Carrera:

Licenciatura en Sistemas de Información

Tema:

Raíces De Ecuaciones

Asignatura:

Métodos Numéricos

Equipo Docente:

Maria Isabel Sanchez

Alumnos:

Gonzalez Coene, Alejandro Rafael LU: 56622
Canteros, Luciana Belen LU: 56889

2024

Índice

Introducción

Este informe compara dos métodos para calcular raíces de funciones no lineales: el método de Newton-Raphson y el de Intervalo Medio. Ambos métodos son bastante útiles para mejorar la precisión y acelerar la convergencia en la resolución de ecuaciones complejas a través de procesos iterativos, lo que facilita su uso en cálculos numéricos. A lo largo de este trabajo, implementamos y comparamos ambos métodos en Python, mostrando sus resultados, ventajas y desventajas en términos de precisión y eficiencia.

Método Newton Rapson

Definición

El método de Newton-Raphson es un método numérico utilizado para encontrar aproximaciones de las raíces de una función no lineal. A partir de una estimación inicial cercana a la raíz, el método utiliza la derivada de la función para ajustar iterativamente esta estimación. En cada paso, se calcula una nueva aproximación siguiendo la fórmula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

donde x_{n+1} es la aproximación actual, $f(x_n)$ es el valor de la función en ese punto, y $f'(x_n)$ es la derivada de la función en ese mismo punto. Al repetir este proceso, la estimación se va acercando a la raíz con gran rapidez.

Ventajas

Convergencia rápida: Convergencia cuadrática cercana a la raíz, reduciendo el error exponencialmente en cada iteración.

Implementación sencilla: Requiere solo la función y su derivada, facilitando su uso en código.

Aplicabilidad amplia: Funciona con muchas funciones no lineales siempre que sean diferenciables.

Eficiencia computacional: Alcanza soluciones precisas en pocas iteraciones, ahorrando tiempo y recursos.

Limitaciones

Requiere derivada: No funciona si la derivada es cero o difícil de calcular.

Dependencia del punto inicial: Si el inicio está lejos de la raíz, puede divergir o hallar raíces incorrectas.

Solo para funciones diferenciables: No aplica a funciones no continuas o no derivables.

Posibles ciclos infinitos: Puede repetirse sin converger en funciones con derivadas cercanas a cero.

Código Fuente

En este proyecto, hemos organizado el código del método de Newton-Raphson en módulos separados, estructurando cada funcionalidad en archivos independientes para facilitar su mantenimiento y comprensión. Esta modularización permite enfocar cada archivo en una tarea específica, mejorando la claridad y escalabilidad del código. La estructura de archivos es la siguiente:

conditionFourier.py: Contiene funciones para analizar y aplicar condiciones de Fourier, que facilitan la evaluación de los puntos de convergencia en el método de Newton-Raphson.

newtonRaphson.py: Implementa el núcleo del método de Newton-Raphson, con funciones clave para realizar iteraciones y aproximaciones de raíces.

plot_graf.py: Gestiona la visualización gráfica de los resultados, permitiendo observar la convergencia y comportamiento de las iteraciones mediante gráficos.

validation_inputs.py: Incluye validaciones de entrada para asegurar que los datos iniciales cumplan con los requisitos del método y se eviten errores en la ejecución.

Esta organización modular facilita futuras modificaciones y permite la reutilización de funciones en otros proyectos donde se necesiten métodos de cálculo numérico o visualización de resultados.

conditionFourier.py

```python
import sympy as sp


def fourier(f, x0, x1) -> dict[float:bool]:
    """
     Evalúa las condiciones de Fourier en los puntos `x0` y `x1` para una función `f`.
```

```
    Args:
        f (callable): Función a evaluar.
        x0 (float): Punto inicial para evaluar.
        x1 (float): Punto final para evaluar.

    Returns:
        dict[float:bool]: Un diccionario que indica si cada punto cumple con la condición.
    """
    x = sp.symbols('x')  # Define la variable simbólica `x`
    f_simb = f(x)  # Convierte `f` a una función simbólica
    result = {x0: False, x1: False}  # Inicializa el diccionario de resultados

    if f(x0) * f(x1) > 0:# Verifica si f(x0) y f(x1)tiene signos iguales
        result

    primera_derivada = sp.diff(f_simb, x)  # Calcula la 1ra derivada de `f`
    segunda_derivada = sp.diff(primera_derivada, x)  # Calcula la 2da derivada de f
    ddf = sp.lambdify(x, segunda_derivada)  # Convierte la 2da derivada en función evaluable

    if f(x0) * ddf(x0) > 0:  # Evalúa la condición de Fourier en `x0`
        result[x0] = True  # Marca `x0` como que cumple la condición
    if f(x1) * ddf(x1) > 0:  # Evalúa la condición de Fourier en `x1`
```

```
        result[x1] = True  # Marca `x1` como que cumple la condición


    return result  # Devuelve el diccionario con los resultados
```

