Matemática Numérica Avançada

Pedro H A Konzen

23 de março de 2022

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

Sumário

Capa i

Licença ii

Prefácio iii

Sumário iv

1 Sistemas Lineares 1
- 1.1 Matrizes Esparsas . . . 1
  - 1.1.1 Sistemas Tridiagonais . . . 3
  - 1.1.2 Matrizes Banda . . . 6
  - 1.1.3 Esquemas de Armazenamento . . . 11
- 1.2 Métodos Iterativos . . . 16
  - 1.2.1 GMRES . . . 16
  - 1.2.2 Método do Gradiente Conjugado . . . 21
  - 1.2.3 Precondicionamento . . . 27

2 Sistemas Não Lineares e Otimização 30
- 2.1 Método de Newton . . . 30
- 2.2 Método Tipo Newton . . . 33
  - 2.2.1 Atualização Cíclica da Matriz Jacobiana . . . 34

Referências Bibliográficas 36

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.

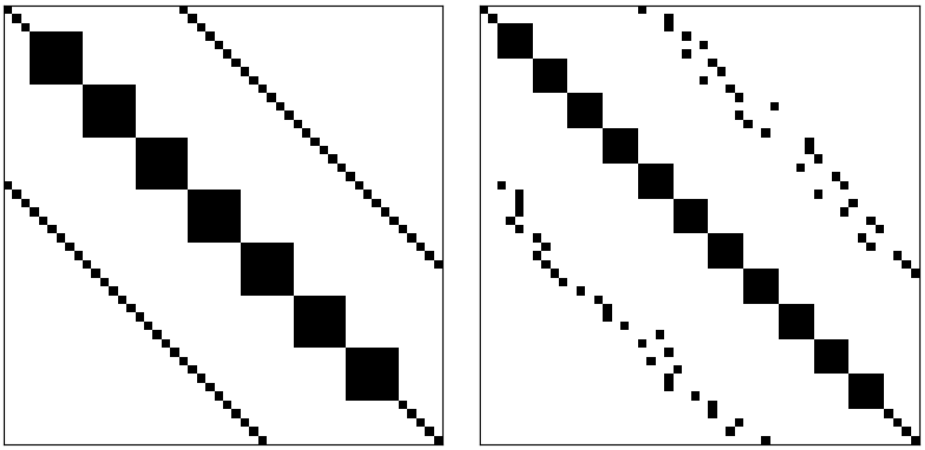

Figura 1.1: Esquerda: exemplo de uma matriz esparsa estruturada. Direita: exemplo de uma matriz esparsa não-estruturada.

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

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} \tag{1.3}$$

com, $a_1 = 0$ e $c_n = 0$. Ou seja, é um sistema cuja a matriz dos coeficientes é tridiagonal.

Uma **matriz tridiagonal** é uma matriz esparsa estruturada. Mais especificamente, é chamada de 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

$$a = (0,a_2,\ldots,a_n) \tag{1.4}$$
$$b = (b_1,b_2,\ldots,b_n) \tag{1.5}$$
$$c = (c_1,c_2,\ldots,c_{n-1},0) \tag{1.6}$$

Ou seja, precisamos armazenar $3n$ pontos flutuantes em vez de n^2, como seria o caso se a matriz dos coeficientes fosse densa.

Algoritmo de Thomas

O Algoritmo de Thomas[1] é uma forma otimizada do Método de Eliminação Gaussiana 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

$$\begin{bmatrix} \tilde{b}_1 & c_1 & & & 0 \\ & \tilde{b}_2 & c_2 & & \\ & & \tilde{b}_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & & \tilde{b}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \tilde{d}_1 \\ \tilde{d}_2 \\ \tilde{d}_3 \\ \vdots \\ \tilde{d}_n \end{bmatrix} \tag{1.7}$$

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

Este é obtido pela seguinte iteração

$$w := \frac{a_i}{b_{i-1}} \tag{1.8}$$

$$b_i := b_i - wc_{i-1} \tag{1.9}$$

$$d_i := d_i - wd_{i-1} \tag{1.10}$$

onde, o ˜ foi esquecido de propósito, indicando a reutilização dos vetores b e d. A solução do sistema é, então, obtida de baixo para cima, i.e.

