessary to retain as many pixels in the representation - one quarter of the pixels might be enough after smoothing with **sigma** = 2, for example. Smaller arrays are therefore used to hold the smoothed images, resulting in a data structure called a *resolution pyramid*.

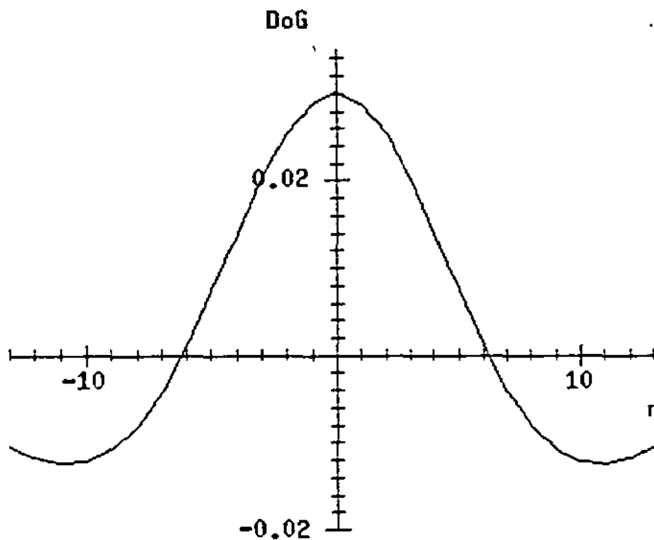
The difference of Gaussians mask

It has already been pointed out that a centre-surround mask can be used to locate grey-level boundaries in an image. It seems reasonable to combine the centre-surround operation with Gaussian smoothing to find the boundaries at different scales in the image. We could simply carry out the two operations one after the other, or generate a combined mask by convolving the Gaussian mask with the centre-surround mask, but it is usually more convenient to use the the *difference of Gaussians* mask, or DoG The DoG is a good approximation to the combined centre-surround+Gaussian operator (which is also called the

Laplacian of the Gaussian).

e can look at the form of the 1-D difference of Gaussians by generating two 1-D masks and subtracting one from the other. It turns out that the best approximation to the Laplacian occurs if the larger mask has a **sigma** about 1.6 times that of the smaller. This code will display the resulting curve (somewhat truncated):

```
vars maskinner, maskouter, dogmask;
gaussmask(5) -> maskinner;
gaussmask(5 * 1.6) -> maskouter;
gaussmask_limit(5) -> gsize;
newarray([% -gsize, gsize %],
    procedure(r);
        maskinner(r) - maskouter(r)
    endprocedure) -> dogmask;

rc_graphplot(-gsize, 1, gsize, 'r', dogmask, 'DoG') -> ;
```

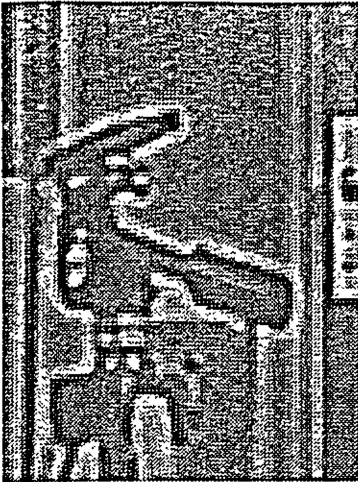


(If the window containing the earlier graph has been iconified or is behind other windows, you will have to make it visible again.)

You can see that it has a smooth centre-surround structure, as expected. (We cannot use our earlier procedure **gauss_weight** to generate this curve, as it is important that the values in the positive and negative masks are normalised.)

The DoG is not separable. However, an efficient implementation is still possible by simply taking the difference between two Gaussian convolutions with different **sigma** values. We can look at the result of a DoG convolution with

```
rci_show(convolve_dog_2d(image, 1)) -> ;
```



where the second argument (1) is the **sigma** parameter for the inner part of the mask.

Zero-crossings

The raw output of the DoG operation does not yield any obvious simplification of the image. However, the significant features of the output of a centre-surround operator are the places at which positive and negative values are adjacent - its *zero-crossings*. For a full discussion of this, see Marr and other references in the bibliography in TEACH *VISION.

Note that the zero-crossings of any array that contains both positive and negative values can be found - the concept is not linked to any particular convolution operation. It is therefore wrong to talk about the zero-crossings of an image (as people quite often do); you need to refer to the zero-crossings of the output of the DoG operator, for example.

The most effective way to display the zero-crossings of a convolution operation is to *threshold* the output array. Thresholding is the operation of producing a *binary image* by assigning one value to all the pixels that exceed some limit (the threshold), and another value to all the others. (It is not usually very effective as a way of segmenting raw grey-level images, except in circumstances where the illumination and background can be carefully controlled.) We can apply it to visualising zero-crossings by setting the threshold to zero.

It is handy to have a small procedure to do the convolution, thresholding and display. The threshold procedure comes from *FLOAT_ARRAYPROCS:

```
define showzeros(image, sigma);
    lvars image, sigma, newimage;
    convolve_dog_2d(image, sigma) -> newimage;
    ;;; Threshold so all values above 0 get set to 1, all others
    ;;; to -1. Re-use the array to reduce garbage collections.
    float_threshold(-1, 0, 1, newimage, newimage) -> newimage;
    rci_show(newimage) ->
enddefine;
```

Now we can try it for a variety of scales. In the displays, the zero-crossings are, of course, the boundaries separating the black from the white regions - white shows the positive values and black the negative ones

```
showzeros(image, 1);
```

```
showzeros(image, 2);
showzeros(image, 4);
```

You can see how the positions of edges are accurately represented in the small-scale output, which also includes a lot of detailed texture, whilst the large-scale output retains the main features, but has only approximate positions for the edges.

Marr proposed that zero-crossings at different scales are combined by biological visual systems to obtain evidence for significant image boundaries. Whether this is so is open to doubt, but it is clear that processes analogous to DoG convolutions do operate in the early stages of biological vision. The results of such operations must therefore be an efficient way to code image structure, since retinal mechanisms have evolved to transmit this particular kind of information along the optic nerve. This makes sense: raw grey-levels, for example, contain a great deal of redundant information about the overall brightnesses of surfaces, whereas the DoG outputs sacrifice this to allow more information to be carried about texture and boundaries.

In practical computer vision, the DoG filter tends not to be used for edge detection. However, it has proved valuable in other ways - for example, a fast stereo-matching algorithm by Nishihara (to be discussed later) uses thresholded DoG convolutions to generate image regions to match. Furthermore, the idea of using zero-crossings for different values of **sigma** has been generalised into the important idea of *scale space* analysis.

We will leave the DoG at this point and turn to a popular edge-detector used in computer vision, but first it is necessary to say what is meant by the local grey-level gradient of an image.

The grey-level gradient

The outputs of the X and Y differencing masks

```
                                    ------
  -----------                       | -1 |
  | -1 | +1 |          and          |----|
  -----------                       | +1 |
                                    ------
```

are often termed the X and Y *grey-level gradients*. If you imagine a surface above the image, whose height is proportional to the grey-level at each point, then it should be clear that what these operators

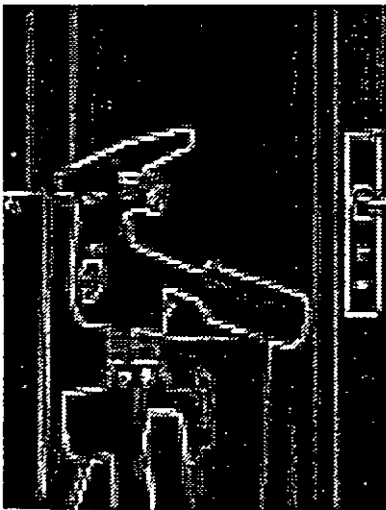
give are measures of the slope, or gradient, of the surface if you move along the X and Y directions respectively. These measures can be combined to give the *total gradient* or *gradient magnitude*, which is how much the surface slopes in the direction which goes most steeply downhill. If the Y gradient is zero, for example, then the total gradient is just equal to the X gradient, but if the surface slopes in both the X and Y directions then we need to combine the two values.

The correct formula is, in fact, just that of Pythagoras:

```
                      2                        2                     2
   (total gradient)    =   (X gradient)    + (Y gradient)
```

We can display the total gradient for our image:

```
    vars maskx, masky, imagex, imagey;
    newarray([0 1 0 1], 0) -> maskx;     ;;; for X differences
    -1 -> maskx(1, 0);
     1 -> maskx(0, 0);
    newarray([0 1 0 1], 0) -> masky;     ;;; for Y differences
    -1 -> masky(0, 1);
     1 -> masky(0, 0);
    convolve_2d(image, maskx, false, false) -> imagex;
    convolve_2d(image, masky, false, false) -> imagey;

    rci_show(float_arrayhypot(imagex, imagey, false)) -> ;
```



(We use 2 x 2 masks containing some zeros to make the X and Y gradient arrays come out the same size. For a practical program one would do something more efficient. The routine **float_arrayhypot** (from *FLOAT_ARRAYPROCS) takes its name from the fact that the Pythagorean formula calculates the hypotenuse of a triangle. It uses the C library routine **hypot** - see MAN *HYPOT.)

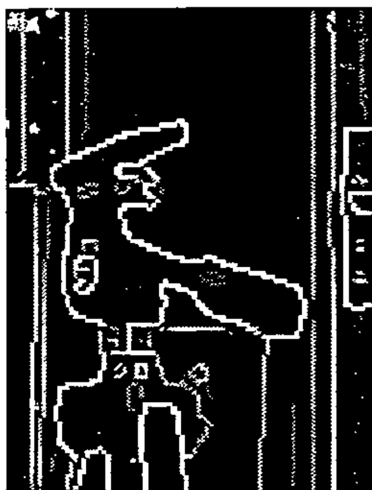
You can see how regions stand out where the grey-level changes most rapidly. It is also possible to calculate the *gradient direction* at each point in the image - i.e. the direction in the image for which the gradient changes most rapidly. For this you would use the C procedure **atan2** or the Pop-11 procedure **arctan2** applied to the X and Y gradients.

The Canny edge detector

Although the DoG is not the basis for most edge detectors in computer vision, the idea of combining Gaussian smoothing with simple local operators certainly is One of the more successful kinds of edge detector is based on ideas published by Canny (see *Readings in Computer Vision*, listed in the bibliography in TEACH *VISION, p. 184). This combines Gaussian smoothing with the simple horizontal and vertical difference operators The results are then combined as above to give the total smoothed gradient at each pixel. Positions where the gradient is a *local maximum* along gradient direction are then found, and then a thresholding operation is used to find those lines where the gradient is above some limit. (Actually the thresholding is more complicated, but that is the basic idea.) For more details of the local implementation, see HELP *CANNY.

Canny's method can be applied to the usual image.

```
vars edges;
vars sigma = 1;            ;;; smoothing
vars t1 = 5, t2 = 10;      ;;; hysteresis thresholds
canny(image, sigma, t1, t2) -> (imagex, imagey, edges);
rci_show(edges) -> ;
```



Here **sigma** is, as usual, the scale parameter for Gaussian smoothing, whilst **t1** and **t2** are thresholds used for deciding which parts of the edges to keep. In the display, the brightest edges are those with the strongest associated grey-level gradient. The other results returned by **canny**, **imagex** and **imagey**, are the smoothed X and Y gradients, which you can display for comparison with the unsmoothed versions we have looked at previously.

You could look at the Canny edges produced with different values of **sigma**, as we did for the DoG zero-crossings.

We can throw away the gradient strength information and make a binary array which records only the positions of *edge pixels* with:

```
vars bin_edges;
float_threshold(0, t1/2, 1, edges, false) -> bin_edges;
rci_show(bin_edges) -> ;
```

This *edge map* often forms the basis for subsequent processing. For example, simple shapes like straight lines and ellipses can be found, particularly in the images of industrial objects to which computer vision is likely to be applied. These can be used to build up a geometrical description of the image at a higher level of abstraction. Alternatively, in model-based vision, the expected appearance of a model is often matched against the edge map.

Summary

You should now.

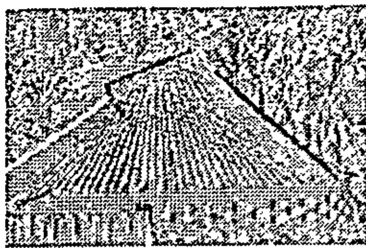
- know what the Gaussian function and Gaussian convolution masks look like, and have some idea of how to implement them;
- have a qualitative understanding of why Gaussian masks are good for smoothing;
- know what the DoG operator is and does;
- know what a zero-crossing is;
- have seen how the zero-crossings of the DoG change depending on the image scale selected;
- know what is meant by grey-level gradient;
- know in outline how Canny-type edge detectors work.

Here are links to:

- The index of teach files
- The next file in the series
- The School of Cognitive and Computing Sciences home page

Copyright University of Sussex 1994. All rights reserved.

HOUGH TRANSFORMS



David Young, January 1993, revised January 1994

This teach file deals with one method of finding specified shapes in images: the *Hough transform*

Contents

Please read the introduction giving general information about the nature of this document.

- Preliminaries
- Introduction
- Representing straight lines
- The set of lines through an image feature
- Parameter space
- Accumulator arrays
- Mapping image data into parameter space
- Mapping accumulator peaks to image lines
- Finer points of the straight-line Hough transform
  - Negative r values
  - Smoothing and peak detection
  - Edge weighting
  - Use of gradient direction
  - Hierarchical Hough transforms
- Hough transforms for other shapes
- Summary

Preliminaries

You should have read the 3 previous teach files, VISION1, VISION2, VISION3. As usual, it will be helpful to get the necessary libraries loaded now, and to load a test image. It is particularly important, when working through this teach file, to run the examples in order, as they are more interdependent than usual.

```
uses popvision          ;;; search vision libraries
uses rci_show           ;;; image display
uses rc_context         ;;; allows multiple windows for graphics
uses rc_graphplot       ;;; graph drawing
uses arrayfile          ;;; array storage
uses float_byte         ;;; type conversion
uses float_arrayprocs   ;;; array arithmetic
uses canny              ;;; Canny edge detection
uses straight_hough     ;;; Hough transform
```

```
vars image;
arrayfile(popvision_data dir_>< 'stereol.pic') -> image;
float_byte(image, false, false, 0, 255) -> image;
false -> popradians;        ;;; this teach file uses°
```

Introduction

It is useful to be able to find simple shapes - straight lines, circles, ellipses and the like - in images. Man-made objects, for instance, frequently have shapes with straight and circular edges, which project to straight and elliptical boundaries in an image. One kind of algorithm for identifying these extended image features involves following edges, and linking together edgelets which seem to lie on straight lines or smooth curves. An alternative approach, which is the subject of this teach file, involves accumulating the evidence provided by each edge element for the shape being sought. Since, usually, the shape can be at any position in the image, and often at any orientation and of any size as well, the whole set of different possibilities has to be taken into account.

The Hough transform is used in a variety of related methods for shape detection. These methods are fairly important in applied computer vision; in fact, Hough published his transform in a patent application, and various later patents are also associated with the technique.

To see how the methods work, we will take one of the simplest possible shapes as an example: the straight line. This is often a building block for higher-level descriptions of image structure, and a basis for matching in model-based vision.

A straight line in an image has two ends, but the Hough transform illustrated here does not find end-points. Rather it finds the infinite straight lines on which the image edges lie. A part of a line lying between two end-points is called a *line segment*, whilst in this file the term *line* will mean an infinite straight line. These infinite lines are worth extracting, since once found they can be followed through the image and end-points located.

Representing straight lines

Positions in images have been represented so far by coordinates corresponding to the column and row indices of array elements. Now it will be helpful to use a different coordinate system, with its origin in the centre of the image. We will call this the **(x, y)** coordinate system, and it will have **y** running up the screen and **x** running from left to right in the conventional way. The following code will illustrate these coordinates for you, with **x** and **y** each running from -100 to +100 pixels from the origin.

```
rc_new_window(300, 300, 550, 50, true);     ;;; get a window
```

(If the size or position of the window is inconvenient, adjust it with the mouse now.)

```
[-100 100 -100 100] -> rcg_usr_reg;      ;;; set image size
rc_graphplot([], 'x', [], 'y') -> ;      ;;; draw axes
```

(In this file the *RC_GRAPHPLOT package is used to draw the axes and set the scales and origins of coordinate systems for the illustrations. This gives an easy way of making the axes correspond to the coordinate system for *RC_GRAPHIC commands. You do not need to bother with these details of the examples, unless you want to experiment further with these facilities - you should be able to see what is going on by looking at the graphics in the display windows.)

It is easy to translate between the (x, y) coordinates and array (**column**, **row**) coordinates

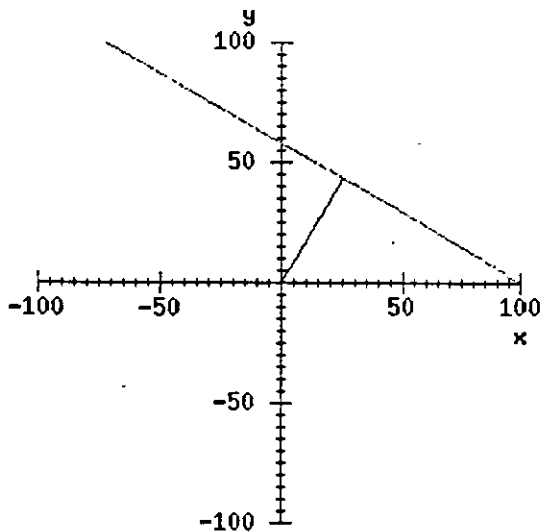
There are various ways to specify a line. One method, particularly suited to the Hough transform, is as follows. Imagine yourself standing on the image plane at the origin of the coordinates, facing the positive **x** direction (to the right on the screen) Turn a specified angle to your left, and then walk a specified number of pixels forward. Turn through 90° and go forward; you are now walking along the required line in the image.

The following code illustrates this on the screen Suppose you turn through 60° and walk 50 pixels forward. Then your path can be drawn with:

```
XpwSetColor(rc_window, 'blue') -> ;      ;;; draw in blue
rc_jumpto(0, 0);                    ;;; start at the origin
60 -> rc_heading;                   ;;; face 60° left of x-axis
rc_draw(50);                        ;;; go 50 pixels
```

and then the line we are interested in with:

```
XpwSetColor(rc_window, 'red') -> ;       ;;; draw in red
rc_turn(90);                        ;;; turn 90° left
rc_draw(150);                       ;;; draw some of the line
rc_turn(180);                       ;;; turn right round
rc_draw(300);                       ;;; draw the other way
XpwSetColor(rc_window, 'black') -> ;       ;;; go back to black
```



It should be clear that any line can be drawn by setting two quantities in this algorithm: the angle intially turned through, called **theta**, and the distance moved from the origin, called **r**. The red line is therefore the line with *parameters* **r** = 50 pixels, **theta** = 60°. It will be useful to incorporate the moves above into a procedure for drawing lines specified this way:

```
define draw_rt_line(r, theta);
    ;;; Draw a line specified by r, theta
    lvars r, theta;
    lconstant lngth  = 300;              ;;; assume 300 long enough
    rc_jumpto(0, 0);             ;;; start at the origin
```

```
        theta -> rc_heading;            ;;; turn by theta
        rc_jump(r);                     ;;; move, do not draw
        rc_turn(90);                    ;;; turn through 90
        rc_jump(lngth/2);               ;;; move along the line
        rc_turn(180);                   ;;; face back along it
        rc_draw(lngth);                 ;;; draw the line
    enddefine;
```

So the red line could be drawn with **draw_rt_line(50, 60)**. If you are still unclear about the **(r, theta)** parameterisation of the line, use **draw_rt_line** to draw some more lines on the window.

You may have met other ways of describing lines, e.g. the **(m, c)** parameterisation, where **m** is the slope and **c** is the y-axis intercept, and the line has the equation **y = mx + c**. This is not so useful for our purposes, because for lines nearly parallel to the y-axis, **m** gets extremely large.

The set of lines through an image feature

Suppose we have detected an image feature at particular **(x, y)** coordinates, perhaps using an edge detector. We will assume that we know the position of the feature, but that any orientation associated with it (e.g. the direction of the grey-level gradient) is not accurate enough to be useful. If we are looking for lines in the image, then a feature at a particular point is evidence that there might be a line passing through that point. Many different lines pass through a given point, and the feature is evidence for any or all of them, so we need a way of finding the set of lines through a point.

If we have a feature at **(x, y)**, and we know the **theta** parameter of a line through the feature, then we can calculate the line's **r** parameter with this equation:

```
    r = x cos(theta) + y sin(theta)
```

You might be able to check this statement with a little geometry and algebra, but it is not necessary to do so here; we can verify experimentally that it seems to hold. Suppose there is a feature at **x** = 30, **y** = 70. Clear the window and mark the feature with:

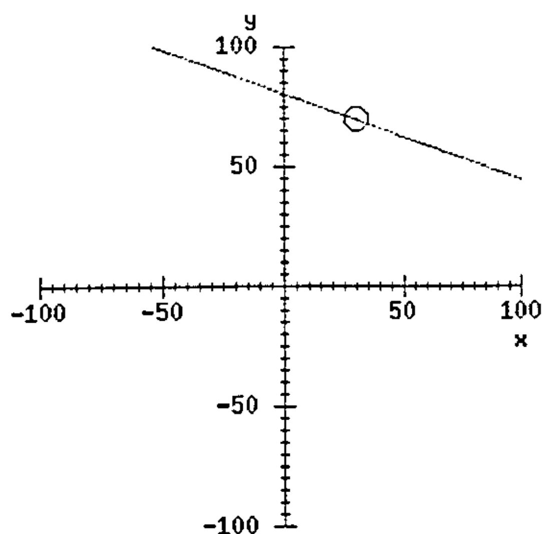
```
    rc_graphplot([], 'x', [], 'y') -> ;      ;;; just draw axes
    rc_jumpto(30, 70);                       ;;; go to x=30, y=70
    rc_point_here();                         ;;; draw a dot
```

The dot is probably almost invisible, so let's put a circle round it:

```
    0 -> rc_heading;                         ;;; face right
    rc_jump(5);                              ;;; move 5 pixels
    90 -> rc_heading;                        ;;; face top
    rc_arc_around(5, 360);                   ;;; draw circle
```

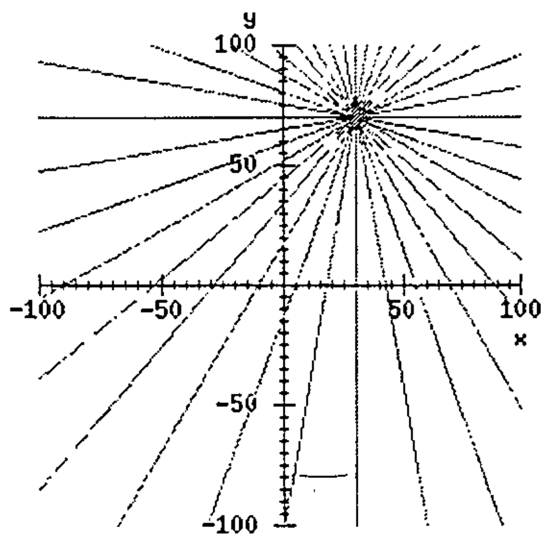
Now, arbitrarily choosing **theta** = 70°, we can draw a line through the feature by calculating **r** using the equation above, and then calling **draw_rt_line**. Thus:

```
    vars r, theta;
    70 -> theta;
    30 * cos(theta) + 70 * sin(theta) -> r; ;;; x=30, y=70
    XpwSetColor(rc_window, 'red') -> ;
    draw_rt_line(r, theta);
```



And it is easy enough now to draw as many lines as we like through the feature, particularly if the formula above is made into a procedure:

```
define line_r(x, y, theta) -> r;
    lvars x, y, theta, r;
    x * cos(theta) + y * sin(theta) -> r
enddefine;

for theta from 0 by 10 to 350 do    ;;; lines at 10° intervals
    line_r(30, 70, theta) -> r;
    draw_rt_line(r, theta);
endfor;
```



Parameter space

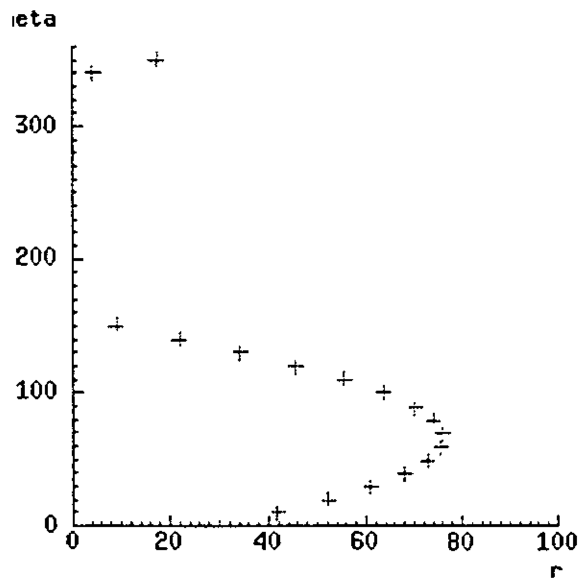
Clearly, we do not normally want to draw a lot of lines. What we do want to do is to record the set of

lines in a data structure. To this end, it is useful to introduce a *parameter space* for lines. (This is sometimes called a *Hough space*) A two-dimensional parameter space is just a plane for which the coordinates are the parameters. We can visualise the parameter space for **r** and **theta** in another window, set up like this:

```
vars xy_context;                        ;;; save the present window
rc_context(false) -> xy_context;        ;;; and its coord system
false -> rc_window;
rc_new_window(300, 300, 550, 400, true);
[0 100 0 360] -> rcg_usr_reg;            ;;; bounds of parameter space
rc_graphplot([], 'r', [], 'theta') -> ; ;;; draw axes
```

A point in this window is to correspond to a line in the x-y window we have used so far. Let us plot the points in parameter space that correspond to the lines we have drawn. This involves using almost the same code we had above. Instead of calling **draw_rt_line**, we just need to mark a point in parameter space - a simple procedure to draw a cross (from LIB *RCG_UTILS) is used to mark the points in the new window.

