

Tema 4

Aplicaciones de Física y Matemática Usando Python

Índice

1. Introducción	3
1.1. Objetivo del Material	3
1.2. Enfoque matemático de los modelos físicos	3
1.3. Fórmulas a programas funcionales	3
1.4. Variables y operaciones en Python	4
1.4.1. Fundamento Teórico	4
1.4.2. Ejemplo Práctico	4
1.4.3. Análisis del Programa	5
1.5. Operaciones matemáticas y librerías	6
1.5.1. Fundamento Teórico	6
1.5.2. Ejemplo Práctico	7
1.5.3. Análisis del Programa	7
1.6. Funciones: traducir fórmulas a código	9
1.6.1. Fundamento Teórico	9
1.6.2. Ejemplo Práctico	9
1.6.3. Análisis del Programa	9
2. Programas de Física	11
2.1. Equilibrio de Fuerzas	11
2.1.1. Descripción del Programa	11
2.1.2. Fundamento Teórico	11
2.1.3. Ejemplo Practico	12
2.1.4. Análisis del Problema	12
2.2. Ciclos: simulaciones paso a paso	16
2.2.1. Fundamento Teórico	17
2.2.2. Ejemplo Práctico	17
2.2.3. Análisis del Programa	17
2.2.4. Uso de los Ciclos	19
2.3. Cinemática: Movimiento Rectilíneo Uniforme (MRU)	19
2.3.1. Descripción del Programa	19
2.3.2. Fundamento Teórico	20
2.3.3. Ejemplo Practico	20

2.3.4.	Análisis del Problema	20
2.4.	Cinemática: Movimiento Rectilíneo Uniformemente Acelerado (MRUA)	24
2.4.1.	Descripción del Programa	24
2.4.2.	Fundamento Teórico	25
2.4.3.	Ejemplo Practico	25
2.4.4.	Análisis del Problema	25
2.5.	Densidad y diccionarios	28
2.5.1.	Descripción del Programa	29
2.5.2.	Fundamento Teórico	29
2.5.3.	Ejemplo Practico	29
2.5.4.	Análisis del Problema	29
2.6.	Termodinámica: Calor específico	33
2.6.1.	Descripción del Programa	34
2.6.2.	Fundamento Teórico	34
2.6.3.	Ejemplo Practico	34
2.6.4.	Análisis del Problema	34
2.7.	Termodinámica: Equilibrio térmico	37
2.7.1.	Descripción del Programa	37
2.7.2.	Fundamento Teórico	37
2.7.3.	Ejemplo Practico	38
2.7.4.	Análisis del Problema	38
3.	Programas de Física	42
3.1.	Secuencias Matemáticas: Fibonacci	42
3.1.1.	Descripción del Programa	42
3.1.2.	Fundamento Teórico	42
3.1.3.	Ejemplo Practico	43
3.1.4.	Análisis del Problema	43
3.2.	Razón áurea	47
3.2.1.	Fundamento Teórico	47
3.2.2.	Ejemplo Práctico	47
3.2.3.	Análisis del Programa	47
4.	Proceso general para resolver problemas con Python	49
5.	Referencias	50

1. Introducción

Python es un lenguaje de programación versátil y ampliamente utilizado en el análisis, modelado y simulación de fenómenos físicos y matemáticos. Su sintaxis sencilla y la gran cantidad de bibliotecas disponibles permiten resolver problemas de dinámica, termodinámica, álgebra, análisis numérico, entre otros.

1.1. Objetivo del Material

El objetivo de este material es que el alumno desarrolle programas que representen y calculen situaciones típicas de la física y las matemáticas, aplicando las ecuaciones teóricas correspondientes y utilizando Python como herramienta principal.

1.2. Enfoque matemático de los modelos físicos

Los fenómenos físicos descritos en este material se representan mediante modelos matemáticos que establecen relaciones entre variables como posición, velocidad, aceleración, fuerza, masa y temperatura. De acuerdo con textos clásicos de física como Serway y Jewett (2014) y Giancoli (2008), el análisis físico se basa en la correcta identificación de variables y en la aplicación rigurosa de ecuaciones fundamentales.

Python permite traducir estos modelos matemáticos directamente a expresiones computacionales, conservando la estructura algebraica original y facilitando la verificación de resultados mediante cálculos repetitivos y simulaciones.

1.3. Fórmulas a programas funcionales

Python es un lenguaje de programación de alto nivel caracterizado por su sintaxis sencilla y legible, lo que permite traducir fácilmente fórmulas matemáticas y físicas a programas funcionales. Esta característica lo convierte en una herramienta ideal para estudiantes de Física y Matemáticas, ya que facilita la comprensión de los fenómenos sin complicaciones innecesarias del lenguaje.

Una de las principales ventajas de Python es que permite pasar directamente de las expresiones matemáticas a código ejecutable, facilitando la automatización de cálculos, la simulación de sistemas físicos y el análisis de resultados.

De esta manera, Python no sustituye a la Física ni a las Matemáticas, sino que las complementa como una herramienta de apoyo para la resolución de problemas.

1.4. Variables y operaciones en Python

Las variables son uno de los elementos fundamentales en la programación, ya que permiten almacenar y manipular información dentro de un programa. En Física y Matemáticas, las variables representan magnitudes físicas o valores matemáticos que intervienen en la resolución de un problema, como la masa, la velocidad, el tiempo o la temperatura.

El uso de variables en Python facilita la traducción directa de fórmulas físicas y matemáticas a código ejecutable, permitiendo realizar cálculos de manera clara, ordenada y eficiente.

1.4.1. Fundamento Teórico

Una variable es un espacio en memoria donde se almacena un dato que puede ser utilizado y modificado durante la ejecución de un programa. En el contexto de la Física y las Matemáticas, las variables suelen representar magnitudes asociadas a unidades de medida específicas.

Algunos ejemplos de variables comúnmente utilizadas son:

- Masa
- Velocidad
- Tiempo
- Temperatura

Las operaciones matemáticas entre variables se realizan de forma muy similar a las expresiones algebraicas tradicionales, lo que permite que las fórmulas físicas se escriban casi de la misma manera en Python.

1.4.2. Ejemplo Práctico

Se considera un movimiento rectilíneo uniforme, donde la posición final de un objeto se calcula a partir de la velocidad y el tiempo mediante la expresión:

$$x = v \cdot t$$

Esta ecuación puede implementarse directamente en Python utilizando variables.

1.4.3. Análisis del Programa

En el siguiente programa se definen variables que representan magnitudes físicas, se realiza una operación matemática entre ellas y se muestra el resultado en pantalla. El objetivo es demostrar cómo una fórmula física puede transformarse fácilmente en código.

■ *Pseudocódigo*

```
Inicio
    Definir masa = 2.0          // kg
    Definir velocidad = 20.0    // m/s
    Definir tiempo = 10.0       // s

    posicion <- velocidad * tiempo

    Mostrar "Posicion final:", posicion, "m"
Fin
```

Ejemplo 1: Pseudocódigo: Definición de variables y cálculo de posición

■ *Definición de variables físicas*

```
masa = 2.0          # kg
velocidad = 20.0    # m/s
tiempo = 10.0       # s
```

Ejemplo 2: Definición de variables que representan magnitudes físicas con sus respectivas unidades.

■ *Operación matemática*

```
posicion = velocidad * tiempo
```

Ejemplo 3: Cálculo de la posición final utilizando la fórmula del movimiento rectilíneo uniforme.

■ *Impresión del resultado*

```
print("Posicion final:", posicion, "m")
```

Ejemplo 4: Impresión del resultado obtenido en pantalla.

El programa calcula correctamente la posición final del objeto a partir de la velocidad y el tiempo proporcionados, demostrando que las fórmulas físicas pueden implementarse en Python de manera directa y comprensible.

■ *Código completo*

```
masa = 2.0          # kg
velocidad = 20.0    # m/s
tiempo = 10.0       # s
posicion = velocidad * tiempo
print("Posicion final:", posicion, "m")
```

Ejemplo 5: Variables y operaciones en Python

■ *Resultado*

```
Posición final: 200.0 m
```

1.5. Operaciones matemáticas y librerías

Python soporta una amplia variedad de operaciones matemáticas y trigonométricas, lo que lo convierte en una herramienta muy útil para la resolución de problemas en Física y Matemáticas. Estas operaciones permiten modelar fenómenos físicos de manera precisa y eficiente mediante el uso de expresiones matemáticas similares a las utilizadas en la teoría.

Para realizar cálculos matemáticos avanzados, como funciones trigonométricas, raíces cuadradas y conversiones angulares, Python utiliza librerías especializadas. En el caso de la Física, una de las más importantes es la librería `math`.

1.5.1. Fundamento Teórico

Las funciones trigonométricas son ampliamente utilizadas en Física para el análisis de fuerzas, movimientos y sistemas vectoriales. Sin embargo, en Python estas funciones trabajan con ángulos expresados en radianes, por lo que es necesario convertir los ángulos dados en grados a radianes antes de realizar los cálculos.

