Davi Ferreira de Souza

Prof. Douglas Rodrigues

Métodos Numéricos Computacionais

13 de abril de 2025

Relatório Trabalho Prático 1 - Métodos Numéricos Computacionais

Implementação

A implementação do programa foi realizada na linguagem de programação Python, devido à sua ampla utilização na área científica e à disponibilidade de bibliotecas especializadas. Para o desenvolvimento do trabalho, foram utilizadas as bibliotecas matplotlib, os, time e math, as quais forneceram suporte para a construção da interface gráfica em terminal, geração automática de gráficos comparativos, controle de tempo de execução e operações matemáticas auxiliares.

O programa desenvolvido tem como objetivo resolver equações não lineares associadas a problemas reais de diferentes áreas do conhecimento, utilizando os métodos numéricos de **Bisseção**, **Falsa Posição** e **Newton-Raphson**. Além de calcular as raízes aproximadas, o sistema registra o número de iterações realizadas, o tempo de processamento e os erros associados a cada método, permitindo uma análise comparativa entre eles. Para isso, o código foi estruturado com base em um menu interativo, que possibilita ao usuário escolher qual problema deseja analisar e visualizar os resultados de forma clara e organizada.

Definição das variáveis globais:

```
#Inicialização de variáveis globais para contagem de iterações e para o problema 4 (função de multiplas variáveis)
iteracoes_bicessao = 0
iteracoes_falsa_posicao = 0
iteracoes_newton = 0

epsilon = 0
sigma = 0
```

Nesta etapa do projeto, foram definidas variáveis globais com o objetivo de facilitar o controle de certas informações ao longo da execução do programa. As variáveis iteracoes_bicessao, iteracoes_falsa_posicao e iteracoes_newton foram utilizadas para armazenar a quantidade de iterações realizadas por cada um dos métodos numéricos implementados, permitindo uma análise comparativa entre os desempenhos.

Além disso, as variáveis epsilon e sigma foram declaradas globalmente para viabilizar a resolução do Problema 4, que envolve uma função de múltiplas variáveis e requer esses parâmetros como entradas fornecidas pelo usuário. A utilização de variáveis globais para esses casos específicos foi adotada como uma solução prática e eficiente, evitando alterações estruturais nos demais trechos do código já implementados.

Definição das funções dos problemas:

```
def funcao_problema_1(x):
      return 5 * x - 100

def funcao_problema_2(x):
      return 1e-12 * (math.exp(x / 0.025) - 1) - 1e-2
```

```
def funcao_problema_3(x):
      if(0.1-x) < 1e-12:
      return float('inf')
      return (x**2)/(0.1-x) - 1e-5

def funcao_problema_4(x):
      return 24 * epsilon * (2 * (sigma/x)**13 - (sigma/x)**7)

def funcao_problema_5(x):
      return x - 1 - math.exp(-x)
```

Para cada um dos cinco problemas propostos no trabalho, foi implementada uma função específica que representa matematicamente a equação a ser resolvida. Essas funções foram definidas separadamente com o objetivo de facilitar a modularização do código, permitindo que sejam utilizadas como parâmetros de entrada nas chamadas dos métodos numéricos (Bisseção, Falsa Posição e Newton-Raphson).

Essa abordagem promove maior flexibilidade e reutilização do código, além de tornar mais clara a associação entre cada função e o respectivo problema a ser analisado.

Geração e salvamento dos gráficos:

```
def   plot_graficos(valores_fx,   valores_tempo,   valores_iteracoes,
numproblema):

      #Define a xLabel com o nome dos métodos
      metodos = ["Bisseção", "Falsa posição", "Newton"]

      #1 - Plot do módulo da função nas raizes encontradas
      mpl.figure(figsize = (10,6)) #Define o tamanho da figura
      mpl.bar(metodos,  valores_fx,  color  =  "skyblue")  #Plota  com  os
```

```
valores de x sendo os nomes dos métodos e os valores de y sendo
|f(raizmetodo)| com a cor skyblue
     mpl.ylabel("|f(x)| na raiz encontrada") #Define a label y
     mpl.title("Precisão dos métodos (quanto menor, melhor [se -1
não convergiu]):") #Define o titulo do grafico
     mpl.grid(True, axis = 'y') #Habilita a grid
     mpl.savefig(f"grafico_erro_problema{numproblema}.png") #Salva o
gráfico com o número do problema correspondente no nome
     mpl.close()

     #2 - Plot do tempo de execução dos métodos
     mpl.figure(figsize = (10,6)) #Repete para o primeiro grafico
     mpl.bar(metodos, valores_tempo, color = "skyblue")
     mpl.ylabel("Tempo de execução do método em segundos")
     mpl.title("Tempo de execução dos métodos (quanto menor,
melhor)")
     mpl.grid(True, axis = 'y')
     mpl.savefig(f"grafico_tempo_problema{numproblema}.png")
     mpl.close()

     #3 - Plot do número de iterações
     mpl.figure(figsize = (10,6))
     mpl.bar(metodos, valores_iteracoes, color = "skyblue")
     mpl.ylabel("Número de iterações")
     mpl.title("Número de iterações dos métodos (quanto menor,
melhor)")
     mpl.grid(True, axis = 'y')
     mpl.savefig(f"grafico_iteracoes_problema{numproblema}.png")
     mpl.close()
```

A função plot_graficos foi desenvolvida com o objetivo de gerar uma análise visual comparativa entre os três métodos numéricos implementados (Bisseção, Falsa Posição e Newton-Raphson). Essa função recebe como parâmetros:

- O módulo da função calculado nas raízes obtidas por cada método (valores_fx),

- Os tempos de execução de cada método (valores_tempo),
- O número de iterações realizadas por cada método até a convergência (valores_iteracoes),
- E o número do problema em análise (numproblema), utilizado na identificação dos arquivos gerados.

Internamente, é criado um vetor de rótulos com os nomes dos métodos, que será utilizado como eixo X nas representações gráficas. A seguir, são gerados três gráficos de barras distintos:

1. Precisão dos Métodos:

   Este gráfico representa o valor absoluto da função $|f(x)|$ avaliada na raiz encontrada por cada método. Valores próximos de zero indicam maior precisão na aproximação da raiz. Caso o método não tenha convergido para uma solução válida, o valor é definido como -1, permitindo a fácil visualização de falhas de convergência no gráfico.

2. Tempo de Execução:

   O segundo gráfico mostra o tempo necessário para a execução de cada método, medido em segundos. Esse parâmetro é útil para comparar a eficiência computacional entre os métodos.

3. Número de Iterações:

   O terceiro gráfico representa a quantidade de iterações necessárias para que cada método atingisse a convergência. Esse dado complementa a análise de desempenho dos métodos, permitindo identificar quais são mais rápidos não apenas em tempo, mas também em número de passos.

Todos os gráficos são salvos automaticamente como imagens no diretório do projeto, com nomes formatados de acordo com o número do problema (por exemplo,

grafico_erro_problema1.png, grafico_tempo_problema1.png, etc.), o que facilita a organização e posterior inclusão no relatório.

Definição dos métodos:

Método da bisseção:

```
def metodo_bissecao(f, a, b, erro = 1e-6, max_iter = 1000):
     global iteracoes_bicessao
     iteracoes = 1 #Começa a contagem utilizando a variável global

     if(f(a) * f(b)>0): #Checa se o método pode convergir ou não
     return None

     xi = (a + b)/2 #Define o primeiro xi

     while ((abs(f(xi)) > erro or abs(b-a) > erro) and iteracoes < max_iter): #Checa os erros da função e também a quantidade maxima de iterações
     iteracoes += 1 #Incrementa a quantidade de iterações
     xi = (a + b) / 2 #Calcula o novo xi

     if(f(a) * f(xi) == 0): #Checa se achou raiz
          return xi
     elif(f(a) * f(xi) < 0): #Se f(a) * f(xi) < 0, então o b se torna xi
          b = xi
     else: #Se não, a se torna xi
          a = xi

     iteracoes_bicessao = iteracoes #Guarda a quantidade de iterações na variável global
     return xi if iteracoes < max_iter else None #Checa se o método convergiu, se convergiu retorna a raiz, se não, retorna None
```

A função metodo_bissecao busca encontrar uma raiz da função f dentro do intervalo entre a e b. Primeiro, verifica se a função muda de sinal no intervalo; se não mudar, retorna None porque o método não pode ser aplicado.

