UNIVERSIDAD NACIONAL
AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

IMPLEMENTACION DE ESTRATEGIAS EVOLUTIVAS

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

RODOLFO A. SANCHEZ GUZMAN

DIRECTOR DE TESIS: DR. GUILLERMO FERNANDEZ ANAYA

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Dentro de la rama del conocimiento denominada "Inteligencia Artificial" existe una área de estudio conocida como "Computación Evolutiva", que agrupa varias técnicas diseñadas para explorar el espacio solución de un problema a través de una metáfora artificial del proceso de selección natural guiado por el principio de supervivencia de los individuos más aptos. Las ideas básicas de la computación evolutiva están inspiradas en los mecanismos de evolución descritos por Charles Darwin en El Origen de las Especies. De tal forma que la computación evolutiva propone la búsqueda de mecanismos de solución de problemas con base en la evolución de posibles soluciones, utilizando un proceso continuo de transformación, selección y evaluación de estas.

En muchas áreas de la ingeniería , disciplinas científicas y en actividades cotidianas nos encontramos frecuentemente con problemas que requieren ser resueltos utilizando los recursos disponibles de la mejor manera. El proceso que nos permite asignar esos recursos de la "mejor manera" es conocido como proceso de optimización. En términos generales este proceso está centrado en la optimización de funciones, es decir, dada una función f multidimensional, el problema de optimización de f puede verse como el problema de encontrar un subconjunto S del dominio de la función. Con la condición que no exista elemento en el complemento de S que al ser evaluado por f tenga un valor menor que la evaluación de cualquier elemento de S. Dada la definición anterior se hace necesario que el codominio de la función f sea un conjunto ordenado, es decir un conjunto en el que este definida la relación "menor que". A la función f se le conoce como Función Objetivo y a los elementos del dominio de la función se les llama variables objeto; los elementos de S son llamados mínimos de f .

Esta definición tan general nos permite expresar muchos problemas de ingeniería como problemas de optimización de funciones, por lo que es útil disponer de alguna técnica que pueda resolverlo sin pedir más restricciones que la definición de los dominios de las variables objeto y la definición de la Función Objetivo. Desafortunadamente no existe un método general para la optimización de funciones, lo que justifica la investigación

encaminada a encontrar técnicas cada vez más generales que nos auxilien en la solución de dichos problemas.

Existen muchos algoritmos de optimización de funciones. Sin embargo, la mayoría de ellos requieren de varias condiciones criticas que debe satisfacer la función a optimizar (Función Objetivo) para que puedan ser aplicados, entre ellas linealidad, derivabilidad o un número no muy grande de variables. Además, el número de variables que pueden manejar depende de la Función Objetivo, y la mayoría de ellos no tienen un buen desempeño si los puntos estimados inicialmente (soluciones iniciales al problema) se encuentran lejos del punto óptimo (solución óptima al problema).

Existen actualmente tres técnicas en computación evolutiva que buscan resolver problemas de optimización complejos, estas son: Algoritmos Genéticos, Estrategias Evolutivas y Programación Evolutiva. El trabajo presentado en esta tesis esta relacionado directamente con la técnica de optimización llamada Estrategias Evolutivas, esta técnica es poco conocida y fue descrita inicialmente I. Rechenberg [1] en 1973.

Uno de los procesos básicos utilizado por las Estrategias Evolutivas, común a otras técnicas de computación evolutiva, es el uso de mecanismos pseudoaleatorios para la creación de poblaciones de individuos y para su evolución hacia generaciones cada vez más aptas en la solución de un problema de optimización.

Más allá de esta similitud, una de las características de diferenciación de las técnicas evolutivas es el concepto de individuo utilizado por cada una de ellas, es decir, la definición estructural y semántica de aquello que se considera un individuo en evolución. Por ejemplo, el algoritmo genético original desarrollado por Holland [18, 20] define un individuo como una cadena de bits de longitud fija que, bajo un enfoque de interpretación específico, expresa una posible solución a un problema de optimización determinado. La programación genética [21, 22] define a un individuo como una estructura de datos tipo árbol, cuyos nodos son bloques funcionales que operan sobre el conjunto de operandos representados por las ramas que penden de él. Y las Estrategias Evolutivas definen a un individuo mediante un vector n-dimensional de números reales.

Otro aspecto fundamental en el funcionamiento de las técnicas evolutivas es el promover la convergencia de sus individuos hacia poblaciones con mejores cualidades en la solución del problema planteado, a pesar de la característica aleatoria subyacente en el proceso. Este efecto es logrado, en una forma no determinística, con la supervivencia de los individuos más aptos y, por lo tanto, con la reducción de la esperanza de vida de los individuos menos aptos.

Para determinar cuales son los individuos más aptos, se implementa un mecanismo capaz de determinar una métrica de aptitud[1] para cada individuo de la población. Este

[1] El termino "métrica de aptitud" se refiere a un valor comparable que califica la capacidad de un individuo para resolver un problema.

mecanismo, llamado generalmente Función Objetivo, es utilizado para clasificar a los individuos de una generación y llevar a cabo sobre ellos un proceso de selección para conformar una nueva generación de individuos. El proceso de selección promueve la convergencia de la población hacia una solución óptima del problema, que resulta superior a una metodología de búsqueda ciega, es decir, a una metodología completamente aleatoria.

La técnica de Estrategias Evolutivas no es necesariamente superior a otras técnicas de computación evolutiva o de inteligencia artificial en general. Históricamente, las distintas técnicas propuestas por la investigación en computación evolutiva han tenido éxitos y fracasos y las Estrategias Evolutivas no es una excepción a esta regla.

El abordar un trabajo de desarrollo de una herramienta de solución a problemas de optimización, tomando como base el paradigma de Estrategias Evolutivas, surge del interés por el estudio de dicha técnica, por dar a conocer sus características y la forma en que se utiliza.

Esta tesis tiene la siguiente organización: en el segundo capítulo se explica el funcionamiento y los diferentes tipos de Estrategias Evolutivas. En el tercer capítulo se presenta la implementación de las Estrategias Evolutivas, realizadas con el lenguaje de programación Scilab y C++. En el cuarto capítulo se muestra el desempeño de las dos implementaciones de Estrategias Evolutivas, con respecto a un conjunto de Funciones Objetivo frecuentemente usadas para este fin. Y se demuestra el uso de esta herramienta, en el desarrollo de una técnica nueva que permite reducir el número de elementos a ser procesados y el tiempo de aprendizaje de una Red Neuronal Artificial. En el quinto capítulo se presentan algunas conclusiones importantes de este trabajo. En el apéndice A se encuentran los programas en Scilab de las Estrategias Evolutivas y en el apéndice B los programas en C++.

Capítulo 2

Estrategias Evolutivas

2.1. Descripción de las Estrategias Evolutivas

La idea principal del funcionamiento de las estrategias evolutivas consiste en utilizar un conjunto de reglas básicas (aplicadas a una población) que tienen analogía con las reglas de evolución biológica [5, 17]. Los principales elementos y reglas son:

- población: conjunto de individuos, quienes inicialmente tienen valores aleatorios.
- individuo: cada individuo de la población es una solución en potencia del problema a resolver (vector de números reales).
- mutación: modificación aleatoria en alguna parte de la información que conforma al individuo de la población.
- recombinación: proceso para generar un nuevo individuo de la población, que se obtiene a través de la selección aleatoria (varios métodos) de los elementos que componen dos o más individuos.
- evaluación: asignación de aptitud a cada individuo de la población por medio de la Función Objetivo, con el propósito de determinar su eficacia para resolver cierto problema.
- codificación: transforma los elementos del dominio del problema a resolver a un vector de números reales, que es el individuo.
- decodificación: transformación que se aplica a cada individuo de la población para poder medir su aptitud.
- selección: separación de los individuos más aptos de los menos aptos que existen en la población.

Existen dos tipos principales de estrategias evolutivas [6], la estrategia $(m,l)-ES$ y las estrategia $(m+l)-ES$. La diferencia entre estos dos tipos de estrategias es que en la estrategia $(m,l)-ES$ se tienen poblaciones de m padres los cuales perecen (no pueden sobrevivir a la siguiente iteración) quedando sólo l hijos de los que se seleccionan otros m padres para la siguiente iteración, es decir ningún individuo puede sobrevivir por siempre y el proceso de selección se realiza con los nuevos descendientes únicamente. En cambio en la estrategia $(m+l)-ES$, tanto los padres como los hijos participan en el proceso de selección permitiendo a un individuo permanecer en las nuevas generaciones de manera indefinida; mientras los hijos que se generen en la población no sean más aptos que él.

2.2. Descripción formal

Se asume que el problema de optimización n-dimensional es de la forma:

$$x^* = min\{f(\overrightarrow{x}) \mid \overrightarrow{x} \in \Re^n\}$$

Los elementos y operadores de las estrategias evolutivas se representan con las siguientes expresiones:

Los individuos $\overrightarrow{a}$ son elementos del espacio de los individuos I. $I = \Re^n \times \Re^n_+$

Un individuo $\overrightarrow{a} = (\overrightarrow{x}, \overrightarrow{\sigma}) \in I$ consiste de los componentes:

$x \in \Re^n$ **: vector de variables objeto. Esta es la única parte de $\overrightarrow{a}$ que es usada en la Función Objetivo.**

$\overrightarrow{\sigma} \in \Re^n_+$ **: vector de desviaciones estándar de la distribución normal.**

El vector $\overrightarrow{\sigma}$ es el modelo interno que tiene cada individuo de la población. Este vector define una distribución normal n-dimensional, que se usa para explorar el dominio de la función $f(\overrightarrow{x})$.

$m, l \in \aleph$ **son el número de padres y descendientes (hijos) respectivamente.**

$P^{(t)} = \{\overrightarrow{a_1}, \ldots, \overrightarrow{a_k}\} \in I^k$ **es una población de $k \in \{m, l\}$ individuos en la iteración t .**

$rec : I^m \longrightarrow I$ **recombinación**

$mut : I \longrightarrow I$ **mutación**

$Sel^k_m : I^k \longrightarrow I^m$ **selección,** $k \in \{l, m+l\}$

Una iteración de la estrategia evolutiva, transición de la población $P^{(t)}$a la siguiente población de padres $P^{(t+1)}$, se representa con:

$$ES : I^m \longrightarrow I^m$$

Verificándose que:

$$ES\left(P^{(t)}\right)_{(m,l)} = Sel_m^k\left(\bigsqcup_{i=1}^{l}\left\{mut\left(rec\left(P^{(t)}\right)\right)\right\}\bigsqcup Q\right) \tag{2.1}$$

donde:

$Q \in \left\{P^{(t)}, 0\right\}$ **dependiendo del tipo de operador selección.**

Para el operador selección (m,l) **:** $Q = 0, k = l$

Para el operador selección $(m+l)$ **:** $Q = P^{(t)}, k = m+l$

$\bigsqcup$ **es el operador unión en multiconjuntos.**

El operador recombinación $rec : I^m \longrightarrow I$ **es aplicado antes que el operador mutación para crear un individuo obtenido de la población de padres. Este operador crea solo un individuo cada vez que se aplica. Primero selecciona** $\rho\,(1 \leq \rho \leq m)$ **padres de** $P^{(t)} \in I^m$ **usando una distribución de probabilidad uniforme, y mezcla los elementos de los** ρ **padres, para crear un descendiente.**

El operador recombinación modifica a las variables objeto así como también al modelo interno, y el operador puede ser diferente para los componentes $\overrightarrow{x}$ **y** $\overrightarrow{\sigma}$ **que forman un individuo. Por lo anterior hay dos tipos de operadores para la recombinación, uno para el vector** $\overrightarrow{x}$ **y otro para el vector** $\overrightarrow{\sigma}$ **:**

$$rec = (re_x \circ co_x) \times (re_\sigma \circ co_\sigma)$$

donde:

$co : I^m \longrightarrow I^\rho$ **selecciona los** ρ **padres.**

$re : I^\rho \longrightarrow I$ **crea un descendiente.**

Si $\rho = 2$ **se obtiene una recombinación bisexual y cuando** $\rho = m$ **se obtiene una recombinación global, que es la más usada y se utiliza en esta tesis.**

El operador mutación $mut : I \longrightarrow I$ **se define como:**

$$mut = mu_x \circ mu_\sigma$$

Este operador se usa después de haber aplicado el operador recombinación a un individuo $\overrightarrow{a}$**.**

$$\overrightarrow{a} = (x_1, \ldots x_n, \sigma_1, \ldots, \sigma_n)$$

Primero se muta el modelo interno $\overrightarrow{\sigma}$ **del individuo** $\overrightarrow{a}$**. Y después se modifica** $\overrightarrow{x}$ **usando el nuevo modelo interno** $\widetilde{\overrightarrow{\sigma}}$ **obtenido al mutar** $\overrightarrow{\sigma}$**.**

$mu_\sigma : \Re_+^n \longrightarrow \Re_+^n$ **muta el previamente recombinado** $\overrightarrow{\sigma}$ **:**

$$mu_\sigma\left(\overrightarrow{\sigma}\right) = \left(\sigma_1 \exp\left(z_1 + z_0\right), \ldots, \sigma_n \exp\left(z_n + z_0\right)\right) = \widetilde{\overrightarrow{\sigma}}$$

donde:

$z_0 \sim N\left(0, \tau_0^2\right)$

$z_i \sim N\left(0, \tau^2\right) \forall i \in \{1, \ldots, n\}$

$mu_x : \Re^n \longrightarrow \Re^n$ **muta el vector recombinado** $\overrightarrow{x}$ **, usando** $\widetilde{\overrightarrow{\sigma}}$ **:**

$$mu_x\left(\overrightarrow{x}\right) = \left(x_1 + z_1, \ldots, x_n + z_n\right) = \widetilde{\overrightarrow{x}}$$

donde:

$z_i = N\left(0, \widetilde{\overrightarrow{\sigma_i}}^2\right)$

Combinando los operadores mutación, recombinación, y selección como se han definido en esta tesis, la etapa principal de la estrategia evolutiva $(m, l) - ES$ **queda definida a partir de la ecuación 2.1 como:**

$$ES\left(P^{(t)}\right)_{(m,l)} = Sel_m^l\left(\bigsqcup_{i=1}^{l}\left\{mut\left(rec\left(P^{(t)}\right)\right)\right\}\right)$$

El operador selección obtiene un conjunto consistente de los m **individuos más aptos, de un conjunto de descendientes (hijos) de tamaño** l**.**

Para el caso de la estrategia evolutiva $(m+l)-ES$ el operador selección obtiene un conjunto consistente de los m individuos más aptos, de una población de hijos de tamaño $(m+l)$. Por lo que esta estrategia evolutiva se define como:

$$ES\left(P^{(t)}\right)_{(m+l)} = Sel_m^{m+l}\left(\bigsqcup_{i=1}^{l}\left\{mut\left(rec\left(P^{(t)}\right)\right)\right\}\bigsqcup P^{(t)}\right)$$

El siguiente algoritmo muestra el pseudocódigo de la estrategia evolutiva $(m,l)-ES$:

$t = 0$

inicializar $P^{(0)} = \{\overrightarrow{a_1}, \ldots \overrightarrow{a_m}\} \in I^m$

while $\left(T\left(P^{(t)}\right) = 0\right)$ **do**

$\widetilde{P} = 0$

for $i = 1$ to l do

$\left(\widetilde{\overrightarrow{x}}, \widetilde{\overrightarrow{\sigma}}\right) = mut\left(rec\left(P^{(t)}\right)\right)$

evaluar $f\left(\widetilde{\overrightarrow{x}}\right)$

$\widetilde{P} = \widetilde{P}\bigsqcup\left\{\left(\widetilde{\overrightarrow{x}}, \widetilde{\overrightarrow{\sigma}}\right)\right\}$

end

$P^{(t+1)} = Sel_m^l\left(\widetilde{P}\right)$

$t = t+1$

end

El proceso de auto-adaptación del modelo interno de cada individuo, tiene como base la existencia de un vínculo indirecto entre la aptitud del individuo y su modelo interno. Así como también en la gran diversidad de modelos internos que existen en la población de padres.