plot_graf.py

```python
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt



def plot_newtonraphson_results(f, table: pd.DataFrame, execution_time):

    """

     Genera un gráfico de las iteraciones del método de Newton-Raphson junto con los puntos calculados.



    Args:

        f (callable): Función a graficar.

          table (pd.DataFrame): Tabla con las iteraciones de Newton-Raphson, que contiene las columnas 'x' y 'f(x)'.

            execution_time (float): Tiempo de ejecución para mostrar en el gráfico.

    """

    x_values = table['x']  # Extrae los valores de 'x' de la tabla

    fx_values = table['f(x)']  # Extrae los valores de 'f(x)' de la tabla
```

```
    x_range = np.linspace(min(x_values) - 0.05, max(x_values) + 0.05, 1500)
# Define el rango de x para graficar f(x)

    y_range = f(x_range)  # Evalúa f(x) en el rango definido



    plt.figure(figsize=(8, 6))  # Crea la figura para el gráfico

    plt.plot(x_range, y_range, label='f(x)', color='r', linewidth=2)  #
Grafica f(x) en el rango definido



    plt.plot(x_values, fx_values, marker='o', linestyle='-', color='b',
label='Newton-Raphson Puntos')  # Grafica los puntos de Newton-Raphson



    plt.title('Newton-Raphson Iteraciones')  # Título del gráfico

    plt.xlabel('x')  # Etiqueta del eje x

    plt.ylabel('f(x)')  # Etiqueta del eje y

    plt.grid(True)  # Activa la cuadrícula

    plt.legend()  # Muestra la leyenda



    plt.text(0.95, 0.95, f'Time: {execution_time:.5f} s', fontsize=12,
color='green',

             verticalalignment='top', horizontalalignment='right',

             transform=plt.gcf().transFigure, bbox=dict(facecolor='white',
alpha=0.5))  # Muestra el tiempo de ejecución en el gráfico



    plt.savefig("images/grafico_newton_raphson.png", format='png', dpi=300)
# Guarda el gráfico como imagen
```

```python
def plot_function(f):
    """
    Genera un gráfico simple de la función f(x).


    Args:
        f (callable): Función a graficar.
    """
    x_vals = np.linspace(-1, 1, 30)  # Define valores de x para graficar
    y_vals = f(x_vals)  # Evalúa f(x) en el rango definido
    plt.figure(figsize=(8, 6))  # Crea la figura para el gráfico
      plt.plot(x_vals, y_vals, label='f(x)', color='blue', linewidth=2)  # Gráfica f(x) en el rango definido


    plt.title('Grafico de la Función')  # Título del gráfico
    plt.xlabel('x')  # Etiqueta del eje x
    plt.ylabel('f(x)')  # Etiqueta del eje y
    plt.grid(True)  # Activa la cuadrícula
    plt.legend()  # Muestra la leyenda


     plt.savefig("images/graf_function.png", format='png', dpi=300)  # Guarda el gráfico como imagen
```

validation_inputs.py

Python

```
import sympy as sp

from sympy.core.sympify import SympifyError



def validar_funcion():

    """

       Solicita al usuario una función con variable 'x' y la valida, devolviendo la función y su derivada.



    Returns:

        Tuple (f, df): Función lambdificada y su derivada.

    """

    x = sp.symbols('x')  # Define la variable simbólica `x`

    while True:

         f_input = input("Digite la función (con variable x): ")  # Solicita la función al usuario



        try:

                f_simb = sp.sympify(f_input)  # Convierte la entrada a una expresión simbólica

                df_simb = sp.diff(f_simb, x)  # Calcula la derivada de la función



                 f = sp.lambdify(x, f_simb)  # Lambdifica `f_simb` para evaluación numérica

                df = sp.lambdify(x, df_simb)  # Lambdifica `df_simb` para evaluación numérica
```

```
            if not f_simb.has(x):  # Verifica si la función depende de `x`
                print("Error: La función debe depender de la variable 'x'.")
                print("Digite una función válida (ej. 'sqrt(x) + x**2')\n")
                continue  # Solicita una nueva entrada si no depende de `x`
            return f, df  # Retorna la función y su derivada lambdificadas
        except SympifyError:  # Maneja errores de entrada inválida
            print("Error: El input no es una función válida. Inténtalo de nuevo.")
            print("Digite una función válida (ej. 'sqrt(x) + x**2')\n")


def validar_error() -> float:
    """
    Solicita un valor de error al usuario, validando que sea un número menor o igual a 0.001.

    Returns:
        float: El valor de error validado.
    """
    while True:
        e_input = input("\nIngrese el error (default 0.001):")  # Solicita el error al usuario
        if e_input == "":
            return 0.001  # Asigna el valor predeterminado si el usuario no ingresa nada
        try:
            error = float(e_input)  # Convierte el input a `float`
```

```
                if error > 0.001:  # Verifica que el error sea menor o igual a
0.001
                    print("El error debe ser menor a 0.001")
                    continue
                return error  # Retorna el error validado
            except ValueError:  # Maneja errores de conversión de input inválido
                print("Error no válido, por favor ingrese un número.")


def validar_x(i: int) -> float:
    """
    Solicita y valida un valor numérico de `x` al usuario.

    Args:
        i (int): Índice de `x` para etiquetar la solicitud.

    Returns:
        float: El valor numérico de `x` validado.
    """
    while True:
        x_input = input(f"ingrese el x_{i}:")# Pide el valor de x al user
        try:
            x = float(x_input)  # Convierte el input a `float`
            return x  # Retorna el valor validado
        except ValueError:  # Maneja errores de conversión de input inválido
            print(f"x{i} no válido, por favor ingrese un número.")
```

newtonRaphson.py

Importando las funciones anteriores se usa el archivo newtonRaphson.py como el archivo main donde se ejecutan todas las funciones

```python
import pandas as pd

from time import time

from validacition_inputs import validar_funcion, validar_error, validar_x

from plot_graf import plot_newtonraphson_results, plot_function

from conditionFourier import fourier
```

```python
def NewtonRaphson():

    """

      Implementa el método de Newton-Raphson para encontrar una raíz de la función ingresada, verificando condiciones previas.

    Genera una tabla de iteraciones y un gráfico del proceso.

    """

    f, df = validar_funcion()  # Solicita y valida la función y su derivada



    plot_function(f)  # Grafica la función ingresada
```

