Matemática Numérica Avançada

Pedro H A Konzen

17 de outubro de 2023

Licença

Prefácio

Nestas notas de aula são abordados métodos numéricos aplicados a problemas de grande porte. Como ferramenta computacional de apoio, exemplos de aplicação de códigos Python, são apresentados, mais especificamente, códigos com suporte das bibliotecas NumPy e SciPy.

Agradeço a todos e todas que de modo assíduo ou esporádico contribuem com correções, sugestões e críticas. :)

Pedro H A Konzen

Conteúdo

Capa i

Licença ii

Prefácio iii

Sumário v

1 Sistemas Lineares 1
- 1.1 Matrizes Esparsas . . . 1
  - 1.1.1 Sistemas Tridiagonais . . . 3
  - 1.1.2 Matrizes Banda . . . 7
  - 1.1.3 Esquemas de Armazenamento . . . 13
- 1.2 Métodos Iterativos . . . 17
  - 1.2.1 GMRES . . . 17
  - 1.2.2 Método do Gradiente Conjugado . . . 23
  - 1.2.3 Precondicionamento . . . 29

2 Sistemas Não Lineares e Otimização 32
- 2.1 Método de Newton . . . 32
- 2.2 Método Tipo Newton . . . 35
  - 2.2.1 Atualização Cíclica da Matriz Jacobiana . . . 35
- 2.3 Problemas de Minimização . . . 37
  - 2.3.1 Métodos de declive . . . 38
  - 2.3.2 Método do Gradiente . . . 39
  - 2.3.3 Método de Newton . . . 42
  - 2.3.4 Método do Gradiente Conjugado . . . 45

3 Autovalores e Autovetores **48**
3.1 Método da Potência 48
3.1.1 Autovalor dominante 48
3.1.2 Método da Potência Inverso 51
3.2 Iteração QR 52

4 Integração **56**
4.1 Integração Autoadaptativa 56
4.2 Integrais múltiplas 59
4.2.1 Regras de Newton-Cotes 59
4.2.2 Regras Compostas de Newton-Cotes 62

Referências Bibliográficas **67**

Capítulo 1

Sistemas Lineares

[Vídeo] | [Áudio] | [Contatar]

Neste capítulo, apresentam-se métodos numéricos para a resolução de sistemas lineares de grande porte. Salvo explicitado ao contrário, assume-se que os sistemas são quadrados e têm solução única.

1.1 Matrizes Esparsas

[Vídeo] | [Áudio] | [Contatar]

Uma matriz é dita ser **esparsa** quando ela tem apenas poucos elementos não nulos. A ideia é que os elementos não nulos não precisam ser guardados na memória do computador, gerando um grande benefício na redução da demanda de armazenamento de dados. O desafio está no desenvolvimento de estruturas de dados para a alocação eficiente de tais matrizes, i.e. que sejam suficientemente adequadas para os métodos numéricos conhecidos.

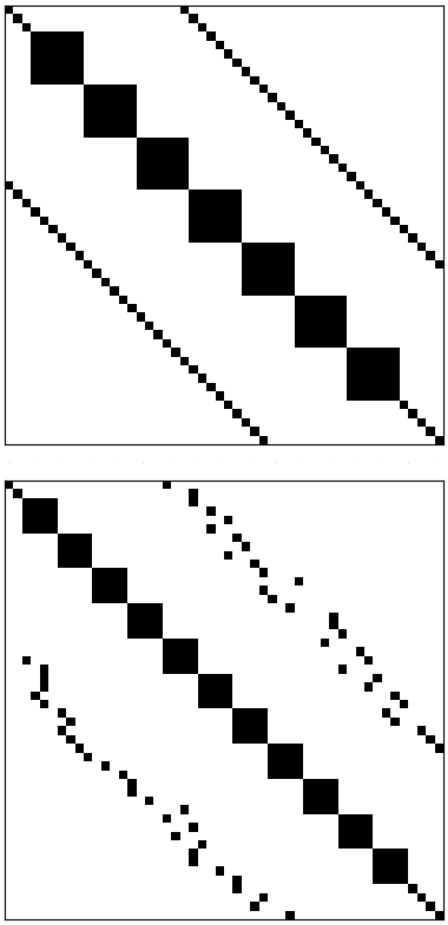

Figura 1.1: Em cima: exemplo de uma matriz esparsa estruturada. Em baixo: exemplo de uma matriz esparsa não-estruturada.

Matrizes esparsas podem ser classificadas como **estruturadas** ou **não-estruturadas**. Uma matriz estruturada é aquela em que as entradas não-nulas formam um padrão regular. Por exemplo, estão dispostas em poucas diagonais ou formam blocos (submatrizes densas) ao longo de sua diagonal principal. No caso de não haver um padrão regular das entradas não-nulas, a matriz esparsa é dita ser não-estruturada. Consulte a Figura 1.1 para exemplos.

A **esparsidade** de uma matriz é a porcentagem de elementos nulos que

ela tem, i.e. para uma matriz quadrada $n \times n$ tem-se que a esparsidade é

$$\frac{n_{\text{nulos}}}{n^2} \times 100\% \tag{1.1}$$

Por exemplo, a matriz identidade de tamanho $n = 100$ tem esparsidade

$$\frac{100^2 - 100}{100^2} \times 100\% = 99\% \tag{1.2}$$

1.1.1 Sistemas Tridiagonais

Um sistema tridiagonal tem a seguinte forma matricial

$$\begin{bmatrix} a_{1,1} & a_{1,2} & & & & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & & & \\ & a_{3,2} & a_{3,3} & a_{3,4} & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & a_{n-1,n} \\ 0 & & & & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{1.3}$$

Ou seja, é um sistema cuja a matriz dos coeficientes é tridiagonal.

Uma **matriz tridiagonal** é uma matriz esparsa estruturada. Mais especificamente, é um caso particular de uma matriz banda, em que os elementos não nulos estão apenas em algumas de suas diagonais. Para armazenarmos tal matriz precisamos alocar apenas os seguintes três vetores

$$d_1 = (0, a_{2,1}, \ldots, a_{n-1,n}) \tag{1.4}$$
$$d_0 = (a_{1,1}, a_{2,2}, \ldots, a_{n,n}) \tag{1.5}$$
$$d_{-1} = (a_{2,1}, \ldots, a_{n,n-1}, 0) \tag{1.6}$$

Ou seja, precisamos armazenar $3n$ pontos flutuantes em vez de n^2, como seria o caso se a matriz dos coeficientes fosse densa. Com isso, podemos alocar a matriz do sistema da seguinte forma

$$\tilde{A} = \begin{bmatrix} * & a_{2,1} & \cdots & a_{n-1,n} \\ a_{1,1} & a_{2,2} & \cdots & a_{n,n} \\ a_{2,1} & \ldots & a_{n,n-1} & * \end{bmatrix} \tag{1.7}$$

Ou seja, $\tilde{A} = [\tilde{a}_{i,j}]_{i,j=1}^{3,n}$, sendo

$$\tilde{a}_{1+i-j,j} = a_{i,j} \tag{1.8}$$

Exemplo 1.1.1. Seja o seguinte sistema linear

$$2x_1 - x_2 = 0 \tag{1.9}$$

$$x_{i-1} - 6x_i + 4x_{i+1} = \text{sen}\left(i\frac{\pi}{2(n-1)}\right) \tag{1.10}$$

$$x_{n-1} + x_n = 1 \tag{1.11}$$

O seguinte código Python faz a alocação de seu vetor dos termos constantes b e de sua matriz de coeficientes no formato compacto de $\tilde{A}$.

```
1  import numpy as np
2  n = 100000
3
4  # alocação
5  # vetor dos termos constantes
6  b = np.empty(n)
7  b[0] = 0.
8  for i in range(1,n-1):
9    b[i] = np.sin(i*np.pi/(2*(n-1)))
10 b[n-1] = 1.
11 print(b)
12 print(f"b size: {b.size*b.itemsize/1024} Kbytes")
13
14 # matriz compacta
15 tA = np.zeros((3,n))
16
17 # indexação
18 def ind(i,j):
19 return 1+i-j,j
20
21 tA[ind(0,0)] = 2.
22 tA[ind(0,1)] = -1.
23 for i in range(1,n-1):
24   tA[ind(i,i-1)] = 1.
25   tA[ind(i,i)] = -3.
26   tA[ind(i,i+1)] = 4.
27 tA[ind(n-1,n-2)] = 1.
28 tA[ind(n-1,n-1)] = 1.
29 print(tA)
30 print(f"tA size: {tA.size*tA.itemsize/1024**2:1.1f} Mbytes")
```

Algoritmo de Thomas (TDMA)

O Algoritmo de Thomas[1] (ou, TDMA, *Tridiagonal Matrix Algorithm*) é uma forma otimizada do Método de Eliminação Gaussiana[2] aplicada à sistemas tridiagonais. Enquanto este requer $O(n^3)$ operações, esse demanda apenas $O(n)$.

Eliminando os termos abaixo da diagonal em (1.3), obtemos o sistema equivalente

$$\begin{bmatrix} a_{1,1} & a_{1,2} & & & 0 \\ & \tilde{a}_{2,2} & a_{2,3} & & \\ & & \tilde{a}_{3,3} & \ddots & \\ & & \ddots & \ddots & a_{n-1,n} \\ 0 & & & a_{n,n-1} & \tilde{a}_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_{n-1} \\ \tilde{b}_n \end{bmatrix} \tag{1.12}$$

Este é obtido pela seguinte iteração

$$w := \frac{a_{i+1,i}}{a_{i,i}} \tag{1.13}$$

$$a_{i,i} := a_{i,i} - w a_{i-1,i} \tag{1.14}$$

$$b_i := b_i - w b_{i-1} \tag{1.15}$$

onde, o $\tilde{}$ foi esquecido de propósito, indicando a reutilização da matriz $\tilde{A}$ e do vetor b. A solução do sistema é, então, obtida de baixo para cima, i.e.

$$x_n = \frac{b_n}{a_{n,n}} \tag{1.16}$$

$$x_i = \frac{b_i - a_{i,i+1} x_{i+1}}{a_{i,i}}, \tag{1.17}$$

com $i = n-1, n-2, \ldots, 1$.

Código 1.1: TDMA

```
1 def tdma(ta, b):
2   a = ta.copy()
```

[1] Llewellyn Hilleth Thomas, 1903 - 1992, físico e matemático aplicado britânico. Fonte: Wikipedia.

[2] Carl Friedrich Gauss, 1777-1855, matemático alemão. Fonte: Wikipédia.

```
3     x = b.copy()
4     # eliminação
5     for i in range(1,n):
6       w = a[2,i-1]/a[1,i-1]
7       a[1,i] -= w * a[0,i]
8       x[i] -= w * x[i-1]
9     # resolve
10    x[n-1] = x[n-1]/a[1,n-1]
11    for i in range(n-2,-1,-1):
12      x[i] = (x[i] - a[0,i+1]*x[i+1])/a[1,i]
13    return x
```

Exercício 1.1.1. Considere o seguinte sistema linear

$$2x_1 - x_2 = 0 \tag{1.18}$$

$$x_{i-1} - 6x_i + 4x_{i+1} = \operatorname{sen}\left(i\frac{\pi}{2(n-1)}\right) \tag{1.19}$$

$$x_{n-1} + x_n = 1 \tag{1.20}$$

a) Compute sua solução usando o Algoritmo de Thomas para $n = 3$.

b) Compare a solução obtida no item anterior com a gerada pela função `scipy.linalg.solve`.

c) Compare a solução com a obtida no item anterior com a gerada pela função `scipy.linalg.solve_banded`.

d) Use o módulo Python datetime para comprar a demanda de tempo computacional de cada um dos métodos acima. Compute para $n = 10{,}100{,}1000{,}10000$.

Exercício 1.1.2. Considere que o problema de valor de contorno (PVC)

$$-u'' = \operatorname{sen}\pi x, \quad 0 < x < 1, \tag{1.21}$$

$$u(0) = 0, \tag{1.22}$$

$$u(1) = 0 \tag{1.23}$$

seja simulado com o Método das Diferenças Finitas[3]. Vamos assumir uma

[3]Consulte mais em Notas de Aula: Matemática Numérica.

discretização espacial uniforme com n nodos e tamanho de malha

$$h = \frac{1}{n-1}. \tag{1.24}$$

Com isso, temos os nodos $x_i = (i-1)h$, $i = 1,2,\ldots,n$. Nos nodos internos, aplicamos a fórmula de diferenças central

$$u''(x_i) \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}, \tag{1.25}$$

onde, $u_i \approx u(x_i)$. Com isso, a discretização da EDO fornece

$$-\frac{1}{h^2}u_{i-1} + \frac{2}{h^2}u_i - \frac{1}{h^2}u_{i+1} = \operatorname{sen} \pi x_i \tag{1.26}$$

para $i = 2,3,\ldots,n-1$. Das condições de contorno temos $u_1 = u_n = 0$. Logo, o problema discreto lê-se: encontrar $u = (u_1,u_2,\ldots,u_n) \in \mathbb{R}^n$ tal que

$$u_1 = 0 \tag{1.27}$$

$$-\frac{1}{h^2}u_{i-1} + \frac{2}{h^2}u_i - \frac{1}{h^2}u_{i+1} = \operatorname{sen} \pi x_i \tag{1.28}$$

$$u_n = 0 \tag{1.29}$$

a) Calcule a solução analítica do PVC.

b) Use a função `scipy.linalg.solve_banded` para computar a solução do problema discreto associado para diferentes tamanhos de malha $h = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$. Compute o erro da solução discreta em relação à solução analítica.

c) Compare a demanda de tempo computacional se a função `scipy.linalg.solve` for empregada na computação da solução discreta.

1.1.2 Matrizes Banda

Uma matriz banda é aquela em que os elementos não nulos estão dispostos em apenas algumas de suas diagonais. Consulte a Figura 1.2.

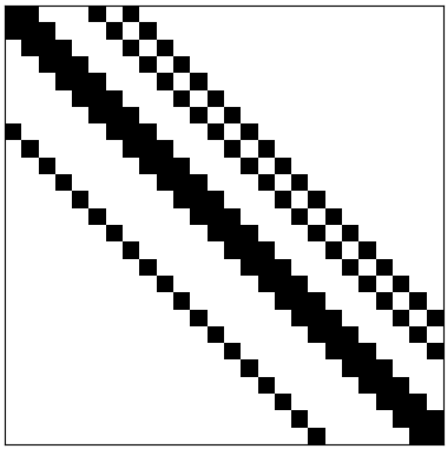

Figura 1.2: Exemplo de uma matriz banda.

