Trabalho Modelagem Computacional

Grupo: Gabriel Couto, Sarah Ferrari, André Casemiro, Guilherme Maranhão e Gabriel Lacerda
Data: 15 de nov, 2024

Newton Raphson - *Sarah Ferrari e Gabriel Couto*

O código em Python implementa o método de Newton-Raphson, permitindo encontrar raízes de funções matemáticas. Ele recebe como parâmetros a função, sua derivada e um valor inicial x0, retornando a raiz aproximada com erro máximo de 0,00001 e limite de 100 iterações. O cálculo itera até atingir a precisão desejada ou o limite de iterações, garantindo flexibilidade e eficiência no processo.

<u>Código:</u>

```python
import numpy as np

def newton_raphson(f, g, x0, E=0.00001, N=100):
    """
    Método de Newton-Raphson para encontrar raízes de funções.

    Parâmetros:
    f  - Função da qual queremos encontrar a raiz.
    g  - Derivada da função f.
    x0 - Estimativa inicial da raiz.
    E  - Tolerância para critério de parada (default: 0.00001).
    N  - Número máximo de iterações (default: 100).

    Retorna:
    Aproximação da raiz da função f ou None se falhar.
    """
    for n in range(1, N + 1):
        try:
```

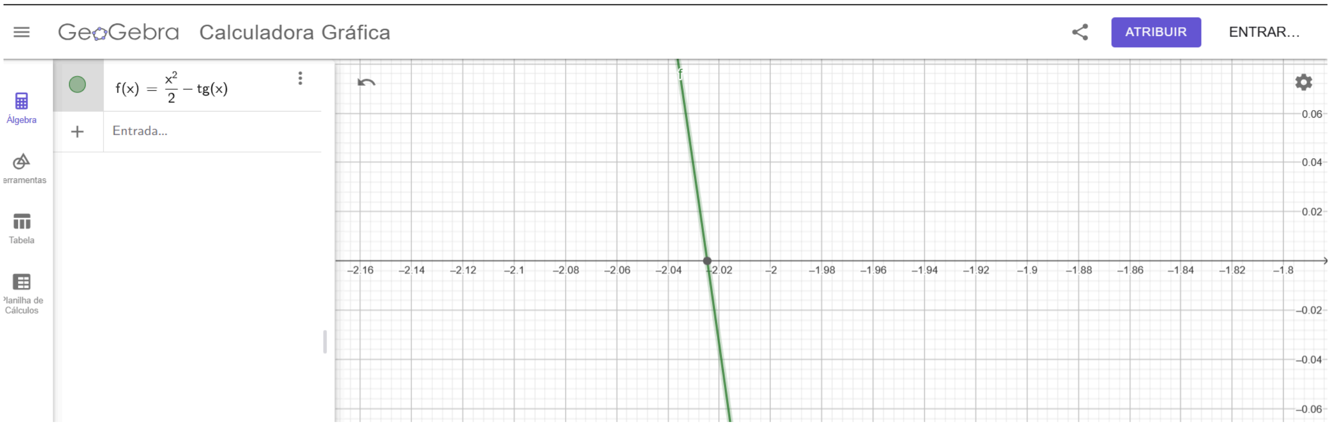
```
            fx = f(x0)
            gx = g(x0)
            if gx == 0:
                print(f"Iteração {n}: Derivada zero. Newton-Raphson
falhou.")
                return None

            x_novo = x0 - fx / gx
        except ZeroDivisionError:
            print(f"Iteração {n}: Divisão por zero. Newton-Raphson
falhou.")
            return None
        except Exception as e:
            print(f"Erro na iteração {n}: {e}")
            return None

        print(f"Iteração {n}: x = {x_novo:.15f}, f(x) = {fx:.15f}")

        if abs(x_novo - x0) < E:  # Critério de parada
            return x_novo

        x0 = x_novo

    print("Número máximo de iterações atingido.")
    return None

# Função 1: f(x) = (x^2)/2 - tan(x)
def f1(x):
    return (x**2) / 2 - np.tan(x)

# Derivada da função 1: g(x) = x - sec(x)^2
def g1(x):
```

```python
    return x - (1 / np.cos(x))**2

# Função 2: f(x) = x^5 - 9x^3 + x + 3
def f2(x):
    return x**5 - 9 * x**3 + x + 3

# Derivada da função 2: g(x) = 5x^4 - 27x^2 + 1
def g2(x):
    return 5 * x**4 - 27 * x**2 + 1

# Testando o método de Newton-Raphson com ambas as funções
print("Raiz da função f(x) = (x^2)/2 - tan(x):")
x0_f1 = -2 # Estimativa inicial para f1
raiz1 = newton_raphson(f1, g1, x0_f1)
if raiz1 is not None:
    print(f'Raiz aproximada: x = {raiz1:.15f}\n')

print("Raiz da função f(x) = x^5 - 9x^3 + x + 3:")
x0_f2 = 0.5  # Estimativa inicial para f2
raiz2 = newton_raphson(f2, g2, x0_f2)
if raiz2 is not None:
    print(f'Raiz aproximada: x = {raiz2:.15f}\n')
```