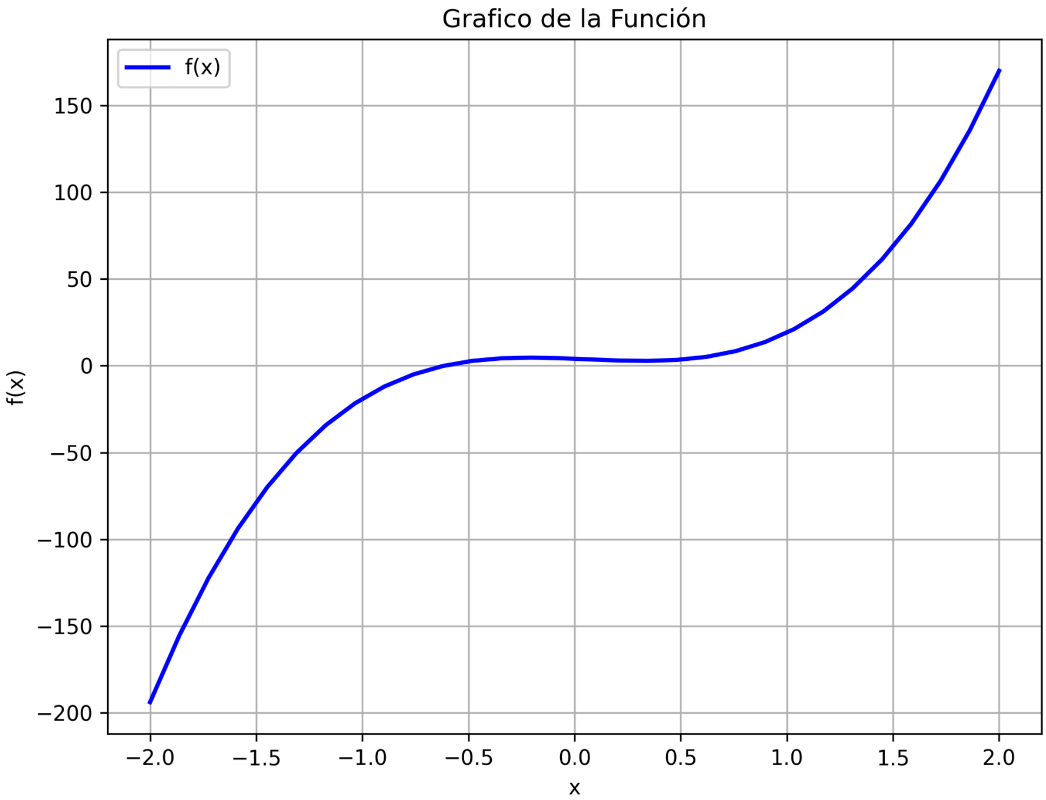
```python
    x0 = validar_x(i=0)  # Solicita y valida el valor inicial `x0`

    x1 = validar_x(i=1)  # Solicita y valida el valor inicial `x1`


    # Verifica la condición de Fourier en los puntos `x0` y `x1`

    fourier_result = fourier(f, x1=x1, x0=x0)


    value = next((clave for clave, valor in fourier_result.items() if valor), None)  # Busca el primer punto que cumple la condición


    if value:  # Si existe un punto que cumple la condición de Fourier

        print('Se cumple la condición de Fourier correctamente!!')

        print(f'Se inicializa la iteración con {value}')

        error = validar_error()  # Solicita y valida el error permitido

        x0 = value  # Establece `x0` como el punto que cumple la condición

        x1 = None  # Limpia `x1` para la iteración


        count = 0  # Inicializa el contador de iteraciones

        table = pd.DataFrame(columns=["x", "f(x)", "e"])  # Crea un DataFrame para registrar las iteraciones para almacenarlo como tabla

        row = [x0, f(x0), ""]  # Primera fila con el valor inicial

        table.loc[len(table)] = row

        start = time()  # Inicia el contador de tiempo


        while True:

            x1 = x0 - f(x0) / df(x0)  # Aplica la fórmula de Newton-Raphson
```

```
            error_actual = abs(x1 - x0)  # Calcula el error actual

            row = [x1, f(x1), error_actual]# Guarda el resultado en una fila

            table.loc[len(table)] = row

            if error_actual < error:  # Verifica el error

                break  # Finaliza el ciclo si se cumple el criterio de error

            x0 = x1  # Actualiza `x0` para la siguiente iteración

            count += 1  # Incrementa el contador


        end = time()  # Registra el tiempo final

        print(table)  # Muestra la tabla de resultados

        print(f"Tiempo de ejecución: {end-start} seg")


        plot_newtonraphson_results(

            f=f,

            table=table,

            execution_time=end-start  # Grafica los resultados de la iteración

        )

        table.to_csv('data/table_newton_raphson.csv')  # Guarda la tabla de iteraciones en un archivo CSV

    else:

        print("x0 y x1 no cumplen con la condición de Fourier")  # Mensaje de error si no se cumple la condición


if __name__ == '__main__':
```

```
    NewtonRaphson()  # Llama a la función principal si el script se ejecuta directamente
```

Resultados

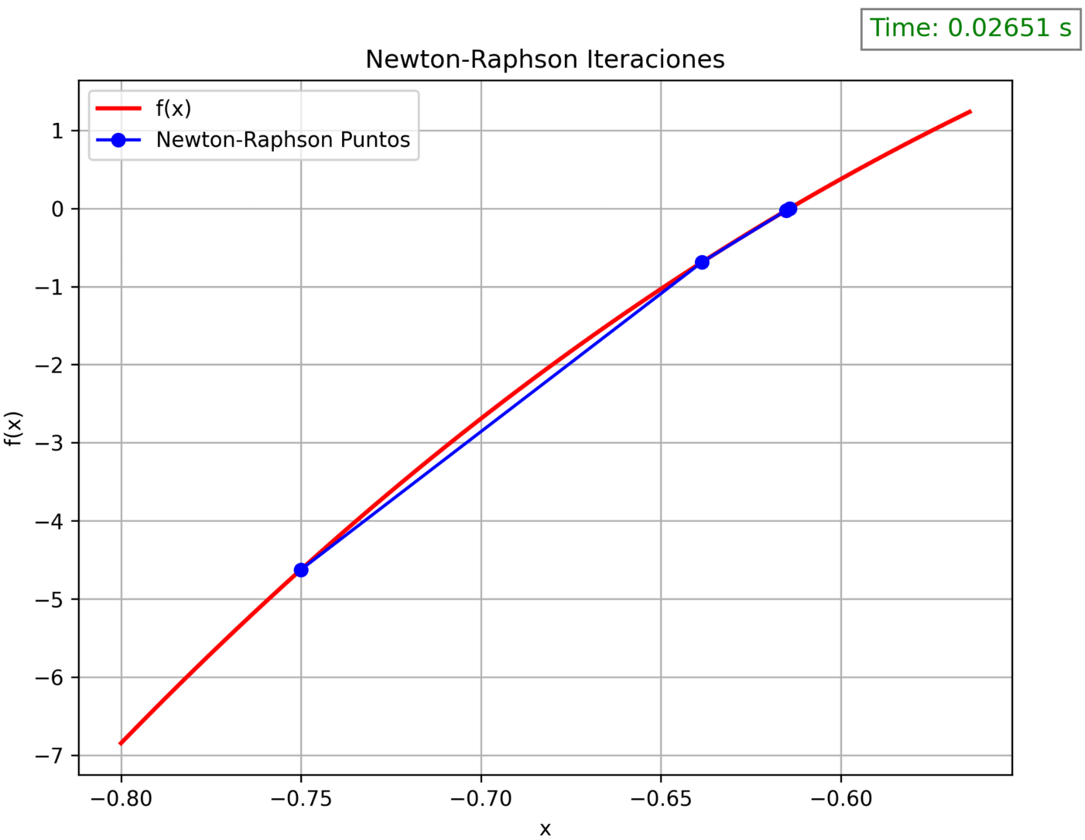
Función 1 $f(x) = 24x^3 - 4x^2 - 5x + 4$