La conversión de grados a radianes se realiza mediante la siguiente relación:

$$\theta_{rad} = \theta_{deg} \cdot \frac{\pi}{180}$$

1.5.2. Ejemplo Práctico

Se analiza un vector de magnitud unitaria aplicado con un ángulo de 60° . Para obtener sus componentes rectangulares, se emplean las funciones trigonométricas seno y coseno.

1.5.3. Análisis del Programa

En el siguiente programa se utiliza la librería `math` para convertir un ángulo de grados a radianes y calcular las componentes horizontal y vertical de un vector. El objetivo es demostrar el uso de librerías matemáticas en Python aplicadas a problemas físicos.

■ *Pseudocódigo*

```
Inicio
  Definir angulo_grados = 60
  Definir PI = 3.1416

  angulo_rad <- angulo_grados * (PI / 180)

  componente_x <- cos(angulo_rad)
  componente_y <- sin(angulo_rad)

  Mostrar "Componente X:", componente_x
  Mostrar "Componente Y:", componente_y
Fin
```

Ejemplo 6: Pseudocódigo: Operaciones matemáticas y librerías

■ *Importación de la librería matemática*

```
import math
```

Ejemplo 7: Importación de la librería `math` para realizar operaciones matemáticas y trigonométricas.

■ *Definición del ángulo*

```
angulo = 60
```

Ejemplo 8: Definición del ángulo en grados.

■ *Conversión a radianes*

```
angulo_rad = math.radians(angulo)
```

Ejemplo 9: Conversión del ángulo de grados a radianes utilizando la función `radians`.

■ *Cálculo de componentes trigonométricas*

```
fx = math.cos(angulo_rad)
fy = math.sin(angulo_rad)
```

Ejemplo 10: Cálculo de las componentes horizontal y vertical del vector mediante funciones trigonométricas.

■ *Impresión de resultados*

```
print("Fx:", fx)
print("Fy:", fy)
```

Ejemplo 11: Impresión de los valores calculados para las componentes del vector.

El programa calcula correctamente las componentes horizontal y vertical del vector utilizando funciones trigonométricas, demostrando la utilidad de la librería `math` para la resolución de problemas físicos en Python.

■ *Código completo*

```
import math
angulo = 60
angulo_rad = math.radians(angulo)
fx = math.cos(angulo_rad)
fy = math.sin(angulo_rad)
print("Fx:", fx)
print("Fy:", fy)
```

Ejemplo 12: Operaciones matemáticas y librerías

■ *Resultado*

```
Fx: 0.5000000000000001
Fy: 0.8660254037844386
```

1.6. Funciones: traducir fórmulas a código

Las funciones permiten organizar el código en bloques reutilizables que representan procesos específicos. En el contexto de la Física y las Matemáticas, una función puede interpretarse como una “receta” que recibe ciertos datos de entrada, aplica una fórmula y devuelve un resultado.

El uso de funciones evita la repetición de código, mejora la claridad de los programas y facilita la traducción directa de expresiones matemáticas a instrucciones computacionales.

1.6.1. Fundamento Teórico

Una función es un conjunto de instrucciones que se ejecutan cuando es llamada. Recibe valores de entrada llamados parámetros, realiza operaciones sobre ellos y devuelve un resultado. En Física, las funciones suelen representar relaciones matemáticas entre magnitudes físicas.

Por ejemplo, la densidad de un material se define como la relación entre su masa y su volumen:

$$\rho = \frac{m}{V}$$

Esta expresión puede implementarse fácilmente en Python mediante una función.

1.6.2. Ejemplo Práctico

Se implementa una función que calcula la densidad de un material a partir de su masa y su volumen, permitiendo reutilizar la fórmula para distintos valores.

1.6.3. Análisis del Programa

El siguiente programa define una función en Python que recibe la masa y el volumen como parámetros, calcula la densidad aplicando la fórmula correspondiente y devuelve el resultado. Posteriormente, la función es utilizada para obtener la densidad de un material específico.

■ *Pseudocódigo*

```
Funcion calcular_densidad(masa, volumen)
    Retornar masa / volumen
FinFuncion

Inicio
```

```

    rho <- calcular_densidad(2.7, 0.001)
    Mostrar "La densidad es:", rho
Fin

```

Ejemplo 13: Pseudocódigo: Cálculo de densidad

■ *Definición de la función*

```

def densidad(masa, volumen):
    return masa / volumen

```

Ejemplo 14: Definición de la función que calcula la densidad de un material.

■ *Llamada a la función*

```

rho = densidad(2.7, 0.001)

```

Ejemplo 15: Uso de la función para calcular la densidad a partir de valores dados.

■ *Impresión del resultado*

```

print("Densidad:", rho, "kg/m^3")

```

Ejemplo 16: Impresión del valor de la densidad calculada.

El programa calcula correctamente la densidad del material a partir de los valores proporcionados, demostrando cómo una fórmula física puede representarse mediante una función en Python de forma clara y reutilizable.

■ *Código completo*

```

# Programa para calcular la densidad de un material
def densidad(masa, volumen):
    return masa / volumen

# Llamada a la funcion
rho = densidad(2.7, 0.001)

# Impresion del resultado
print("Densidad:", rho, "kg/m^3")

```

Ejemplo 17: Operaciones de Cálculo de densidad

■ *Resultado*

```

Densidad: 2700.0 kg/m^3

```

2. Programas de Física

En esta sección se presentan diversos programas desarrollados en Python que permiten modelar y analizar fenómenos fundamentales de la física. Cada programa se basa en ecuaciones físicas bien establecidas y tiene como objetivo facilitar la comprensión de conceptos mediante simulaciones y cálculos computacionales.

2.1. Equilibrio de Fuerzas

El estudio del equilibrio de fuerzas es fundamental en la mecánica clásica, ya que permite analizar sistemas en los que un cuerpo permanece en reposo o se mueve con velocidad constante. Mediante el uso de Python, es posible automatizar los cálculos vectoriales y visualizar si un sistema cumple o no con las condiciones de equilibrio estático.

2.1.1. Descripción del Programa

Este programa analiza un sistema de fuerzas que actúan sobre un punto material, calcula la fuerza resultante mediante descomposición vectorial y determina si el sistema está en equilibrio.

2.1.2. Fundamento Teórico

El equilibrio estático se presenta cuando la resultante de todas las fuerzas que actúan sobre un cuerpo es nula. Para analizar este tipo de sistemas, es necesario descomponer cada fuerza en sus componentes rectangulares y realizar la suma vectorial correspondiente.

Para que un cuerpo esté en equilibrio estático, la suma vectorial de todas las fuerzas debe ser cero:

$$\sum F_x = 0, \quad \sum F_y = 0$$

Las componentes rectangulares de una fuerza se obtienen mediante:

$$F_x = F \cos(\theta), \quad F_y = F \sin(\theta)$$

Magnitud de la fuerza resultante::

$$F_R = \sqrt{\left(\sum F_x\right)^2 + \left(\sum F_y\right)^2}$$

2.1.3. Ejemplo Practico

Se analizan tres fuerzas de 10 N separadas en 120° (0° , 120° y 240°). El programa calcula que la fuerza resultante es cero, confirmando un equilibrio perfecto.

2.1.4. Análisis del Problema

En esta sección se explica paso a paso la lógica utilizada en el programa, desde la importación de librerías hasta la verificación del equilibrio. El objetivo es mostrar cómo se traduce el modelo físico en instrucciones computacionales claras y ordenadas.

■ *Pseudocódigo*

```
Inicio
  Recibir lista_de_fuerzas (magnitud, angulo)

  suma_fx <- 0
  suma_fy <- 0

  Para cada fuerza en lista_de_fuerzas
    angulo_rad <- angulo * (PI / 180)
    suma_fx <- suma_fx + (magnitud * cos(angulo_rad))
    suma_fy <- suma_fy + (magnitud * sin(angulo_rad))
  FinPara

  resultante <- raiz_cuadrada(suma_fx^2 + suma_fy^2)

  Si resultante < 0.01 Entonces
    Mostrar "Sistema en equilibrio"
  Sino
    Mostrar "Sistema no esta en equilibrio"
  FinSi
Fin
```

Ejemplo 18: Pseudocódigo: Equilibrio de fuerzas

■ *Importar la librería matemática*

```
# Programa para calcular el equilibrio de fuerzas
import math
```

Ejemplo 19: Importación de la librería `math`. Se importa la librería `math` para usar funciones matemáticas como seno, coseno, raíz cuadrada, etc.

■ *Definición de la función principal*

```
def equilibrio_fuerzas(fuerzas):  
    # Parametros: lista de tuplas (magnitud, angulo_en_grados)  
    )  
    # Retorna: fuerza_resultante, angulo, en_equilibrio
```

Ejemplo 20: Definición de la función principal. Esta función recibe una lista de fuerzas y se encarga de calcular la fuerza resultante del sistema para determinar si existe equilibrio.