Exemplo 1.1.2. Consideremos o seguinte problema de Poisson[4]

$$-\Delta u = f(x,y),\ (x,y) \in (0,\pi) \times (0,\pi), \tag{1.30}$$
$$u(0,y) = 0,\ y \in [0,\pi], \tag{1.31}$$
$$u(\pi,y) = 0,\ y \in [0,\pi], \tag{1.32}$$
$$u(x,0) = 0,\ x \in [0,\pi], \tag{1.33}$$
$$u(x,\pi) = 0,\ x \in [0,\pi]. \tag{1.34}$$

Para fixarmos as ideias, vamos assumir

$$f(x,y) = \operatorname{sen}(x)\operatorname{sen}(y) \tag{1.35}$$

Vamos empregar o Método de Diferenças Finitas[5] para computar uma aproximação para a sua solução. Começamos assumindo uma malha uniforme de n^2 nodos

$$x_i = (i-1)h \tag{1.36}$$
$$y_j = (j-1)h \tag{1.37}$$

[4]Baron Siméon Denis Poisson, 1781 - 1840, matemático, engenheiro e físico francês. Fonte: Wikipedia.

[5]Observamos que $\Delta u = u_{xx} + u_{yy}$.

com tamanho de malha $h = \pi/(n-1)$, $i = 1,2,\ldots,n$ e $j = 1,2,\ldots,n$. Empregando a Fórmula de Diferenças Central[6] encontramos o seguinte problema discreto associado

$$u_{i,1} = u_{1,j} = 0 \tag{1.38}$$

$$\begin{aligned} &-\frac{1}{h^2}u_{i-1,j} - \frac{1}{h^2}u_{i,j-1} + \frac{4}{h^2}u_{i,j} \\ &-\frac{1}{h^2}u_{i+1,j} - \frac{1}{h^2}u_{i,j+1} = f(x_i, y_j) \end{aligned} \tag{1.39}$$

$$u_{i,n} = u_{n,j} = 0 \tag{1.40}$$

Este é um sistema linear $n^2 \times n^2$. Tomando em conta as condições de contorno, ele pode ser reduzido a um sistema $(n-2)^2 \times (n-2)^2$

$$Aw = b \tag{1.41}$$

usando a enumeração das incógnitas $(i,j) \to k = i - 1 + (j-2)(n-2)$, i.e.

$$u_{i,j} = w_{k=i-1+(j-2)(n-2)}, \quad i,j = 2,\ldots,n-2 \tag{1.42}$$

Consulte a Figura 1.3 para uma representação da enumeração em relação a malha.

[6] Consulte mais em Notas de Aula: Matemática Numérica.

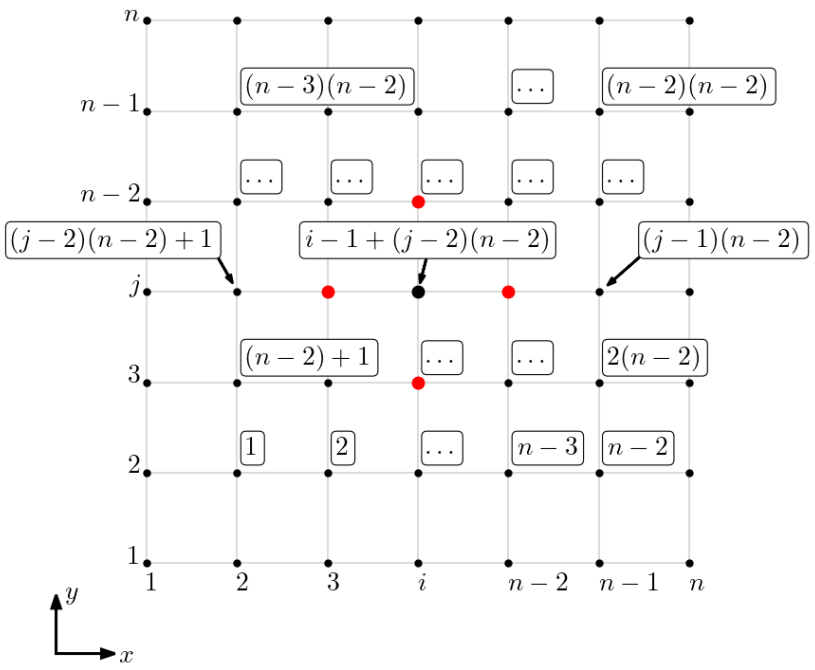

Figura 1.3: Representação da enumeração das incógnitas referente ao problema discutido no Exercício 1.1.2.

Afim de obtermos uma matriz diagonal dominante, vamos ordenar as equações do sistema discreto como segue

- $j = 2$, $i = 2$:

$$4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.43}$$

- $j = 2$, $i = 3, \ldots, n-2$:

$$-w_{k-1} + 4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.44}$$

- $j = 2$, $i = n-1$:

$$-w_{k-1} + 4w_k - w_{k+n-2} = h^2 f_{i,j} \tag{1.45}$$

- $j = 3, \ldots, n-2$, $i = 2$:

$$-w_{k-(n-2)} + 4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.46}$$

- $j = 3, \ldots, n-2$, $i = 3, \ldots, n-2$:

$$-w_{k-1} - w_{k-(n-2)} + 4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.47}$$

- $j = 3, \ldots, n-2$, $i = n-1$:

$$-w_{k-1} - w_{k-(n-2)} + 4w_k - w_{k+n-2} = h^2 f_{i,j} \tag{1.48}$$

- $j = n-1$, $i = 2$:

$$-w_{k-(n-2)} + 4w_k - w_{k+1} = h^2 f_{i,j} \tag{1.49}$$

- $j = n-1$, $i = 3, \ldots, n-2$:

$$-w_{k-1} - w_{k-(n-2)} + 4w_k - w_{k+1} = h^2 f_{i,j} \tag{1.50}$$

- $j = n-1$, $i = n-1$:

$$-w_{k-1} - w_{k-(n-2)} + 4w_k = h^2 f_{i,j} \tag{1.51}$$

Com isso, obtemos uma matriz com 5 bandas, consulte a Figura 1.4.

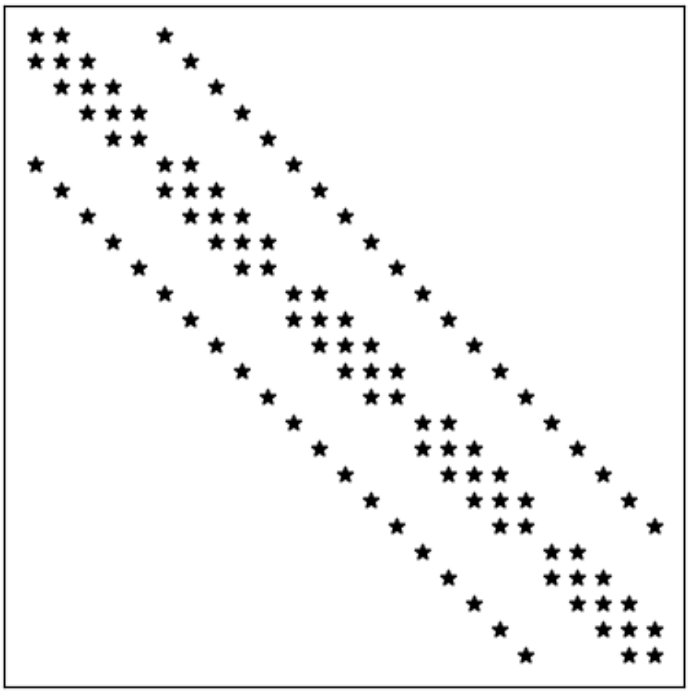

Figura 1.4: Representação da matriz do sistema discreto construído no Exemplo 1.1.2.

Exercício 1.1.3. Consideremos o problema trabalho no Exemplo 1.1.2.

a) Use a função `scipy.linalg.solve` para computar a solução do problema discreto associado para diferentes tamanhos de malha $h = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$. Compute o erro da solução discreta em relação à solução analítica. Compare as aproximações com a solução analítica

$$u(x,y) = \frac{1}{2}\,\mathrm{sen}(x)\,\mathrm{sen}(y). \tag{1.52}$$

b) Compare a demanda de tempo e memória computacional se a função `scipy.linalg.solve_banded` for empregada na computação da solução discreta.

c) Baseado no Algoritmo de Thomas, implemente o Método de Eliminação Gaussiana otimizado para a matriz banda deste problema. Compare com

as abordagens dos itens a) e b).

1.1.3 Esquemas de Armazenamento

A ideia é armazenar apenas os elementos não-nulos de uma matriz esparsa, de forma a economizar a demanda de armazenamento computacional. Cuidados devem ser tomados para que a estrutura de armazenamento utilizada seja adequada para a computação das operações matriciais mais comuns.

Formato COO

O formato COO (*COOrdinate format*) é o esquema de armazenamento mais simples. As estrutura de dados consiste em três arranjos: (1) um arranjo contendo as entradas não-nulas da matriz; (2) um arranjo contendo seus índices de linha; (3) um arranjo contendo seus índices de coluna.

Exemplo 1.1.3. Consideremos a seguinte matriz

$$A = \begin{bmatrix} 2. & 0. & 1. & 0. \\ 0. & 3. & 2. & -1. \\ 0 & -1 & -2. & 0. \\ 0 & 0 & 0 & 1. \end{bmatrix} \tag{1.53}$$

O formato COO está disponível na biblioteca SciPy com o método scipy.sparse.coo_matrix[7]. Neste caso, temos

```
1  import numpy as np
2  import scipy as sp
3  from scipy.sparse import coo_matrix
4
5  data = np.array([2.,1.,3.,2.,-1.,-1.,-2.,1.])
6  row = np.array([0,0,1,1,1,2,2,3])
7  col = np.array([0,2,1,2,3,1,2,3])
8  coo = coo_matrix((data, (row, col)), shape=(4,4))
9  print("coo = \n", coo)
10 print("A = \n", coo.toarray())
```

- Vantagens do formato COO são:

[7] Versões recentes do SciPy estão migrando para a nomenclatura do NumPy. Consulte mais em SciPy Sparse.

  - permite a entrada de dados duplicados (simplicidade);
  - possível conversão rápida entre os formatos CSR e CSC[8].
- Desavantagens do formato COO são:
  - complexidade em operações aritméticas;
  - complexidade na extração de submatrizes.

Observação 1.1.1. O SciPy conta com vários métodos para o tratamento e operação com matrizes esparsas armazenadas no formato COO. Consulte mais em scipy.sparse.coo_matrix.

Formato CSR

O formato *Compressed Sparse Row* (CSR) é uma variação do COO que busca diminuir a alocação de dados repetidos. Assim como o COO, o formato conta com três arranjos d, c, p:

- d é o arranjo contendo os elementos não-nulos da matriz, ordenados por linhas (i.e., da esquerda para direita, de cima para baixo);
- c é o arranjo contendo o índice das colunas das entradas não-nulas da matriz (como no formato COO);
- p é um arranjo cujos elementos são a posição no arranjo c em que cada linha da matriz começa a ser representada. O número de elementos de i-ésima linha da matriz dado por $p_{j+1} - p_j$.

Exemplo 1.1.4. No Exemplo 1.1.3, alocamos a matriz

$$A = \begin{bmatrix} 2. & 0. & 1. & 0. \\ 0. & 3. & 2. & -1. \\ 0 & -1 & -2. & 0. \\ 0 & 0 & 0 & 1. \end{bmatrix} \tag{1.54}$$

no formato COO. Aqui, vamos converter a alocação para o formato CSR e, então, verificar seus atributos.

```
1 from scipy.sparse import csr_matrix
```

[8] Estes formatos são mais eficientes para a computação matricial e são apresentados na sequência.

```
csr = coo.tocsr()
d = csr.data
c = csr.indices
p = csr.indptr
print(d)
print(c)
print(p)
```

Para fixarmos as ideias, temos

$$d = (2., 1., 3., 2., -1., -1., -2., 1.) \tag{1.55}$$
$$c = (0, 2, 1, 2, 3, 1, 2, 3) \tag{1.56}$$
$$p = (0, 2, 5, 7, 8) \tag{1.57}$$

Assim sendo, o elemento `p[i=2] = 5` aponta para o `c[k=5] = 1`, o que fornece que `A[i=2,j=1]=d_{k}=-1.`. Verifique!

- Vantagens do formato CSR:
  - operações aritméticas eficientes;
  - fatiamento por linhas eficiente;
  - multiplicação matriz vetor eficiente.
- Desvantagens do formato CSR:
  - fatiamento por colunas não eficiente;
  - custo elevado de realocamento com alteração da esparsidade da matriz.

Observação 1.1.2. O SciPy conta com vários métodos para o tratamento e operação com matrizes esparsas armazenadas no formato CSR. Consulte mais em scipy.sparse.csr_matrix.

Formato CSC

O formato *Compressed Sparse Column* (CSC) é uma variação análoga do CSR, mas para armazenamento por colunas. O formato conta com três arranjos d, l, p:

- d é o arranjo contendo os elementos não-nulos da matriz, ordenados por colunas (i.e., de cima para baixo, da esquerda para direita);

- l é o arranjo contendo o índice das linhas das entradas não-nulas da matriz;
- p é um arranjo cujos elementos são a posição no arranjo l em que cada coluna da matriz começa a ser representada. O número de elementos de j-ésima coluna da matriz dado por $p_{j+1} - p_j$.

Exemplo 1.1.5. No Exemplo 1.1.3, alocamos a matriz

$$A = \begin{bmatrix} 2. & 0. & 1. & 0. \\ 0. & 3. & 2. & -1. \\ 0 & -1 & -2. & 0. \\ 0 & 0 & 0 & 1. \end{bmatrix} \tag{1.58}$$

no formato COO. Aqui, vamos converter a alocação para o formato CSC e, então, verificar seus atributos.

```
1 from scipy.sparse import csc_matrix
2 csc = coo.tocsc()
3 d = csc.data
4 l = csc.indices
5 p = csc.indptr
6 print(d)
7 print(l)
8 print(p)
```

Para fixarmos as ideias, temos

$$d = (2., 3., -1., 1., 2., -2., -1., 1.) \tag{1.59}$$
$$l = (0, 1, 2, 0, 1, 2, 1, 3) \tag{1.60}$$
$$p = (01368) \tag{1.61}$$

Assim sendo, o elemento `p[j=2] = 3` aponta para o `l[k=3] = 0`, o que informa que `A[i=0,j=2]=d_{k}=1.`. Verifique!

- Vantagens do formato CSC:
  - fatiamento por colunas eficiente;
  - operações aritméticas eficientes;
  - multiplicação matriz vetor eficiente[9].

[9] CSR é mais eficiente em muitos casos.

- Desvantagens do formato CSC:
  - fatiamento por linhas não eficiente;
  - custo elevado de realocamento com alteração da esparsidade da matriz.

Observação 1.1.3. O SciPy conta com vários métodos para o tratamento e operação com matrizes esparsas armazenadas no formato CSC. Consulte mais em scipy.sparse.csc_matrix.

Observação 1.1.4. Além dos formatos COO, CSR e CSC, exitem ainda vários outros que podem empregados e que são mais eficientes em determinadas aplicações. Recomendamos a leitura de [7, Seção 3.4] e da documentação do scipy.sparse.

Exercício 1.1.4. Considere o problema de Poisson dado no Exemplo 1.1.2.

a) Armazene a matriz do problema discreto associado usando o formato COO.

b) Converta a matriz armazenada para o formato CSR[10]. Então, compute a solução do problema discreto com o método spsolve[11].

c) Converta a matriz armazenada para o formato CSC[12]. Então, compute a solução do problema discreto com o método spsolve.

d) Compare a eficiência da computação entre os itens b) e c) para tamanhos de malha $h = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$.

