Josue Santana Robledo Corona 325073061

Maestría en Ciencias de la Robótica e Inteligencia Artificial

Algoritmos Bio-Inspirados

Centro Universitario de Ciencias Exactas e Ingeniería.

Mtro. Carlos Alberto López Franco

1. Introducción

Este reporte explora la implementación de este algoritmo, que se distingue por utilizar la proporción áurea para reducir el intervalo de búsqueda de manera óptima en cada iteración, minimizando así el número de evaluaciones de la función requeridas. En las siguientes secciones, se desglosa el funcionamiento del algoritmo, se presenta un análisis de su rendimiento aplicado a una función de prueba y se discuten sus ventajas y limitaciones en comparación con otros métodos.

2. Descripción del código

El código implementa el algoritmo de Golden Search optimizado con proporción áurea para encontrar el mínimo de una función dentro de un intervalo dado. El funcionamiento consiste en dividir el intervalo de búsqueda usando la proporción áurea (aprox. 0.618), reduciendo el espacio de búsqueda de manera más eficiente en cada iteración hasta alcanzar la precisión deseada. En esta versión del código no se reutilizan variables, se vuelven a calcular en cada ciclo.

3. Componentes del Código

Variables y parámetros iniciales

```
Rango=[-4,8]
a0,b0= Rango[0],Rango[1]
fi=((-1+sqrt(5))/2)
ro=1-fi
presicion = 0.0001
ciclo=0
```

Función Objetivo

```
def funcion(x):
  return (x-2)**2 + (0.5*x)
```

Reutilizamos función del algoritmo anterior.

4. Algoritmo Principal

El algoritmo sigue estos pasos en cada iteración:

División del intervalo en proporción aurea:

```
a1 = a0 + ro * (b0 - a0)  # Punto en 38.2% del intervalo
b1 = a0 + fi * (b0 - a0)  # Punto en 61.8% del intervalo
```

Evaluación y decisión:

```
    if funcion(a1) > funcion(b1):
      a0 = a1
      estado = "Decrece"
    else:
```

b0 = b1
estado = "Crece"

Aquí ve el código si la función crece o decrece y en base a ello define sus nuevos límites. Funciona igual que el algoritmo anterior

Criterio de parada: El ciclo continúa hasta que el tamaño del intervalo sea menor que 0.0001.

5. Resultados Obtenidos

El código genera una tabla detallada que muestra en cada iteración

23 1.7499 1.7501 0.0002 0.0001 0.0001 1.7499 1.7499 0.9375 0.9375 Decrece
24 1.7499 1.7501 0.0001 0.0000 0.0001 1.7499 1.7500 0.9375 0.9375 Decrece
25 1.7499 1.7500 0.0001 0.0000 0.0000 1.7500 1.7500 0.9375 0.9375 Crece
Mínimo en x ≈ 1.7499803071307691, f(x) ≈ 0.9375000003878091

En esta tabla podemos ver el número de ciclos y las variables que se fueron utilizando durante el código, lo que nos ayuda a ver la evolución del código hacia encontrar el mínimo. Y finalmente vemos el valor final al que llega y su evaluación en la función.

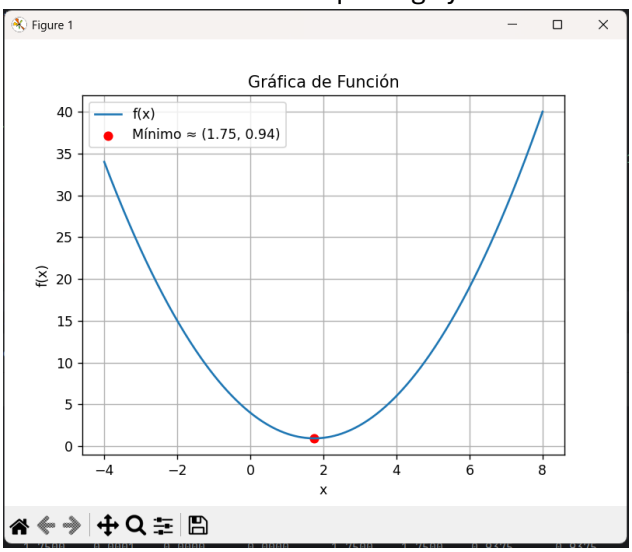

Imagen 1
Gráfica resultante

6. Ventajas del Método

Su principal ventaja es que es sencillo de entender y aplicar, puede no ser el más eficiente ya que en cada ciclo vuelve a calcular todas sus variables, pero eventualmente

llega al mínimo. Además de que resulta ser más eficiente que el código anterior, haciendo 25 vuetas vs 29 del Ternary, lo que es una optimización, pero aún podemos eficientarlo.

7. Limitaciones

Aunque más eficiente, el método conserva algunas limitaciones, como que aún puede encontrar solo un mínimo a la vez, o calcula sus variables con cada ciclo.