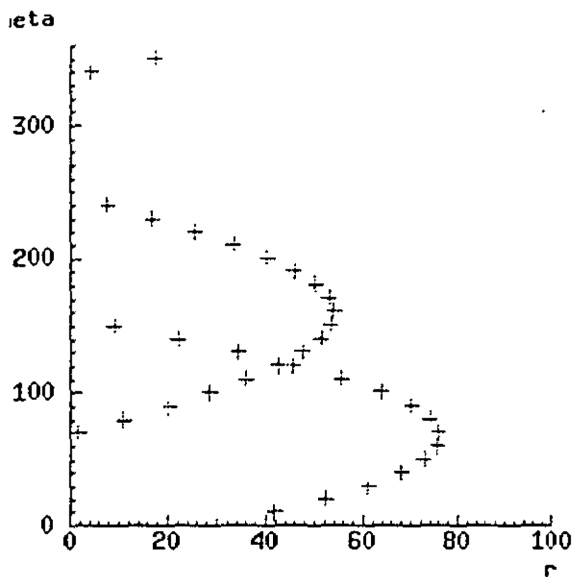
```
XpwSetColor(rc_window, 'red') -> ;  ;;; draw in red
for theta from 0 by 10 to 350 do    ;;; lines at 10° intervals
    line_r(30, 70, theta) -> r;
    rcg_plt_plus(r, theta);         ;;; draw a cross
endfor;
```



There appears to be one problem - some of the values of **r** were negative, so have vanished off the left of the graph. In fact, it is safe to ignore the negative values - an explanation of this follows later.

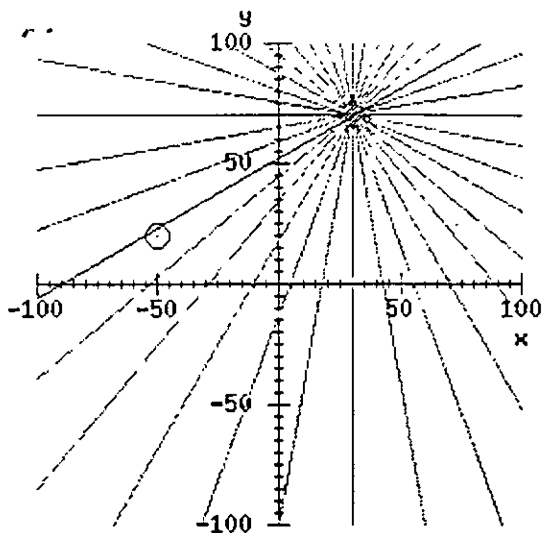
Suppose there is a second feature in the image, at, say, **x** = -50, **y** = 20. We can mark the parameters for the lines that pass through it as well, with:

```
for theta from 0 by 10 to 350 do    ;;; lines at 10° intervals
    line_r(-50, 20, theta) -> r;
    rcg_plt_plus(r, theta);
endfor;
```



As you can see from the display, the two lines of crosses intersect at about **r** = 45, **theta** = 120°. These, then, must be the parameter values for a line that passes through *both* the two features. To check, the next piece of code switches back to the **x-y** window, marks the new feature, and then draws the line with the parameters **r** = 45, **theta** = 120°.

```
vars rt_context;                   ;;; to hold the r-theta window
rc_context(false) -> rt_context; ;;; and save it
xy_context -> rc_context();       ;;; go back to x-y window

;;; Mark the point at x=-50, y=20 with a circle - see above
XpwSetColor(rc_window, 'black') -> ;
rc_jumpto(-50, 20);
rc_point_here();
0 -> rc_heading;
rc_jump(5);
90 -> rc_heading;
rc_arc_around(5, 360);

XpwSetColor(rc_window, 'blue') -> ;      ;;; draw in blue
draw_rt_line(45, 120);
```



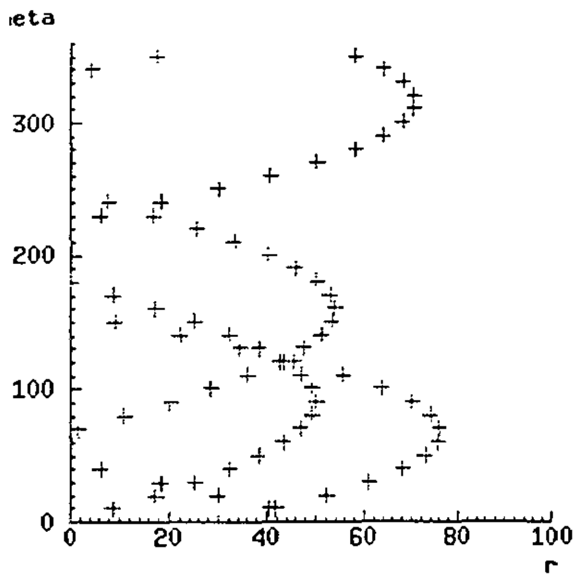
So we have found (approximately) the parameters of the line through the two features, via the parameter space representation. This is a most cumbersome way to find a line through two points, but it comes into its own if we have a large number of points to deal with - as with a processed image.

Suppose that we have a third feature, on almost the same line as the other two - say at **x** = 0, **y** = 50. Then its Hough space representation will intersect at the same point as the previous two:

```
rt_context -> rc_context();
for theta from 0 by 10 to 350 do
    line_r(0, 50, theta) -> r;
    rcg_plt_plus(r, theta);
endfor;
```

whereas a point somewhere off the line (e.g. at **x** = 50, **y** = -50) will not contribute to the cluster near **r** = 45, **theta** = 120:

```
XpwSetColor(rc_window, 'blue') -> ;
for theta from 0 by 10 to 350 do
    line_r(50, -50, theta) -> r;
    rcg_plt_plus(r, theta);
endfor;
```



If more and more points that are on our line are detected, then more and more evidence will accumulate in the **r** = 45, **theta** = 120 region of the parameter space. Points that are not on the line will contribute to other parts of the space, but will not form a definite cluster unless they happen to lie on another straight line. If several straight lines are present, then several clusters will form in parameter space.

Accumulator arrays

To detect lines in practice, we need to record where the points fall in parameter space. To do this, we use an array, called an *accumulator array*, such that elements of the array correspond to small regions of parameter space. To start with, every element of the accumulator array will be set to zero. For each point in parameter space to be recorded, the appropriate element of the array is incremented. An element of the accumulator array is sometimes called a *bin*.

Dividing parameter space up into regions for this purpose is known as *quantisation*. How many rows and columns to use in the accumulator array, that is, how to quantise parameter space, is an important issue - it is beyond this teach file, though it is fair to say that trial and error often play a large part.

We will use an accumulator array with column number corresponding to **r** and row number corresponding to **theta**. Suppose (somewhat arbitrarily) we use 100 bins along **r** and 100 along **theta**, numbering them from 0 to 99, giving 10,000 bins altogether. To get from **r** and **theta** to **column** and **row** in the accumulator array, we can use

```
round(r) -> accumulator_column;
round(theta/3.6) -> accumulator_row;
```

The **round** procedure returns the nearest integer to its argument. We can see that the *quantisation interval* is 1 pixel in **r**, and 3.6° in **theta**. You can imagine drawing a grid with lines separated by 3.6 units vertically and 1 unit horizontally on the parameter space window. All the crosses that fall into one cell of the grid will contribute to one element of the accumulator array.

The increments to a bin of the accumulator array are sometimes referred to as *votes*. Each edge element in the image votes for those bins in parameter space which represent lines that pass close to that element.

It follows that the bins which get the most votes correspond to lines which have many edge elements lying close to them.

To see if this works, we will use data from a real image.

Mapping image data into parameter space

We will use edge data from the example at the end of TEACH *VISION3. Here is the code to get the edge map, and to display it:

```
vars bin_edges;
vars sigma = 1;             ;;; smoothing
vars t1 = 5, t2 = 10;       ;;; hysteresis thresholds
canny(image, sigma, t1, t2) -> (, , bin_edges);
float_threshold(0, t1/2, 1, bin_edges, false) -> bin_edges;

2 -> rci_show_scale;
rci_show(bin_edges) -> rc_window;
rci_show_setcoords(bin_edges);
```

The call to **rci_show** differs from previous usage in assigning its result to **rc_window**. This will allow us to draw the lines we find on the display. The final line sets up the *RC_GRAPHIC scales and origin to facilitate this.

Since the boundslist of **bin_edges** is [83 173 67 188], its centre is close to **column** 128, **row** 128. So to get from **row** and **column** in the image to **x** and **y**, you can just subtract 128 from each of them. (Of course **y** will then run downwards again - reverse it if you wish.)

You should now be able to write Hough transform code that creates and fills a straight line accumulator array, using the data from **bin_edges**. It is not very complicated. You need to create an array with boundslist [0 99 0 99] and initialise it to zero. Then you will have a double outer **for** loop to scan the edge array. For each non-zero pixel encountered, the code will loop over **theta**, as in the examples above, but incrementing **theta** by 3.6° rather than by 10° on each step. For each **theta**, **r** will be found as before, but instead of drawing something, the code needs to add 1 to the appropriate element of the accumulator array. It will have to test for negative values of **r**, and ignore them.

If you want to try that, do so now. Rather than embedding more code in this teach file, the results you

should get will be demonstrated using an efficient library program, *STRAIGHT_HOUGH:

```
vars accumulator, rtheta, xc, yc;
straight_hough(bin_edges, false, 100, 100, false, false,
    false, false, false, false) -> (accumulator, rtheta, xc, yc);
```

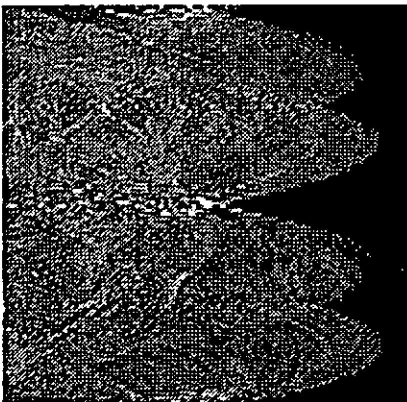
The <false> arguments just correspond to options that we do not need to use at the moment. The **rtheta** result will be described below, and **xc** and **yc** report the position in the array of the origin of **(x, y)** coordinates. The **accumulator** result is the accumulator array, which we can display as for an image:

```
rci_show(accumulator) -> ;
```

Although there are no axes in this display, they are laid out as for the earlier windows, except that **theta** now runs down the screen. The display may look like a rather murky view of a nebula, but a few bright dots are clear - particularly those at the top, where **theta** = 0°, and in the middle, where **theta** = 180°. These correspond to the strong vertical lines in the image, which get more votes by far than any other lines.

We can see more structure in the accumulator array if we compress the brightness scale for the higher values. One way to do this is to take the square root of all the accumulator bins before display. This sort of thing is very simple in Pop-11:

```
rci_show(mapdata(accumulator, sqrt)) -> ;
```



If there is still not a lot visible, try adjusting the brightness of your screen.

The shapes that now appear will be familiar from the earlier graphs. You can see how brighter points are visible where the curves for separate edge elements intersect.

Mapping accumulator peaks to image lines

Having got a filled accumulator array, the next thing to do is to locate the peaks in it. It is easy to write a program to find the location of the largest element of an array, or you can use **array_peak** from the *ARRAY_PEAKS library. This will tell you that the largest element of the accumulator array given by **straight_hough** is at column 39, row 0 You need to convert that into **r** and **theta** This is not difficult, but in fact **straight_hough** helps by returning as its second result a Pop-11 procedure for doing the conversion.

This makes it simple to plot the line corresponding to the peak in question We can use **rt_draw_line** to

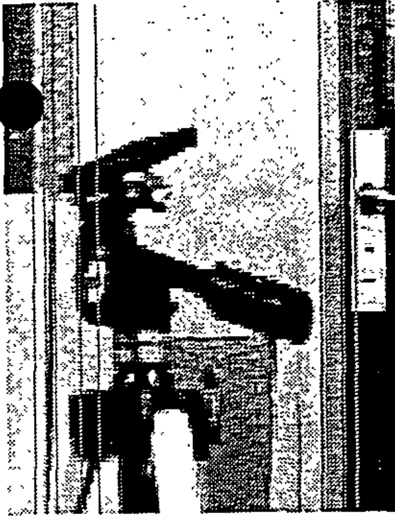
draw on the edge display, provided we move the coordinate system to have its origin at (xc, yc).

```
rtheta(39, 0) -> (r, theta);        ;;; convert to r, theta
rc_shift_frame_by(xc, yc);          ;;; shift coord origin
XpwSetColor(rc_window, 'red') -> ;
draw_rt_line(r, theta);
```

The long vertical line now marked in red is, not surprisingly, the one that received the most votes in the accumulator array. (The red line was drawn on the edge map display, so if that has got covered up, you will need to uncover it by removing or destroying windows. Don't forget that windows created by **rci_image** can be removed just by clicking on the image.)

Smaller peaks in the accumulator array correspond to shorter lines. By giving some more arguments to **straight_hough**, we can tell it to return a list of these peaks in descending order of number of votes. For details see the help file. The next chunk of code illustrates the results by plotting some of the most prominent lines on the original image, using a procedure provided in lib *STRAIGHT_HOUGH for finding the coordinates at which a line crosses the image boundary.

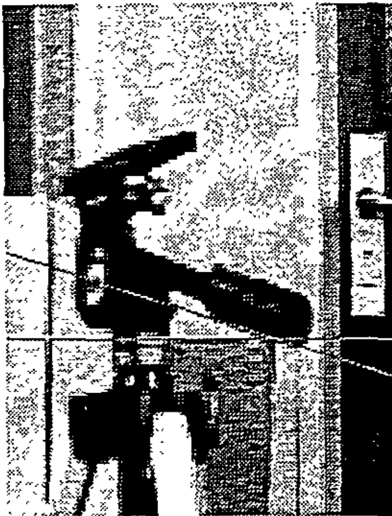
```
;;; get the list of accumulator peaks
vars peaklist;
straight_hough(bin_edges, false, 100, 100, false, false, false,
    false, 5, false) -> (accumulator, peaklist, xc, yc);

rci_show(image) -> rc_window;
rci_show_setcoords(image);      ;;; prepare to draw on image
XpwSetColor(rc_window, 'red') -> ;

vars peak, x0, y0, x1, y1;
for peak in peaklist do
    hough_linepoints(peak, xc, yc, image) -> (x0, y0, x1, y1);
    rc_jumpto(x0, y0);
    rc_drawto(x1, y1);
endfor;
```



The strong vertical lines are naturally the ones detected. The penultimate argument to **straight_hough** is a threshold, determining how many votes are needed before a peak appears in the results (relative to the average number of votes in a bin). By reducing this threshold you can see more lines - ultimately the horizontal line in the middle is picked up, but you will also find that the edge elements start to interact to produce inaccurate or spurious lines.

You can restrict attention to near the tripod handle with a boundslist-type argument, and so detect the diagonal and horizontal lines in this region, with

```
    straight_hough(bin_edges, [120 160 130 150], 20, 20, false, false,
        false, false, 5, false) -> (accumulator, peaklist, xc, yc);
```



A 20 x 20 accumulator is appropriate because the small region means that line parameters are less precisely fixed by the data. Repeat the display code above to see the lines found. Since the Hough transform is a *global* method, in that it integrates information over the entire image or region that it is given, it is not surprising that we need a way of focussing attention if it is to give information about features of limited size.

Finer points of the straight-line Hough transform

In order to discuss the main principles, this teach file has skated over some quite important details

Negative r values

One detail is the question of why it is safe to ignore negative **r** values returned by **line_r** Going back to the original explanation of line parameters, a negative **r** means that after turning through **theta**, you walk backwards away from the origin. This is just equivalent to turning through an extra 180° and walking forwards - in other words a line with parameters **(-r, theta)** is identical to one with parameters **(r, theta+180)**. If negative values of **r** were included in the parameter space, each line would get recorded twice. In fact, in an efficient implementation, **theta** only needs to run from 0 to 180°, and the negative **r** parameters are converted using **-r -> r** and **theta + 180 -> theta** before being recorded.

Smoothing and peak detection

A second issue is how to detect peaks in the accumulator array. Finding local maxima (i.e. elements with values larger than those of their 8 neighbours) is the obvious thing to do. However, if the "true" parameters of a line happen to lie close to a boundary in the quantised parameter space, the votes will get spread over two or more bins, and looking at single bins may not reveal the peak. This can be helped to some extent by smoothing the accumulator array using convolution, before searching for peaks. In addition, one can sometimes do better than assuming that the peak position is at the centre of a bin - if an adjacent bin is large as well, it might be worth averaging their indices to estimate the peak position. The *STRAIGHT_HOUGH library incorporates these refinements in its peak detection - see the help file for details.

Edge weighting

We have allowed each edge element in the **bin_edges** array the same number of votes. It seems reasonable that edge elements supported by a strong grey-level gradient should have more votes. It is easy to implement this: instead of incrementing the accumulator bins by 1 each time, you add in *edge strength* values depending on the size of the gradient for the current edge element. The **straight_hough** procedure implements this directly, and you can try it out easily by omitting the thresholding step used to binarise **bin_edges**. Alternatively, run the code in TEACH *STRAIGHT_HOUGH. You should find that the performance is somewhat improved. Using weighted votes this way fits in well with the idea of accumulation of evidence - strong edges provide more evidence. In fact, it is possible to relate the Hough transform to formal theories of inference, such as Bayesian theory.

Use of gradient direction

Thresholding and edge weighting make use of the gradient magnitude, but can we also make use of the gradient direction? In fact, if we assume that the orientation of the edgelet at each pixel is known (e.g. it might be taken to be at right angle to the gradient direction), then we can simplify the Hough transform considerably. Each edge element then only votes for one bin in the accumulator, since **theta** is fixed by the edge orientation.

This has not been used here, for two reasons. First, the local gradient direction is rarely accurate enough to allow us to implement this scheme reliably as it stands - the information has to be used much more carefully. Second, the Hough transform presented here illustrated the general method a good deal better, and is more amenable to generalisation.

Hierarchical Hough transforms

I mentioned earlier that the choice of quantisation is a big problem. One way round that is to start by doing a Hough transform with a very coarse quantisation of parameter space. You then subdivide those bins with lots of votes, and do another transform into just these parts of parameter space, with the finer quantisation. The process is repeated until the bins are as small as necessary.

This method has been used successfully, but may be delicate: a lot of small peaks in one part of parameter space will stop you seeing a single larger peak in another part.

Hough transforms for other shapes

The Hough transform is not restricted to detecting straight lines, though that is a common use. Other geometrical shapes that can be described with a few parameters are also well suited to it.

For example, a circle is specified with three numbers: the **X** and **Y** coordinates of its centre, and **R**, its radius. To find circles using a Hough transform, you need a three-dimensional accumulator array. Each edge element votes for all the circles that it could lie on, and the 3-D array is searched for peaks that give circle positions and radii. If you happen to know the radius in advance, you only need a 2-D accumulator - you should not find it difficult to implement this yourself if you want to experiment.

More complicated shapes can be found - a general ellipse, for example, needs a 5-D parameter space. Hough transforms have also been used for finding vanishing points - the points to which the images of parallel sets of lines appear to converge.

Finally, the method can be generalised to detecting any shape that can be represented by a finite number of line segments, but which may appear at any orientation, scale and position in the image. The parameter space is, in general, 4-dimensional (2 for position, 1 for orientation, 1 for scale), and it helps if straight line segments are extracted from the image first. This *generalised Hough transform* is described in several of the books mentioned in TEACH *VISION, for example Ballard & Brown's 'Computer Vision'.

Summary

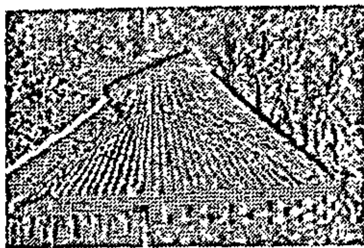
You should now have a clear grasp of how the Hough transform for straight line detection works, and you should have some idea of how it can be generalised.

Here are links to:

- The index of teach files
- The next file in the series
- The School of Cognitive and Computing Sciences home page

Copyright University of Sussex 1994. All rights reserved.

STEREOSCOPIC VISION AND PERSPECTIVE PROJECTION


David Young, February 1993 revised Jan 1994

This teach file gives an introduction to *stereoscopic vision*. It also describes *perspective projection*.

Contents

Please read the introduction giving general information about the nature of this document.

- Preliminaries
- Introduction
- Image and camera coordinates
- Perspective projection
- Stereo geometry for parallel cameras
- Stereo geometry for converging cameras
- Three-D layout from a stereo pair
- The correspondence problem
  - Matching using feature similarity
  - Other matching methods
- Summary

Preliminaries

You need to have read and understood TEACH VISION1. It will be helpful to have looked at VISION2, VISION3 and VISION4.

This is a long teach file, but it has not been split as the material belongs together. You could split it yourself in two ways. The first is to take it in two parts, perhaps breaking after the section on "Stereo geometry for converging cameras". The second way, which is possibly better, is to go through the file a first time without paying too much attention to the equations and Pop-11 code, but executing the latter so as to see the demonstrations; then on a second pass you could study the equations and look at the code more carefully to see how the ideas are put into practice.

Before proceeding, execute the following chunk of code to avoid having to load the libraries later.

```
uses popvision          ;;; search vision libraries
uses rci_show           ;;; image display
uses rc_context         ;;; allows multiple windows for graphics
uses rc_graphplot       ;;; graph drawing
```

```
uses arrayfile              ;;; array storage
uses float_byte             ;;; type conversion
uses float_arrayprocs       ;;; array arithmetic
uses canny                  ;;; Canny edge detection
```

We will need two images of a scene, from different viewpoints, so execute this now:

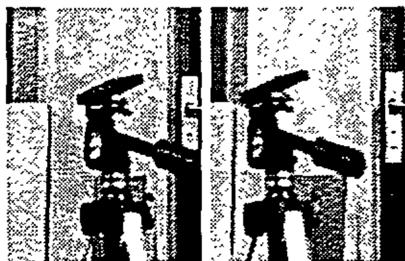
```
vars imageR, imageL;
arrayfile(popvision_data dir_>< 'stereo1.pic') -> imageR;
float_byte(imageR, false, false, 0, 255) -> imageR;
arrayfile(popvision_data dir_>< 'stereo2.pic') -> imageL;
float_byte(imageL, false, false, 0, 255) -> imageL;
```

Introduction

Viewing a scene from two (or more) different positions simultaneously allows us to make inferences about 3-D structure, provided that we can match up corresponding points in the images. The visual systems of humans and some other animals make use of this, and it is very important in attempts to develop practical computer vision systems.

Display the *stereo pair* of images which we read in above, with:

```
1 -> rci_show_scale;          ;;; keep them small
500 -> rci_show_x;           ;;; set the position for the L window
350 -> rci_show_y;
rci_show(imageL) -> ;
rci_show_x + 120 -> rci_show_x;  ;;; position for the R window
rci_show(imageR) -> ;
```



(This code carefully positions the windows beside one another. If you want to move the windows, do it by changing the values for **rci_show_x** and **rci_show_y** and re-executing the code, to keep the relative positions the same. Remember that you can get rid of unwanted image windows just by clicking on the image.)

You may find it possible to fuse these images in your own visual system. I did it by iconifying all the other windows so that only the stereo pair was visible against a plain background. (If you have a patterned X background enlarge an xterm or other window to cover most of the screen, and put it behind the images by selecting "back" from its titlebar menu.) I then looked at the images from a distance of about 10 inches (25 cm). So that I could only see the left image with my left eye and the right image with my right eye, I held a piece of A4 card up so that one long edge was against the screen between the two images, and the opposite edge was touching my nose. I consciously relaxed my eyes as if I was looking at something in the distance; although this made the images go out of focus, after a while I could get them to fuse as if I was looking at a single picture. I found I could then get this back into focus without breaking the fusion. At that point it looked as if there was a tiny tripod clearly standing out in front of the screen, with the door handle and other objects in the background further away.

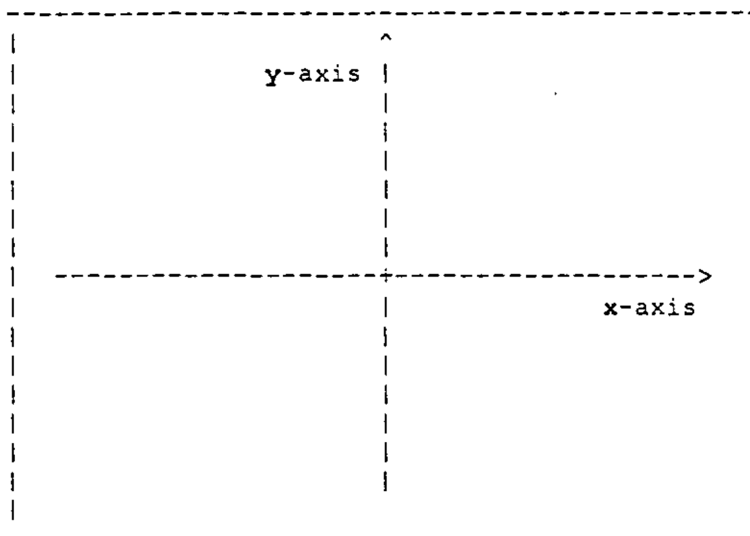
In fact, the Victorians invented technology for presenting stereo pairs more conveniently than this, and were enthusiastic about viewing them. You can still find Victorian stereoscopes and collections of stereo pairs in antique shops. Every so often, film-makers discover they can use stereoscopic presentation to dramatic effect, though the audience has to wear glasses with coloured or polarising filters to ensure that a different image goes to each eye. Map-makers use stereo presentation of aerial photographs to estimate ground topography.

The stereo pair of images on the screen was obtained using two camera positions separated horizontally. Viewing the images normally, you can see how the change in camera position has resulted in a change in the position of the tripod's image relative to the images of other objects. There are two computational issues to address: first, how these image differences can be translated into 3-D information; and second, how the images can be matched to one another so that the differences between them can be measured.

Image and camera coordinates

Before we can look at stereo geometry, we need to introduce the relationship between image and object positions for a single camera.

Positions in the image will be expressed using a coordinate system with its origin at the centre of the image, **x** running from left to right, and **y** running from bottom to top:

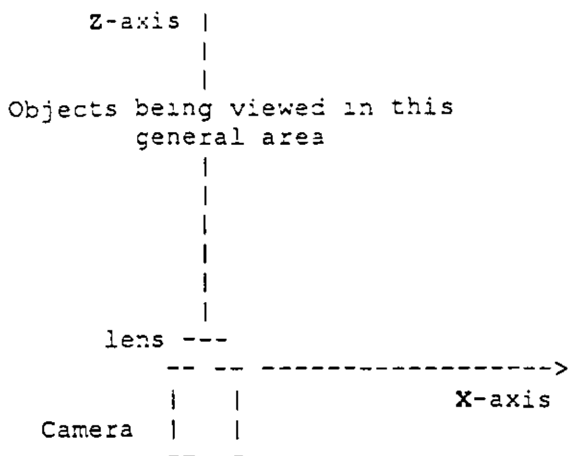


It is easy to convert from **x** and **y** to **column** and **row** indices in the array representing the image. If the image centre is at column **c0** and row **r0**, then **column** = **x** + **c0** and **row** = **r0** - **y**.

Positions in space will be expressed using 3-D coordinates **X**, **Y** and **Z** (not to be confused with the **X** and **Y** used in some earlier files as synonyms for **column** and **row**). The (**X**, **Y**, **Z**) system has its origin at the camera lens, with **Z** pointing out along the optic axis of the camera (i.e. at right angles to the image plane), **X** parallel to **x**, and **Y** parallel to **y**.