La función recibe una lista de fuerzas, donde cada fuerza es una tupla:

```
(magnitud, angulo_en_grados)
```

Ejemplo 21: Estructura de una fuerza. Cada fuerza se representa mediante una tupla que contiene su magnitud y el ángulo de aplicación medido en grados.

■ *Inicializar sumas de componentes*

```
suma_fx = 0 # Suma de componentes horizontales  
suma_fy = 0 # Suma de componentes verticales
```

Ejemplo 22: Inicialización de componentes. Estas variables almacenan la suma de las componentes horizontales y verticales de todas las fuerzas que actúan sobre el sistema.

■ *Descomponer cada fuerza en componentes*

```
# Descomponer cada fuerza en componentes  
for magnitud, angulo_grados in fuerzas:  
    angulo_rad = math.radians(angulo_grados) # Convertir a  
    radianes  
    fx = magnitud * math.cos(angulo_rad) # Componente  
    horizontal  
    fy = magnitud * math.sin(angulo_rad) # Componente  
    vertical
```

Ejemplo 23: Cada fuerza se convierte a componentes usando trigonometría.

Luego se van acumulando:

```
suma_fx += fx # Acumular componente x  
suma_fy += fy # Acumular componente y
```

■ *Magnitud de la fuerza resultante*

```
# Calcular magnitud de la fuerza resultante
f_resultante = math.sqrt(suma_fx**2 + suma_fy**2)
```

Ejemplo 24: Cálculo de la fuerza resultante $F_R = \sqrt{(\sum F_x)^2 + (\sum F_y)^2}$

■ *Dirección de la fuerza resultante*

```
# Calcular direccion de la fuerza resultante
if fuerza_resultante > 0.01: # Si hay fuerza resultante
    angulo_resultante = math.degrees(math.atan2(suma_fy,
    suma_fx))
else:
    angulo_resultante = 0
```

Ejemplo 25: *atan2* calcula correctamente el ángulo tomando en cuenta el signo de *fx* y *fy*.

Convertimos el ángulo a grados.

Si la fuerza es casi cero (<0.01 N), se considera que no hay dirección (equilibrio).

■ *Verificar si hay equilibrio*

```
# Verificar equilibrio (tolerancia de 0.01 N)
en_equilibrio = fuerza_resultante < 0.01
```

Ejemplo 26: Si la fuerza neta es muy pequeña (tolerancia de 0.01 N), el sistema está en equilibrio.

■ *Retorno de valores*

```
return fuerza_resultante, angulo_resultante, en_equilibrio
```

Ejemplo 27: Verificación de equilibrio del sistema

La función devuelve los siguientes valores:

1. Magnitud de la fuerza resultante.
2. Ángulo de la fuerza resultante.

3. Valor lógico (True o False) que indica si el sistema se encuentra en equilibrio.

■ *Ejemplo de uso*

```
# Ejemplo: tres fuerzas de 10 N separadas 120 grados
fuerzas = [
    (10, 0),
    (10, 120),
    (10, 240)
]
```

Ejemplo 28: Tres fuerzas iguales pero separadas 120°, que deben dar equilibrio perfecto.

■ *Impresión de resultados*

```
Fr, angulo, equilibrio = equilibrio_fuerzas(fuerzas)
print(f"Fuerza resultante:{Fr.2f} N")
print(f"Angulo: {angulo:.2f} grados")
print(f"Sistema en equilibrio: {equilibrio}")
```

Ejemplo 29: Formato con dos decimales.

■ *Código completo*

```
# Programa para calcular el equilibrio de fuerzas
import math

def equilibrio_fuerzas(fuerzas):
    # Parametros: lista de tuplas (magnitud, angulo_en_grados)
    # Retorna: fuerza_resultante, angulo, en_equilibrio

    suma_fx = 0 # Suma de componentes horizontales
    suma_fy = 0 # Suma de componentes verticales

    # Descomponer cada fuerza en componentes
    for magnitud, angulo_grados in fuerzas:
        angulo_rad = math.radians(angulo_grados) # Convertir a radianes
        fx = magnitud * math.cos(angulo_rad) # Componente horizontal
        fy = magnitud * math.sin(angulo_rad) # Componente vertical
```

```

        suma_fx += fx # Acumular componente x
        suma_fy += fy # Acumular componente y

# Calcular magnitud de la fuerza resultante
fuerza_resultante = math.sqrt(suma_fx**2 + suma_fy**2)

# Calcular direccion de la fuerza resultante
if fuerza_resultante > 0.01: # Si hay fuerza resultante
    angulo_resultante = math.degrees(math.atan2(suma_fy,
suma_fx))
else:
    angulo_resultante = 0

# Verificar equilibrio (tolerancia de 0.01 N)
en_equilibrio = fuerza_resultante < 0.01

return fuerza_resultante, angulo_resultante,
en_equilibrio

# Ejemplo: tres fuerzas de 10 N separadas 120 grados
fuerzas = [
    (10, 0),      # 10 N hacia la derecha
    (10, 120),    # 10 N a 120 grados
    (10, 240)     # 10 N a 240 grados
]

Fr, angulo, equilibrio = equilibrio_fuerzas(fuerzas)

print(f"Fuerza resultante: {Fr:.2f} N")
print(f"Angulo: {angulo:.2f} grados")
print(f"Sistema en equilibrio: {equilibrio}")

```

Ejemplo 30: Equilibrio de fuerzas en el plano

■ *Resultado*

```

Fuerza resultante: 0.00 N
Angulo: 0.00 grados
Sistema en equilibrio: True

```

2.2. Ciclos: simulaciones paso a paso

Los ciclos permiten repetir un conjunto de instrucciones varias veces de manera controlada. En Física y Matemáticas, los ciclos son una herramienta

fundamental para realizar simulaciones paso a paso, ya que permiten calcular la evolución de un sistema en distintos instantes de tiempo o bajo diferentes condiciones iniciales.

Gracias a los ciclos, es posible modelar fenómenos físicos de forma iterativa, analizar tendencias y observar cómo cambian las magnitudes físicas a lo largo del tiempo.

2.2.1. Fundamento Teórico

Un ciclo es una estructura de control que permite ejecutar repetidamente un bloque de código mientras se cumpla una condición o durante un número determinado de iteraciones. En simulaciones físicas, los ciclos se utilizan para actualizar variables como la posición, la velocidad o la energía de un sistema en cada paso del proceso.

Existen principalmente dos tipos de ciclos en Python:

- **Ciclo for:** se utiliza cuando se conoce de antemano el número de iteraciones.
- **Ciclo while:** se utiliza cuando la repetición depende de una condición lógica.

2.2.2. Ejemplo Práctico

Se presenta una simulación sencilla de movimiento rectilíneo uniforme, donde se calcula la posición de un objeto en distintos instantes de tiempo. Para ello, se emplean ciclos `for` y `while`, los cuales permiten repetir los cálculos paso a paso.

2.2.3. Análisis del Programa

El programa simula el movimiento de un objeto que se desplaza con velocidad constante. En cada iteración, el tiempo avanza un segundo y se calcula la posición correspondiente utilizando la expresión:

$$x = v \cdot t$$

El objetivo es mostrar cómo los ciclos permiten automatizar cálculos repetitivos y representar la evolución de un sistema físico en el tiempo.

- ***Definición de variables iniciales***

Se definen la velocidad del objeto, el tiempo total de simulación y la posición inicial.

- *Uso del ciclo for*

El ciclo `for` se utiliza cuando se conoce previamente el número de iteraciones, representando cada segundo de la simulación.

- *Uso del ciclo while*

El ciclo `while` se emplea cuando la repetición depende de una condición lógica, en este caso, mientras el tiempo sea menor o igual al tiempo total.

- *Pseudocódigo*

```
Inicio
  Definir velocidad <- 2.0
  Definir tiempo_total <- 10

  Para t <- 1 Hasta tiempo_total
    posicion <- velocidad * t
    Mostrar t, posicion
  FinPara

  t <- 1
  Mientras t <= tiempo_total Hacer
    posicion <- velocidad * t
    Mostrar t, posicion
    t <- t + 1
  FinMientras
Fin
```

Ejemplo 31: Pseudocódigo: Movimiento con estructuras FOR y WHILE

- *Código completo*

```
# Simulacion simple de movimiento rectilineo uniforme

velocidad = 2.0      # m/s
tiempo_total = 10    # segundos
posicion = 0.0       # metros

print("Simulacion usando ciclo for")

# Ciclo for: numero fijo de pasos
for t in range(1, tiempo_total + 1):
    posicion = velocidad * t
    print(f"Tiempo: {t} s | Posicion: {posicion} m")

print("\nSimulacion usando ciclo while")
```

```
# Reiniciar variables
t = 1
posicion = 0.0

# Ciclo while: depende de una condicion
while t <= tiempo_total:
    posicion = velocidad * t
    print(f"Tiempo: {t} s | Posición: {posicion} m")
    t += 1
```

Ejemplo 32: Simulación paso a paso de movimiento rectilíneo uniforme usando ciclos `for` y `while`.