Calcula uma estimativa inicial da raiz como o ponto médio entre a e b. Depois, entra em um laço que continua enquanto o valor da função nesse ponto ainda não for suficientemente pequeno, a diferença entre a e b ainda for grande, e o número máximo de iterações não tiver sido atingido.

Dentro do laço, a estimativa de raiz é atualizada como o novo ponto médio e o intervalo é ajustado com base no sinal da função. A cada iteração, o número de passos é contado.

Ao final, salva o número de iterações em uma variável global. Se o método convergir dentro do limite de iterações, retorna a raiz encontrada; caso contrário, retorna None.

Método da falsa posição:

```
def metodo_falsa_posicao(f, a, b, erro = 1e-6, max_iter = 1000):
      global iteracoes_falsa_posicao
      iteracoes = 1 #Inicializa novamente a variável global

      if(f(a) * f(b)>0): #Checa critério de convergência do método
      return None

      denominador = f(b)-f(a)
      if(denominador < 1e-12): #Checa se na primeira iteração terá
divisão por zero.Se sim, retorna None
      return None
      xi = (a * f(b) - b * f(a)) / (denominador) #Calculo do primeiro
xi

      while (abs(f(xi)) > erro and abs(b-a) > erro and iteracoes <
max_iter):
      iteracoes += 1 #Incrementa iterações
```

```
    denominador = f(b)-f(a) #Novamente checa por divisões por zero
    if(denominador < 1e-12):
        return None
    xi = (a * f(b) - b * f(a)) / (denominador)

    if(f(a) * f(xi) == 0): #Checa se achou raiz
        return xi
    elif(f(a) * f(xi) < 0): #B anda
        b = xi
    else: #A anda
        a = xi

    iteracoes_falsa_posicao = iteracoes #Atualiza o valor da
variável global
    return xi if iteracoes < max_iter else None #Checa se o método
convergiu, se não convergiu retorna None
```

A função metodo_falsa_posicao tenta encontrar uma raiz da função f no intervalo entre a e b. Primeiro, ela verifica se a função muda de sinal no intervalo; se não mudar, o método não pode ser aplicado e retorna None. Em seguida, calcula uma primeira estimativa de raiz usando a fórmula da falsa posição.

O laço while continua executando enquanto o valor da função no ponto estimado ainda for maior que o erro permitido, a diferença entre os extremos do intervalo ainda for grande e o número máximo de iterações não tiver sido atingido. Dentro do laço, a estimativa de raiz é atualizada e o intervalo é ajustado com base no sinal da função em xi. A cada iteração, também é feito um controle para evitar divisão por zero.

Ao final, o número de iterações é salvo em uma variável global e, se o método convergir, a raiz encontrada é retornada; caso contrário, retorna None.

Método de Newton:

```
def metodo_newton(f, x0, erro = 1e-6, max_iter = 1000):
      iteracoes = 1
      global iteracoes_newton
      df = derivada(f, x0) #Calcula a derivada de f(x) no ponto
      if abs(df) < 1e-12: #Checa se a derivada da zero para retornar
None
      return None

      xi = x0 - (f(x0) / derivada(f, x0)) #Calcula o primeiro xi

      while (abs(f(xi)) > erro and iteracoes < max_iter): #Checa os
critérios de parada
      iteracoes += 1 #Incrementa iterações
      xiantes = xi #Guarda o xi anterior numa variável
      df = derivada(f, xi) #Realiza o calculo da derivada

      if abs(df) < 1e-12: #Checa se a derivada da 0 novamente
            iteracoes_newton = iteracoes
            return None

      xi = xi - (f(xi) / df) #Calcula o novo xi

      if (abs(xi - xiantes) / max(xi, 1) < erro): #Checa critérios de
parada
            iteracoes_newton = iteracoes
            return xi

      iteracoes_newton = iteracoes
      return xi if iteracoes < max_iter else None
```

A função metodo_newton tenta encontrar a raiz de uma função f a partir de um valor inicial x0. Primeiro, calcula a derivada da função no ponto inicial. Se a derivada for muito próxima de zero, o método não pode continuar e retorna None para evitar divisão por zero.

Depois, calcula a primeira estimativa da raiz usando a fórmula do método de Newton. Entra então em um laço que continua enquanto o valor da função ainda não estiver suficientemente próximo de zero e o número máximo de iterações não tiver sido alcançado.

A cada passo, calcula a nova derivada no ponto atual e verifica se ela continua válida. Se a derivada for zero, encerra o processo. Caso contrário, atualiza a estimativa da raiz. Também verifica se o novo valor mudou muito pouco em relação ao anterior; se sim, considera que a solução já convergiu.

Por fim, armazena o número de iterações numa variável global e retorna a raiz encontrada, ou None se o método não convergiu a tempo.

Função Análise:

```
def analise_problema(funcao_problema, a0,b0,x0, erro, numproblema):

      global iteracoes_bicessao #Chama as variáveis globais
      global iteracoes_falsa_posicao
      global iteracoes_newton

      print(f"Erro = {erro}\n\n") #Imprime o erro a ser analisado

      print("Resultados obtidos:\n") #Apresentação dos resultados
      print("----------Raizes----------")

      tempo_bissecao = time.time(); #Uso da biblioteca time para
checar o tempo de execução.
      raiz_metodo_bissecao                                    =
metodo_bissecao(funcao_problema,a0,b0,erro)    #Calcula    a    raiz
utilizando método da bisseção
      tempo_bissecao = time.time() - tempo_bissecao #Termina o
calculo de tempo
      print(f"Método da bisseção (a0 = {a0} b0 = {b0}): {"Não
convergiu"     if     raiz_metodo_bissecao     ==     None     else
```

```
raiz_metodo_bissecao}") #Imprime a raiz somente se o método convergiu


    tempo_metodo_falsa_posicao = time.time() #Novamente calculo do
tempo de execução
    raiz_metodo_falsa_posicao                                    =
metodo_falsa_posicao(funcao_problema,a0,b0,erro)  #Calcula  a  raiz
utilizando o método da falsa posição
    tempo_metodo_falsa_posicao       =       time.time()       -
tempo_metodo_falsa_posicao
    print(f"Método da falsa posição (a0 = {a0} b0 = {b0}): {"Não
convergiu"    if    raiz_metodo_falsa_posicao    ==    None    else
raiz_metodo_falsa_posicao}")


    tempo_metodo_newton = time.time()
    raiz_metodo_newton = metodo_newton(funcao_problema, x0, erro)
#Calcula a raiz utilizando o método de newton
    tempo_metodo_newton = time.time() - tempo_metodo_newton
    print(f"Método de Newton (x0 = {x0}): {"Não convergiu" if
raiz_metodo_newton == None else raiz_metodo_newton}")


    print("\n----------Tempos de execução----------"); #Apresenta
os demais resultados
    print(f"Método da bisseção: {tempo_bissecao} segundos")
    print(f"Método da falsa posição: {tempo_metodo_falsa_posicao}
segundos")
    print(f"Método de Newton: {tempo_metodo_newton} segundos")

    print("\n----------Número de iterações----------")
    print(f"Método da bisseção: {iteracoes_bicessao} iterações")
    print(f"Método da falsa posição: {iteracoes_falsa_posicao}
iterações")
    print(f"Método de Newton: {iteracoes_newton} iterações")

    print("\n----------Erros finais----------") #Mostra os erros
finais e também checa se o método chegou a convergir
```

```
    print(f"Método da bisseção: {abs(funcao_problema(raiz_metodo_bissecao)) if raiz_metodo_bissecao is not None else "Não convergiu"}")
    print(f"Método da falsa posição: {abs(funcao_problema(raiz_metodo_falsa_posicao)) if raiz_metodo_falsa_posicao is not None else "Não convergiu"}")
    print(f"Método de Newton: {abs(funcao_problema(raiz_metodo_newton)) if raiz_metodo_newton is not None else "Não convergiu"}")

    #Plotar gráficos
    valores_fx = [abs(funcao_problema(raiz_metodo_bissecao)) if raiz_metodo_bissecao is not None else -1,
                  abs(funcao_problema(raiz_metodo_falsa_posicao)) if raiz_metodo_falsa_posicao is not None else -1,
                  abs(funcao_problema(raiz_metodo_newton)) if raiz_metodo_newton is not None else -1] #Coloca -1 se o método não convergiu para que seja possível visualizar no gráfico. Também guarda os erros num vetor para serem plotados na função plot_graficos
    valores_tempo = [tempo_bissecao, tempo_metodo_falsa_posicao, tempo_metodo_newton] #Guarda os tempos de execução
    valores_iteracoes = [iteracoes_bicessao, iteracoes_falsa_posicao, iteracoes_newton] #Guarda a quantidade de iterações

    plot_graficos(valores_fx, valores_tempo, valores_iteracoes, numproblema) #Chama a função de plotar gráficos
```

A função analise_problema tem como objetivo analisar os métodos numéricos para encontrar as raízes de uma função. Ela recebe uma função (funcao_problema), os intervalos iniciais (a0, b0) e o valor inicial para o método de Newton (x0), o erro tolerado (erro) e o número do problema (numproblema).

