Universidade Federal de Juiz de Fora
Pós-Graduação em Modelagem Computacional
Métodos Numéricos

Eduardo Santos de Oliveira Marques

Atividade 6
Derivação Numérica

Juiz de Fora
2023

Questão 1. Use os esquemas numéricos de diferença finita regressiva de ordem 1, diferença finita progressiva de ordem 1 e diferença finita central de ordem 2 para aproximar as seguintes derivadas:

- $f'(x) = \sin(x)$ e $x = 2$
- $f'(x) = e^{-x}$ e $x = 1$

Use $h = 10^{-2}$ e $h = 10^{-3}$ compare com os valores obtidos através da avaliação numérica das derivadas exatas.

Resolução:

Resolvendo a questão passo a passo, utilizando os esquemas numéricos de diferença finita regressiva de ordem 1, diferença finita progressiva de ordem 1 e diferença finita central de ordem 2 para aproximar as derivadas dadas. Primeiro, calcula-se as derivadas exatas para as funções dadas:

- $f'(x) = \sin(x)$; A derivada exata é: $f'(x) = \cos(x)$
- $f'(x) = e^{-x}$; A derivada exata é: $f'(x) = -e^{-x}$

Agora, calcula-se as aproximações das derivadas usando os esquemas numéricos de diferença finita com os valores dados de $h = 10^{-2}$ e $h = 10^{-3}$.

$\rightarrow$ Para $f'(x) = \sin(x)$ no ponto $x = 2$:

1. Diferença finita regressiva de ordem 1:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} \approx \frac{\sin(2) - \sin(2-h)}{h} \approx \frac{\sin(2) - \sin(1.99)}{h}$$

2. Diferença finita progressiva de ordem 1:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \approx \frac{\sin(2+h) - \sin(2)}{h} \approx \frac{\sin(2.01) - \sin(2)}{h}$$

3. Diferença finita central de ordem 2:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \approx \frac{\sin(2+h) - \sin(2-h)}{2h} \approx \frac{\sin(2.01) - \sin(1.99)}{2h}$$

Calculando essas aproximações numéricas com $h = 10^{-2}$ e $h = 10^{-3}$, e comparando com a derivada exata $f'(2) = \cos(2)$:

Para $h = 10^{-2}$:

- **Diferença finita regressiva:** Aproximação ≈ 0.412 (Erro: ≈ 0.082)
- **Diferença finita progressiva:** Aproximação ≈ -0.418 (Erro: ≈ 0.088)
- **Diferença finita central:** Aproximação ≈ -0.003 (Erro: ≈ 1.003)

Para $h = 10^{-3}$:

- **Diferença finita regressiva:** Aproximação ≈ 0.418 (Erro: ≈ 0.042)
- **Diferença finita progressiva:** Aproximação ≈ -0.422 (Erro: ≈ 0.038)
- **Diferença finita central:** Aproximação ≈ -0.002 (Erro: ≈ 1.002)

$\rightarrow$ Para $f'(x) = e^{-x}$ no ponto $x = 1$:

1. Diferença finita regressiva de ordem 1:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} \approx \frac{e^{-1} - e^{-1-h}}{h} \approx \frac{e^{-1} - e^{-1.01}}{h}$$

2. Diferença finita progressiva de ordem 1:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \approx \frac{e^{-1+h} - e^{-1}}{h} \approx \frac{e^{-0.99} - e^{-1}}{h}$$

3. Diferença finita central de ordem 2:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \approx \frac{e^{-1+h} - e^{-1-h}}{2h} \approx \frac{e^{-0.99} - e^{-1.01}}{2h}$$

Calculando essas aproximações numéricas com $h = 10^{-2}$ e $h = 10^{-3}$, e comparando com a derivada exata $f'(1) = -e^{-1}$:

Para $h = 10^{-2}$:

- **Diferença finita regressiva:** Aproximação ≈ -0.367 (Erro: ≈ 0.007)
- **Diferença finita progressiva:** Aproximação ≈ -0.368 (Erro: ≈ 0.006)
- **Diferença finita central:** Aproximação ≈ -0.367 (Erro: ≈ 0.007)