El tamaño de la población de padres m tiene que ser claramente mas grande que uno, por ejemplo $m = 15$ [2], y una relación de $l/m \approx 7$ es recomendada para establecer el tamaño de la población de hijos con respecto a la de padres[1].

[1]Frecuentemente se usa una estrategia evolutiva: $(15, 100) - ES$.

Hasta hoy, no se han obtenido resultados teóricos sobre el proceso de auto-adaptación del modelo interno, pero experimentalmente se han identificado tres condiciones principales para que el proceso de auto-adaptación del modelo interno sea exitoso [2]:

- Tener preferencia por estrategias evolutivas del tipo $(m, l) - ES$.
- El número de individuos obtenidos a través del operador selección debe ser claramente mayor que uno.
- Usar el operador recombinación.

El criterio para terminar el proceso de optimización, mas sencillo y frecuentemente utilizado $T : I^m \longrightarrow \{0, 1\}$, consiste en detener la evolución cuando el número previamente establecido de iteraciones se ha completado. Este criterio se aplica independientemente de la diversidad de la población o del número de iteraciones necesarias para que mejore la población.

2.2.1. Características de las Estrategias $(m + l) - ES$ y $(m, l) - ES$

Obsérvese que las descripciones presentadas suponen que el conjunto sobre el que se definen las variables objeto es el conjunto de los números reales y un individuo es un vector n-dimensional sobre este conjunto. Esto es una ventaja sobre los algoritmos genéticos [23, 24, 25, 26] que utilizan cadenas binarias o alfabetos discretos como individuos, dado que resolver problemas con magnitudes cuantitativas en ese caso suponía una codificación especial y en las estrategias evolutivas se manejan directamente los números reales. Por lo tanto podemos utilizar estrategias evolutivas en problemas de optimización, que puedan codificar sus soluciones en un vector en $\Re^n$ [19]. A pesar que la estrategia evolutiva descrita anteriormente es aplicable directamente a problemas de minimización, esto no le impide resolver problemas de maximización si se observa que $max(x) = min(-x)$.

El comportamiento del algoritmo utilizando estrategias $(m, l) - ES$ difiere del $(m + l) - ES$ [9]. En el primero, dado que los padres no pueden sobrevivir a la siguiente generación, la aptitud de los individuos de la población no siempre decrece, pudiendo presentar comportamientos oscilatorios. En cambio en la estrategia $(m+l)-ES$ siempre se tiene al mejor individuo en la generación siguiente, por lo cual la mejor aptitud de la población se comporta como una función decreciente respecto a las iteraciones, lo que puede presentar una convergencia mas rápida, pero con el peligro de caer más fácilmente en mínimos locales que la estrategia $(m, l) - ES$.

La definición de la Función Objetivo es quizá la parte más importante cuando se utilizan estrategias evolutivas. La Función Objetivo debe elegirse de tal manera que se asigne una aptitud cada vez menor si el vector que evalúa se aproxima a la solución del problema. Este el caso en que se trata de encontrar x tal que $f(x)$ es el mínimo.

Las características deseables de una Función Objetivo es que sea suave (que no tenga discontinuidades o cambios bruscos), que sólo tenga un mínimo (el mínimo global) y que si tiene mínimos locales estos no sean pronunciados. Además es deseable que la Función Objetivo no asigne a diferentes vectores del dominio de la función la misma aptitud si éstos se encuentran cerca entre sí, esto es, que la superficie n-dimensional descrita por la Función Objetivo no tengan regiones constantes, debido a que las regiones constantes no brindan información al algoritmo para que éste evolucione adecuadamente [32, 29, 30].

Capítulo 3

Diseño de las Estrategias Evolutivas

Existen varias formas de implementar una estrategia evolutiva, pero todas se definen utilizando operadores generales que se aplican a cada individuo de una Población (conjunto de posibles soluciones a un problema de optimización) y a su modelo interno correspondiente (rango de mutación que tienen los individuos). Los operadores y la secuencia de aplicación de los mismos (para los casos presentados en esta tesis) es: recombinación ->mutación ->selección. El problema a optimizar es representado dentro la Función Objetivo del algoritmo.

Existen muchas variaciones sobre la forma de implementar estos operadores. Para este caso la recombinación es del tipo global discreta y los otros operadores siguen lo definido en [6, 9]. Las estrategias evolutivas utilizadas son del tipo $(m+l)-ES$ y $(m,l)-ES$, estas son las mas generales y las mas estudiadas actualmente.

La principal diferencia entre la implementación en C++ y Scilab, radica en que la versión en C++ no incluye la funcionalidad de mutaciones correlacionadas [31, 28, 27]. Debido a que en trabajos realizados anteriormente con mis colegas; este tipo de mutaciones han mostrado ser deficientes en la resolución de problemas complejos de optimización [3, 4].

3.1. Implementación en C++

3.1.1. Instalación

Esta implementación utiliza una librería para el manejo de matrices realizada también en C++ (MatClass). La librería puede ser obtenida a través de la Internet. No se especifica alguna dirección de la Internet, debido a que es frecuente que existan cambios en las

direcciones asociadas a los proyectos de investigación. Por lo que es mas fácil encontrar esta librería, usando un buscador para la Internet.

Antes de instalar los archivos correspondientes a esta estrategia evolutiva, se deben instalar los archivos de la librería de matrices. Las clases y métodos de la librería MatClass, que son utilizados por la estrategia evolutiva y funcionan en su totalidad se listan a continuación:

Clase matrix

1. nCols()
2. nRows()
3. reset()
4. print()
5. put()
6. putReal()
7. newLine()
8. multijEq()
9. multij()
10. sumsq()
11. exp()
12. normf()
13. heapMap()
14. shellMap()
15. setRow()
16. setCol()
17. setSub()
18. rowOf()
19. colOf()
20. row()
21. col()
22. sub()
23. +, -, x, =, +=, x=, ||,

Clase matRandom

1. uniform()
2. normal()
3. seed()

Clase outFile

1. open()
2. put()
3. putIndex()
4. newLine()
5. close()

Clase inFile

1. open()
2. get()
3. getIndex()
4. nextLine()
5. close()

Para instalar la estrategia evolutiva, se deben modificar las variables asociadas a las ruta donde se encuentran los archivos de la librería de matrices, así como la ruta del directorio donde se instalara la estrategia evolutiva; estas se encuentra en el archivo Makefile.

Ejemplo:

- CCFLAGS=**-I**/home/mi_directorio/directorio_fuente_LibMatrices

  -I/mi_directorio/directorio_de_Estrategia_Evolutiva
  -L/directorio_de_librería_de_Matrices

Para compilar la librería de matrices, se teclea:

 # make

Verificando que la Sesión del interprete de comandos este ubicada en el directorio donde se encuentran los archivos fuente de esta librería.

Para compilar los ejemplos de la estrategia evolutiva, se teclea:

```
# make all
```

Teniendo la Sesión del interprete de comandos ubicada en el directorio donde se encuentran los archivos fuente de la estrategia evolutiva. Todo el código fuente de este algoritmo esta en el archivo Est_Evo.h, con la intención de simplificar la utilización del mismo.

Esta estrategia evolutiva fue diseñada usando el compilador GNU C/C++ y el sistema operativo FreeBSD. El compilador cumple con el estándar ANSI, lo cual facilita que este algoritmo funcione con otros compiladores y plataformas. Además el compilador GNU C/C++ puede ser instalado en otros sistemas operativos.

La estrategia evolutiva, así como MatClass también fueron compilados utilizando DJGPP (GNU C/C++ para DOS). Este compilador, así como el mencionado anteriormente; puede ser obtenidos en el sitio de la Internet: Sitio de GNU `www.gnu.org`.

3.1.2. Utilización

El primer ejemplo (ver apéndice: ex1.cpp), muestra la forma en que se declara un Objeto perteneciente a la Clase Est_Evo y es inicializado. Una vez que existe el Objeto se le pueden aplicar los diferentes métodos que contiene esta Clase, el método mas general es: evolve. El cual a su vez llama a métodos mas particulares que permiten realizar todo el proceso de optimización. Este ejemplo usa como Función Objetivo una Hiper-esfera cuya solución es el vector nulo (ver gráfica 4.1). La dimensión de la Hiper-esfera es el número de variables con el que trabaja la estrategia evolutiva, la cual se especifica al inicializar un Objeto de la Clase Est_Evo. La Hiper-esfera es la Función Objetivo que por "default" utiliza esta estrategia evolutiva.

El segundo ejemplo (ver apéndice: ex5.ccp), modifica la Clase Est_Evo para implementar una Función Objetivo diferente a la que por "default" se utiliza. Lo anterior se hace por medio de la característica de Herencia, de la Programación Orientada a Objetos; esto permite definir una nueva Clase utilizando otras previamente definidas (en este caso Est_Evo y MatClass). Por lo cual es posible crear un nueva Clase, con las características necesarias para un problema de optimización de propósito particular. El único método (función) que hay que modificar, al definir una nueva Clase usando la Clase Est_Evo es: Objective; en la cual se implementa la Función Objetivo del problema a optimizar. La gráfica 4.3 muestra la Función Objetivo, para el caso bidimensional.

3.2. Implementación en Scilab

Scilab fue desarrollado en el INRIA, Institut National de Recherche en Informatique et Automatique, un instituto de investigación francés. Sus principales características son:

- Software para cálculo científico.
- Interactivo.
- Programable.
- Software con licencia de uso GNU.
- Disponible para diferentes plataformas: Unix, Linux, Sun, Alpha, Windows.

El sitio oficial de Scilab es: `http://www.scilab.org` , en donde se encuentra información general, manuales, FAQs (frequent asked questions), diferencias con Matlab.

3.2.1. Instalación

Unix/Linux/Windows (solo en Scilab 2.5)

Usuarios:

- Se debe ejecutar la siguiente instrucción dentro la sesión

  de Scilab: exec('<PATH>/loader.sce')

- Antes de usar este toolbox se puede poner esta instrucción

  en el archivo .scilab, para que este toolbox se instale de
  forma automática al usar Scilab.

Administrador:

- Ejecutar una sola vez y para todos los usuarios, la siguiente

  instrucción dentro de la sesión de Scilab: exec('<PATH>/builder.sce')
  Esta operación requiere que se tengan permisos de escritura
  en <PATH>/macros para generar los archivos *.bin, names y lib
  en el directorio <PATH>/macros.

3.2.2. Utilización

Los ejemplos utilizados son los mismos que en la implementación en C++, y las gráficas de los mismos se muestran en el siguiente capitulo. Al igual que en C++, la Función Objetivo por "default" es la Hiper-esfera, a menos que se modifique el archivo asig_apt.sci.

Esta implementación en Scilab ha sido usada en los siguientes proyectos de investigación realizados en el Grupo LINDA [33] :

- Design of a Walking Machine Structure Using Evolutionary Strategies [35].
- Evolutionary Algorithms for Plants Simultaneous Stabilization [3, 4].
- Simultaneous and Robust Observability using Evolutionary Strategies [36].
- Strong Simultaneous Stabilization using Evolutionary Strategies [34].

3.3. Ambiente de Computo Utilizado

3.3.1. Características y ventajas del sistema operativo UNIX FreeBSD

FreeBSD (`http://www.freebsd.org`) es un sistema operativo para arquitecturas x86, IA-64, DEC Alpha, PC-98 y UltraSPARC. Este sistema operativo es un derivado de BSD UNIX, la versión de UNIX desarrollada en la Universidad de California, Berkeley. FreeBSD es desarrollado y mantenido por un numeroso equipo de personas, ofrece varias ventajas en lo referente a comunicaciones en red, rendimiento, seguridad y compatibilidad, todavía inexistentes en otros sistemas operativos; incluyendo los comerciales de mayor "renombre".

FreeBSD es el servidor adecuado para servicios de Computo Intensivo, Internet o Intranet. Proporciona servicios de red robustos, incluso en situaciones de alta carga, haciendo un uso eficaz de la memoria para mantener buenos tiempos de respuesta con cientos o miles de procesos simultáneos de usuarios.

La calidad de FreeBSD combinada con el hoy en día bajo costo del hardware de alta velocidad para PC's, hace de este sistema operativo una alternativa muy económica sobre las estaciones de trabajo UNIX comerciales, y sobre los sistemas operativos inmaduros (distribuciones de Linux). Además existe gran cantidad de aplicaciones tanto a nivel servidor como usuario.

El desarrollo de esta tesis se llevo a cabo utilizando este sistema operativo en la implementación y prueba de los algoritmos en C++ y Scilab, así como en el uso de las herramientas de Desktop asociadas al desarrollo de esta misma.

Las especificaciones técnicas del equipo y del software utilizado son: PC AMD 850 Mhz, 246 MB Ram, FreeBSD 4.8, Scilab 2.7, GCC 2.95.

Capítulo 4

Uso de las Estrategias Evolutivas

4.1. Pruebas sobre el desempeño de las Estrategias Evolutivas en C++ y Scilab

4.1.1. Conjunto de Funciones Objetivo utilizadas

En esta sección se muestran las expresiones n-dimensionales de las cinco Funciones Objetivo [9] utilizadas para medir el desempeño de las dos implementaciones de Estrategias Evolutivas. Así como las gráficas tridimensionales y las gráficas de curvas de nivel de cada función.