Primeiro, a função inicializa variáveis globais para contar o número de iterações de cada método. Em seguida, imprime o erro que será analisado.

A função então calcula a raiz da função utilizando o método da bisseção, medindo o tempo de execução. Se o método não encontrar a raiz, será mostrado "Não convergiu". O mesmo é feito para o método da falsa posição e para o método de Newton, com os respectivos tempos de execução e verificações de convergência.

Depois de calcular as raízes, a função imprime os tempos de execução, o número de iterações e os erros finais de cada método. Para os erros, se algum método não convergiu, será mostrado "Não convergiu".

Por fim, a função prepara os dados necessários para gerar os gráficos, como os erros absolutos das raízes, os tempos de execução e o número de iterações. Esses dados são passados para a função plot_graficos, que gera os gráficos e os salva com o número do problema.

Função main():

```
def main():
      os.system("chcp 65001") #Decodificação UTF-8
      os.system("clear");   #Limpa Console

      print("Selecione o problema para analisar: ") #Menu
      print("1 - Problema 1 - Economia: Equilíbrio de Oferta e
Demanda")
      print("2 - Problema 2 - Engenharia: Equação Não-Linear de um
Diodo")
      print("3 - Problema 3 - Química: Equilíbrio de Dissociação de
um Ácido Fraco")
      print("4 - Problema 4 - Física: Ponto de Equilíbrio no
Potencial de Lennard-Jones")
      print("5 - Problema 5 - Engenharia (Controle): Ganho Crítico
para Estabilidade de um Sistema de Realimentação")
```

```
    print("6 - Sair\n")

    escolha = int(input("Insira sua escolha: "))

    while escolha < 1 or escolha > 6: #Checa se a escolha é valida
    print("Escolha inválida.")
    escolha = int(input("Insira a escolha novamente: "))

    match escolha: #Mudança de menus baseada na escolha
    case 1:
         problema_1()
    case 2:
         problema_2()
    case 3:
         problema_3()
    case 4:
         problema_4()
    case 5:
         problema_5()
    case 6:
         quit()
```

A função main é responsável por apresentar um menu ao usuário, permitindo que ele escolha qual problema deseja analisar. Primeiro, o console é preparado para decodificação UTF-8, e o console é limpo.

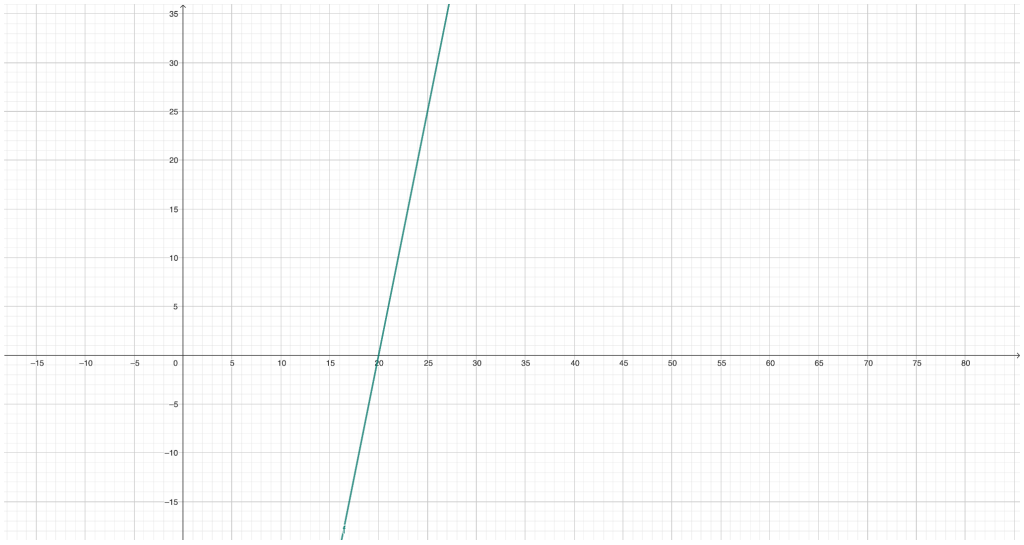
A função valida a escolha do usuário, garantindo que ela esteja dentro do intervalo de 1 a 6. Se o valor inserido for inválido, o programa pede ao usuário que insira a escolha novamente.

A estrutura match é utilizada para direcionar a execução do programa para a função correspondente ao problema escolhido. Caso o usuário escolha a opção 6, o programa é encerrado.

Resultados Obtidos:

Problema 1:

Gráfico da função:



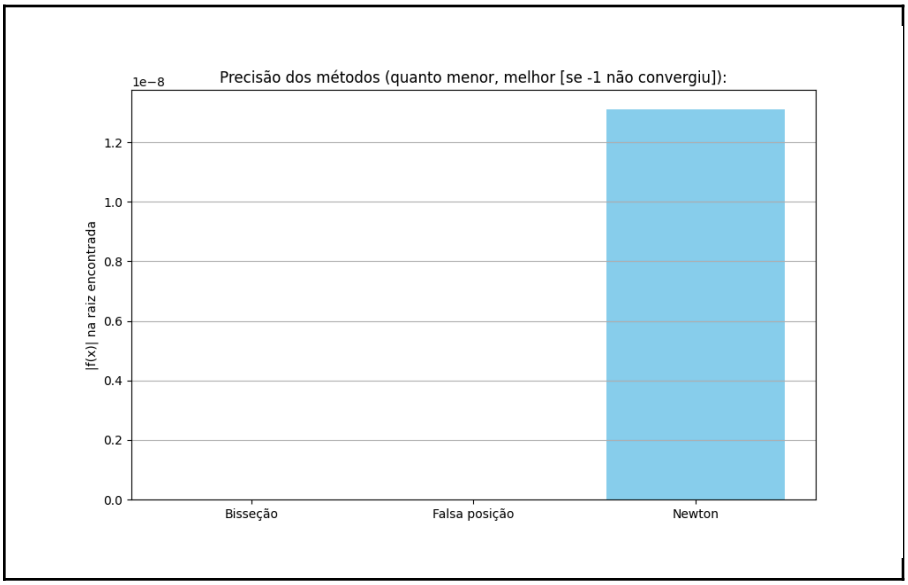

```
Valores utilizados:
a0 = 15;
b0 = 25
x0 = 30;
erro = 0.000001
```

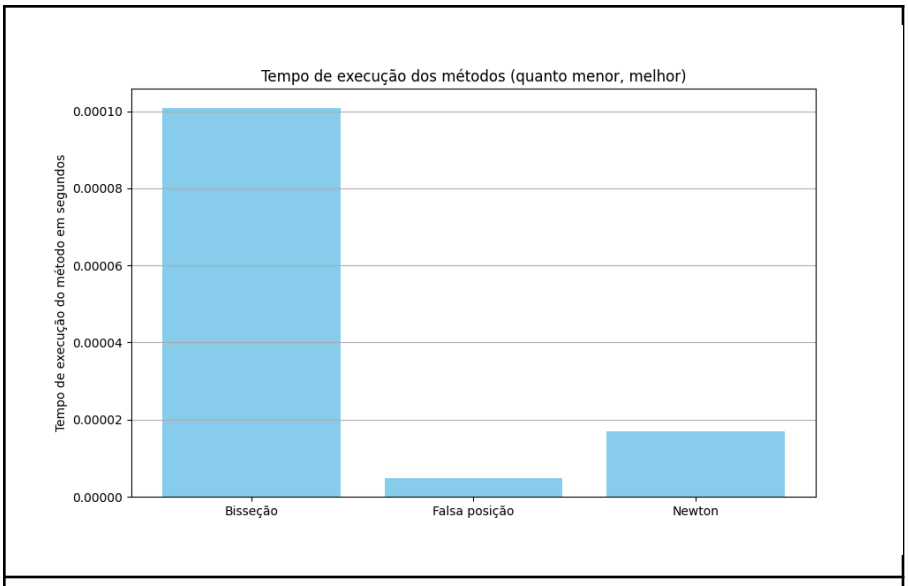
Resultados obtidos:

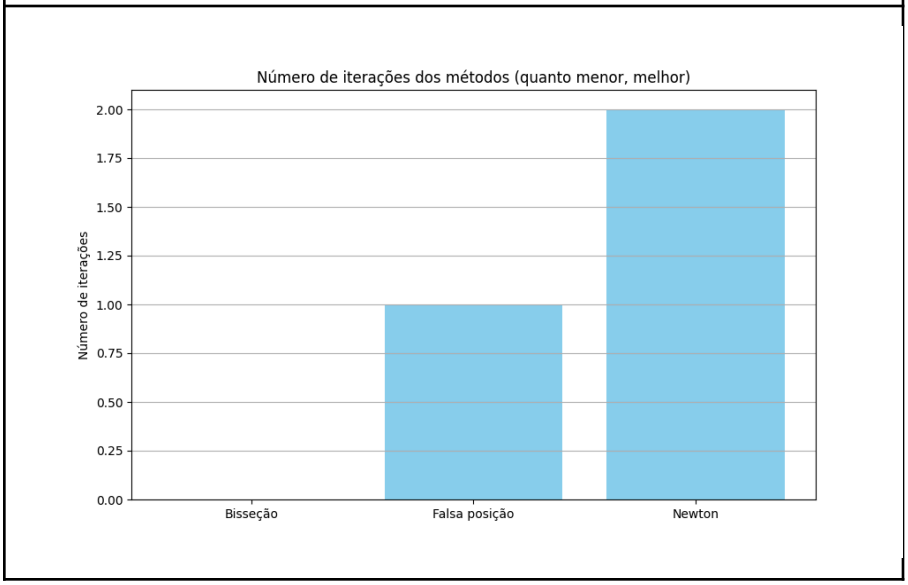
```
----------Raízes----------
Método da bisseção (a0 = 15.0 b0 = 25.0): 20.0
Método da falsa posição (a0 = 15.0 b0 = 25.0): 20.0
```

```
Método de Newton (x0 = 30.0): 20.000000002621338

----------Tempos de execução----------
Método da bisseção: 6.103515625e-05 segundos
Método da falsa posição: 4.100799560546875e-05 segundos
Método de Newton: 1.0967254638671875e-05 segundos

----------Número de iterações----------
Método da bisseção: 1 iterações
Método da falsa posição: 1 iterações
Método de Newton: 2 iterações

----------Erros finais----------
Método da bisseção: 0.0
Método da falsa posição: 0.0
Método de Newton: 1.3106685514685523e-08
```

Gráficos resultados:

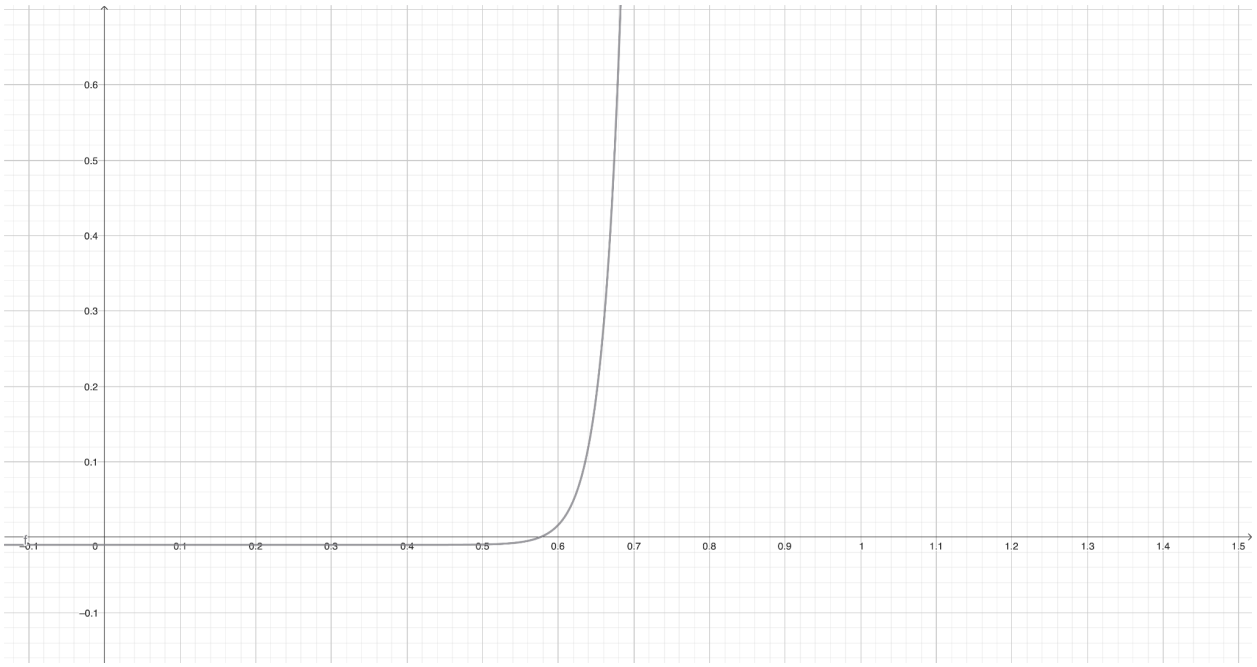



Tempo de execução dos métodos (quanto menor, melhor)
Tempo de execução do método em segundos
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Bisseção
Falsa posição
Newton


Número de iterações dos métodos (quanto menor, melhor)
Número de iterações
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Bisseção
Falsa posição
Newton

Problema 2:

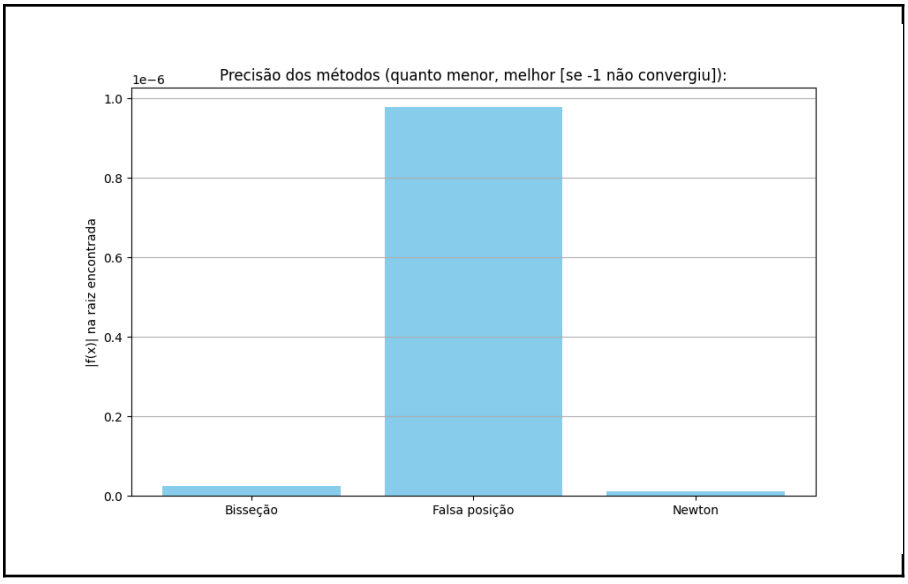
Gráfico da função:

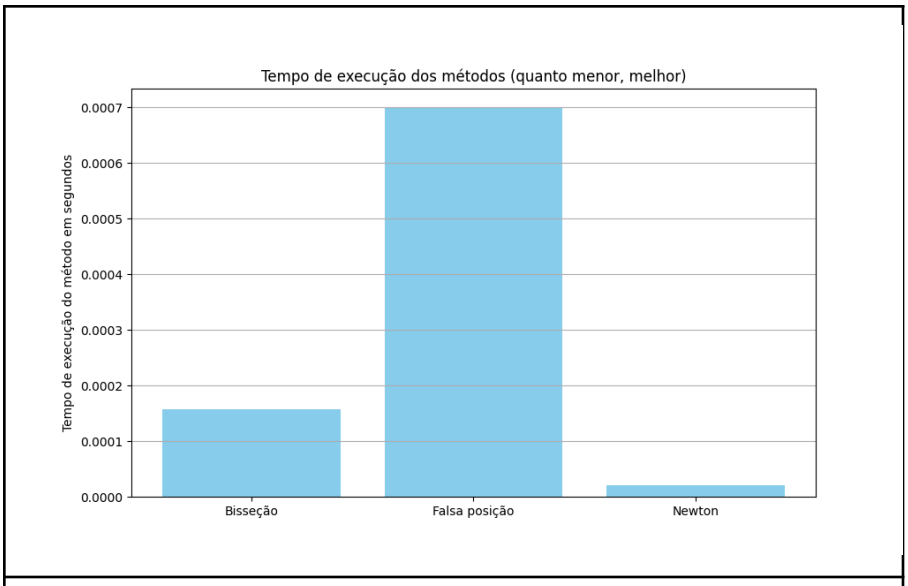


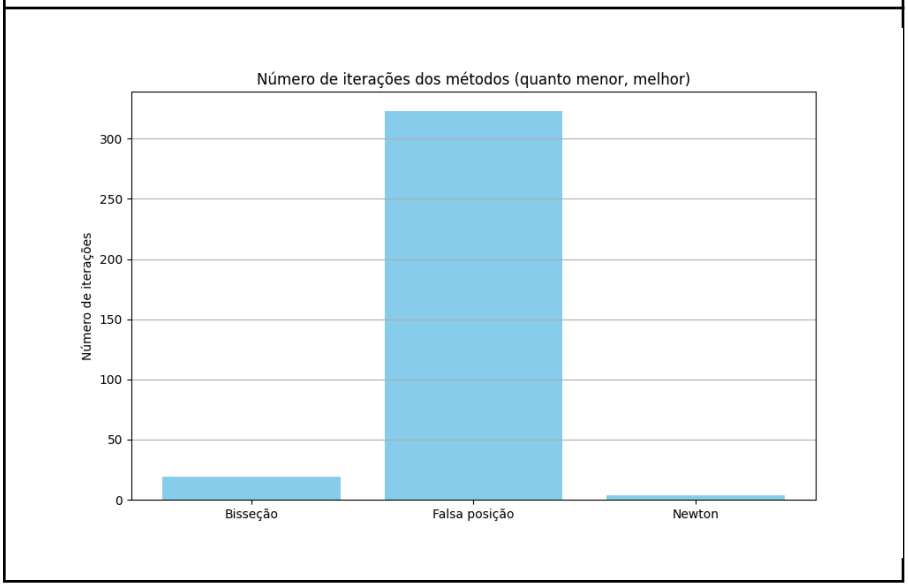

```
Valores utilizados:
a0 = 0.5;
b0 = 0.7;
x0 = 0.6;
erro = 0.000001
```

```
----------Raizes----------
Método da bisseção (a0 = 0.5 b0 = 0.7): 0.5756462097167967
Método da falsa posição (a0 = 0.5 b0 = 0.7): 0.575643827342428
Método de Newton (x0 = 0.6): 0.5756463011704945

----------Tempos de execução----------
Método da bisseção: 0.00015783309936523438 segundos
Método da falsa posição: 0.0006990432739257812 segundos
Método de Newton: 2.09808349609375e-05 segundos

----------Número de iterações----------
Método da bisseção: 19 iterações
Método da falsa posição: 323 iterações
Método de Newton: 4 iterações

----------Erros finais----------
Método da bisseção: 2.5413653599939967e-08
Método da falsa posição: 9.783155752833561e-07
Método de Newton: 1.1167799456035521e-08
```

Resultados obtidos:Gráficos resultados:

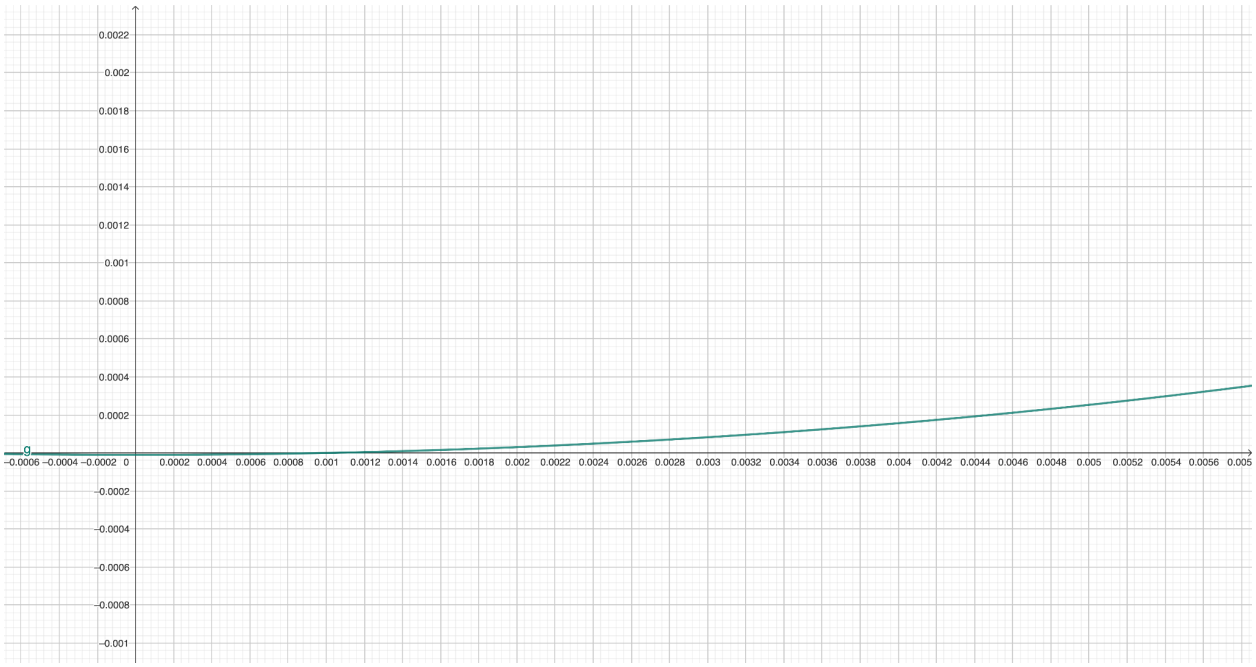



Tempo de execução dos métodos (quanto menor, melhor)
Tempo de execução do método em segundos
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
Bisseção
Falsa posição
Newton


Número de iterações dos métodos (quanto menor, melhor)
Número de iterações
300
250
200
150
100
50
0
Bisseção
Falsa posição
Newton

Problema 3:

Gráfico da função:



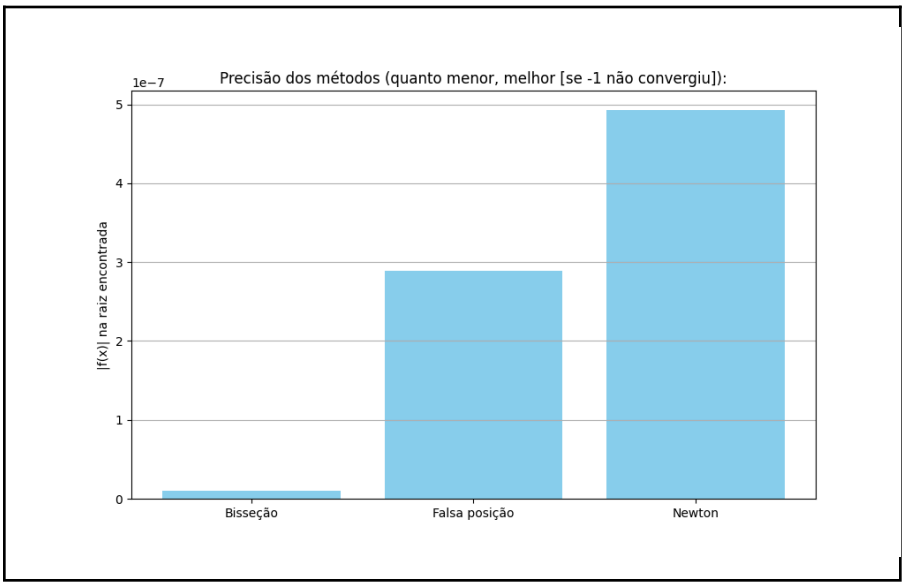

```
Valores utilizados:
x0 = 0.0006
b0 = 0.0014
x0 = 0.0008
erro = 0.000001
```