■ *Resultado*

```
Tiempo: 8 s | Posición: 16.0 m
Tiempo: 9 s | Posición: 18.0 m
Tiempo: 10 s | Posición: 20.0 m
```

2.2.4. Uso de los Ciclos

El uso de ciclos permite automatizar cálculos repetitivos que serían tediosos de realizar manualmente. En Física, esto resulta especialmente útil para simular movimientos, procesos dinámicos y sistemas que evolucionan con el tiempo.

El ciclo `for` se emplea comúnmente en simulaciones con un número fijo de pasos, mientras que el ciclo `while` es adecuado cuando la simulación continúa hasta que se cumple una condición específica, como alcanzar un valor límite o un estado de equilibrio.

2.3. Cinemática: Movimiento Rectilíneo Uniforme (MRU)

El Movimiento Rectilíneo Uniforme es uno de los modelos más simples de la cinemática, ya que describe un desplazamiento en línea recta con velocidad constante. Su simplicidad lo convierte en un excelente punto de partida para la simulación de movimientos usando Python.

2.3.1. Descripción del Programa

Simula el movimiento de un objeto con velocidad constante y genera una tabla posición–tiempo.

2.3.2. Fundamento Teórico

El Movimiento Rectilíneo Uniforme (MRU) se caracteriza por una velocidad constante. La ecuación fundamental que relaciona posición, velocidad y tiempo es:

$$x = x_0 + vt$$

Donde:

- x = posición final (m)
- x_0 = posición inicial (m)
- v = velocidad constante (m/s)
- t = tiempo transcurrido (s)

La velocidad se define como:

$$v = \frac{\Delta x}{\Delta t} = \text{constante}$$

2.3.3. Ejemplo Practico

Un automóvil viaja a 20 m/s durante 10 s. Se calcula y muestra la posición cada 0.5 s.

2.3.4. Análisis del Problema

- *Pseudocódigo*

```
Funcion calcular_mrue(v0, a, t)
    posicion <- (v0 * t) + (0.5 * a * t^2)
    velocidad_final <- v0 + (a * t)
    Retornar posicion, velocidad_final
FinFuncion

Inicio
    Para tiempo <- 0 Hasta 10 Paso 2
        (pos, vel) <- calcular_mrue(v0, a, tiempo)
        Mostrar tiempo, pos, vel
    FinPara
Fin
```

Ejemplo 33: Pseudocódigo: Movimiento rectilíneo uniformemente acelerado (MRUA)

- *Función: Movimiento Rectilíneo Uniforme*

```
# Programa para calcular MRU
def movimiento_rectilineo_uniforme(x0, v, t):
    # Parametros: x0=posicion inicial, v=velocidad, t=tiempo
    # Retorna: posicion final
```

Ejemplo 34: Define la ecuación del MRU (Movimiento Rectilíneo Uniforme).

```
# Ecuacion del MRU:  $x = x_0 + v \cdot t$ 
    posicion = x0 + v * t

    return posicion
```

Ejemplo 35: La ecuación del MRU. Devuelve la posición final tras un tiempo t .

■ *Función: Simular MRU*

```
def simular_mru(x0, v, tiempo_total, dt=0.5):
    # Simula el MRU durante un intervalo de tiempo
    # Parametros: x0=posicion inicial, v=velocidad,
    # tiempo_total=tiempo de simulacion, dt=paso de tiempo
```

Ejemplo 36: Sirve para simular paso por paso el movimiento, generando una tabla (tiempo, posición).

Parámetros:

1. $x_0 \rightarrow$ posición inicial
2. $v \rightarrow$ velocidad
3. tiempo total $\rightarrow$ cuánto tiempo queremos simular
4. $dt \rightarrow$ paso de tiempo (cada cuánto calcular)

Inicialización

```
datos = [] # Lista para almacenar resultados
t = 0     # Tiempo inicial
```

1. datos: guardará pares (tiempo, posición)
2. t: es el tiempo inicial

Bucle de simulación

```
# Simular hasta el tiempo total
while t <= tiempo_total:
```

Ejemplo 37: Se simula desde $t = 0$ hasta $t = \text{tiempo total}$.

Dentro del bucle:

```
x = movimiento_rectilineo_uniforme(x0, v, t) # Calcular
      posicion
```

Ejemplo 38: Calcular posición en cada instante

```
datos.append((t, x)) # Guardar tiempo y posicion
```

Ejemplo 39: Guardar tiempo y posición

```
t += dt # Incrementar tiempo
return datos
```

Ejemplo 40: Incrementar tiempo

■ *Configuración del ejemplo*

```
# Ejemplo: Un auto que viaja a 20 m/s
x_inicial = 0 # posicion inicial en metros
velocidad = 20 # velocidad en m/s
tiempo = 10 # tiempo total en segundos
```

Ejemplo 41: Incrementar tiempo

1. *Empezamos en posición 0 m.*
2. *Velocidad constante 20 m/s.*
3. *Simulación de 10 s.*

■ *Llamar la simulación*

```
# Simular el movimiento
trayectoria = simular_mru(x_inicial, velocidad, tiempo)
```

Ejemplo 42: *Trayectoria* es una lista con valores

■ *Impresión de resultados*

```
print("Tiempo (s)\tPosicion (m)")
for t, x in trayectoria:
    print(f"{t:.1f}\t\t{x:.1f}")
```

Devuelve:

1. Formato: *.1f* → muestra 1 decimal
2. Imprime la tabla completa

■ *Código completo*

```
# Programa para calcular MRU
def movimiento_rectilineo_uniforme(x0, v, t):
    # Parametros: x0=posicion inicial, v=velocidad, t=tiempo
    # Retorna: posicion final

    # Ecuacion del MRU:  $x = x_0 + v \cdot t$ 
    posicion = x0 + v * t

    return posicion

def simular_mru(x0, v, tiempo_total, dt=0.5):
    # Simula el MRU durante un intervalo de tiempo
    # Parametros: x0=posicion inicial, v=velocidad,
    # tiempo_total=tiempo de simulacion, dt=paso de tiempo

    datos = [] # Lista para almacenar resultados
    t = 0      # Tiempo inicial

    # Simular hasta el tiempo total
    while t <= tiempo_total:
        x = movimiento_rectilineo_uniforme(x0, v, t) #
        # Calcular posicion
        datos.append((t, x)) # Guardar tiempo y posicion
        t += dt              # Incrementar tiempo

    return datos

# Ejemplo: Un auto que viaja a 20 m/s
x_inicial = 0 # posicion inicial en metros
velocidad = 20 # velocidad en m/s
tiempo = 10 # tiempo total en segundos

# Simular el movimiento
trayectoria = simular_mru(x_inicial, velocidad, tiempo)
```

```
print("Tiempo (s)\tPosicion (m)")
for t, x in trayectoria:
    print(f"{t:.1f}\t\t{x:.1f}")
```

Ejemplo 43: Movimiento Rectilíneo Uniforme

■ Resultado

Tiempo (s)	Posicion (m)
0.0	0.0
0.5	10.0
1.0	20.0
1.5	30.0
2.0	40.0
2.5	50.0
3.0	60.0
3.5	70.0
4.0	80.0
4.5	90.0
5.0	100.0
5.5	110.0
6.0	120.0
6.5	130.0
7.0	140.0
7.5	150.0
8.0	160.0
8.5	170.0
9.0	180.0
9.5	190.0
10.0	200.0

2.4. Cinemática: Movimiento Rectilíneo Uniformemente Acelerado (MRUA)

El MRUA describe el movimiento de un objeto cuya velocidad cambia de manera uniforme debido a una aceleración constante. Este tipo de movimiento es común en situaciones reales como vehículos en aceleración o caída libre, por lo que resulta ideal para ser modelado computacionalmente.

2.4.1. Descripción del Programa

Calcula posición y velocidad para un movimiento con aceleración constante.

2.4.2. Fundamento Teórico

El Movimiento Rectilíneo Uniformemente Acelerado (MRUA) se caracteriza por una aceleración constante. Las ecuaciones fundamentales son:

Ecuación de velocidad:

$$v = v_0 + at$$

Ecuación de posición:

$$x = x_0 + v_0t + \frac{1}{2}at^2$$

Ecuación independiente del tiempo:

$$v^2 = v_0^2 + 2a(x - x_0)$$

Donde:

- v = velocidad final (m/s)
- v_0 = velocidad inicial (m/s)
- a = aceleración constante (m/s²)
- t = tiempo (s)
- x = posición final (m)
- x_0 = posición inicial (m)

2.4.3. Ejemplo Practico

Un vehículo acelera con $a = 2 \text{ m/s}^2$ desde una velocidad inicial de $v_0 = 5 \text{ m/s}$. El programa calcula la posición y la velocidad cada 2 s durante 10 s.