ya que queremos obtener la raiz de la izq tomamos el intervalo (-0.75, -0,5)

```
Digite la función (con variable x): 24*x**3-4*x**2-5*x+4
ingrese el x_0:-0.75
ingrese el x_1:-0.5
Se cumple la condicion de fourier correctamente!!
Se inicializa la interacion con -0.75

Ingrese el error (default 0.001):
          x       f(x)         e
0 -0.750000 -4.625000
1 -0.638554 -0.687149  0.111446
2 -0.615235 -0.026873   0.02332
3 -0.614246 -0.000047  0.000989
tiempo de ejecucion: 0.026505708694458008 seg
```

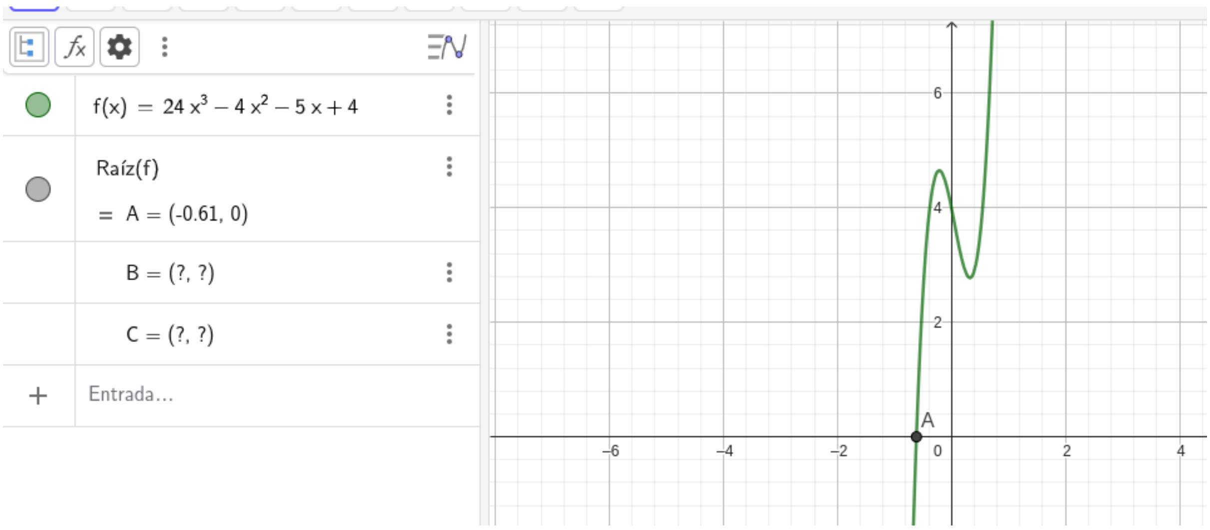
Dando como resultado 4 iteraciones



Dando como resultado -0.614246

```
-0.614246
```

Comparación con geogebra



Función 2 $f(x) = x^3 - 4x + 1$

```
Digite la función (con variable x): x**3-4*x+1
ingrese el x_0:1
ingrese el x_1:2
Se cumple la condicion de fourier correctamente!!
Se inicializa la interacion con 2.0

Ingrese el error (default 0.001):
          x          f(x)          e
0  2.000000  1.000000e+00
1  1.875000  9.179688e-02     0.125
2  1.860979  1.103129e-03  0.014021
3  1.860806  1.663940e-07  0.000173
tiempo de ejecucion: 0.018723487854003906 seg
```