$$x_n = \frac{d_n}{b_n} \tag{1.11}$$

$$x_i = \frac{d_i - c_i x_{i+1}}{b_i}, \tag{1.12}$$

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

Listing 1.1: Algoritmo de Thomas

```
1  import numpy as np
2
3  def TDMA(a,b,c,d):
4      n = b.size
5      for i in np.arange(1,n):
6        w = a[i]/b[i-1]
7        b[i] = b[i] - w*c[i-1]
8        d[i] = d[i] - w*d[i-1]
9      x = np.empty(n)
10     x[n-1] = d[n-1]/b[n-1]
11     for i in np.arange(n-2,-1,-1):
12       x[i] = (d[i] - c[i]*x[i+1])/b[i]
13     return x
```

Exercício 1.1.1. Considere o seguinte sistema linear

$$2x_1 - x_2 = 0 \tag{1.13}$$

$$x_{i-1} - 3x_i + 4x_{i+1} = \operatorname{sen}\left(i\frac{\pi}{2(n-1)}\right) \tag{1.14}$$

$$x_{n-1} + x_n = 1 \tag{1.15}$$

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 timeit 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)

$$-u'' = \operatorname{sen} \pi x, \quad 0 < x < 1, \tag{1.16}$$
$$u(0) = 0, \tag{1.17}$$
$$u(1) = 0 \tag{1.18}$$

seja simulado com o Método das Diferenças Finitas[2]. Vamos assumir uma discretização espacial uniforme com n nodos e tamanho de malha

$$h = \frac{1}{n-1}. \tag{1.19}$$

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

$$u''(x_i) \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2}, \tag{1.20}$$

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

$$-\frac{1}{h^2}u_{i-1} + \frac{2}{h^2}u_i - \frac{1}{h^2}u_{i+1} = \operatorname{sen} \pi x_i \tag{1.21}$$

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

$$u_1 = 0 \tag{1.22}$$
$$-\frac{1}{h^2}u_{i-1} + \frac{2}{h^2}u_i - \frac{1}{h^2}u_{i+1} = \operatorname{sen} \pi x_i \tag{1.23}$$
$$u_n = 0 \tag{1.24}$$

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

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.

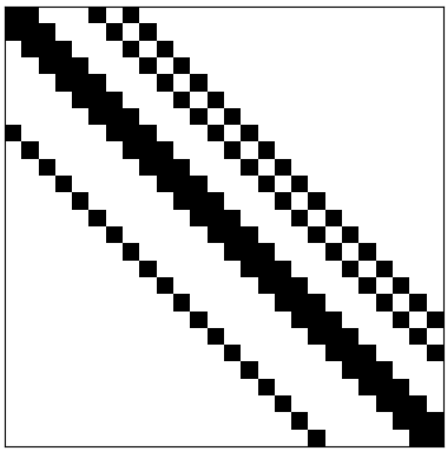

Figura 1.2: Exemplo de uma matriz banda.

Exemplo 1.1.1. Consideremos o seguinte problema de Poisson[3]

$$-\Delta u = f(x,y),\ (x,y) \in (0,\pi) \times (0,\pi), \tag{1.25}$$
$$u(0,y) = 0,\ y \in [0,\pi], \tag{1.26}$$
$$u(\pi,y) = 0,\ y \in [0,\pi], \tag{1.27}$$
$$u(x,0) = 0,\ x \in [0,\pi], \tag{1.28}$$
$$u(x,\pi) = 0,\ x \in [0,\pi]. \tag{1.29}$$

Para fixarmos as ideias, vamos assumir

$$f(x,y) = \operatorname{sen}(x)\operatorname{sen}(y) \tag{1.30}$$

Vamos empregar o Método de Diferenças Finitas[4] para computar uma aproximação para a sua solução. Começamos assumindo uma malha uniforme de n^2 nodos

$$x_i = (i-1)h \tag{1.31}$$
$$y_j = (j-1)h \tag{1.32}$$

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[5] encontramos o seguinte problema discreto associado

$$u_{i,1} = u_{1,j} = 0 \tag{1.33}$$

$$\begin{aligned} &-\frac{1}{h^2}u_{i-1,j} - \frac{1}{h^2}u_{i,j-1} + \frac{4}{h^2}u_{i,j} \\ &-\frac{1}{h^2}u_{i+1,j} - \frac{1}{h^2}u_{i,j+1} = f(x_i,y_j) \end{aligned} \tag{1.34}$$