Obtenção dos Pontos:

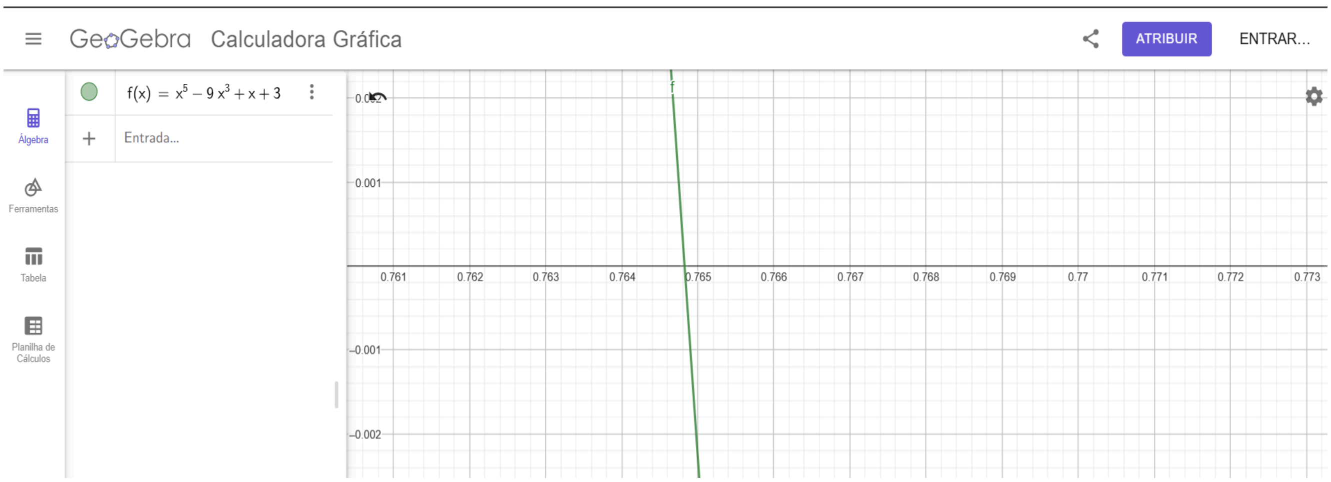
Para determinar os valores iniciais x0, utilizamos o software GeoGebra, analisando os gráficos das funções e identificando pontos próximos às raízes.

Gráfico de f(x) = (x^2)/2 - tan(x):



Escolhemos o valor de x0 = −2, pois está próximo de uma raiz visível no gráfico e dentro do intervalo onde a função é contínua, evitando as descontinuidades de tan(x).

Gráfico de f(x)= x^5 - 9x^3 + x + 3:



Escolhemos o valor de x0=0.7, pois é onde o ponto está próximo de uma raiz real, como observada acima no comportamento da função.

Resultados

Os valores iniciais escolhidos, combinados com o método de Newton-Raphson, permitem aproximar as raízes das funções, com as justificativas baseadas na análise gráfica.