1.2 Métodos Iterativos

1.2.1 GMRES

O GMRES (do inglês, *Generalized Minimal Residual Method*[13]) é um Método de Subespaço de Krylov e é considerado uma das mais eficientes técnicas

[10] Use o método coo_matrix.tocsr().
[11] `scipy.sparse.linalg.spsolve` é uma implementação do Método LU otimizado para matrizes esparsas.
[12] Use o método coo_matrix.tocsc().
[13] Desenvolvido por Yousef Saad e H. Schultz, 1986. Fonte: Wikipedia.

para a resolução de sistemas lineares gerais e de grande porte (esparsos).

Método de Subespaço de Krylov

A ideia básica é resolver o sistema linear

$$Ax = b \tag{1.62}$$

por um **método de projeção**. Mais especificamente, busca-se uma solução aproximada $x_m \in \mathbb{R}^{\times}$ no subespaço afim $x_0 + \mathcal{K}_m$ de dimensão m, impondo-se a **condição de Petrov[14]-Galerkin[15]**

$$b - Ax_m \perp \mathcal{L}_m, \tag{1.63}$$

onde $\mathcal{L}_m$ também é um subespaço de dimensão m. Quando $\mathcal{K}_m$ é um **subespaço de Krylov[16]**, i.e.

$$\mathcal{K}_m(A,r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\}, \tag{1.64}$$

temos o Método de Subespaço de Krylov. Aqui, temos o **resíduo**

$$r_0 = b - Ax_0, \tag{1.65}$$

sendo x_0 uma aproximação inicial para a solução do sistema. Notemos que com isso, temos que a aproximação calculada é tal que

$$A^{-1}b \approx x_m = x_0 + q_{m-1}(A)r_0, \tag{1.66}$$

onde q_{m-1} é um dado polinômio de grau $m-1$. No caso particular de $x_0 = 0$, temos

$$A^{-1}b \approx q_{m-1}(A)x_0. \tag{1.67}$$

Diferentes versões deste método são obtidas pelas escolhas do subespaço $\mathcal{L}_m$ e formas de precondicionamento do sistema.

[14] Georgi Iwanowitsch Petrov, 1912 - 1987, engenheiro soviético. Fonte: Wikipedia.
[15] Boris Galerkin, 1871 - 1945, engenheiro e matemático soviético. Fonte: Wikipédia.
[16] Alexei Nikolajewitsch Krylov, 1863 - 1945, engenheiro e matemático russo. Fonte: Wikipédia.

GMRES

O GMRES é um Método de Subespaço de Krylov assumindo $\mathcal{L}_m = A\mathcal{K}_m$, com

$$\mathcal{K}_m = \mathcal{K}_m(A,v_1) = \text{span}\{v_1, Av_1, \ldots, A^{m-1}v_1\}, \tag{1.68}$$

onde $v_1 = r_0/\|r_0\|$ é o vetor unitário do resíduo $r_0 = b - Ax_0$ para uma dada aproximação inicial x_0 da solução do sistema $Ax = b$.

Vamos derivar o método observando que qualquer vetor x em $x_0 + \mathcal{K}_m$ pode ser escrito como segue

$$x = x_0 + V_m y \tag{1.69}$$

onde, $V_m = [v_1, \ldots, v_m]$ é a matriz $n \times m$ cujas colunas formam uma base ortogonal $\{v_1, \ldots, v_m\}$ de $\mathcal{K}_m$ e $y \in R^m$. Aqui, V_m é computada usando-se o seguinte **Método de Arnoldi[17]- Gram[18]-Schmidt[19] Modificado** [7, Subseção 6.3]:

1. Dado v_1 de norma 1
2. Para $j = 1, \ldots, m$:
   (a) $w_j := Av_j$
   (b) Para $i = 1, \ldots, j$:
      i. $h_{i,j} := (w_j, v_i)$
      ii. $w_j := w_j - h_{i,j}v_i$
   (c) $h_{j+1,j} := \|w_j\|$
   (d) Se $h_{j+1,j} = 0$, então pare.
   (e) $v_{j+1} = w_j/h_{j+1,j}$

Seja, então, $\bar{H}_m = [h_{i,j}]_{i,j=1}^{m+1,m}$ a matriz de Hessenberg[20] cujas entradas não nulas são computadas pelo algoritmo acima (Passos 2(a)i-ii). Pode-se

[17] Walter Edwin Arnoldi, 1917 - 1995, engenheiro americano estadunidense. Fonte: Wikipédia.
[18] Jørgen Pedersen Gram, 1850 - 1916, matemático dinamarquês. Fonte: Wikipédia.
[19] Erhard Schmidt, 1876 - 1959, matemático alemão. Fonte: Wikipédia.
[20] Karl Adolf Hessenberg, 1904 - 1959, engenheiro e matemático alemão. Fonte: Wikipédia.

mostrar[21] que

$$J(y) = \|b - Ax\| \tag{1.70}$$
$$= \|b - A(x_0 + V_m y)\| \tag{1.71}$$
$$= \|\beta e_1 - \bar{H}_m y\| \tag{1.72}$$

onde, $\beta = \|r_0\|$.

A aproximação GMRES x_m é então computada como

$$x_m = x_0 + V_m y_m, \tag{1.73}$$
$$y_m = \min_y \|\beta e_1 - \bar{H}_m y\| \tag{1.74}$$

Observamos que este último é um pequeno problema de minimização, sendo que requer a solução de um sistema $(m+1) \times m$ de mínimos quadrados, sendo m normalmente pequeno.

Em resumo, a solução GMRES x_m é computada seguindo os seguintes passos:

1. Escolhemos uma aproximação inicial x_0 para a solução de $Ax = b$.
2. Calculamos o resíduo $r_0 = b - Ax_0$.
3. Calculamos o vetor unitário $v_1 = r_0/\|r_0\|$.
4. Usamos o Método de Arnoldi-Gram-Schmidt Modificado para calculamos uma base ortogonal V_m de $\mathcal{K}_m$ e a matriz de Hessenberg $\bar{H}_m$ associada.
5. Calculamos $y_m = \min_y \|\beta e_1 - \hat{H}_m y\|$.
6. Calculamos $x_m = x_0 + V_m y$.

Observação 1.2.1 (Convergência)**.** Pode-se mostrar que o GMRES converge em ao menos n passos.

Observação 1.2.2 (GMRES com a ortogonalização de Householder)**.** No algoritmo acima, o Método Modificado de Gram-Schmidt é utilizado no processo de Arnoldi. Uma versão numericamente mais eficiente é obtida quando a **Transformação de Householder**[22] é utilizada. Consulte mais em [7, Subsetion 6.5.2].

[21] Consulte [7, Proposição 6.5].

[22] Alston Scott Householder, 1904 - 1993, matemático americano estadunidense. Fonte: Wikipédia.

Observação 1.2.3 (GMRES com Reinicialização)**.** O *Restarted GMRES* é uma variação do método para sistemas que requerem uma aproximação GMRES x_m com m grande. Nestes casos, o método original pode demandar um custo muito alto de memória computacional. A ideia consiste em assumir m pequeno e, caso não suficiente, recalcular a aproximação GMRES com $x_0 = x_m$. Este algoritmo pode ser descrito como segue.

1. Computamos $r_0 = b - Ax_0$, $\beta = \|r_0\|$ e $v_1 = r_0/\beta$
2. Computamos V_m e $\hat{H}_m$ pelo método de Arnoldi
3. Computamos

$$y_m = \min_y \|\beta e_1 - \hat{H}_m y\| \tag{1.75}$$

$$x_m = x_0 + V_m y_m \tag{1.76}$$

4. Se $\|b - Ax_m\|$ é satisfatória, paramos. Caso contrário, setamos $x_0 := x_m$ e voltamos ao passo 1.

A convergência do *Restarted GMRES* não é garantida para matrizes que não sejam positiva-definidas.

Exercício 1.2.1. Considere o problema discreto do Exercício 1.1.2.

a) Compute a solução com a implementação *Restarted GMRES*

scipy.sparse.linalg.gmres.

b) Por padrão, o intervalo de iterações entre as inicializações é `restart=20`. Compare o desempenho para diferentes intervalos de reinicialização.

c) Compare o desempenho entre as abordagens dos ítens a) e b) frente a implementação do Método de Eliminação Gaussiana disponível em

scypi.sparse.linalg.spsolve.

Exercício 1.2.2. Considere o problema discreto trabalhado no Exemplo 1.1.2.

a) Compute a solução com a implementação *Restarted GMRES*

scipy.sparse.linalg.gmres.

b) Por padrão, o intervalo de iterações entre as inicializações é `restart=20`. Compare o desempenho para diferentes intervalos de reinicialização.

c) Compare o desempenho entre as abordagens dos ítens a) e b) frente a implementação do Método de Eliminação Gaussiana disponível em

scypi.sparse.linalg.spsolve.

Exercício 1.2.3. Consideremos o seguinte problema de Poisson[23] com condições de contorno não homogêneas.

$$-\Delta u = f(x,y),\ (x,y) \in D, \tag{1.77}$$
$$u = g, \quad \text{em } \partial D \tag{1.78}$$

Para fixarmos as ideias, vamos assumir o domínio $D = (0,1) \times (0,1)$, a fonte

$$f(x,y) = 2\pi^2 \operatorname{sen} \pi(x+y) \tag{1.79}$$

e os valores no contorno

$$g = \operatorname{sen} \pi(x+y), \quad (x,y) \in \partial D. \tag{1.80}$$

Observamos que a solução analítica deste problema é

$$u(x,y) = \operatorname{sen} \pi(x+y). \tag{1.81}$$

Vamos empregar o Método de Diferenças Finitas[24] para computar uma aproximação para a solução. Assumimos uma malha uniforme de n^2 nodos

$$x_i = (i-1)h \tag{1.82}$$
$$y_j = (j-1)h \tag{1.83}$$

com tamanho de malha $h = 1/(n-1)$, $i = 1,2,\ldots,n$ e $j = 1,2,\ldots,n$. Empregando a Fórmula de Diferenças Central[25] encontramos o seguinte problema discreto associado

$$u_{i,1} = g(x_i, 0) \tag{1.84}$$

[23] Baron Siméon Denis Poisson, 1781 - 1840, matemático, engenheiro e físico francês. Fonte: Wikipedia.
[24] Observamos que $\Delta u = u_{xx} + u_{yy}$.
[25] Consulte mais em Notas de Aula: Matemática Numérica.

$$u_{1,j} = g(0, y_j) \tag{1.85}$$

$$\begin{aligned} &-\frac{1}{h^2}u_{i-1,j} - \frac{1}{h^2}u_{i,j-1} + \frac{4}{h^2}u_{i,j} \\ &-\frac{1}{h^2}u_{i+1,j} - \frac{1}{h^2}u_{i,j+1} = f(x_i, y_j) \end{aligned} \tag{1.86}$$

$$u_{i,n} = g(x_i, 1) \tag{1.87}$$

$$u_{n,j} = g(1, y_j) \tag{1.88}$$

Este pode ser escrito na forma matricial

$$Aw = b \tag{1.89}$$

onde, A é $(n-2)**2 \times (n-2)**2$ e assumindo a enumeração

$$u_{i,j} = w_{k=i-1+(j-2)(n-2)}, \quad i,j = 2, \ldots, n-2. \tag{1.90}$$

Consulte a Figura 1.4.

1. Compute a solução do problema discreto associado usando a seguinte implementação Python do GMRES

   scipy.sparse.linalg.gmres

2. Compare o desempenho com a aplicação do Método LU implemento em

   scipy.sparse.linalg.spsolve

Exercício 1.2.4. Faça sua própria implementação do método GMRES. Valide-a e compare-a com a resolução do exercício anterior (Exercício 1.2.3).

1.2.2 Método do Gradiente Conjugado

O Método do Gradiente Conjugado é uma das mais eficientes técnicas iterativas para a resolução de sistema linear com matriz esparsa, simétrica e definida positiva. Portanto, vamos assumir que o sistema

$$Ax = b \tag{1.91}$$

onde, a A é **simétrica** e **definida positiva**.

O método pode ser derivado a partir do Método de Arnoldi[26] [7, Seção 6.7] ou como uma variação do Método do Gradiente. Este é caminho que será adotado aqui.

Método do Gradiente

A ideia é reformular o sistema $Ax = b$ como um problema de minimização. Vamos começar definindo o funcional

$$J(y) = \frac{1}{2} y^T A y - y^T b. \tag{1.92}$$

O vetor y que minimiza J é a solução de $Ax = b$. De fato, denotando x a solução de $Ax = b$, temos

$$J(y) = \frac{1}{2} y^T A y - y^T b + \frac{1}{2} x^T A x - \frac{1}{2} x^T A x \tag{1.93}$$

$$= \frac{1}{2} (y - x)^T A (y - x) - \frac{1}{2} x^T A x \tag{1.94}$$

O termo é independente de y e, portanto, J é mínimo quando

$$\frac{1}{2} (y - x)^T A (y - x) \tag{1.95}$$

é minimizado. Agora, como A é definida positiva[27], o menor valor deste termo ocorre quando $y - x = 0$, i.e. $y = x$.

Observamos, também, que o gradiente de J é

$$\nabla J = Ay - b \tag{1.96}$$

i.e., é o oposto do resíduo $r = b - Ay$. Com isso, temos que $y = x$ é a única escolha tal que $\nabla J = 0$. Ainda, temos que ∇J é o vetor que aponta na direção e sentido de maior crescimento de J. Isso nos motiva a aplicarmos a seguinte iteração[28]

$$x^{(0)} = \text{aprox. inicial} \tag{1.97}$$

[26] Walter Edwin Arnoldi, 1917 - 1995, engenheiro americano estadunidense. Fonte: Wikipédia.

[27] $x^T A x > 0$ para todo $x \neq 0$.

[28] Iteração do Método do Máximo Declive.

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla J\left(x^{(k)}\right) \tag{1.98}$$

onde, $\alpha_k > 0$ é um escalar que regula o tamanho do passo a cada iteração. Lembrando que $-\nabla J = r$, temos que a iteração é equivalente a

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)}. \tag{1.99}$$

Notamos que $x^{(k+1)}$ é um ponto na reta $\left\{x^{(k)} + \alpha r^{(k)} : \alpha \in \mathbb{R}\right\}$ que tem a mesma direção de $\nabla J\left(x^{(k)}\right)$ e passa pelo ponto $x^{(k)}$. O procedimento de escolher um $\alpha^{(k)}$ entre todos os possíveis, é conhecido como pesquisa linear (em inglês, *line search*).