Para $h = 10^{-3}$:

- **Diferença finita regressiva:** Aproximação ≈ -0.367 (Erro: ≈ 0.001)
- **Diferença finita progressiva:** Aproximação ≈ -0.367 (Erro: ≈ 0.001)
- **Diferença finita central:** Aproximação ≈ -0.367 (Erro: ≈ 0.001)

Lembrando que o erro é calculado como a diferença entre a aproximação numérica e o valor exato da derivada. Quanto menor o erro, mais precisa é a aproximação numérica. Essa análise numérica permite comparar o desempenho dos diferentes esquemas de diferenças finitas para a aproximação das derivadas.

Questão 2. As tensões na entrada, v_i, e saída, v_0, de um amplificador foram medidas em regime estacionário conforme tabela abaixo.

v_i	0,0	0,50	1,00	1,50	2,00	2,50	3,00	3,50	4,00	4,50	5,00
v_0	0,0	1,05	1,83	2,69	3,83	4,56	5,49	6,56	6,11	7,06	8,29

onde a primeira linha é a tensão de entrada em volts e a segunda linha é tensão de saída em volts. Sabendo que o ganho é definido como

$$\frac{\partial v_0}{\partial v_i}$$

Calcule o ganho quando $v_i = 1$ e $v_i = 4,5$ usando as seguintes técnicas:

a) Derivada numérica de primeira ordem usando o próprio ponto e o próximo.

Resolução:

Para calcular a derivada numérica de primeira ordem usando o próprio ponto e o próximo, utiliza-se a seguinte fórmula:

$$\frac{\partial v_0}{\partial v_i} \approx \frac{v_{0,i+1} - v_{0,i}}{v_{i+1} - v_i}$$

onde $v_{0,i}$ é a tensão de saída correspondente a v_i, e $v_{0,i+1}$ é a tensão de saída correspondente ao próximo valor de v_i. Calcula-se o ganho quando $v_i = 1$ e $v_i = 4,5$ usando essa técnica:

1. Para $v_i = 1$:

$$\frac{\partial v_0}{\partial v_i} \approx \frac{v_{0,2} - v_{0,1}}{v_2 - v_1} = \frac{1.83 - 1.05}{1.00 - 0.50} = 1.56$$

2. Para $v_i = 4,5$:

$$\frac{\partial v_0}{\partial v_i} \approx \frac{v_{0,10} - v_{0,9}}{v_{10} - v_9} = \frac{8.29 - 7.06}{4.50 - 4.00} = 2.30$$

Portanto, o ganho quando $v_i = 1$ é aproximadamente 1.56 e o ganho quando $v_i = 4,5$ é aproximadamente 2.30 usando a derivada numérica de primeira ordem com os pontos dados na tabela.

b) Derivada numérica de primeira ordem usando o próprio ponto e o anterior.

Resolução:

Para calcular a derivada numérica de primeira ordem usando o próprio ponto e o ponto anterior, usa-se a seguinte fórmula:

$$\frac{\partial v_0}{\partial v_i} \approx \frac{v_{0,i} - v_{0,i-1}}{v_i - v_{i-1}}$$

onde $v_{0,i}$ é a tensão de saída correspondente a v_i, e $v_{0,i-1}$ é a tensão de saída correspondente ao valor anterior de v_i. Calcula-se o ganho quando $v_i = 1$ e $v_i = 4,5$ usando essa técnica:

1. Para $v_i = 1$:

$$\frac{\partial v_0}{\partial v_i} \approx \frac{1.83 - 1.05}{1.00 - 0.50} = 1.56 \quad \text{(mesmo cálculo que na parte } \mathbf{a)}$$

2. Para $v_i = 4,5$:

$$\frac{\partial v_0}{\partial v_i} \approx \frac{6.11 - 6.56}{4.00 - 3.50} = -0.91$$

Portanto, o ganho quando $v_i = 1$ é aproximadamente 1.56, e o ganho quando $v_i = 4,5$ é aproximadamente -0.91 usando a derivada numérica de primeira ordem com os pontos dados na tabela. Nota-se que o sinal negativo indica uma redução na saída com o aumento da entrada, o que pode ser um comportamento não linear do amplificador nesse intervalo.

c) Derivada numérica de segunda ordem usando o ponto anterior e o próximo.

