INE5202 - Cálculo Numérico

Segundo Exercício Programa

Objetivo: implementar o método de Newton para a resolução de sistemas lineares em diferentes aplicações.
Entrega: 09/06/24
Alunos: Jéssica Regina dos Santos e Marcos Vinicius Moreira Machado
Matrículas: 22100626 e 23100478
Professor(a): Priscila Calegari

---

1. Problema: Newton e os fractais

O método de Newton é conhecido por sua capacidade de convergir para soluções de equações não lineares com uma taxa quadrática, desde que a aproximação inicial esteja suficientemente próxima à solução desejada.
Esse método tem sido estudado em relação às suas propriedades iterativas, especialmente em duas dimensões, onde surgem regiões de convergência com fronteiras complexas.
Os fractais surgiram como uma ferramenta para compreender essas estruturas complexas.
Benoit Mandelbrot, nos anos 1960, identificou padrões em formas geométricas irregulares, dando origem ao conceito de conjuntos fractais, que exibem auto-similaridade.
Para analisar os sistemas não lineares, propõe-se discretizar uma região do plano e aplicar o método de Newton a partir de pontos nessa região.

Tarefa 1:
Duas tarefas são sugeridas:

1. Implementar o método de Newton para os sistemas não lineares fornecidos e construir dois tipos de mapas: um mostrando para onde a solução aproximada converge e outro mostrando como a sequência de iterações converge.
2. O primeiro sistema linear é dado por:

$$\begin{cases} x^2 - y^2 - 1 = 0 \\ 2xy = 0 \end{cases}$$

Soluções: $(x_1^*, y_1^*) = (1, 0), (x_2^*, y_2^*) = (-1, 0)$

$$T = \left\{ \left(-1 + \frac{2i}{N}, -1 + \frac{2j}{N} \right) \right\}, \text{com } N \geq 100 \text{ e } 0 \leq i, j \leq N.$$

3. O segundo sistema linear é dado por:

$$\begin{cases} x^3 - 3xy^2 - 1 = 0 \\ 3x^2y - y^3 = 0 \end{cases}$$

Soluções: $(x_1^*, y_1^*) = (1,0), (x_2^*, y_2^*) = \left(-\frac{1}{2}, -\frac{\sqrt{3}}{2}\right), (x_3^*, y_3^*) = \left(-\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$

$T = \left\{\left(-1.5 + \frac{3i}{N}, -1.5 + \frac{3j}{N}\right)\right\}$, com $N \geq 100$ e $0 \leq i, j \leq N$.

T é a região de análise.
As cores são atribuídas às regiões com base na convergência das soluções e no número de iterações realizadas.

Nosso código tem como objetivo gerar mapas de convergência e de iterações para dois sistemas de equações não lineares utilizando o Método de Newton.
A seguir, explicamos detalhadamente cada bloco do código e sua função.

Definição de Variáveis Globais

```
5    tol = 1e-6
6    kmax = 20
```

Estas variáveis globais definem a tolerância (“tol”) para o critério de parada e o número máximo de iterações (“kmax”) para o Método de Newton.

Definição das Funções e Jacobianas dos Sistemas
Sistema 1: função “F1” e sua Jacobiana “J1”.

```
9    def F1(x: np.ndarray):
10       n = len(x)
11       F = np.zeros(n)
12       F[0] = x[0]**2 - x[1]**2 - 1
13       F[1] = 2 * x[0] * x[1]
14       return F
15
16   def J1(x: np.ndarray):
17       n = len(x)
18       J = np.zeros((n, n))
19       J[0][0] = 2 * x[0]  # df0/dx0
20       J[0][1] = -2 * x[1]  # df0/dx1
21       J[1][0] = 2 * x[1]  # df1/dx0
22       J[1][1] = 2 * x[0]  # df1/dx1
23       return J
```

Estas funções definem o sistema de equações não lineares e sua matriz Jacobiana (derivadas de F), que são usadas no Método de Newton.