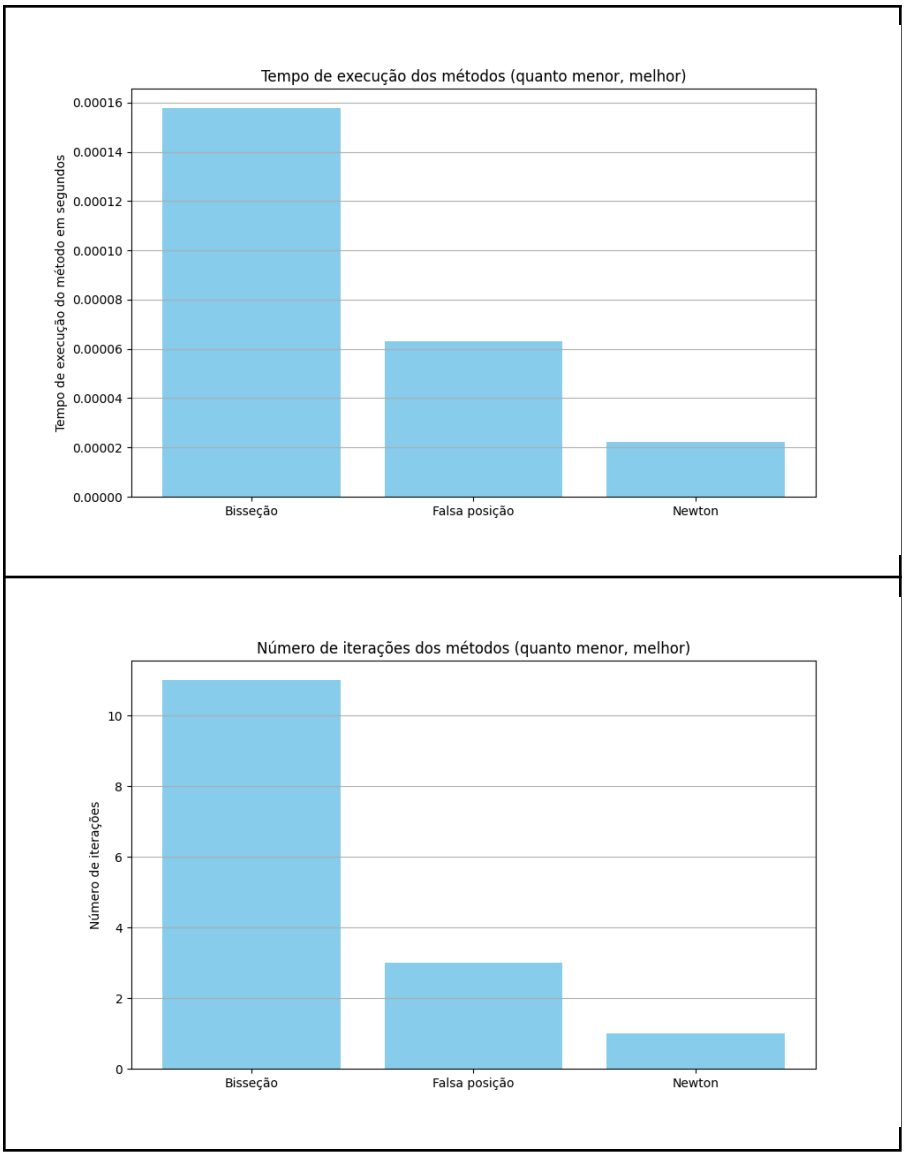
Resultados obtidos:

```
----------Raizes----------
Método da bisseção (a0 = 0.0006 b0 = 0.0014): 0.0009945312500000002
Método da falsa posição (a0 = 0.0006 b0 = 0.0014): 0.000980589853142212
Método de Newton (x0 = 0.0008): 0.0010191164657730512

----------Tempos de execução----------
Método da bisseção: 0.00015783309936523438 segundos
Método da falsa posição: 6.318092346191406e-05 segundos
```

```
Método de Newton: 2.2172927856445312e-05 segundos

----------Número de iterações----------
Método da bisseção: 11 iterações
Método da falsa posição: 3 iterações
Método de Newton: 1 iterações

----------Erros finais----------
Método da bisseção: 9.719465859679019e-09
Método da falsa posição: 2.8921240129222987e-07
Método de Newton: 4.929187710333371e-07
```

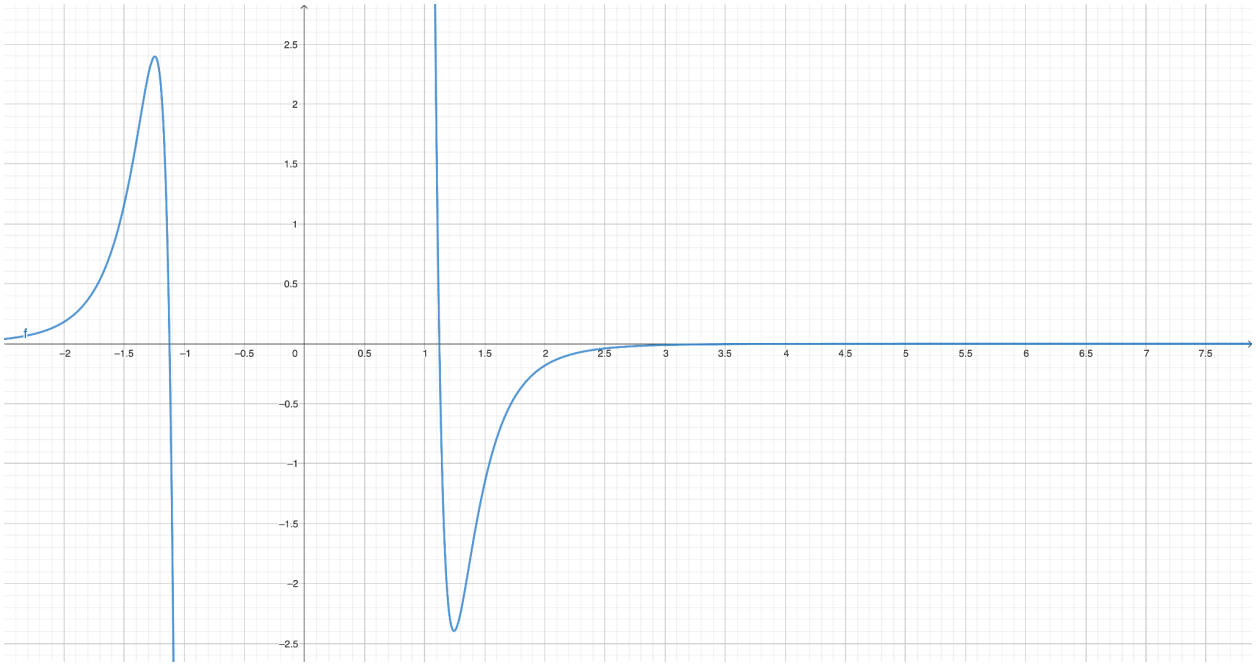
Gráficos resultados:




Tempo de execução dos métodos (quanto menor, melhor)
Tempo de execução do método em segundos
0.00016
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Bisseção
Falsa posição
Newton
Número de iterações dos métodos (quanto menor, melhor)
Número de iterações
10
8
6
4
2
0
Bisseção
Falsa posição
Newton

Problema 4:

Gráfico da função:



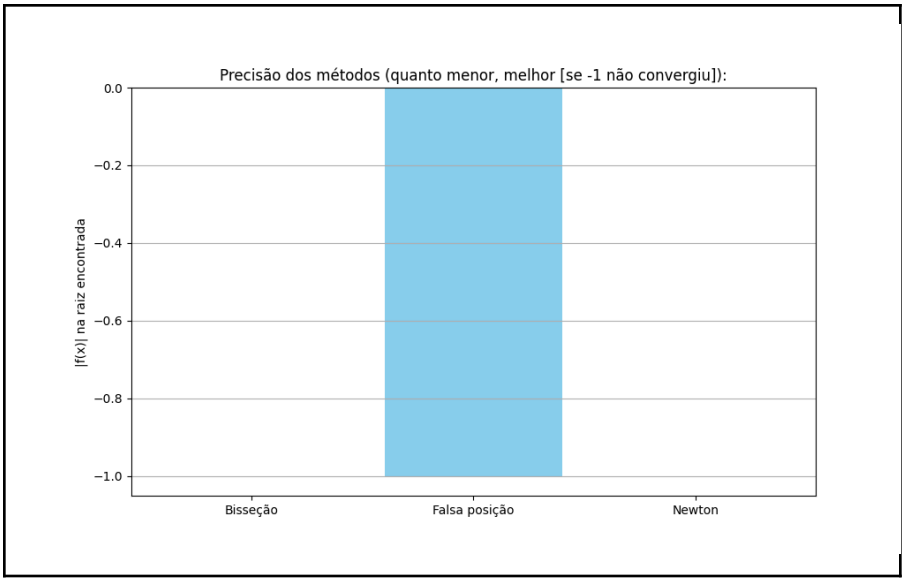

```
Valores utilizados:
ε = 1;
σ = 1;
a0 = 1;
b0 = 1.5;
x0 = 1;
erro = 0.000001;
```

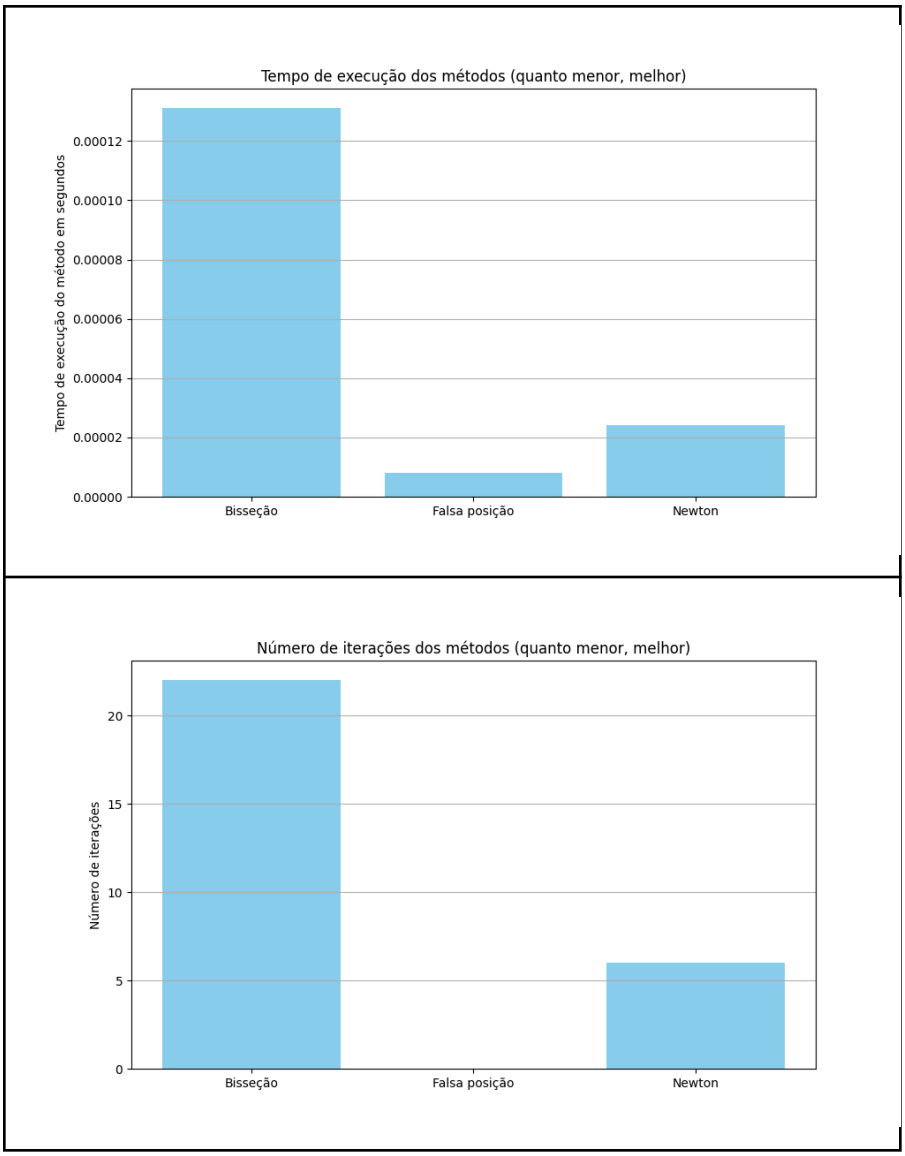
Resultados obtidos:

```
----------Raizes----------
Método da bisseção (a0 = 1.0 b0 = 1.5): 1.1224620342254639
Método da falsa posição (a0 = 1.0 b0 = 1.5): Não convergiu
Método de Newton (x0 = 1.0): 1.1224620482957846

----------Tempos de execução----------
Método da bisseção: 0.00013113021850585938 segundos
```

```
Método da falsa posição: 8.106231689453125e-06 segundos
Método de Newton: 2.4080276489257812e-05 segundos

----------Número de iterações----------
Método da bisseção: 22 iterações
Método da falsa posição: 0 iterações
Método de Newton: 6 iterações

----------Erros finais----------
Método da bisseção: 8.04845339708038e-07
Método da falsa posição: Não convergiu
Método de Newton: 7.765295073625111e-10
```

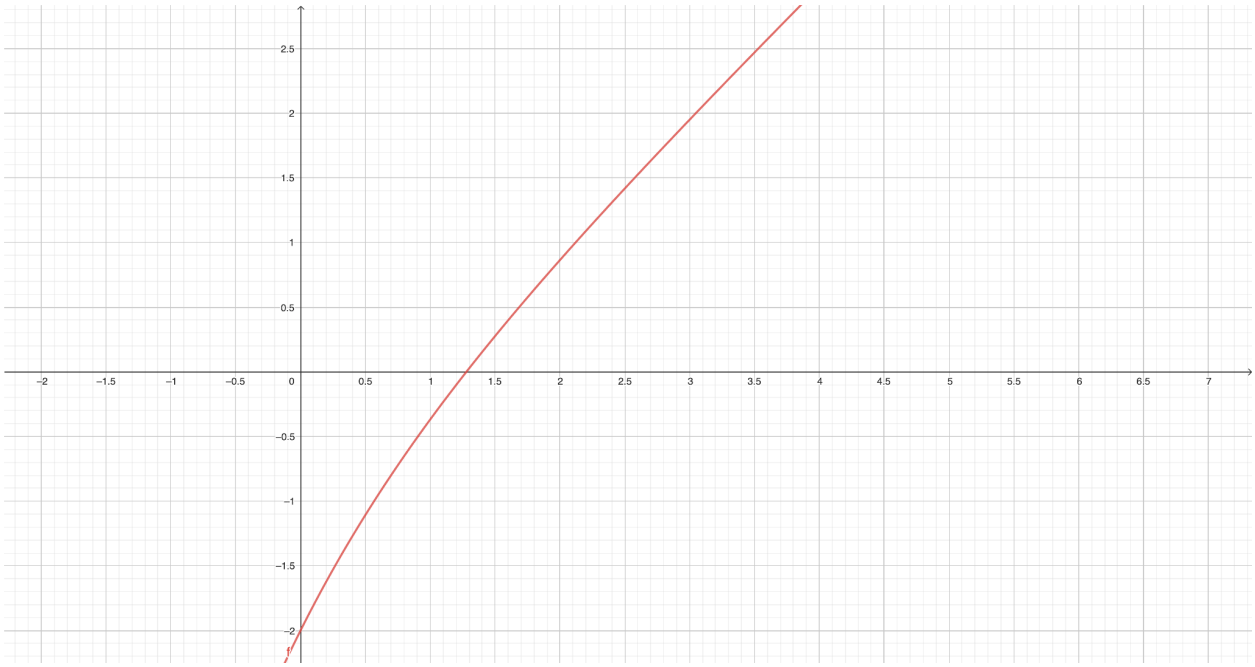
Gráficos resultados:




Tempo de execução dos métodos (quanto menor, melhor)
Tempo de execução do método em segundos
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Bisseção
Falsa posição
Newton
Número de iterações dos métodos (quanto menor, melhor)
Número de iterações
20
15
10
5
0
Bisseção
Falsa posição
Newton