Suppose we have a situation where **Y** is actually vertical - as is the case for the tripod test images. Then looking down on the camera and scene from above gives this layout of the **X** and **Z** coordinate axes :

^



The **Y** axis is pointing at you out of the screen. The intersection of the axes is at the optical centre of the camera lens. The (**X**, **Y**, **Z**) system is called *camera coordinates* because it is fixed with respect to the camera, centred on it, and aligned with it. In robotics, other coordinate systems fixed with respect to an object being manipulated, part of a robot arm, or some mechanical framework in which the robot is working, are often employed as well.

Perspective projection

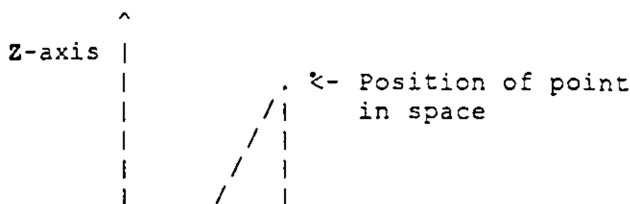
What is the relationship between the camera coordinates of a point in space and the coordinates of its image? If we assume a camera with a planar image surface and no lens distortions (and this *camera model* is usually quite close to reality), the mathematical form of the relation is quite simple:

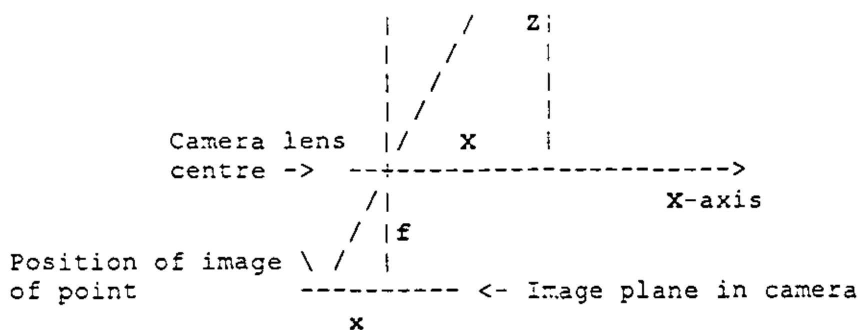
$$x = \frac{X f}{Z}$$

$$y = \frac{Y f}{Z}$$

These are the equations of *perspective projection*. The constant f is the *focal length* of the camera, and is the distance from the optical centre of the lens to the image plane. **X**, **Y** and **Z** specify the position of a point in space; **x** and **y** specify the position of its image in the image plane. These equations allow us to work out **x** and **y** if we know **X**, **Y** and **Z**, but do not allow us to go in the other direction, because knowing **x** and **y** is not sufficient on its own to pin down the position of the point in 3-D.

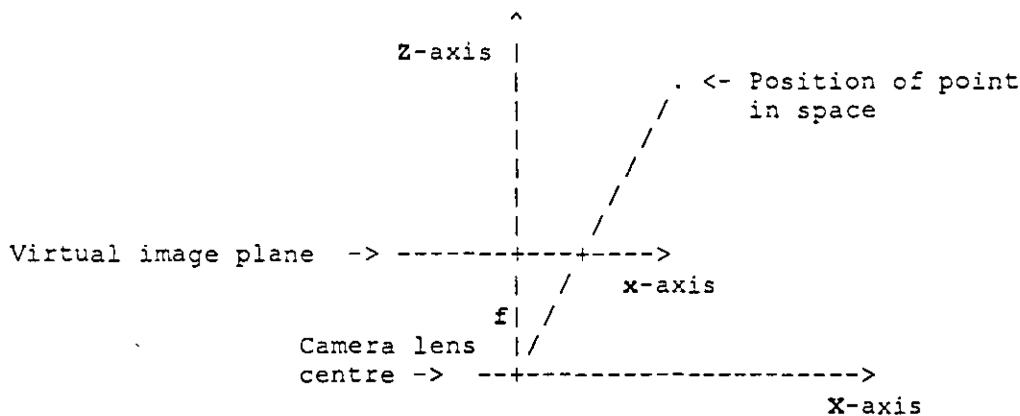
There is a simple geometrical derivation of these formulae. A lens behaves approximately like the pinhole of a pinhole camera (except it produces a much brighter image), and so we can draw a straight line from a point in space, through the lens centre, to its image. For a point on the same horizontal level as the camera (**Y** = 0), the resulting diagram, looking down on the scene as in the diagram above, is:





The relation between **x** and **X** now follows from similar triangles (via **x/f = X/Z**). Since a movement along the **Y** axis will not affect **x**, this relation continues to hold even if **Y** is non-zero. The corresponding relation between **y** and **Y** is obtained by drawing the side-view diagram.

It looks from the diagram above as if the **x**-axis has to run from right to left, in the opposite direction to the **X**-axis. However, if you look at the image actually formed in the camera it is left-to-right inverted and upside-down compared with the way we display the digitised image on the screen. (You can see this with an ordinary camera by putting tracing paper where the film should go, and keeping the back and the lens open.) When we display the image "right way round" on the screen, the **x** and **y** axes get flipped, so that they run in the same directions as the **X** and **Y** axes. The easiest way to deal with this is to imagine that the camera forms a *virtual image* which is *in front of* the lens instead of behind it, and to pretend that this is what gets digitised. The diagram then looks like this:



This is just a different way of modelling the camera; it is less like the reality in terms of physical layout, but it is easier to work with than the image-behind-the-lens model because the virtual image is "right way round". The similar triangles argument still applies, and gives the same mathematical relationships between **x** and **X**, and **y** and **Y**, so the models are formally equivalent.

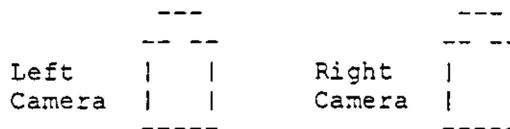
The projection equations obviously work if all the quantities are measured in the same units. However, we would normally like to measure distances in the image in *pixel units*, where 1 pixel unit is just the distance between the centres of two neighbouring pixels, because then array indices correspond to position. Distances in the scene will usually be measured in something like inches or millimetres. It turns out that the **x** equation is still valid as long as **x** and **f** are in the same units as one another, and **X** and **Z** are in the same units as one another (and similarly for the **y** equation). Thus if we know **f** in pixel units, everything works out conveniently. Finding the value of f in pixel units is basic *camera calibration*, and is usually best done by making measurements of the image of an object of known size and distance It is also possible to estimate **f** in pixel units from the focal length of the lens (usually stamped on it in millimetres) and the number of pixels per millimetre in the camera detector

'ery high accuracy requires a camera model which takes into account lens distortions and detector irregularities. Such models require more elaborate calibration.

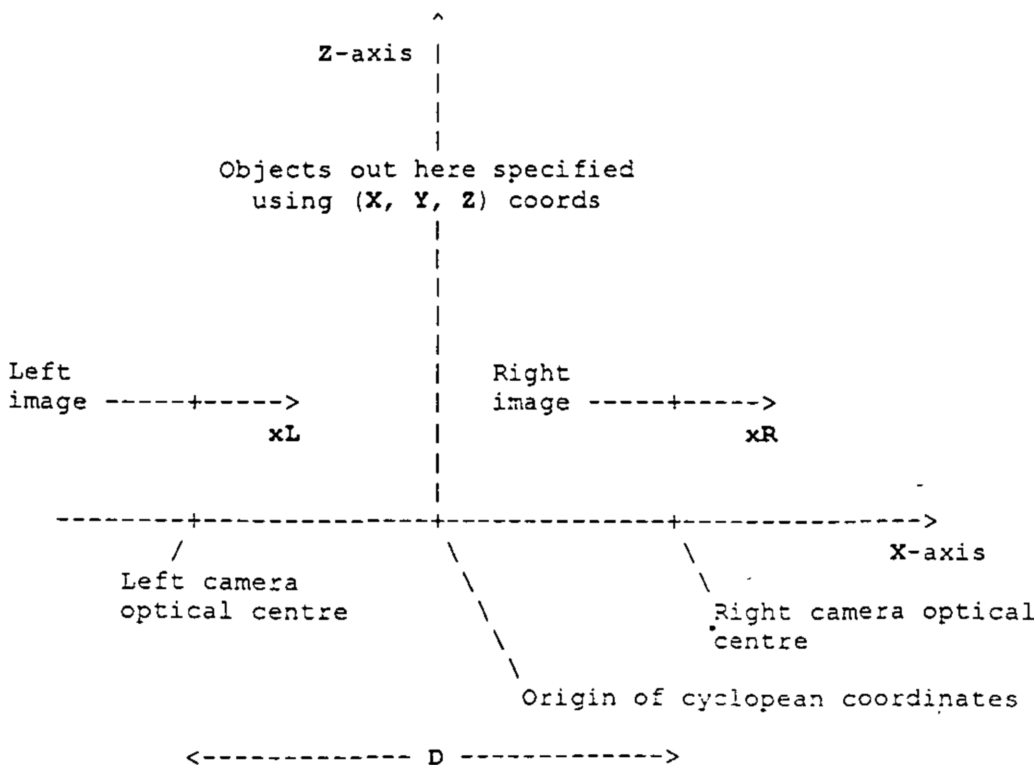
Stereo geometry for parallel cameras

Now suppose we have two cameras imaging the scene. It will be simplest to start by assuming that their optical axes are lined up parallel to one another. We also assume that they are side by side - or more exactly, that the line joining their optical centres is parallel to their **x**-axes. This means that the image of a point will have the same **y** coordinate for the two cameras The top view is something like:

```
                    Objects being viewed

                   in this general region
```



ɔsitions in space could be described in either of the two camera coordinate systems, but it is convenient ɔ describe positions relative to an imaginary third camera half-way between the two real cameras. We can call this the *cyclopean* coordinate system (the cyclops only had one eye in the middle of its head). Schematically, looking down on the scene:



Here **xL** and **xR** are the **x** coordinates for an image point in the left and right images respectively. **D** is the separation between the cameras' centres

Now if a point at (**X**, **Y**, **Z**) in cyclopean coordinates produces images at (**xL**, **y**) and (**xR**, **y**) in the two cameras, we can find its position in space from the image positions using the formulae:

$$X = \frac{D\ (xL + xR)}{2\ (xL - xR)}$$

$$Y = \frac{D\ y}{xL - xR}$$

$$Z = \frac{D\ f}{xL - xR}$$

The quantity **xL-xR** which appears in all three formulae is called the *stereo disparity* (or just the disparity) of the point. It is the difference in the horizontal positions of the images of a point.

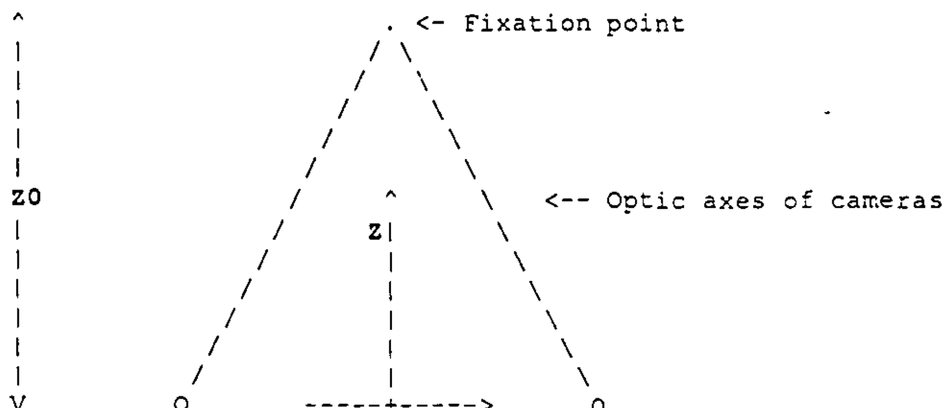
D must be measured in the same units as **X**, **Y** and **Z**, and **xL**, **xR**, **y** and **f** can all be in pixel units.

Deriving these equations is fairly straightforward. The main thing to use is that the perspective projection equations in **x** for the left and right cameras become **xL = (X + D/2)f/Z** and **xR = (X - D/2)f/Z**. This is because to get from **X** in cyclopean coordinates to **X** in left or right camera coordinates you simply adjust for the offsets of the cameras from the cyclopean origin, whilst **Z** is the same in all three coordinate systems. A little algebra then gives the expressions above for **X** and **Z**, and that for **Y** follows easily.

Note the structure of the **Z** equation. **Z** is often termed the *depth* of an object relative to the cameras. A small disparity yields a large depth, and *vice versa*.

Stereo geometry for converging cameras

It is not usually convenient to set up the cameras with their axes parallel, because this limits the region of space in which objects are visible in both images. It is more normal to aim the cameras so that their axes are angled inwards, and converge on the objects of interest. The point in space where the optical axes intersect might be described as the *fixation point* for the cameras. The image of an object at the fixation point is centred for both cameras, and so has zero disparity.



```
                Left                    X       Right
                Camera                          Camera
```

It turns out that this makes the geometry considerably more complicated, and the exact equations for getting from image points to space points are much less straightforward than for parallel cameras. Matching points will not even generally lie at the same y coordinate. Provided the amount of convergence is small, though, we can ignore this *vertical disparity*, and there is a reasonable approximation which can be used for the equations. Call the depth of the fixation point **Z0**. Then the disparities can be adjusted by adding the quantity **fD/Z0**, which is the disparity an object at the fixation point would have if the axes were parallel. The new equations become:

$$X = \frac{D\,(xL + xR)}{2\,p}$$

$$Y = \frac{D\,y}{p}$$

$$Z = \frac{D\,f}{p}$$

where $$p = xL - xR + \frac{f\,D}{Z0}$$

Z0 is another parameter that must be found by calibration.

It will be convenient to have the formulae available as a Pop-11 procedure, which we define next.

```
define images_to_world(colL, colR, row, c0, r0, D, f, p0)
        -> (X, Y, Z);
    ;;; Calculates the cyclopean coordinates of a point, using
    ;;; the near-parallel approximation. Arguments are:
    ;;;     cL - the column in the left image
    ;;;     cR - the column in the right image
    ;;;     row - the row, assumed same for both images
    ;;;     c0 - the column at the origin of image coords
    ;;;     r0 - the row at the origin of image coords
    ;;;     D - the camera separation in world units
    ;;;     f - the focal length in pixel units
    ;;;     p0 - the disparity shift in pixel units.
    lvars colL, colR, row, c0, r0, D, f, p0, X, Y, Z;
    ;;; Convert from col/row to image coords
    lvars xL, xR, y;
    colL - c0 -> xL;
    colR - c0 -> xR;
    r0 - row -> y;
    ;;; Get adjusted disparity
    lvars p;
    xL - xR + p0 -> p;
    ;;; and apply stereo formulae
    D * (xL + xR) / (2 * p) -> X;
    D * y / p -> Y;
    D * f / p -> Z
enddefine;
```

This procedure implements our simplified model of *stereo geometry*.

Three-D layout from a stereo pair

(This section and the following one incorporate a fair amount of Pop-11 code. This is to make it easy for you to see what is being done at the programming level, if you want to, and to experiment. However, as pointed out at the start of the file, you can follow the main points simply by executing the Pop-11 sections when you get to them, and reading the text between - you can always return to look at the programs later.)

We are now in a position to work out some of the layout of the tripod stereo pair. We will assume the following parameters for the camera set-up:

```
vars
    c0 = 128,      ;;; Column at x-origin, c0     =  128
    r0 = 128,      ;;; Row at y-origin, r0        =  128
    D  =  20,      ;;; Separation, D              =   20 cm
    f  = 200,      ;;; Focal length, f            =  200 pixel units
    Z0 = 100;      ;;; Fixation distance, Z0      =  100 cm
```

We can define a procedure that converts from image positions to cyclopean coordinates for this specific set-up by making a closure (see HELP *CLOSURES) of our general procedure. We do this with:

```
define im_to_w =
        images_to_world(% c0, r0, D, f, D*f/Z0 %)
enddefine;
```

Now we need to get some matching points. We will take the edges from the Canny edge detector (see TEACH VISION3) as the features to match, so first apply the Canny operation, and threshold the results to give binary edge maps:

```
vars cannL, cannR;
vars sigma = 1, t1 = 5, t2 = 10;    ;;; Canny parameters
canny(imageL, sigma, t1, t2) -> ( , , cannL);
float_threshold(0, t1/2, 1, cannL, cannL) -> cannL; ;;; threshold
canny(imageR, sigma, t2, t2) -> ( , , cannR);
float_threshold(0, t1/2, 1, cannR, cannR) -> cannR; ;;; threshold
```

Display the edge maps with:

```
2 -> rci_show_scale;                      ;;; bigger is easier to see
vars winCL, winCR;                        ;;; to save the windows
500 -> rci_show_x;                       ;;; position the windows
rci_show_y + 170 -> rci_show_y;
rci_show(cannL) -> winCL;
rci_show_x + 240 -> rci_show_x;
rci_show(cannR) -> winCR;
```

(Output is displayed a little further on.)

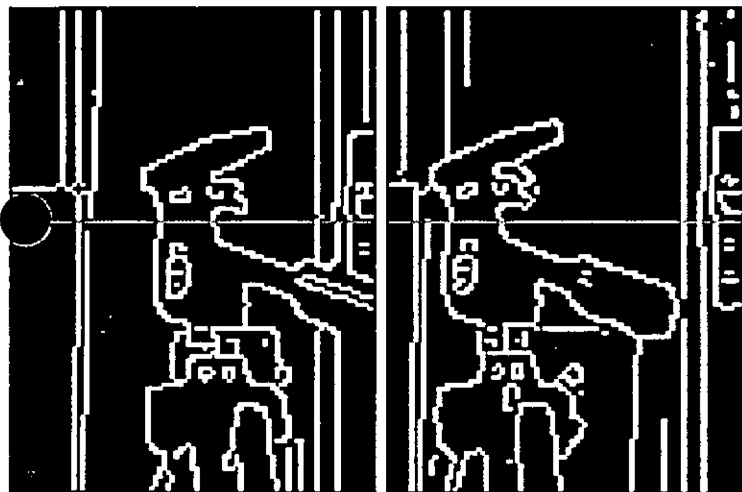
We need a procedure for getting the features in a given row. This is simple enough:

```
define features(image, cstart, cend, row) -> list;
    ;;; Returns a list of the column numbers for which there are
    ;;; non-zero pixels in the given row of the image, between
    ;;; columns cstart and cend.
    lvars image, cstart, cend, row, list;
```

```
        lvars col;
        [%                    ;;; start building list
            for col from cstart to cend do
                if image(col, row) /= 0 then
                    col                 ;;; put this column in the list
                endif
            endfor
        %] -> list
    enddefine;
```

Now we choose a row - say 120, which goes through the tripod head. We'll display the position of this row on the screen:

```
    vars row = 120;             ;;; set up the row
    winCL -> rc_window;         ;;; draw on the left edge map
    rci_show_setcoords(cannL);  ;;; set the scales correctly
    XpwSetColor(rc_window, 'red') -> ;
    rc_jumpto(83, row);         ;;; draw a line along one row
    rc_drawto(173, row);
    winCR -> rc_window;         ;;; draw on the right edge map
    XpwSetColor(rc_window, 'red') -> ;
    rc_jumpto(83, row);
    rc_drawto(173, row);
```



and collect the right and left features for it, for all columns from 83 to 173 (i.e. the whole width of the image):

```
    vars featuresL, featuresR;
    features(cannL, 83, 173, row) -> featuresL;
    features(cannR, 83, 173, row) -> featuresR;
    featuresL ==>
prints:
    ** [100 120 133 159 163 167]
    featuresR ==>
prints:
    ** [90 93 99 112 158 162 166]
```

The list for the right image has one more entry than that for the left image. Inspecting the edge maps shows that this is due to the fact that the second edge from the left, at column 93 in the right image, is not ~esented in the left image. We can therefore make the two lists correspond by omitting this feature - of ~urse, this is cheating, and a real stereo system will have to do this automatically. For now, though, do

```
vars reduced_featuresR;
delete(93, featuresR) -> reduced_featuresR;
```

Now we can look at what layout the stereo procedure gives. We need a diagram looking down on the tripod - this will produce it, showing the cyclopean coordinate system:

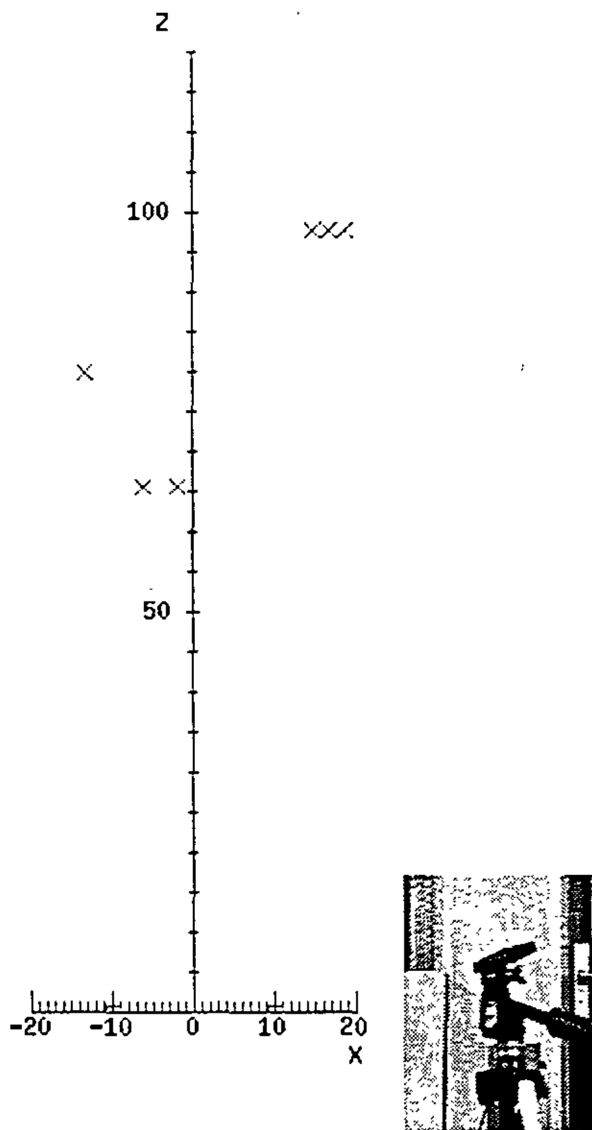
```
false -> rc_window; ;;; avoid destroying present rc_window
rc_new_window(200, 600, 10, 10, false); ;;; start a new window
[-20 20 0 120] -> rcg_usr_reg;           ;;; set the axis limits
rc_graphplot([], 'X', [], 'Z') -> ;
```

Move the window with the mouse if it is not in a convenient place. We can mark the positions of the two cameras, remembering that their separation is **D**, so they are **D/2** either side of the origin:

```
XpwSetColor(rc_window, 'red') -> ;
rcg_plt_square(-D/2, 0);              ;;; mark X = -D/2, Z = 0
rcg_plt_square( D/2, 0);
```

Now we can apply our stereo procedure to each pair of features, and plot the corresponding positions in the **X-Z** plane:

```
vars colL, colR, X, Y, Z;
for colL, colR in featuresL, reduced_featuresR do
    im_to_w(colL, colR, row) -> (X, Y, Z);
    rcg_plt_cross(X, Z)
endfor;
```



We see the position of the two sides of the tripod head shown at about **Z** = 66, the object at the left further away at about **Z** = 80, and three points on the door frame to the right at about **Z** = 98.

You can easily repeat this with other rows from the image, but it is tedious manually deciding which features to delete in order to make the others match up.

The correspondence problem

The problem of pairing up the features is known as the *correspondence problem*. If we apply no constraints, any feature in the left image could match any feature in the right image. A small modification of the code above allows you to display the positions of all the possible matches between the two lists of features.

We have already used the most basic constraint, when we looked for matching features in a single row of the image That is, we have assumed that a feature with a particular y value can only match a feature in

the other image with the same y value. This is known as the *epipolar constraint*. For converging cameras, using the same y value is actually an approximation, and to be exact we should search for matching features on *epipolar lines*, which are curves close to the rows of the image. We will continue to use the approximation that the epipolar lines are the same as the image rows.

There remains the problem of matches between all the features with the same y. There are three main constraints that can help:

- A given feature can match at most one feature from the other image.
- Similar features match each other.
- Features close together in the image should have similar disparities.

The bases of these constraints are discussed in various books - those by Frisby and Marr (see TEACH *VISION) are probably best. The first constraint is straightforward to apply, but there is considerable flexibility in the way the other two can be used. We will initially explore the use of the second constraint.

Matching using feature similarity

There are many ways of measuring feature similarity. For example, some systems make use of the orientation of the edge. Here we will adopt a measure based on the grey-levels in the image: we will add up the differences in grey-levels in a patch of pixels round the feature, on the assumption that matching regions should have a similar local structure and overall grey-level. To avoid negative and positive differences cancelling out, we will square each difference before doing the addition. Measures like this, based on *sums of squares*, are extremely common. Since our features are likely to correspond to the boundaries of objects, we will look at two patches, one on each side of the feature, and accept a match if either the grey-level differences to the left of the feature are small, or the differences to the right of the feature are small. (Horizontal edges are no good for matching anyway, so taking left-right patches is reasonable, though one could undoubtedly improve on this using edge orientation information.)

This is implemented in the following procedure:

```
define mismatch(col1, col2, row, image1, image2) -> result;
    ;;; Returns a measure of the mismatch between a feature at
    ;;; (col1, row) in image1 and a feature at (col2, row) in
    ;;; image2.
    lvars col1, col2, row, image1, image2, q;
    ;;; These constants set the size of the patches at 5x5
    lconstant rsize = 2, csize = 5;
    lvars r, c, d1, d2, s1=0, s2=0;     ;;; initialise sums
    for r from row-rsize to row+rsize do
        for c from 1 to csize do
            ;;; Get-grey-level differences to left and right
            image1(col1-c, r) - image2(col2-c, r) -> d1;
            image1(col1+c, r) - image2(col2+c, r) -> d2;
            ;;; Add up sums of squares.
            s1 + d1 * d1 -> s1;
            s2 + d2 * d2 -> s2;
        endfor
    endfor;
    min(s1, s2) -> result       ;;; take the smaller mismatch
enddefine;
```

The procedure is called **mismatch** because the larger the result it returns, the worse the match between the two features. To keep the argument lists short, **mismatch** has the parameters controlling the patch

size (5 x 5) built into it. These values happen to be OK for this demonstration, but of course a more general library routine might well have them as arguments.