Ejemplo 1:

$$F(x) = \sum_{i=1}^{n} x_i^2$$

Mínimo: $x_i^* = 0 \,\forall\, i = 1 \ldots n \; F(x^*) = 0$

Ejemplo 2:

$$F(x) = \sum_{i=1}^{n} \left[\left(x_1 - x_i^2\right)^2 + (1 - x_i)^2 \right]$$

Mínimo: $x_i^* = 1 \,\forall\, i = 1 \ldots n \; F(x^*) = 0$

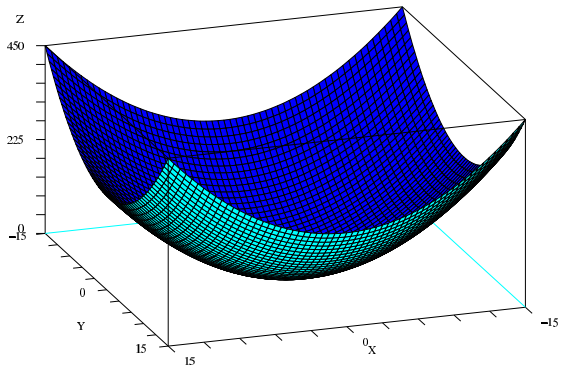

Figura 4.1: Función Objetivo 1

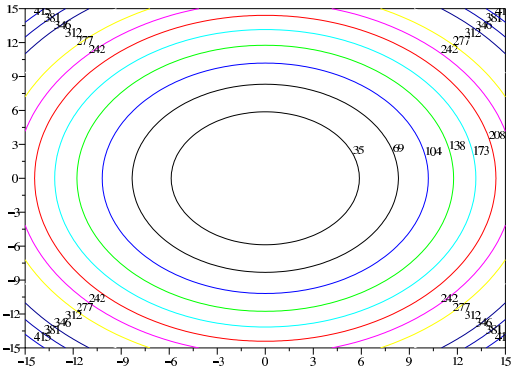

Figura 4.2: Curvas de Nivel 1

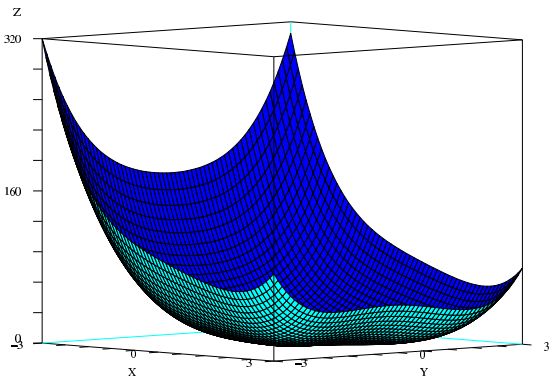

Figura 4.3: Función Objetivo 2

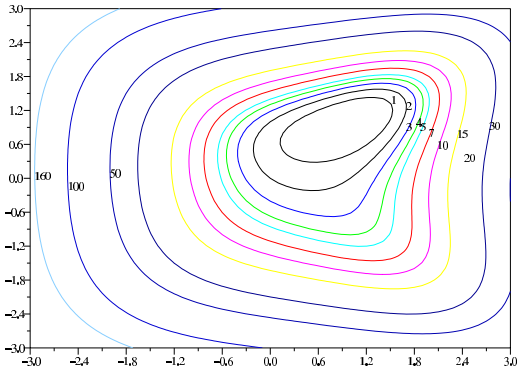

Figura 4.4: Curvas de Nivel 2

Ejemplo 3:

$$F(x) = \sum_{i=1}^{n} |x_i|$$

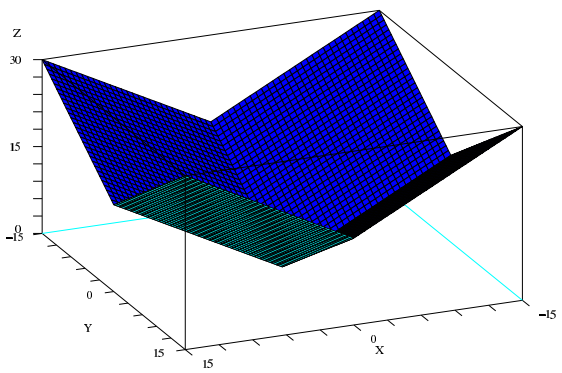

Figura 4.5: Función Objetivo 3

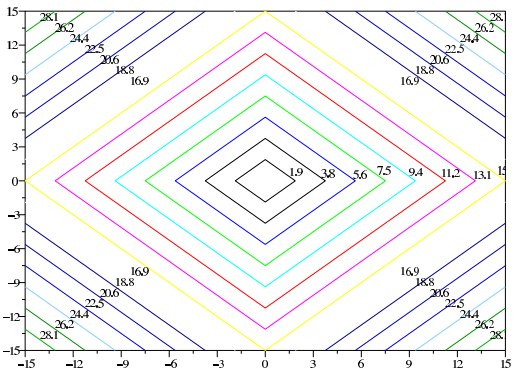

Figura 4.6: Curvas de Nivel 3

Mínimo: $x_i^* = 0 \,\forall\, i = 1 \ldots n \; F(x^*) = 0$

Ejemplo 4:

$$F(x) = \sum_{i=1}^{n} |x_i| + \prod_{i=1}^{n} |x_i|$$

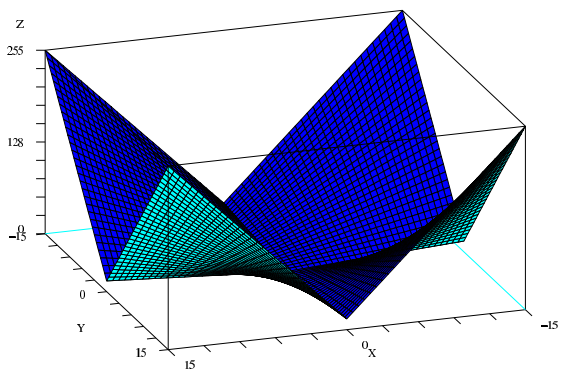

Figura 4.7: Función Objetivo 4

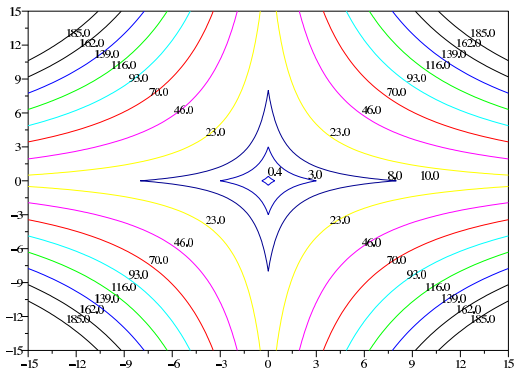

Figura 4.8: Curvas de Nivel 4

Mínimo: $x_i^* = 0 \, \forall \, i = 1 \ldots n \; F(x^*) = 0$

Ejemplo 5:

$$F(x) = \sum_{i=1}^{n} x_i^{10}$$

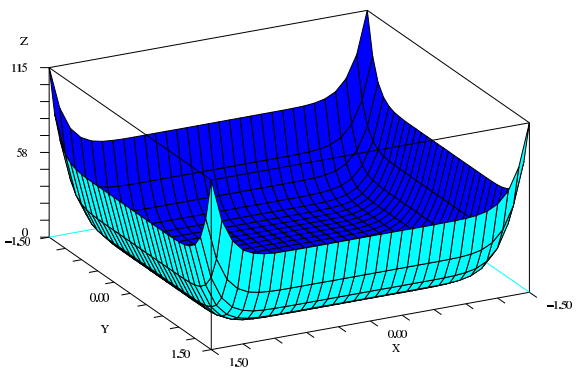

Figura 4.9: Función Objetivo 5

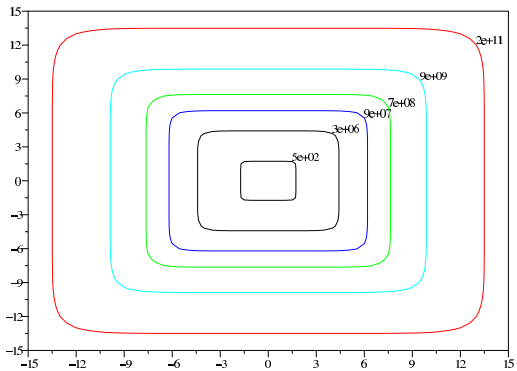

Figura 4.10: Curvas de Nivel 5

Mínimo: $x_i^* = 0 \, \forall \, i = 1 \ldots n \; F(x^*) = 0$

4.1.2. Resultados utilizando Scilab

En las siguientes gráficas se muestran los resultados obtenidos por la implementación de Estrategias Evolutivas en Scilab, para encontrar el mínimo global de las Funciones Objetivo descritas anteriormente. La optimización de cada Función Objetivo se realizó usando los dos tipos de Estrategias Evolutivas $(m+l)$ y (m,l) . Las gráficas muestran como cambia el valor de aptitud del mejor individuo de la población (eje vertical), con respecto al número de generación o iteración en el que esta la población (eje horizontal).

Este algoritmo encontró satisfactoriamente el mínimo global de todos los ejemplos presentados y los parámetros que fueron constantes durante estas pruebas son:

- Número de padres = 70
- Número de hijos = 490
- Número de variables (dimensión) de la Función Objetivo = 10

Los individuos de la población inicial que se usaron en los ejemplos fueron generados aleatoriamente dentro del intervalo: $[-15, 15]$

Ejemplo 1:

Tipo de Estrategia=$(m+l)$ Número de iteraciones=33 Tiempo de CPU=22.6 seg.

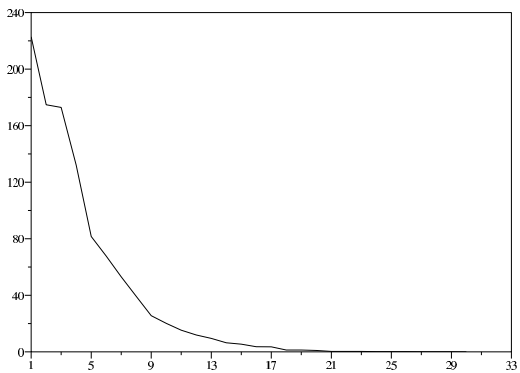

Figura 4.11: Evolución de la mejor aptitud $F1 - ES(m+l)$

Tipo de Estrategia=(m, l) Número de iteraciones=33 Tiempo de CPU=21.4 seg.

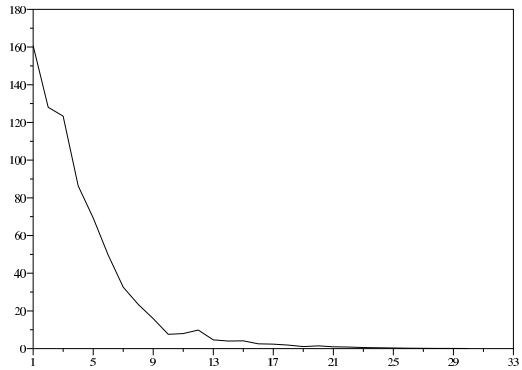

Figura 4.12: Evolución de la mejor aptitud $F1 - ES(m, l)$

Ejemplo 2:

Tipo de Estrategia=$(m + l)$ Número de iteraciones=41 Tiempo de CPU=42.4 seg.

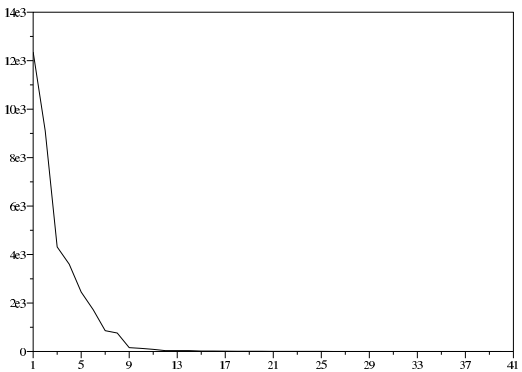

Figura 4.13: Evolución de la mejor aptitud $F2 - ES(m + l)$

Tipo de Estrategia =(m, l) Número de iteraciones =50 Tiempo de CPU =50 seg.

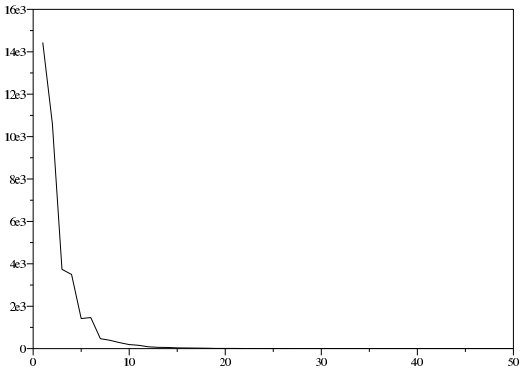

Figura 4.14: Evolución de la mejor aptitud $F2 - ES(m, l)$

Ejemplo 3:

Tipo de Estrategia=$(m + l)$ Número de iteraciones=50 Tiempo de CPU=37.9 seg.

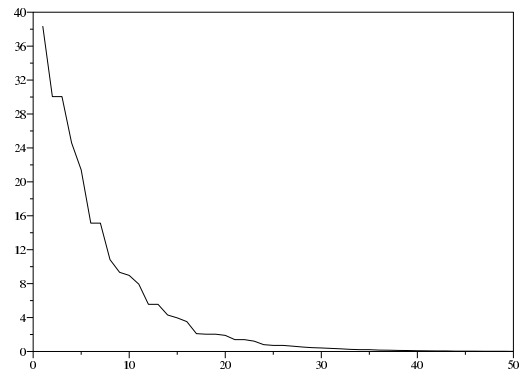

Figura 4.15: Evolución de la mejor aptitud $F3 - ES(m + l)$

Tipo de Estrategia =(m, l) Número de iteraciones =50 Tiempo de CPU =35.8 seg.

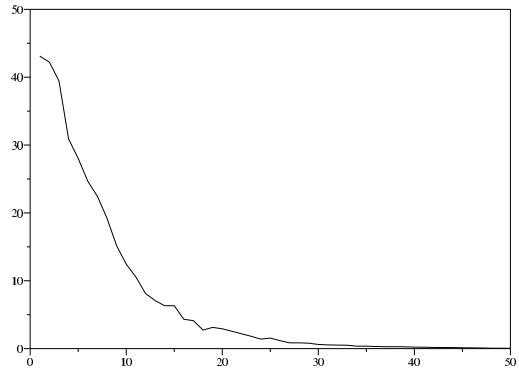

Figura 4.16: Evolución de la mejor aptitud $F3 - ES(m, l)$

Ejemplo 4:

Tipo de Estrategia=$(m + l)$ Número de iteraciones=50 Tiempo de CPU=48.6 seg.