8. Conclusiones

El código implementa correctamente el algoritmo de búsqueda Golden y demuestra su efectividad para encontrar el mínimo de la función cuadrática especificada. Los resultados obtenidos coinciden con el anterior código, lo que valida su funcionamiento, ahora toma menos ciclos, lo que lo hace una mejora respecto al algoritmo anterior, tardo 4 ciclos menos en encontrar el mínimo.Implementarlo no fue nada difícil, ya que fue casi un código igual al anterior, solo hizo falta agregar unas variables más y cambiar la forma de calcular a1 y b1.

9. Código

```
import matplotlib.pyplot as plt

import numpy as np

from math import sqrt


# Definir el rango inicial de búsqueda [a0, b0]

Rango = [-4, 8]

a0, b0 = Rango[0], Rango[1]  # Extraer los límites inferior y superior


# Calcular la proporción áurea (ϕ) y su complemento (ρ)

# ϕ = (√5 - 1)/2 ≈ 0.618 (proporción áurea)

# ρ = 1 - ϕ ≈ 0.382

fi = ((-1 + sqrt(5)) / 2)  # ϕ (phi) - razón áurea

ro = 1 - fi  # ρ (rho) - complemento de la razón áurea


# Parámetros del algoritmo

presicion = 0.0001  # Precisión deseada para la convergencia

ciclo = 0  # Contador de iteraciones
```

```
# Función objetivo que queremos minimizar: f(x) = (x-2)^2 + 0.5*x
def funcion(x):
    return (x-2)**2 + (0.5*x)

# NOTA: Hay un error en el encabezado de la tabla - muestra "1/3 delta" y "2/3 delta"
# pero en realidad este es el método de la sección dorada, no de trisección
print(f"{'Ciclo':<6}{'a0':<10}{'b0':<10}{'delta':<10}{'1/3 delta':<12}{'2/3 delta':<12}{'a1':<10}{'b1':<10}{'f(a1)':<12}{'f(b1)':<12}{'Estado':<12}")
print("-" * 110)

# Bucle principal del algoritmo de la sección dorada (Golden Section Search)
while b0 - a0 > presicion:  # Continuar mientras el intervalo sea mayor que la precisión
    # Calcular puntos usando la proporción áurea
    a1 = a0 + ro * (b0 - a0)  # Punto a 38.2% del intervalo (ρ)
    b1 = a0 + fi * (b0 - a0)  # Punto a 61.8% del intervalo (ϕ)

    # Comparar los valores de la función en los puntos a1 y b1
    if funcion(a1) > funcion(b1):
        # Si f(a1) > f(b1), el mínimo está en [a1, b0]
        a0 = a1  # Descartar el subintervalo izquierdo [a0, a1]
        estado = "Decrece"  # La función está decreciendo hacia la derecha
    else:
        # Si f(a1) <= f(b1), el mínimo está en [a0, b1]
        b0 = b1  # Descartar el subintervalo derecho [b1, b0]
        estado = "Crece"  # La función está creciendo hacia la derecha

    ciclo += 1  # Incrementar el contador de iteraciones

    # Imprimir resultados de la iteración actual
```

```
    # NOTA: Las columnas "1/3 delta" y "2/3 delta" son engañosas ya que
    # este no es el método de trisección sino el de la sección dorada
    print(f"{ciclo:<6}{a0:<10.4f}{b0:<10.4f}{(b0 - a0):<10.4f}{(b0 - a0)/3:<12.4f}{2*(b0 - a0)/3:<12.4f}{a1:<10.4f}{b1:<10.4f}{funcion(a1):<12.4f}{funcion(b1):<12.4f}{estado}")

# Una vez alcanzada la precisión deseada, calcular el mínimo aproximado
xmin = (a0 + b0) / 2  # Tomar el punto medio del intervalo final como aproximación
ymin = funcion(xmin)   # Evaluar la función en el punto mínimo

# Mostrar el resultado final
print(f"Mínimo en x ≈ {xmin}, f(x) ≈ {funcion(xmin)}")

# Crear visualización de la función y el mínimo encontrado
x = np.linspace(Rango[0], Rango[1], 400)  # Generar 400 puntos en el rango original
y = funcion(x)  # Calcular los valores de la función en esos puntos

# Configurar y mostrar la gráfica
plt.plot(x, y, label="f(x)")  # Graficar la función
plt.scatter(xmin, ymin, color="red", label=f'Mínimo ≈ ({xmin:.2f}, {ymin:.2f})')  # Marcar el mínimo
plt.title("Gráfica de Función y Mínimo Encontrado")  # Título del gráfico
plt.xlabel("x")  # Etiqueta del eje X
plt.ylabel("f(x)")  # Etiqueta del eje Y
plt.legend()  # Mostrar leyenda
plt.grid(True)  # Activar grid
plt.show()  # Mostrar el gráfico
```