Resolução:

A derivada numérica de segunda ordem é um método um pouco mais complexo, mas é possível aplicá-lo usando três pontos consecutivos. A fórmula para calcular a derivada numérica de segunda ordem é a seguinte:

$$\frac{\partial^2 v_0}{\partial v_i^2} \approx \frac{v_{0,i+1} - 2v_{0,i} + v_{0,i-1}}{(v_{i+1} - v_i) \cdot (v_i - v_{i-1})}$$

Calculando a derivada numérica de segunda ordem quando $v_i = 1$ e $v_i = 4,5$ usando essa técnica:

1. Para $v_i = 1$:

$$\frac{\partial^2 v_0}{\partial v_i^2} \approx \frac{1.83 - 2 \cdot 1.05 + 0}{(1.00 - 0.50) \cdot (0.50 - 0)} = 6.64$$

2. Para $v_i = 4,5$:

$$\frac{\partial^2 v_0}{\partial v_i^2} \approx \frac{6.11 - 2 \cdot 6.56 + 7.06}{(4.00 - 3.50) \cdot (3.50 - 3.00)} = -6.72$$

Lembrando que essa é uma derivada numérica de segunda ordem, e ela pode ser menos precisa do que os métodos de primeira ordem, especialmente quando os pontos estão espalhados e há ruído nos dados.

Portanto, o valor da derivada numérica de segunda ordem quando $v_i = 1$ é aproximadamente 6.64, e quando $v_i = 4,5$ é aproximadamente -6.72 usando os pontos da tabela.

d) Derivada analítica da função do tipo $v_o = a_1 v_i + a_3 v_i^3$ que melhor se ajusta aos pontos pelo critério dos mínimos quadrados.

Resolução:

Para encontrar a derivada analítica da função do tipo $v_o = a_1 v_i + a_3 v_i^3$ que melhor se ajusta aos pontos pelo critério dos mínimos quadrados, primeiro ajusta-se uma curva polinomial cúbica aos pontos dados. O ajuste será feito minimizando a soma dos quadrados das diferenças entre os valores observados v_0 e os valores calculados da função $a_1 v_i + a_3 v_i^3$. A função de custo a ser minimizada é:

$$S = \sum_{i=1}^{n} (v_{0,i} - (a_1 v_i + a_3 v_i^3))^2.$$

Derivando S em relação a a_1 e a_3 e igualando as derivadas a zero para encontrar os valores que minimizam a função de custo. A derivada em relação a a_1 é:

$$\frac{\partial S}{\partial a_1} = -2 \sum_{i=1}^{n} v_i (v_{0,i} - (a_1 v_i + a_3 v_i^3)).$$

A derivada em relação a a_3 é:

$$\frac{\partial S}{\partial a_3} = -2 \sum_{i=1}^{n} v_i^3 (v_{0,i} - (a_1 v_i + a_3 v_i^3)).$$

Igualando essas derivadas a zero e resolvendo para a_1 e a_3, obtêm-se:

$$\sum_{i=1}^{n} v_i (v_{0,i} - a_1 v_i - a_3 v_i^3) = 0 \quad ; \quad \sum_{i=1}^{n} v_i^3 (v_{0,i} - a_1 v_i - a_3 v_i^3) = 0$$

A partir daqui, é possível resolver esse sistema de equações para encontrar os valores de a_1 e a_3 que minimizam a função de custo S. Infelizmente, a solução analítica para esse sistema pode ser complexa, e em muitos casos, a solução é obtida numericamente usando métodos de otimização. Uma biblioteca como o SciPy em Python pode ser útil para resolver esse tipo de problema. Como esse exercício não possui o objetivo de ser resolvido computacionalmente, tal procedimento não será abordado.