A cada iteração, queremos escolher α_k de forma que $J\left(x^{(k+1)}\right) \leq J\left(x^{(k)}\right)$. Isso pode ser garantido fazendo a seguinte escolha[29]

$$J\left(x^{(k+1)}\right) = \min_{\alpha \in \mathbb{R}} J\left(x^{(k)} + \alpha r^{(k)}\right) \tag{1.100}$$

A fim de resolver este problema de minimização, vamos denotar

$$g(\alpha) = J\left(x^{(k)} + \alpha r^{(k)}\right). \tag{1.101}$$

Então, observamos que

$$\begin{aligned} g(\alpha) &= \frac{1}{2}\left(x^{(k)} + \alpha r^{(k)}\right)^T A\left(x^{(k)} + \alpha r^{(k)}\right) - \left(x^{(k)} + \alpha r^{(k)}\right)^T b \\ &= \frac{1}{2} x^{(k)^T} A x^{(k)} + \frac{\alpha}{2} x^{(k)^T} A r^{(k)} + \frac{\alpha}{2} r^{(k)^T} A x^{(k)} \\ &+ \frac{\alpha^2}{2} r^{(k)^T} A r^{(k)} - x^{(k)^T} b - \alpha r^{(k)^T} b \end{aligned} \tag{1.102}$$

Agora, usando o fato de A ser simétrica, obtemos

$$g(\alpha) = J\left(x^{(k)}\right) + \alpha r^{(k)^T} A x^{(k)} + \frac{\alpha^2}{2} r^{(k)^T} A r^{(k)} - \alpha r^{(k)^T} b \tag{1.103}$$

$$= J\left(x^{(k)}\right) - \alpha r^{(k)^T} r^{(k)} + \frac{\alpha^2}{2} r^{(k)^T} A r^{(k)} \tag{1.104}$$

a qual, é uma função quadrática. Seu único mínimo, ocorre quando

$$0 = g'(\alpha) \tag{1.105}$$

[29] Chamada de pesquisa linear exata. Qualquer outra escolha para α é conhecida como pesquisa linear não exata.

$$= -r^{(k)^T} r^{(k)} + \alpha r^{(k)^T} b. \tag{1.106}$$

Logo, encontramos

$$\alpha = \frac{r^{(k)^T} r^{(k)}}{r^{(k)^T} A r^{(k)}} \tag{1.107}$$

Com isso, temos a iteração do Método do Gradiente

$$x^{(0)} = \text{aprox. inicial} \tag{1.108}$$

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)}, \tag{1.109}$$

$$\alpha_k = \frac{r^{(k)^T} r^{(k)}}{r^{(k)^T} A r^{(k)}} \tag{1.110}$$

Observação 1.2.4. Observamos que, a cada iteração, precisamos computar $Ar^{(k)}$ (no cálculo de α_k) e $Ax^{(k)}$ (no cálculo do resíduo). Essas multiplicações matriz-vetor são os passos computacionais mais custosos do método. Podemos otimizar isso usando o fato de que

$$r^{(k+1)} = r^{(k)} - \alpha_k A r^{(k)}. \tag{1.111}$$

Exercício 1.2.5. Faça sua implementação do Método do Gradiente.

Exercício 1.2.6. Use a implementação feita no exercício anterior (Exercício 1.2.6) nos seguintes itens.

a) Compute a solução do problema discreto do Exemplo 1.1.2 pelo Método do Gradiente. Quantas iterações são necessárias para obter um resíduo com norma $\leq 10^{-14}$?

b) Compute a solução do problema discreto do Exercício 1.2.3 pelo Método do Gradiente. Quantas iterações são necessárias para obter um resíduo com norma $\leq 10^{-14}$?

c) Compare a aplicação do Método GMRES[30] e do Método LU[31] nos itens anteriores.

[30] scipy.sparse.linalg.gmres
[31] scipy.sparse.linalg.spsolve

Exercício 1.2.7. Considere o Exercício 1.2.3.

a) Use sua implementação do Método do Gradiente para computar uma solução aproximada, cuja norma do resíduo $\leq 10^{-14}$.

b) Compare o desempenho com a aplicação da implementação GMRES

scipy.sparse.linalg.gmres

Método do Gradiente Conjugado

O Método do Gradiente consiste em uma iteração da forma

$$x_0 = \text{aprox. inicial}, x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}, \tag{1.112}$$

com $p^{(k)} = r^{(k)}$. Ou seja, a nova aproximação $x^{(k+1)}$ é buscada na direção de $p^{(k)}$. Aqui, a ideia é usar uma melhor direção para buscar a solução.

O Método do Gradiente Conjugado é um Método de Gradiente que busca encontrar a solução de $Ax = b$ pela computação do mínimo do seguinte funcional[32]

$$J(y) = \frac{1}{2}\langle y,y\rangle_A - \langle b,y\rangle, \tag{1.113}$$

onde, $\langle\cdot,\cdot\rangle$ denota o produto interno padrão e

$$\langle x,y\rangle_A := x^T Ay \tag{1.114}$$

é o **produto interno induzido por** A, lembrando que A é positiva definida[33]. Associada a este produto interno, temos a norma

$$\|x\|_A := \sqrt{\langle x,x\rangle_A}, \tag{1.115}$$

chamada de **norma da energia**. O produto interno associado é também conhecido como **produto interno da energia**. Com isso, definimos que dois vetores x e y são **conjugados**, quando eles são ortogonais com respeito ao produto interno da energia, i.e. quando

$$\langle x,y\rangle_A = 0. \tag{1.116}$$

[32] Compare com o funcional J dado em (1.92).
[33] Mostre que $\langle\cdot,\cdot\rangle_A$ é de fato um produto interno.

Aqui, a ideia é desenvolver um método iterativo em que o erro a cada passo seja conjugado a todas as direções de busca anteriores. Consulte o desenvolvimento detalhado do método em [9, Seção 7.7].

Código 1.2: Algoritmo do Gradiente Conjugado

```
1  import numpy as np
2  import scipy as sp
3  from scipy.linalg import norm
4
5  def mgc(A, b, x, tol=1e-14):
6    n, = b.shape
7    r = b - A*x
8    p = r
9    nu = np.dot(r,r)
10   for it in np.arange(n):
11     q = A*p
12     mu = np.dot(p,q)
13     alpha = nu/mu
14     x = x + alpha*p
15     r = r - alpha*q
16     nu0 = np.dot(r,r)
17     beta = nu0/nu
18     p = r + beta*p
19     nu = nu0
20     if (norm(r) < tol):
21       print(it)
22       return x
23   raise ValueError("Falha de convergencia.")
```

Exercício 1.2.8. Use a implementação acima (Listing 1.2) na resolução dos seguintes itens.

1. Compute a solução do problema discreto do Exemplo 1.1.2 pelo Método do Gradiente Conjugado. Quantas iterações são necessárias para obter um resíduo com norma $\leq 10^{-14}$?

2. Compute a solução do problema discreto do Exercício 1.2.3 pelo Método do Gradiente Conjugado. Quantas iterações são necessárias para obter um resíduo com norma $\leq 10^{-14}$?

3. Compare a aplicação do Método GMRES[34] e da implementação SciPy do Método do Gradiente Conjugado[35]

Exercício 1.2.9. Considere o Exercício 1.2.3.

a) Use sua implementação do Método do Gradiente Conjugado acima (Listing 1.2) para computar uma solução aproximada, cuja norma do resíduo $\leq 10^{-14}$.

b) Compare o desempenho com a aplicação de sua implementação do Método do Gradiente (Exercício 1.2.6).

c) Compare o desempenho com a aplicação da implementação GMRES

scipy.sparse.linalg.gmres

1.2.3 Precondicionamento

Precondicionamento refere-se a modificar o sistema linear original de forma que a computação de sua solução possa ser feita de forma mais eficiente. No lugar do sistema original

$$Ax = b \tag{1.117}$$

resolvemos o sistema equivalente

$$MAx = Mb, \tag{1.118}$$

onde $M = P^{-1}$ e a matriz P é chamada de **precondicionador** do sistema. De forma geral, a escolha do precondicionador é tal que $P \approx A$, mas com inversa fácil de ser computada. Além disso, uma característica esperada é que MA tenha esparsidade parecida com A.

Precondicionamento ILU

A ideia é tomar P igual a uma fatoração LU incompleta (ILU, do inglês, *Incomplete LU*). Incompleta no sentido que entradas de L e de U sejam adequadamente removidas, buscando-se uma boa esparsidade e ao mesmo tempo uma boa aproximação LU para A.

[34] scipy.sparse.linalg.gmres
[35] scipy.sparse.linalg.cg

ILU(0)

O precondicionamento ILU(0) impõe que as matrizes L e U tenham o mesmo padrão de esparsidade da matriz A.

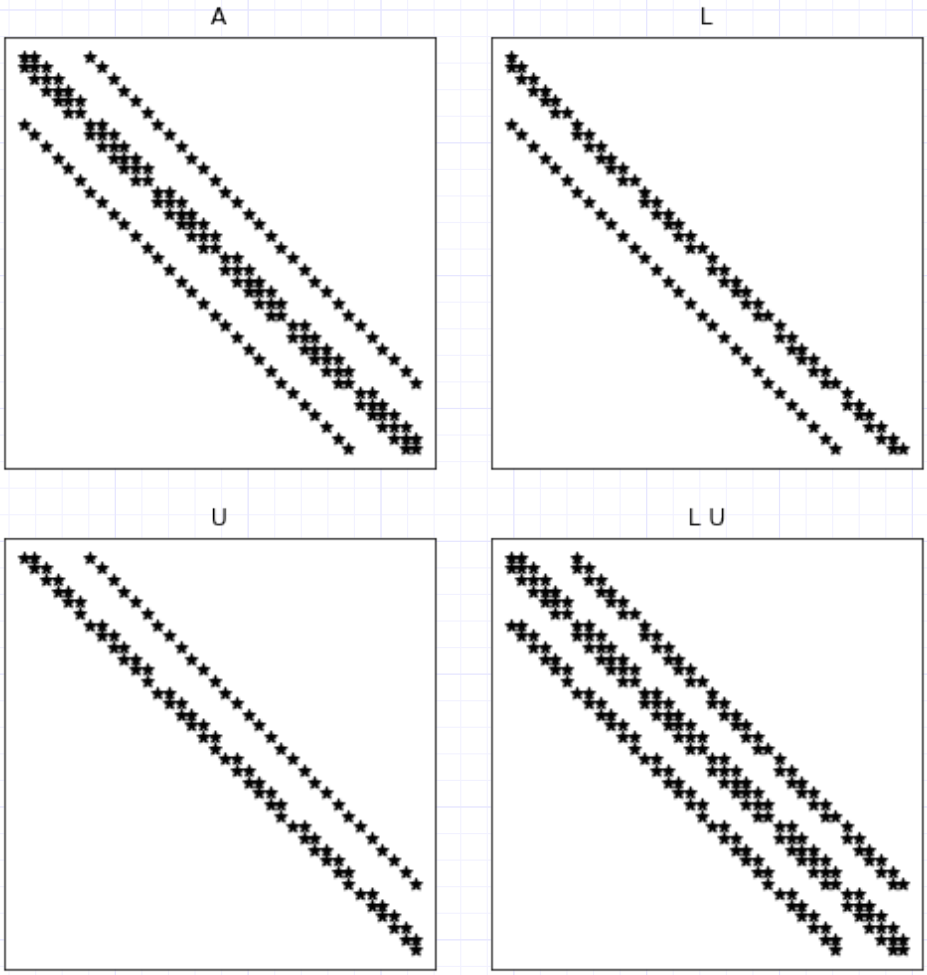

Figura 1.5: Representação das matrizes ILU(0).

Exemplo 1.2.1. ex:possoinDnhIlu0 Consideramos o sistema linear $Ax = b$ associado ao problema discreto trabalhado no Exercício 1.2.3. Para uma malha $n \times n = 8 \times 8$, obtemos as matrizes representadas na Figura 1.5.

Observamos que a matriz LU contém duas diagonais com elementos não

nulos a mais que a matriz original A. Estes elementos são chamados de *fill-in*.

Código 1.3: Algoritmo ILU(0)

```
1  import scipy.sparse as sp
2
3  def ilu0(A):
4    n,n = A.get_shape()
5    LU = A.copy()
6    nz = A.nonzero()
7    for i in range(1,n):
8      for k in [_ for _ in range(i) if (i,_) in zip(nz[0],nz[1])]:
9        LU[i,k] = LU[i,k]/LU[k,k]
10         LU[i,j] = LU[i,j] - LU[i,k]*LU[k,j]
11
12   return LU
```

Exercício 1.2.10. Considere o problema discreto do Exercício 1.2.3.

a) Compute a solução com o método GMRES com precondicionamento ILU(0).

b) Compare com a resolução com o método GMRES sem precondicionamento.

c) Compare com a resolução com o método CG sem precondicionamento.

d) O precondicionamento ILU(0) é eficiente para o método CG?

Capítulo 2

Sistemas Não Lineares e Otimização

[Vídeo] | [Áudio] | [Contatar]

Neste capítulo, apresentam-se métodos numéricos para a resolução de problemas de otimização. Salvo explicitado ao contrário, assume-se que os problemas são bem definidos.

2.1 Método de Newton

Consideremos o seguinte problema: $F : D \subset \mathbb{R}^n \to \mathbb{R}^n$ encontrar $x^* \in \mathbb{R}^n$ tal que

$$F(x^*) = 0. \tag{2.1}$$

Salvo explicitado ao contrário, assumiremos que $F \in C^1(D)$, i.e. F é uma função continuamente diferenciável. Vamos, também, denotar por $J_F(x) = [\jmath_{i,j}(x)]_{i,j=1}^{n,n}$ a **matriz jacobiana**[1] com

$$\jmath_{i,j}(x) = \frac{\partial f_i(x)}{\partial x_j}, \tag{2.2}$$

onde $F(x) = (f_1(x), f_2(x), \ldots, f_n(x))$ e $x = (x_1, x_2, \ldots, x_n)$.

[1] Carl Gustav Jakob Jacobi, 1804 - 1851, matemático alemão. Fonte: Wikipédia.

A iteração básica do **Método de Newton**[2] consiste em: dada uma aproximação inicial $x^{(0)} \in \mathbb{R}^n$,

$$\text{resolver} \quad J_F\left(x^{(k)}\right)\delta^{(k)} = -F\left(x^{(k)}\right) \tag{2.3}$$

$$\text{calcular} \quad x^{(k+1)} = x^{(k)} + \delta^{(k)} \tag{2.4}$$

para $k = 0,1,2,\ldots$ até convergência $x^{(k)} \to x^*$.

Observação 2.1.1. Para $x^{(0)}$ suficientemente próximo da solução x^*, o Método de Newton é quadraticamente convergente. Mais precisamente, este resultado de convergência local requer que $J_F^{-1}(x^*)$ seja não singular e que $J_F : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ seja Lipschitz[4] contínua. Consulte [6, Seção 7.1] para mais detalhes.

