Universidad:
Universidad Nacional Del Nordeste

Facultad:
Facultad de Ciencias Exactas y Naturales y Agrimensura

Asignatura:
Métodos Computacionales

Tema:
Raíces De Ecuaciones

Integrantes:
Gauna, Octavio Victor, LU:56997
Soto, Nahuel Federico, LU: 40215

Día Y Horario De Clase:
Jueves de 15 a 17hs

Equipo Docente:
Maria Isabel Sanchez

Año Lectivo:
2024

Introducción:

Este informe presenta el análisis de dos métodos numéricos para el cálculo de raíces de funciones no lineales: el **método de Newton-Raphson** y el **método de Aceleración de Aitken** aplicado al método de punto fijo. Ambos métodos son efectivos para mejorar la precisión y velocidad de convergencia en procesos iterativos, facilitando la solución de ecuaciones complejas en el ámbito numérico. A lo largo de este trabajo, se implementan y comparan ambos métodos mediante código en Python, mostrando sus resultados y ventajas comparativas.

Método de Newton-Raphson:

Definición:

- El método de Newton-Raphson es un proceso iterativo que se basa en la aproximación de la raíz de una función utilizando la tangente en un punto inicial. Si f(x) es nuestra función y f′(x) es su derivada, la fórmula de iteración para Newton-Raphson es:

$$x\,n+1 = x\,n - f'(x\,n) f(x\,n)$$

- La aproximación se actualiza sucesivamente hasta que la diferencia entre x_{n+1} y x_n sea menor que una tolerancia prefijada, indicando que se ha alcanzado la precisión deseada.

Ventajas:

- Alta velocidad de convergencia si el punto inicial está cerca de la raíz.
- Precisión y eficiencia en menos iteraciones que otros métodos.

Limitaciones:

- Requiere el cálculo de derivadas, lo cual puede ser complejo.
- No garantiza convergencia si el punto inicial no es adecuado, o si la derivada es muy pequeña (división por un valor cercano a cero). Debe cumplir con ciertas condiciones.

Método de Aceleración de Aitken:

Definición:

- La aceleración de Aitken es una técnica para mejorar la convergencia de secuencias, aplicable a métodos como el de punto fijo. En este caso, aceleramos la secuencia generada en cada iteración para acercarnos más rápidamente a la raíz.
- Dado un valor inicial x_0, el método de punto fijo calcula sucesivamente $x_{n+1}=g(x_n)$. La técnica de Aitken se aplica cada dos iteraciones de punto fijo, combinando valores previos para lograr una aproximación más rápida.

Ventajas:

- Mejora la convergencia de métodos iterativos, permitiendo alcanzar la solución en menos iteraciones.
- Reduce errores en cada paso de aceleración.

Limitaciones:

- Puede fallar si la función no cumple con las condiciones de convergencia.
- En algunos casos, el error puede no disminuir si el método de punto fijo diverge.

Código fuente Newton-Raphson:

1. Definimos f y sus respectivas derivadas usando sympy, que permite una diferenciación simbólica, también el error tolerable.

```
# Definir la variable simbólica
x = symbols('x')

# Ingreso de la función original
f_str = input("Ingrese la función f(x) en términos de x (por ejemplo: 6*x**2 + 3*x - 2): ")
f_sym = eval(f_str)

# Ingresar el error tolerable
error_tolerable = float(input("Ingrese el error tolerable (por ejemplo: 0.001): "))

# Calcular la derivada primera y segunda
f_prime_sym = diff(f_sym, x)
f_double_prime_sym = diff(f_prime_sym, x)

# Convertir las funciones a evaluables
f = lambdify(x, f_sym, "numpy")
f_prime = lambdify(x, f_prime_sym, "numpy")
f_double_prime = lambdify(x, f_double_prime_sym, "numpy")
```

2. Aplicamos método de tanteo, donde se revisan valores de x para detectar cambios de signo en la función, indicando la presencia de una raíz.

```
# Método de tanteo
def tanteo():
```

```
    print("Tanteo:")
    prev_value = f(-10)  # Inicializar valor función
    for i in range(-9, 11):  # Rango tanteo
      current_value = f(i)
      print(f"f({i}) = {current_value}")
      # Detectar cambio de signo
      if prev_value * current_value < 0:
            return i-1, i  # Retornar hay cambio signo
      prev_value = current_value  # Actualizamos el valor anterior
    # En caso de no encontrar un cambio de signo
    return None, None
```