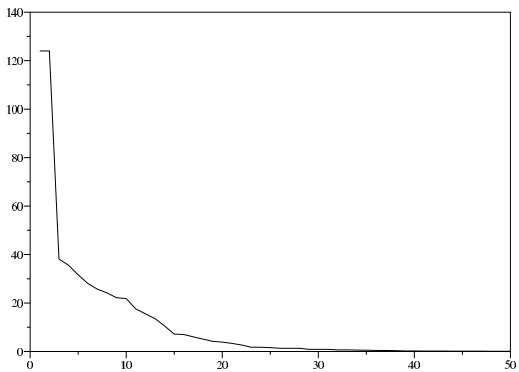

Figura 4.17: Evolución de la mejor aptitud $F4 - ES(m + l)$

Tipo de Estrategia =(m, l) Número de iteraciones =60 Tiempo de CPU =49.7 seg.

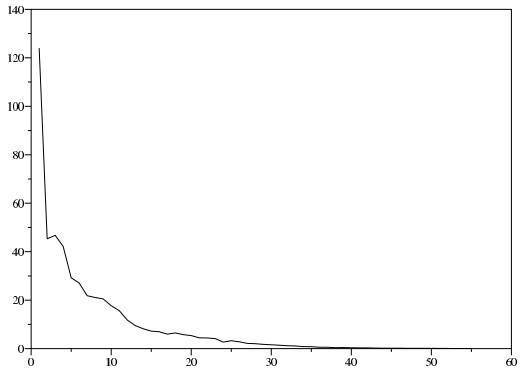

Figura 4.18: Evolución de la mejor aptitud $F4 - ES(m, l)$

Ejemplo 5:

Tipo de Estrategia=$(m+l)$ Número de iteraciones=37 Tiempo de CPU=26.4seg.

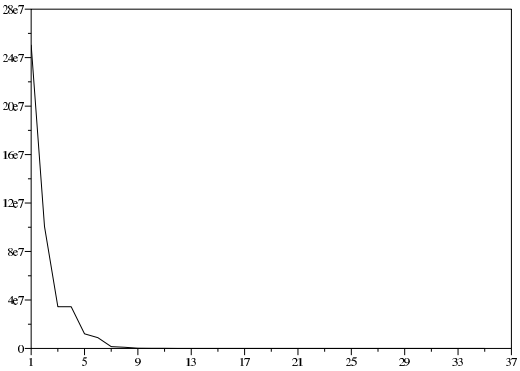

Figura 4.19: Evolución de la mejor aptitud $F5 - ES(m + l)$

Tipo de Estrategia =(m, l) Número de iteraciones =37 Tiempo de CPU =25 seg.

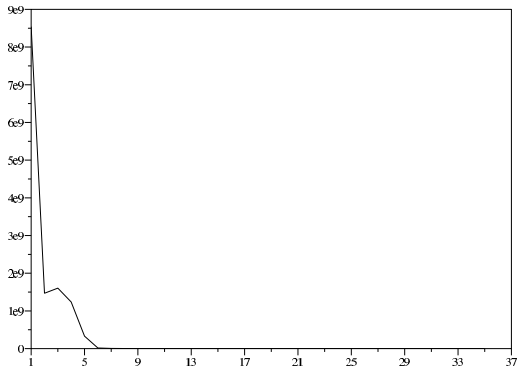

Figura 4.20: Evolución de la mejor aptitud $F5 - ES(m, l)$

4.1.3. Resultados utilizando C++

En las siguientes gráficas se muestran los resultados obtenidos por la implementación de Estrategias Evolutivas en C++, para encontrar el mínimo global de las Funciones Objetivo descritas anteriormente. La optimización de cada Función Objetivo se realizo usando los dos tipos de Estrategias Evolutivas $(m + l)$ y (m, l) . Las gráficas muestran como cambia el valor de aptitud del mejor individuo de la población (eje vertical), con respecto al número de generación o iteración en el que esta la población (eje horizontal).

Este algoritmo encontró satisfactoriamente el mínimo global de todos los ejemplos presentados y los parámetros que fueron constantes durante estas pruebas son:

- Número de padres = 70
- Número de hijos = 490
- Número de variables (dimensión) de la Función Objetivo = 10

Los individuos de la población inicial que se usaron en los ejemplos fueron generados aleatoriamente dentro del intervalo: $[-15, 15]$

Obsérvese que el tiempo de CPU usado para encontrar el mínimo global en todos los ejemplos, fue mucho menor que el tiempo requerido por la implementación en Scilab. Debido principalmente a que Scilab es un software de propósito general y C++ permite desarrollar programas ("a la medida") de propósito específico.

Ejemplo 1:

Tipo de Estrategia=$(m+l)$ Número de iteraciones=33 Tiempo de CPU=1.7 seg.

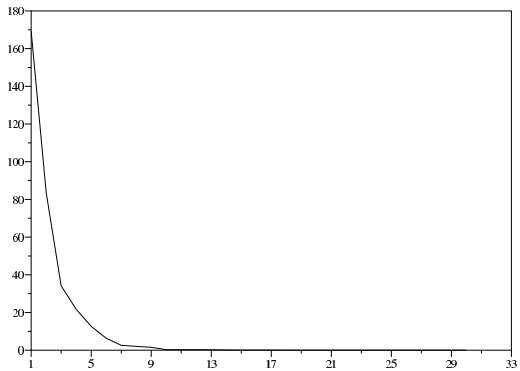

Figura 4.21: Evolución de la mejor aptitud $FC1 - ES(m+l)$

Tipo de Estrategia =(m,l) Número de iteraciones =33 Tiempo de CPU =1.5 seg.

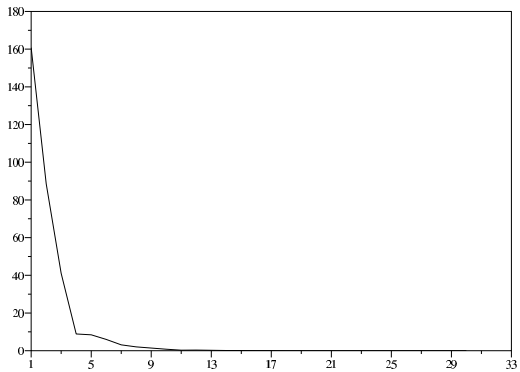

Figura 4.22: Evolución de la mejor aptitud $FC1 - ES(m,l)$

Ejemplo 2:

Tipo de Estrategia=$(m+l)$ Número de iteraciones=41 Tiempo de CPU=2.4 seg.

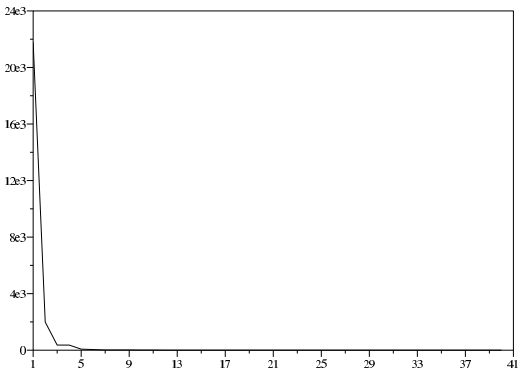

Figura 4.23: Evolución de la mejor aptitud $FC2 - ES(m+l)$

Tipo de Estrategia =(m,l) Número de iteraciones =41 Tiempo de CPU =2.1 seg.

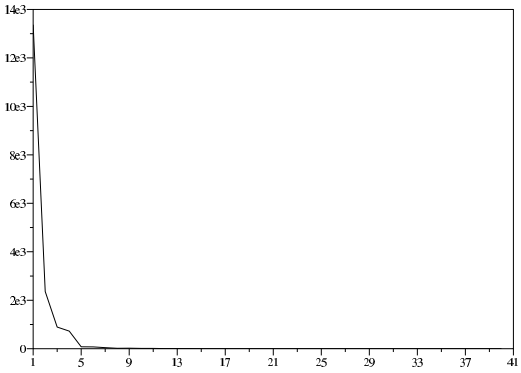

Figura 4.24: Evolución de la mejor aptitud $FC2 - ES(m,l)$

Ejemplo 3:

Tipo de Estrategia=$(m+l)$ Número de iteraciones=25 Tiempo de CPU=1.2 seg.

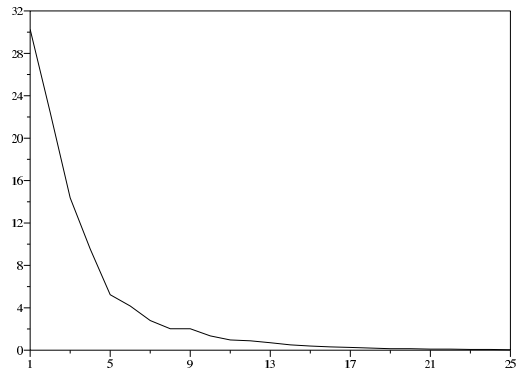

Figura 4.25: Evolución de la mejor aptitud $FC3 - ES(m+l)$

Tipo de Estrategia =(m,l) Número de iteraciones =20 Tiempo de CPU =1.1 seg.

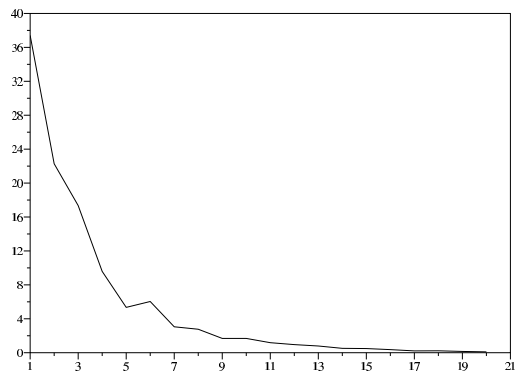

Figura 4.26: Evolución de la mejor aptitud $FC3 - ES(m,l)$

Ejemplo 4:

Tipo de Estrategia=$(m + l)$ Número de iteraciones=30 Tiempo de CPU=1.7

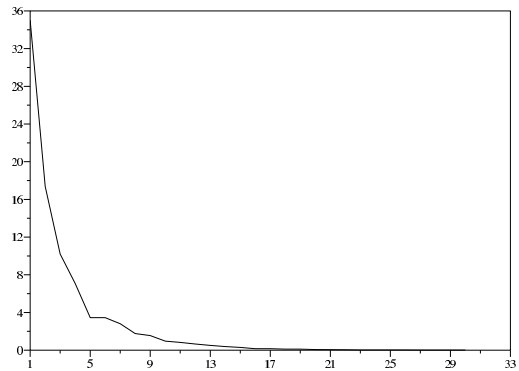

Figura 4.27: Evolución de la mejor aptitud $FC4 - ES(m + l)$

Tipo de Estrategia =(m, l) Número de iteraciones =20 Tiempo de CPU =1.1 seg

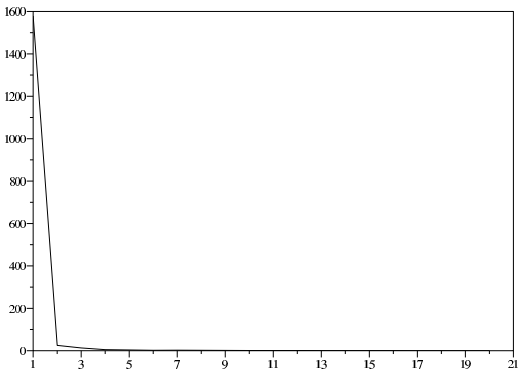

Figura 4.28: Evolución de la mejor aptitud $FC4 - ES(m, l)$

Ejemplo 5:

Tipo de Estrategia=$(m+l)$ Número de iteraciones=20 Tiempo de CPU=1.3

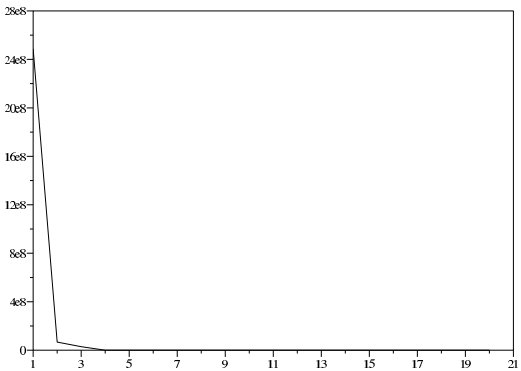

Figura 4.29: Evolución de la mejor aptitud $FC5 - ES(m+l)$

Tipo de Estrategia =(m,l) Número de iteraciones =20 Tiempo de CPU =1.1 seg

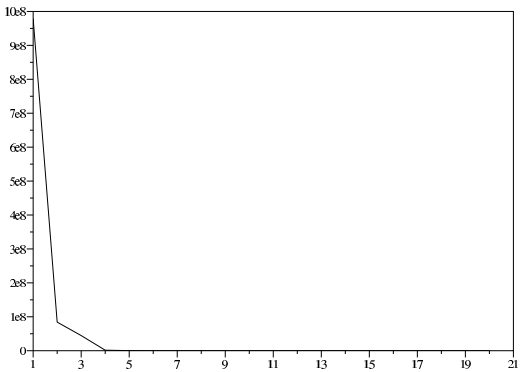

Figura 4.30: Evolución de la mejor aptitud $FC5 - ES(m,l)$

4.2. Reducción del espacio de muestreo para Redes Neuronales utilizando Estrategias Evolutivas

4.2.1. Introducción

Una de las áreas mas importantes de la Inteligencia Artificial es el Reconocimiento de Patrones, su estudio actualmente se enfoca en encontrar nuevas técnicas que permitan mejorar el desempeño de las arquitecturas de estos sistemas; debido principalmente a la complejidad de los Patrones y al número de elementos que son analizados.

Por lo cual en esta tesis se desarrolla una técnica para encontrar diferentes distribuciones de puntos bidimensionales, o áreas de muestreo de datos, para diferentes tipos de Patrones (estímulos) en 2D. Estas distribuciones de puntos de muestreo pueden ser utilizadas en diferentes tipos de redes neuronales, debido a que son usadas en la primera capa de una red neuronal artificial; y por lo tanto no afectan el desempeño de esta. Esta técnica permite reducir el número de elementos a ser procesados y el tiempo de convergencia durante la fase de aprendizaje de una red neuronal artificial. La estructura de la red neuronal puede ser sintetizada automáticamente, de tal forma que los nodos no usados durante la etapa de aprendizaje pueden ser eliminados de la red neuronal.

El sistema para la reducción del espacio de muestreo, utiliza el algoritmo evolutivo conocido como Estrategias Evolutivas; el cual fue desarrollado por Rechenberg [1, 7, 8, 6, 5]. Con este algoritmo es posible, obtener una distribución única de áreas (puntos) de muestreo para cada clase de estimulo. La áreas de muestreo de datos obtenidas para cada clase de estimulo, son las mas significativas y características; con respecto al conjunto de elementos que existan en esa misma clase.