Exemplo 2.1.1. Consideremos a **Equação de Burgers**[5]

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{2.5}$$

com $\nu > 0$, condição inicial

$$u(0,x) = \sin(\pi x) \tag{2.6}$$

e condições de contorno de Dirichlet[6] homogêneas

$$u(t,0) = u(t,1) = 0. \tag{2.7}$$

Aplicando o **Método de Rothe**[7] com aproximação de Euler[8] implícita, obtemos

$$\frac{u(t+h_t,x) - u(t,x)}{h_t} + u(t+h_t,x)u_x(t+h_t,x) \approx \nu u_{xx}(t+h_t,x), \tag{2.8}$$

onde $h_t > 0$ é o passo no tempo. Agora, aplicamos diferenças finitas para obter

$$\frac{u(t+h_t,x_i) - u(t,x_i)}{h_t} + u(t+h_t,x_i)\frac{u(t+h_t,x_{i+1}) - u(t+h_t,x_i)}{h_x} \tag{2.9}$$

[2][3]

[4]Rudolf Otto Sigismund Lipschitz, 1832 - 1903, matemático alemão. Fonte: Wikipédia.
[5]Johannes Martinus Burgers, 1895 - 1981, físico holandês. Fonte: Wikipedia.
[6]Johann Peter Gustav Lejeune Dirichlet, 1805 - 1859, matemático alemão. Fonte: Wikipédia.
[7]Erich Hans Rothe, 1895 - 1988, matemático alemão. Fonte: Wikipédia.
[8]Leonhard Paul Euler, 1707 - 1783, matemático e físico suíço. Fonte: Wikipédia.

$$\approx \nu \frac{u(t+h_t,x_{i-1}) - 2u(t+h_t,x_i) + u(t+h_t,x_{i+1})}{h_x^2}, \tag{2.10}$$

onde, $x_i = (i-1)h_x$, $i = 1,\ldots,n_x$ e $h_x = 1/(n_x-1)$ é o tamanho da malha.

Rearranjando os termos e denotando $w_i^{(k)} \approx u(t_k,x_i)$, $t_k = (k-1)h$, obtemos o seguinte **problema discreto**

$$w_1^{(k+1)} = 0 \tag{2.11}$$

$$\frac{1}{h_t}w_i^{(k+1)} - \frac{1}{h_t}w_i^{(k)} + \frac{1}{h_x}w_i^{(k+1)}w_{i+1}^{(k+1)} - \frac{1}{h_x}w_i^{(k+1)}w_i^{(k+1)} \tag{2.12}$$

$$-\frac{\nu}{h_x^2}w_{i-1}^{(k+1)} + 2\frac{\nu}{h_x^2}w_i^{(k+1)} - \frac{\nu}{h_x}w_{i+1}^{(k+1)} = 0, \tag{2.13}$$

$$w_{n_x}^{(k+1)} = 0, \tag{2.14}$$

sendo $w_i^{(0)} = \text{sen}(\pi x_i)$, $i = 1,\ldots,n_x$ e $k = 1,2,\ldots$.

Este problema pode ser reescrito como segue: para cada $k = 1,2,\ldots$, encontrar $v \in \mathbb{R}^{n_x}$, tal que

$$F(v;v^{(0)}) = 0, \tag{2.15}$$

onde $v_i \approx w_i^{(k+1)}$, $v_i^{(0)} \approx w_i^{(k)}$ e

$$f_1(v,v^{(0)}) = v_1, \tag{2.16}$$

$$f_i(v;v^{(0)}) = \frac{1}{h_t}v_i - \frac{1}{h_t}v_i^{(0)} + \frac{1}{h_x}v_iv_{i+1} - \frac{1}{h_x}v_iv_i \tag{2.17}$$

$$-\frac{\nu}{h_x^2}v_{i-1} + 2\frac{\nu}{h_x^2}v_i - \frac{\nu}{h_x}v_{i+1}, \tag{2.18}$$

$$f_{n_x}(v;v^{(0)}) = v_{n_x}. \tag{2.19}$$

A matriz jacobiana associada $J = [\jmath_{i,j}]_{i,j}^{n_x,n_x}$ contém

$$\jmath_{i,j} = 0, \quad j \neq i-1,i,i+1, \tag{2.20}$$

$$\jmath_{1,1} = 1, \tag{2.21}$$

$$\jmath_{1,2} = 0, \tag{2.22}$$

$$\jmath_{i,i-1} = -\frac{\nu}{h_x^2}, \tag{2.23}$$

$$\jmath_{i,i} = \frac{1}{h_t} + \frac{1}{h_x} v_{i+1} - \frac{2}{h_x} v_i + \frac{2\nu}{h_x^2}, \tag{2.24}$$

$$\jmath_{i,i+1} = \frac{1}{h_x} v_i - \frac{\nu}{h_x^2}, \tag{2.25}$$

$$\jmath_{n_x,n_x-1} = 0 \tag{2.26}$$

$$\jmath_{n_x,n_x} = 1. \tag{2.27}$$

Exercício 2.1.1. Considere o problema discreto apresentado no Exemplo 2.1.1 para diferentes valores do coeficiente de difusão $\nu = 1., 0.5, 0.1, 0.01, 0.001$. Simule o problema com cada uma das seguintes estratégias e as compare quanto ao desempenho computacional.

a) Simule-o aplicando o Método de Newton com o *solver* linear `npla.solve`.

b) Observe que a jacobiana é uma matriz tridiagonal. Simule o problema aplicando o Método de Newton com o *solver* linear `npla.solve_banded`.

c) Aloque a jacobiana como uma matriz esparsa. Então, simule o problema aplicando o Método de Newton com *solver* linear adequado para matrizes esparsas.

2.2 Método Tipo Newton

Existem várias modificações do Método de Newton[9] que buscam reduzir o custo computacional. Há estratégias para simplificar as computações da matriz jacobiana[10] e para reduzir o custo nas computações das atualizações de Newton.

2.2.1 Atualização Cíclica da Matriz Jacobiana

Geralmente, ao simplificarmos a matriz jacobina J_F ou aproximarmos a atualização de Newton $\delta^{(k)}$, perdemos a convergência quadrática do método

[9] Isaac Newton, 1642 - 1727, matemático, físico, astrônomo, teólogo e autor inglês. Fonte: Wikipédia.

[10] Carl Gustav Jakob Jacobi, 1804 - 1851, matemático alemão. Fonte: Wikipédia.

(consulte a Observação 2.1.1). A ideia é, então, buscar uma convergência pelo menos super-linear, i.e.

$$\|e^{(k+1)}\| \approx \rho_k \|e^{(k)}\|, \tag{2.28}$$

com $\rho_k \to 0$ quando $k \to \infty$. Aqui, $e^{(k)} := x^* - x^{(k)}$. Se a convergência é superlinear, então podemos usar a seguinte aproximação

$$\rho_k \approx \frac{\|\delta^{(k)}\|}{\|\delta^{(k-1)}\|} \tag{2.29}$$

ou, equivalentemente,

$$\rho_k \approx \left(\frac{\|\delta^{(k)}\|}{\|\delta^{(0)}\|}\right)^{\frac{1}{k}} \tag{2.30}$$

Isso mostra que podemos acompanhar a convergência das iterações pelo fator

$$\beta_k = \frac{\|\delta^{(k)}\|}{\|\delta^{(0)}\|}. \tag{2.31}$$

Ao garantirmos $0 < \beta_k < 1$, deveremos ter uma convergência superlinear.

Vamos, então, propor o seguinte método tipo Newton de atualização cíclica da matriz jacobiana.

1. Escolha $0 < \beta < 1$
2. $J := J_F(x^{(0)})$
3. $J\delta^{(0)} = -F(x^{(0)})$
4. $x^{(1)} = x^{(0)} + \delta^{(0)}$
5. Para $k = 1, \ldots$ até critério de convergência:
   (a) $J\delta^{(k)} = -F(x^{(k)})$
   (b) $x^{(k+1)} = x^{(k)} + \delta^{(k)}$
   (c) Se $\|\delta^{(k)}\|/\|\delta^{(0)}\| > \beta$:
      i. $J := J_F(x^{(k)})$

Exercício 2.2.1. Implemente uma versão do Método Tipo Newton apresentado acima e aplique-o para simular o problema discutido no Exemplo 2.1.1 para $\nu = 1., 0.1, 0.01, 0.001, 0.0001$. Faça uma implementação com suporte para matrizes esparsas.

2.3 Problemas de Minimização

Vamos considerar o seguinte **problema de minimização**: dada a **função objetivo** $f : D \subset \mathbb{R}^n \to \mathbb{R}$, resolver

$$\min_{x \in D} f(x). \tag{2.32}$$

No que segue e salvo dito explicitamente ao contrário, vamos assumir que o problema está bem determinado e que f é suficientemente suave. Ainda, vamos assumir as seguintes notações:

- gradiente de f

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right) \tag{2.33}$$

- derivada direcional de f com respeito a $d \in \mathbb{R}^n$

$$\frac{\partial f}{\partial d} = \nabla f(x) \cdot d \tag{2.34}$$

- matriz hessiana de f, $H = [h_{i,j}]_{i,j=1}^{n,n}$

$$h_{i,j}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{2.35}$$

Observação 2.3.1 (Condições de Otimização)**.** Se $\nabla f(x^*) = 0$ e $H(x^*)$ é positiva definida, então x^* é um mínimo local de f em uma vizinhança não vazia de x^*. Consulte mais em [6, Seção 7.2]. Um ponto x^* tal que $\nabla f(x^*) = 0$ é chamado de **ponto crítico**.

2.3.1 Métodos de declive

Um método de declive consiste em uma iteração tal que: dada uma aproximação inicial $x^{(0)} \in \mathbb{R}^n$, computa-se

$$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}, \tag{2.36}$$

com **tamanho de passo** $\alpha_k > 0$, para $k = 0,1,2,\ldots$ até convergência. As **direções descendentes** são tais que

$$d^{(k)} \cdot \nabla f(x^{(k)}) < 0, \quad \text{se } \nabla f(x^{(k)}) \neq 0, \tag{2.37}$$

$$d^{(k)} = 0, \quad \text{noutro caso.} \tag{2.38}$$

Observação 2.3.2. (Condição de Convergência) Da Série de Taylor[11], temos que

$$f(x^{(k)} + \alpha_k d^{(k)}) - f(x^{(k)}) = \alpha_k \nabla f(x^{(k)}) \cdot d^{(k)} + \varepsilon, \tag{2.39}$$

com $\varepsilon \to 0$ quando $\alpha_k \to 0$. Como consequência da continuidade da f, para α_k suficientemente pequeno, o sinal do lado esquerdo é igual a do direito desta última equação. Logo, para tais α_k e $d^{(k)} \neq 0$ uma direção descendente temos garantido que

$$f(x^{(k)} + \alpha_k d^{(k)}) < f(x^{(k)}). \tag{2.40}$$

Notamos que um método de declive fica determinado pelas escolhas da direção de declive d_k e o tamanho do passo α_k. Primeiramente, vamos a este último item.

Pesquisa linear

A computação de α_k consiste em resolver o seguinte problema de minimização:

$$\min_{\alpha \in \mathbb{R}} \phi(\alpha) = f(x^{(k)} + \alpha d^{(k)}). \tag{2.41}$$

Entretanto, a resolução exata deste problema é muitas vezes não factível. Técnicas de aproximações para a resolução deste problema são, então, aplicadas. Tais técnicas são chamadas de **pesquisa linear não exata**.

[11] Brook Taylor, 1685 - 1731, matemático britânico. Fonte: Wikipédia.

A **condição de Armijo** é que a escolha de α_k deve ser tal que

$$f\left(x^{(k)} + \alpha_k d^{(k)}\right) \leq f(x^{(k)}) + \sigma\alpha_k \nabla f\left(x^{(k)}\right) \cdot d^{(k)}\Big), \tag{2.42}$$

para alguma constante $\sigma \in (0, 1/2)$. Ou seja, a redução em f é esperada ser proporcional à derivada direcional de f com relação a direção $d^{(k)}$ no ponto $x^{(k)}$. Em aplicações computacionais, σ é normalmente escolhido no intervalo $[10^{-5}, 10^{-1}]$.

A condição (2.42) não é suficiente para evitar escolhas muito pequenas de α_k. Para tanto, pode-se empregar a **condição de curvatura**, a qual requer que

$$\nabla f\left(x^{(k)} + \alpha_k d^{(k)}\right) \cdot d^{(k)} \geq \beta \nabla f\left(x^{(k)}\right) d^{(k)}\Big), \tag{2.43}$$

para $\beta \in [\sigma, 1/2]$. Notemos que o lado esquerdo de (2.43) é igual a $\phi'(\alpha_k)$. Ou seja, este condição impõe que α_k seja maior que $\beta\phi'(0)$. Normalmente, escolhe-se $\beta \in [10^{-1}, 1/2]$. Juntas, (2.42) e (2.43) são conhecidas como **condições de Wolfe**[12].

2.3.2 Método do Gradiente

O Método do Gradiente (**Método do Máximo Declive**) é um Método de Declive tal que as direções descendentes são o oposto do gradiente da f, i.e.

$$d^{(k)} = -\nabla f(x^{(k)}). \tag{2.44}$$

É fácil verificar que as condições (2.37)-(2.38) são satisfeitas.

Exemplo 2.3.1. Consideremos o problema de encontrar o mínimo da **função de Rosenbrock**[13]

$$f(x) = \sum_{i=1}^{n} 100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2. \tag{2.45}$$

O valor mínimo desta função é zero e ocorre no ponto $x = 1$. Esta função é comumente usada como caso padrão para teste de métodos de otimização.

[12] Philip Wolfe, 1927 - 2016, matemático estadunidense. Fonte: Wikipédia.
[13] Howard Harry Rosenbrock, 1920 - 2010, engenheiro britânico. Fonte: Wikipedia.

Para o Método do Gradiente, precisamos de suas derivadas parciais:

$$\frac{\partial f}{\partial x_1} = -400x_1\left(x_2 - x_1^2\right) - 2(1 - x_1) \tag{2.46}$$

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{n} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2(1 - x_{i-1})\delta_{i-1,j} \tag{2.47}$$

$$\frac{\partial f}{\partial x_n} = 200\left(x_n - x_{n-1}^2\right) \tag{2.48}$$

Código 2.1: Algoritmo do Gradiente

```
1  import numpy as np
2  import numpy.linalg as npla
3  import scipy.optimize as spopt
4
5  # fun obj
6    def fun(x):
7    '''
8    Função de Rosenbrock
9    '''
10   return sum(100.*(x[1:]-x[:-1]**2.)**2. + (1.-x[:-1])**2.)
11
12 # gradiente da fun
13 def grad(x):
14   xm = x[1:-1]
15   xm_m1 = x[:-2]
16   xm_p1 = x[2:]
17   der = np.zeros_like(x)
18   der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
19   der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
20   der[-1] = 200*(x[-1]-x[-2]**2)
21
22   return der
23
24 # problem dimension
25 n = 2
26
```

```
# num max iters
maxiter = 100000
# tolerancia
tol = 1e-10

# aprox. inicial
x = np.zeros(n)

siga = [x]
for k in range(maxiter):
  # direcao descendente
  d = -grad(x)

  # pesquisa linear
  alpha = spopt.line_search(fun, grad, x, d)[0]

  # atualizacao
  x = x + alpha * d

  nad = npla.norm(alpha * d)
  nfun = npla.norm(fun(x))

  if ((k+1) % 10 == 0):
    print(f"{k+1}: {alpha:1.2e} {nad:1.2e} {nfun:1.2e}")
    siga.append(x)

  if ((nfun < tol) or np.isnan(nfun)):
    break
```

Exercício 2.3.1. Aplique o Método do Gradiente para resolver o problema de minimização com as seguintes funções objetivos:

a) Função de Beale

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \quad (2.49)$$

Solução: f(3, 0.5) = 0.

b) Função de Goldstein-Price

$$f(x,y) = \left[1 + (x + y + 1)^2 \left(19 - 14x + 3x^2 - 14y + 6xy + 3y^2\right)\right]$$

$$\times \left[30 + (2x - 3y)^2 \left(18 - 32x + 12x^2 + 48y - 36xy + 27y^2\right)\right] \tag{2.50}$$

Solução: f(0,-1) = 3.

c) Função de Booth

$$f(x,y) = (x + 2y - 7)^2 + (2x + y - 5)^2 \tag{2.51}$$

Solução: f(1,3)=0.

d) Função de Rastringin:

$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right] \tag{2.52}$$

Solução: f(0)=0.