3. Aplicar condiciones de Fourier, estas condiciones ayudan a elegir un punto inicial en el intervalo para que Newton-Raphson tenga mayor probabilidad de converger.

```
def condiciones_fourier(x0, x1):
      if f(x0) * f(x1) < 0:
            print(f"1. f(x0) * f(x1) < 0: Hay una raíz entre x0 = {x0} y 
x1 = {x1}")
      else:
            print(f"No se cumple la primera condición de Fourier: No hay 
una raíz entre {x0} y {x1}")

    # Evaluar segunda condición para ambos puntos
      print("\n2. Evaluando f(x) * f''(x) para x0 y x1:")
      if f(x0) * f_double_prime(x0) > 0:
            print(f"f(x0) * f''(x0) > 0: El método Newton-Raphson debe 
iniciar con x0 = {x0}")

        return x0

      elif f(x1) * f_double_prime(x1) > 0:
            print(f"f(x1) * f''(x1) > 0: El método Newton-Raphson debe 
iniciar con x1 = {x1}")
            return x1
      else:
            print("No se cumple la segunda condición de Fourier para 
ninguno de los puntos.")
            return None
```

4. Implementación del Método de Newton-Raphson: Se aplica la iteración de Newton-Raphson hasta que el error sea menor a la tolerancia.

```
def newton_raphson(x0, tol=1e-3, max_iter=100):
    iteraciones = 0
    error = float('inf')
    print("\nIteración | Raíz estimada        | Error")
    while error > tol and iteraciones < max_iter:
        x1 = x0 - f(x0) / f_prime(x0)
        error = abs(x1 - x0)
        print(f"     {iteraciones + 1}          |     {x1:.15f}          |
{error:.15f}")
        x0 = x1
        iteraciones += 1
    return x0, iteraciones, error
```

5. Se grafica la función junto con los intervalos x0x_0x0, x1x_1x1, y la raíz estimada para visualizar la convergencia.

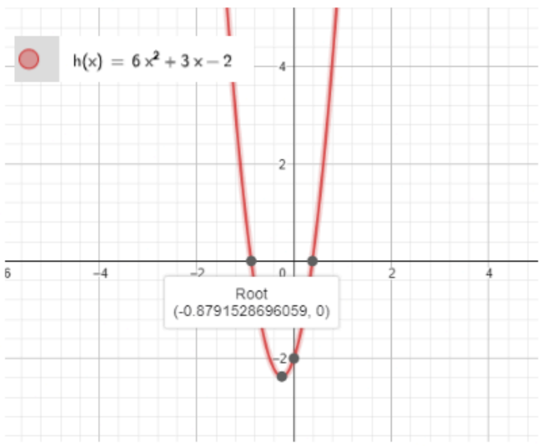
```
def graficar_funcion(x0, x1, raiz):
    x_vals = np.linspace(-10, 10, 400)
    y_vals = f(x_vals)
    plt.plot(x_vals, y_vals, label='f(x)')

    # Marcar la raíz
    plt.scatter(raiz,f(raiz),color='red',zorder=5,label=f'Raíz
aproximada:{raiz:.5f}')
    # Marcar los intervalos x0 y x1
    plt.axvline(x=x0,color='green',linestyle='--',label=f'x0= {x0}')
    plt.axvline(x=x1,color='blue',linestyle='--',label=f'x1 = {x1}')
    # Ejes y título
    plt.axhline(0, color='black',linewidth=0.5)
    plt.axvline(0, color='black',linewidth=0.5)
    plt.title('Gráfico de la función con Raíz y Intervalos')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.grid(True)
    plt.legend()
    plt.show()
```

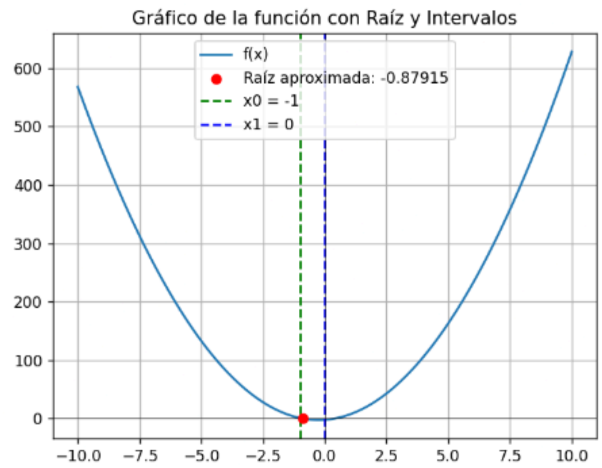
Ejecución:

Probamos con la siguiente función:
f(x): 6*x**2 + 3*x**1 - 2 **error**: 0.0001

```
Tanteo:
f(-9) = 457
f(-8) = 358
f(-7) = 271
f(-6) = 196
f(-5) = 133
f(-4) = 82
f(-3) = 43
f(-2) = 16
f(-1) = 1
f(0) = -2
Existe una raíz entre x = -1 y x = 0 porque f(-1) y f(0) tienen signos opuestos.
1. f(x0) * f(x1) < 0: Hay una raíz entre x0 = -1 y x1 = 0

2. Evaluando f(x) * f''(x) para x0 y x1:
f(x0) * f''(x0) > 0: El método Newton-Raphson debe iniciar con x0 = -1

Aplicando método de Newton-Raphson:

Iteración | Raíz estimada       | Error
   1      |   -0.888888888888889     |  0.111111111111111
   2      |   -0.879227053140097     |  0.009661835748792
   3      |   -0.879152873978877     |  0.000074179161219

Raíz aproximada: -0.879152873978877 en 3 iteraciones
Error final: 0.000074179161219
```

Veamos en geogebra y la ejecución en python:





Código de Aceleración de Aitken:

Permite al usuario ingresar la función de punto fijo y la función original para hallar raíces con fsolve.

```
def punto_fijo():
      # Ingreso de la función original
      f_str = input('Ingrese la función original en términos de x (por
ejemplo: x**3 - 3*x + 2): ')
      f = lambda x: eval(f_str)
      # Graficar la función
      x_vals = np.linspace(-20, 20, 100)
      y_vals = [f(x) for x in x_vals]   # Evaluar la función para cada
valor de x

      plt.figure(figsize=(10, 6))
      plt.plot(x_vals, y_vals, 'b-', linewidth=2)
      plt.title('Gráfica de la función original f(x)')
      plt.xlabel('x')
      plt.ylabel('f(x)')
      plt.axhline(0, color='red', linestyle='--')
      plt.grid(True)

      # Encontrar raíces
      raices = []
      for i in range(-20, 21):
            try:
```

```
            root = fsolve(f, i)[0]
            if abs(f(root)) < 1e-5 and all(abs(root - r) > 1e-5 for r in
raices):
            raices.append(root)
            plt.plot(root, f(root), 'ro', markersize=8, label=f'Raíz en
x = {root:.5f}')
          except:
               pass
     plt.legend()
     plt.show()
     # Ingreso de la función de punto fijo
     g_str = input('Ingrese la función de punto fijo g(x) en términos de
x (por ejemplo: np.sqrt(3 + 2/x)): ')
     g = lambda x: eval(g_str)
     # Ingreso del valor inicial
     while True:
       try:
          x0 = float(input('Ingrese el valor inicial x0: '))
          gx0 = g(x0)
          if not np.isfinite(gx0):
            raise ValueError("Valor indefinido")
          break
       except Exception as e:
          print('Error:', e)

    # Ingreso del error relativo deseado
     error_prefijado = float(input('Ingrese el error relativo deseado
(por ejemplo: 0.001): '))

    # Derivada de g para verificar convergencia
     h = 1e-5
     derivada_g_value = (g(x0 + h) - g(x0)) / h

     if abs(derivada_g_value) >= 1:
        print(f'WARNING: La función g(x) no garantiza convergencia porque
|g\'(x0)| = {derivada_g_value:.6f} >= 1.')
        return
     else:
          print(f'La función g(x) es convergente: |g\'(x0)| =
{derivada_g_value:.6f} < 1')
```

```
  # Iteraciones de Aitken
    iteraciones = []
    error_relativo = np.inf
    x_n = x0
    cant_iter = 0

    while error_relativo >= error_prefijado:
      # Aplicar el método de punto fijo
      x_n1 = g(x_n)   # Primera iteración de punto fijo
      x_n2 = g(x_n1) # Segunda iteración de punto fijo

      if cant_iter % 3 == 2:
          # Aplicar Aitken cada 2 iteraciones de punto fijo
          p_accel =(x_n*x_n2-x_n1*x_n1) / (x_n+x_n2-2*x_n1)
          error_rel = abs(p_accel - x_n2)
          x_n = p_accel  # Actualizar x_n con el valor acelerado
          iteraciones.append((cant_iter+1,x_n,error_rel , 'Aitken'))
      else:
          error_relativo = abs(x_n1 - x_n)   # Calcular error para la primera iteración
          x_n = x_n1  # Actualizar x_n
          iteraciones.append((cant_iter + 1, x_n, error_relativo, 'Punto Fijo'))

      cant_iter += 1

    # Mostrar resultados
    print('\nResultados de la iteración de punto fijo con Aitken:')
    print(f'{"Iteración":<12}{"x":<12}{"Error":<12} {"Método":<12}')
    print(f'{"-----":<12}{"---":<12}{"-------":<12} {"------":<12}')
    for iteracion in iteraciones:
                        print(f'{iteracion[0]:<12}{iteracion[1]:<12.6f}{iteracion[2]:<12.6f} {iteracion[3]:<12}')
```