$$u_{i,n} = u_{n,j} = 0 \tag{1.35}$$

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

$$Aw = b \tag{1.36}$$

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

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

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

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

$$u_{i,j} = w_{k=i-1+(j-2)(n-2)}, \quad i,j = 2,\ldots,n-2 \tag{1.37}$$

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

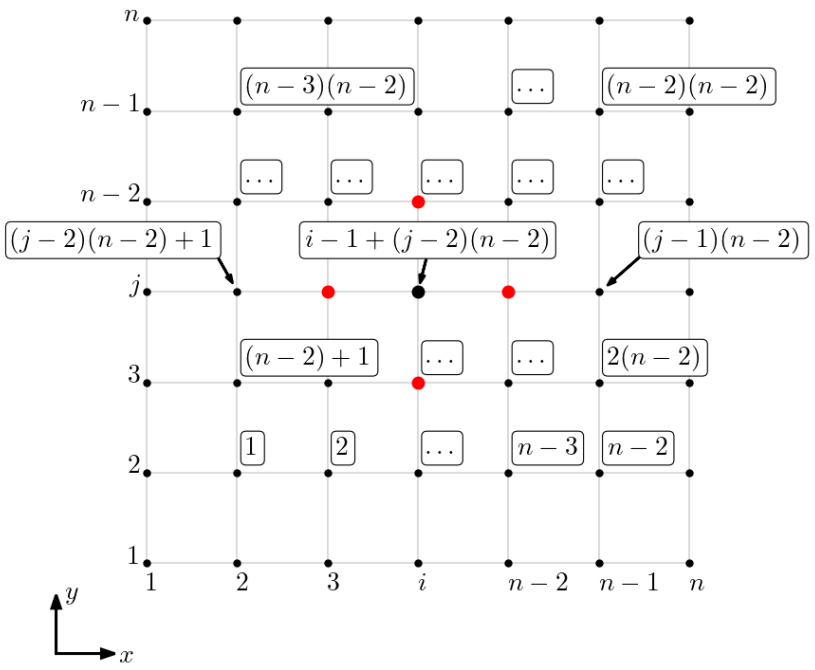

Figura 1.3: Representação da enumeração das incógnitas referente ao problema discutido no Exercício 1.1.1.

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

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

$$4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.38}$$

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

$$-w_{k-1} + 4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.39}$$

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

$$-w_{k-1} + 4w_k - w_{k+n-2} = h^2 f_{i,j} \tag{1.40}$$

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

$$-w_{k-(n-2)} + 4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.41}$$

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

$$-w_{k-1} - w_{k-(n-2)} + 4w_k - w_{k+1} - w_{k+n-2} = h^2 f_{i,j} \tag{1.42}$$

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

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

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

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

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

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

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

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

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

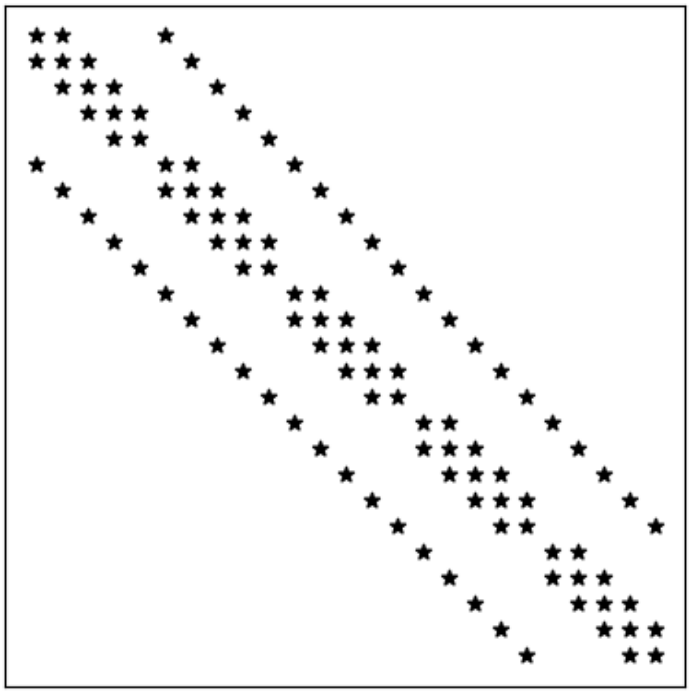

Figura 1.4: Representação da matriz do sistema discreto construído no Exemplo 1.1.1.