2.4.4. Análisis del Problema

▪ *Pseudocódigo*

```
Inicio
  Definir v0 = 5, a = 2, t_total = 10
  Para t desde 0 hasta t_total con saltos de 2:
    Calcular posicion = v0 * t + 0.5 * a * t^2
    Calcular velocidad_f = v0 + a * t
    Mostrar t, posicion y velocidad_f
Fin
```

Ejemplo 44: Pseudocodigo para calculo de MRUA con aceleracion.

■ *Definición de la función MRUA*

```
# Programa para calcular MRUA
def mrua(v0, a, t):
    # Parametros: v0=velocidad inicial, a=aceleracion, t=
    tiempo
    # Retorna: posicion y velocidad finales
```

Ejemplo 45: Esta función calcula el Movimiento Rectilíneo Uniformemente Acelerado, también llamado MRUA.

¿Qué devuelve?

1. la posición final
2. la velocidad final

■ *Ecuación de la posición*

```
# Ecuacion de posicion
posicion = v0 * t + 0.5 * a * t**2
```

Ejemplo 46: Ecuación del MRUA $x(t) = v_0t + \frac{1}{2}at^2$

Donde:

1. v_0 es la velocidad inicial
2. a es la aceleración
3. t es el tiempo transcurrido
4. $x(t)$ es la posición final

■ *Ecuación de la velocidad*

```
# Ecuacion de velocidad: v = v0 + a*t
velocidad = v0 + a * t
```

Ejemplo 47: Ecuación básica del MRUA: $v(t) = v_0 + at$. La velocidad aumenta o disminuye uniformemente dependiendo del signo de la aceleración.

■ *Retornar los resultados*

```
return posicion, velocidad
```

Ejemplo 48: La función regresa dos valores: posición y velocidad.

▪ *Definición del ejemplo*

```
# Ejemplo: Un coche que acelera desde 5 m/s  
v_inicial = 5 # velocidad inicial en m/s  
aceleracion = 2 # aceleracion en m/s^2
```

Significa:

1. El coche empieza moviéndose a 5 m/s
2. Aumenta su velocidad a razón de 2 m/s²

Es decir: cada segundo, aumenta 2 m/s.

▪ *Encabezado de la tabla*

```
print("Tiempo (s)\tPosicion (m)\tVelocidad (m/s)")
```

Ejemplo 49: Solo muestra la tabla con nombres de columnas.

▪ *Ciclo for para calcular valores cada 2 segundos*

```
for tiempo in range(0, 11, 2): # De 0 a 10 segundos cada 2 s
```

Ejemplo 50: Esto genera: 0, 2, 4, 6, 8, 10. Es decir, calcula el movimiento desde 0 hasta 10 segundos en pasos de 2 s.

▪ *Llamar a la función MRUA*

```
x, v = mrua(v_inicial, aceleracion, tiempo)
```

Ejemplo 51: Obtiene: la posición x y la velocidad v

▪ *Imprimir cada fila*

```
print(f"{tiempo}\t\t{x:.2f}\t\t{v:.2f}")
```

Ejemplo 52: Imprime el tiempo, la posición y la velocidad con 2 decimales.

■ Código completo

```
# Programa para calcular MRUA
def mrua(v0, a, t):
    # Parametros: v0=velocidad inicial, a=aceleracion, t=
    tiempo
    # Retorna: posicion y velocidad finales

    # Ecuacion de posicion:  $x = v_0 * t + (1/2) * a * t^2$ 
    posicion = v0 * t + 0.5 * a * t**2

    # Ecuacion de velocidad:  $v = v_0 + a * t$ 
    velocidad = v0 + a * t

    return posicion, velocidad

# Ejemplo: Un coche que acelera desde 5 m/s
v_inicial = 5 # velocidad inicial en m/s
aceleracion = 2 # aceleracion en m/s^2

print("Tiempo (s)\tPosicion (m)\tVelocidad (m/s)")
for tiempo in range(0, 11, 2): # De 0 a 10 segundos cada 2 s
    x, v = mrua(v_inicial, aceleracion, tiempo)
    print(f"{tiempo}\t\t{x:.2f}\t\t{v:.2f}")
```

Ejemplo 53: Movimiento Rectilíneo Uniformemente Acelerado

■ Resultado

Tiempo (s)	Posicion (m)	Velocidad (m/s)
0	0.00	5.00
2	14.00	9.00
4	36.00	13.00
6	66.00	17.00
8	104.00	21.00
10	150.00	25.00

2.5. Densidad y diccionarios

La densidad es una propiedad física que permite identificar sustancias y comparar materiales. Mediante el uso de Python, se puede calcular de manera rápida y precisa, además de automatizar la comparación con valores de referencia conocidos.

Para esta comparación, Python utiliza diccionarios, una estructura de datos clave–valor donde cada material se asocia con su densidad. Esto permite organizar datos reales y comparar fácilmente la densidad calculada con valores previamente definidos.

2.5.1. Descripción del Programa

Calcula la densidad a partir de masa y volumen, e identifica el material comparándolo con una base de datos. También haciendo uso de los diccionarios que nos permitirá comparar datos reales.

2.5.2. Fundamento Teórico

La densidad es una propiedad intensiva de la materia que relaciona la masa de un objeto con el volumen que ocupa. Se define como:

$$\rho = \frac{m}{V}$$

Donde:

- p = velocidad final (m/s)
- m = masa del objeto (kg)
- V = volumen del objeto (m³)

La densidad es característica de cada sustancia y permite identificar materiales. Por ejemplo:

- Agua: 1000 kg/m³
- Aluminio: 2700 kg/m³
- Hierro: 7874 kg/m³
- Oro: 19320 kg/m³

2.5.3. Ejemplo Practico

Un objeto de 2.7 kg y volumen 0.001 m³ tiene densidad 2700 kg/m³, coincidiendo con el aluminio.

2.5.4. Análisis del Problema

▪ *Pseudocódigo*

```
Inicio
  Definir masa <- 2.7
  Definir volumen <- 0.001           // [cite: 785, 788]

  densidad_calc <- masa / volumen    // [cite: 664]
```

```

Definir diccionario_materiales      // [cite: 677]
    Aluminio <- 2700
    Hierro   <- 7874
    Cobre    <- 8960                // [cite: 678, 679,
    681]

material_cercano <- Ninguno          // [cite: 694]
diferencia_minima <- Infinito        // [cite: 695]

Para cada material en diccionario_materiales // [
cite: 700]
    densidad_ref <- valor(material)
    diferencia <- ValorAbsoluto(densidad_calc -
densidad_ref) // [cite: 701, 705]

    Si diferencia < diferencia_minima Entonces // [
cite: 703, 706]
        diferencia_minima <- diferencia          // [
cite: 709]
        material_cercano <- nombre(material)      // [
cite: 710]
    FinSi
FinPara

Mostrar "Densidad calculada:", densidad_calc      // [
cite: 731]
Mostrar "Posible material:", material_cercano     // [
cite: 732]
Fin

```

Ejemplo 54: Pseudocódigo: Cálculo de densidad y comparación con materiales conocidos

■ *Función calcular densidad*

```

# Programa para calcular densidad
def calcular_densidad(masa, volumen):
    # Parametros: masa (kg), volumen (m^3)
    # Retorna: densidad (kg/m^3)

    # Formula de densidad: rho = m / V
    densidad = masa / volumen

    return densidad

```

Ejemplo 55: Define una función que recibe masa y volumen, despues se aplica su respectiva formula, asi devolviendo la densidad calculada.

$$\rho = \frac{m}{V}$$

■ *Función identificar material*

```
def identificar_material(densidad):
    # Identifica material segun su densidad
    # Parametros: densidad (kg/m^3)
    # Retorna: nombre del material mas cercano
```

Ejemplo 56: Recibe la densidad calculada y busca un material similar.

Diccionario de densidades de materiales

```
# Diccionario de densidades de materiales comunes
materiales = {
    'Aluminio': 2700,
    'Hierro': 7874,
    'Cobre': 8960,
    'Plomo': 11340,
    'Agua': 1000,
    'Madera': 600
}
```

Ejemplo 57: Guarda densidades típicas en kg/m³.

Variables para encontrar coincidencia

```
# Buscar el material mas cercano
material_cercano = None
diferencia_minima = float('inf')
```

Ejemplo 58: *material cercano*: almacenará el nombre del material más parecido.

float('inf'): representa un número muy grande, para poder comparar.

Bucle para comparar densidades

```
for nombre, dens_ref in materiales.items():
    diferencia = abs(densidad - dens_ref) # Calcular
    diferencia
    if diferencia < diferencia_minima:    # Si es mas
    cercano
        diferencia_minima = diferencia
        material_cercano = nombre
    return material_cercano
```

Ejemplo 59: Se calcula qué tan lejos está la densidad del material; Si es más cercana que las anteriores se guarda el material.