Ejecución:

Probamos con la siguiente función:

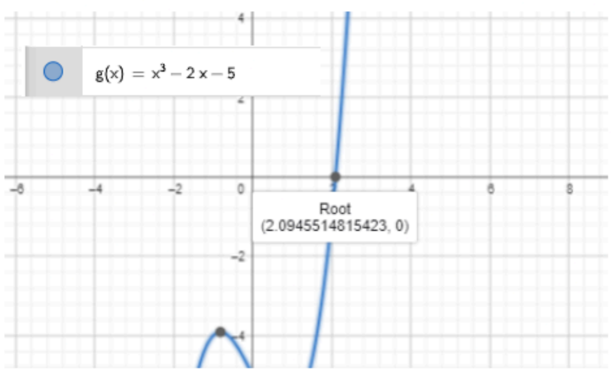
f(x): x**3 - 2*x - 5   **g(x)**: (2*x + 5)**(1/3)   **valor inicial**: 2   **error**: 0.0001

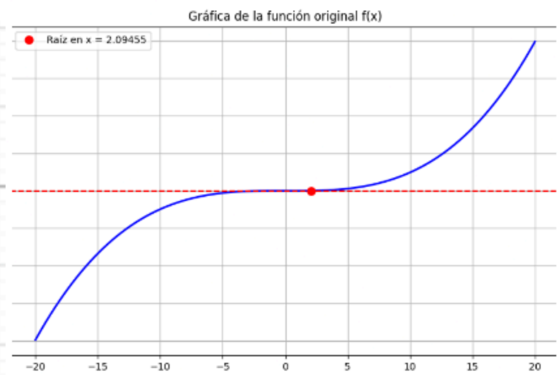
La función g(x) es convergente: |g'(x0)| = 0.154080 < 1

Resultados de la iteración de punto fijo con Aitken:

Iteración	x	Error	Método
1	2.080084	0.080084	Punto Fijo
2	2.092351	0.012267	Punto Fijo
3	2.094551	0.000051	Aitken

Veamos en geogebra y la ejecución en python:





Conclusión:

En este trabajo, se analizaron y compararon los métodos de Newton-Raphson y la aceleración de Aitken aplicados al cálculo de raíces de funciones no lineales. Ambos métodos mostraron ser efectivos, aunque sus características y rendimiento variaron en función de la estructura de la función y las condiciones de inicio.

El método de Newton-Raphson demostró una alta rapidez de convergencia en la resolución de la función cuadrática analizada, alcanzando la raíz en pocas iteraciones y con un error final pequeño. Sin embargo, esta eficiencia depende en gran medida de la elección de un buen punto inicial y de la facilidad para calcular la derivada de la función. En funciones donde la derivada se aproxima a cero o cambia de signo abruptamente, el método puede divergir o presentar problemas de precisión.

Por otro lado, el método de Aceleración de Aitken aplicado a la iteración de punto fijo mostró ser una alternativa útil para mejorar la convergencia cuando el método de punto fijo por sí solo sería lento o no alcanzaría la precisión deseada en un número aceptable de iteraciones. Aitken permite reducir significativamente el número de iteraciones al aplicar una optimización sobre el proceso de punto fijo, lo que lo convierte en una herramienta valiosa para sistemas iterativos. Sin embargo, este método depende de que la función de punto fijo cumpla con las condiciones necesarias para la convergencia.

En conclusión, ambos métodos son herramientas potentes dentro de la resolución numérica de ecuaciones, cada uno con sus ventajas y limitaciones. La elección entre uno u otro depende de la naturaleza de la función y de las condiciones específicas del problema a resolver. Para funciones con derivadas bien definidas y puntos iniciales cercanos a la raíz, Newton-Raphson es preferible. En casos donde el cálculo de derivadas es complejo o se requiere optimizar un proceso iterativo, la aceleración de Aitken en punto fijo puede ofrecer ventajas significativas.

Bibliografía:

Documentación de SymPy y NumPy para Python:

Python: https://docs.python.org/3/

SymPy: https://docs.sympy.org

NumPy: https://numpy.org/doc/stable/

Estas librerías fueron utilizadas para la implementación en Python del método de Newton-Raphson, y sus documentaciones son recursos clave para entender las funciones y métodos aplicados.