INSERT INTO ENCIMADAS
SELECT NUMERO_UNICO, NUMERO_UNICO_ENCIMADO
FROM ENCIMADAS_DUP;

For a given feature in the left image, we can now pick the feature in the right image which matches it best. We can also do the same thing in reverse, starting from a feature in the right image and looking for the best match in the left image. If the two agree (i e. two features select each other as closest matches), then it is reasonable to accept the pairing as a definite match. The following procedures implement this Any feature that is only visible in one image should be weeded out. The first constraint (only one match for any given feature) is built in. In addition, to avoid having to check for wild matches, a disparity limit of 40 pixels is rather arbitrarily built in too - this would, of course, be unacceptable in a library procedure.

```
define bestmatch(col1, f2, row, image1, image2) -> b;
    ;;; Returns the best match for the feature at (col1, row) in
    ;;; image1 from the list of features at (col2, row) in image2
    ;;; where col2 is an element of f2.
    ;;; Ignores features with a disparity greater than this limit.
    lconstant disp_limit = 40;
    lvars col1, f2, row, image1, image2, b;
    lvars col2, q, best = false;
    for col2 in f2 do
        if abs(col1 - col2) <= disp_limit then
            mismatch(col1, col2, row, image1, image2) -> q;
            if not(best) or q < best then
                q -> best;
                col2 -> b
            endif
        endif
    endfor
enddefine;

define findmatches(fL, fR, row, imageL, imageR) -> (newfL, newfR);
    ;;; Given a list of features fL in the left image and fR in the
    ;;; right image, returns two lists of the same length which
    ;;; contain features in one-to-one correspondence.
    lvars fL, fR, row, imageL, imageR, newfL = [], newfR = [];
    lvars colL, colR;
    for colL in fL do
        ;;; Find best match for left feature
        bestmatch(colL, fR, row, imageL, imageR) -> colR;
        ;;; Find best match for right feature, and see if it agrees
        if colR     ;;; test there is some matching feature
        and bestmatch(colR, fL, row, imageR, imageL) == colL then
            colL :: newfL -> newfL;
            colR :: newfR -> newfR
        endif
    endfor;
    ;;; Make the lists run left to right (not essential)
    ncrev(newfL) -> newfL;
    ncrev(newfR) -> newfR
enddefine;
```

That then provides the basis of a simple stereo matching program, using grey-level matching starting from edge features. Applied to the features we had before, it gives.

```
    findmatches(featuresL, featuresR, row, imageL, imageR)
        -> (featuresL, featuresR);
    featuresL =>
prints:
    ** [100 120 133 159 163 167]
    featuresR =>
```

prints:
```
 ** [90 99 112 158 162 166]
```

- the feature at column 93 in the right image has been omitted, as required for a correct set of matches.

We can check how this behaves on the complete images. We need a procedure to iterate over rows and get all the matches We will store each match as a vector containing the column and row numbers, and place all of them in one long vector.

```
define findallmatches(imageL, imageR, edgeL, edgeR) -> matchvec;
    ;;; Returns the columns and row for each pair of matching
    ;;; features in the two images. edgeL and edgeR must be the edge
    ;;; maps for imageL and imageR.
    lvars imageL, imageR, edgeL, edgeR, matchvec;
    lvars row, featuresL, featuresR, colL, colR,
        (ic0, ic1, ir0, ir1) = explode(boundslist(imageL)),
        (ec0, ec1, er0, er1) = explode(boundslist(edgeL));
    ;;; Must avoid pixels adjacent to the boundaries to ensure
    ;;; matching procedure does not try to go outside the array.
    ;;; Next constants must correspond to those in mismatch
    lconstant rsize = 2, csize = 5;
    lvars
        cstart = max(ec0, ic0 + csize),
        cend = min(ec1, ic1 - csize);
    {%      ;;; start building big vector
        for row from max(er0, ir0+rsize) to min(er1, ir1-rsize) do
            ;;; Get the features.
            features(edgeL, cstart, cend, row) -> featuresL;
            features(edgeR, cstart, cend, row) -> featuresR;
            ;;; Find one-to-one matches
            findmatches(featuresL, featuresR, row, imageL, imageR)
                -> (featuresL, featuresR);
            ;;; Get the matches for this row into vectors
            for colL, colR in featuresL, featuresR do
                {% colL, colR, row %}  ;;; left on stack
            endfor
        endfor
        %} -> matchvec;     ;;; finish building big vector
enddefine;
```

This involves a fair amount of computation, so the next step will take some time:

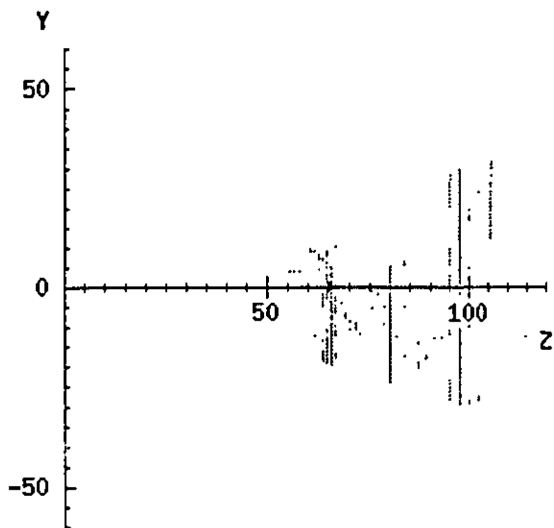
```
vars matchvec;
findallmatches(imageL, imageR, cannL, cannR) -> matchvec;
```

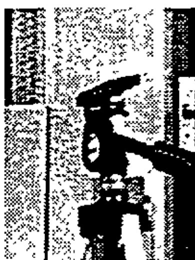
(The program could be made more efficient, since it calculates many match strengths twice.)

We now have all the matches. First, we can use them to plot all the matches in the (**Z**, **Y**) plane, superimposing different **X** values. This gives a side view of the scene, with the camera positions both at the origin. The cameras are looking left to right in the diagram drawn by the code that follows.

```
rc_new_window(300, 300, rci_show_x, rci_show_y, false);
[0 120 -60 60] -> rcg_usr_reg;       ;;; set coord region
rc_graphplot([], 'Z', [], 'Y') -> ; ;;; show coords, set scales
XpwSetColor(rc_window, 'red') -> ;

appdata(matchvec,
    procedure(match);
        lvars match, X, Y, Z;
```

```
        ;;; Use the 2-D to 3-D procedure.
        im_to_w(explode(match)) -> (X, Y, Z);
        rc_drawpoint(Z, Y);
    endprocedure);
```

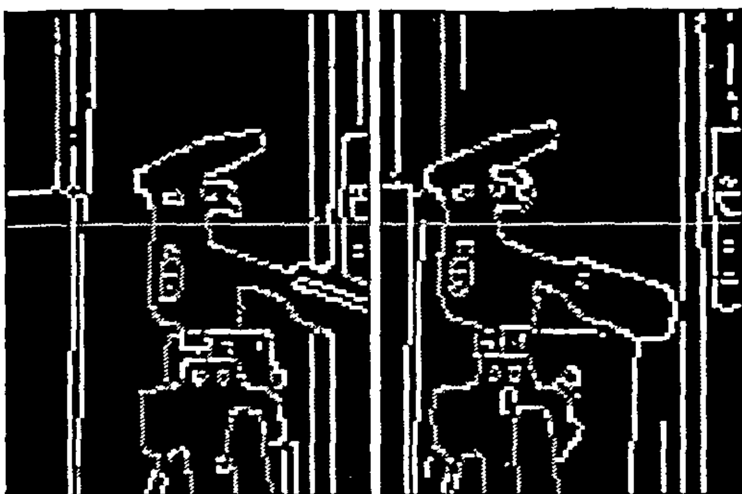




The results are far from perfect, but the tripod at about **Z** = 65, the object at the left at **Z** = 80, and the background objects at **Z** = 90 to 110 show up clearly, with a few false matches scattered around. The results can also be projected onto the (X, Z) plane, which you can try, or indeed plotted as if they were viewed from any given direction, though that requires coordinate rotations which cannot be covered here

Second, we can look at the matches superimposed on the edge maps, by colour-coding the features according to disparity.

```
vars colours = {'red' 'blue' 'green' 'yellow' 'purple' 'orange'
    'pink' 'firebrick' 'magenta' 'brown'};
rci_show_setcoords(cannL);  ;;; Set coord system
appdata(matchvec,
    procedure(match);
        lvars match, colL, colR, row, colour;
        dlocal rc_window;
        explode(match) -> (colL, colR, row);
        ;;; arbitrary mapping from disparities to colours
        colours((colL-colR+40) div 8 + 1) -> colour;
        winCL -> rc_window;
        XpwSetColor(rc_window, colour) -> ;
        rci_drawpoint(colL, row);
        winCR -> rc_window;
        XpwSetColor(rc_window, colour) -> ;
        rci_drawpoint(colR, row);
    endprocedure);
```



This makes it easy to see where incorrect matches have occurred, and also provides a form of *depth map* of the scene, with different colours indicating different distances away from the camera.

Other matching methods

Although matching based on grey-level least-squares differences worked reasonably well, it is computationally quite expensive, and there were some erroneous matches. One way round these problems might be to match higher-level features, such as straight line segments, and this kind of approach is sometimes used in practice.

Another set of techniques involves using the third constraint listed above: that points near each other in the image will usually have similar disparities, because the scene is made of coherent surfaces This constraint has been formalised in various ways, for example in terms of the *disparity gradient limit* used in the system developed by a group at Sheffield, described in detail in TEACH *PMF. Prior to that, the constraint was discussed by Marr and used in an interesting way in his cooperative algorithm for finding correspondences. Using disparity smoothness is probably particularly appropriate when trying to match images with a high density of features, as is the case with random dot stereograms. It would be less suitable for the rather sparse features we have been working with.

Marr also proposed a method of stereo matching based on scale space ideas. In this, the zero-crossings of large-scale DoG filters are matched initially, giving rough estimates for the disparities; these are used as a guide to initial disparities for matching at a smaller scale, and so on until matching is done on the smallest visible details at high accuracy. Nishihara developed this approach in a program intended for real-time stereo matching, which works on the kind of binary images displayed in the zero-crossing section of TEACH VISION3.

A rather different way of estimating stereo disparities has recently become prominent. In this, the outputs of a set of convolution masks, sensitive to particular spatial frequencies (see TEACH VISION3), are used to estimate local phase shifts between the left and right images. This kind of method is probably best suited to images with small disparities.

Summary

On a first reading, you should:

- know what the terms *stereoscopic vision*, *perspective projection*, *stereo disparity*, *fixation point* and *correspondence problem* mean;
- have some idea how the pinhole camera model allows us to relate the positions of points in 3-D to the positions of their images in 2-D;
- have seen how 3-D information is implicit in a stereo pair of images;
- understand how grey-level matching can help solve the correspondence problem;
- be aware of other approaches to solving the correspondence problem.

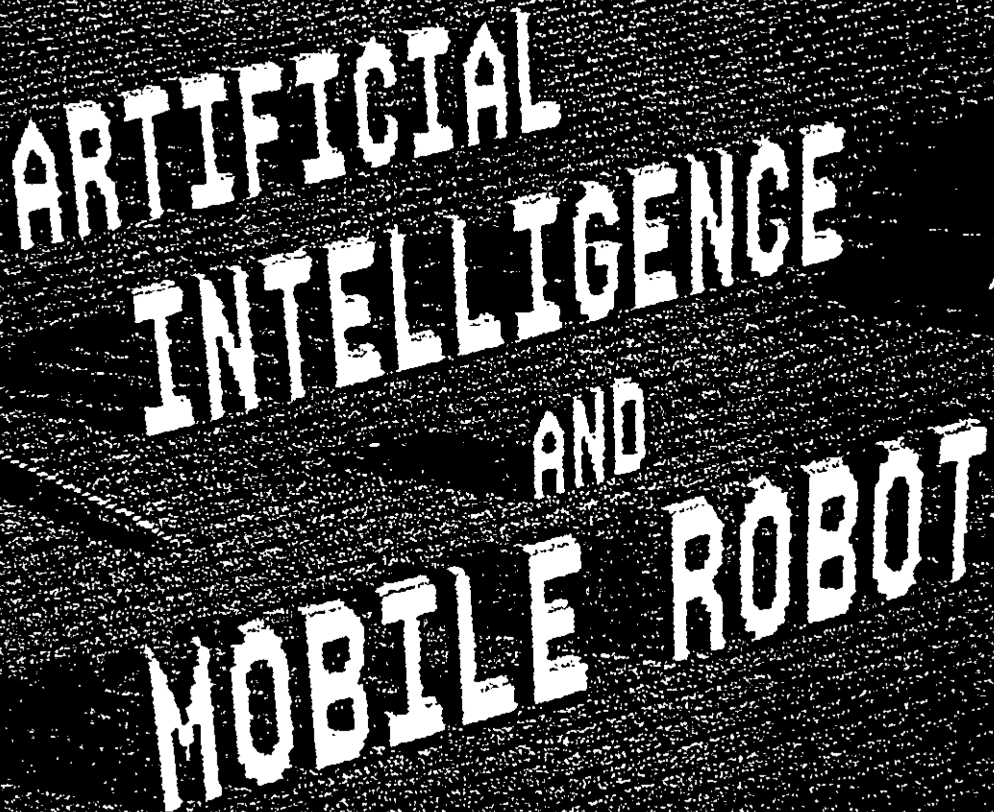
When you have followed the file in detail, you should also:

- understand the image, camera and cyclopean coordinate systems;
- know the equations for perspective projection, and understand how they were derived using similar triangles;
- understand the image-in-front-of-the-lens camera model;
- have looked at, though not necessarily derived, the stereo position recovery equations for parallel and converging cameras;
- understand the term *epipolar constraint*;
- understand the sum-of-squares match quality measure, and how it was applied;
- understand the Pop-11 code used;
- be in a position to try out, if you wish, alternative approaches to stereo matching, and investigate their consequences.

Here are links to:

- The index of teach files
- The next file in the series
- The School of Cognitive and Computing Sciences home page

Copyright University of Sussex 1994. All rights reserved.



C A S E S T U D I E S O
S U C C E S S F U L R O B O T S Y S T E M

E D I T E D B
D A V I D K O R T E N K A M
R. P E T E R B O N A S S
R O B I N M U R P H

INTRODUCTION

Mobile Robots

A Proving Ground for Artificial Intelligence

R. Peter Bonasso, David Kortenkamp, and Robin Murphy

This book is about artificial intelligence (AI) as applied to *mobots*—mobile robots that perform tasks useful to people in an intelligent way. Within these pages, you will find reports on intelligent robots in office environments, outdoors on land and under water, alone or in concert, and in and among people or in remote locations.

Why is this book relevant today? The impetus has come from the success of the American Association for Artificial Intelligence (AAAI) Robot Competition and Exhibition held over the past five years. Since the first show in 1992 in San Jose, California, 30 different teams have competed, and almost that many more have exhibited intelligent robots. Participant teams and exhibitors have come from as far away as Korea and as near as the host city itself, they have included universities, companies, and ad hoc garage-based teams; they have been composed of undergraduates, young entrepreneurs, and senior professors alike. All the teams, however, have been unified in their purpose—to realize robotic agents that, on their own, can serve and work with humans in natural environments. That is their passion. This passion has a rather straightforward technical explanation: to integrate software and hardware to get the robot to act intelligently and autonomously in a given task and environment. The true breadth and depth of this undertaking, however, can only be sensed when you watch one or more of these creatures running unattended in the competition arena. Actor Alan Alda—leaping with excitement at the 1996 competition when one robot, after making little headway in its task, finally succeeded with but scant seconds left on the clock—remarked that only then could he understand why the competitors undertook such a tedious, yet ultimately rewarding, task.

However, the competition is really only a focused reflection of the broader progress that AI researchers have made toward building an artificial person. Research in the field of mobile robots is critical to the AI comm because it forces researchers to connect perception to action to support i ent behav-

tor. This connection can take many forms, as is shown in the chapters of this book, and it lies at the heart of what it means to be an intelligent system. Indeed, the revolution in practical, intelligent robots has been fomented in large measure by AI researchers trying to bring their work to bear on mobile robots. This progress crept along at a snail's pace until a fundamental paradigm shift took place around 1985.

The Sense-Plan-Act Paradigm

The link between mobile robotics and AI was probably first forged with SHAKEY the robot, developed at the Stanford Research Institute in the late 1960s (Nilsson 1969). After the dormant 1970s, the Defense Advanced Research Projects Agency (DARPA) Strategic Computing Program generated renewed research in the 1980s with its autonomous land vehicle (for example, Andresen et al. [1985]). By this time (largely as a result of the funding support from the government), AI research was beginning to produce, among other things, useful automated planning systems. Such large software systems could take a goal, a starting situation, and a desired situation and generate an ordered, finite set of actions—actions for human agents for the most part—that would bring about the desired situation. Written largely in Lisp, such planning systems admitted the intractability of solving general problems by incorporating large amounts of domain knowledge, including domain-specific analytic functions used to predict the future implications of actions. It seemed only reasonable then to apply these systems to robot platforms as a first step in realizing an autonomous artificial agent. The general approach taken, overly simplified, is shown in figure 1.

Sensors from the robot would populate the lower regions of a large structure, usually a complex semantic net, that served as a model of the world. This action could be time consuming—even with simple sensors. Data had to be conditioned to present to the world model as definitive a signal as possible. When the sensor was a camera, all visual data in each frame (typically, 512 x 512 x 8 bits a frame) went through preprocessing, known as *early vision*, to yield the image structures that the world model could process. The world model would interpret the sensor data by percolating the implications up through the network until logical propositions about the state of the world were produced. These propositional accounts of the world state then served as input, along with the goal and possibly some user preferences, to the planning process, which would churn through the possibilities to produce a set of actions that would provably bring about the desired situation. Each step of this plan would then be passed to the control level of the robot for execution, which meant that the plan had to include actions down to the actuator level. For example, if the first part of a plan to search for and retrieve an item was to move to a location, the first two

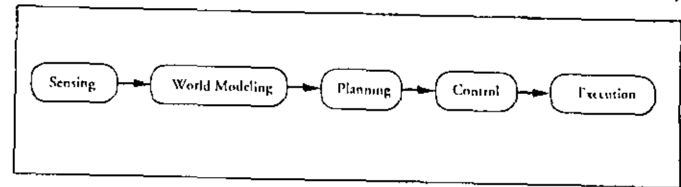


Figure 1. The sense-plan-act (SPA) paradigm.
The world model interpreted the sensor data such that a plan could be generated. Each step of the plan was then executed using the robot control system.

commands might be "turn 0.56 radians" and then "move 122.25 centimeters."

Now the robot is moving. In a few seconds, its cameras detect in its path a large pothole that is not predicted by the world model. The sensor information is processed; the world model is populated; and at some point while the world model is trying to assert that the large black blob on the ground is a depression, not a shadow, the robot falls into the pothole. Even if downward-looking sonars are used, the crash might occur during one of the levels of plan operator decomposition. The point is that the sense-plan-act (SPA) cycle could not run fast enough to keep up with the state of the robot or the world. Much of the early work in applying AI to robots involved making the SPA cycle faster—faster central processing units, more memory, finer-tuned search algorithms—or making the robot run more slowly, indeed, so slowly that it hardly appeared to be moving at all.

From the standpoint of building an artificial person, or ***android***, figure 2 (Bonasso and Dean 1996) describes this situation more generally. The robot's view of the world was transmitted directly to the brain using a general sensing apparatus. The amount of data moving from the sensors to the centralized computing resources was significant; therefore, the communication was slow and expensive. In the brain, the world was reconstructed into a form that the reasoning could process. The robots relied on building and maintaining these complex representations of the environment. These representations were motivated not by the task at hand but by a particular technology concerned with general representations of the world, physically resulting in much off-board computing or very large robots.

Even ignoring the computational overhead, it was difficult to keep the complex representations in sync with the real world. These baroque representations turned out to be impractical and unnecessary. Not only was the representation difficult, if not impossible, to maintain, but the associated planning systems attempted to use similarly rich representations for planning and prediction. Uncertainty made the underlying dynamics difficult to model and the representations of dubious value for most tasks.

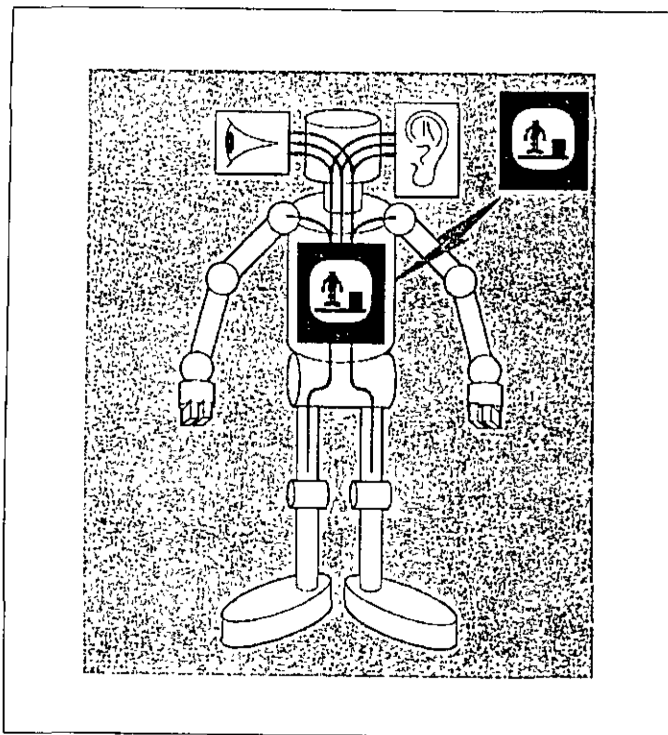


Figure 2. The early architecture as represented in an artificial person, or android (following the early Asimov novels, we depict the brains as being in the robot's belly)
Most of the computing was centralized, with the main processing concerned with the reconstruction of the world and with deliberative reasoning such as planning

The Paradigm Shift

In 1986, Rod Brooks published a landmark paper on the *subsumption architecture*, which heralded a fundamentally different approach to getting robots to come alive. Brooks, a veteran of the computer vision community and its attendant frustrations, began thinking about how animals seemed to bring fast, specific behaviors to bear to survive in the world. Brooks was not to the first to use the findings of ethological research (for example, Arbib [1981]), but he was the

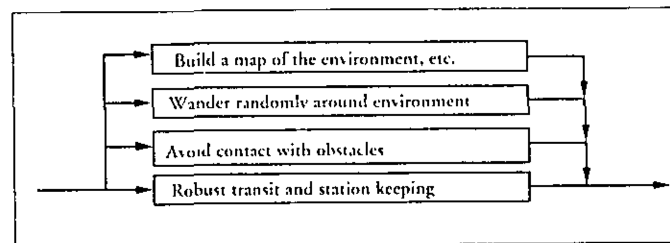


Figure 3. The new control paradigm
Behaviors are run in parallel with arbitration at the robot actuators.

first to bring those findings to fruition with real mobile robots. He developed the subsumption language that would allow one to model something analogous to animal behaviors in tight sense-act loops using asynchronous finite-state machines. The first set of behaviors for a robot might simply be used to avoid letting anything come too close by running a little ways away but otherwise standing still. Another higher-level behavior might be to move in a given direction. This behavior would dominate the obstacle-avoidance behavior by suppressing its output to the actuators unless an object got too close. The higher levels subsumed the lower levels, hence the name of the architecture. The result was a robot that could wander around a laboratory for hours without colliding into objects or moving people, using only simple sonar or infrared sensors.

In short order, Brooks and his students were developing highly mobile robots—mobots, both wheeled and legged—that could chase moving objects or people; run to, or hide from, light or sound; or negotiate a cluttered landscape such as might be found in a rugged outdoor environment. In addition, they grew in sophistication. There was HERBERT, a soda-can-collecting robot (Connell 1990); GENGHIS, a robot that learned to walk (Maes and Brooks 1990; Brooks 1989), TOTO, a hallway-navigating robot (Mataric 1992); and POLLY, a tour-guide robot (Horswill 1993).

The AI community was alternately fascinated and frustrated by these "creatures." They were frustrated because they did not want to give up the expected power of formal systems or computer vision, but they did want the robots to survive and perform at the natural pace of humans and their environments, much as Brooks's mobots were demonstrating. However, there was hope because essentially Brooks was advocating not so much insect intelligence but a complete rearrangement of the SPA cycle, as shown in figure 3.

The idea was to build up capability in the robot through behaviors that ran in parallel, accomplishing possibly competing goals. These behaviors could execute well within the cycle times of most natural environments, yet with a reasonably simple arbitration among goals based on priorities, useful tasks could be

accomplished. The world model was now distributed among the behaviors, with only the relevant part of the model being processed for each behavior. Simple plan generation, mostly for path planning, and the compilation of the resulting network of actions were done before run time. Returning to our previous example, the robot under the new control paradigm would detect the pothole with one of its low-level behaviors, merge an avoidance vector with the current direction vector, and smoothly veer from the danger on its way to its goal.

This paradigm shift in terms of our android architecture is shown in figure 4. It is characterized by simpler representations computed closer to the sensing apparatus and tailored to the particular tasks at hand. This change in representation resulted in smaller computational requirements. Coupled with improved technology in computers and batteries, on-board computing thus became practical, and the robots became smaller. This shift to reactive kinds of control also marked the first steps toward using distributed computing. In particular, robot manufacturers were marketing sensor systems with self-contained processing and standard communication buses. Thus, the practicality of plug-and-play behaviors increased the number of researchers who could develop reactive robots.

Intelligent Robots: The New Wave

Following Brooks's success, several research groups, some new and some not so new, began readvocating or exploiting this shift to increased emphasis on sensing and acting and reduced emphasis on planning. Arbib and Arkin (Arkin 1987) pioneered the new paradigm from a cognitive science perspective within the robotics community, calling it *action-oriented perception*. This approach has proven useful for manipulation (Iberall and Lyons 1984), navigation (Arkin et al. 1987), and sensor fusion (Murphy 1996). Efforts by Khatib (1985) and Payton (1986) illustrate the reactive movement originating in engineering fields.