■ *Función identificar material*

```
for nombre, dens_ref in materiales.items():
    diferencia = abs(densidad - dens_ref) # Calcular
    diferencia
    if diferencia < diferencia_minima:    # Si es mas
    cercano
        diferencia_minima = diferencia
        material_cercano = nombre
return material_cercano
```

Ejemplo 60: Valores de masa y volumen en unidades correctas.

```
dens = calcular_densidad(masa_obj, volumen_obj)
material = identificar_material(dens)
```

Ejemplo 61: Se obtiene la densidad y luego el material probable.

■ *Imprimir resultados*

```
print(f"Densidad calculada: {dens:.2f} kg/m^3")
print(f"Posible material: {material}")
```

Ejemplo 62: *dens:.2f* → limita la densidad a 2 decimales.

■ *Código completo*

```
# Programa para calcular densidad
def calcular_densidad(masa, volumen):
    # Parametros: masa (kg), volumen (m^3)
    # Retorna: densidad (kg/m^3)

    # Formula de densidad: rho = m / V
    densidad = masa / volumen

    return densidad

def identificar_material(densidad):
    # Identifica material segun su densidad
    # Parametros: densidad (kg/m^3)
    # Retorna: nombre del material mas cercano

    # Diccionario de densidades de materiales comunes
    materiales = {
        'Aluminio': 2700,
        'Hierro': 7874,
```

```

        'Cobre': 8960,
        'Plomo': 11340,
        'Agua': 1000,
        'Madera': 600
    }

    # Buscar el material mas cercano
    material_cercano = None
    diferencia_minima = float('inf')

    for nombre, dens_ref in materiales.items():
        diferencia = abs(densidad - dens_ref) # Calcular
diferencia
        if diferencia < diferencia_minima:      # Si es mas
cercano
            diferencia_minima = diferencia
            material_cercano = nombre

    return material_cercano

# Ejemplo: Calcular densidad de un objeto
masa_obj = 2.7 # kg
volumen_obj = 0.001 # m^3 (1 litro)

dens = calcular_densidad(masa_obj, volumen_obj)
material = identificar_material(dens)

print(f"Densidad calculada: {dens:.2f} kg/m^3")
print(f"Posible material: {material}")

```

Ejemplo 63: Calculo de densidad

■ *Resultado*

```

Densidad calculada: 2700.00 kg/m^3
Posible material: Aluminio

```

2.6. Termodinámica: Calor específico

El concepto de calor específico es esencial en la termodinámica, ya que describe cómo diferentes materiales responden a los cambios de temperatura. El uso de programación permite simplificar los cálculos y aplicar este concepto a distintos escenarios térmicos.

2.6.1. Descripción del Programa

Calcula el calor necesario para modificar la temperatura de una sustancia.

2.6.2. Fundamento Teórico

El calor específico es la cantidad de energía necesaria para elevar la temperatura de 1 kg de una sustancia en 1 grado Celsius (o Kelvin). La ecuación fundamental de calorimetría es:

$$Q = mc\Delta T$$

Donde:

- Q = calor transferido (J)
- m = masa de la sustancia (kg)
- c = calor específico (J/(kg · K))
- $\Delta T = T_f - T_i$ = cambio de temperatura (K o °C)

Valores típicos de calor específico:

- Agua: $c = 4186 \text{ J/(kg} \cdot \text{K)}$
- Hierro: $c = 450 \text{ J/(kg} \cdot \text{K)}$
- Cobre: $c = 385 \text{ J/(kg} \cdot \text{K)}$
- Aluminio: $c = 900 \text{ J/(kg} \cdot \text{K)}$

2.6.3. Ejemplo Practico

Calentar 2 kg de agua de 20°C a 100°C requiere aproximadamente 670 kJ.

2.6.4. Análisis del Problema

- *Pseudocódigo*

```
Inicio
  Definir masa <- 2.0           // kg
  Definir c <- 4186             // J/(kg*C) (agua)
)
  Definir temperatura_i <- 20   // C
  Definir temperatura_f <- 100  // C
```

```

    delta_T <- temperatura_f - temperatura_i
    Q <- masa * c * delta_T

    Mostrar "Masa:", masa
    Mostrar "Calor necesario (J):", Q
    Mostrar "Calor en kJ:", Q / 1000
Fin

```

Ejemplo 64: Pseudocódigo: Cálculo de calorimetría

■ *Función calcular calor*

```

# Programa para calcular calor especifico
def calcular_calor(masa, calor_especifico, delta_T):
    # Parametros: masa (kg), calor_especifico (J/(kg*K)),
    delta_T (K)
    # Retorna: calor transferido (J)

```

Ejemplo 65: Define una función que recibe tres parámetros:

- masa → cantidad de sustancia (en kg)
- calor especifico → propiedad del material (J/(kg·K))
- delta T → cambio de temperatura

Cálculo dentro de la función

```

# Formula: Q = m * c * delta_T
calor = masa * calor_especifico * delta_T

return calor

```

Ejemplo 66: Usa la fórmula del calor sensible: $Q = m c \Delta T$ y regresa el valor del calor calculado.

■ *Definición de valores del ejemplo*

```

# Ejemplo: Calentar agua
masa_agua = 2.0      # kg
c_agua = 4186        # J/(kg*K) - calor especifico del agua
temp_inicial = 20    # grados Celsius
temp_final = 100     # grados Celsius

```

1. 2 kg de agua
2. 4186 J/(kg·K) → calor específico del agua

3. Temperatura va de 20°C a 100°C

■ *Cálculo de cambio de temperatura*

```
# Calcular cambio de temperatura
delta_temperatura = temp_final - temp_inicial
```

Ejemplo 67: $\Delta T = T_f - T_i = 100^\circ\text{C} - 20^\circ\text{C} = 80^\circ\text{C}$

■ *Cálculo del calor necesario*

```
# Calcular calor necesario
Q = calcular_calor(masa_agua, c_agua, delta_temperatura)
```

Ejemplo 68: Llama la función usando los datos del agua.

■ *Impresión de resultados*

```
print(f"Masa de agua: {masa_agua} kg")
print(f"Cambio de temperatura: {delta_temperatura} grados C")
print(f"Calor necesario: {Q:.2f} J")
print(f"Calor necesario: {Q/1000:.2f} kJ")
```

Ejemplo 69: Muestra el calor en julios (J) y también lo muestra convertido a kilojulios (kJ)

■ *Código completo*

```
# Programa para calcular calor especifico
def calcular_calor(masa, calor_especifico, delta_T):
    # Parametros: masa (kg), calor_especifico (J/(kg*K)),
    delta_T (K)
    # Retorna: calor transferido (J)

    # Formula: Q = m * c * delta_T
    calor = masa * calor_especifico * delta_T

    return calor

# Ejemplo: Calentar agua
masa_agua = 2.0          # kg
c_agua = 4186            # J/(kg*K) - calor especifico del agua
temp_inicial = 20        # grados Celsius
temp_final = 100         # grados Celsius
```

```
# Calcular cambio de temperatura
delta_temperatura = temp_final - temp_inicial

# Calcular calor necesario
Q = calcular_calor(masa_agua, c_agua, delta_temperatura)

print(f"Masa de agua: {masa_agua} kg")
print(f"Cambio de temperatura: {delta_temperatura} grados C")
print(f"Calor necesario: {Q:.2f} J")
print(f"Calor necesario: {Q/1000:.2f} kJ")
```

Ejemplo 70: Calculo de calor especifico

▪ *Resultado*

```
Masa de agua: 2.0 kg
Cambio de temperatura: 80 grados C
Calor necesario: 669760.00 J
Calor necesario: 669.76 kJ
```

2.7. Termodinámica: Equilibrio térmico

El equilibrio térmico ocurre cuando dos o más cuerpos alcanzan la misma temperatura después de intercambiar calor. Este fenómeno es común en sistemas reales y puede analizarse eficientemente mediante modelos matemáticos implementados en Python.

2.7.1. Descripción del Programa

Calcula la temperatura final de dos cuerpos que se mezclan térmicamente.