Este sistema esta basado en trabajos realizados anteriormente con mis colegas [10, 11, 12, 13, 14, 15, 16]; en los cuales se muestra la importancia del procesamiento de información en la primera capa de las redes neuronales artificiales.

4.2.2. Estructura y Procedimiento

La primera capa de una red neuronal artificial es considerada usualmente como pasiva, sin embargo, las investigaciones realizadas en el área de neurofisiología muestran que los receptores en la retina tienen gran actividad en el procesamiento de la información. La teoría de percepción de la Gestalt da gran importancia a la composición de estímulos, de tal forma que los estímulos son entendidos como una combinación global de la información de la figura así como del fondo de la misma. Considerando estas ideas, con este ejemplo (sobre el uso de Estrategias Evolutivas) se describe una técnica nueva que permite:

1. Obtener las zonas o regiones de los estímulos, que son mas representativas para un determinado tipo de Patrón bidimensional (Clase). Con el fin de ser usadas en la primera capa de una red neuronal artificial.

2. Obtener información sobre las características globales de los diferentes tipos de estímulos, en función de los resultados obtenidos para cada clase.

La eficiencia de las Estrategias Evolutivas, para resolver un problema, depende de la adecuada selección o diseño de la estrategia evolutiva. Para este caso la estrategia evolutiva seleccionada es del tipo $(m+l)-ES$ con mutaciones simples. La cual de forma general a continuación se describe:

> Todos los individuos de la población "A" son combinados, de tal forma que se genera una nueva población "B". Cada uno de los individuos de la población B es mutado, decodificado y se le asigna su valor de aptitud. Ambas poblaciones se unen (A u B), y los individuos que tienen mayor valor de aptitud son seleccionados. Estos individuos remplazan a los elementos de la población A y este proceso se repite, hasta que se finaliza con el número de iteraciones preestablecido o el umbral de error es alcanzado.

Con el fin de evitar los mínimos locales (soluciones parciales) asociados a este problema. El procedimiento de la estrategia evolutiva $(m+l)-ES$ es modificado de la siguiente forma:

> Si el valor de aptitud de los individuos mas eficientes de la población no cambia durante "g" iteraciones consecutivas (en este caso g=10), entonces un porcentaje de los individuos de la población es mutado (10 %). Los individuos que son mutados son seleccionados aleatoriamente, y la información de cada uno de estos individuos es aleatoria. Esta mutación hace heterogénea la población, para que puedan explorar otras zonas del espacio solución. De tal forma que los individuos de la población pueden escapar de los mínimos locales; y en sucesivas iteraciones toda la población.

4.2.3. Función Objetivo

La aptitud de cada individuo de la población es determinada usando la siguiente expresión:

$$A^K = \frac{1}{\gamma}\left[\gamma - \alpha B_J^K - \lambda C^K\right]$$

$$\gamma = \alpha + \lambda$$

$$\alpha, \lambda \in \Re^{+}$$

$$A^{K} \in [0,1]$$

El valor de la aptitud A^K esta en función de la aptitudes parciales B_J^K y C^K. Donde α y λ son coeficientes usados para determinar la influencia (el peso), que tiene cada una de las aptitudes parciales sobre el valor de la aptitud final A^K.

El valor de la aptitud parcial B_J^K , del individuo K ; se define como:

$$B_J^K = \frac{1}{N_J} \sum_{I=1}^{N_J} S_{IJ}^K$$

$$B_J^K \in [0,1]$$

donde N_J es el número total de Patrones que existen en la clase J.

La similaridad entre el Patrón de entrada y el individuo K se obtiene con:

$$S_{IJ}^K = \frac{1}{W_J^K} \sum_{L=1}^{W_J^K} [D_{KJL}(x,y) \overline{xor} P_{IJL}(x,y)]$$

$$S_{IJ}^K \in [0,1]$$

donde:

W_J^K **es el número de puntos de muestreo que tiene el individuo K.**

P_{IJL} **es el valor del Patrón** I **perteneciente a la clase** J **, en la posición (x,y).**

D_{KJL} **es el valor del punto de muestreo del individuo** K **en la posición (x,y).**

El valor de la aptitud parcial C^K **, del individuo** K **; se define como:**

$$C^K = 1 - \frac{|T - H|}{M}$$

$$T, H, M \in \mathbb{N}$$

$$H \in [0, M]$$

$$T \in [1, M]$$

$$C^K \in [0, 1]$$

donde:

T **es el número de puntos de muestreo que se quiere obtener.**

M **es el máximo número de puntos que el individuo** K **puede tener.**

H**es el número de puntos de muestreo que tiene el individuo** K**.**

La expresión para calcular la aptitud de cada individuo (A^K)**, busca obtener las zonas de muestreo que permitan caracterizar una clase de forma única, con respecto a las otras clases usadas en el proceso de optimización. Y además satisfacer la condición referente al número de puntos de muestreo deseados.**

4.2.4. Experimentos y Resultados

Para poder obtener los resultados descritos previamente, la Estrategia Evolutiva utilizo un conjunto de 480 Patrones en la Función Objetivo; cada uno de ellos formado por una matriz de 10x10 pixels. Cada uno de estos Patrones contenía una de las cuatro letras previamente definidas como clases (O,Q,G,C). Cada una de las letras tuvo uno de los tres grados de distorsión, seleccionados de forma aleatoria, por lo que 30, 40 y 60 % de los pixels fueron alterados.

La Estrategia Evolutiva obtuvo un modelo representativo de cada uno de los cuatro tipos de estímulos. El conjunto de modelos representativos pudo clasificar exitosamente los diferentes tipos de estímulos en su correspondiente clase. Además el conjunto de 480 patrones que se utilizo en la etapa de clasificación, es diferente al usado en la Función Objetivo. Los estímulos en la etapa de clasificación se seleccionaron aleatoriamente y se obtuvo un error porcentual bajo.

Preferencia Puntos de Muestreo	Iteraciones	Error (%)	Puntos de Muestreo	Clase
30	100	1.3	31	"O"
		2.1	31	"Q"
		1.6	29	"G"
		1.9	37	"C"
40	59	0.1	39	"O"
	100	4.8	26	"Q"
	56	1.1	39	"G"
	100	1.6	41	"C"
60	61	0.8	58	"O"
	100	6.7	60	"Q"
	100	2.6	57	"G"
	100	3.1	66	"C"

Cuadro 4.1: Desempeño de las Estrategias Evolutivas en la optimización de zonas de muestreo.

El software desarrollado para esta tarea permitió especificar, el número de puntos de muestreo deseados. Con lo cual se puede parametrizar la representación global de cada clase de Patrón.

Como se muestra en la tabla 4.1 el error porcentual fue bajo; durante la etapa de clasificación de estímulos de las cuatro clases. El error promedio en la clasificación de los Patrones, utilizando representaciones globales cercanas a los 30, 40 y 60 puntos fue de: 1.7, 1.9 y 3.3 respectivamente.

Los resultados muestran un buen desempeño alcanzado por la Estrategia Evolutiva, para crear prototipos o esquemas que representan a un conjunto de estímulos, aun en tareas confusas como esta.

Tener una técnica capaz de seleccionar y optimizar las zonas mas importantes de información, puede permitir sintetizar redes neuronales artificiales de forma eficiente. Esto se debe a las características globales de los Patrones, las cuales determinan la configuración de la capa de entrada de la red neuronal artificial.

4.2.5. Conclusiones

La teoría de percepción de la Gestalt muestra que el ser humano clasifica patrones utilizando información parcial, tiende a completar patrones y a diferenciar las figuras del fondo, buscado los elementos de información mas importantes y significativos que le permiten no solo clasificar un objeto característico, sino también toda una diversidad de objetos (objetos pertenecientes a la misma clase). Las configuraciones de puntos o zonas de muestreo, indican que la información así como la no información se complementan formando una sola entidad; lo cual tiene un rol muy importante en el Reconocimiento de Patrones.

Usar estos filtros de información, permite reducir el número de elementos a ser procesados y el tiempo de aprendizaje de una red neuronal artificial. Es posible sintetizar la estructura de una red neuronal artificial, debido a que es posible detectar y eliminar (antes de la fase de aprendizaje de la red neuronal artificial) los nodos de la red que no son requeridos. La técnica descrita en esta tesis fue desarrollada para ser usada durante la fase de preprocesamiento de información, de tal forma que no depende de una arquitectura específica de red neuronal artificial. Debido a este hecho, es posible usar esta técnica en diferentes tipos de redes neuronales artificiales y en las tareas que son realizadas por este tipo de redes.

Capítulo 5

Conclusiones

Las Estrategias Evolutivas son algoritmos heurísticos de solución a problemas de optimización de funciones. Se basan en el principio de evolución natural, por lo que se dice que las soluciones evolucionan conforme el algoritmo itera.

La idea principal del funcionamiento de las Estrategias Evolutivas consiste en utilizar sucesivamente un conjunto de operadores básicos aplicados a una población (posibles soluciones al problema de optimización), con el fin de obtener una solución óptima del problema. Ejemplos de estos operadores son: recombinación, mutación y selección.

Usar las Estrategias Evolutivas tiene varias ventajas sobre los métodos numéricos clásicos, que son utilizados en la optimización de funciones analíticas. Las estrategias evolutivas son una valiosa herramienta para resolver problemas de optimización complejos no-lineales, debido a que la definición y el potencial del algoritmo no depende de las limitaciones de la matemática determinística.

Se presentaron varios ejemplos, en los que se mostró la facilidad con la que se implementa la Función Objetivo de cada problema. Y se desarrollo una técnica nueva para la reducción del espacio de muestreo en Redes Neuronales utilizando Estrategias Evolutivas.

El fin de implementar las Estrategias Evolutivas en dos lenguajes de programación, es permitir al usuario tener la posibilidad de escoger la opción que mas se adapte a las necesidades del problema de optimización a resolver. Además los algoritmos presentados, pueden ser utilizados en computadoras de propósito general. Existen todavía muchas vertientes en la investigación sobre Estrategias Evolutivas, ejemplos de estas son:

- Caracterización del espacio solución para clases de problemas.
- Nuevas estrategias de optimización para poblaciones múltiples en interacción.

Capítulo 6

Apéndice A

6.1. Programas en Scilab

Evol_Str Toolbox. Optimisation Algorithm.

Archivo: evol_str.sci

```
function [solution,solution_steps,solution_angles,solution_fitness,
fitness_history,cpu_t]=evol_str(str_type,n_parents,size_parent,n_sons,
correlation_flag,num_iter,convergence_epsilon)
//
// Calling Sequence:
//
```

```
// [solution,solution_steps,solution_angles,solution_fitness,fitness_history,
cpu_t]=evol_str()
//
// [solution,solution_steps,solution_angles,solution_fitness,fitness_history,
cpu_t]=evol_str(str_type,n_parents,size_parent,n_sons,correlation_flag,
num_iter,convergence_epsilon)
//
//
// [solution,solution_steps,solution_angles,solution_fitness,fitness_history,
cpu_t]=evol_str(str_type,n_parents,size_parent,n_sons,correlation_flag,
num_iter)
//
// example:
//
//[solution,solution_steps,solution_angles,solution_fitness,fitness_history,
cpu_t]=evol_str(1,7,10,49,'n',500,0.001)
//
// Where :
// solution_steps and solution_angles are strategy paremeters for the solution.
// cpu_t is the total cpu time.
//
// the convergence criterion in each iteration is below the condition:
//
// sum |current_worst_individual-current_best_individual| <= convergence_epsilon
//
//
// Evolution-Strategy (ES)
//
// The ES is based on Darwinian evolutionary theory. With this optimization
// procedure is possible find solutions that minimize function values. The
// function will be programing by the user with the name asig_apt.sci
// ejem : asig_apt(population)
// where: the matrix "population" is the set of solutions (individuals) to the prob-
lem.
```

```
//
//
// The (M+L) strategy join Parents' and Sons' population in the selection process.
// The (M,L) strategy only use the Sons' population in the selection process.
//
// Description of the Program :
//
// Files description:
// evol_str.sci - Main program to start ES.
// recombin.sci - Discrete recombination method.
// mutation.sci - Algorithm for the mutation.
// selectio.sci - Method to choose the best individuals.
// asig_apt.sci - Objective Function defined by the user.
// conver_c.sci - Method to check convergence criterion.
// disturb.sci - Mutation algorithm with enlarged step sizes.
// rotate_d.sci - Algorithm to calculate the rotation matrix.
//
// Version : 1.0
// Date : 23-January-1998
// Author : R.A. S-GUZMAN. University of Mexico (UNAM)
// Facultad de Ingenieria (FI-DIE); LINDA - GROUP.
// e-mail : rodolfo@grupolinda.org
// begin initialisation
// initial parameters
[nargout,nargin]=argn(0);
lines( %inf,72) // adjust display output; no "more" pause
format("v",10); // sets printing format
if nargin<6
disp('Evolution-Strategy (M,L)-ES and (M+L)-ES ');
disp(' ');
disp('The (M+L) strategy join Parents' and Sons' population in the selection process.');
disp('The (M,L) strategy only use the Sons' population in the selection process.');
disp('What kind strategy will be used ( (M,L)=0 or (M+L)=1 ) ? ');
```

```
str_type= read( %io(1),1,1);
disp('Enter number of parents ');
n_parents = read( %io(1),1,1);
disp('Enter the variables by parent ');
size_parent = read( %io(1),1,1);
disp('Enter number of sons ');
n_sons = read( %io(1),1,1);
disp('Correlated Mutations (y/n)? ');
correlation_flag = read( %io(1),1,1,'(a)');
disp('Iteration number ? ');
num_iter = read( %io(1),1,1);
disp('If the population is homogeneous. ');disp('Do you want apply mutation process to the population');
disp('on the basis of the enlarged step constant (y/n)? ');
flag_s = read( %io(1),1,1,'(a)');
if flag_s=='y'
disp('Enter epsilon of convergence (example: 0.01) ');
convergence_epsilon = read( %io(1),1,1);
end;
end; // user's inputs parameters
if n_sons<n_parents
disp();disp('Error : The number of Sons should be greater or equal than Parents number....');
abort
end;
if exists('convergence_epsilon')==0
convergence_epsilon=[];
end;
if correlation_flag=='y' // define the number of angles of each individual.
n_angles = (size_parent-size_parent/2)*(size_parent-1);
else
n_angles = 1;
end;
ini_val=str2code(unix_g('date'));
ini_seed=sum(ini_val(find(ini_val>=0 & ini_val<=9)));
```

```
rand('seed',ini_seed); //sets it to a different value each time.
rand('uniform');
actual_pob = rand(n_parents,size_parent)*30-15; //initialisation of population [-
15,15] and
steps_pob = rand(n_parents,size_parent); // strategy parameters with an
angles_pob = rand(n_parents,n_angles); // uniform distribution (Parents).
// clean variables stage
new_pob=[]; // initialisation of population and
new_steps_pob=[]; // strategy parameters for the Sons.
new_angles_pob=[];
fitness_pob=[]; // fitness values of the population.
best_fitness_pob=[]; // the best fitness value of the current population.
current_best_individual=[];
fitness_history=[];
solution_fitness= %inf; // for the first iteration it supposed to be the worst.
solution=[]; // solution to the problem.
solution_steps=[]; // strategy parameters for
solution_angles=[]; // the solution.
cpu_t=0; // total CPU time.
flag_s=[]; // flag stop
// end clean variables stage
fitness_pob=asig_apt(actual_pob); // evaluate initial population
// display statistics of initial population.
mprintf('Best_fitness %f Average-fitness %f \n',min(fitness_pob),
sum(fitness_pob)/n_parents);
// end initialisation
// Iterative process
timer();// reset timer
for iteration=1:num_iter,
// generate the sons and store it in the new population.
new_pob=[]; // clean the variables of new population (sons).
new_steps_pob=[];
new_angles_pob=[];
fitness_pob=[];
```

```
[new_pob,new_steps_pob,new_angles_pob]=recombin(actual_pob,
steps_pob,angles_pob,n_sons); // recombination process.
[new_pob,new_steps_pob,new_angles_pob]=mutation(new_pob,
new_steps_pob,new_angles_pob); // mutation process.
fitness_pob=asig_apt(new_pob); // put fitness value to new population (sons).
// end generate the sons and store it in the new population.
// check for join parents population with sons population.
if str_type==1
new_pob(n_sons+1:n_sons+n_parents,:)=actual_pob;
new_steps_pob(n_sons+1:n_sons+n_parents,:)=steps_pob;
new_angles_pob(n_sons+1:n_sons+n_parents,:)=angles_pob;
fitness_pob(n_sons+1:n_sons+n_parents,:)=asig_apt(actual_pob); // put fitness
value to current population (parents).
end;
// end check for join parents population with sons population.
// selection stage; the population for the next iteration its obtained.
[actual_pob,steps_pob,angles_pob,fitness_pob]=selectio(new_pob,
new_steps_pob,new_angles_pob,fitness_pob,n_parents);
// end selection stage; the population for the next iteration its obtained.
// actual solution stage
best_fitness_pob=fitness_pob(n_parents); // the best individual of the population
is obtained.
current_best_individual=n_parents;
if best_fitness_pob<solution_fitness // the best current solution in the
solution_fitness=best_fitness_pob; // iterative process its always stored.
solution=actual_pob(current_best_individual,:);
solution_steps=steps_pob(current_best_individual,:);
solution_angles=angles_pob(current_best_individual,:);
end;
// end actual solution stage
// check convergence criterion.
worst_fitness_pob=fitness_pob(1); // It's the worst individual of the population.
current_worst_individual=1;
flag_s=conver_c(actual_pob(current_worst_individual,:),
actual_pob(current_best_individual,:),best_fitness_pob,
```

```
convergence_epsilon);
if flag_s==’s’
break; // the program stop.
end;
if flag_s==’d’ // the iterative process to be continued.
disp(’The population is homogeneous ’);
[actual_pob,steps_pob,angles_pob]=disturb(actual_pob,steps_pob,
angles_pob);
end;
// end check convergence criterion.
// statistics
fitness_history(iteration)=best_fitness_pob; // store the best fitness at
// each iteration.
// display statistics of the current population.
mprintf(’Iteration %d Best_quality %f Average-fitness %f \n’,iteration,
best_fitness_pob,sum(fitness_pob)/n_parents);
// end statistics
end; // for iteration
cpu_t=timer(); // get the total CPU time.
// end of iterative process
```