Formalizing the Reactions

For the AI formalists, the work of Stan Rosenschein and Leslie Kaelbling (1986) stands out. They proved that if one could represent robot goals of state achievement and maintenance in the form of an electronic circuit, a consistent semantics could be maintained between the memory states of the circuit and the states of the world represented by these states. The REX language compiled propositional goal states and robot actions for achieving these states into circuits (actually C-based simulations of circuits) that executed in bounded time and usually on the order of 10 hertz. Thus, the programmer was allowed to use a propositional language to specify desired goals, yet the robot was able to execute the required resulting actions in real time. To deal with multiple goals that would

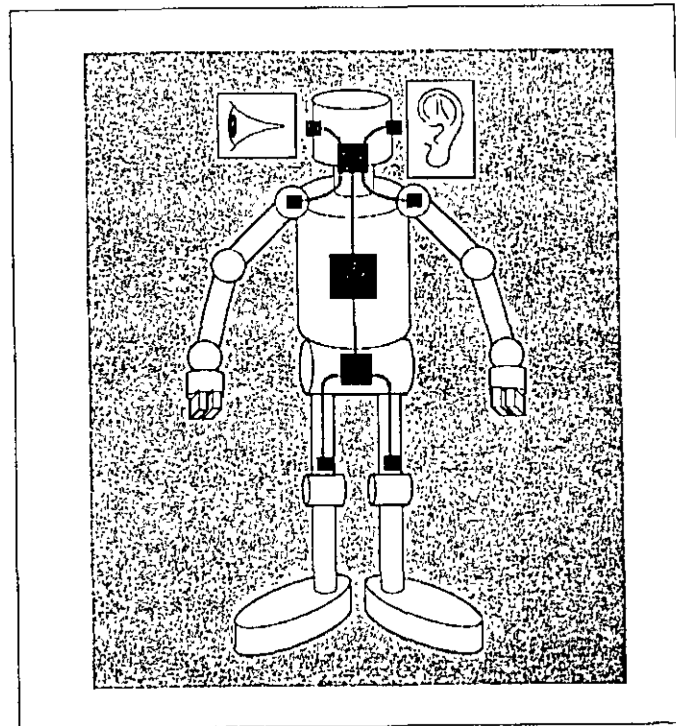


Figure 4. The architecture of the reactive android.
Most of the computing is distributed in sense-act behaviors running close to the robot's sensors. These robots were fast and reactive, but few were considered useful for normal human endeavors.

contend for the robot's sensors or actuators (the REX compiler would flag conflicting commands to the robot), REX programs typically included a scheme to arbitrate among active circuits.

Animate Vision

What of the use of cameras for robots? Human and animal vision are the most

powerful perception systems. However, we have seen how the processing of the low-level data alone, much less the addition of rapid control of a pan-tilt head, seemed to have little chance of fitting into the new paradigm. Fortunately, in the late 1980s, an analogous paradigm shift was taking place in the way researchers were approaching vision for agents.

Again using ethology, several researchers began using animal visual behaviors as models for computational counterparts. For example, a frog primarily used its motion detection to catching flying food. Other animals keyed on specific aspects of the color spectrum for certain tasks. Indeed, psychophysical studies showed the human visual system to be, not surprisingly, even more adept. The human retina is arranged in such a manner as to have a higher concentration of receptors in the center and decreasing numbers radiating outward. Thus, humans don't process square arrays of data, for example, 512 x 512 x 8 bits, in a time step; rather, they use lower-resolution peripheral vision to watch for indications of motion or looming objects while they concentrate the higher-resolution center of the retina—the *fovea*—to reason about a specific object or part of an object in great detail. Humans don't take in everything at once in all its color and motion dimensions; instead, they concentrate on a narrow portion of their visual field.

Moreover, humans move this portion rapidly about the environment in patterns dictated by the task at hand and the last time step of visual information that was produced. For example, when looking at a picture of a group of people, if one is asked what the ages of the people are, one's eyes move in a pattern that concentrates on the faces of the people, with a few scans to determine the height of the people. To determine where a cup is in the picture, the eyes dart quickly about the picture for a table, then move to the objects on the table. Only when told that they must remember as many objects in the picture as possible will the eye-scanning machinery move in a pattern resembling the scan of a full image as found in the classic algorithms of computer vision (Ballard 1991).

Now computer vision paradigms were being recast into small, quick behaviors that not only dealt with a given field of view more efficiently but also where to next point the pan-tilt head of the camera. These well-defined, compact routines, such as tracking a given color or attending to peripheral motion, were much like the behaviors being developed by the nouveau planning community and could now be incorporated as another part of the paradigm shift in programming robots.

A New Kind of Mapping

Just as AI had much to learn from the attempts to make robots intelligent, so did the robotics community stand to gain from the same endeavor. A good example was the use of maps for robot navigation. Early maps in the robotics community were geometric in nature, often as grids with each cell representing some amount of space in the real world. The grid had a single-coordinate system in which elements were represented. These maps became sophisticated at representing the spatial structure of the world (Moravec and Elfes 1985). It also was easy to do path planning and obstacle avoidance with geometric maps (for example, Lozano-Perez and Wesley [1979] and Brooks [1982]). However, geometric maps, as a part of the traditional world model of the robot, can require vast amounts of memory for large areas; in addition, the robot must know precisely where it is so that it can reason from the map or add to it. Just as with computer vision, trying to maintain accurate geometric maps was computationally intensive and extremely difficult in real-world situations.

The solution, in keeping with the paradigm shift in vision, was the use of topological maps (Kuipers and Byun 1987; Brooks 1985). Patterned after how humans represent space, topological maps represent the world as a graph of places connected by arcs, thus using no metric or geometric information, only the notions of proximity and order. With a topological map, the robot navigates locally from place to place, minimizing movement errors. Moreover, topological maps are clearly much more compact in their representation of space.

This notion was rapidly adopted by the AI and robotics communities. Kuipers and Byun (1991) continued their work on topological maps, producing a representation of space called the *spatial semantic hierarchy*. Another implementation of topological maps, by Kortenkamp and Weymouth (1994), used both sonar and vision to determine places in a topological representation. The first part of this book introduces several other topological-based map representations and also some initial attempts at integrating topological and grid-based map representations.

Return to Planning

Although the movement away from general representations was considered healthy, the resulting degree of specialization was viewed with some alarm. As Chuck Thorpe of Carnegie Mellon University once remarked about Brooks's robots: "I wouldn't want one to be my chauffeur." In point of fact, many researchers exploring the new paradigm had no intention of throwing out the classic planning baby with the bath water. However, it was clear that planning in both its form and its function had to be rethought.

Two researchers involved in this rethinking by looking at the psychophysical aspects of human activity were Phil Agre and David Chapman (1987). Their research pointed to evidence that humans somehow put together plans for action based on the set of routine behaviors they can carry out. Moreover, logical decomposition planning is rarely invoked in the course of human affairs, and when it is, it serves primarily as a guide to the gene[illegible] rection in

which one should head rather than a production of rigid sets of action.

During the late 1980s and early 1990s, several approaches along these lines were being pursued at once for intelligent robots. There were attempts to expand on the mobot approach (Maes 1990); others went further in the direction of enumerating all possible actions using planning prior to run time (for example, Kaelbling [1988] and Schoppers [1987]). Still others tried a combination of these approaches (for example, Bonasso [1991]).

One of the most important of these efforts was the work by Jim Firby (1989) on reactive action packages (RAPs). In his dissertation, Firby described a three-layered architecture with classic planning at the top, a reactive layer of behaviors at the bottom, and a middle layer with the goals of the resulting plan executed as dynamic sequences of these behaviors (that is, RAPs). When this framework was significantly expanded (Bonasso et al. 1995; Gat 1992), it became possible to program a large variety of robots—or any group of computer-controlled machines for that matter—to carry out a variety of tasks over long duration in the vicinity of, and in concert with, human counterparts. Erann Gat's chapter in this book on the three-layered approach explains why it has become a popular approach for the design and implementation of intelligent robots

Figure 5 shows the current approach to intelligent robotics in our android form. We might call this approach *P-SA*; that is, the robot plans based on initial conditions and common knowledge (P) and then executes this plan using sense-act (SA) behaviors, replanning only when the reactive behaviors run out of routine solutions. In this architecture, simple representations are tailored to specific tasks. Layered software allows behaviors such as obstacle avoidance to coordinate smoothly with behaviors such as path following. A new level of routines—cached plans—execute between the reactive behaviors and the central brain, and planning and other deliberate reasoning guide the procedures and behaviors in accomplishing the primary task and interacting with humans In addition, there have been some remarkable advances in hardware Plug-and-play subsystems that combine sensors and effectors are much more common.

The AAAI Mobile Robot Competition and Exhibition

By 1992, AAAI had decided that there was enough interest in AI as applied to mobile robots that it held a Robot Competition and Exhibition at its Tenth National Conference on Artificial Intelligence (AAAI-92) in San Jose. Ten teams competed in three events (Dean and Bonasso 1993) The events consisted of finding tall poles in a large arena while avoiding stationary obstacles

The disadvantages of the SPA approach were fairly well acknowledged at the first competition, but we still saw a mixed bag of generalists and specialists, and the advantages and disadvantages were illustrated in a fairly dramatic way. One

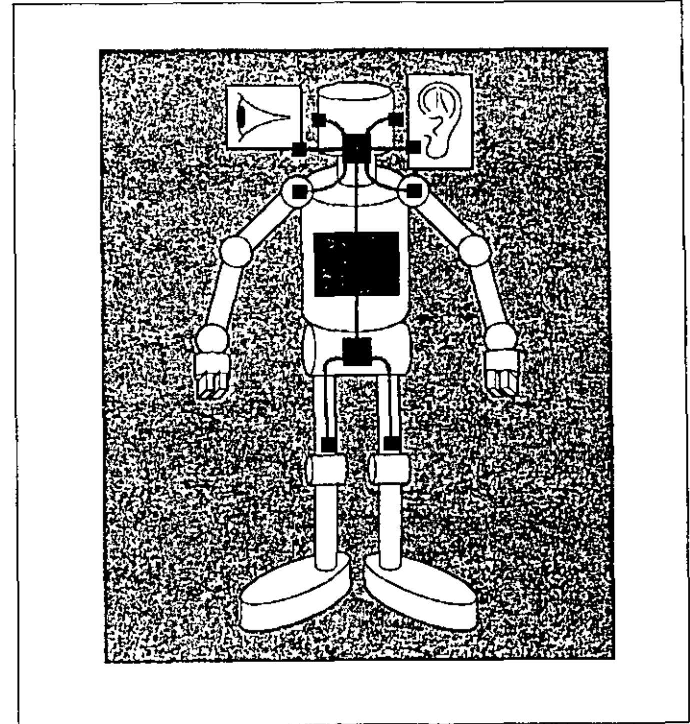


Figure 5 Today's hybrid approach to intelligent robotics. Reactive behaviors close to the sensors are orchestrated by cached plans operating between the behaviors and the central processor Deliberative reasoning takes place at the central brain and serves to guide the functioning of the rest of the system

robot builder took the P-SA idea to the extreme. A robot called SCARECROW, built by Dave Miller and his son Jacob (Yeaple 1992), had no computer and instead utilized simple electromechanical feedback loops to generate relatively simple but remarkably effective behavior given the task at hand—locating and identifying tall poles in a cluttered arena. SCARECROW did not use a systematic method for exploring its environment; instead, it performed what was essentially a random walk. SCARECROW offered a dramatic illustration of a theoretical result—a short random walk in an undirected graph will visit every location in the graph with high probability What SCARECROW lacked in intelligence (like its

namesake in the *Wizard of Oz*, SCARECROW had no brains), it made up for in raw speed; as a result, it was still a contender going into the final event and finished fourth overall, beating many more traditionally intelligent robots.

Nonetheless, the winner of the 1992 competition was a robot strong on reactive behaviors (in particular, a fast sonar activation and response scheme) and lean on deliberative algorithms. David Kortenkamp's chapter in this book recounts the design of this winning robot, CARMEL.

The event was an unqualified success and led to a second competition and exhibition at AAAI-93 in Washington, D.C. The tasks were more complicated, including pushing boxes into a pattern, escaping from an office with real furniture, and navigating in an office building (Konolige 1994, Nourbakhsh et al. 1993). This step up in complexity proved to be difficult for the robots, and the next competition, at AAAI-94 in Seattle, again focused on office navigation as well as on trash cleanup (Simmons 1995). The following year, the competition moved to the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95) in Montreal, Canada, with a slightly harder set of tasks built on the previous competition (Hinkle et al. 1996). The tasks involved giving the robots navigation directions in an office building and picking up and sorting trash. At the AAAI-96 in Portland, Oregon competition, (Kortenkamp et al. 1997), there was another incremental increase in the difficulty of the tasks, including having robots detect the occupancy of a room and having robots catch moving balls. By AAAI-97, held in Providence, Rhode Island, the state of the art was sufficiently advanced for the organizers to stage four competitions at once. "Find Life on Mars" combined the open space navigation of the first competition with vision-based search and small article manipulation. "Where's the Remote" required the recognition and fetching of everyday objects in a home, without engineering the environment. The "Home Vacuum" contest involved keeping several rooms of a house cleaned, while not disturbing the human occupants. In a break from the technical direction of all previous competitions, "Hors d'Oeuvres Anyone?" required the robots to move about a crowded reception area while serving hors d'oeuvres and being entertaining as well—an event that was, by far, the most popular of the contests.

The robot competitions provide a good yardstick with which to measure progress in the field (although certainly not the only measure). The first competition involved finding tall poles rising above small static obstacles. Teams could mark the poles in any way they wanted. The object was simply to visit the poles. In the most recent competition, one task was to use a sparse map to visit two conference rooms in an office building and determine if they were occupied. Along the way, people could be walking in the corridors, and hallways and doorways could be blocked. The robots also had to estimate how long it would take them to finish the task. The second task involved picking up tennis balls, as well as a moving "squiggle" ball, and placing them in a pen. This is a significant amount of progress in five years, and many of the case studies in this book d--

ument the different robots that have performed exceptionally in the competition as well as robots that have performed well in a separate competition for outdoor mobile robots patterned after the ARPA Unmanned Ground Vehicle Initiative.

About This Book

This book consists of thirteen case studies of AI techniques applied to mobile robots. These case studies were largely taken from competitors in the AAAI robot competitions, allowing the reader to compare and contrast the different approaches to tasks that require sensing, acting, and planning. Here, we give a brief overview of each case study and explain why it has been included in this volume. The case studies in this book are divided into three parts, navigation and mapping, vision for mobile robots, and mobile robot architectures.

Navigation and Mapping

The chapters in this section concentrate on the issue of getting around in the world. Substantial progress has been made in this area in the last five years, and there are many excellent solutions to the problems of obstacle avoidance, pose (position and orientation) estimation, and global path planning. The following chapters offer tried-and-true methods for mobile robot navigation and mapping:

Sebastian Thrun and his colleagues Arno Bucken, Wolfram Burgard, Dieter Fox, Thorsten Fröhlinghaus, Daniel Hennig, Thomas Hofmann, Micheal Krell, and Timo Schmidt describe a mobile robot called RHINO in their chapter on map learning and high-speed navigation. They present an innovative technique for constructing topological maps from grid maps. They also give algorithms for performing localization within maps and doing path planning with both grid maps and topological maps. Finally, they discuss vision-based approaches to building maps.

David Kortenkamp and his colleagues Marcus Huber, Charles Cohen, Ulrich Raschke, Frank Koss, and Clare Congdon describe CARMEL, the robot that won the first AAAI Mobile Robot Competition and Exhibition in 1992. This robot integrated a high-speed obstacle-avoidance technique called *vector-field histogramming* (VFH) with a simple, yet effective, object-recognition technique. Algorithms for VFH, path planning, vision-based localization, and object recognition are described in detail.

Illah Nourbakhsh describes a robot called DERVISH that won the 1994 AAAI Mobile Robot Competition and Exhibition in Seattle, Washington. It was the first mobile robot to use probabilistic state progression to navigate reliably with

little explicit map information. DERVISH also used a unique sonar configuration to perform robust obstacle avoidance.

Sven Koenig and Reid Simmons show how to use partially observable Markov decision process models (POMDPs) to build a navigation architecture that allows for localization, pose estimation, path planning, and learning. They present results using their robot XAVIER, which wanders the hallways of Carnegie Mellon University for hours at a time while it responds to requests made over the internet.

Vision for Mobile Robots

In the second section, we begin to look closely at the specific uses of computer vision in mobile robotics. Although vision is not necessary to perform many mobile robot tasks, it is desirable for many reasons, including its long-range high resolution and its passive sensor (that is, it doesn't emit energy). Thus, it is ideal for outdoor mobile robots, and several of the chapters in this part specifically address this domain.

Ian Horswill's chapter describes the design of a robot that uses real-time vision as its sole sensing modality and performs complicated navigation tasks that included giving tours of parts of the Massachusetts Institute of Technology Artificial Intelligence Laboratory. POLLY takes a minimalist approach based on Brooks's subsumption architecture. POLLY provides proof that relatively simple algorithms and components, specialized to task requirements, can create robust systems.

Robin Murphy describes the Colorado School of Mines entries in the 1994 and 1995 Autonomous Unmanned Vehicle Systems (AUVS) International Unmanned Ground Vehicle Competitions. She presents architectural paradigms for coordination and control of perception, with a special emphasis on how sensing is interfaced with acting. She describes a simplified version of Dickmanns's road-following algorithm and explains how it was implemented on two outdoor mobile robots.

Bill Schiller and his colleagues Yu-Feng Du, Don Krantz, Craig Shankwitz, and Max Donath also look at mobile robots that operate in outdoor road-following environments. Their autonomous land experimental vehicle (ALX) uses vision for road following and sonar sensors for obstacle detection and collision avoidance. A real-time control strategy arbitrates between processes for visual perception, path tracking, obstacle detection, and collision avoidance.

Mobile Robot Architectures

A mobile robot architecture reflects the overall design principles of its designers, which have been learned from many years of trial and error. At the 1995 AAAI Spring Symposium, James Albus (1995) stated that "an architecture is a description of how a system is constructed from basic components and how those components fit together to form the whole." The case studies in this section look closely at the issues of architecture and how it influences mobile robot design and performance.

Erann Gat's chapter on three-layer architectures, discusses the approach alluded to in the previous section, Return to Planning, that has been used successfully on a variety of robots as well as to control life-support systems. Gat shows how three different groups of researchers arrived at the same architectural conclusion and gives us an example of how he used a three-layer architecture to win an event in the second national AI robot competition.

Kurt Konolige and Karen Myers describe the SAPHIRA architecture for autonomous mobile robots in their chapter. This architecture emphasizes coordination of behavior, coherence modeling, and communication with other agents. Konolige and Myers show how this architecture was used to perform a complicated task that involved attending to a human agent, taking advice about the environment using natural language, and recognizing gestures. Their robot, FLAKEY, placed second in the 1992 AAAI Mobile Robot Competition and Exhibition. Successors to FLAKEY, using the same architecture, placed second in the 1994 robot competition and first in the 1996 competition.

Jim Firby, Peter Prokopowicz, and Michael Swain present an animate agent architecture, which integrates reactive plan execution, behavioral control, and active vision within a single software framework. This framework includes reactive skills and the RAP reactive planning system. Their robot—CHIP—is one of the few discussed in this book that performs manipulation tasks. CHIP placed second at the 1995 AAAI Mobile Robot Competition in trash pickup and sorting.

Ron Arkin and Tucker Balch explore the exciting research area of cooperating mobile robots. They present two applications: an outdoor task in which teams of high mobility multipurpose wheeled vehicles (HMMWVs) must coordinate their actions and an indoor trash-collection task (for which they placed first in the 1994 AAAI competition). Their approach to multirobot coordination is centered on a reactive schema-based control architecture that has its roots in neuroscience and psychology.

Housheng Hu and Michael Brady describe a series of mobile robots that use a distributed architecture called the ***locally intelligent control agent*** (LICA). LICA encourages a modular approach to building complex control systems. The authors show how path planning and localization can be done on robots using their architecture. In particular, they give detailed algorithms for global path planning and dynamic localization using artificial beacons.

Finally, Don Brutzman describes the difficulties unique to underwater robots in his chapter about PHOENIX, an autonomous underwater vehicle (AUV). Most vision sensors are useless underwater, and unlike land vehicles, the very medium in which the autonomous underwater vehicle travels generates forces that must be reckoned with in even the simplest of behaviors. Brutzman describes a particularly successful approach to AUV development using simulations and distributed computer graphics.

The Future

Already the technical successes we have seen in the robot competitions over the years are finding their way into practical applications. Today unattended mobots fetch and carry in semistructured hospital environments. Vacuum mobots are used regularly in North American industry to clean large storage and staging spaces during off-work hours. Mobots are also used to semiautonomously explore uninhabitable venues such as Terran volcanoes and Martian landscapes. The case studies in this book are ample proof that mobots have technically evolved to a point where, today, they are poised to help humankind in broad ranging tasks from mapping the ocean floor and long-term nursing home care to planetary colonization.

For mobots to move to the next level of competence necessary to complete such tasks, however, they need a broader base of technical support. It is our hope that, from this book, AI researchers will be inspired to expend additional effort in mobile robot research. One of the editors is fond of saying that "acting and sensing are still the hardest parts." So naturally, new developments in robot perception and low-level control will always be necessary to advance the state of the art and meet the challenge of applications in difficult environments such as under water or outer space.

Mobots can benefit from all artificial intelligence disciplines, however, and, as we have previously explained, the robot architectures that support more traditional AI research are already in place. Mobots need to reason about their acts, both for feasibility and for rationality. Thus they can benefit from advances in planning and logical theories of sensing and acting. Most future robot tasks will involve working with humans. Consequently, spoken language generation and understanding must be developed for them to be effective team members. For missions of long duration—such as those involving deep sea or planetary exploration—mobots must adapt their behavior, and even their preferences over time. This requirement involves machine learning at all levels of competency.

The time has thus come, and the technology is here, for artificial intelligence and robotics to more closely join forces in improving the quality of life on earth and in establishing new civilizations in the cosmos.

Acknowledgements

The authors wish to acknowledge Tom Dean's insights into the evolving architectural issues of the AAAI robot competitions and his android graphical depictions. These first appeared in an invited talk for AAAI 96 entitled "A Retrospective of the AAAI Robot Competitions."

PART ONE

Mapping and Navigation

This is a book about mobile robots. While it is possible for a robot to be mobile and not do mapping and navigation, sophisticated tasks require that a mobile robot build maps and use them to move around. Levitt and Lawton (1990) posit three basic questions that define mobile robot mapping and navigation:

- Where am I?
- How do I get to other places from here?
- Where are other places relative to me?

Each of the case studies in part one of this book address one or more of these questions. Each uses a different approach to representing and using spatial information. As such, they span the spectrum of options for mapping and navigation.

On one side of the spectrum are purely metric maps. In these representations, the robot's environment is defined by a single, global coordinate system in which all mapping and navigation takes place. Typically, the map is a grid with each cell of the grid representing some amount of space in the real world. These grids became quite sophisticated at representing the spatial structure of the world (see, for example, Moravec and Elfes [1985]). The case study of CARMEL by Kortenkamp and his colleagues describes a mobile robot that uses a grid-based approach to mapping and navigation. These approaches typically work well in bounded environments, with little consistent structure and where the robot has opportunities to realign itself with the global coordinate system using external markers.

On the other side of the spectrum are qualitative maps, in which the robot's environment is represented as places and connections between places. Indeed, the idea of a map that contains no metric or geometric information, but only the notions of proximity and order, is enticing because such an approach eliminates the inevitable problems of dealing with movement uncertainty in mobile robots. Movement errors do not accumulate globally in qualitative maps as they do in maps with a global coordinate system since the robot only navigates locally, between places. Qualitative maps can also be more compact in th[illegible] representation of space, in that they represent only interesting places and [illegible] entire

environment Qualitative maps (also referred to as topological maps) have become increasingly popular in mobile robotics (see, for example, Brooks 1985; Kuipers and Byun 1991; and Kortenkamp and Weymouth 1994). The case studies by Nourbakhsh and by Koenig and Simmons describe the current state-of-the-art in qualitative mapping These techniques work well in structured environments (i.e., office buildings) where there are distinctive places that are goals for the robot

There have been efforts to combine metric and qualitative maps so that the strengths of both representations can be used (Asada et al 1988; Kuipers and Levitt, 1988). The first case study in this part, by Thrun and his colleagues, gives an overview of both metric and topological mapping and describes their approach to integrating these two representations.

David Kortenkamp

CHAPTER ONE

Map Learning and High-Speed Navigation in RHINO

Sebastian Thrun, Arno Bücken, Wolfram Burgard, Dieter Fox, Thorsten Fröhlinghaus, Daniel Hennig, Thomas Hofmann, Michael Krell, and Timo Schmidt

Building autonomous mobile robots has been a primary goal of robotics and artificial intelligence. This chapter surveys some of the best methods for indoor mobile robot navigation that have been developed in our lab over the past few years. The central objective of our research is to construct reliable mobile robots, with a special emphasis on autonomy, learning, and human interaction.

In the last few years, we have built a mobile robot system, RHINO, which is capable of exploring and navigating in unknown indoor office environments with a speed of approximately 90 centimeters per seconds. While doing so, it can learn metric and topological maps and, based on these, perform all kinds of missions. In addition, RHINO is capable of locating and retrieving objects, delivering them to specific locations or dumping them into trash bins without human intervention, and giving tours to visitors.

The purpose of this chapter is to present the key ideas and algorithms underlying our research in a coherent and accessible form in order to share our experiences and, if possible, provide some guidance for building autonomous mobile robots The following list summarizes the primary software design principles underlying our approach.

- Distributed and decentralized processing. Control is distributed and decentralized. Several on-board and off-board machines are dedicated to several subproblems of modeling and control. All communication between modules is asynchronous. There is no central clock, and no central process controls all other processes.
- Any-time algorithms. Any-time algorithms are able to make decisions regardless of the time spent for computation (Dean and Boddy 1988). Whenever possible, any-time algorithms are employed to ensure that the robot operates in real-time

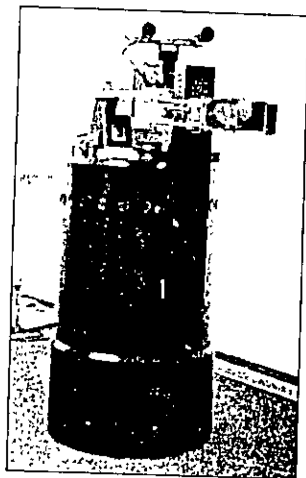


Figure 1 The robot RHINO

- Hybrid architecture. Fast, reactive mechanisms are integrated with computationally intense, deliberative modules.
- Models. Models, such as the two-dimensional maps described below, are used at all levels of the architecture. Whenever possible, models are learned from data.
- Learning. Machine learning algorithms are employed to increase the flexibility and the robustness of the system. Thus far, learning has proven most useful close to the sensory side of the system, where algorithms such as artificial neural networks interpret the robot's sensors.
- Modularity. The software is modular. A plug-and-play architecture allows us to quickly reconfigure the system, depending on the particular configuration and application.
- Sensor fusion. Different sensors have different perceptual characteristics. To maximize the robustness of the approach, most of the techniques described here rely on more than just a single type of sensor.

These principles are important, since robots and their environments are complex physical systems. Robot environments are dynamic and inherently unpredictable, and robot hardware often fails or malfunctions, as do our computers and our communication networks. To control a robot reliably, the software must react adequately and timely to sudden changes, failures, delays, or other unforeseen events—which requires special care when designing robot control software.