Bissecção - *André Casemiro, Guilherme Maranhão e Gabriel Lacerda*

O código em Python abaixo implementa o **Método da Bissecção**, uma técnica numérica para encontrar raízes de funções contínuas em um intervalo [a, b]. O método divide o intervalo ao meio iterativamente, selecionando a sub-região onde ocorre uma mudança de sinal da função, até atingir a precisão desejada ou o número máximo de 100 iterações. Ele garante a convergência para uma raiz se o intervalo inicial contiver uma mudança de sinal.

Código:

```python
import math

# Definindo as funções matemáticas
def f1(x):
    return x**2 / 2 - math.tan(x)

def f2(x):
    return x**5 - 9 * x**3 + x + 3

# Implementação do método da bisseção
def bisection_method(func, a, b, tolerance, max_iterations):
    print(f"Avaliando os extremos do intervalo: f({a}) = 
{func(a):.5f}, f({b}) = {func(b):.5f}")

    # Verifica se o método é aplicável no intervalo dado
    if func(a) * func(b) >= 0:
        print(f"Não é possível aplicar o método no intervalo ({a}, 
{b}) devido à falta de sinais opostos.")
        return None

    iteration = 0
    while (b - a) / 2 > tolerance and iteration < max_iterations:
        c = (a + b) / 2  # Calcula o ponto médio
```

```
        if func(c) == 0:  # Encontrou a raiz exata
            print(f"Raiz exata encontrada em x = {c:.5f}")
            return c, iteration
        elif func(a) * func(c) < 0:  # A raiz está na metade esquerda
            b = c
        else:  # A raiz está na metade direita
            a = c
        iteration += 1

    return c, iteration  # Retorna a raiz aproximada e o número de
iterações

# Função para ajustar automaticamente o intervalo, caso inválido
def find_valid_interval(func, a, b, step=0.1, max_steps=50):
    for _ in range(max_steps):
        if func(a) * func(b) < 0:  # Verifica se o intervalo é válido
            return a, b
        a -= step
        b += step
    return None, None  # Retorna None se nenhum intervalo válido for
encontrado

# Configurações do problema
tolerance = 10**-5  # Precisão desejada
max_iterations = 100  # Limite máximo de iterações

# Intervalos iniciais para as funções
intervals = [
    (f1, -2.2, 1),
    (f2, -3, 0.7)
]
```

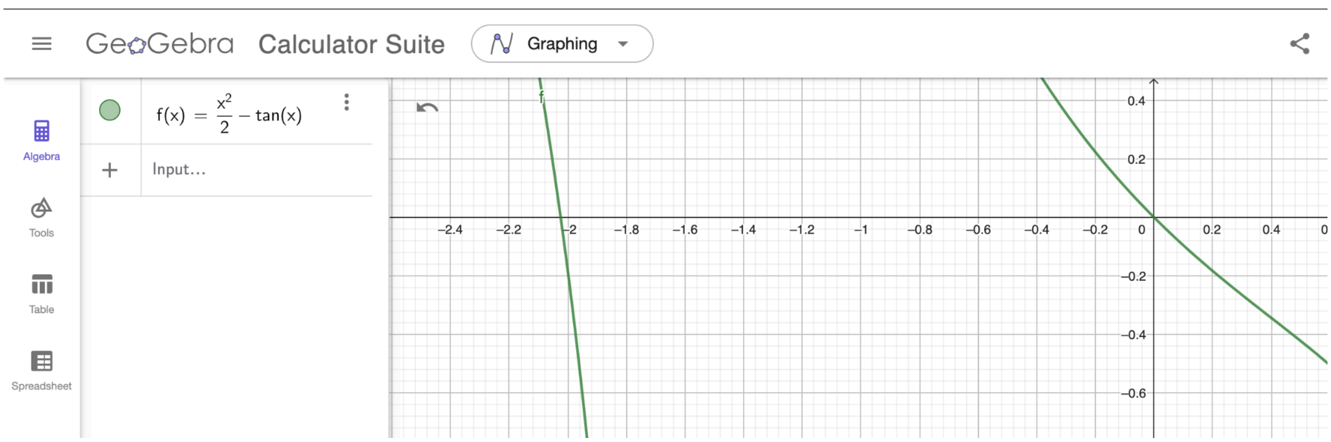
```python
# Processando cada função
results = []
for i, (func, a, b) in enumerate(intervals):
   print(f"\nAnalisando a função f{i+1} no intervalo inicial ({a}, {b}):")

   # Verifica se o intervalo inicial é válido
   if func(a) * func(b) >= 0:
       print(f"O intervalo fornecido ({a}, {b}) não é válido. Buscando um novo intervalo...")
       a, b = find_valid_interval(func, a, b)
       if a is None or b is None:
           print("Não foi possível encontrar um intervalo válido após múltiplas tentativas.")
           results.append((None, None))
           continue
       print(f"Novo intervalo válido encontrado: ({a}, {b})")

   # Calcula a raiz usando o método da bisseção
   root, iterations = bisection_method(func, a, b, tolerance, max_iterations)
   results.append((root, iterations))

# Apresentando os resultados
print("\nResultados Finais:")
for i, (root, iterations) in enumerate(results):
   print(f"Função f{i+1}:")
   if root is not None:
       print(f"  - Raiz aproximada: x = {root:.5f}")
       print(f"  - Iterações necessárias: {iterations}")
   else:
```

```
        print("  - Nenhuma raiz foi encontrada dentro dos limites fornecidos.")
    print("-" * 30)
```