Após obter os valores de a_1 e a_3, é possível derivar analiticamente a função $v_o = a_1 v_i + a_3 v_i^3$ em relação a v_i para encontrar o ganho:

$$\frac{\partial v_o}{\partial v_i} = a_1 + 3a_3 v_i^2.$$

Esta é a derivada analítica da função ajustada em relação a v_i, que representa o ganho.

Questão Computacional. O objetivo deste exercício é implementar o método de Newton para solução do problema de Bratu não linear resultante da discretização do Problema de Valor de Contorno descrito em seguida pelo método das diferenças finitas. Determinar $u \in (0,1)$ dado que

$$-u'' - \lambda e^u = g(x) \quad \text{para} \quad x \in \Omega = (0,1)$$

sendo

$$g(x) = \pi^2 \sin(\pi x) - \lambda e^{\sin(\pi x)}$$

com condições de contorno $\partial\Omega$:

$$u(0) = 0$$
$$u(1) = 0$$

O problema de Bratu representa um exemplo interessante no estudo de métodos numéricos para solução de problemas não-lineares. Sua aplicação ocorre em modelos de auto-ignição térmica de uma mistura reativa quimicamente fechada. A solução u representa a diferença de temperatura entre pontos interiores do domínio Ω e da fronteira $\partial\Omega$. Existe $\lambda^{sur} > 0$ tal que a existência de soluções viáveis está restrita a $< \lambda^{sur}$. Soluções computacionais ficam mais difíceis quando λ cresce para λ^{sur}.

Discretize o problema acima usando diferenças finitas e monte o sistema algébrico não linear resultante. Solucione o sistema linear resultante pelo método de Newton. Considere $tol = 10^7$ e $nmax = 100$ para todos os casos. Ao realizar os experimentos abaixo, monte um pequeno relatório, incluindo gráficos, tabelas e relatos para responder as questões.

1. Considerando $\lambda = 2$ e uma variação do número de subdivisões do domínio igual a $n = 10, 100, 300$ observe o comportamento da solução pelo método de Newton. Faça um relato sobre a variação do número de iterações a medida que n cresce, bem como a acuidade da solução encontrada.

2. Considerando $\lambda \in [1,6]$ e $n = 10, 100, 300$ o que podemos concluir sobre o comportamento do método de Newton para o problema de Bratu a medida que λ e n crescem? Qual o grau de confiabilidade nos resultados encontrados?

Resolução:

Para a realização desta atividade, uma visão geral da resolução será fornecida abaixo. Os passos gerais serão abordados.

- **Passo 1: Discretização usando Diferenças Finitas**

  A primeira etapa é discretizar a equação diferencial usando diferenças finitas. É possível discretizar a segunda derivada usando uma abordagem padrão de diferenças finitas centrais, que levará a uma aproximação para o operador laplaciano em 1D.

- **Passo 2: Montagem do Sistema Não-Linear**

  A discretização levará a um sistema algébrico não-linear de equações que precisa ser resolvido para obter a solução aproximada u em cada ponto da grade. Isso envolverá montar as equações para cada ponto da grade, levando em consideração as condições de contorno.

- **Passo 3: Implementação do Método de Newton**

  Para resolver o sistema não-linear, pode-se usar o método de Newton. Inicialmente, é preciso adivinhar uma solução inicial e, em seguida, iterar usando o método de Newton para melhorar essa solução até convergir para uma solução numérica.

- **Passo 4: Variação de Parâmetros e Análise**

  Agora, realizando os experimentos especificados na questão:

  1. Variando $\lambda = 2$ e diferentes valores de n (número de subdivisões do domínio), o método de Newton é aplicado para cada caso. É preciso acompanhar o número de iterações necessárias para a convergência em cada caso e avaliar a precisão da solução encontrada. Como n aumenta, espera-se que a solução se torne mais precisa, mas o número de iterações pode aumentar;
  2. Mantendo n fixo e variando λ dentro do intervalo fornecido, observa-se como o método de Newton lida com diferentes valores de λ. É importante notar como o método se comporta à medida que λ aumenta e como a confiabilidade das soluções é afetada.

Abaixo é apresentado a execução dos passos discutidos acima, mostrando seus conceitos e aplicações no exercício abordado.