2.3.3 Método de Newton

O Método de Newton[14] para problemas de otimização é um Método de Declive com direções descendentes

$$d^{(k)} = -H^{-1}(x^{(k)})\nabla f(x^{(k)}), \tag{2.53}$$

assumindo que a hessiana H seja definida positiva dentro de uma vizinhança suficientemente grande do ponto de mínimo x^*.

Observação 2.3.3. A cada iteração de Newton é necessário computar a inversa da matriz hessiana. Este é um passo crítico para o desempenho computacional. Desta forma, a escolha do método para o cálculo da inversa é normalmente explicitado, este é chamado de *solver* linear. Por exemplo, **Newton-Krylov** é o nome dado ao Método de Newton que utiliza um Método de Subespaço de Krylov como *solver* linear. Mais especificamente, **Newton-GMRES** quando a inversa é computada com o GMRES. Uma escolha natural é **Newton-GC**, tendo em vista que o Método de Gradiente Conjugado é ideal para matriz simétrica e definida positiva.

[14] Isaac Newton, 1642 - 1727, matemático, físico, astrônomo, teólogo e autor inglês. Fonte: Wikipédia.

Observação 2.3.4. Na implementação computacional não é necessário computar a inversa da hessiana, mais apenas computar $d^{(k)}$ resolvendo o seguinte sistema linear

$$H(x^{(k)})d^{(k)} = -\nabla f(x^{(k)}). \tag{2.54}$$

Exemplo 2.3.2. Seguindo o Exemplo 2.3.1, temos que a hessiana associada é a matriz simétrica $H = [h_{i,j}]_{i,j=1}^{n,n}$ com

$$h_{1,1} = \frac{\partial^2 f}{\partial x_1^2} = 1200x_1^2 - 400x_2 + 2 \tag{2.55}$$

$$h_{1,2} = \frac{\partial^2 f}{\partial x_1 \partial x_2} = -400x_1 \tag{2.56}$$

$$\begin{aligned} h_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j} &= 200(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 400(\delta_{i+1,j} - 2x_i\delta_{i,j}) \\ &\quad - 400\delta_{i,j}(x_{i+1} - x_i^2) + 2\delta_{i,j} \end{aligned} \tag{2.57}$$

$$h_{n-1,n} = \frac{\partial^2 f}{\partial x_{n-1} \partial x_n} = -400x_{n-1} \tag{2.58}$$

$$h_{n,n} = \frac{\partial^2 f}{\partial x_{n-1}} = 200 \tag{2.59}$$

Notemos que a hessiana é uma matriz tridiagonal.

Código 2.2: Algoritmo de Newton

```
1  import numpy as np
2  import numpy.linalg as npla
3  import scipy.optimize as spopt
4
5  # fun obj
6  def fun(x):
7    '''
8    Funcao de Rosenbrock
9    '''
10   return sum(100.*(x[1:]-x[:-1]**2.)**2. + (1.-x[:-1])**2.)
11
```

```
12 # gradiente da fun
13 def grad(x):
14   xm = x[1:-1]
15   xm_m1 = x[:-2]
16   xm_p1 = x[2:]
17   der = np.zeros_like(x)
18   der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
19   der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
20   der[-1] = 200*(x[-1]-x[-2]**2)
21
22   return der
23
24 def hess(x):
25   x = np.asarray(x)
26   H = np.diag(-400*x[:-1],1) - np.diag(400*x[:-1],-1)
27   diagonal = np.zeros_like(x)
28   diagonal[0] = 1200*x[0]**2-400*x[1]+2
29   diagonal[-1] = 200
30   diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
31   H = H + np.diag(diagonal)
32
33   return H
34
35 # dimensao
36 n = 2
37
38 # num max iters
39 maxiter = 100000
40 # tolerancia
41 tol = 1e-10
42
43 # aprox. inicial
44 x = np.zeros(n)
45
46 for k in range(maxiter):
47
48   # direcao descendente
49   d = npla.solve (hess(x),-grad(x))
50
51   # pesquisa linear
```

```
52    alpha = spopt.line_search(fun, grad, x, d)[0]
53
54    # atualizacao
55    x = x + alpha * d
56
57    nad = npla.norm(alpha * d)
58    nfun = npla.norm(fun(x))
59
60    print(f"{k+1}: {alpha:1.2e} {nad:1.2e} {nfun:1.2e}")
61
62    if ((nfun < tol) or np.isnan(nfun)):
63      break
```

Observação 2.3.5. Métodos Tipo Newton Métodos Tipo Newton são aqueles que utilizam uma aproximação para a inversa da matriz hessiana. Uma estratégia comumente aplicada, é a de atualizar a hessiana apenas em algumas iterações, baseando-se em uma estimativa da taxa de convergência. Na Seção 2.2, exploramos esta técnica no contexto de resolução de sistemas não lineares.

Exercício 2.3.2. Aplique o Método de Newton para minimizar a função de Rosenbrock no caso de várias dimensões. Para tanto, utilize uma implementação eficiente com suporte para matrizes esparsas. Compare o desempenho entre os métodos Newton-GMRES e Newton-GC.

Exercício 2.3.3. Aplique o Método de Newton para minimizar as funções dadas no Exercício 2.3.1.

2.3.4 Método do Gradiente Conjugado

Métodos do Gradiente Conjugado são obtidos escolhendo-se as direções descendentes

$$d^{(k)} = -\nabla f(x^{(k)}) + \beta_k d^{(k-1)}, \tag{2.60}$$

onde β_k é um escalar escolhido de forma que as direções $\{d^{(k)}\}$ sejam mutuamente ortogonais com respeito a uma dada norma. Por exemplo, o **Método de Fletcher-Reeves** consiste em escolher

$$\beta_k = \frac{\nabla f(x^{(k)}) \cdot \nabla f(x^{(k)})}{\nabla f(x^{(k-1)}) \cdot \nabla f(x^{(k-1)})}, \tag{2.61}$$

o que garante que as direções sejam mutuamente ortogonais com respeito ao produto interno euclidiano.

Observação 2.3.6. Outras variações comumente empregadas são o Método de Polak-Ribière e suas variantes. Consulte mais em [5, Seção 5.2].

Exemplo 2.3.3. Implementação do Método de Fletcher-Reeves para a minimização da função de Rosenbrock dada no Exemplo 2.3.1.

Código 2.3: Algoritmo de Fletcher-Reeves

```
1  import numpy as np
2  import numpy.linalg as npla
3  import scipy.optimize as spopt
4
5  # fun obj
6  def fun(x):
7    '''
8    Funcao de Rosenbrock
9    '''
10   return sum(100.*(x[1:]-x[:-1]**2.)**2. + (1.-x[:-1])**2.)
11
12 # gradiente da fun
13 def grad(x):
14   xm = x[1:-1]
15   xm_m1 = x[:-2]
16   xm_p1 = x[2:]
17   der = np.zeros_like(x)
18   der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
19   der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
20   der[-1] = 200*(x[-1]-x[-2]**2)
21
22   return der
23
24 # dimensao do prob
25 n = 2
26
27 # num max iters
28 maxiter = 100000
29 # tolerancia
30 tol = 1e-10
31
```

```
32 # aprox. inicial
33 x = np.zeros(n)
34
35 # iteracoes CG
36 gf = grad(x)
37 d = -gf
38
39 for k in range(maxiter):
40
41   # pesquisa linear
42   alpha = spopt.line_search(fun, grad, x, d)[0]
43
44   # atualizacao
45   x = x + alpha * d
46
47   nad = npla.norm(alpha * d)
48   nfun = npla.norm(fun(x))
49
50   print(f"{k+1}: {alpha:1.2e} {nad:1.2e} {nfun:1.2e}")
51
52   if ((nfun < tol) or np.isnan(nfun)):
53     break
54
55   # prepara nova iter
56   ngf = grad(x)
57
58   beta = np.dot(ngf,ngf)/np.dot(gf,gf)
59   d = -ngf + beta * d
60
61   gf = ngf
```

Exercício 2.3.4. Teste o Método de Fletcher-Reeves para a minimização das funções dadas no Exercício 2.3.1.

Capítulo 3

Autovalores e Autovetores

[Vídeo] | [Áudio] | [Contatar]

Neste capítulo, estudamos métodos numéricos para a computação de autovalores e autovetores de matrizes.

3.1 Método da Potência

O método da potência é uma técnica iterativa para computar o autovalor dominante de uma matriz e um autovetor associado. Modificações do método, tornam possível sua aplicação para a determinação de outros autovalores próximos a um dado número. Desta forma, pode ser utilizá-lo em conjunto com outra técnica de aproximação de autovalores e autovetores.

3.1.1 Autovalor dominante

Vamos denotar o **espectro** de uma dada matriz $A \in \mathbb{C}^{n\times n}$ por

$$\sigma(A) = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}, \tag{3.1}$$

sendo $\lambda_i \in \mathbb{C}$ o i-ésimo **autovalor** de uma **matriz diagonalizável**[1] $A = [a_{i,j}]_{i,j=1}^{n,n}$, i.e. existe um vetor $0 \neq v^{(i)} \in \mathbb{C}^n$ tal que

$$Av^{(i)} = \lambda_i v^{(i)}, \tag{3.2}$$

[1] Existe uma base de $\mathbb{C}^{n\times n}$ formada apenas de autovetores de A.

sendo $v^{(i)}$ chamado de **autovetor** associado a λ_i. No desenvolvimento do **Método da Potência**, vamos assumir que os autovalores podem ser ordenados da seguinte forma

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|. \tag{3.3}$$

Assim sendo, o λ_1 é chamado de **autovalor dominante**. Na sequência, vamos denotar por A^H a matriz adjunta[2] de A, i.e. $A^H = [\bar{a}_{j,i}]_{i,j=1}^{n,n}$.

A **iteração** do Método da Potência consiste em

$$z^{(k)} = Aq^{(k-1)}, \tag{3.4}$$
$$q^{(k)} = z^{(k)} / \left\| z^{(k)} \right\|, \tag{3.5}$$
$$\nu_k = (q^{(k)})^H A q^{(k)}, \tag{3.6}$$

com dada aproximação inicial $q^{(0)}$ para o autovetor associado a λ_1. Quando o método é bem sucedido, tem-se $\nu_k \to \lambda_1$ quando $k \to \infty$.

Para mostrar a **convergência** do método, basta mostrar que $q^{(k)}$ converge para um autovetor associado a λ_1. Primeiramente, notemos que[3]

$$q^{(k)} = \frac{A^k q^{(0)}}{\|A^k q^{(0)}\|}, \quad k \geq 1. \tag{3.7}$$

Como A é diagonalizável, temos que existe uma base $\{v^{(1)}, v^{(2)}, \ldots, v^{(n)}\}$ de $\mathbb{C}^{n\times n}$ formada apenas de autovetores de A. Segue que

$$q^{(0)} = \sum_{i=1}^{n} \alpha_i v^{(i)}, \tag{3.8}$$

onde $\alpha_i \in \mathbb{C}$. Vamos assumir que $\alpha_1 \neq 0$[4]. Ainda, $Av^{(i)} = \lambda_i v^{(i)}$, donde

$$A^k q^{(0)} = \sum_{i=1}^{n} \alpha_i A^k v^{(i)} \tag{3.9}$$
$$= \sum_{i=1}^{n} \alpha_i \lambda_i^k v^{(i)} \tag{3.10}$$

[2] Também, chamada de matriz conjugada transposta de A.
[3] Segue por indução matemática.
[4] Condição necessária para a convergência.

$$= \alpha_1 \lambda_1^k \left[v^{(1)} + \sum_{i=2}^{n} \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_k}{\lambda_1} \right)^k v^{(i)} \right] \tag{3.11}$$

Como $|\lambda_i|/|\lambda_1| < 1$, $i = 2, \ldots, n$, temos que

$$y^{(k)} = \sum_{i=2}^{n} \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_k}{\lambda_1} \right)^k v^{(i)} \to 0, \tag{3.12}$$

quando $k \to \infty$. Logo, temos que

$$q^{(k)} = \frac{\alpha_1 \lambda_1^k \left(v^{(1)} + y^{(k)} \right)}{\| \alpha_1 \lambda_1 \left(v^{(1)} + y^{(k)} \right) \|} \tag{3.13}$$

$$= \frac{\lambda_1}{|\lambda_1|} \frac{\left(v^{(1)} + y^{(k)} \right)}{\| v^{(1)} + y^{(k)} \|}. \tag{3.14}$$

Por fim, observamos que $\lambda_1/|\lambda_1|$ tem o mesmo sinal de λ_1. Portanto, $q^{(k)}$ tende a um múltiplo não nulo de $v^{(1)}$ quando $k \to \infty$.

Observação 3.1.1 (Taxa de convergência.)**.** Pode-se mostrar a seguinte taxa de convergência para o Método da Potência

$$\left\| \tilde{q}^{(k)} - v^{(1)} \right\| \leq C \left| \frac{\lambda_2}{\lambda_1} \right|^k, \quad k \geq 1, \tag{3.15}$$

onde

$$\tilde{q}^{(k)} = v^{(1)} + \sum_{i=2}^{n} \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_i}{\lambda_1} \right)^k v^{(i)}. \tag{3.16}$$

Consulte [6, Seção 5.3.].

Código 3.1: Algoritmo do Método da Potência

```
1 import numpy as np
2 import numpy.linalg as npla
3
4 def metPot(A, q0, maxiter=1000, tol=1e-14):
5   q = q0/npla.norm(q0)
6   nu0 = np.dot(q, A @ q)
7   print(f"{0}: {nu0}")
```

```
8
9     info = -1
10    for k in range(maxiter):
11      z = A @ q
12      q = z/npla.norm(z)
13      nu = np.dot(q, A @ q)
14
15      print(f"{k+1}: {nu}")
16      if (np.fabs(nu-nu0) < tol):
17        info = 0
18        break
19
20      nu0 = nu
21
22    return nu, q, info
```

Exercício 3.1.1. Use o Método da Potência para computar o autovalor dominante da matriz de coeficientes do problema discreto associado ao Exercício 1.1.2. Estude sua convergência para diferentes tamanhos da matriz.