2.7.2. Fundamento Teórico

Cuando dos cuerpos a diferentes temperaturas se ponen en contacto, hay transferencia de calor del cuerpo más caliente al más frío hasta alcanzar el equilibrio térmico. Por el principio de conservación de la energía:

$$Q_{cedido} = Q_{ganado}$$

Para dos cuerpos en contacto:

$$m_1 c_1 (T_{eq} - T_1) = m_2 c_2 (T_{eq} - T_2)$$

Despejando la temperatura de equilibrio T_{eq} :

$$T_{eq} = \frac{m_1 c_1 T_1 + m_2 c_2 T_2}{m_1 c_1 + m_2 c_2}$$

Donde:

- T_{eq} = temperatura de equilibrio (K o °C)
- m_1, m_2 = masas de los cuerpos (kg)
- c_1, c_2 = calores específicos (J/(kg · K))
- T_1, T_2 = temperaturas iniciales (K o °C)

2.7.3. Ejemplo Practico

1 kg de agua a 80°C se mezcla con 2 kg a 20°C → Temperatura de equilibrio: 40°C

2.7.4. Análisis del Problema

▪ *Pseudocódigo*

```
Inicio
  Definir m1, c1, T1      // cuerpo caliente
  Definir m2, c2, T2      // cuerpo frio

  Teq <- (m1*c1*T1 + m2*c2*T2) / (m1*c1 + m2*c2)

  Q_cedido <- m1 * c1 * (T1 - Teq)
  Q_ganado <- m2 * c2 * (Teq - T2)

  Mostrar "Temperatura de equilibrio:", Teq
  Mostrar "Verificacion (Q_ced - Q_gan):",
  ValorAbsoluto(Q_cedido - Q_ganado)
Fin
```

Ejemplo 71: Pseudocódigo: Equilibrio térmico entre dos sustancias

▪ *Función temperatura equilibrio*

```
# Programa para calcular equilibrio termico
def temperatura_equilibrio(m1, c1, T1, m2, c2, T2):
  # Calcula temperatura de equilibrio entre dos cuerpos
  # Parametros: masas (m1, m2), calores especificos (c1, c2)
  ),
```

```
# temperaturas iniciales (T1, T2)
# Retorna: temperatura de equilibrio
```

Ejemplo 72: Esta función calcula la temperatura final común al mezclar dos cuerpos con temperaturas distintas.

Fórmula usada

$$m_1 c_1 (T_{eq} - T_1) = m_2 c_2 (T_{eq} - T_2)$$

Despejando:

$$T_{eq} = \frac{m_1 c_1 T_1 + m_2 c_2 T_2}{m_1 c_1 + m_2 c_2}$$

```
# Formula: m1*c1*(Teq - T1) + m2*c2*(Teq - T2) = 0
# Despejando Teq:
numerador = m1 * c1 * T1 + m2 * c2 * T2
denominador = m1 * c1 + m2 * c2

T_eq = numerador / denominador # Temperatura de
equilibrio

return T_eq
```

Ejemplo 73: Esto es lo que implementa el código

Función suma calores

```
def suma_calores(m1, c1, T1, m2, c2, T2):
    # Calcula intercambio de calor entre dos cuerpos
    # Retorna: calor_cedido, calor_ganado,
    temperatura_equilibrio
```

Ejemplo 74: Calcula cuánto calor: Cede el cuerpo caliente, gana el cuerpo frío y además devuelve la temperatura de equilibrio

```
# Calcular temperatura de equilibrio
T_eq = temperatura_equilibrio(m1, c1, T1, m2, c2, T2)
```

Ejemplo 75: Primero llama a la función anterior

```
# Calcular calor cedido (cuerpo caliente)
if T1 > T2:
    Q_cedido = m1 * c1 * (T1 - T_eq)
    Q_ganado = m2 * c2 * (T_eq - T2)
else:
    Q_cedido = m2 * c2 * (T2 - T_eq)
    Q_ganado = m1 * c1 * (T_eq - T1)

return Q_cedido, Q_ganado, T_eq
```

Ejemplo 76: Luego determina cuál cuerpo es el caliente

Calor perdido por el caliente:

$$Q_{cedido} = m_1 c_1 (T_1 - T_{eq})$$

Calor ganado por el frío:

$$Q_{ganado} = m_2 c_2 (T_{eq} - T_2)$$

Se calcula según qué cuerpo inició más caliente.

■ *Ejemplo práctico*

```
# Ejemplo: Mezclar agua caliente y fria
m_caliente = 1.0 # kg de agua caliente
T_caliente = 80 # grados Celsius
m_fria = 2.0 # kg de agua fria
T_fria = 20 # grados Celsius
c_agua = 4186 # J/(kg*K)
```

Ejemplo 77: Agua caliente y fría.

```
# Calcular equilibrio
Q_ced, Q_gan, T_final = suma_calores(
    m_caliente, c_agua, T_caliente,
    m_fria, c_agua, T_fria
)

print(f"Temperatura de equilibrio: {T_final:.2f} grados C")
print(f"Calor cedido: {Q_ced:.2f} J")
print(f"Calor ganado: {Q_gan:.2f} J")
print(f"Verificacion (deben ser iguales): {abs(Q_ced - Q_gan)
    :.2f} J")
```

Ejemplo 78: Se calcula y luego se imprimen los resultados.

Finalmente se verifica que: El calor cedido = calor ganado (se usa valor absoluto para que coincidan)

■ *Código completo*

```
# Programa para calcular equilibrio termico
def temperatura_equilibrio(m1, c1, T1, m2, c2, T2):
    # Calcula temperatura de equilibrio entre dos cuerpos
    # Parametros: masas (m1, m2), calores especificos (c1, c2)
    ),
    # temperaturas iniciales (T1, T2)
    # Retorna: temperatura de equilibrio
```

```

# Formula:  $m_1 \cdot c_1 \cdot (T_{eq} - T_1) + m_2 \cdot c_2 \cdot (T_{eq} - T_2) = 0$ 
# Despejando  $T_{eq}$ :
numerador = m1 * c1 * T1 + m2 * c2 * T2
denominador = m1 * c1 + m2 * c2

T_eq = numerador / denominador # Temperatura de
equilibrio

return T_eq

def suma_calores(m1, c1, T1, m2, c2, T2):
    # Calcula intercambio de calor entre dos cuerpos
    # Retorna: calor_cedido, calor_ganado,
    temperatura_equilibrio

    # Calcular temperatura de equilibrio
    T_eq = temperatura_equilibrio(m1, c1, T1, m2, c2, T2)

    # Calcular calor cedido (cuerpo caliente)
    if T1 > T2:
        Q_cedido = m1 * c1 * (T1 - T_eq)
        Q_ganado = m2 * c2 * (T_eq - T2)
    else:
        Q_cedido = m2 * c2 * (T2 - T_eq)
        Q_ganado = m1 * c1 * (T_eq - T1)

    return Q_cedido, Q_ganado, T_eq

# Ejemplo: Mezclar agua caliente y fria
m_caliente = 1.0 # kg de agua caliente
T_caliente = 80 # grados Celsius
m_fria = 2.0 # kg de agua fria
T_fria = 20 # grados Celsius
c_agua = 4186 # J/(kg*K)

# Calcular equilibrio
Q_ced, Q_gan, T_final = suma_calores(
    m_caliente, c_agua, T_caliente,
    m_fria, c_agua, T_fria
)

print(f"Temperatura de equilibrio: {T_final:.2f} grados C")
print(f"Calor cedido: {Q_ced:.2f} J")
print(f"Calor ganado: {Q_gan:.2f} J")
print(f"Verificacion (deben ser iguales): {abs(Q_ced - Q_gan)
:.2f} J")

```

Ejemplo 79: Equilibrio térmico y suma de calores

■ *Resultado*

```
Temperatura de equilibrio: 40.00 grados C  
Calor cedido: 167440.00 J  
Calor ganado: 167440.00 J  
Verificación (deben ser iguales): 0.00 J
```

3. Programas de Física

En esta sección se presentan programas que abordan conceptos matemáticos fundamentales mediante algoritmos implementados en Python. Estos ejemplos muestran cómo la programación puede utilizarse como una herramienta para explorar y comprender estructuras matemáticas.

3.1. Secuencias Matemáticas: Fibonacci

La secuencia de Fibonacci es un ejemplo clásico de una relación matemática recursiva. Su simplicidad y sus múltiples aplicaciones la convierten en un caso ideal para ser implementado y analizado mediante programación.

3.1.1. Descripción del Programa

Genera los primeros n términos de la secuencia de Fibonacci y calcula la razón entre términos consecutivos.

3.1.2. Fundamento Teórico

La sucesión de Fibonacci es una secuencia matemática definida por la siguiente relación de recurrencia:

$$F(n) = F(n-1) + F(n-2)$$

con valores iniciales:

$$F(0) = 0, \quad F(1) = 1$$

Los primeros términos son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Una propiedad interesante es que la razón entre términos consecutivos converge al número áureo:

$$\lim_{n \rightarrow \infty} \frac{F(n+1)}{F(n)} = \phi = \frac{1 + \sqrt{5}}{2} \approx 1,618$$

Esta secuencia tiene aplicaciones en:

1. Patrón de crecimiento de poblaciones
2. Filotaxis (disposición de hojas en plantas)
3. Espirales en conchas y galaxias
4. Análisis de algoritmos

3.1.3. Ejemplo Practico

El programa genera los primeros 15 términos y muestra la aproximación al número áureo.