función proporcionada sin raíz calculada de la función $f(x) = x^3 - 4x + 1$

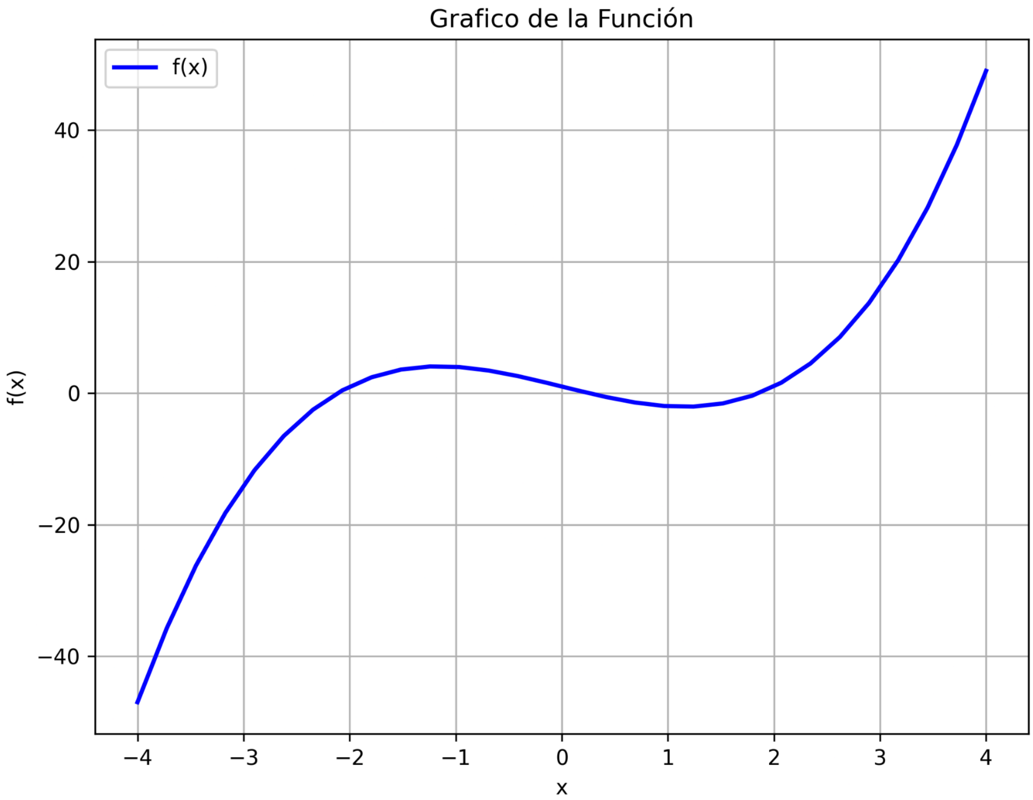
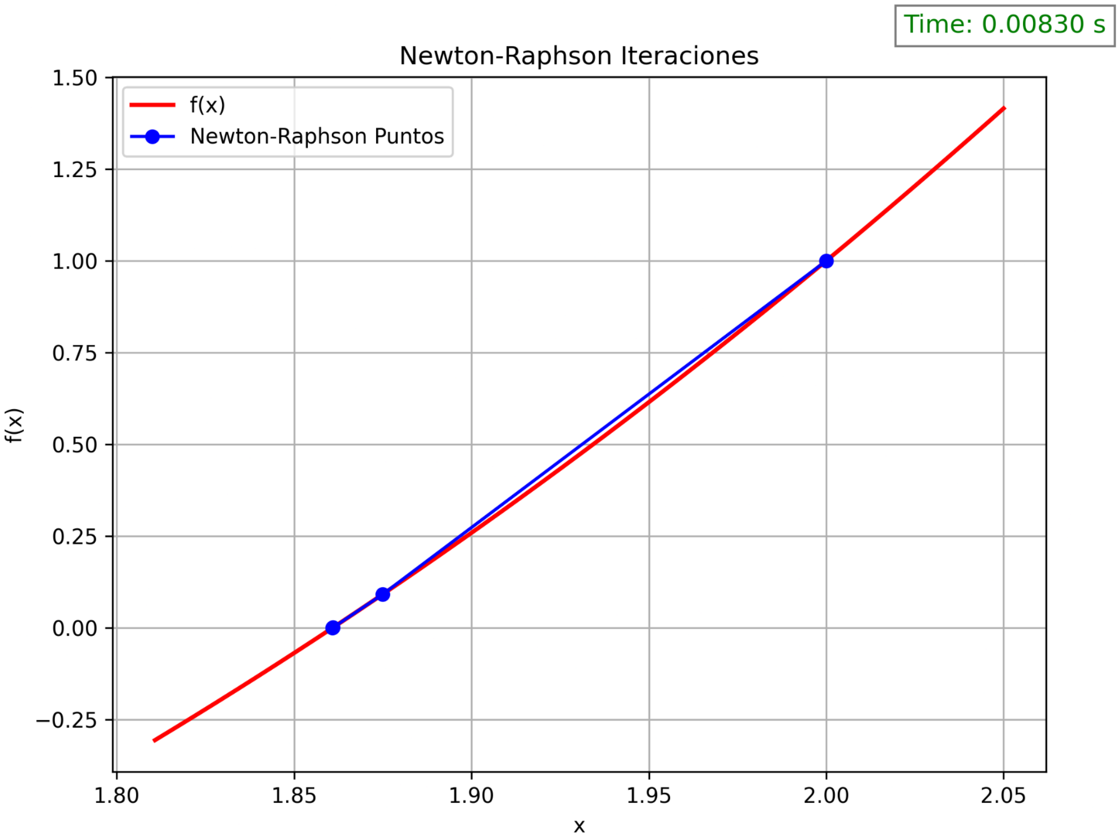


Grafico de la Función
f(x)
40
20
0
−20
−40
−4
−3
−2
−1
0
1
2
3
4
x
f(x)

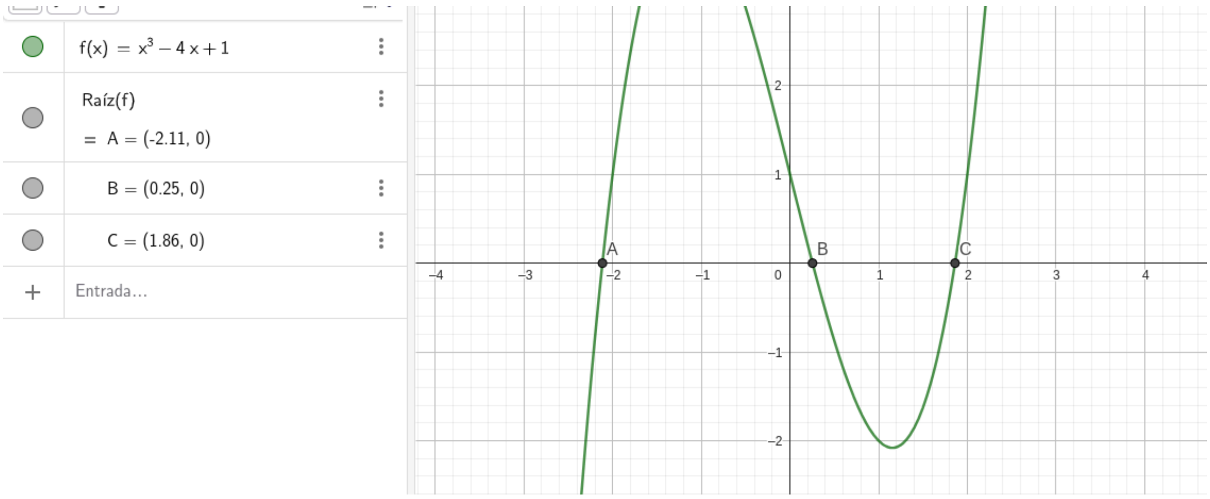


En este gráfico se ve como inicia la iteración en el punto (2,1) y va iterando 3 veces hasta llegar a la raíz

Tomamos el intervalo 1 y 2 para obtener la raíz que es

```
1.860806
```

Comparación con geogebra



La raíz C es la que buscamos que es 1.86

Método Intervalo Medio

Definición

El Método del Intervalo Medio, también conocido como Método de la Bisección, es un método numérico utilizado para encontrar una aproximación a la raíz de una función continua en un intervalo específico. Este método se fundamenta en el Teorema del Valor Intermedio, el cual afirma que, si una función f(x) es continua en un intervalo [a,b] y toma valores de signos opuestos en los extremos $(f(a) \cdot f(b) < 0)$, entonces existe al menos un punto c en el intervalo tal que $f(c) = 0$.