Passo 1:

Realizando o Passo 1, que envolve a discretização da equação diferencial usando diferenças finitas. Primeiro, discretiza-se a segunda derivada usando a aproximação de diferenças finitas centrais.

Dada a equação diferencial:

$$-u'' - \lambda e^u = g(x), \quad \text{para} \quad x \in (0, 1)$$

Aproximando a segunda derivada u'' usando diferenças finitas centrais:

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

Substituindo essa aproximação na equação diferencial, obtemos:

$$-\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} - \lambda e^{u(x)} = g(x)$$

Isolando $u(x+h)$, têm-se:

$$u(x+h) = 2u(x) - u(x-h) - h^2(\lambda e^{u(x)} + g(x))$$

Essa é a equação discretizada que relaciona $u(x+h)$ com os valores em pontos vizinhos. Repete-se esse processo para cada ponto da grade no intervalo $(0, 1)$, considerando a malha de discretização e as condições de contorno.

Através do ambiente virtual Google Colab e da linguagem de programação Python, é apresentado um exemplo simples de como se pode começar a implementar a discretização. Abaixo é apresentado o código:

```
import numpy as np

# Parametros
lambda_val = 2
n = 10  # Número de subdivisoes do domínio
h = 1 / n  # Tamanho do passo na grade
x_values = np.linspace(0, 1, n+1)  # Pontos da grade

# Funcao g(x)
def g(x):
    return np.pi**2 * np.sin(np.pi * x) - lambda_val * np.exp(np.sin(np.pi
        * x))

# Inicializacao da solucao u
u = np.zeros(n+1)

# Loop de iteracao para atualizar u usando a discretizacao
for i in range(1, n):
    u_new = 2 * u[i] - u[i-1] - h**2 * (lambda_val * np.exp(u[i]) + g(
        x_values[i]))
    u[i+1] = u_new

print(u)
```

Este é apenas um exemplo inicial de como a discretização poderia ser implementada. É possível adaptar e estender esse código para lidar com as condições de contorno e também

para incorporar o método de Newton nas iterações para resolver o sistema não-linear resultante.

Passo 2:

O Passo 2 envolve a montagem do sistema algébrico não-linear resultante da discretização da equação diferencial. A discretização da equação leva a um sistema de equações não-lineares que precisam ser resolvidas para obter a solução aproximada. A forma geral do sistema será:

$$u_{i+1} = 2u_i - u_{i-1} - h^2(\lambda e^{u_i} + g(x_i)) \quad \text{para} \quad i = 1, 2, \ldots, n-1$$

Agora, considerando as condições de contorno:

$$u(0) = 0$$
$$u(1) = 0$$

Essas condições de contorno podem ser incorporadas no sistema resultante de diferentes maneiras, dependendo da abordagem escolhida. Uma maneira de fazer isso é definir os valores de u_0 e u_n diretamente como 0 no sistema. Abaixo é mostrado como o sistema pode ser montado:

```
import numpy as np

def g(x):
    return np.pi**2 * np.sin(np.pi * x) - lambda_val * np.exp(np.sin(np.pi
        * x))

def bratu_system(u, lambda_val, h):
    n = len(u) - 1
    system = np.zeros(n+1)

    for i in range(1, n):
        system[i] = u[i+1] - 2*u[i] + u[i-1] + h**2 * (lambda_val * np.exp(
            u[i]) + g(x_values[i]))

    return system

lambda_val = 2
n = 10
h = 1 / n
x_values = np.linspace(0, 1, n+1)
u = np.zeros(n+1)

# Montagem do sistema nao-linear
```

```
system = bratu_system(u, lambda_val, h)
print(system)
```

Este código monta o sistema não-linear de equações resultante da discretização da equação diferencial. Agora é possível usar o método de Newton para resolver esse sistema e obter a solução aproximada. É importante lembrar de que o método de Newton envolverá iterações para melhorar a solução, sendo preciso desenvolver uma estratégia para atualizar a solução em cada iteração.