Archivo: recombin.sci

```
function [new_pob,new_steps,new_angles]=recombin(actual_pob,
steps_pob,angles_pob,n_sons)
//
//[new_pob,new_steps,new_angles]=recombin(actual_pob,steps_pob,
angles_pob,n_sons)
//
//A new population is get by the recombination of the actual population.
//Discrete global recombination is used.
//
//new_pob,new_steps,new_angles are Real matrix.
//actual_pob,steps_pob,angles_pob are Real matrix.
[n_parents,size_son]=size(actual_pob);
```

```
size_angles=size(angles_pob,2):// the size of strategy angles vector is get.
// discrete global recombination stage
for cont=1:n_sons,
new_pob(cont,1:size_son)=actual_pob( (n_parents-1)*rand()+1,1:size_son);
new_steps(cont,1:size_son)=steps_pob( (n_parents-1)*rand()+1,1:size_son);
new_angles(cont,1:size_angles)=angles_pob( (n_parents-1)*rand()+1,
1:size_angles);
end
// end discrete global recombination stage
```

Archivo: mutation.sci

```
function [new_pob_out,new_steps_out,new_angles_out]=mutation(new_pob,
new_steps,new_angles)
//
// [new_pob_out,new_steps_out,new_angles_out]=mutation(new_pob,
new_steps,new_angles)
//
// Mutation algorithm for disturb the population.
// The input and output parameters are matrix.
[n_sons,size_son]=size(new_pob);
size_angles=size(new_angles,2);
// parameters for mutation.
standard_deviation_0=1/((2*size_son).^(1/2));
standard_deviation_var=1/(2*((size_son).^(1/4)));
angle_standard_deviation=0.0873; // 5 grades aprox.
r_0=standard_deviation_0*rand(n_sons,1,'normal');
for m=1:n_sons,
for n=1:size_son,
z_0(m,n)=r_0(m,1);
end;
end;
// end parameters for mutation.
// modification to steps strategy parameters.
z_i=standard_deviation_var*rand(n_sons,size_son,'normal');
```

```
new_steps_out=new_steps.*exp(z_i+z_0);
// it prevent steps sizes=0
index=find(new_steps_out<=0);
size_row=size(index,1);
new_steps_out(index)=rand(size_row,1);
if size_row~=0
disp('there was a step size=0');
end;
// end it prevent steps sizes=0
// end modification to steps strategy parameters.
// modification to angles strategy parameters.
z_angles=angle_standard_deviation*rand(n_sons,size_angles,
'normal'); //N(0,0.0076)
new_angles_out=new_angles+z_angles;
// end modification to angles strategy parameters.
// modification to the possible solution .
if size_angles==1 // implicate not correlated mutations
z_sons=new_steps_out.*rand(n_sons,size_son,'normal');
new_pob_out=new_pob+z_sons;
else // implicate correlated mutations
for cont=1:n_sons,
t_matrix=rotate_d(new_angles_out(cont,:),size_son);
vector_son=new_steps_out(cont,:)';// it's only to make operations.
cor(cont,1:size_son)=(t_matrix*vector_son)';
end; //for
new_pob_out=new_pob+cor;
end; // if
// end modification to the possible solution .
```

Archivo: rotate_d.sci

```
function [new_dist]=rotate_d(angles,size_son)
//
// [new_dist]=rotate_d(angles,size_son)
//
```

```
// new_dist is a rotation matrix for the Normal distribution.
// It's used to rotate the Normal distribution in function of angles.
new_dist=eye(size_son,size_son);
for p=1:(size_son-1),
for q=p+1:size_son,
j=0.5*((2*size_son)-p)*(p+1)-(2*size_son)+q;
rotation_j=eye(size_son,size_son);
rotation_j(p,p)=cos(angles(j));
rotation_j(q,q)=cos(angles(j));
rotation_j(p,q)=sin(angles(j));
rotation_j(q,p)=-sin(angles(j));
new_dist=new_dist*rotation_j;
end;
end;
```

Archivo: conver_c.sci

```
function flag_s=conver_c(worst_ind,best_ind,best_fitness_pob,
convergence_epsilon)
//
// flag_s=conver_c(worst_ind,best_ind,best_fitness_pob,
convergence_epsilon)
//
// This function check convergence criterion below the condition:
// sum |current_worst_individual-current_best_individual| <=
convergence_epsilon
//
// flag_s is a string
// worst_ind and best_ind are Real vectors.
// convergence_epsilon and best_fitness_pob are Real values.
flag_s=[]; // it's use to avoid loops, when flag_s='d' arise.
if best_fitness_pob==0 // if the problem is resolved the algorithm stop.
flag_s='s';
disp(' ');
disp('The solution was found');
```

```
end;
if convergence_epsilon<>[] // check convergence criterion
if sqrt(sum ((worst_ind-best_ind).^2) ) <=convergence_epsilon
flag_s='d';
end;
end;
```

Archivo: disturb.sci

```
function [new_pob,new_steps,new_angles]=disturb(actual_pob,steps_pob,
angles_pob)
//
//[new_pob,new_steps,new_angles]=disturb(actual_pob,steps_pob,
angles_pob)
//
// The input and output parameters are matrix.
// This function apply mutation process to the population on the
basis of the enlarged step constant.
// Note: The enlarged step constant = random number of 2 throught 100.
[n_indi,size_indi]=size(actual_pob);
size_angles=size(angles_pob,2);
esc=(100-2)*rand()+2; // random number of 2 throught 100.
// esc=enlarged step constant.
new_steps=esc*steps_pob;// it's the standard deviation for all Population.
new_angles=esc*angles_pob;
if size_angles==1
new_pob=actual_pob+(new_steps.*rand(n_indi,size_indi,'normal'));
else // this is the mutation when there is angles strategy parameters.
for cont=1:n_indi,
t_matrix=rotate_d(new_angles(cont,:),size_indi);
cor(cont,1:size_indi)=(t_matrix*(new_steps(cont,:)'))';
end; //for
new_pob=actual_pob+cor;
end; // end if
```

Archivo: selectio.sci

```
function [actual_pob,steps_pob,angles_pob,fitness_pob]=selectio(new_pob,
new_steps_pob,new_angles_pob,fitness,n_parents)
//
//[actual_pob,steps_pob,angles_pob,fitness_pob]=selectio(new_pob,
new_steps_pob,new_angles_pob,fitness,n_parents)
//
// The best vectors of new_pob,new_steps_pob and new_angles_pob are select-
ed.
// The input and output paremeters are matrix.
n_vars=size(new_pob,2);
n_col=size(new_angles_pob,2);
[apts,index]=sort(fitness);
s_ix=size(index,1);
index=index((s_ix-n_parents)+1:s_ix);
apts=apts((s_ix-n_parents)+1:s_ix);
// The row number of outputs matrix is n_parents.
for parents=n_parents:-1:1, // This meaning that the population is always con-
stant.
actual_pob(parents,1:n_vars)=new_pob(index(parents),1:n_vars);
steps_pob(parents,1:n_vars)=new_steps_pob(index(parents),1:n_vars);
angles_pob(parents,1:n_col)=new_angles_pob(index(parents),1:n_col);
fitness_pob(parents,1)=apts(parents);
end;
```

Archivo: asig_apt.sci

```
function fitness_values=asig_apt(population)
// fitness_values=asig_apt(population)
// The input and output parameters are matrix.
[row,col]=size(population);
fitness_values=zeros(row,1);
for cont=1:row,
for Var=1:col,
H1=((population(cont,1)-(population(cont,Var).^2)).^2)+
```

```
((1-population(cont,Var)).^2);
fitness_values(cont,1)=fitness_values(cont,1)+H1;
end;
end;
```

Archivo: asig_apt.cat

NAME

asig_apt - objetive function defined by the user (Evolution Strategy)

CALLING SEQUENCE

fitness_values=asig_apt(population)

DESCRIPTION

The input and output parameters are matrix.

Archivo: conver_c.cat

NAME

conver_c - check convergence criterion (Evolution Strategy)

CALLING SEQUENCE

flag_s=conver_c(worst_ind,best_ind,best_fitness_pob,convergence_epsilon)

DESCRIPTION

This function check convergence criterion below the condition:

sum |current_worst_individual-current_best_individual| <= convergence_epsilon

flag_s is a string

worst_ind and best_ind are Real vectors.

convergence_epsilon and best_fitness_pob are Real values.

Archivo: disturb.cat

NAME

disturb - mutation algorithm with enlarged step sizes (Evolution Strategy)

CALLING SEQUENCE

[new_pob,new_steps,new_angles]=disturb(actual_pob,steps_pob,angles_pob)

DESCRIPTION

This function apply mutation process to the population on the basis of the enlarged step constant.

The input and output parameters are Real matrix.

Note: The enlarged step constant = random number of 2 throught 100

Archivo: evol_str.cat

NAME
evol_str - Optimization Method (Evolution Strategy)
Main Program.
CALLING SEQUENCE
[solution,solution_steps,solution_angles,solution_fitness,fitness_history,cpu_t]=
evol_str()
[solution,solution_steps,solution_angles,solution_fitness,fitness_history,cpu_t]=
evol_str(str_type,n_parents,size_parent,n_sons,correlation_flag,num_iter,
convergence_epsilon)
[solution,solution_steps,solution_angles,solution_fitness,fitness_history,cpu_t]=
evol_str(str_type,n_parents,size_parent,n_sons,correlation_flag,num_iter)
example:
[solution,solution_steps,solution_angles,solution_fitness,fitness_history,cpu_t]=
evol_str(1,7,10,49,'n',500,0.001)
DESCRIPTION
Evolution-Strategy (ES):
The ES is based on Darwinian evolutionary theory. With this optimization
procedure is possible find solutions that minimize function values.
The (M+L) strategy join Parents' and Sons' population in the selection process.
The (M,L) strategy only use the Sons' population in the selection process.
[solution,solution_steps,solution_angles,solution_fitness,fitness_history,cpu_t]=
evol_str()
Where :
solution_steps and solution_angles are strategy paremeters for the solution.
cpu_t is the total cpu time.
The convergence criterion in each iteration is below the condition:
sum|current_worst_individual-current_best_individual|<= convergence_epsilon
The objetive function will be programing by the user with the name asig_apt.sci
ejem : asig_apt(population)
where: the matrix "population" is the set of solutions (individuals) to the problem.
Files description:
evol_str.sci - Main program to start ES.

recombin.sci - Discrete recombination method.

mutation.sci - Algorithm for the mutation.

selectio.sci - Method to choose the best individuals.

asig_apt.sci - Objective Function defined by the user.

conver_c.sci - Method to check convergence criterion.

disturb.sci - Mutation algorithm with enlarged step sizes.

rotate_d.sci - Algorithm to calculate the rotation matrix.

Archivo: mutation.cat

NAME

mutation - mutation algorithm for disturb the population (Evolution Strategy)

CALLING SEQUENCE

[new_pob_out,new_steps_out,new_angles_out]=mutation(new_pob,new_steps, new_angles)

DESCRIPTION

The input and output parameters are Real matrix.