Exercício 1.1.3. Consideremos o problema trabalho no Exemplo 1.1.1.

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

$$u(x,y) = \frac{1}{2}\,\mathrm{sen}(x)\,\mathrm{sen}(y). \tag{1.47}$$

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 1.1.2. Consideremos a seguinte matriz

$$A = \begin{bmatrix} 2. & 0. & 1. & 0. \\ 0. & 3. & 2. & -1. \\ 0 & -1 & -2. & 0. \\ 0 & 0 & 0 & 1. \end{bmatrix} \tag{1.48}$$

O formato COO está disponível na biblioteca SciPy com o método scipy.sparse.coo_matrix[6]. 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())
```

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

- Vantagens do formato COO são:
  - permite a entrada de dados duplicados (simplicidade);
  - possível conversão rápida entre os formatos CSR e CSC[7].
- 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 1.1.3. No Exemplo 1.1.2, alocamos a matriz

$$A = \begin{bmatrix} 2. & 0. & 1. & 0. \\ 0. & 3. & 2. & -1. \\ 0 & -1 & -2. & 0. \\ 0 & 0 & 0 & 1. \end{bmatrix} \tag{1.49}$$

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

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

```
1      from scipy.sparse import csr_matrix
2      csr = coo.tocsr()
3      d = csr.data
4      c = csr.indices
5      p = csr.indptr
6      print(d)
7      print(c)
8      print(p)
```

Para fixarmos as ideias, temos

$$d = (2., 1., 3., 2., -1., -1., -2., 1.) \tag{1.50}$$

$$c = (0, 2, 1, 2, 3, 1, 2, 3) \tag{1.51}$$

$$p = (0, 2, 5, 7, 8) \tag{1.52}$$

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 1.1.4. No Exemplo 1.1.2, alocamos a matriz

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

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

$$d = (2., 3., -1., 1., 2., -2., -1., 1.) \tag{1.54}$$
$$l = (0, 1, 2, 0, 1, 2, 1, 3) \tag{1.55}$$
$$p = (01368) \tag{1.56}$$

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[8].

- 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 [4, Seção 3.4] e da documentação do scipy.sparse.

Exercício 1.1.4. Considere o problema de Poisson dado no Exemplo 1.1.1.

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

b) Converta a matriz armazenada para o formato CSR[9]. Então, compute a solução do problema discreto com o método spsolve[10].

c) Converta a matriz armazenada para o formato CSC[11]. 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 $h = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$.

[8] CSR é mais eficiente em muitos casos.
[9] Use o método coo_matrix.tocsr().
[10] `scipy.sparse.linalg.spsolve` é uma implementação do Método LU otimizado para matrizes esparsas.
[11] Use o método coo_matrix.tocsc().

1.2 Métodos Iterativos

1.2.1 GMRES

O GMRES (do inglês, *Generalized Minimal Residual Method*[12]) é um Método de Subespaço de Krylov e é considerado uma das mais eficientes técnicas 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

$$Ax = b \tag{1.57}$$

por um **método de projeção**. Mais especificamente, busca-se uma solução aproximada $x_m \in \mathbb{R}^{\kappa}$ no subespaço afim $x_0 + \mathcal{K}_m$ de dimensão m, impondo-se a **condição de Petrov[13]-Galerkin[14]**

$$b - Ax_m \perp \mathcal{L}_m, \tag{1.58}$$

onde $\mathcal{L}_m$ também é um subespaço de dimensão m. Quando $\mathcal{K}_m$ é um **subespaço de Krylov[15]**, i.e.

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}, \tag{1.59}$$

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

$$r_0 = b - Ax_0, \tag{1.60}$$

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

$$A^{-1}b \approx x_m = x_0 + q_{m-1}(A)r_0, \tag{1.61}$$

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

$$A^{-1}b \approx q_{m-1}(A)x_0. \tag{1.62}$$

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

[12] Desenvolvido por Yousef Saad e H. Schultz, 1986. Fonte: Wikipedia.
[13] Georgi Iwanowitsch Petrov, 1912 - 1987, engenheiro soviético. Fonte: Wikipedia.
[14] Boris Galerkin, 1871 - 1945, engenheiro e matemático soviético. Fonte: Wikipédia.
[15] 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