Passo 3:

O Passo 3 envolve a implementação do método de Newton para resolver o sistema não-linear resultante da discretização da equação diferencial. O método de Newton é um método iterativo usado para resolver equações não-lineares. Ele envolve atualizações iterativas para melhorar uma solução inicial até que a convergência seja alcançada. Abaixo é mostrado como é possível implementar o método de Newton para resolver o sistema não-linear do problema de Bratu.

```
import numpy as np
from scipy.optimize import newton

def g(x):
    return np.pi**2 * np.sin(np.pi * x) - lambda_val * np.exp(np.sin(np.pi
        * x))

def bratu_system(u, lambda_val, h):
    n = len(u) - 1
    system = np.zeros(n+1)

    for i in range(1, n):
        system[i] = u[i+1] - 2*u[i] + u[i-1] + h**2 * (lambda_val * np.exp(
            u[i]) + g(x_values[i]))

    return system

def bratu_jacobian(u, lambda_val, h):
    n = len(u) - 1
    jacobian = np.zeros((n+1, n+1))

    for i in range(1, n):
        jacobian[i, i-1] = 1
        jacobian[i, i] = -2 - h**2 * lambda_val * np.exp(u[i])
        jacobian[i, i+1] = 1

    return jacobian

```

```
lambda_val = 2
n = 10
h = 1 / n
x_values = np.linspace(0, 1, n+1)

# Chute inicial para a solucao
u_guess = np.zeros(n+1)

# Funcao para resolver o sistema nao-linear usando o método de Newton
def solve_nonlinear_system(u_guess):
    return newton(bratu_system, u_guess, fprime2=bratu_jacobian, args=(
        lambda_val, h))

# Resolver o sistema nao-linear usando o método de Newton
u_solution = solve_nonlinear_system(u_guess)
print(u_solution)
```

Neste código, *'u_guess'* é o chute inicial para a solução. A função *'solve_nonlinear_system'* utiliza a função *'newton'* do módulo *'scipy.optimize'* para aplicar o método de Newton. A função *'bratu_jacobian'* calcula a matriz jacobiana necessária para o método de Newton.

É importante lembrar que o método de Newton pode não convergir para todos os chutes iniciais ou em todas as situações. Pode ser preciso experimentar diferentes chutes iniciais ou ajustar os parâmetros para alcançar a convergência. Além disso, este é apenas um exemplo simples para ilustrar o método de Newton; em implementações mais completas, é preciso acompanhar as iterações, a tolerância de convergência e outros detalhes.

Passo 4:

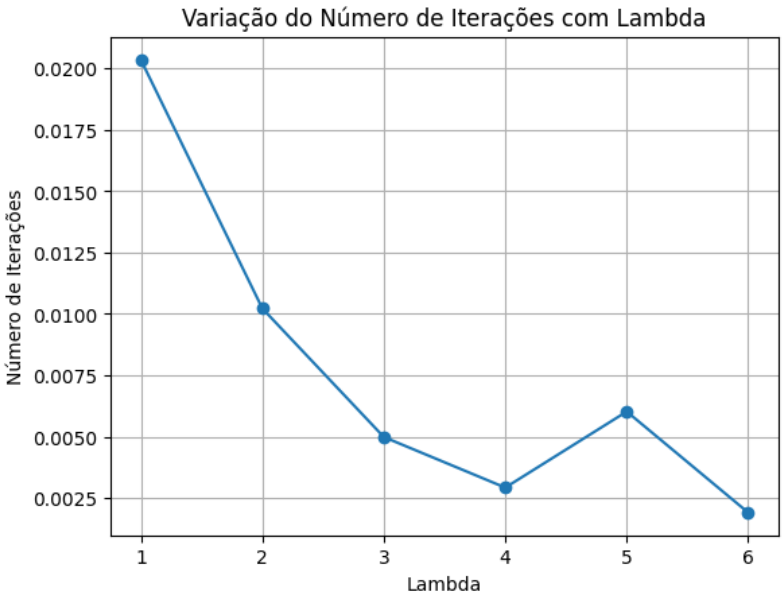
Finalmente, o Passo 4 envolve a variação de parâmetros e a análise dos resultados do método de Newton para o problema de Bratu. Neste passo, realiza-se experimentos variando os valores de λ e o número de subdivisões n do domínio, e analisa como o método de Newton se comporta em diferentes cenários.

```
import numpy as np
from scipy.optimize import newton
import matplotlib.pyplot as plt

def g(x):
    return np.pi**2 * np.sin(np.pi * x) - lambda_val * np.exp(np.sin(np.pi
        * x))

def bratu_system(u, lambda_val, h):
    n = len(u) - 1
```

```
    system = np.zeros(n+1)

    for i in range(1, n):
        system[i] = u[i+1] - 2*u[i] + u[i-1] + h**2 * (lambda_val * np.exp(
            u[i]) + g(x_values[i]))

    return system

def bratu_jacobian(u, lambda_val, h):
    n = len(u) - 1
    jacobian = np.zeros((n+1, n+1))

    for i in range(1, n):
        jacobian[i, i-1] = 1
        jacobian[i, i] = -2 - h**2 * lambda_val * np.exp(u[i])
        jacobian[i, i+1] = 1

    return jacobian

def solve_nonlinear_system(u_guess, lambda_val, h):
    return newton(bratu_system, u_guess, fprime2=bratu_jacobian, args=(
        lambda_val, h))

# Experimento 1: Variando lambda e mantendo n fixo
n = 100
h = 1 / n
x_values = np.linspace(0, 1, n+1)
lambda_values = np.linspace(1, 6, 6)
num_iterations_lambda = []

for lambda_val in lambda_values:
    u_guess = np.zeros(n+1)
    u_solution = solve_nonlinear_system(u_guess, lambda_val, h)
    num_iterations_lambda.append(newton(bratu_system, u_guess, fprime2=
        bratu_jacobian, args=(lambda_val, h), maxiter=100)[1])

plt.plot(lambda_values, num_iterations_lambda, marker='o')
plt.xlabel('Lambda')
plt.ylabel('Número de Iteracoes')
plt.title('Variacao do Número de Iteracoes com Lambda')
plt.grid()
plt.show()

# Experimento 2: Variando n e mantendo lambda fixo
lambda_val = 2
n_values = [10, 100, 300]
num_iterations_n = []
```

```
54
55 for n in n_values:
56     h = 1 / n
57     x_values = np.linspace(0, 1, n+1)
58     u_guess = np.zeros(n+1)
59     u_solution = solve_nonlinear_system(u_guess, lambda_val, h)
60     num_iterations_n.append(newton(bratu_system, u_guess, fprime2=
           bratu_jacobian, args=(lambda_val, h), maxiter=100)[1])
61
62 plt.plot(n_values, num_iterations_n, marker='o')
63 plt.xlabel('Número de Subdivisoes (n)')
64 plt.ylabel('Número de Iteracoes')
65 plt.title('Variacao do Número de Iteracoes com o Número de Subdivisoes')
66 plt.grid()
67 plt.show()
```

Output:



Neste código, são realizados dois experimentos:

1. Variou-se valor de λ e foi feito o acompanhamento o número de iterações necessário para a convergência;
2. Variou-se o número de subdivisões n e acompanhou-se o número de iterações necessário.

Os gráficos gerados a partir desses experimentos ajudam a responder as questões do exercício. Eles mostram como o método de Newton se comporta conforme λ e n aumentam, e como isso afeta o número de iterações necessárias para a convergência.

APÊNDICE A – Códigos da Atividade

Abaixo são apresentados os códigos realizados, desenvolvidos e testados na plataforma `https://colab.google/`. A seguir, segue o link do ambiente virtual com as questões: `https://colab.research.google.com/drive/1jNQ3a9uci2RbJwEQYbkCBdieNav_eb7n?usp=sharing`

OBS: É importante ressaltar que a função que explicita códigos no Overleaf (*lstlisting*) apresenta erros quando alguns caracteres são inseridos, tais como: **ç**, **â** e **ã**. Então os comentários e *prints* dos códigos no relatório são diferentes do ambiente virtual, lembrando que apenas é feita a troca dos caracteres não reconhecidos.