Archivo: recombin.cat

NAME

recombin - discrete recombination method (Evolution Strategy)

CALLING SEQUENCE

[new_pob,new_steps,new_angles]=recombin(actual_pob,steps_pob,angles_pob, n_sons)

DESCRIPTION

A new population is get by the recombination of the actual population.

Discrete global recombination is used.

new_pob,new_steps,new_angles are Real matrix.

actual_pob,steps_pob,angles_pob are Real matrix.

Archivo: rotate_d.cat

NAME

rotate_d - rotation matrix generator (Evolution Strategy)

dCALLING SEQUENCE

[new_dist]=rotate_d(angles,size_son)

DESCRIPTION

It's used to rotate the Normal distribution in function of angles.

new_dist is a rotation Real matrix for the Normal distribution.

angles is Real vector and size_son is Real value.

Archivo: selectio.cat

NAME

selectio - choose the best individuals (Evolution Strategy)

CALLING SEQUENCE

[actual_pob,steps_pob,angles_pob,fitness_pob]=selectio(new_pob,

new_steps_pob,new_angles_pob,fitness,n_parents)

DESCRIPTION

The best vectors of new_pob,new_steps_pob and new_angles_pob are selected.

The input and output paremeters are Real matrix.

Archivo: whatis

asig_apt - objetive function defined by the user

conver_c - check convergence criterion

disturb - mutation algorithm with enlarged step sizes

evol_str - Optimization Method (Evolution Strategy)

mutation - mutation algorithm for disturb the population

recombin - discrete recombination method

rotate_d - rotation matrix generator

selectio - choose the best individuals

Capítulo 7

Apéndice B

7.1. Programas en C++

Estrategia-Evolutiva (EE)

Version 1.0

21-Agosto-1999

Autor: R.A. SANCHEZ-GUZMAN

GRUPO - LINDA

Facultad de Ingenieria (FI-DIE)

Universidad Nacional Autonoma de Mexico (UNAM)

e-mail : rodolfo@grupolinda.org

Est-Evo 1.0. Optimisation Algorithm.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Archivo: Est_Evo.h

```
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <matrix.hpp>// Clase basica para matrices
#include <matrand.hpp>// Clase de generadores de numeros aleatorios para matrices
#include <matmath.hpp>// Clase de funciones transcendentales para matrices
#ifndef CLOCKS_PER_SEC // Se utiliza para obtener el tiempo de CPU en segundos
#define CLOCKS_PER_SEC CLK_TCK
#endif
class Est_Evo
{
protected:
int ES_type;
int nSons;
int nGenerations;
char Pre_File_Name[50];
double CPU_Time; // Tiempo de CPU en segundos.
int Iter_Backup; // especifica que cada Iter_Backup iteraciones se salvara en disco las variables de Est_Evo.
int Act_Iteracion; // contiene la iteracion actual
clock_t T_inicial,T_final,T_inicial_backup; // Estos tiempos se usan para sacar el tiempo de CPU
```

```
matrix Pop;
matrix Steps_Pop;
matrix New_Pop;
matrix New_Steps_Pop;
matrix Fitness_History;
public:
int Limit_Seed();
Est_Evo(const int rows,const int cols,const int ES_form,const int N_Sons,const int Iterations,const char *fName="ES_Results_",const int N_Backup=1);
void recombination();
void mutation();
double Objective(const matrix Individuo); // Funcion objetivo definida por el usuario, calcula la aptitud para un individuo.
matrix Asig_Fitness(); //Obtiene un vector con la aptitud para cada individuo de la matriz Poblacion.
void selection(const matrix Fitness_Values);
void crea_archivo (const char *Sufijo,const matrix Matriz);
void save_results (const char *Sufijo);
matrix lee_archivo (const char *Sufijo);
void load_results (const char *Sufijo);
matrix Saca_Pop();
matrix Saca_Steps_Pop();
matrix Saca_New_Pop();
matrix Saca_New_Steps_Pop();
void Pon_Pop(const matrix Matriz);
void Pon_Steps_Pop(const matrix Matriz);
void evolve();
matrix une_pop();
void una_iteracion();
};
int Est_Evo::Limit_Seed()
//Genera un numero entero aleatorio entre 1 y 30000 para cumplir especificacion.
//Se utiliza para inicializar la semilla del Objeto matRandom.
{
int num;
```

```
num= 1 + rand() % 30000;
return num;
}
Est_Evo::Est_Evo(const int rows,const int cols,const int ES_form,const int N_Sons,
const int Iterations,const char *fName,const int N_Backup): Pop(rows,cols),
Steps_Pop(rows,cols),New_Pop(N_Sons,cols),
New_Steps_Pop(N_Sons,cols),Fitness_History(Iterations,1)
{
srand(time(NULL)); // Inicializa la semilla aleatoriamente
matPrecision(10); // Se establece que el numero de digitos despues del punto
decimal es 10.
ES_type=ES_form;
nSons=N_Sons;
nGenerations=Iterations;
strncpy(Pre_File_Name,fName,49);
Pre_File_Name[50]='\0';
Iter_Backup=N_Backup;
FILE *fp;
char In_File_Name[100];
strcpy(In_File_Name,Pre_File_Name);
strcat(In_File_Name,"pop");
strcat(In_File_Name,".dat");
if((fp=fopen(In_File_Name,"r"))==NULL) //Se intenta leer estado anterior
{
int Val=Limit_Seed(); //Se obtiene valor valido para la semilla
matRandom rando( Val, Val, Val) ;
rando.uniform(Pop);
Pop=(Pop*30)-15;
//Pop.print();
rando.uniform(Steps_Pop);
Act_Iteracion=1;
CPU_Time=0;
}
else
```

```
{
fclose(fp);
load_results("todo");
}
}
void Est_Evo::recombination()
{
int Val=Limit_Seed(); //Se obtiene valor valido para la semilla
matRandom aleto( Val, Val, Val) ;
int Size_Hijo=Pop.nCols();
for ( int i = 1 ; i <= nSons; i++ )
{
for ( int j = 1 ; j <= Size_Hijo; j++ )
{
New_Pop(i,j)=Pop(rint((Pop.nRows()-1)*aleto.uniform()+1),j);
New_Steps_Pop(i,j)=Steps_Pop(rint((Steps_Pop.nRows()-1)*aleto.uniform()+1),j);
} // for j
} // for i
//New_Pop.print();
//New_Steps_Pop.print();
}
void Est_Evo::mutation()
{
// parameters for mutation.
double standard_deviation_0= (float) 1/pow(2*Pop.nCols(),(float) 1/2);
double standard_deviation_var= (float) 1/pow(2*Pop.nCols(),(float) 1/4);
int Size_Hijo=Pop.nCols();
matrix R_0(nSons,1);
matrix Z_0(nSons,Size_Hijo);
int Val=Limit_Seed(); //Se obtiene valor valido para la semilla
matRandom alea( Val, Val, Val) ;
alea.normal(R_0);
R_0*=standard_deviation_0;
for ( int i = 1 ; i <= nSons; i++ )
```

```
{
for ( int j = 1 ; j <= Size_Hijo; j++ )
{
Z_0(i,j)=R_0(i,1);
} // for j
} // for i
// end parameters for mutation.
// modification to steps strategy parameters.
matrix Z_i(nSons,Size_Hijo);
alea.normal(Z_i);
Z_i*=standard_deviation_var;
New_Steps_Pop.multijEq(exp(Z_i+Z_0));
// este ciclo prevee que existan pasos con un valor de cero
for ( int i = 1 ; i <= nSons; i++ )
{
for ( int j = 1 ; j <= Size_Hijo; j++ )
{
if (New_Steps_Pop(i,j)==0) // Si es asi se asigna un numero aleatorio
{ // con distribucion uniforme en [0,1]
New_Steps_Pop(i,j)=alea.uniform();
out.newLine();
out.put("Hubo un paso con valor igual a cero.");
}
} // for j
} // for i
// este ciclo prevee que existan pasos con un valor de cero
// end modification to steps strategy parameters.
// modification to the possible solution .
matrix Z_sons, Norm_Rand(nSons,Size_Hijo);
alea.normal(Norm_Rand);
Z_sons=New_Steps_Pop.multij(Norm_Rand);
New_Pop+=Z_sons;
// end modification to the possible solution .
//New_Pop.print();
```

```
//New_Steps_Pop.print();
}
double Est_Evo::Objective(const matrix Individuo) // Se define la Funcion-Objetivo.
// Este ejemplo implementa una Hiper-Esfera cuya solucion es un vector de ceros.
{
// El parametro de entrada se define como constante, para que no pueda ser mod-
ificado dentro de esta funcion.
double Score=0.0;
Score=Individuo.sumsq();// suma de los cuadrados de los elementos
return Score;
}
matrix Est_Evo::une_pop() //Determina si se unen los padres con los hijos.
{
matrix Poblacion;
if (ES_type==1)
{
Poblacion=Pop|New_Pop; // Concatenacion vertical.
} // if
else
{
Poblacion=New_Pop;
} // else
return Poblacion;
}
void Est_Evo::selection(const matrix Fitness_Values)
// Los parametros de entrada se define como constantes, para que no pueda ser
modificados dentro de esta funcion.
{
matrix Join_P,Join_S_P,Vec;
indexArray Indices;
int Num_Individuos=Pop.nRows();
if (ES_type==1)
{
Join_P=Pop|New_Pop;
Join_S_P=Steps_Pop|New_Steps_Pop;
```

```
Fitness_Values.heapMap(Indices);
//Se tuvo que hacer este "for" porque la funcion miembro rowsOf de la clase "matrix" no sirve.
for ( int j = 1 ; j <= Num_Individuos; j++ )
{
Pop.setRow(j,Vec.rowOf(Join_P,Indices(j)));
Steps_Pop.setRow(j,Vec.rowOf(Join_S_P,Indices(j)));
}
} // if
else
{
Fitness_Values.heapMap(Indices);
//Se tuvo que hacer este "for" porque la funcion miembro rowsOf de la clase "matrix" no sirve.
for ( int j = 1 ; j <= Num_Individuos; j++ )
{
Pop.setRow(j,Vec.rowOf(New_Pop,Indices(j)));
Steps_Pop.setRow(j,Vec.rowOf(New_Steps_Pop,Indices(j)));
}
} // else
// Act_Iteracion es solo un indice, para registrar la historia de la evolucion (es usado en Fitness_History)
Fitness_History(Act_Iteracion,1)=Fitness_Values(Indices(1),1); // Obtiene el mejor valor de aptitud de la Poblacion cada vez que pasa por este metodo.
//cout <<Index(1) <<"\n";
//Vec=Vec.rowOf(New_Pop,Index(1));
//Vec.print("Best");
}
void Est_Evo::crea_archivo (const char *Sufijo,const matrix Matriz)
{
outFile Salida; // Clase para el manejo de archivos de salida.
char Out_File_Name[100];
strcpy(Out_File_Name,Pre_File_Name);
strcat(Out_File_Name,Sufijo);
strcat(Out_File_Name,".dat");
```

```
matFormat(PLAIN);
Salida.open(Out_File_Name);
Salida.putIndex(Matriz.nRows()).putIndex(Matriz.nCols()).newLine();//Se alma-
cenan las dimensiones de la matriz y se hace nueva linea.
Matriz.put(Salida);
Salida.close();// Se cierra el archivo.
}
void Est_Evo::save_results (const char *Sufijo)
{
matrix Matriz;
if (Sufijo=="pop" || Sufijo=="todo")
{
Matriz=Pop;
crea_archivo("pop",Matriz); //Se redefine el sufijo, para poder generar todos los
archivos, cuando Sufijo=todo.
}
if (Sufijo=="stepspop" || Sufijo=="todo")
{
Matriz=Steps_Pop;
crea_archivo("stepspop",Matriz);
}
if (Sufijo=="fitness" || Sufijo=="todo")
{
Matriz=Fitness_History.row(1,Act_Iteracion); // almacena la historia de la apti-
tud para este intervalo de iteraciones.
crea_archivo("fitness",Matriz);
}
if (Sufijo=="solucion" || Sufijo=="todo")
{
Matriz=Pop.row(1);// El indice de la solucion es siempre 1, debido a la forma en
que se lleva a cabo la etapa de seleccion.
crea_archivo("solucion",Matriz);
}
if (Sufijo=="stepsolucion" || Sufijo=="todo")
{
```

```
Matriz=Steps_Pop.row(1);// El indice de los pasos de la solucion es siempre 1, debido a la forma en que se lleva a cabo la etapa de seleccion.
crea_archivo("stepsolucion",Matriz);
}
if (Sufijo=="cpu_t" || Sufijo=="todo")
{
Matriz.reset(1,1); // se redefine el tamanio de la matriz, para que no haya basura de las matrices anteriores.
Matriz(1,1)=CPU_Time;
crea_archivo("cpu_t",Matriz);
}
}
matrix Est_Evo::lee_archivo (const char *Sufijo)
{
inFile Entrada; // Clase para el manejo de archivos de entrada.
char In_File_Name[100];
INDEX filas,columnas;
matrix Matriz;
strcpy(In_File_Name,Pre_File_Name);
strcat(In_File_Name,Sufijo);
strcat(In_File_Name,".dat");
Entrada.open(In_File_Name);
Entrada.getIndex(filas).getIndex(columnas).nextLine(); //Se obtienen las dimensiones de la matriz y se lee la siguiente linea.
Matriz.reset(filas,columnas);
Matriz.get(Entrada);
Entrada.close(); // Se cierra el archivo.
return Matriz;
}
void Est_Evo::load_results (const char *Sufijo)
{
// Los archivos asociados a la Solucion y sus respectivos Pasos no se leen, porque estas variables estan contenidas en los
// archivos de la Poblacion y en el de Pasos.
if (Sufijo=="pop" || Sufijo=="todo")
```

```
{
Pop=lee_archivo("pop");
}
if (Sufijo=="stepspop" || Sufijo=="todo")
{
Steps_Pop=lee_archivo("stepspop");
}
if (Sufijo=="fitness" || Sufijo=="todo")
{
matrix Temp=lee_archivo("fitness");
int N_Filas=Temp.nRows();
Act_Iteracion=N_Filas+1;
int Total_Iter=(Act_Iteracion-1)+nGenerations;
nGenerations=Total_Iter;
Fitness_History.reset(Total_Iter,1);
Fitness_History.setSub(1,N_Filas,1,Temp.nCols(),Temp);
}
if (Sufijo=="cpu_t" || Sufijo=="todo")
{
matrix Matriz=lee_archivo("cpu_t"); //Se usa una matriz temporal, debido al
propotipo de la funcion.
CPU_Time=Matriz(1,1);
}
}
matrix Est_Evo::Saca_Pop ()
{
return Pop;
}
matrix Est_Evo::Saca_Steps_Pop()
{
return Steps_Pop;
}
matrix Est_Evo::Saca_New_Pop()
{
```

```
return New_Pop;
}
matrix Est_Evo::Saca_New_Steps_Pop()
{
return New_Steps_Pop;
}
void Est_Evo::Pon_Pop(const matrix Matriz)
{
Pop=Matriz;
}
void Est_Evo::Pon_Steps_Pop(const matrix Matriz)
{
Steps_Pop=Matriz;
}
void Est_Evo::una_iteracion()
{
recombination();
mutation();
matrix Poblacion=une_pop(); //poblacion actual despues de determinar, si se un-
en los padres con los hijos en la etapa de seleccion.
int Num_Individuos=Poblacion.nRows();
matrix Vec,Fitness_Values(Num_Individuos,1);
for ( int m = 1 ; m <= Num_Individuos; m++ )
{
Fitness_Values(m,1)=Objective(Vec.rowOf(Poblacion,m));
}
selection(Fitness_Values);
out.put("Iteracion:").putReal(Act_Iteracion).put("Aptitud:").
putReal(Fitness_History(Act_Iteracion,1));
out.newLine();
if ((Act_Iteracion % Iter_Backup)==0) // Determina en que iteracion se salvan
las variables.
{
T_final=clock();
```

```
CPU_Time+= (double) (T_final-T_inicial_backup) / CLOCKS_PER_SEC ; // Se saca el tiempo de CPU en segundos para este intervalo de iteraciones .
save_results("todo"); // Salva todas las variables en archivos diferentes.
T_inicial_backup=T_final;// Se actualiza el tiempo inicial, para el siguiente intervalo.
}
}
void Est_Evo::evolve()
{
T_inicial=clock();
T_inicial_backup=T_inicial;
for ( int counter = Act_Iteracion ; counter <= nGenerations; counter++ )
{
Act_Iteracion=counter; // Se actualiza para ser utilizada para almacenar la historia de las aptitudes durante el backup y al final de la corrida.
una_iteracion();
if (Fitness_History(Act_Iteracion,1)==0) // Verifica si la aptitud es cero
{
out.newLine();
out.put("Solucion Encontrada....");
out.newLine();
break;
}
}
T_final=clock();
CPU_Time+= (double) (T_final-T_inicial) / CLOCKS_PER_SEC ; // Se saca el tiempo de CPU en segundos de todas la iteraciones definidas inicialmente.
save_results("todo"); // Salva todas las variables en archivos diferentes.
}
```