Exercício 3.1.2. Use o Método da Potência para computar o autovalor dominante da matriz de coeficientes do problema discreto associado ao Exemplo 1.1.2. Estude sua convergência para diferentes tamanhos da matriz.

3.1.2 Método da Potência Inverso

Esta variação do Método da Potência nos permite computar o autovalor mais próximo de um dado número $\mu \in \mathbb{C}$, $\mu \notin \sigma(A)$. A ideia é aplicar o método para a matriz

$$M_\mu^{-1} = (A - \mu I)^{-1}. \tag{3.17}$$

Neste contexto, μ é chamado de **deslocamento** (em inglês, *shift*).

Notemos que se (λ_i, v_i) é um autopar de A, então $\xi_i = (\lambda_i - \mu)^{-1}$ é autovalor de M_μ^{-1} associado ao autovetor v_i. De fato,

$$(\lambda_i - \mu)v_i = (A - \mu I)v_i \tag{3.18}$$
$$M_\mu^{-1}(\lambda_i - \mu)v_i = v_i \tag{3.19}$$

$$M_\mu^{-1} v_i = (\lambda_i - \mu)^{-1} v_i. \tag{3.20}$$

Isso também mostra que A e M_μ^{-1} têm os mesmos autovetores.

Agora, se μ for suficientemente próximo de λ_m, autovalor simples de A, então

$$|\lambda_m - \mu| < |\lambda_i - \mu|, \quad i = 1,2,\ldots,n,\ i \neq m. \tag{3.21}$$

Com isso, $\xi_i = (\lambda_m - \mu)^{-1}$ é o autovalor dominante de M_μ^{-1} e, portanto, a iteração do Método da Potência aplicada a M_μ^{-1} fornece este autovalor. Mais especificamente, como A e M_μ^{-1} tem os mesmos autovetores, a **iteração do Método da Potência Inverso** é dada como segue

$$(A - \mu I) z^{(k)} = q^{(k-1)}, \tag{3.22}$$
$$q^{(k)} = z^{(k)} / \|z^{(k)}\|, \tag{3.23}$$
$$\nu_k = (q^{(k)})^H A q^{(k)}. \tag{3.24}$$

Exercício 3.1.3. Use o Método da Potência para computar diferentes autovalores da matriz de coeficientes do problema discreto associado ao Exercício 1.1.2. Estude sua convergência para diferentes tamanhos da matriz.

Exercício 3.1.4. Use o Método da Potência para computar diferentes autovalores da matriz de coeficientes do problema discreto associado ao Exemplo 1.1.2. Verifique se há vantagem em aplicar os métodos GMRES e GC na resolução de (3.22).

3.2 Iteração QR

A iteração QR é um método para a computação aproximada de todos os autovalores de uma dada matriz A. A ideia é explorar um método iterativo para a computação da **decomposição de Schur**[5] de A, i.e. encontrar uma

[5] Issai Schur, 1875 - 1941, matemático russo-alemão. Fonte: Wikipédia.

matriz unitária U[6] tal que

$$U^H AU = \begin{bmatrix} \lambda_1 & t_{12} & \cdots & t_{1n} \\ 0 & \lambda_2 & & t_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & \vdots & 0 & \lambda_n \end{bmatrix} \tag{3.25}$$

Assumindo $A \in \mathbb{R}^{n\times n}$, sejam $Q^{(0)} \in \mathbb{R}^{n\times n}$ uma **matriz ortogonal**[7] e $T^{(0)} = (Q^{(0)})^T A Q^{(0)}$. A iteração QR consiste em

determinar $Q^{(k)}, R^{(k)}$ tal que

$$Q^{(k)}R^{(k)} = T^{(k-1)} \qquad \text{(fatoração QR)} \tag{3.26}$$

$$T^{(k)} = R^{(k)}Q^{(k)}, \tag{3.27}$$

para $k = 1,2,\ldots$.

Ou seja, a cada iteração k, computa-se a fatoração da matriz $T^{(k-1)}$ como o produto de uma matriz ortogonal $Q^{(k)}$ com uma matriz triangular superior $R^{(k)}$. Então, computa-se uma nova aproximação $T^{(k)}$ pela multiplicação de $R^{(k)}$ por $Q^{(k)}$. Com isso, temos

$$T^{(k)} = R^{(k)}Q^{(k)} \tag{3.28}$$

$$= (Q^{(k)})^T (Q^{(k)}R^{(k)})Q^{(k)} \tag{3.29}$$

$$= (Q^{(k)})^T T^{(k-1)} Q^{(k)} \tag{3.30}$$

$$= (Q^{(k)})^T R^{(k-1)} Q^{(k-1)} Q^{(k)} \tag{3.31}$$

$$= (Q^{(k)})^T (Q^{(k-1)})^T Q^{(k-1)} R^{(k-1)} Q^{(k-1)} Q^{(k)} \tag{3.32}$$

$$= (Q^{(k-1)}Q^{(k)})^T T^{(k-2)} (Q^{(k-1)}Q^{(k)}) \tag{3.33}$$

$$= (Q^{(0)} \cdots Q^{(k)})^T A (Q^{(0)} \cdots Q^{(k)}). \tag{3.34}$$

Observação 3.2.1 (Convergência do método QR)**.** Se $A \in \mathbb{R}^{n\times n}$ for uma matriz com autovalores tais que

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|, \tag{3.35}$$

[6]Uma matriz U é dita unitária quando $U^{-1} = U^H$.
[7]Uma matriz Q é dita ortogonal quando $Q^T Q = I$.

então

$$\lim_{k\to\infty} T^{(k)} = \begin{bmatrix} \lambda_1 & t_{12} & \cdots & t_{1n} \\ 0 & \lambda_2 & & t_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & \vdots & 0 & \lambda_n \end{bmatrix}. \tag{3.36}$$

Para a taxa de convergência, temos

$$|t_{i,i-1}^{(k)}| = \mathcal{O}\left(\left|\frac{\lambda_i}{\lambda_{i-1}}\right|^k\right), \quad i = 2,\ldots,n, \tag{3.37}$$

quando $k \to \infty$.

Ainda, se A for uma **matriz simétrica**, então $T^{(k)}$ tende a uma matriz diagonal quando $k \to \infty$.

Observação 3.2.2. Variantes do método QR permitem sua aplicação para matrizes mais gerais. Consulte [3].

Uma forma de eficiente do método QR é chamada de **iteração Hessenberg[8]-QR**. Consiste em inicializar $T^{(0)}$ como uma **matriz de Hessenberg superior**, i.e. $t_{i,j}^{(0)} = 0$ para $i > j+1$. A computação de $T^{(0)}$ é feito com **matrizes de Householder[9]** e a fatoração QR de $T^{(k)}$ utiliza de **matrizes de Givens[10]**.

Código 3.2: Algoritmo da Iteração QR

```
1 import numpy as np
2 import numpy.linalg as npla
3
4 def qr_iter(A, Q=None, maxiter=10, tol=1e-5):
5     Q = np.eye(A.shape[0]) if Q==None else Q
6     T0 = Q.T @ A @ Q
7     info = -1
8     for k in range(maxiter):
9         Q, R = npla.qr(T0)
```

[8]Karl Adolf Hessenberg, 1904 - 1959, matemático e engenheiro alemão. Fonte: Wikipédia.
[9]Alston Scott Householder, 1904 - 1993, matemático estadunidense. Fonte: Wikipédia.
[10]James Wallace Givens Jr., 1910 - 1993, matemático estadunidense. Fonte: Wikipédia.

```
        T = R @ Q
        if (npla.norm(T-T0) < tol):
            info = 0
            break
        T0 = T
    return T, info
```

Observação 3.2.3 (Computação dos autovalores)**.** Se v é autovetor associado ao autovalor simples λ de A, então

$$Av = \lambda v \tag{3.38}$$

$$(Q^{(k)})^T A Q^{(k)} (Q^{(k)})^T v \approx \lambda (Q^{(k)})^T v \tag{3.39}$$

Denotando, $y = (Q^{(k)})^T v$, temos

$$T^{(k)} y = \lambda y. \tag{3.40}$$

Com isso, podemos computar y e, então, temos $v \approx Q^{(k)} y$.

Exercício 3.2.1. Use a iteração QR para computar os autovalores da matriz de coeficientes do problema discreto associado ao Exercício 1.1.2.

Exercício 3.2.2. Use a iteração QR para computar os autovalores da matriz de coeficientes do problema discreto associado ao Exemplo 1.1.2. Use `spla.hessenberg` para computar $T^{(0)}$.

Capítulo 4

Integração

[Vídeo] | [Áudio] | [Contatar]

Neste capítulo, estudamos métodos numéricos para a integração de funções reais.

4.1 Integração Autoadaptativa

Vamos considerar o problema de integrar

$$I(a,b) = \int_a^b f(x)\,dx \tag{4.1}$$

pela **Regra de Simpson**[1]. Em um dado subintervalo $[\alpha,\beta] \subset [a,b]$, temos

$$I(\alpha,\beta) = \underbrace{\frac{h_0}{3}\left[f(\alpha) + 4f(\alpha+h_0) + f(\beta)\right]}_{S(\alpha,\beta)} - \frac{h_0^5}{90} f^{(4)}(\xi), \tag{4.2}$$

onde $h_0 = (\beta-\alpha)/2$ e $\xi \in (\alpha,\beta)$. Ou seja, temos que

$$I(\alpha,\beta) - S(\alpha,\beta) = -\frac{h_0^5}{90} f^{(4)}(\xi). \tag{4.3}$$

A ideia é explorarmos esta informação de forma a obtermos uma estimativa para o erro de integração no intervalo $[\alpha,\beta]$ sem necessitar computar $f^{(4)}$.

[1] Consulte mais sobre a Regra de Simpson em Seção 10.1 Regras de Newton-Cotes.

Aplicando a Regra de Simpson na partição $[\alpha,(\alpha+\beta)/2] \cup [(\alpha+\beta)/2,\beta]$, obtemos

$$I(\alpha,\beta) - S_2(\alpha,\beta) = -\frac{(h_0/2)^5}{90}\left(f^{(4)}(\xi) + f^{(4)}(\eta)\right), \tag{4.4}$$

onde $\xi \in (\alpha,(\alpha+\beta)/2)$, $\eta \in (\alpha+\beta)/2,\beta)$ e

$$S_2(\alpha,\beta) = S(\alpha,(\alpha+\beta)/2) + S((\alpha+\beta)/2,\beta). \tag{4.5}$$

Agora, **vamos assumir que** $f^{(4)}(\xi) \approx f^{(4)}(\eta)$ de forma que temos

$$I(\alpha,\beta) - S_2(\alpha,\beta) \approx -\frac{1}{16}\frac{h_0^5}{90} f^{(4)}(\xi). \tag{4.6}$$

De (4.3) e (4.6), obtemos

$$\frac{h_0^5}{90} f^{(4)}(\xi) \approx \frac{16}{15} \underbrace{[S(\alpha,\beta) - S_2(\alpha,\beta)]}_{\mathcal{E}(\alpha,\beta)}. \tag{4.7}$$

Isto nos fornece a seguinte estimativa *a posteriori* do erro

$$|I(\alpha,\beta) - S_2(\alpha,\beta)| \approx \frac{|\mathcal{E}(\alpha,\beta)|}{15}. \tag{4.8}$$

Na prática, costuma-se utilizar a seguinte estimativa mais restrita

$$|I(\alpha,\beta) - S_2(\alpha,\beta)| \approx \frac{|\mathcal{E}(\alpha,\beta)|}{10}. \tag{4.9}$$

Para garantir uma precisão global em $[a,b]$ igual a uma dada tolerância, é suficiente impor que

$$\frac{|\mathcal{E}(\alpha,\beta)|}{10} \leq \epsilon\frac{\beta-\alpha}{b-a}. \tag{4.10}$$

Código 4.1: Algoritmo Simpson Autoadaptativo

```
1 def simpson(f,a,b):
2   return (b-a)/6. * (f(a) + 4*f((a+b)/2.) + f(b))
3
4 def simad(f,a,b,tol=1e-8,hmin=1e-10):
5   # intervalo calculado
```

```
6     S = (a,a)
7     # intervalo a ser calculado
8     N = (a,b)
9     # intervalo ativo
10    A = N
11    # aprox calculada em [a, b]
12    J = 0.
13    # aprox calculada em A
14    JA = 0.
15    # num f eval
16    nfe = 0
17    # info
18    info = 0
19    while (S != (a,b)):
20      print(S,N,A)
21      # tamanho do intervalo
22      h = A[1]-A[0]
23      while (h > hmin):
24        J0 = simpson(f,A[0],A[1])
25        JA = simpson(f,A[0],A[0]+h/2.)
26        JA += simpson(f,A[0]+h/2.,A[1])
27        nfe += 9
28        # est erro
29        est = np.fabs(J0-JA)/10.
30        # criterio de parada
31        if (est < tol*h/(b-a)):
32          break
33        else:
34          A = (A[0],A[0]+h/2.)
35          h = h/2.
36          print("\t",S,N,A,h,est)
37          if (h < hmin):
38            print("Atencao! h < hmin !")
39            info = -1
40            break
41      J += JA
42      S = (a, A[1])
43      N = (A[1], b)
44      A = N
45    return J, nfe, info
```

Exemplo 4.1.1.

$$\int_{-3}^{4} \operatorname{arctg}(10x)\,dx = -3\operatorname{arctg}(30) - \frac{\ln(1601)}{20} + \frac{\ln(901)}{20} + 4\operatorname{arctg}(40) \tag{4.11}$$

$$\approx 1.54203622 \tag{4.12}$$

Exercício 4.1.1. Implemente uma abordagem autoadaptativa usando a Regra do Trapézio. Valide-a e compare com o exemplo anterior.

4.2 Integrais múltiplas

Vamos trabalhar com métodos para a computação de integrais múltiplas

$$\int\int_R f(x,y)\,dA. \tag{4.13}$$

Em uma região retangular $A = [a,b] \times [c,d]$, podemos reescrevê-la como uma **integral iterada**

$$\int\int_R f(x,y)\,dA = \int_a^b \int_c^d f(x,y)\,dy\,dx. \tag{4.14}$$

4.2.1 Regras de Newton-Cotes

Regra do Trapézio

A Regra do Trapézio[2] nos fornece

$$\int_c^d f(x,y)\,dy = \frac{h_y}{2}\left[f(x,c) + f(x,d)\right] - \frac{h_y^3}{12} f''(x,\eta) \tag{4.15}$$

com $h_y = (d-c)$ e $\eta \in (c,d)$. De forma iterada, temos

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \frac{h_y}{2}\int_a^b f(x,c)\,dx + \frac{h_y}{2}\int_a^b f(x,d)\,dx \tag{4.16}$$

$$- \frac{h_y^3}{12}\int_a^b f''(x,\eta)\,dx. \tag{4.17}$$

[2] Notas de Aula - Matemática Numérica.