This chapter focuses exclusively on robot navigation. In particular, it de-

scribes our current best approaches for mapping indoor environments, and for high-speed exploration and navigation in dynamic environments. It is organized into sections on mapping, localization, and navigation.

The mapping section describes our method for constructing two-dimensional occupancy grids using sonar sensors and the cameras. Artificial neural networks are used to interpret sonar measurements. On top of the grid-based maps, more compact topological maps are constructed that facilitate fast planning.

Localization is the problem of aligning the robot's coordinate system with the global world coordinates. The localization section describes algorithms for self-localization and position tracking.

Approaches to global path planning, exploration, and reactive collision avoidance are described in the section on navigation.

Most of the software has been developed using a B21 mobile robot manufactured by Real World Interface Inc. The robot, shown in figure 1, is a synchro-drive robot equipped with a stereo camera system and 24 ultrasonic transducers (in short, sonars). Its maximum velocity is 90 centimeters per second. Our robot is equipped with two on-board 486 personal computers that are connected via a radio Ethernet link to several SUN Sparc stations. Currently some of the processing is done off-board. For robots equipped with Pentium computers, the entire software is run locally on the robot (except for the stereo vision system). A second radio link communicates video images to a Datacube, a special-purpose machine for processing images in real-time. The robot is also equipped with active infrared proximity sensors and tactile sensors, which are currently being incorporated into our map-building and collision-avoidance routines.

Mapping

Mapping refers to the process of constructing a model of the environment based on sensor measurements. This section describes an approach that integrates grid-based and topological representations. Grid-based approaches, such as those proposed by Moravec and Elfes (1988) and Borenstein and Koren (1991) and many others, represent environments by evenly-spaced grids. Each grid cell contains a value which indicates the presence or absence of an obstacle in the corresponding region of the environment. Topological approaches, such as those described in Engelson and McDermott (1992), Kortenkamp and Weymouth (1994), Kuipers and Byun (1990), Mataric (1994), Pierce and Kuipers (1994), and Torrance (1994), represent robot environments by graphs. Nodes in such graphs correspond to distinct situations, places, or landmarks (such as doorways). They are connected by arcs if there exists a direct path between them.

As argued in more detail elsewhere (Thrun [forthcoming]), both grid-based and topological representations exhibit orthogonal strengths and weaknesses:

Grid-based maps are considerably easier to learn, partially because they facilitate accurate localization, partially because they are easy to maintain. Topological maps, on the other hand, are more compact and thus facilitate fast planning.

Grid-Based Maps

The metric maps considered here are two-dimensional, discrete occupancy grids, as originally proposed by Elfes (1987) and Moravec (1988) and since implemented successfully in various systems. Each grid-cell <x, y> in the map has an occupancy value attached—denoted by $Prob(occ_{x,y})$—which measures the robot's subjective belief whether or not its center can be moved to the center of that cell (the occupancy map models the configuration space of the robot; see for example, Latombe 1991). This section describes the two major steps in building grid-based maps (see also Thrun 1993): sensor interpretation and integration. Examples of metric maps are shown in various places in this chapter.

Sonar Sensor Interpretation

Sonar sensors measure approximate echo distances to nearby obstacles, along with noise. To build metric maps, sensor reading must be "translated" into occupancy values $Prob(occ_{x,y})$ for each grid cell <x, y>. The idea here is to train an artificial neural network using Back-Propagation (Rumelhart, Hinton, and Williams 1986) to map sonar measurements to occupancy values (Thrun 1993; van Dam, Krose, and Groen 1996). The input to the network consists of the four sensor readings closest to <x, y>, along with two values that encode <x, y> in polar coordinates relative to the robot (angle to the first of the four sensors, and distance). The output target for the network is 1, if <x, y> is occupied, and 0 otherwise. Training examples can be obtained by operating a robot in a known environment and recording and labeling its sensor readings, note that each sonar scan can be used to construct many training examples for different *x*-*y* coordinates. In our implementation, training examples are generated with a mobile robot simulator.

Once trained, the network generates values that can be interpreted as probability for occupancy. Figure 2 shows three examples of sonar scans (top row, bird's eye view) along with their neural network interpretation (bottom row). The darker a value in the circular region around the robot, the larger the occupancy value computed by the network. Figures 2a and b show situations in a corridor. Here the network predicts the walls correctly. Notice the interpretation of the erroneous long reading in the left side of figure 2a and the erroneous short reading in 2b. For the area covered by those readings, the network outputs roughly 0.5, which indicates its uncertainty. Figure 2c shows a different situation in which the interpretation of the sensor values is less straightforward. This example illustrates that the network interprets sensors in the context of neighboring sensors. Long readings are interpreted as free space only if the neighboring sensors agree. Otherwise, the network returns values close to 0.5, which again indicates uncertainty. Situations such as the one shown in figure 2c—that defy simple interpretation—are typical for cluttered indoor environments.

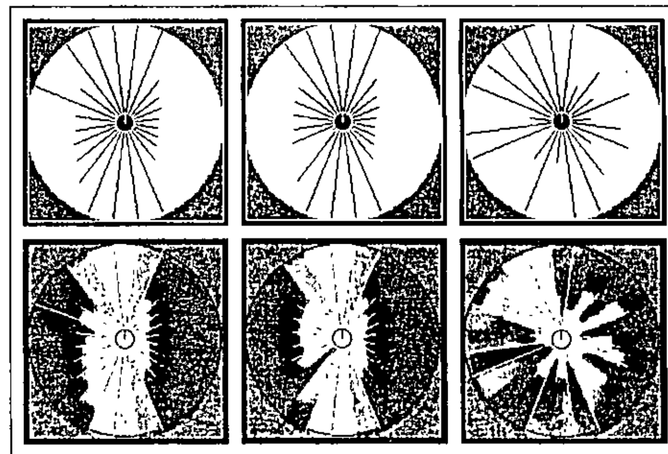


Figure 2 Sonar sensor interpretation.
Three example sonar scans (top row) and local occupancy maps (bottom row), generated by the neural network. Bright regions indicate free space, and dark regions indicate walls and obstacles (enlarged by a robot radius).

Training a neural network to interpret sonar sensors has two key advantages over hand-crafted approaches to sensor interpretation. First, since neural networks are trained based on examples, they can easily be adapted to new circumstances. For example, the walls in the competition ring of the 1994 AAAI robot competition (Simmons 1995) were much smoother than the walls in the building in which the software was originally developed. Even though time was short, the neural network could quickly be retrained to accommodate this new situation. Secondly, multiple sensor readings are interpreted simultaneously. Most existing approaches interpret each sensor reading individually, one-by-one, which can be problematic in practice. For example, the reflection properties of most surfaces depend strongly on the angle of the surface to the sonar beam, which can only be detected by interpreting multiple sonar sensors simultaneously.

Stereo Camera Interpretation

A second source of occupancy information is the stereo camera system, which

provides pairs of images recorded simultaneously from different spatial viewpoints. Stereo images are transmitted via a radio link to a Datacube, a special-purpose computer for image processing. Like the human visual apparatus, stereo images can be used to compute depth information (i.e., proximity). Put shortly, our approach (Fröhlinghaus and Buhmann 1996a, Fröhlinghaus and Buhmann 1996b) analyzes stereo images for co-occurrences of vertical edges. By analyzing the disparity of vertical edges found in both images, the proximity of obstacles projects the obstacle onto a two-dimensional occupancy grid.

Figure 3 illustrates the stereo image analysis. Images are first preprocessed (separately for both channels) using a set of three horizontal Gabor filters (Sanger 1988). Gabor filters, which are similar to local Fourier transforms, basically band-filter images to obtain two local coefficients: The amplitude of a certain filter-specific frequency, and its phase. Vertical edges characterized by a sharp horizontal brightness difference yield large amplitudes; their precise location in the image is determined with subpixel accuracy based on the phase. Figure 3a depicts one of the images, and figure 3b shows the vertical edges extracted from this image using Gabor filters. To establish correspondence between the left and the right image, both amplitude and phase information is used to identify the projection of the same real-world edge in both images. Once such a vertical edge has been identified in both images, its disparity (spatial shift between both images) is used to estimate its proximity to the cameras (depth). A simple geometric projection yields the *x-y* location of the edge relative to the robot. Figure 3c shows the projected location (bird's eye view) of the edges found in figure 3b; notice that the information concerning the height of an edge is lost when projecting the edge information onto the two-dimensional plane. To construct the final occupancy grid (like the one shown in figure 3d), edges are enlarged by a robot radius, and the area between the edge projections and the robot is marked as free space. The last step relies on the assumption that vertical edges correspond to corners of obstacles, since occupancy maps represent configuration space, these edges are increased by a robot radius. To compensate for some of the uncertainty in depth estimation, corners of obstacles are smoothed with a Gaussian, and the free-space between obstacles and the robot is smoothed with a logistic function ($f(x) = (1 + e^{-x})^{-1}$). Notice in our implementation, the Datacube preprocesses images at about 4 Hertz. The entire generation of occupancy maps, most of which is done on a SUN Sparc station, requires currently less than a second.

When the original image is compared (figure 3a) with the resulting map (figure 3b), the door posts and two of the posters can clearly be identified in the final map (large dark areas in the map). The region close to the white poster does not contain clear vertical edges; thus the robot fails to identify the white wall. This example illustrates that strictly speaking, occupancy maps derived from stereo vision contain only edges of obstacles—large unstructured obstacles such as walls are "invisible" and hence will not be mapped. Consequently, stereo vision alone would not be sufficient for building accurate maps. On the other hand, stereo vision gives more accurate obstacle information than sonar sensors do, due to the higher resolution of cameras. It also frequently detects obstacles that are "invisible" to sonar sensors, such as objects that absorb sound. As we will demonstrate later, stereo vision is well-suited to augment sonar information.

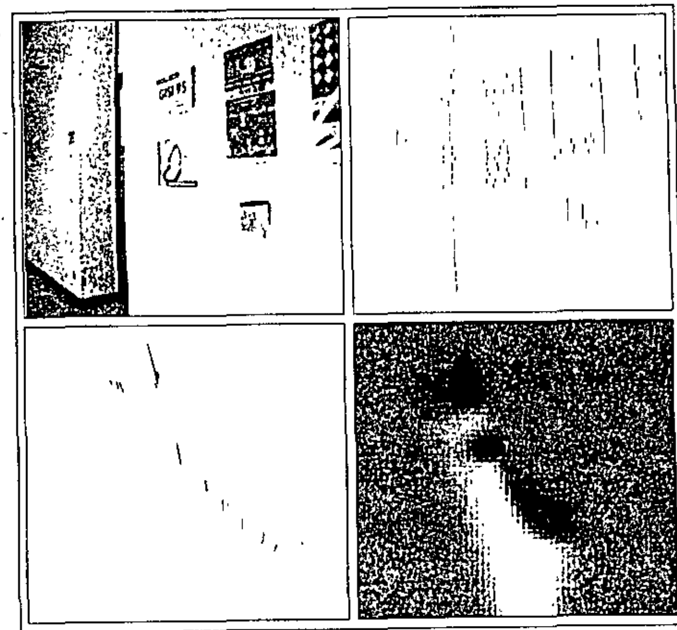


Figure 3. Estimation of occupancy maps using stereo vision.
Top left(a): Left camera image. Top right (b). Sparse disparity map.
Bottom left (c): Two-dimensional edge projections.
Bottom right (d). Local occupancy map.

Integration Over Time

Sensor interpretations must be integrated over time to yield a single, consistent map. To do so, it is convenient to interpret the interpretation of the *t*-th sensor reading (denoted by $s^{(t)}$) as the probability that a grid cell $\langle x, y \rangle$ is occupied, conditioned on the sensor reading $s^{(t)}$.

$Prob(occ_{x,y} \mid s^{(t)})$

A map is obtained by integrating these probabilities for all available sensor

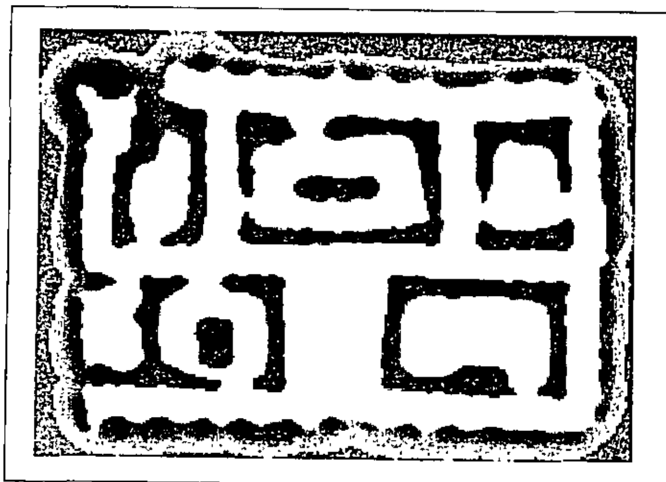


Figure 4. Grid-based map, constructed at the 1994 AAAI autonomous mobile robot competition with the techniques described here. (Height is 23.1 meters. Width is 32.2 meters.)

readings, denoted by $s^{(1)}, s^{(2)}, \ldots, s^{(T)}$. In other words, the desired occupancy value for each grid cell $<x, y>$ can be written as the probability

$Prob(occ_{x,y} \mid s^{(1)}, s^{(2)}, \ldots, s^{(T)})$,

which is conditioned on all sensor readings. A straightforward approach to estimating this quantity is to apply Bayes' rule. To do so, one has to assume independence of the noise in different readings. More specifically, given the true occupancy of a grid cell $<x, y>$, the conditional probability $Prob(s^{(t)}|occ_{x,y})$ must be assumed to be independent of $Prob(s^{(t')}|occ_{x,y})$ if $t \neq t'$. This Markov assumption (Chung 1960) is commonly made in approaches to building occupancy grids. The desired probability can now be computed as follows:

$$Prob(occ_{x,y} \mid s^{(1)}, s^{(2)}, \ldots, s^{(T)}) \tag{1}$$
$$= 1 - \left(1 + \frac{Prob(occ_{x,y})}{1 - Prob(occ_{x,y})} \prod_{\tau=1}^{T} \frac{Prob(occ_{x,y} \mid s^{(\tau)})}{1 - Prob(occ_{x,y} \mid s^{(\tau)})} \frac{1 - Prob(occ_{x,y})}{Prob(occ_{x,y})}\right)^{-1}$$

Here $Prob(occ_{x,y})$ denotes the prior probability for occupancy (which, if set to 0.5, can be omitted in this equation). The derivation of this formula is straightforward and can be found in Moravec (1988) and Pearl (1988). Notice that this formula can be used to update occupancy values incrementally

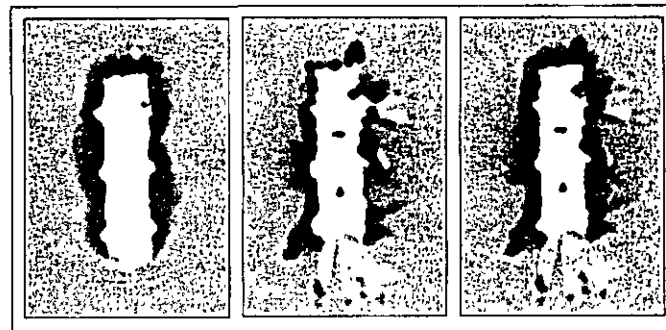


Figure 5. Maps built in a single run. Left (a): Using only sonar sensors. Center (b): Using only stereo information. Right (c): Integrating both. Notice that sonar models the world more consistently but misses the two sonar-absorbing chairs which are found using stereo vision.

An example map of a competition ring constructed at the 1994 AAAI autonomous mobile robot competition is shown in figure 4. This map has been constructed exclusively from sonar information. Figure 5 illustrates the advantage of integrating sonar and stereo vision. All three maps shown there show the same experiment. The map shown in figure 5a has been constructed based purely on sonar information. While sonar models walls well, it misses the two sound-absorbing chairs in the middle of the hallway. Stereo vision (figure 5b) detects these chairs, but fails to detect a glass door at the top of the diagram. The map in figure 5c, which is superior to both single-sensor maps since is contains the chairs and models the walls consistently, is obtained by integrating both maps. Notice that both maps were integrated by taking the maximum occupancy value at each grid cell. Taking the maximum is the most conservative way to integrate maps, since all obstacles are preserved. In principle, the maps could also have been integrated by (1); however, then the result is influenced by the relative frequency of sonar and camera measurements, which in this example makes the obstacles disappear.

Topological Maps

Topological maps represent robot environments as graphs, where nodes correspond to distinct places, and arcs represent adjacency. A key advantage of topological representations is their compactness. In our approach, topological maps are built on top of the grid-based maps. The basic idea is simple but very effective: The free-space of a grid-based map is partitioned into a small number of

regions, separated by critical lines. Critical lines correspond to narrow passages such as doorways. The partitioned map is then mapped into an isomorphic graph. The precise algorithm works as described in the following paragraphs:

Thresholding. Initially, each occupancy value in the occupancy grid is thresholded. Cells whose occupancy value is below the threshold are considered free-space (denoted by C). All other points are considered occupied (denoted by $\bar{C}$).

Voronoi Diagram. Consider an arbitrary point $<x, y> \in C$ in free-space. The basis points of $<x, y>$ are the closest point(s) $<x', y'>$ in the occupied space $\bar{C}$, i.e., all points $<x', y'> \in \bar{C}$ that minimize the Euclidean distance to x, y. We will call these points $<x', y'>$ the basis points of $<x, y>$, and the distance between $<x, y>$ and its basis points the clearance of $<x, y>$. The Voronoi diagram, which is a form of skeletonization (Latombe 1991), is the set of points in free space that have at least two different (equidistant) basis points. Figure 6a sketches the Voronoi diagram for the map shown in figure 4.

Critical Points. The key idea for partitioning the free-space is to find "critical points." Critical points $<x, y>$ are points on the Voronoi diagram that minimize clearance locally. In other words, each critical point $<x, y>$ has the following two properties: (a) it is part of the Voronoi diagram, and (b) there exists an $\varepsilon > 0$ for which the clearance of all points in an ε-neighborhood of $<x, y>$ is not smaller.

Critical Lines. Critical lines are obtained by connecting each critical point with its basis points (cf. figure 6b). Critical points have exactly two basis points (otherwise they would not be local minima of the clearance function). Critical lines partition the free space into disjoint regions (see also figure 6c).

Topological Graph. The partitioning is mapped into an isomorphic graph. Each region corresponds to a node in the topological graph, and each critical line to an arc.

Figure 6d shows an example of a topological graph. The compression is enormous: The topological graph has 67 nodes, whereas the original map contains 27,280 occupied cells. Notice that critical lines are useful for decomposing metric maps primarily for two reasons. First, when passing through a critical line, the robot is forced to move in a considerably smaller region. Hence, the loss in performance inferred by planning using the topological map (as opposed to the grid-based map) is rather small. Secondly, narrow regions are more likely blocked by obstacles (such as doors, which can be open or closed).

Localization

Localization is the process of aligning the robot's local coordinate system with the global coordinate system of the map. Localization is particularly important

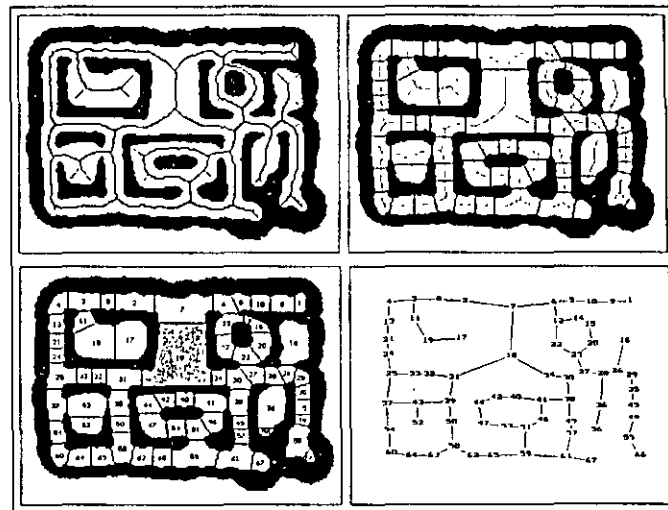


Figure 6. Extracting the topological graph from the map depicted in figure 4. Top left (a): Voronoi diagram. Top right (b): Critical points and lines. Bottom left (c): Regions. Bottom right (d): The final graph.

(and particularly difficult) for map-based approaches that learn their maps, since the accuracy of a metric map depends crucially on the alignment of the robot with its map (Feng, Borenstein, and Everett 1994; Rencken 1993). Figure 7 gives an example that illustrates the importance of localization in robot mapping. In figure 7a, the position is determined based solely on dead-reckoning. After approximately fifteen minutes of robot operation, the position error is approximately eleven and one-half meters. Obviously, the resulting map is too erroneous to be of practical use. Figure 7b is the result of applying the position tracking method described below. In fact, none of the maps presented in the previous section would have been possible without our methods for localizing the robot based on sensor input. Identifying and correcting for slippage and drift is thus a most important issue in map building.

An excellent overview of different approaches to localization can be found in Feng, Borenstein, and Everett (1994). Traditionally, localization addresses two subproblems, which are often attacked separately: position tracking and global localization.

Position Tracking. Position tracking refers to the problem of estimating the location of the robot while it is moving. Drift and slippage impose limit[illegible] the

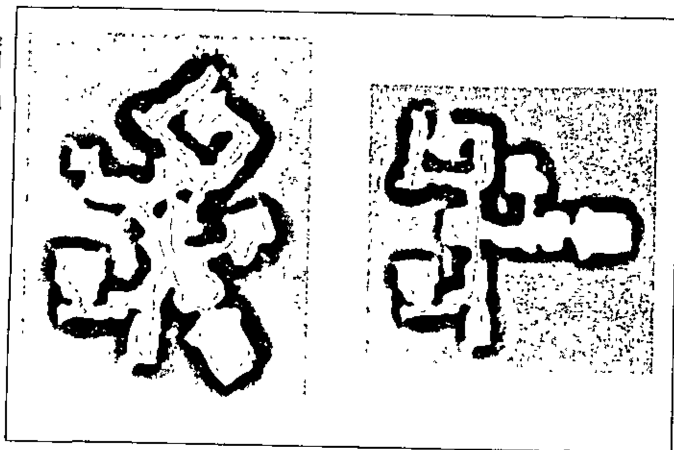


Figure 7. Map constructed without (a)(left) and with (b)(right) the position estimation mechanism described in this chapter. In (a), only the wheel encoders are used to determine the robot's position. The positional error accumulates to more than eleven meters, and the resulting map is clearly unusable. This illustrates the importance of sensor-based position estimation for map building

ability to estimate the location of the robot within its global map. As figure 7 demonstrates, even the smallest errors in the robot's odometry can have devastating effects. The problem of position tracking is particularly difficult to solve if mapping is interleaved with localization.

Global Localization. Global localization is the problem of determining the position of the robot under global uncertainty. This problem arises, for example, when a robot uses a map that has been generated in a previous run, and when it is not informed about its initial location within the map.

Global localization and position tracking are two sides of the same coin: localization under uncertainty. In our current implementation, different computational mechanisms are used for localization within a previously learned map (global localization and position tracking), and position tracking when interleaved with exploration and mapping. Each of these approaches exploits the specific properties of the two problems.

Probabilistic Model

The problem of localization is most generally described in probabilistic terms.

Let $l^{(t)}$ denote the location of the robot at time t. For mobile robots, l is usually three-dimensional (x-y location and heading direction). l is not directly accessible; i.e., the robot does not know where it is. Instead, it maintains an internal belief as to where it might be and uses its sensors periodically to update this belief. It is convenient to denote the belief as a probability density $Prob^{(t)}(l)$ over locations l. The problem of localization is then to estimate $Prob^{(t)}(l)$ so that it matches as closely as possible the true location, $l^{(t)}$. Initially, at time $t = 0$, $Prob^{(0)}(l)$ may be distributed uniformly, assuming that the robot does not know its initial location, or, alternatively, $Prob^{(0)}(l)$ may contain a single peak at $l^{(0)}$ if the robot happens to know its location. $Prob(l)$ is updated whenever the robot senses, using mostly ad hoc approaches to determine the conditional probability $Prob(s^{(t)} \mid l)$:

$$Prob^{(t)}(l) \leftarrow \alpha \, Prob^{(t-1)}(l) \cdot Prob(s^{(t)} \mid l) \tag{2}$$

Here α is a normalizer, $s^{(t)}$ is the sensor input at time t, and $Prob(s^{(t)} \mid l)$ is the probability of observing $s^{(t)}$ when at l. Strictly speaking, the update formula (2) is only valid under a conditional independence (Markov) assumption: Given the true location of the robot and the true model of the environment, subsequent sensor readings must be conditionally independent. In practice, we have found this approach to be fairly robust to various kinds of violations of this assumption (dependencies are due to nonstationary environments model errors, crude representations of the space of locations, or unmodeled robot dynamics). However, the key thing to note here is that the problem of localization requires (a) sensors (to obtain $s^{(t)}$) and (b) knowledge about $Prob(s^{(t)} \mid l)$, i.e., the probability of observing $s^{(t)}$ at l. We currently employ and integrate a variety of different sensor modalities: wheel encoders, map matching, sonar modeling, maneuverability, wall orientation, and landmarks.

Wheel Encoders. Wheel encoders measure the revolution of the robot's wheels. Based on their measurements, odometry yields an estimate of the robot's location $l^{(t)}$ which, when expressed relative to the robot's previous location, $l^{(t-1)}$, is impressively accurate. To model errors in odometry, we assume that the position at time t is distributed normally around the very location measured by odometry. The probability $Prob(s^{(t)} \mid l)$, thus, is normally distributed.

Map Matching. As described above, every sensor reading is converted into a "local" map (such as the ones shown in figures 2 and 3). The robot can localize itself by comparing the global with the local map. More specifically, the pixel-wise correlation of the local and the global map—which is a function of the robot's location—is a measure of their correspondence (Schiele and Crowley 1994). The more correlated the maps are, the more likely is the corresponding location of the robot. Thus, the probability $Prob(s^{(t)} \mid l)$ is assumed to be proportional to the cor-

relation of both maps if the robot were at l. See Thrun (forthcoming) for details.