Para aplicar el método, se puede calcular la cantidad de iteraciones necesarias para lograr un error tolerable E mediante la siguiente fórmula:

$$n \geq \frac{log(b-a) - log\, E}{log\, 2}$$

Por ejemplo, si se desea una precisión de E<0.001 en el intervalo inicial [0.3,0.4], se requerirán aproximadamente 7 iteraciones.

Cada iteración del método consiste en calcular el punto medio del intervalo actual:

$$c = \frac{a+b}{2}$$

donde c es el nuevo punto medio del intervalo [a,b]. Se evalúa la función en este punto:

- Si $f(c) = 0$, entonces c es la raíz exacta.
- Si $f(c) \neq 0$, el intervalo se reduce seleccionando el subintervalo [a,c] o [c,b] en el cual ocurre el cambio de signo.

Este proceso continúa hasta que el error absoluto entre los extremos del intervalo sea menor al error tolerable E, logrando así una aproximación precisa de la raíz.

Ventajas

Convergencia garantizada: El método siempre converge hacia una raíz en funciones continuas donde se cumple el cambio de signo en el intervalo inicial.

Estabilidad numérica: Debido a su simplicidad, es menos propenso a errores numéricos y es fácil de implementar.

Aplicabilidad amplia: Puede aplicarse a cualquier función continua donde exista un cambio de signo en el intervalo inicial, independientemente de la forma de la función.

Control de precisión: La precisión de la aproximación se controla fácilmente ajustando el criterio de finalización, ya sea por el tamaño del intervalo o el valor de $f(c)$

Limitaciones

Convergencia lenta: El método converge linealmente, lo que significa que puede requerir muchas iteraciones para alcanzar una precisión alta, especialmente en intervalos grandes.

Requiere cambio de signo: Sólo se aplica a funciones continuas donde la raíz esté dentro de un intervalo con cambio de signo en los valores de los extremos.

No adecuado para raíces múltiples: Puede no detectar raíces dobles o múltiples en el mismo intervalo, ya que se basa en el cambio de signo.

Poca eficiencia para funciones complejas: Para funciones que varían mucho en magnitud o tienen pendientes pronunciadas, el método puede requerir más iteraciones para alcanzar una precisión adecuada

Código Fuente

En este proyecto, justo como fue visto anteriormente en el método Newton Rapson hemos organizado el código del método de Newton-Raphson en módulos separados, estructurando cada funcionalidad en funciones independientes para facilitar su mantenimiento y comprensión. Esta modularización permite enfocar cada archivo en una tarea específica, mejorando la claridad y escalabilidad del código. La estructura de archivos es la siguiente:

f(x): Define la función de la cual se busca una raíz, en este caso una función cúbica siendo $f(x) = 24x^3 - 4x^2 - 5x + 4$, pero puede cambiarse según la necesidad. Esta función se evalúa en cada iteración dentro de calcular_intervalo_medio para determinar en qué subintervalo puede encontrarse la raíz.

calcular_intervalo_medio(a, b, E): Esta función implementa el Método del Intervalo Medio para encontrar una raíz aproximada de una función $f(x)$ en el intervalo [a,b] con un error máximo especificado E. Primero, verifica que el intervalo sea válido (es decir, que f(a) y f(b) tengan signos opuestos). Calcula el número de iteraciones necesarias para alcanzar la precisión deseada y luego, en un bucle, ajusta el intervalo en cada iteración utilizando el punto medio, hasta que el tamaño del intervalo sea menor que el error tolerable. Devuelve una lista con los valores de cada iteración y los puntos medios calculados.

graficar_intervalo_medio(puntos_medios, a, b): Genera un gráfico de la función f(x) en el intervalo [a,b] y marca los puntos medios calculados en cada iteración. Utiliza matplotlib para mostrar la curva de f(x) y destaca cada punto medio con un marcador rojo, etiquetando con el número de iteración correspondiente. Esto proporciona una representación visual de cómo se ajusta el intervalo en cada paso hacia la raíz.

tabla_intervalo_medio(valores_iteraciones): Convierte los valores de cada iteración en un DataFrame de Pandas y lo guarda como un archivo CSV. La tabla resultante incluye columnas como el número de iteración, el valor de f(x), el intervalo [a,b] y el error. Esta estructura permite revisar y analizar los valores iterativos de manera organizada.

Iniciamos importando las librerías

```python
import math #para cálculos

import numpy as np #para listas

import matplotlib.pyplot as plt #para gráficos

import pandas as pd #para dataframe

import time #para cálculo de tiempo de ejecución
```