Obtenção dos Intervalos:

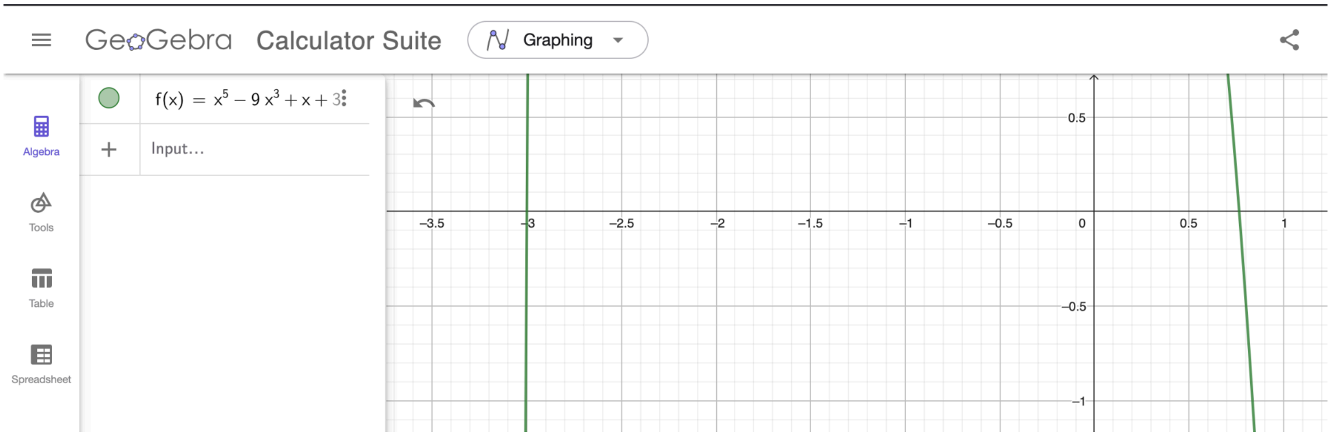
Para aplicar o Método da Bissecção, utilizamos o software GeoGebra para analisar os gráficos das funções e identificar intervalos [a, b] em que ocorre uma mudança de sinal, garantindo que f(a) . f(b) < 0.

Gráfico de f(x) = (x^2)/2 - tan(x):



Selecionamos o intervalo [−2.2, 1], pois os valores da função nesse intervalo apresentam uma mudança de sinal, indicando a presença de uma raiz. Este intervalo foi escolhido com base na análise gráfica e no comportamento contínuo da função longe das descontinuidades de tan(x).

Gráfico de f(x)= x^5 - 9x^3 + x + 3:



Escolhemos o intervalo [−3, 0.7], onde o gráfico da função mostra uma mudança de sinal clara, confirmando que há pelo menos uma raiz neste intervalo. A escolha também foi feita com base no comportamento observado no gráfico.

Resultados

Os intervalos escolhidos são suportados por imagens dos gráficos que evidenciam as mudanças de sinal, assegurando a aplicabilidade do Método da Bissecção.