Sonar Modeling. Another source of information for localization, which we have begun to explore more recently (Burgard et al. 1996a), is obtained using a simplistic model of sonar sensors. In essence, it is assumed that each grid cell in the global map possesses a certain probability of being detected by a sonar sensor. More specifically, our model assumes that with probability $Prob(detect_{i,x,y})$ the i-th sonar sensor detects an obstacle at $<x, y>$ (where $Prob(detect_{i,x,y})$ is a monotonic function of $Prob(occ_{x,y})$, which is 0 if $<x, y>$ does not lie in the perceptual field of the i-th sensor, see Burgard, Fox, Hennig, and Schmidt (1996a). For simplicity, we also assume that the detection probability is independent for different values of i, x, and y

Sonar sensors return the distance to the nearest obstacle. Thus, the probability that an obstacle is detected at distance d is given by the probability that one or more cells at distance d respond, times the probability that no obstacle at distance smaller than d is detected. Put mathematically, let $d(i, <x, y>)$ denote the distance of $<x, y>$ to the i-th sonar sensor. Then the probability of measuring a specific distance, say s_i, is given by

$$Prob(s_i) = \beta\Big(1 - \prod_{d(i,\langle x,y\rangle)=s_i} (1 - Prob(detect_{i,x,y}))\Big) \prod_{d(i,\langle x,y\rangle)<s_i} (1 - Prob(detect_{i,x,y}))$$

where β is a normalization factor that ensures that the probabilities for different measurements s_i sum up to 1. Here only cells that are in the primary perceptive field of the i-th sonar sensor are considered (a fifteen degree cone). Notice that the distribution of $Prob(s_i)$ bears close resemblance to a geometric distribution.

The probability of observing the entire sonar scan $s^{(t)}$ at l, $Prob(s^{(t)} \mid l)$, is the product of the individual sensor probabilities:

$$Prob(s^{(t)} \mid l) = \prod_{i=1}^{24} Prob(s_i^{(t)}) \quad (3)$$

Notice that the model of sonar sensors is extremely simplistic, partially because it considers only the main cone of sonar sensors (ignoring the side cones), and partially because it assumes independence between different grid cells, ignoring cumulative effects in the reflection of sound. However, it can be computed very efficiently, which is important if the location of the robot shall be estimated in real-time.

Maneuverability. Assuming the map is correct, the mere fact that a robot moves to a location $<x, y>$ makes it unlikely that this location is occupied. Thus, the global map can be used to derive further probabilistic constraints on the robot location. Our approach assumes that the probability of being at $<x, y>$ is proportional to the probability of this grid cell being unoccupied:

$Prob(<x, y>) = \gamma(1 - Prob(occ_{x,y}))$

where γ is an appropriate normalization factor.

Wall Orientation. A final source of information, which can be used to c... ...ect

rotational errors, is the global wall orientation (Crowley 1989, Hinkel and Knieriemen 1988). This approach rests on the restrictive assumption that walls are either parallel or orthogonal to each other, or differ by more than fifteen degrees from these canonical wall directions. In the beginning of robot operation, the global orientation of walls is estimated by searching straight line segments in consecutive sonar measurements (cf. figure 8). Once the global wall orientation has been estimated, it is used to readjust the robot's orientation based on future sonar measurements. See Thrun (forthcoming) for more details.

Landmarks. Landmarks are used in various approaches to mobile robot localization (see, for example, Betke and Gurvits 1993, Kortenkamp and Weymouth 1994, Neven and Schöner 1995 and references in Thrun 1996). We recently have begun to explore mechanisms that enable a robot to select its own landmarks, based on sonar and camera input. The key idea underlying this approach is to train artificial neural networks to recognize landmarks by minimizing the average localization error (assuming that update rule (2) is applied in localization). As a result, our robot successfully "discovered" a variety of useful visual landmarks, such as doors, wall color, ceiling lights and so on. Details of the algorithm and performance results are surveyed in Thrun (1996)

This list of sources for estimating l has been developed over the last few years. Some of these methods make strong assumptions on the correctness of the global map (such as the maneuverability method), and hence cannot be interleaved with map learning. These approaches differ significantly in computational complexity. For example, the map- matching approach, as it is currently implemented, requires extensive comparisons of maps, whereas wall orientations can be determined very efficiently

Global Localization

Global localization addresses the problem of mobile robot localization under global uncertainty. To localize the robot globally, the entire density $Prob(l)$ for arbitrary locations l is computed. In our current implementation, $Prob(l)$ is approximated using a grid representation, just like the occupancy grids described in the nest section. The orientation of the robot is represented with 1° resolution. Currently, only the wheel encoders, sonar modeling and maneuverability are used to localize the robot. To update $Prob(l)$ in real-time while the robot is in motion, only a subset of sonar readings is currently considered in global positioning, since the process of updating $Prob(l)$ is computationally expensive. From the previous list of sensor modalities, the global localization approach utilizes wheel encoders and sonar sensors, using the sonar modeling and maneuverability approach.

Figure 9a shows a path taken by the robot in the arena depicted in figure 4 (same arena, different run). As shown in that figure, twelve sonar sw... ...ere

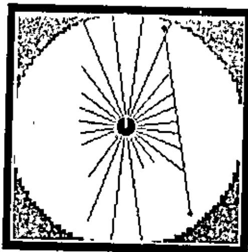


Figure 8. Wall, detected by considering five adjacent sonar measurements. Wall orientations are used to correct for dead-reckoning errors in the robot orientation.

used for the global position estimation, each of which consisted of eight sensor readings. Figure 9b shows the logarithm of the density *Prob(l)* (maximized over all orientations) after evaluating six of these twelve sensor scans. Although those six readings appear to be insufficient for uniquely localizing the robot, the density indicates that the robot is most likely in a corridor. After evaluating all twelve sensor readings (figure 9c), the position of the robot is uniquely determined. Notice that the probability of the "correct" grid cell in figure 9c is approximately 0.96, while the value of the second largest peak is less than $8 \cdot 10^{-6}$. Notice the approach deals adequately with uncertainty and ambiguities, as demonstrated by the empirical examples. The global localization approach has also given very reliable results for real-time position tracking (Burgard et al. 1996b). However, since this approach estimates the robot's location in a previously learned map, it is not applicable during exploration and map learning.

Position Tracking When Learning Maps

When learning maps, the robot knows initial location by definition (i.e., is defined to be origin of the global coordinate system). Thus, during exploration, position control seeks to compensate for short-term localization errors such as slippage and drift. The key assumption here is that the position of the robot is known except for some small error, so that instead of estimating an entire probability distribution, it suffices to keep track of the most likely location of the robot. Our current best approach for position tracking differs from the above approach to localization in two aspects (cf. Thrun 1993, Thrun [forthcoming]):

First, the approach estimates only the point *l* that maximizes *Prob(l)*, instead of the entire density. The advantage of tracking only one value is twofold: the space of localization does not have to be represented discretely (or by parametric densities), and the approach is computationally much more efficient. It comes at the obvious disadvantage that complex distributions, such as multimodal distri-

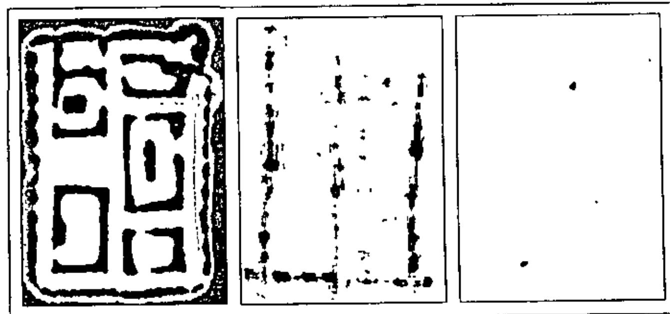


Figure 9. Initial self-localization. Left (a): Path of the robot. Center (b): Prob(l) after evaluating six sonar readings. Right (c): Prob(l) after evaluating twelve sonar readings, both plotted logarithmically.

butions, cannot be represented. Thus, once the position is lost, this approach is unable to recover. However, if the initial location is known, we almost never observed that the location was estimated inaccurately.

Second, primarily for historical reasons, our approach to position tracking relies only on wheel encoders, map matching, and wall orientation to estimate location. Position control based on wheel encoders and map matching alone works well if the robot travels through mapped terrain, but ceases to function if the robot explores and maps unknown terrain. The third mechanism, which arguably relies on a restrictive assumption concerning the nature of indoor environments, has proven extremely valuable when autonomously exploring and mapping large-scale indoor environments.

Position tracking is implemented in an any-time fashion, using gradient descent to estimate the location that maximizes *Prob(l)*. When a new sonar reading arrives, the previous gradient search is terminated and its result is incorporated into the current position estimation. In practice, we have found this approach to be fast enough to accurately track the robot position even if the robot is mapping unknown terrain with maximum velocity. Notice that all maps shown in this chapter (with the exception of the map shown in figure 7a) have been generated using this position-tracking approach.

Navigation

This section is concerned with robot motion. RHINO's navigation system consists of two modules: A global planner (Thrun 1993) and a reactive collision

avoidance module (Fox, Burgard, and Thrun 1995, Fox, Burgard, and Thrun 1997). Control is generated hierarchically: The global path planner generates minimum-cost paths to the goal(s) using the map. As a result, it communicates intermediate subgoals to the collision avoidance routine, which controls the velocity and the exact motion direction of the robot reactively, based on the most recent sensor measurements only. Both modules adjust their plans and controls continuously in response to the current situation.

Both approaches—the global path planner and the reactive collision avoidance approach—are characterized by orthogonal strengths and weaknesses: The collision avoidance approach is easily trapped in local minima, such as U-shaped obstacle configurations (Latombe 1991). However, it reacts in real-time to unforeseen obstacles such as humans and is capable of changing the motion direction while the robot is moving. The global planner, in contrast, does not suffer from the local minimum problem, since it plans globally. It alone, however, is not sufficient to control the robot, since it does not take robot dynamics into account and since learned maps are incapable of capturing moving obstacles. Thus, global planning alone would simply not avoid collisions with humans and other rapidly moving obstacles.

Path Planning with Grid-Based Maps

The idea for path planning is to let the robot always move on a minimum-cost path to the goal (or the nearest goal, if multiple goals exist): The cost for traversing a grid cell is determined by its occupancy value. The minimum-cost path is computed using a modified version of value iteration, a popular dynamic programming algorithm (Bellman 1957; Howard 1960):

Initialization. The grid cell that contains the target location is initialized with 0, all others with ∞.

$$V_{x,y} \leftarrow \begin{cases} 0, & \text{if } (x,y) \text{ target cell} \\ \infty, & \text{otherwise} \end{cases}$$

Update Loop. For all nontarget grid cells <x, y> do:

$$V_{x,y} \leftarrow \min_{\substack{\xi=-1,0,1 \\ \zeta=-1,0,1}} \left\{ V_{x+\xi,y+\zeta} + Prob\left(occ_{x+\xi,y+\zeta}\right) \right\}$$

Value iteration updates the value of all grid cells by the value of their best neighbors plus the costs of moving to this neighbor (just like A^*) (Nilsson 1982). Cost is here equivalent to the probability $Prob(occ_{x,y})$ that a grid cell <x, y> is occupied. The update rule is iterated. When the update converges, each value $V_{x,y}$ measures the cumulative cost for moving to the nearest goal. However, control can be generated at any time, long before value iteration converges.

Determine Motion Direction. To determine where to move, the robot generates a minimum-cost path to the goal. This is done by steepest descent in V, starting at the actual robot position. The steepest descent path is then postprocessed to maintain a minimum clearance to the walls and, if possible, to move parallel to walls, using the global wall orientation described in the previous section. Determining the motion direction is done in regular time intervals and is fully interleaved with updating V.

Figure 10 shows V after convergence with one and two goals, respectively, using the map shown in figure 4. The gray level indicates the cumulative costs V for moving toward the nearest goal point. Notice that every local minimum in the value function corresponds to a goal. Thus, for every point <x, y>, steepest descent in V leads to the nearest goal point.

Unfortunately, plain value iteration is too inefficient to allow the robot to navigate and learn maps in real-time. Strictly speaking, the basic value iteration algorithm can only be applied if the cost function does not increase (which frequently happens when the map is modified). This is because when the cost function increases, previously adjusted values V might become too small. While value iteration quickly decreases values that are too large, increasing too small a value can be arbitrarily slow (Thrun 1993). Consequently, the basic value iteration algorithm requires that the value function be initialized completely (initialization step) whenever the map—and thus the cost function—is updated. This is very inefficient, since the map is updated almost constantly. To avoid complete reinitializations, and to further increase the efficiency of the approach, the basic paradigm was extended in the following way:

Selective Reset Phase. Every time the map is updated, values $V_{x,y}$ that are too small are identified and reset. This is achieved by the following loop, which is iterated:

For all nongoal <x, y> do:

$$V_{x,y} \leftarrow \infty \text{ if } V_{x,y} < \min_{\substack{\xi=-1,0,1 \\ \zeta=-1,0,1}} \left\{ V_{x+\xi,y+\zeta} + Prob\left(occ_{x+\xi,y+\zeta}\right) \right\}$$

Notice that the remaining $V_{x,y}$-values are not affected. Resetting the value table is a version of value iteration.

Bounding Box. To focus value iteration, a rectangular bounding box $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ is maintained that contains all grid cells in which $V_{x,y}$ may change. This box is easily maintained in the value iteration update. As a result, value iteration focuses on a small fraction of the grid only and hence converges much faster. Notice that the bounding box bears similarity to prioritized sweeping (Moore and Atkeson 1993).

Value iteration is a very general procedure which has several properties that make it attractive for real-time mobile robot navigation:

Any-time Algorithm. As previously mentioned, value iteration can be under-

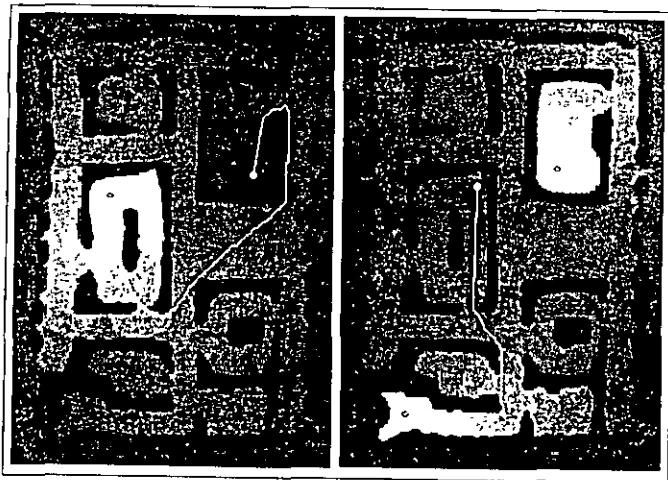


Figure 10 Path planning with dynamic programming. Value functions V, computed by value iteration for (a)(left) one goal and (b)(right) two goals (goals are marked by "0") By following the gray-scale gradient, the robot moves to the next unexplored area on a minimum-cost path.

stood as an any-time planner (Dean and Boddy 1988). Consequently, value iteration allows the robot to move in real-time, even though some of its motion commands might be suboptimal

Full Exception Handling. Value iteration preplans for arbitrary robot locations. This is because V is computed for every location in the map, not just the current location of the robot. Consequently, the robot can quickly react if it finds itself in an unexpected location, and generate appropriate motion directions without any additional computational effort. This is particularly important in our approach, since the robot uses a fast routine for avoiding collisions (described below) which adjusts the motion direction commanded by the planner based on sensor readings

Exploration. To autonomously acquire a map, the robot has to explore. For exploration, the same value iteration algorithm is employed, with the only exception being that goals correspond to unexplored grid cells. Figure 11a shows an autonomous exploration run. At the current point, the robot has already explored the major hallways and is about to continue to explore an interior room. Circular motion, such as found in the bottom of this plot, occurs when two unexplored regions are about equally far away (= same costs). Notice that the complete exploration run shown here took less than fifteen minutes The robot moved constantly and frequently reached a velocity of 80 to 90 centimeters per second (see also Buhmann et al. 1995; Fox, Burgard, and Thrun 1995).

Figure 11b shows the exploration value function. All white regions are unexplored, and the gray level indicates the cumulative costs V for moving toward the nearest unexplored point. The value function indicates the robot would continue exploration by moving straight ahead.

Multiagent Exploration. Since value iteration generates values for all grid-cells, it can easily be used for collaborative multiagent exploration.

In grid maps of size thirty by thirty meters, optimized value iteration done from scratch requires approximately two to ten seconds on a SUN Sparc station. In cases where the selective reset step does not reset large fractions of the map (which is the common situation), value iteration converges in less than a second. For example, the planning time in the map shown in figure 4 lies generally under two seconds, and most of the time under a tenth of a second. In light of these results, one might be inclined to think that grid-based maps are sufficient for autonomous robot navigation. However, value iteration (and similar planning approaches) requires time quadratic in the number of grid cells, imposing intrinsic scaling limitations that prohibit efficient planning in large-scale domains. Due to their compactness, topological maps scale much better to large environments. In what follows we will describe our approach to path planning with topological maps.

Path Planning with Topological Maps

The enormous compactness of topological maps—when compared to the underlying grid-based map—increases the efficiency of planning. To replace the grid-based planner by a topological planner, the planning problem is split into three subproblems, all of which can be tackled separately and very efficiently.

Topological Planning. First, paths are planned using the abstract topological map. Shortest paths in the topological maps can easily be found using one of the standard graph search algorithms, such as Dijkstra's or Floyd and Warshal's shortest path algorithm, A^*, or dynamic programming. In our implementation, we used the value iteration approach described in the path planning with grid-based maps section.

Triplet Planning. To translate topological plans into motion commands, a so-called "triplet planner" generates (metric) paths for each set of three adjacent topological regions in the topological plan. Specifically, let T_1, T_2, …, T_n denote the plan generated by the topological planner, where each T_i corresponds to a region

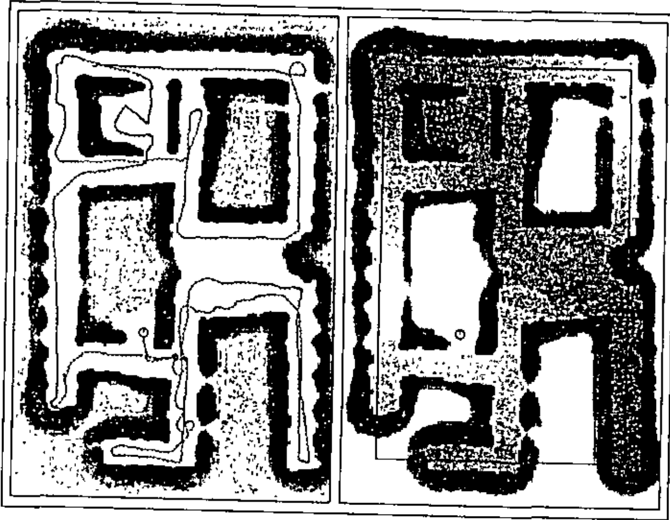


Figure 11. Autonomous exploration. Left (a) Exploration path. Right (b) Value function during exploration. Notice that the large black rectangle in (b) indicates the global wall orientation.

in the map. Then, for each triplet $\langle T_i, T_{i+1}, T_{i+2}\rangle$ ($i = 1, \ldots, n-1$ and $T_{n+1} := T_n$), and each grid cell in T_i, the triplet planner generates shortest paths to the cost-nearest point in T_{i+2} in the grid-based map, under the constraint that the robot exclusively moves through T_i and T_{i+1}. For each triplet, all shortest paths can be generated in a single value iteration run. Each point in T_{i+2} is marked as a (potential) goal point, and value iteration is used to propagate costs through T_{i+1} to T_i just as described in the path planning with grid-based maps section. Triplet plans are used to "translate" the topological plan into concrete motion commands. When the robot is in T_i, it moves according to the triplet plan obtained for $\langle T_i, T_{i+1}, T_{i+2}\rangle$. When the robot crosses the boundary of two topological regions, the next triplet plan $\langle T_{i+1}, T_{i+2}, T_{i+3}\rangle$ is activated. Notice that the triplet planner can be used to move the robot to the region that contains the goal location.

Final Goal Planning. The final step involves moving to the actual goal location, which again is done with value iteration. Notice that the computational cost for this final planning step does not depend on the size of the map. Instead, it depends on the size and the shape of the final topological region T_n, and the location of the goal.

The key advantage of this decomposition is that all the expensive com on

required for path planning can be done off-line, for all path planning problems. As documented in Thrun (forthcoming), planning using the topological map is between three and four orders of magnitude more efficient than planning with the grid-based map, for maps similar to those shown in this chapter. On the other hand, plans generated with the topological map are typically between one and four percent longer than plans generated using the grid-based map, numbers that are considerably small given the huge computational savings.

Collision Avoidance

The task of the collision avoidance routine is to navigate the robot to subgoals generated by the planner while avoiding collisions with obstacles. It adjusts the actual velocity of the robot and chooses the concrete motion direction. For obvious reasons, the collision avoidance module must operate in real-time. When the robot moves as fast as 90 centimeters per second, it is imperative that the robot dynamics (inertia, torque limits) be taken into account, particularly because the path planner considers only robot kinematics. The remainder of this section describes the "dynamic window approach" to collision avoidance (Fox, Burgard, and Thrun 1995; Fox, Burgard, and Thrun 1997), currently our best collision avoidance routine.

The key idea of the dynamic window approach is to choose control in the velocity space of the robot. Figure 12 shows an example of the robot traveling down a hallway with a certain velocity, and figure 13 shows the corresponding velocity space. The velocity space is a projection of the configuration space (with a fixed kinematic configuration). The horizontal axis in figure 13 measures the rotational velocity, denoted by ω, and the vertical axis depicts the translational velocity, denoted by v. The actual velocity of the robot is a single point in this diagram (in the center of the white region). The robot sets its velocities in regular time intervals (in our current implementation every 0.25 seconds). To ensure that the robot travels safely and makes progress toward the goal, it has to obey a variety of constraints, which are described in turn.

Hard Constraints. Hard constraints are imposed by the requirement to not to collide with obstacles and by the dynamics of the robot.

Torque Limits. Torque limits impose bounds as to how the robot might change its velocity in the immediate next decision interval. In the example shown in figure 13, these bounds are visualized by the rectangle V_d.

Safety. Obstacles impose additional constraints on the velocity. Velocities with which the robot would inevitably collide, even if decelerated maximally after the decision interval, are not admissible. The dark regions in figure 13 illustrate such regions in the velocity space. Notice that obstacles such as walls are directly mapped into the velocity space.

These constraints are hard constraints, i.e., for obvious reason robot

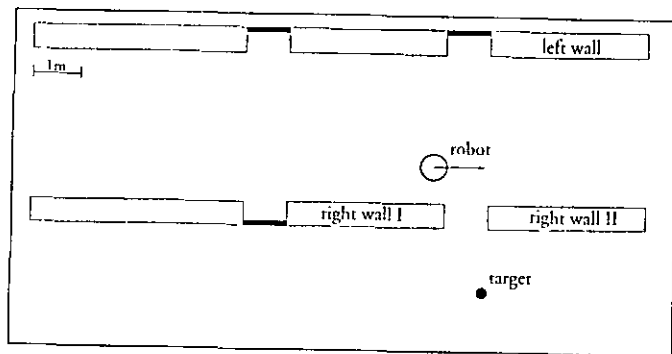


Figure 12 Example situation

must obey them. In figure 13, the space of admissible velocities under these constraints is depicted in white color. These constraints do not specify preferences, nor do they contain goal-related information.

Soft Constraints. Soft constraints impose preferences on the motion direction and velocity of the robot. Currently, three soft constraints are used:

1. Target heading. The target heading, denoted *heading*(v, ω), is defined as the absolute angle of the target (subgoal) relative to the robot's heading direction after 0.25 seconds of robot motion. The target point is usually set by the path planner. If *heading*(v, ω) = 0, the target would be right in front of the robot at the beginning of the next time interval. To make progress toward its target, it is desired that the robot move towards it, i e., the target heading be as close as possible to zero.
2. Clearance. The clearance, denoted by *dist*(v, ω), is defined as the free distance in front of the robot, assuming that the robots sets its velocity once and does not change it thereafter. By maximizing clearance, the robot avoids being close to obstacles.
3. Translational velocity The translational velocity v is also maximized, which causes the robot to always move as fast as permitted by the other constraints.

Notice that all three soft constraints, *heading*(v, ω), *dist*(v, ω), and v are functions of v and ω. The actual velocity, denoted by $\langle v^*, \omega^* \rangle$, is obtained by maximizing a linear combination of these constraints:

$$\langle v^*, \omega^* \rangle = \operatorname*{arg\,max}_{v,\omega} \left(-\alpha_1 \cdot |heading(v,\omega)| + \alpha_2 \cdot dist(v,\omega) + \alpha_3 \cdot v \right) \quad (4)$$

Here α_1, α_2, and α_3 are constants which trade off the three different soft constraints, thus determining the overall behavior of the robot. In our approach, (4) is optimized by discrete grid search

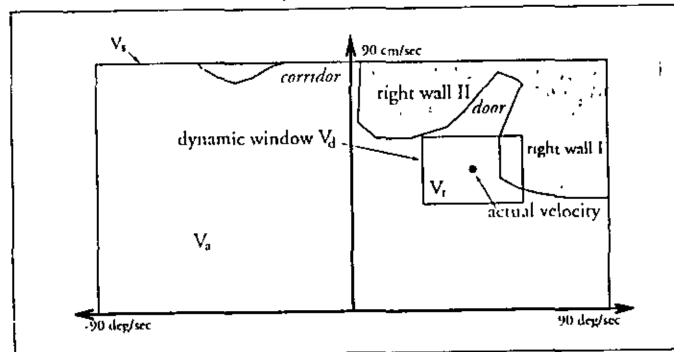


Figure 13 Velocity space

Figure 14 depicts the value (argument of the "argmax") corresponding to the situation depicted in figure 12, as a function of the velocities v and ω. Values that would violate the safety constraint are set to zero The shape of the remaining function indicates that higher translational velocities are generally preferable and that the rotational velocity possesses an "optimal" value, which is dictated by the target heading (ridge toward the right in figure 14). The global maximum, marked by the vertical line in figure 14, corresponds to a sharp right turn for moving directly through the door (see figure 12). Notice that the dynamic window (torque limits) is not shown in figure 14. If the dynamics of the robot do not permit such a turn, the robot would instead move straight, decelerate and eventually backup' Notice that taking the robot dynamics into account is important if the robot travels at more than 30 centimeters per second—if the dynamics were ignored, an attempt to turn at high speed could easily result in a collision.

Using Sensors. While the constraints are sufficient to generate collision-free robot control if the model of the environment is correct, sensors are erroneous and models err Thus, our approach employs a variety of additional strategies to recover from errors in perception.