Definimos la función

```python
def f(x):

    return 24*x**3-4*x**2-5*x+4
```

Función del Intervalo Medio

```python
start_time = time.time()

def calcular_intervalo_medio(a, b, E):

#si no se cumple esta condición el programa no se ejecuta
```

```
    if f(a) * f(b) >= 0:
        print("El método del intervalo medio no puede aplicarse a este intervalo.")
        return None

# se calcula la cantidad de iteraciones necesarias
    n = math.ceil((math.log(b - a) - math.log(E)) / math.log(2))
    print(f"Cantidad de iteraciones estimadas: {n}\n")

#Se crean un contador de iteraciones y listas de puntos medios como los valores obtenidos en cada iteracion
    iteraciones = 0
    puntos_medios = []
    valores_iteraciones = []

# Bucle principal que se ejecuta mientras la longitud del intervalo sea mayor que el error E.
    while abs(b - a) > E:
        c = (a + b) / 2 # Cálculo del punto medio del intervalo actual
        puntos_medios.append(c)  # Almacena el punto medio en la lista 'puntos_medios'
        #print(f"Iteración {iteracion}: Intervalo [{a}, {b}], Punto medio = {c}")
        error = abs(b - a)
        valores_iteraciones.append([iteraciones + 1, c, f(c), (a, b), error])
```

```
        # Comprobación de si `f(c)` es exactamente cero, en cuyo caso `c` es
        la raíz y termina el bucle.
        if f(c) == 0:
            break
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c
        #print(f"Error cometido: {abs(b - a) / 2}\n")
        iteraciones += 1
    c = (a + b) / 2
    puntos_medios.append(c)
    error = abs(b - a)
    valores_iteraciones.append([iteraciones + 1, c, f(c), (a, b), error])
    iteraciones += 1

    #print(f"Raíz aproximada: {(a + b) / 2}")
    #print(f"Error final: {(b - a) / 2}")
    return valores_iteraciones, puntos_medios
end_time = time.time()
```

Gráfico

```python
def graficar_intervalo_medio(puntos_medios, a, b):
    plt.figure(figsize=(10, 6)) #Se especifica tamaño de la figura
```

```
    x_vals = np.linspace(a, b, 400)

    y_vals = f(x_vals)


    plt.plot(x_vals, y_vals, label="f(x)", color='blue')

    plt.axhline(0, color='black', linewidth=0.5)


    for i, c in enumerate(puntos_medios):

        plt.plot(c, f(c), 'ro')

                        plt.text(c,   f(c),   f'Iter   {i+1}',   fontsize=12,
verticalalignment='bottom')


    plt.xlabel('x')

    plt.ylabel('f(x)')

    plt.title('Método del Intervalo Medio')

    plt.legend()

    plt.grid(True)

    plt.show()
```

Tabla

```python
def tabla_intervalo_medio(valores_iteraciones):

        df  =  pd.DataFrame(valores_iteraciones,  columns=['Iteración',  'x',
'f(x)', 'Intervalo (a, b)', 'Error'])


    # Guardar el DataFrame en un archivo CSV

    df.to_csv('../data/tabla_intervalo_medio.csv', index=False)
```

```
    return df
```

Realizamos la ejecución:

Gráfico

```python
a = -0.75
b = -0.5
E = 0.001
valores_iteraciones, puntos_medios = calcular_intervalo_medio(a, b, E)


df = tabla_intervalo_medio(valores_iteraciones)
graficar_intervalo_medio(puntos_medios, a, b)
execution_time = end_time - start_time
print(f"El tiempo de ejecución fue: {execution_time:.6f} segundos")
```

Tabla

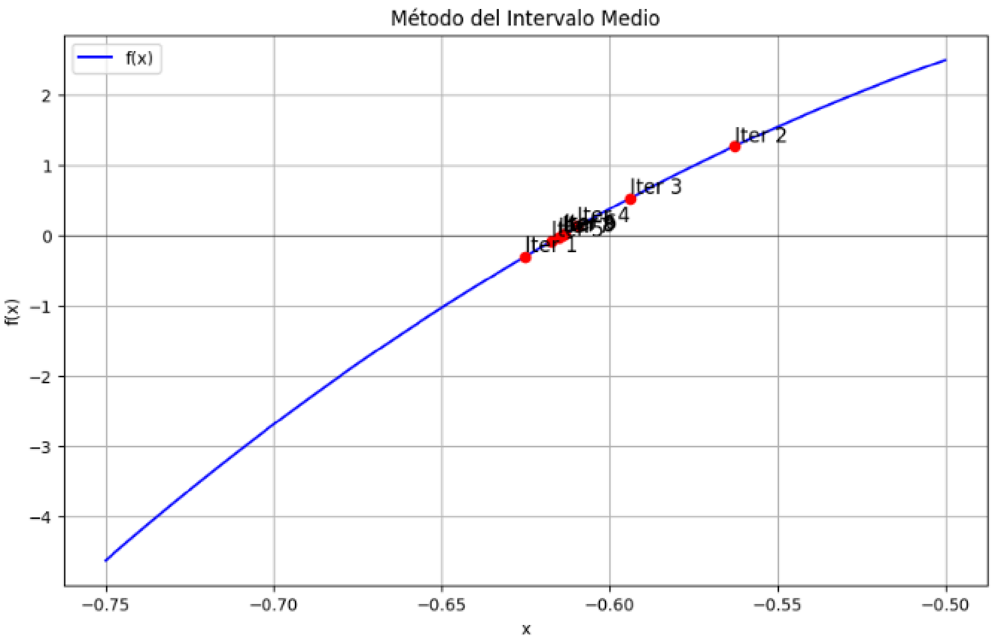
```python
#Se calcula el intervalo medio proporcionando error e intervalo
calcular_intervalo_medio(a, b, E)
#Se guarda la tabla
df_intervalo_medio = pd.read_csv('../data/tabla_intervalo_medio.csv')
#se muestra el dataframe
df_intervalo_medio
```

Resultados

Función 1 $f(x) = 24x^3 - 4x^2 - 5x + 4$

Gráfico

```
Cantidad de iteraciones estimadas: 8
```




```
El tiempo de ejecución fue: 0.000455 segundos
```