Problema 5:

Gráfico da função:



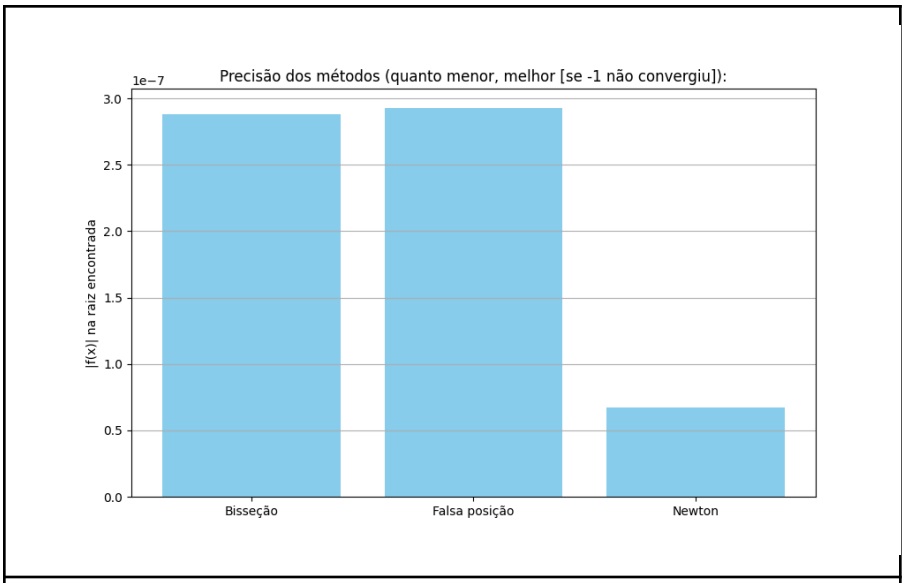

```
Valores utilizados:
a0 = 1;
b0 = 1.5;
x0 = 1.2;
erro = 0.000001;
```

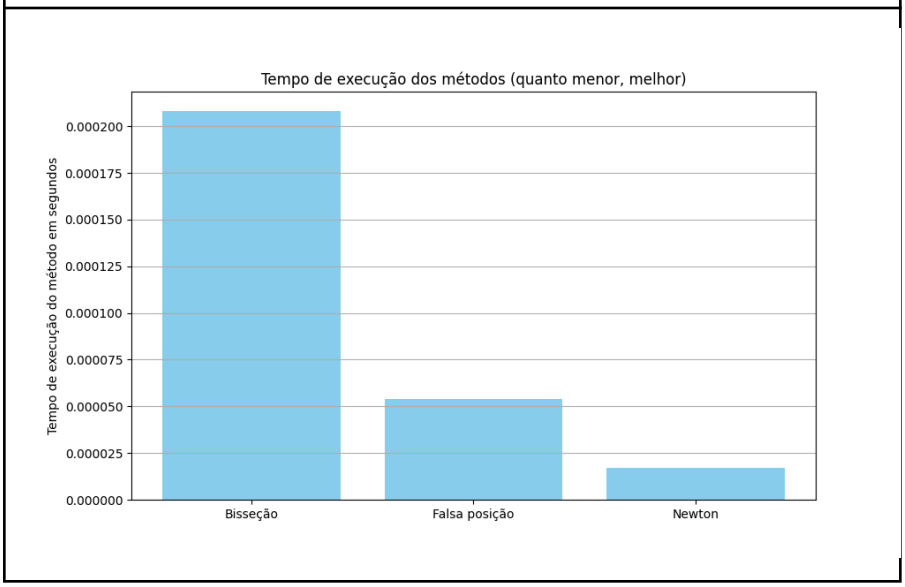
Resultados obtidos:

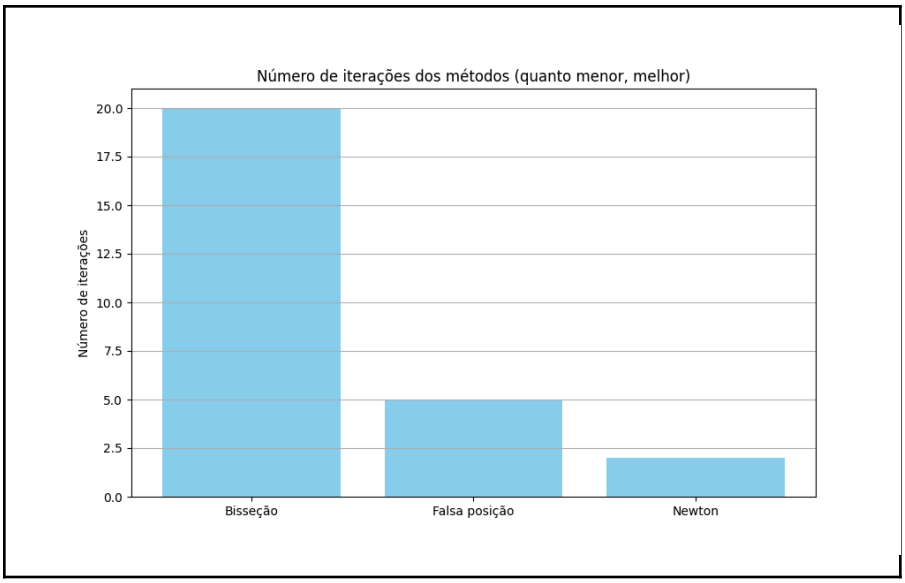
```
----------Raizes----------
Método da bisseção (a0 = 1.0 b0 = 1.5): 1.2784643173217773
Método da falsa posição (a0 = 1.0 b0 = 1.5): 1.27846477180008
Método de Newton (x0 = 1.2): 1.2784644902110947

----------Tempos de execução----------
Método da bisseção: 0.00020813941955566406 segundos
Método da falsa posição: 5.412101745605469e-05 segundos
Método de Newton: 1.6927719116210938e-05 segundos
```

```
----------Número de iterações----------
Método da bisseção: 20 iterações
Método da falsa posição: 5 iterações
Método de Newton: 2 iterações

----------Erros finais----------
Método da bisseção: 2.8821615410956625e-07
Método da falsa posição: 2.92818240876791e-07
Método de Newton: 6.718328543486862e-08
```

Gráficos resultados:







CONCLUSÃO

O desenvolvimento deste trabalho permitiu a aplicação prática dos métodos numéricos de Bisseção, Falsa Posição e Newton-Raphson na resolução de problemas reais de diversas áreas, destacando suas particularidades, vantagens e limitações. A análise comparativa dos resultados evidenciou que a escolha do método adequado depende fortemente das características da função estudada, da disponibilidade de informações sobre derivadas e da definição cuidadosa de intervalos ou estimativas iniciais.

O método da Bisseção mostrou-se robusto e confiável, garantindo convergência sempre que o intervalo inicial satisfaz o teorema do valor intermediário, embora exigindo um maior número de iterações em comparação aos demais. Já o método da Falsa Posição, apesar de se basear em uma estratégia semelhante à Bisseção, apresentou instabilidades em alguns cenários (como no Problema 4), onde não convergiu, possivelmente devido à natureza da função ou à escolha do intervalo inicial. Por outro lado, o método de Newton-Raphson destacou-se pela

rapidez e eficiência, atingindo a convergência com menos iterações e menor tempo de execução na maioria dos casos, mas revelou dependência crítica de uma estimativa inicial próxima à raiz e da existência de derivadas não nulas.

Os gráficos gerados auxiliaram na visualização clara do desempenho de cada método, reforçando que não há um método universalmente superior, mas sim abordagens complementares. Por exemplo, em problemas com derivadas de fácil cálculo e boas estimativas iniciais (como no Problema 2), Newton-Raphson foi ideal, enquanto a Bisseção foi a alternativa mais segura em funções contínuas sem informações adicionais (Problema 4). A análise de erros também ressaltou a importância da tolerância definida, influenciando diretamente na precisão e no esforço computacional.

Este trabalho reforçou a relevância dos métodos numéricos como ferramentas essenciais para resolver problemas complexos em engenharia, física, química e economia, onde soluções analíticas são frequentemente inviáveis. A experiência adquirida na implementação e comparação dos métodos proporcionou insights valiosos para a seleção de estratégias adequadas em futuros projetos, equilibrando precisão, eficiência e robustez computacional.