Sensor Data. In reality, exact locations of obstacles are unknown. Maps react very slowly to changes in the environment, they are incapable of modeling fast-moving obstacles, and they typically lag behind because of the computational costs involved in sensor interpretation and localization Thus, the dynamic window approach works with raw proximity data, as obtained by sonar sensors or computer vision (see below). Every sensor reading can potentially constrain the motion of the robot; however, sensor readings are only memorized for approximately three seconds. Such an approach is maximally conservative in the sense

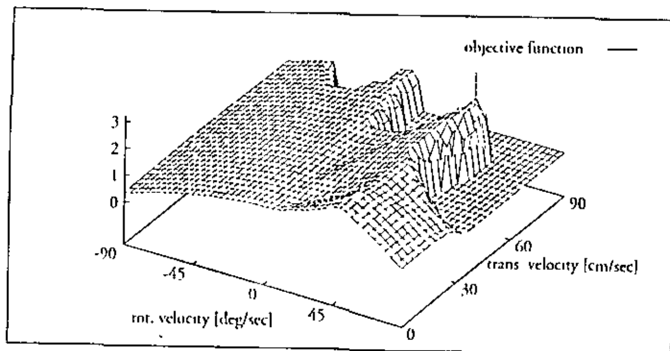


Figure 14 Example of an objective function The vertical axis depicts value.

that no sensor reading is ignored Nevertheless, the relatively short temporal window makes it adapt quickly to moving obstacles.

Smoothing The objective function is smoothed to increase the side-clearance of the robot and to decrease the sensitivity to noise in the sensor readings.

Safety Margin To travel safely at high speed, the robot is enlarged by a safety margin This safety margin increases with the forward velocity As a result, the robot keeps safe distances from obstacles when traveling at high speed, while still being able to move through narrow doors with low speed

Rotate Away Mode. Because of sensor noise and changes in the environment, it can happen that every nonzero velocity violates a hard constraint. In such cases, which are rare in practice, the robot turns completely away from the nearest obstacle, from which point on it resumes normal operation

Figure 15 shows an example of the robot traveling through a cluttered environment, based purely on sonar information All obstacles in the corridor are smoothly circumvented with a maximal speed of 90 centimeters per second. Although in this experiment the robot decelerated to approximately 20 centimeters per second when passing through the narrowest passage, it still maintained an average speed of 65 centimeters per second. In extensive experiments, we found that the dynamic window approach controls the robot very reliably even in populated environments with obstacles that are hard to detect for sonar sensors

Real-Time Vision

Sonars are convenient sensors in that they directly generate proximity information. However, they fail to detect obstacles outside their perceptual range, e g., small objects on the floor, and they also frequently fail to detect obstacles with

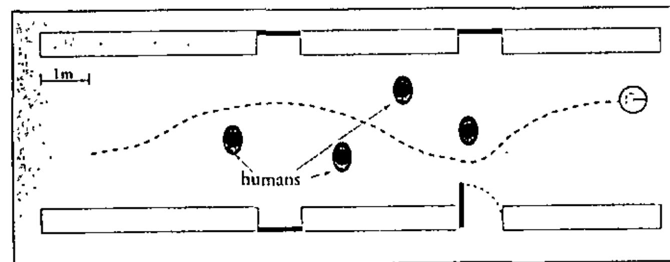


Figure 15. Trajectory through a cluttered corridor

sound-absorbing surfaces. To supplement the sonar information, a real-time vision module for obstacle detection is integrated into collision avoidance. Our monocular vision module is able to robustly detect and locate obstacles on the floor, based on image segmentation and discriminant analysis The main processing stages are as follows.

1. *Preprocessing.* The color image is low-pass filtered and subsampled, typically to 150x120 pixels
2. *Edge Detection.* The subsampled image is convolved with a gradient operator to detect vertically and horizontally oriented edges.
3. *Presegmentation* A fast, pixel-based image segmentation evaluates the local contrast between pixels and links them together if the contrast is below a certain threshold θ_0(Ballard and Brown, 1982). This presegmentation usually results in an over-segmented image with many small region patches.
4. *Segmentation.* Neighboring regions are merged (Zucker 1976) if the mean contrast along their common border is below a threshold θ_t, and if their average color differs by less than a second threshold, σ_t Both thresholds are iteratively increased $(\theta_{t+1}, \sigma_{t+1}) = (\theta_t, \sigma_t) + (\Delta\theta, \Delta\sigma)$ to a final threshold tuple (θ_f, σ_f).
5. *Interpretation* The segmented image is interpreted to identify the major floor region and to detect regions that possibly correspond to obstacles. Based on knowledge about the height and the tilting angle of the camera, the robot first locates the horizon within its camera image It then identifies floor patches using size and location relative to the robot and the horizon as the major criteria. The remaining nonfloor regions are obstacle candidates, given that they touch the floor and given that they fit certain size constraints.
6. *Feature extraction.* For each region of interest, the following features are extracted: (1) the height, (2) the width and (3) the total area, (4/5) two color components (the "*U*" and "*V*" channel of the NTSC signal), (6) average brightness and (7) brightness variance; the latter feature provides textur

formation. Height, width and area are expressed in world-coordinates, computed under the assumption that obstacles extend all the way to the floor.

7. *Recognition.* Finally, regions are classified by a Bayesian classifier. In the "training phase," the feature mean and covariance of typical obstacles that may block the robot's path are estimated from 50 to 100 examples. During recognition, a region is considered an obstacle—and passed to the collision avoidance module—if its Mahalanobis distance to one of the pre-trained obstacle models exceeds a certain threshold. The Mahalanobis distance corresponds to the probability of observing a feature vector assuming normal distribution (Duda and Hart 1973).

Once a region has been classified as obstacle, its world coordinates are passed to the collision avoidance, to supplement the sonar information. The vision module evaluates images with a frequency of more than 1 Hz on a Pentium computer. Obstacles of the size of a bottle are usually detected at a maximal range of five meters, which is sufficient for real-time collision avoidance even if our robot is operated at its maximum speed. Figure 16 shows two typical images together with random color representations of the segmentation result. In various experiments, we found this algorithm to reliably detect small, can-sized obstacles in our university building.

The reader may notice that this algorithm has been successfully used for detecting and grasping free-standing objects on the floor (Buhmann et al 1995), as demonstrated at the 1994 AAAI mobile robot competition (Simmons 1995).

Example Application

The RHINO-software described in this chapter has served as a low-level platform for various indoor mobile robot applications. A complete coverage of our current applications is beyond the scope of this chapter. One of the most recent and most interesting applications, however, is that of a robotic "tour guide" (similar to the one proposed in Horswill 1994). The tour guide offers tours to visitors and explains rooms, locations, and their relation to each other. For this purpose, RHINO is equipped with a CD ROM for storing and replaying music and text.

Figure 17 depicts one of the maps the robot used when giving tours through our university building in Bonn. Maps are recorded by teleoperating (joy-sticking) the robot through the building, using some of the techniques described in this chapter. Each location or object of interest (such as an office) is taught by moving toward it and pressing a button when the robot is facing the object or location at the distance of one meter. The gray shaded numbers in figure 17 depict eleven target locations, and the corresponding numbers with white background show the positions at which the robot was taught. The entire teach-in requires

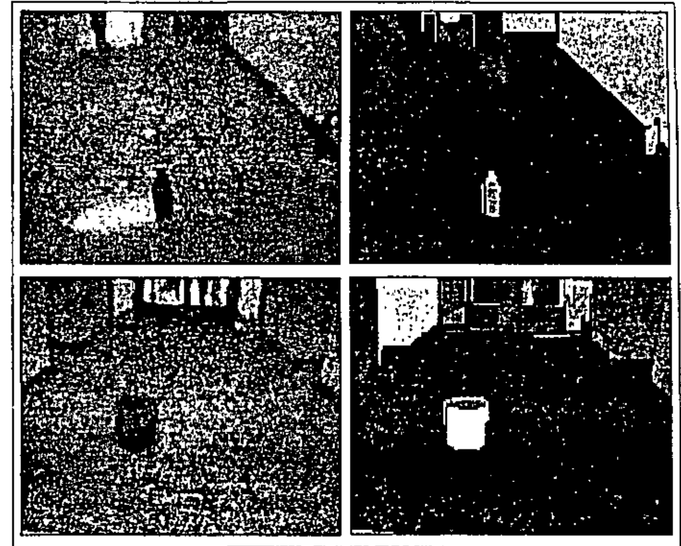


Figure 16: Two indoor scenes with obstacles and the corresponding image segmentation

less than ten minutes, not counting the time required for recording the verbal explanations of the different tour items. When the robot gives a tour, its localization routines quickly align its position with its previously constructed map. It then sequentially navigates to the target locations and replays previously recoded text for each of them (with user-controlled levels of granularity). The duration of the complete tour depends on the amount of information the user wants to obtain and is typically in the order of five minutes. If the visitor loses interest, the tour can be terminated at any time. In approximately thirty testing runs in different buildings, we never observed a failure of the navigation routines, even in populated hallways. We found that the integration of speech, sound, interaction, and fast motion contributes significantly to the interestingness of the guide

Conclusion

This chapter presents our currently best approaches to autonomous mobile robot navigation. Our current approaches are map-based. In particular, integrat-

ed metric-topological maps are learned autonomously using sonar and camera information. Bayesian analysis permits the robot to track its position accurately during navigation and mapping and to localize itself in cases where it is globally ignorant about its position. Path planning is performed by a fast dynamic programming routine, and collisions are avoided by a module that is capable of reacting to unforeseen obstacles by adjusting the motion direction and the velocity of the robot.

The software has been tested thoroughly using various mobile robots at different sites and is now distributed and regularly exhibited by a major mobile robot manufacturer (Real World Interface) along with their robots. The software provides the "low-level" control that allows several AI researchers inside and outside our university to perform high-level AI experiments without having to pay much attention to the low-level navigation.

Certainly, there are a variety of limitations and desiderata that warrant future research. The following list addresses some of the most significant and challenging ones:

- Dynamic environments. Maps, as presented in this chapter, are generally incapable of modeling moving obstacles. In a recent thesis (Schneider 1994), Schneider extended our approach to model semidynamic obstacles such as doors. Such obstacles are dynamic but appear only at fixed locations and thus can be detected by analyzing long-term dependencies in the occupancy grid. Modeling moving objects such as humans is a significant open problem in map-based navigation.
- Three-dimensional maps. Our current maps are two-dimensional, and our planning algorithms are tailored toward navigation of a circular robot. This chapter does not address manipulation. To facilitate manipulation, three-dimensional representations are clearly advantageous. Extending our current approach to 3D representations is an open problem that clearly warrants further research. Planning using such representations is necessarily more complex, and more sophisticated approaches are probably required if the robot is to plan and act in real-time.
- Self-tuning sensor interpretation. Currently, our sensor models are adjusted once and frozen thereafter. It is generally desirable to adjust sensor models while the robot is operating, to compensate for sensor defects and drift.
- A unified approach to localization. Our present approach relies on two quite different methods for localization, one of which is specialized to global position estimation, the other of which is dedicated to position tracking during map learning. Since both attack the same general problem—sensor-based localization under uncertainty—it is desirable to find a single, unified mechanism.
- Unknown state spaces. At various occasions throughout this chapter, we made the assumption that the robot environment is Markov and partially observable. In particular, all our probabilistic approaches, such as our methods for mapping and localization, make strong assumptions as to what the nonobservable quantities (state) of the environment are that make the environment Markov. Since in general it is difficult to specify what constitutes the "state" of the environment, methods that can discover hidden state and model it from data are clearly desirable (see for example, Chrisman [1992], McCallum [1995], Rabiner [1990]).

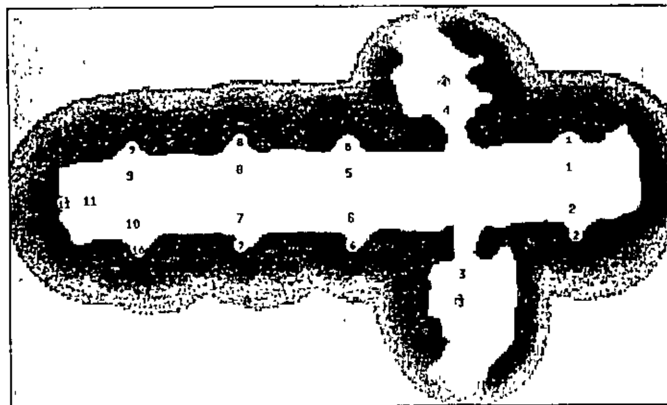


Figure 17. Map used for tours through our building
The numbers in gray circles indicate the different target objects, and the numbers with white background mark the positions at which these targets were taught.

- Other sensors. Integrating sensors other than sonar and cameras into mobile robot navigation is an important problem, since different sensors have different perceptual characteristics. In principle, the general mechanisms for mapping, localization, and navigation are not specialized to a particular type of sensor. However, incorporating other sensors is not trivial, and we believe many interesting research opportunities will come up from actually trying it.
- Scaling up. The largest cycle-free map that has been generated with this approach was approximately one hundred meters long, the largest single cycle measured approximately fifty-eight by twenty meters. What happens if the environment is an order of magnitude larger? Clearly, there are intrinsic limits as to how well a robot can localize itself incrementally without a global positioning system. We believe that by localizing the robot backwards in time, we can increase the size of the environments that can be mapped reliably. However, the general problem will never disappear, and the best we can hope for are incremental improvements in the size of environments that our software can manage reliably.

As this chapter documents, we have found the map-based paradigm to be surprisingly powerful and reliable. While to date, there exists a variety of successful architecture for mobile robot navigation (such as Brooks's (1989) subsumption architecture, each of which is characterized by different advantages and disadvantages, we believe that the map-based paradigm is particularly well-suited for fully autonomous robots that are to perform a multitude of tasks in large indoor environments. Maps are well understood, and it is easy to specify new navigation tasks using a map. We also believe that probabilistic models, such as the maps described in this chapter, are powerful concepts in robot navigation, as long as they compile percepts into compact representations that capture relevant details and are easy to access.

Acknowledgment

We thank the other members of the RHINO team and the XAVIER mobile robot team at CMU for insightful discussion, help and advice. The low-level process communication software TCX (Fedor 1993) was provided by CMU, which is gratefully acknowledged.

Sebastian Thrun's work on this project was sponsored in part by the National Science Foundation under award IRI-9313367, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330 (T. Mitchell, principal investigator). The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of NSF, Wright Laboratory or the United States Government. Thorsten Fröhlinghaus acknowledges financial support by the European Community under grant CORMORANT 8503 (J. Buhmann, principal investigator).

CHAPTER TWO

Integrating High-Speed Obstacle Avoidance, Global Path Planning, and Vision Sensing on a Mobile Robot

David Kortenkamp, Marcus Huber, Charles Cohen, Ulrich Raschke, Frank Koss, and Clare Congdon

The first Mobile Robot Competition was held by AAAI in 1992, where a total of ten robots competed in a series of events (Dean and Bonasso 1993). The robot described in this chapter, CARMEL, won this inaugural competition. It did so by combining high-speed sonar-based obstacle avoidance with task-directed vision. In this case study, we will give a detailed description of the robot and the algorithms that led it to victory.

The competition consisted of three stages. The first stage required roaming a 22 meter by 22 meter (70 foot by 70 foot.) arena while avoiding static and dynamic obstacles. The second stage involved searching for ten distinctive objects in the same arena and visiting each of the objects. Visiting was defined as moving to within two robot diameters of the object. The last stage was a timed race to visit three of the objects located in the second stage and return home. Since the first stage is primarily a subset of the second stage requirements and the third stage implementation is very similar to the second stage implementation, this case study will focus on the second stage of the competition.

The arena boundaries were defined by three-foot high walls and three-foot high cardboard boxes were used as obstacles. The objects to be found were ten-foot tall three-inch diameter poles. To each pole we attached an omnidirectional barcode tag, which allowed our computer vision system to distinguish one pole from another (see the section on vision sensing). The objects could be seen above the obstacles, while the clearance between obstacles was a minimum of 1.5 meters.

CARMEL (computer-aided robotics for maintenance, emergency, and life support) is based on a commercially available Cybermotion K2A mobile robot platform. It is a cylindrical robot about a meter in diameter, standing a bit less than a meter high when equipped with a large hollow shell (for holding electronics and other equipment) on top (see figure 1). It has a top speed of approximately

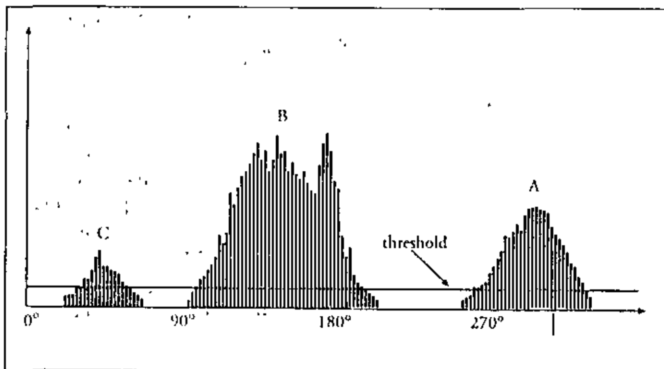


Figure 5. VFH's polar histogram has "mountains" in the direction of obstacles. CARMEL is steered toward a "valley" in the direction of the goal location.

The central idea behind a certainty grid is to fuse sonar readings over time to eliminate errors in individual sonar readings. In CARMEL's case, each cell of the grid array represents a ten centimeter square, and the array covers the entire environment space. As the robot (CARMEL) travels through the environment, its sonar sensors are continuously being fired and returning range readings to objects. Since the approximate location of the robot at any time is known through odometry and the direction of each sonar sensor is known, the location of objects in the grid array can be estimated. Each time an object is detected at a particular cell location, the value of the cell is increased and the values of all the cells between the robot and this cell are decremented (since they must be empty). The cells have minimum and maximum values, which have been chosen arbitrarily for computational convenience. Greater detail of the histogram method can be found in Borenstein and Koren (1991).

To perform the actual obstacle avoidance using the histogram grid, CARMEL creates an intermediate data representation called the polar histogram. The purpose of the polar histogram is to reduce the amount of data that needs to be handled for real-time analysis while at the same time retaining the statistical information of the histogram grid, which compensates for the inaccuracies of the ultrasonic sensors. In this way, the VFH algorithm produces a sufficiently detailed spatial representation of the robot's environment for travel among densely cluttered obstacles, without compromising the system's real-time performance. The spatial representation in the polar histogram can be visualized as a mountainous panorama around the robot, where the height and size of the peaks represent the proximity of obstacles, and the valleys represent possible travel directions (see figure 5). The VFH algorithm steers the robot in the direction of one of the valleys,

based on the direction of the target location. The details of CARMEL's algorithm for creating and using the polar histogram is detailed in Borenstein and Koren (1991b). A high-level description of the algorithm follows:

1. Collect current sonar sensor readings.
2. Create the histogram grid. For each sonar reading do the following:
   a) Given the direction of the sensor, the distance returned, and the robot's current location, determine the cell in the histogram grid in which the obstacle lies. If the sonar did not detect an obstacle, then determine the cell that lies at the maximum range of the sonar and skip the next step.
   b) Increment the value in that cell by a fixed amount (we use three in our implementation) up to some maximum (we use fifteen).
   c) For each cell lying on a straight line between the robot and the cell determined in (a) above, decrement its value by a fixed amount (we use one) down to a minimum of zero.
3. Calculate the polar histogram as follows:
   a) Define an active window surrounding the center of the robot (we use a window of size 33 x 33).
   b) For each direction around the robot, in five-degree increments, total the value in all cells in that direction within the active window.
   c) Smooth the histogram by averaging neighboring directions.
4. Set a threshold such that a polar histogram value below that threshold means that direction is "free," while a polar histogram value above that threshold means that direction is "occupied." This threshold can vary greatly between robots and even between environments.
5. Search the polar histogram for a free direction of travel as follows:
   a) If the direction to the target is free and enough neighboring histogram directions are free (to assure that the path is wide enough for the robot), then the target direction is the free direction.
   b) Otherwise, begin checking to the left and right of the target direction (alternating) for a free segment of the histogram that is wide enough for the robot to pass. The last direction in such a free segment can be returned as the free direction.
   c) If no free segment is found, the robot is trapped (i.e., surrounded on all sides by obstacles).

The combination of EERUF and VFH is uniquely suited to high-speed obstacle avoidance; it has been demonstrated to perform obstacle avoidance in the most difficult obstacle courses at speeds of up to 1.0 meters per second. All of CARMEL's experimental runs were at speeds of 780 millimeters per second or less. At the actual competition, CARMEL's maximum speed was 500 millimeters per second although it would run more slowly when in tight spaces or when approaching an object. CARMEL was distinguished by a graceful motion around obstacles in open terrain.

Global Path Planning

In addition to VFH, CARMEL uses a global path planner that searches the histogram grid created during obstacle avoidance and creates a list of via points (intermediary goal points) that represent the shortest path to the goal. The algorithm was developed at the Oak Ridge National Laboratory (Andersen et al, 1992) and reimplemented on CARMEL. The algorithm is simple and fast:

1. Initialize a planning grid of the same size and granularity as the histogram grid.
2. Threshold the histogram grid to determine occupied cells for planning purposes. Mark these cells as occupied in the planning grid.
3. Expand each occupied cell by the radius of the robot in the planning grid.
4. If the start or target locations fall inside an obstacle, move them to the nearest edge of the obstacle.
5. Create a linked list of structures of the type:

```
struct NODE {
    int x,y;                /* coordinates of grid cell */
    int distance;           /* distance from the target */
    struct NODE *next;      /* forward link */
};
```

6. Initialize the head of the linked list to the coordinates of the target cell and initialize the distance to zero.
7. While there is something in the list and the start cell has not yet been processed:
   a) Pop the head of the list.
   b) Place the distance of that item into its corresponding cell (x,y) in the planning grid.
   c) Put all neighboring cells that are still 0 and are not obstacles at the end of the linked list with a distance equal to one plus the distance of the popped item.
8. Starting at the start cell, follow the minimum path (including diagonal elements) to the target cell. If all the neighbors of the start cell are zero then there is no path.
9. Select those cells where the slope of the path changes and store their coordinates as via points that the robot should try to follow.

If a path is not found, it is possible to iterate the path planner using higher and higher thresholds for occupied cells. This algorithm is very quick, taking less than one second to run on a grid size of 256 by 256 cells (each cell represents 20 centimeters in the environment). The speed of the algorithm lets it run on the fly to extract CARMEL from potential trap situations. With an appropriate threshold, the algorithm rarely fails to find a path—in cases where a path is not

found CARMEL is simply instructed to head in the direction of the goal. One characteristic of this algorithm is that the path is not always the shortest path; however, VFH will cut corners while following via points along the way to a goal, allowing a path to be straightened out.

Grid-based systems have many limitations, including the amount of memory they require and the need to always know very precisely the location of the robot. In the competition, a grid representation was sufficient, as the arena was of a fixed (and known) size and we had mechanisms for determining the location of the robot. Given a larger environment, or an environment of unknown extent, a different representation scheme might be better.

Vision Sensing

The ability to accurately detect and identify objects in the world was important for earning the maximum number of points, as well as for keeping position and orientation errors within tolerable limits. The competition rules allowed the objects (tall poles) to be tagged to allow for easier recognition. Various tagging schemes were considered, but a vision-based system had an important advantage in its potential for long-range sensing. A major concern was the inherently heavy computation generally required for image processing. By intelligently designing the object tags, we greatly reduced the required computation.

Tag Design

The object tag design used for CARMEL consists of a black and white stripe pattern placed on 4 inch PVC tubing. Examples of the object tags that were used are shown in figure 6. The basic stripe pattern is six evenly spaced horizontal black bands (which we refer to as separator bands) of 100 millimeters height, with the top of the top band and the bottom of the bottom band spaced 1,100 millimeters apart. The size of the object tag is not important for the detection of objects. It is important, however, that the algorithm know the physical size of the object so that it can estimate its distance.

The white gaps between the black separator bands correspond to bit positions in a five-bit word, and are also 100 millimeters high. A white space between two black separator bands corresponds to an OFF bit, while a black space corresponds to an ON bit. The five bits between the six bands can then represent thirty-two unique objects. Although we needed to identify only ten objects, we utilized a five-bit code so that we could select bit patterns that did not have consecutive ON bits. Consecutive ON bits create long segments of black, reducing the number of black and white bands in the object tag and, therefore, reducing the number of constraints that an object has to satisfy before being accepted. By using

movement that is fast and continuous, displaying none of the move-stop-move behavior of many other robots at the competition. For the object detection submodule, this results in analysis of images that takes only a few seconds.

It also became apparent during the course of implementing the system that the performance of each submodule affected the design of the supervising executive. In particular, the object detection submodule evolved over time and became much more accurate and could extract objects at much greater ranges. This reduced the need for dealing with uncertainty and error.

The system was tested over many runs in three different environments, with the largest being 22 meters by 22 meters. Error recovery routines for movement errors were tested by forcing CARMEL into traps. Error recovery routines for object location errors were tested by manually moving objects after their positions had been recorded by the planner. In practice, the error recovery routines for visiting objects allowed us to be less concerned with the precise position of the robot, so that we didn't need to perform landmark triangulation as often. Since landmark triangulation is time consuming, requiring a movement and three images, the less often that it needs to be performed, the better.

In the actual competition, CARMEL found and visited all ten objects in just over nine minutes. This far out-paced the competition, none of whom could complete the task in the allotted twenty minutes. For the competition run, CARMEL used two hard-coded view points from which it hoped to see all ten objects; it went immediately to these two points, storing objects that it found in its map. Should CARMEL not have found all ten objects at these first two view points, there were other view points to which it could have gone. Because CARMEL was able to find all ten objects at the first two view points, its global object map was very accurate. CARMEL then proceeded to visit each object in turn, going to the nearest unvisited object first.

Conclusion

The CARMEL project teaches several important lessons in the design of actual mobile robot systems. The first is that a significant advantage is gained by keeping all computation on-board. Second, engineering the environment with simple tags can greatly simplify many robot tasks—much as hanging street signs greatly simplifies many human navigation tasks (see Miller 1994 for further discussion of this point). Third, the CARMEL project gives a blueprint for developing modular robot systems by encapsulating existing submodules and integrating them with a simple planner.

Finally, CARMEL demonstrates that in order to have a successful robot, much attention needs to be devoted to sensing. Fully forty percent of CARMEL's C code was dedicated to vision or sonar processing, with only thirty-five percent of the code devoted to higher-level tasks such as planning or triangulation (the rest of the code was devoted to user interface, communication, or other, low-level, processes). In the case of CARMEL's vision algorithm, we found that fast, reliable, long-range sensing can have a simplifying effect on task execution. If CARMEL had not been able to see as far, we would have had to design much more complicated exploration patterns to assure that the entire arena was explored.

Acknowledgments

The authors wish to thank the other members of the CARMEL team, including Johann Borenstein, Doug Baker, Chris Conley, Roy Feague, LiQuang Feng, Rob Giles, Scott Huffman, Kevin Mangis, Alex Woolf, Annie Wu, and Cigdem Yasar. Support for the CARMEL team was provided by The University of Michigan College of Engineering, The University of Michigan Rackham School of Graduate Studies, the American Association for Artificial Intelligence, ABB Robotics Inc., and ABB Graco Robotics Inc. Support for development of many of the algorithms used on CARMEL was supplied by Department of Energy grant no. DE-FG0286NE37969.