Tabla

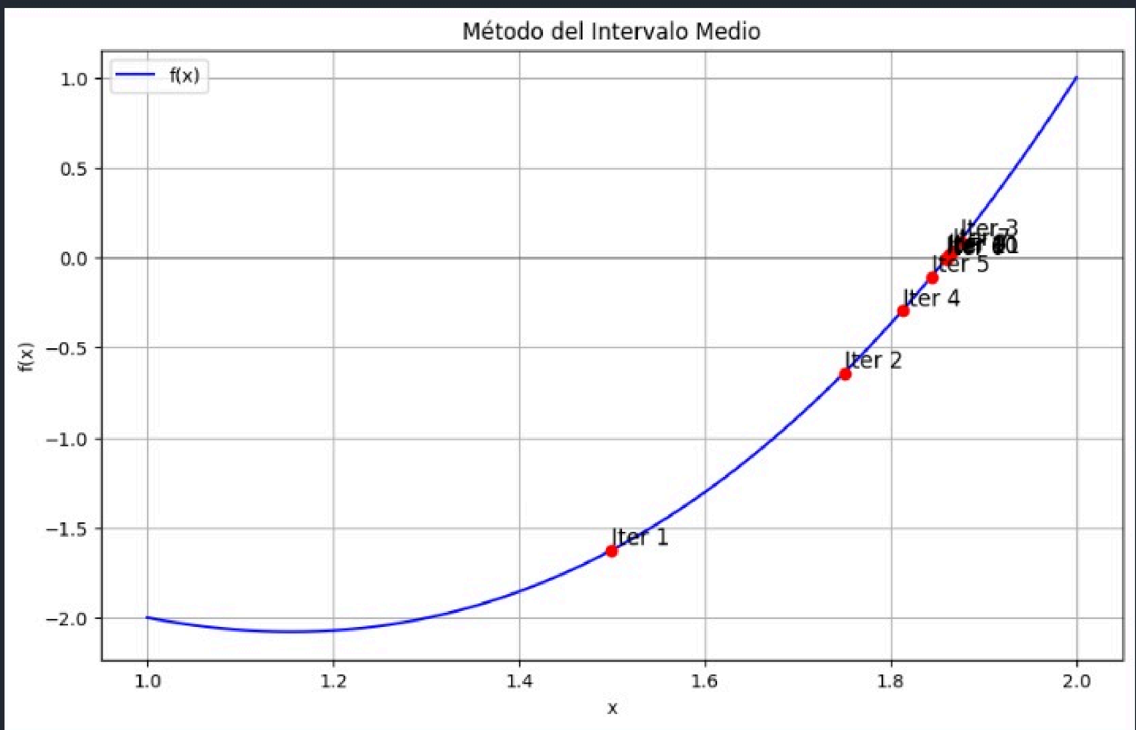
```
Cantidad de iteraciones estimadas: 8
```

	Iteración	x	f(x)	Intervalo (a, b)	Error
0	1	-0.625000	-0.296875	(-0.75, -0.5)	0.250000
1	2	-0.562500	1.275391	(-0.625, -0.5)	0.125000
2	3	-0.593750	0.534912	(-0.625, -0.5625)	0.062500
3	4	-0.609375	0.130707	(-0.625, -0.59375)	0.031250
4	5	-0.617188	-0.080128	(-0.625, -0.609375)	0.015625
5	6	-0.613281	0.026024	(-0.6171875, -0.609375)	0.007812
6	7	-0.615234	-0.026867	(-0.6171875, -0.61328125)	0.003906
7	8	-0.614258	-0.000376	(-0.615234375, -0.61328125)	0.001953
8	9	-0.613770	0.012836	(-0.6142578125, -0.61328125)	0.000977

Función 2 $f(x) = x^3 - 4x + 1$

Gráfico

```
Cantidad de iteraciones estimadas: 10
```




```
El tiempo de ejecución fue: 0.001527 segundos
```

Tabla

	Iteración	x	f(x)	Intervalo (a, b)	Error
0	1	-0.625000	-0.296875	(-0.75, -0.5)	0.250000
1	2	-0.562500	1.275391	(-0.625, -0.5)	0.125000
2	3	-0.593750	0.534912	(-0.625, -0.5625)	0.062500
3	4	-0.609375	0.130707	(-0.625, -0.59375)	0.031250
4	5	-0.617188	-0.080128	(-0.625, -0.609375)	0.015625
5	6	-0.613281	0.026024	(-0.6171875, -0.609375)	0.007812
6	7	-0.615234	-0.026867	(-0.6171875, -0.61328125)	0.003906
7	8	-0.614258	-0.000376	(-0.615234375, -0.61328125)	0.001953
8	9	-0.613770	0.012836	(-0.6142578125, -0.61328125)	0.000977

Conclusiones

Función 1

$f(x) = 24x^3 - 4x^2 - 5x + 4$			
	Iteraciones	Tiempo de ejecución	Error
Newton Raphson	4	0.02651s	0.000989
Intervalo Medio	9	0.000455s	0.000977

Función 2

$f(x) = x^3 - 4x + 1$			
	Iteraciones	Tiempo de ejecución	Error
Newton Raphson	4	0.018723s	0.000173
Intervalo Medio	10	0.001527s	0.000977

A partir de la comparación entre los métodos de Newton-Raphson y el de Intervalo Medio en términos de tiempo de ejecución para encontrar raíces de funciones, podemos observar diferencias significativas en su rendimiento y eficiencia. En ambas funciones analizadas, el método de Intervalo Medio muestra tiempos de ejecución considerablemente menores en comparación con el método de Newton-Raphson, a pesar de requerir más iteraciones para converger a un resultado similar. Para la primera función, el método de Newton-Raphson toma 0.02651 segundos, mientras que el método de Intervalo Medio toma apenas 0.000455 segundos, lo cual representa una diferencia notable. De manera similar, en la segunda función, Newton-Raphson tarda 0.018723 segundos en alcanzar la solución, mientras que el Intervalo Medio requiere solo 0.001527 segundos. Esta diferencia en los tiempos de ejecución se debe en gran parte a la necesidad del método de Newton-Raphson de calcular derivadas en cada iteración, lo cual incrementa la complejidad computacional. Aunque el método de Newton-Raphson tiene la ventaja de converger con menos iteraciones debido a su rapidez en condiciones favorables, el costo computacional de calcular tanto la función como su derivada en cada paso puede hacerlo menos eficiente en cuanto a tiempo, especialmente cuando se trabaja con funciones más complejas. Por otro lado, el método de Intervalo Medio, aunque puede requerir más iteraciones, realiza cálculos más simples en cada paso, lo cual reduce su tiempo de ejecución total y lo convierte en una opción más rápida y eficiente en ciertos casos.

Bibliografía