Archivo: ex1.cpp

```
#include <iostream.h>
#include <math.h>
#include <matrix.hpp>// Clase basica para matrices
#include <Est_Evo.h>// Clase Estrategia Evolutiva
```

```
// Este ejemplo utiliza como Funcion-Objetivo una Hiper-Esfera cuya solucion
es el vector nulo.
// La cual es la Funcion-Objetivo por Default.
int main()
{
//Est_Evo RA(#padres,#variables,1=m+l 0=m,l , # hijos, iteraciones, prefijo del
archivo, salva variables cada N iteraciones);
Est_Evo RA(70,10,1,490,30,"ex1_",10);
RA.evolve();
return 0;
}
```

Archivo: ex5.cpp

```
#include <iostream.h>
#include <math.h>
#include <matrix.hpp>// Clase basica para matrices
#include <Est_Evo.h>// Clase Estrategia Evolutiva
class Mi_Est_Evo : public Est_Evo
{
protected:
public:
Mi_Est_Evo(const int rows,const int cols,const int ES_form,const int N_Sons,const
int Iterations,const char *fName="ES_Results_",const int N_Backup=1);
void una_iteracion();
void evolve();
double Objective(const matrix Vector); // La solucion de esta funcion es (1,3).
};
Mi_Est_Evo::Mi_Est_Evo(const int rows,const int cols,const int ES_form,const
int N_Sons,const int Iterations,const char *fName,const int N_Backup) :
Est_Evo(rows,cols,ES_form,N_Sons,Iterations,fName,N_Backup)
{
}
void Mi_Est_Evo::una_iteracion()
{
recombination();
```

```
mutation();
matrix Poblacion=une_pop(); //poblacion actual despues de determinar, si se un-
en los padres con los hijos en la etapa de seleccion.
int Num_Individuos=Poblacion.nRows();
matrix Vec,Fitness_Values(Num_Individuos,1);
for ( int m = 1 ; m <= Num_Individuos; m++ )
{
Fitness_Values(m,1)=Objective(Vec.rowOf(Poblacion,m));
}
selection(Fitness_Values);
out.put("Iteracion:").putReal(Act_Iteracion).put("Aptitud:").
putReal(Fitness_History(Act_Iteracion,1));
out.newLine();
if ((Act_Iteracion % Iter_Backup)==0) // Determina en que iteracion se salvan
las variables.
{
T_final=clock();
CPU_Time= (double) (T_final-T_inicial) / CLOCKS_PER_SEC ; // Se saca el
tiempo de CPU en segundos para este intervalo de iteraciones .
save_results("todo"); // Salva todas las variables en archivos diferentes.
}
}
void Mi_Est_Evo::evolve()
{
T_inicial=clock();
for ( int counter = Act_Iteracion ; counter <= nGenerations; counter++ )
{
Act_Iteracion=counter; // Se actualiza para ser utilizada para almacenar la histo-
ria de las aptitudes durante el backup y al final de la corrida.
una_iteracion();
if (Fitness_History(Act_Iteracion,1)==0) // Verifica si la aptitud es cero
{
out.newLine();
out.put("Solucion Encontrada....");
out.newLine();
```

```
break;
}
}
T_final=clock();
CPU_Time+= (double) (T_final-T_inicial) / CLOCKS_PER_SEC ; // Se saca el tiempo de CPU en segundos de todas la iteraciones definidas inicialmente.
save_results("todo"); // Salva todas las variables en archivos diferentes.
}
double Mi_Est_Evo::Objective(const matrix Vector) // La solucion de esta funcion es (1,3).
{
double Score=0.0;
// Vector.print();
for ( INDEX i = 1 ; i <= Vector.nCols(); i++ )
{
Score+=pow(Vector(1,i),10);
} // for i
return Score;
}
int main()
{
//Mi_Est_Evo RA(#padres,#variables,1=m+l 0=m,l , # hijos, iteraciones, prefijo del archivo, salva variables cada N iteraciones);
Mi_Est_Evo EST(70,10,0,490,20,"ex4_tipo0_",10);
EST.evolve();
return 0;
}
```

Archivo: Makefile

```
CCFLAGS = -Wall -I/home/rodolfo/Matrix-C++/lib -I/home/rodolfo/EE-C++/ -L/home/rodolfo/Matrix-C++/lib
CCLIBS = -lmat -lm
all: ex1 ex2 ex3 ex4 ex5 ex6
clean:
rm ex1 ex2 ex3 ex4 ex5 ex6
```

```
ex1: ex1.cpp
g++ $(CCFLAGS) -o ex1 ex1.cpp $(CCLIBS)
ex2: ex2.cpp
g++ $(CCFLAGS) -o ex2 ex2.cpp $(CCLIBS)
ex3: ex3.cpp
g++ $(CCFLAGS) -o ex3 ex3.cpp $(CCLIBS)
ex4: ex4.cpp
g++ $(CCFLAGS) -o ex4 ex4.cpp $(CCLIBS)
ex5: ex5.cpp
g++ $(CCFLAGS) -o ex5 ex5.cpp $(CCLIBS)
ex6: ex6.cpp
g++ $(CCFLAGS) -o ex6 ex6.cpp $(CCLIBS)
```

Bibliografía

[1] I. Rechenberg, "Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologishen Evolution" Stuttgart: Frommann Holzboog, 1973.

[2] Schwefel H.-P. (1992) Imitating evolution: Collective, two-level learning processes. In Witt U. (ed) Explaning Process an Change - Approaches to Evolutionary Economics, pages 49-63. The University of Michigan Press, Ann Arbor, MI.

[3] Fernández-Anaya, G., Muñoz-Gutiérrez, S., Sánchez-Guzmán, R.A., and Mayol-Cuevas, W.W, Simultaneous stabilization using evolutionary strategies, International Journal of Control, 68 (6), 1997, pp. 1417-1435.

[4] Muñoz, S., Fernández, G., Sánchez R.A. and Mayol, W.W, 1997. Evolutionary Algorithms for Plants Simultaneous Stabilization. IASTED Int. Conf. Cancún, México. pp. 151-158.

[5] I. Rechenberg, Evolution and Artificial Intelligence, Machine Learning Principle and Tecnics, (Forsyth, R.: Chapman and Hall Computing), 1989.

[6] Schwefel, and Rudolph, Contemporary Evolution Strategies. In Advances in Artificial Life. Berlin Springer, 686-695, 1995.

[7] I. Rechenberg, "The evolution Strategy: A mathematical model of Darwinian evolution" Springer Series in Synergetics, 22, E. Frehland, Ed. Berlin: Springer, 1984, 122-132.

[8] I. Rechenberg, "Evolutionsstrategie 94" Stuttgart: Frommann Holzboog, 1994.

[9] Hans-Paul Schwefel, Evolution and Optimum Seeking, Wiley Interscience, 1995.

[10] Santamaria Romero, E.; Romero Bastida, M. & Figueroa Nazuno, J. Una evolución del algoritmo genético: algoritmo autoadaptativo. Mexicon'89, México D.F; 18-23 de septiembre de 1989.

[11] Vazquez Nava, A. & Figueroa Nazuno, J. Algoritmo genético: un método eficiente para problemas de optimización. XXXIV Congreso Nacional de Física. México D.F; 21-25 de octubre de 1991.

[12] Zozoaga Velazquez, G.; Tapia Hernández, J.G.; Vargas Medina E. & Figueroa Nazuno, J. Reconocedor de patrones guestaltico y su relación con la primera capa de una red neuronal. IV Congreso Nacional sobre Informática y Computación. México Aguascalientes, Ags; 23-25 de octubre de 1991.

[13] Figueroa Nazuno, J; Vargas Medina, E. & Vázquez Nava A. Theoretical and experimental analysis of the first layer in a neural network. Proceedings of the SPIE Conference on Science of Artificial Neural Networks, Orlando, FL, U.S.A, 1992.

[14] Sandoval Rios, M.; Figueroa Nazuno, J. Algoritmos genéticos de especies y objetivos multiples. Simposium Internacional de Computación, CNC-IPN, México D.F., México, 1993.

[15] Figueroa Nazuno, J. Investigación en Memoria Icónica. IV International Conference of Thinking. San Juan, Puerto Rico, 17-20 August, 1989.

[16] Sánchez Guzmán R.A.; Mayol Cuevas, W.W.; Figueroa Nazuno, J. Reducción del espacio de muestreo para redes neuronales utilizando optimización por algoritmo genético. Simposium Internacional de Computación, CNC-IPN, México D.F., México, 1993.

[17] Back, T., Evolutionary Algorithms in Theory and Practice, New York Oxford, Oxford University Press, 1996.

[18] Goldberg, D.E., Genetic Algorithms in Search, Optimization and Machine Learning, (Adison Wesley, Reading, MA), 1989.

[19] Hoffmeister, F., Back, T.,Genetic Algorithms and Evolution Strategies: Similarities and Differences, Parallel Problem Solving from Nature, Proc. 1st Workshop PPSN, pp. 447-461 Springer, Berlin, Lecture Notes in Computer Science, Vol. 496, 1991.

[20] Holland, J.P., Adaptation in Natural and Artificial Systems: an Introduction Analysis with Applications to Biology, Control and Artificial Intelligence, Cambridge, Massachusetts, U.S.A., MIT Press, 1992.

[21] Koza J.R., Genetic Programming, Cambridge and London: The MIT Press. 1992.

[22] Koza J.R., Genetic Programming II, Automatic Discovery of Reusable Programs. Cambridge and London The MIT Press. 1994.

[23] Srinivas, M., and Patnaik, L.M., Genetic Algorithms a survey. IEEE Computer, 27, pp. 17-26, 1994.

[24] Michalewicz, Z., Genetic Algorithms, Numerical Optimization and Constraints, Proceedings of the 6th International Conference on Genetic Algorithms, University Pittsburgh, july 15-19, 1995. Morgan Kauffmann, San Francisco, 1995, pp. 151-158.

[25] Chipperfield, A.J., Fonseca, C.M, and Fleming, P.J., Development of genetic algorithm optimisation tools for multi-objetive optimisation problems in CACSD. IEEE Colloquium on Genetic Algorithms for Control Systems Engineering Digest, London WC2R OBL, Vol. 106, pp. 3/1-3/6, 1992.

[26] Forrest, S., Genetic Algorithms: Principles of Natural Selection Applied to Computation. Science, Vol. 261, pp. 872-878, 1993.

[27] Axelrod, R., The evolution of cooperation, Basic Books, New York, 1984.

[28] Back, T., Optimal mutation rates in genetic search, in: Forrest, pp. 2-9, 1993.

[29] Berlin, V.G. Parallel randomized search strategies, ARC 33, pp. 398-403, 1972.

[30] Box, G.E.P., N.R. Draper, Empirical model-building and response surfaces, Wiley, New York, 1987.

[31] De Jong, K., Evolutionary Computation (journal), MIT Press, Cambridge MA., 1993.

[32] Rechenberg, I., Cybernetic solution path of an experimental problem, Royal Aircraft Establishment Transl., No. 1122, B.F. Toms, Transl. (Ministry of Aviation, Royal Aircraft Establishment, Farnborough, Hants., United Kingdom, 1965.

[33] Laboratorio de INvestigación para el Desarrollo Académico, División de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad Nacional Autónoma de México. http://www.grupolinda.org

[34] G. Fernández, S. Muñoz, R. A. Sánchez and W.W. Mayol, Simultaneous and robust observability using evolutionary strategies and substitutions, IEEE Conference on Decision and Control, CDC99, Phoenix, Arizona, USA, del 7 al 10 de December de 1999.

[35] J. Juárez-Guerrero, S. Muñoz-Gutiérrez and W.W. Mayol-Cuevas. Design of a Walking Machine Structure using Evolutionary Strategies. IEEE Int. Conf on Systems Man and Cybernetics 98. San Diego Calif. pp. 1427-1432. OMNIPRESS. USA. October 11-14. 1998.

[36] S. Muñoz, G. Fernández, R. A. Sánchez and W. W. Mayol, Strong simultaneous stabilization using evolutionary strategies, Memorias de la Quinta Conferencia de Ingeniería Eléctrica CIE99, CINVESTAV, México, D. F., 18-24, del 8 al 10 de Septiembre de 1999.