Sistema 2: função “F2” e sua Jacobiana “J2”.

```python
# Matrizes do Sistema 2
def F2(x: np.ndarray):
    n = len(x)
    F = np.zeros(n)
    F[0] = x[0]**3 - 3*x[0]*(x[1]**2) - 1
    F[1] = 3*(x[0]**2)*x[1] - x[1]**3
    return F

def J2(x: np.ndarray):
    n = len(x)
    J = np.zeros((n, n))
    J[0][0] = 3 * (x[0]**2) - 3 * (x[1]**2)  # df0/dx0
    J[0][1] = -6 * x[0] * x[1]  # df0/dx1
    J[1][0] = 6 * x[0] * x[1]  # df1/dx0
    J[1][1] = 3 * (x[0]**2) - 3 * (x[1]**2)  # df1/dx1
    return J
```

De maneira semelhante ao Sistema 1, estas funções descrevem o segundo sistema de equações e sua matriz Jacobiana.

Implementação do Método de Newton

```python
# Método de Newton para achar solução do sistema
def Newton(x0: np.ndarray, F: callable, J: callable):
    n = len(x0)
    x = np.zeros(n)
    erro = 1  # tamanho do erro
    it = 0  # quantidade de iterações
    Fx = F(x0)
    resmax = max(abs(Fx))
    while (resmax > tol or erro < tol) and it < kmax:
        Jx = J(x0)
        s = np.linalg.solve(Jx, -Fx)
        x = x0 + s
        erro = max(abs(s))
        Fx = F(x)
        resmax = max(abs(Fx))
        x0 = x
        it += 1
    return x, it
```

Essa função aplica o Método de Newton para resolver os sistemas de equações não lineares.
Ela itera até que a solução alcance a tolerância especificada ou atinja o número máximo de iterações.
Em cada iteração, a função calcula o passo “s”, resolvendo o sistema linear “Jx*s = -Fx” e atualiza a estimativa “x”.

Geração dos Mapas de Convergência e Iterações

```python
def gerar_mapas(F, J, solucoes, T, nome_sistema):
    N = T.shape[0]
    mapa_convergencia = np.zeros((N, N))
    mapa_iteracoes = np.zeros((N, N))

    for i in range(N):
        for j in range(N):
            x0 = T[i, j]
            solucao, iteracoes = Newton(x0, F, J)
            erro_min = np.inf
            solucao_index = -1
            for k, s in enumerate(solucoes):
                erro = np.linalg.norm(solucao - s, ord=np.inf)
                if erro < erro_min:
                    erro_min = erro
                    solucao_index = k
            if erro_min < tol:
                mapa_convergencia[i, j] = solucao_index + 1
            else:
                mapa_convergencia[i, j] = 0
            mapa_iteracoes[i, j] = iteracoes
```

```python
# Plotar os mapas de convergência
plt.figure(figsize=(14, 7))

plt.subplot(1, 2, 1)
plt.imshow(mapa_convergencia, extent=(T[0,0,0], T[-1,0,0], T[0,0,1], T[0,-1,1]), origin='lower', cmap='tab10')
plt.colorbar()
plt.title(f'Mapa de Convergência - {nome_sistema}')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(1, 2, 2)
plt.imshow(mapa_iteracoes, extent=(T[0,0,0], T[-1,0,0], T[0,0,1], T[0,-1,1]), origin='lower', cmap='viridis')
plt.colorbar()
plt.title(f'Mapa de Iterações - {nome_sistema}')
plt.xlabel('x')
plt.ylabel('y')

plt.show()
```

Essa função gera os mapas de convergência e de iterações.
Para cada ponto inicial “x0” da grade “T”, ela aplica o Método de Newton e verifica para qual solução o ponto converge, registrando o índice da solução ou “0” se não convergir para nenhuma solução conhecida.
Além disso, registra o número de iterações necessárias para a convergência.