3.1.4. Análisis del Problema

■ *Pseudocódigo*

```

Inicio
  Definir n <- 15

  Crear lista secuencia
  Agregar 0 a secuencia
  Agregar 1 a secuencia

  Para i <- 2 Hasta n
    nuevo_termino <- secuencia[i-1] + secuencia[i-2]
    Agregar nuevo_termino a secuencia
  FinPara

  Si n > 2 Entonces
    razon_aurea <- secuencia[n] / secuencia[n-1]
  FinSi

  Mostrar secuencia
  Mostrar "Razon aurea aproximada:", razon_aurea
Fin

```

Ejemplo 80: Pseudocódigo: Sucesión de Fibonacci y convergencia áurea

■ *Función fibonacci(n)*

```
# Secuencia de Fibonacci
def fibonacci(n):
    # Genera los primeros n numeros de Fibonacci
    # Parametros: n = cantidad de terminos a generar
    # Retorna: lista con los numeros de Fibonacci

    secuencia = [] # Lista para guardar la secuencia
```

Ejemplo 81: Genera los primeros n números de la sucesión de Fibonacci. En su Inicialización se crea una lista vacía donde se guardará la secuencia.

Casos base

La serie Fibonacci empieza con:

1. $F(0) = 0$
2. $F(1) = 1$

```
# Casos base
if n >= 1:
    secuencia.append(0) # F(0) = 0
if n >= 2:
    secuencia.append(1) # F(1) = 1
```

Ejemplo 82: Se agregan esos valores solo si corresponden a la cantidad pedida.

Generación de los demás términos

```
# Generar terminos siguientes
for i in range(2, n):
    # Cada termino es la suma de los dos anteriores
    siguiente = secuencia[i-1] + secuencia[i-2]
    secuencia.append(siguiente)

return secuencia
```

Ejemplo 83: Cada número nuevo es la suma de los dos anteriores y su retorno devuelve una lista con los n valores generados

Ejemplo:

- $F(2) = F(1) + F(0)$
- $F(3) = F(2) + F(1)$
- etc.

Ejemplo de uso

```
# Ejemplo
n_terminos = 15
fib_seq = fibonacci(n_terminos)

print(f"Primeros {n_terminos} numeros de Fibonacci:")
for i, valor in enumerate(fib_seq):
    print(f"F({i}) = {valor}")
```

Ejemplo 84: Genera 15 números de Fibonacci y luego imprime cada número con su índice

■ *Cálculo de la razón áurea*

```
# Calcular razon aurea aproximada
if n_terminos > 2:
    razon = fib_seq[-1] / fib_seq[-2]
    print(f"\nRazon aurea aproximada: {razon:.6f}")
    print(f"Valor exacto: {(1 + 5**0.5) / 2:.6f}")
```

Ejemplo 85: La razón áurea se aproxima dividiendo números consecutivos de Fibonacci: $\phi = \frac{F(n+1)}{F(n)}$.

El valor exacto es:

$$\phi = \frac{1 + \sqrt{5}}{2}$$

■ *Código completo*

```
# Secuencia de Fibonacci
def fibonacci(n):
    # Genera los primeros n numeros de Fibonacci
    # Parametros: n = cantidad de terminos a generar
    # Retorna: lista con los numeros de Fibonacci

    secuencia = [] # Lista para guardar la secuencia

    # Casos base
    if n >= 1:
        secuencia.append(0) # F(0) = 0
    if n >= 2:
        secuencia.append(1) # F(1) = 1

    # Generar terminos siguientes
    for i in range(2, n):
        # Cada termino es la suma de los dos anteriores
        siguiente = secuencia[i-1] + secuencia[i-2]
```

```

        secuencia.append(siguiete)

    return secuencia

# Ejemplo
n_terminos = 15
fib_seq = fibonacci(n_terminos)

print(f"Primeros {n_terminos} numeros de Fibonacci:")
for i, valor in enumerate(fib_seq):
    print(f"F({i}) = {valor}")

# Calcular razon aurea aproximada
if n_terminos > 2:
    razon = fib_seq[-1] / fib_seq[-2]
    print(f"\nRazon aurea aproximada: {razon:.6f}")
    print(f"Valor exacto: {(1 + 5**0.5) / 2:.6f}")

```

Ejemplo 86: Secuencia de Fibonacci

■ *Resultado*

```

Primeros 15 numeros de Fibonacci:
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
F(11) = 89
F(12) = 144
F(13) = 233
F(14) = 377

Razon aurea aproximada: 1.618026
Valor exacto: 1.618034

```

3.2. Razón áurea

La razón áurea es un número irracional que aparece con frecuencia en Matemáticas, Física y en diversos fenómenos naturales. Se representa con la letra griega ϕ y su valor aproximado es:

$$\phi \approx 1,618$$

Una forma de obtener una aproximación de la razón áurea es a partir de la sucesión de Fibonacci, donde la razón entre dos términos consecutivos tiende al valor de ϕ conforme la sucesión avanza.

3.2.1. Fundamento Teórico

La sucesión de Fibonacci se define de manera recursiva, y al dividir un término entre el anterior, el resultado se aproxima progresivamente a la razón áurea. Matemáticamente, se cumple que:

$$\phi = \lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n}$$

El valor exacto de la razón áurea puede calcularse mediante la expresión:

$$\phi = \frac{1 + \sqrt{5}}{2}$$

3.2.2. Ejemplo Práctico

En este ejemplo se compara el valor aproximado de la razón áurea obtenido a partir de la sucesión de Fibonacci con su valor teórico real.

3.2.3. Análisis del Programa

El programa utiliza la librería `math` para calcular el valor real de la razón áurea mediante la raíz cuadrada. Posteriormente, se calcula una aproximación dividiendo los dos últimos términos de una lista que contiene la sucesión de Fibonacci previamente calculada.

El objetivo es mostrar cómo Python permite comparar resultados numéricos aproximados con valores teóricos exactos.

■ *Pseudocódigo*

```

Inicio
  Obtener fib_ultimo de la secuencia
  Obtener fib_penultimo de la secuencia

  phi_aprox <- fib_ultimo / fib_penultimo
  phi_real <- (1 + raiz_cuadrada(5)) / 2

  Mostrar "Phi aproximado:", phi_aprox
  Mostrar "Phi real:", phi_real
Fin

```

Ejemplo 87: Pseudocódigo: Comparación de phi real vs phi aproximado

■ *Importación de la librería matemática*

```
import math
```

Ejemplo 88: Importación de la librería `math` para cálculos matemáticos.

■ *Cálculo de la razón áurea*

```

phi_aprox = fib[-1] / fib[-2]
phi_real = (1 + math.sqrt(5)) / 2

```

Ejemplo 89: Cálculo de la razón áurea aproximada y comparación con el valor real.

■ *Impresión de resultados*

```

print("phi aproximada:", phi_aprox)
print("phi real:", phi_real)

```

Ejemplo 90: Impresión del valor aproximado y del valor real de la razón áurea.

El programa muestra que la razón entre términos consecutivos de la sucesión de Fibonacci se aproxima al valor real de la razón áurea, confirmando su convergencia y demostrando la relación entre programación y conceptos matemáticos fundamentales.

■ *Código completo*

```

# Calculo de la razon aurea usando la sucesion de Fibonacci
import math

# Generar sucesion de Fibonacci
fib = [1, 1]
n = 10 # cantidad de terminos

for i in range(2, n):
    fib.append(fib[i-1] + fib[i-2])

# Razon aurea aproximada
phi_aprox = fib[-1] / fib[-2]

# Razon aurea real
phi_real = (1 + math.sqrt(5)) / 2

# Mostrar resultados
print("phi aproximada:", phi_aprox)
print("phi real:", phi_real)

```

Ejemplo 91: Secuencia de Fibonacci

▪ *Resultado*

```

phi aproximada: 1.6176470588235294
phi real: 1.618033988749895

```

4. Proceso general para resolver problemas con Python

1. Entender el problema.
2. Identificar datos y ecuaciones.
3. Plantear el modelo matemático.
4. Implementar el código en Python.
5. Ejecutar el programa.
6. Analizar los resultados.
7. Verificar y mejorar la solución.

5. Referencias

1. Zapotecatl López, J.L. (2018). *Introducción al pensamiento computacional: conceptos básicos para todos*. Academia Mexicana de Computación.
2. Wing, J. (2006). *Computational thinking*. Communications of the ACM, 49(3), 33-35.
3. Morales Gamboa, R., et al. (2021). *Pensamiento Computacional en México*. Academia Mexicana de Computación.
4. Brennan, K., & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking*. Proceedings of the 2012 Annual Meeting of the American Educational Research Association.
5. Serway, R. A., & Jewett, J. W. (2014). *Física para ciencias e ingeniería*. Cengage Learning.
6. Giancoli, D. C. (2008). *Física: principios con aplicaciones*. Pearson Educación.
7. Cengel, Y. A., & Boles, M. A. (2015). *Termodinámica*. McGraw-Hill.
8. Rosen, K. H. (2012). *Matemática discreta y sus aplicaciones*. McGraw-Hill.