Então, à exceção do termo do erro, aplicamos a Regra do Trapézio para as integrais em x. Obtemos

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \frac{h_y}{2}\frac{h_x}{2}\left[f(a,c)+f(b,c)\right] \tag{4.18}$$
$$+ \frac{h_y}{2}\frac{h_x}{2}\left[f(a,d)+f(b,d)\right] \tag{4.19}$$
$$- \frac{h_y}{2}\frac{h_x^3}{12}f''(\mu',c) \tag{4.20}$$
$$- \frac{h_y}{2}\frac{h_x^3}{12}f''(\mu'',d) \tag{4.21}$$
$$- \frac{h_y^3}{12}\int_a^b f''(x,\eta)\,dx, \tag{4.22}$$

com $h_x = (b-a)$, $\mu',\mu'' \in (a,b)$. Pelos Teorema do Valor Intermediário e pelo Teorema do Valor Médio, podemos ver que o erro é $O(h_x h_y^3 + h_x^3 h_y)$. Por fim, obtemos a Regra do Trapézio para Integrais Iteradas

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \frac{h_y}{2}\frac{h_x}{2}\left[f(a,c)+f(b,c)+f(b,d)+f(a,d)\right] \tag{4.23}$$
$$+ O(h_x h_y^3 + h_x^3 h_y). \tag{4.24}$$

Exemplo 4.2.1. A Regra do Trapézio fornece

$$\int_{1.5}^{2}\int_{1}^{1.5} \ln(x+2y)\,dy\,dx \approx 0.36. \tag{4.25}$$

Verifique!

Regra de Simpson

A Regra do Simpson[3] nos fornece

$$\int_c^d f(x,y)\,dy = \frac{h_y}{3}\left[f(x,y_1)+4f(x,y_2)+f(x,y_3)\right] \tag{4.26}$$
$$- \frac{h_y^5}{90}f^{(4)}(x,\eta) \tag{4.27}$$

[3] Notas de Aula - Matemática Numérica.

com $h_y = (d-c)/2$, $y_j = (j-1)h_y$, $j = 1,2,3$, e $\eta \in (c,d)$. De forma iterada, temos

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \frac{h_y}{3}\left[\int_a^b f(x,y_1)\,dx + 4\int_a^b f(x,y_2)\,dx + \int_a^b f(x,y_3)\,dx\right] \tag{4.28}$$

$$- \frac{h_y^5}{90}\int_a^b f^{(4)}(x,\eta)\,dx \tag{4.29}$$

Então, à exceção do termo do erro, aplicamos a Regra de Simpson para as integrais em x. Obtemos

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \frac{h_x h_y}{9}\left[f(x_1,y_1) + 4f(x_2,y_1) + f(x_3,y_1)\right] \tag{4.30}$$

$$+ \frac{4h_x h_y}{9}\left[f(x_1,y_2) + 4f(x_2,y_2) + f(x_3,y_2)\right] \tag{4.31}$$

$$+ \frac{h_x h_y}{9}\left[f(x_1,y_3) + 4f(x_2,y_3) + f(x_3,y_3)\right] \tag{4.32}$$

$$- \frac{h_x^5 h_y}{270} f^{(4)}(\mu_1,y_1) \tag{4.33}$$

$$- \frac{4h_x^5 h_y}{270} f^{(4)}(\mu_2,y_2) \tag{4.34}$$

$$- \frac{h_x^5 h_y}{270} f^{(4)}(\mu_3,y_3) \tag{4.35}$$

$$- \frac{h_y^5}{90}\int_a^b f^{(4)}(x,\eta)\,dx \tag{4.36}$$

com $h_x = (b-a)/2$, $\mu_1,\mu_2,\mu_3 \in (a,b)$. Pelos Teorema do Valor Intermediário e Teorema do Valor Médio, podemos ver que o erro é $O(h_x h_y^5 + h_x^5 h_y)$. Por fim, obtemos a **Regra de Simpson para Integrais Iteradas**

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \frac{h_x h_y}{9}\left[f(x_1,y_1) + 4f(x_2,y_1) + f(x_3,y_1)\right] \tag{4.37}$$

$$+ \frac{4h_x h_y}{9}\left[f(x_1,y_2) + 4f(x_2,y_2) + f(x_3,y_2)\right] \tag{4.38}$$

$$+ \frac{h_x h_y}{9}\left[f(x_1,y_3) + 4f(x_2,y_3) + f(x_3,y_3)\right] \tag{4.39}$$

$$+ O(h_x h_y^5 + h_x^5 h_y). \tag{4.40}$$

Exemplo 4.2.2. A Regra de Simpson fornece

$$\int_{1.5}^{2}\int_{1}^{1.5} \ln(x+2y)\,dy\,dx \approx 0.361003. \tag{4.41}$$

Verifique!

4.2.2 Regras Compostas de Newton-Cotes

A ideia é particionar a região de integração em células e o resultado da integração é a soma da aplicação da regra de quadratura em cada uma das células.

Regra Composta do Trapézio

Para uma região retangular $R = [a,b] \times [c,d]$, vamos construir a malha

$$M = \{c_k = [x_i,x_{i+1}] \times [y_j,y_{j+1}] :\ k = i + (j-1)n_x\}, \tag{4.42}$$

onde $x_i = (i-1)h_x$, $h_x = (b-a)/n_x$, $i = 1,2,\ldots,n_x+1$ e $y_j = (j-1)h_y$, $h_y = (d-c)/n_y$, $j = 1,2,\ldots,n_y+1$. Consulte a Figura 4.1.

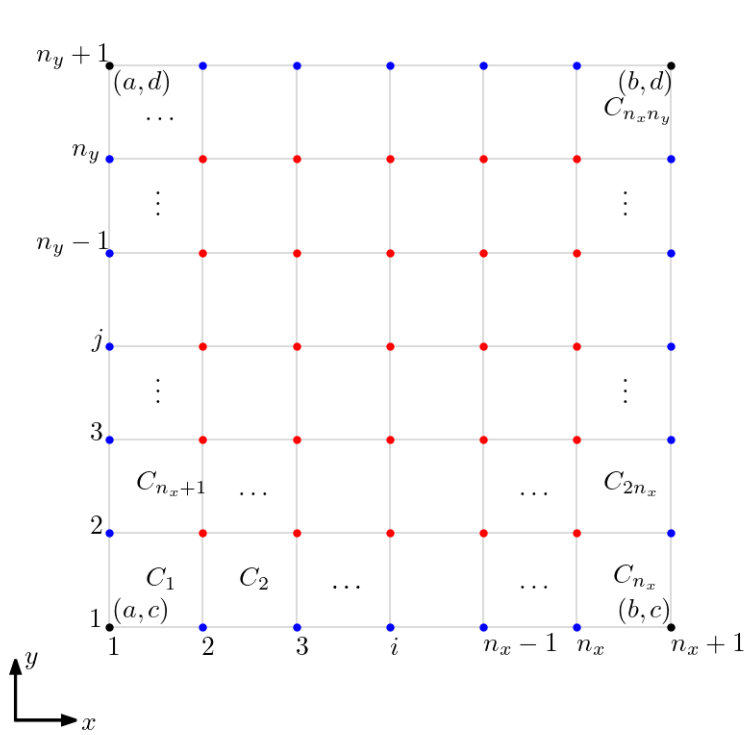

Figura 4.1: Representação da malha para a Regra Composta do Trapézio.

Aplicando a ideia, temos

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \sum_{k=1}^{n_xn_y} \int\int_{C_k} f(x,y)\,dy\,dx \tag{4.43}$$

$$= \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} \int_{x_i}^{x_{i+1}} \int_{y_j}^{y_{j+1}} f(x,y)\,dy\,dx \tag{4.44}$$

Em cada integral em C_k, aplicamos a Regra do Trapézio, segue

$$\int_{x_i}^{x_{i+1}} \int_{y_j}^{y_{j+1}} f(x,y)\,dy\,dx \approx \frac{h_y}{2}\frac{h_x}{2}\,[f(x_i,y_j) + f(x_{i+1},y_j) \tag{4.45}$$

$$+ f(x_{i+1},y_{j+1}) + f(x_{i+1},y_j)] \tag{4.46}$$

Observamos que nos conjuntos de nodos (marcados em azul na Figura 4.1)

$$\{(i,j) :\ i = 2,\ldots,n_x, j = 1 \text{ ou } j = n_y+1\}, \tag{4.47}$$

$$\{(i,j):\ i=1 \text{ ou } i=n_x+1, j=2,\ldots,n_y\} \tag{4.48}$$

a função integranda será avaliada 2 vezes. Já, em todos os nodos internos, $i=2,\ldots,n_x$, $j=2,\ldots,n_y$, a função será avaliada 4 vezes. Com isso, chegamos à Regra Composta do Trapézio

$$\begin{aligned}\int_a^b\int_c^d f(x,y)\,dy\,dx = \frac{h_xh_y}{4}\,&[f(x_1,y_1)+f(x_{n_x+1},y_1)\\ &+f(x_{n_x+1},y_{n_y+1})+f(x_1,y_{n_y+1})\Big]\\ &+\frac{h_xh_y}{2}\sum_{i=2}^{n_x}\Big[f(x_i,y_1)+f(x_i,y_{n_y+1})\Big]\\ &+\frac{h_xh_y}{2}\sum_{j=2}^{n_y}[f(x_1,y_j)+f(x_{n_x+1},y_j)]\\ &+h_xh_y\sum_{i=2}^{n_x}\sum_{j=2}^{n_y}f(x_i,y_j)\\ &+O(h_x^2+h_y^2)\end{aligned} \tag{4.49}$$

Observamos que esta é uma quadratura de $(n_x+1)(n_y+1)$ nodos.

Exercício 4.2.1. Verifique a aplicação da Regra Composta do Trapézio para computar

$$\int_{1.5}^{2}\int_{1}^{1.5}\ln(x+2y)\,dy\,dx. \tag{4.50}$$

Regra Composta de Simpson

Aqui, vamos construir uma malha

$$M=\{C_k=[x_i,x_{i+2}]\times[y_j,y_{j+2}]:\ i=1,3,\ldots,n_x-1, j=1,3,\ldots,n_y-1,\},\tag{4.51}$$

onde $x_i=(i-1)h_x$, $h_x=(b-a)/n_x$, $i=1,2,\ldots,n_x+1$ e $y_j=(j-1)h_y$, $h_y=(d-c)/n_y$, $j=1,2,\ldots,n_y+1$. Com $n_x,n_y\leq 2$ números pares. Consulte a Figura 4.2.

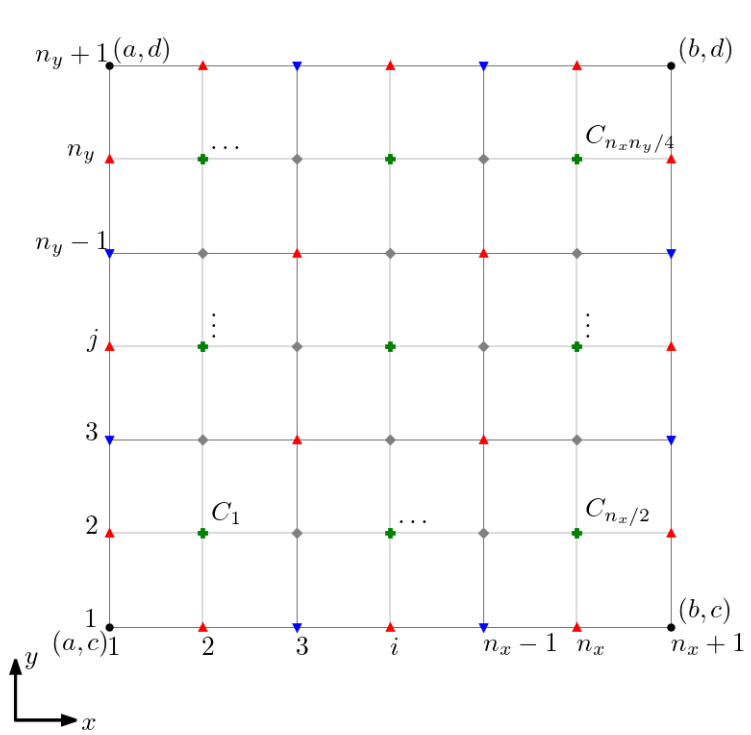

Figura 4.2: Representação da malha para a Regra Composta de Simpson.

A Regra Composta de Simpson para Integrais Iteradas fica

$$\begin{aligned}\int_a^b \int_c^d f(x,y)\,dx\,dy = \frac{h_x h_y}{9} \{ & f(x_1,y_1) + f(x_{n_x+1},y_1) \\ & + f(x_1,y_{n_y+1}) + f(x_{n_x+1},y_{n_y+1}) \\ & + 2 \sum_{i=1}^{n_x/2-1} \left[f(x_{2i+1},y_1) + f(x_{2i+1},y_{n_y+1}) \right] \\ & + 2 \sum_{j=1}^{n_y/2-1} \left[f(x_1,y_{2j+1}) + f(x_{n_x+1},y_{2j+1}) \right] \\ & + 4 \sum_{i=1}^{n_x/2} \left[f(x_{2i},y_1) + f(x_{2i},y_{n_y+1}) \right] \\ & + 4 \sum_{j=1}^{n_y/2} \left[f(x_1,y_{2j}) + f(x_{n_x+1},y_{2j}) \right]\end{aligned}$$

$$
\begin{aligned}
&+ 4 \sum_{i=1}^{n_x/2-1} \sum_{j=1}^{n_y/2-1} f(x_{2j+1}, y_{2j+1}) \\
&+ 8 \sum_{i=1}^{n_x/2-1} \sum_{j=1}^{n_y/2} f(x_{2i+1}, y_{2j}) \\
&+ 8 \sum_{i=1}^{n_x/2} \sum_{j=1}^{n_y/2-1} f(x_{2i}, y_{2j+1}) \\
&+ 16 \sum_{i=1}^{n_x/2} \sum_{j=1}^{n_y/2} f(x_{2i}, y_{2j}) \Bigg\} \\
&+ O(h_x^4 + h_y^4).
\end{aligned} \tag{4.52}
$$

Exercício 4.2.2. Verifique a aplicação da Regra Composta de Simpson para computar

$$\int_{1.5}^{2} \int_{1}^{1.5} \ln(x + 2y)\, dy\, dx. \tag{4.53}$$

Bibliografia

[1] R. Burden and J. Faires. *Análise Numérica*. Cengage Learning, 3. edition, 2016.

[2] J. Davis and P. Rabinowitz. *Methods of Numerical Integration*. Academic Press, Inc., San Diego, 2. edition, 1984.

[3] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 4. edition, 2013.

[4] C. Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[5] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2. edition, 2006.

[6] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer, Berlin, 2. edition, 2007.

[7] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, 2003.

[8] J. Stoer. *Approximate Calculation of Multiple Integrals*. Prentice-Hall, Inc., 1971.

[9] D. Watkins. *Fundamentals of Matrix Computations*. John Wiley, New York, 2002.