Definição da Grade e Geração dos Mapas

Para o Sistema 1:

```python
# Definir a grade T para o sistema 1
N = 500
x = np.linspace(-1, 1, N)
y = np.linspace(-1, 1, N)
T1 = np.array([[[xi, yi] for yi in y] for xi in x])

# Soluções do sistema 1
solucoes1 = [np.array([1, 0]), np.array([-1, 0])]

# Gerar e plotar os mapas para o sistema 1
gerar_mapas(F1, J1, solucoes1, T1, 'Sistema 1')
```

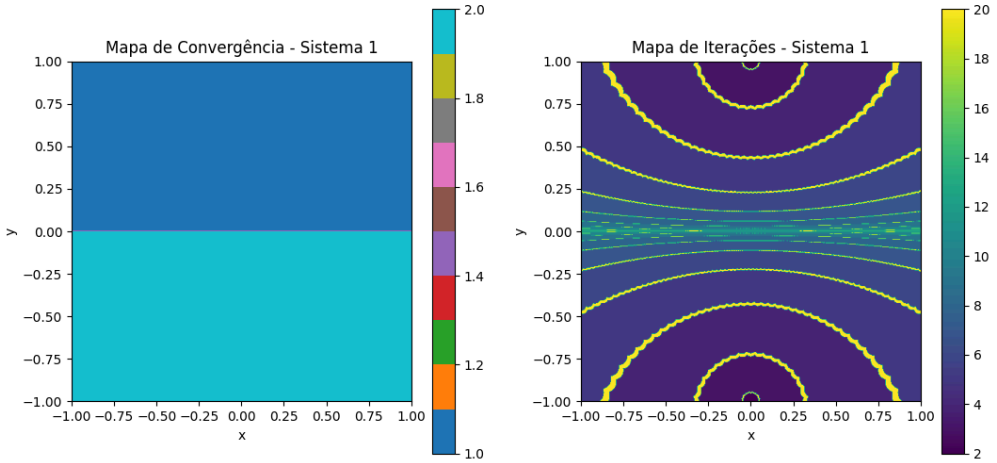
Para o Sistema 2:

```python
# Definir a grade T para o sistema 2
x = np.linspace(-1.5, 1.5, N)
y = np.linspace(-1.5, 1.5, N)
T2 = np.array([[[xi, yi] for yi in y] for xi in x])

# Soluções do sistema 2
solucoes2 = [np.array([1, 0]), np.array([-0.5, -np.sqrt(3)/2]), np.array([-0.5, np.sqrt(3)/2])]

# Gerar e plotar os mapas para o sistema 2
gerar_mapas(F2, J2, solucoes2, T2, 'Sistema 2')
```

Essas gerações definem a grade de pontos iniciais para os dois sistemas e geram os mapas de convergência e de iterações chamando a função “gerar_mapas” com os respectivos parâmetros.

Análise dos Mapas de Convergência e de Iterações

Sistema 1



O Sistema 1 é definido pelas funções:

$F1(x) = x0^2 - x1^2 - 1 \quad e \quad 2x0x1$

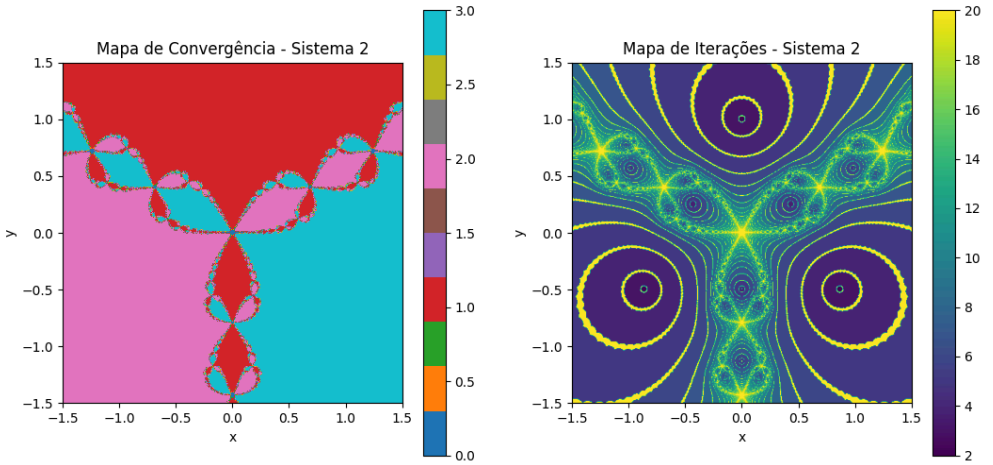
A Jacobiana correspondente é:

$$J1(x) = \begin{bmatrix} 2x0 & -2x1 \\ 2x1 & 2x0 \end{bmatrix}$$

Os mapas gerados para o Sistema 1 apresentam as seguintes características:

- Mapa de Convergência: observamos duas regiões distintas, cada uma convergindo para uma das soluções conhecidas do sistema ([1,0] e [-1,0]). A região superior (em azul escuro) e a inferior (em azul claro) indicam a convergência para [-1,0] e [1,0], respectivamente.
- Mapa de Iterações: o mapa de iterações mostra a quantidade de iterações necessárias para atingir a convergência. Notamos que nas regiões próximas de y = 0, o número de iterações é maior, indicando que pontos iniciais próximos à linha divisória entre as duas soluções necessitam de mais iterações para convergir.

Sistema 2



O Sistema 2 é definido pelas funções:

$F2(x) = x0^3 - 3x0\,x1^2 - 1 \quad e \quad 3x0^2x1 - x1^3$

A Jacobiana correspondente é:

$$J2(x) = \begin{bmatrix} 3x0^2 - 3x1^2 & -6x0x1 \\ 6x0x1 & 3x0^2 - 3x1^2 \end{bmatrix}$$

Os mapas gerados para o Sistema 2 apresentam as seguintes características:

- Mapa de Convergência: este mapa revela um padrão de fractal complexo, indicando regiões de convergência para diferentes soluções conhecidas do sistema ([1,0], [-0.5, -sqrt(3)/2] e [-0.5, sqrt(3)/2]). As cores diferentes indicam as regiões de convergência para cada uma dessas soluções. A presença de padrões fractais indica uma sensibilidade elevada aos pontos iniciais, onde pequenas mudanças podem levar a convergências para soluções diferentes.
- Mapa de Iterações: similar ao mapa do Sistema 1, este mapa indica o número de iterações necessárias para atingir a convergência. As regiões com padrões fractais também apresentam variações no número de iterações, com algumas áreas necessitando de mais iterações para convergir, especialmente nas fronteiras entre as regiões de diferentes soluções.

Conclusões sobre os mapas: os mapas de convergência e de iterações fornecem uma visualização clara do comportamento das soluções dos sistemas não lineares estudados. Para o Sistema 1, as soluções são claramente definidas com regiões bem demarcadas, enquanto para o Sistema 2, a complexidade dos padrões fractais destaca a sensibilidade aos pontos iniciais e a dificuldade potencial em prever a convergência.
A análise dos mapas de iterações revela que pontos próximos às fronteiras entre regiões de convergência diferentes geralmente requerem mais iterações, indicando a complexidade e o comportamento caótico perto dessas fronteiras.

2. Problema: Aproximação de funções

O Método dos Mínimos Quadrados (MMQ) é um método de análise e estimação, usado na aproximação de funções, consistindo de tentar minimizar o resíduo da função aproximadora g(x), descrito na forma:

$$R(a) = \frac{1}{2} \Sigma (r_i)^2, \text{ sendo } r_i = y_i - g(x_i)$$

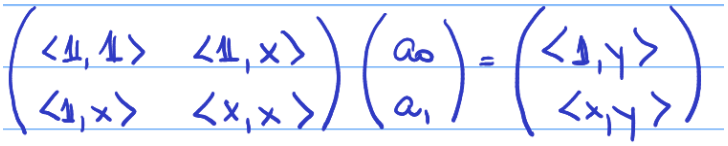
O Algoritmo de Gauss-Newton, baseado nesse método, pode ser usado para encontrar funções aproximadoras para dados tabelados, através do cálculo das derivadas parciais da função resíduo.

Tarefa 2:
A segunda tarefa do Exercício Programa se divide em 3 itens. Sendo fornecido dados relacionados às coordenadas de um cometa, cuja órbita é descrita pela Lei de Kepler, descrita no exercício por $r = a_0/(1-a_1*\cos(\theta))$, a tarefa consiste de:

A. Transformar a função em linear em seus parâmetros.
B. Encontrar a_0 e a_1 para essa função linear.
C. Usar o método de Gauss-Newton para achar uma função aproximadora, na forma fornecida originalmente, e comparar os parâmetros, resíduo e gráficos com a forma linearizada.

Aproximação de uma função linear

Para acharmos a função aproximadora linear, usamos do método visto em aula (com origem no MMQ) de formar um sistema linear com produtos escalares, de forma que a solução do sistema sejam os parâmetros a_0 e a_1 (anotações fornecidas pela professora abaixo):



Para isso, criamos uma função `prod_esc()`, usado para calcular o produto escalar de 2 vetores, e a função principal `aprox_linear()`, que cria o sistema linear $M \cdot a=b$ e o resolve, através da função `numpy.linalg.solve()`.

```python
# calcula o produto escalar de 2 vetores
def prod_esc(x: np.ndarray, y: np.ndarray):
    soma = 0
    n = len(x)
    for i in range(n):
        soma += x[i]*y[i]
    return soma

# realiza aproximação de uma função linear
def aprox_linear(x: np.ndarray, y: np.ndarray):
    n = len(x)
    ones = np.ones(n)
    M = np.zeros((2,2))
    b = np.zeros(2)
    a = np.zeros(2)
    M[0][0] = prod_esc(ones, ones)
    M[0][1] = prod_esc(ones, x)
    M[1][0] = prod_esc(x, ones)
    M[1][1] = prod_esc(x, x)
    b[0] = prod_esc(ones, y)
    b[1] = prod_esc(x, y)
    a = np.linalg.solve(M, b)
    return a
```

Achando uma função aproximadora por Gauss-Newton

Para acharmos a outra função aproximadora, criamos 2 funções auxiliares que criam matrizes: `mR()`, responsável por criar a matriz-coluna r, contendo os resíduos, e a função `J()`, responsável por criar a matriz Jacobiana, contendo as derivadas parciais dos resíduos em relação a a_0 e a_1.

```python
def mR(a: np.ndarray, x: np.ndarray, y: np.ndarray):
    n = len(x)
    mR = np.zeros(n)
    for i in range(n):
        mR[i] = y[i] - g(a, x[i])
    return mR

# cria matriz J com derivadas parciais
def J(x: np.ndarray, a: np.ndarray):
    n = len(x)
    Jac = np.zeros((n,2))
    for i in range(n):
        Jac[i][0] = 2*(g(a, x[i]))*(-1/( -a[1]*np.cos(x[i]) + 1))
        Jac[i][1] = 2*(g(a, x[i]))*(-a[0]*np.cos(x[i])/(-a[1]*np.cos(x[i]) + 1)**2)
    return Jac
```

Depois disso, a função principal dessa parte da tarefa, `Gauss_Newton()`, realiza o algoritmo fornecido no Exercício Programa. Até satisfazer a tolerância de erro desejada ou atingir o número máximo de iterações, o programa avalia o valor da Jacobiana, encontra a solução "s" do sistema $J^T \cdot J \cdot s = J^T \cdot r$, e a subtrai do vetor a (que contêm a_0 e a_1).

```python
# Método de Gauss-Newton
def Gauss_Newton(x: np.ndarray, y: np.ndarray, a0: np.ndarray, tol: float):
    itmax = 200
    it = 1
    erro = 1
    n = len(a0)
    a = np.zeros(n)
    while erro > tol and it < itmax:
        r = mR(a0, x, y)
        Ja = J(x, a0)
        Jt = np.transpose(Ja)
        s = np.linalg.solve(np.matmul(Jt,Ja),np.matmul(Jt, r))
        a = a0 - s
        erro = max(abs(s))
        a0 = a
        it += 1
    return a
```

Comparando os resultados: parâmetros e resíduos

A fim de comparar os resultados, foram criadas as funções auxiliares `g()`, que retorna o valor da função $r = a_0/(1-a_1 \cdot \cos(\theta))$ (função aproximadora de Gauss-Newton), a função `gl()`, que retorna $r = a_0 + a_1 \cdot x$ (função aproximadora linear), e a função `R()`, que retorna o resíduo da função g ou gl, dependendo dos parâmetros passados.

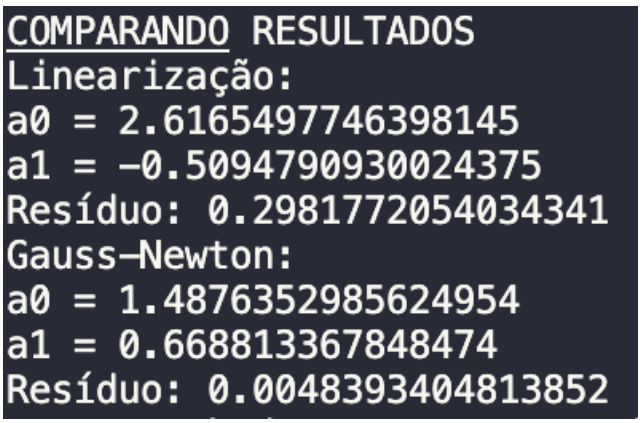
```python
def g(a: np.ndarray, x: int):
    return a[0]/(1 - a[1]*np.cos(x))
```

```python
# g = a0 + a1*x
def gl(a: np.ndarray, x: int):
    return a[0] + a[1]*x
```

```python
# calcula o resíduo, tanto para a função linear quanto a de Gauss-Newton
def R(a: np.ndarray, x: np.ndarray, y: np.ndarray, g: callable):
    soma = 0
    n = len(y)
    for i in range(n):
        r = y[i] - g(a, x[i])
        soma += r**2
    return (soma/2)
```

Já a função `compara_resultados()` imprime na tela os valores de a_0 e a_1 para cada função, além de calcular e imprimir o resíduo de ambas. Dessa forma, pudemos observar como a função de Gauss-Newton gerou resultados mais precisos e obteve um menor resíduo.

```python
# função que compara resultados das 2 aproximações
def comparar_resultados(a1: np.ndarray, a2: np.ndarray, x: np.ndarray, y: np.ndarray):
    print("COMPARANDO RESULTADOS")
    print("Linearização:")
    print(f"a0 = {a1[0]}")
    print(f"a1 = {a1[1]}")
    print(f"Resíduo: {R(a1, x, y, gl)}")
    print("Gauss-Newton:")
    print(f"a0 = {a2[0]}")
    print(f"a1 = {a2[1]}")
    print(f"Resíduo: {R(a2, x, y, g)}")
```



Comparando os resultados

A comparação de resultados entre as duas abordagens (linearização e método de Gauss-Newton) é interessante, porque eles representam diferentes estratégias para encontrar os parâmetros que melhor se ajustam aos dados fornecidos.

Aqui estão algumas observações sobre a comparação:

Parâmetros estimados (a0 e a1): na linearização, os parâmetros estimados são aproximadamente a0 = 2.62 e a1 = -0.51. No método de Gauss-Newton, os parâmetros estimados são aproximadamente a0 = 1.49 e a1 = 0.67.

Resíduo: o resíduo é uma medida de quão bem o modelo se ajusta aos dados, logo, quanto menor o resíduo, melhor é o ajuste. No caso da linearização, o resíduo é aproximadamente 0.30. No método de Gauss-Newton, o resíduo é muito menor, 0.0048.

A diferença nos parâmetros estimados indica que os modelos estão capturando diferentes aspectos dos dados.

O resíduo muito menor no método de Gauss-Newton sugere que esse modelo se ajusta muito melhor aos dados do que a abordagem de linearização. Portanto, o método de Gauss-Newton parece fornecer uma melhor aproximação para o conjunto de dados fornecidos.

É importante notar que o método de Gauss-Seidel pode ser mais complexo computacionalmente, já que envolve iterações através do cálculo de derivadas parciais e solução de sistemas lineares em cada iteração. No entanto, neste caso específico, a melhoria na qualidade do ajuste justifica esse custo adicional.

Em resumo: a comparação mostra que o método de Gauss-Newton produz um ajuste melhor aos dados em comparação com a abordagem de linearização, conforme evidenciado pela diferença nos parâmetros estimados e, especialmente, nos resíduos.