$$\mathcal{K}_m = \mathcal{K}_m(A,v_1) = \text{span}\{v_1, Av_1, \ldots, A^{m-1}v_1\}, \tag{1.63}$$

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

$$x = x_0 + V_m y \tag{1.64}$$

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[16]- Gram[17]-Schmidt[18] Modificado** [4, 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[19] cujas entradas não nulas são computadas pelo algoritmo acima (Passos 2(a)i-ii). Pode-se

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

mostrar[20] que

$$J(y) = \|b - Ax\| \tag{1.65}$$
$$= \|b - A(x_0 + V_m y)\| \tag{1.66}$$
$$= \|\beta e_1 - \bar{H}_m y\| \tag{1.67}$$

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

A aproximação GMRES x_m é então computada como

$$x_m = x_0 + V_m y_m, \tag{1.68}$$
$$y_m = \min_y \|\beta e_1 - \bar{H}_m y\| \tag{1.69}$$

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**[21] é utilizada. Consulte mais em [4, Subsetion 6.5.2].

[20] Consulte [4, Proposição 6.5].

[21] 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

$$y_m = \min_y \|\beta e_1 - \hat{H}_m y\| \tag{1.70}$$

$$x_m = x_0 + V_m y_m \tag{1.71}$$

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 1.1.1.

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[22] com condições de contorno não homogêneas.

$$-\Delta u = f(x,y),\ (x,y) \in D, \tag{1.72}$$
$$u = g, \quad \text{em } \partial D \tag{1.73}$$

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

$$f(x,y) = 2\pi^2 \operatorname{sen} \pi(x+y) \tag{1.74}$$

e os valores no contorno

$$g = \operatorname{sen} \pi(x+y), \quad (x,y) \in \partial D. \tag{1.75}$$

Observamos que a solução analítica deste problema é

$$u(x,y) = \operatorname{sen} \pi(x+y). \tag{1.76}$$

Vamos empregar o Método de Diferenças Finitas[23] para computar uma aproximação para a solução. Assumimos uma malha uniforme de n^2 nodos

$$x_i = (i-1)h \tag{1.77}$$
$$y_j = (j-1)h \tag{1.78}$$

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[24] encontramos o seguinte problema

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

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

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

discreto associado

$$u_{i,1} = g(x_i, 0) \tag{1.79}$$
$$u_{1,j} = g(0, y_j) \tag{1.80}$$

$$\begin{aligned} &-\frac{1}{h^2}u_{i-1,j} - \frac{1}{h^2}u_{i,j-1} + \frac{4}{h^2}u_{i,j} \\ &-\frac{1}{h^2}u_{i+1,j} - \frac{1}{h^2}u_{i,j+1} = f(x_i, y_j) \end{aligned} \tag{1.81}$$

$$u_{i,n} = g(x_i, 1) \tag{1.82}$$
$$u_{n,j} = g(1, y_j) \tag{1.83}$$

Este pode ser escrito na forma matricial

$$Aw = b \tag{1.84}$$

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

$$u_{i,j} = w_{k=i-1+(j-2)(n-2)}, \quad i,j = 2, \ldots, n-2. \tag{1.85}$$

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

$$Ax = b \tag{1.86}$$

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

O método pode ser derivado a partir do Método de Arnoldi[25] [4, 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

$$J(y) = \frac{1}{2} y^T A y - y^T b. \tag{1.87}$$

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

$$J(y) = \frac{1}{2} y^T A y - y^T b + \frac{1}{2} x^T A x - \frac{1}{2} x^T A x \tag{1.88}$$

$$= \frac{1}{2} (y - x)^T A (y - x) - \frac{1}{2} x^T A x \tag{1.89}$$

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

$$\frac{1}{2} (y - x)^T A (y - x) \tag{1.90}$$

é minimizado. Agora, como A é definida positiva[26], o menor valor deste termo ocorre quando $y - x = 0$, i.e. $y = x$.

Observamos, também, que o gradiente de J é

$$\nabla J = Ay - b \tag{1.91}$$

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

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

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

direção e sentido de maior crescimento de J. Isso nos motiva a aplicarmos a seguinte iteração[27]

$$x^{(0)} = \text{aprox. inicial} \tag{1.92}$$

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla J\left(x^{(k)}\right) \tag{1.93}$$

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

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)}. \tag{1.94}$$

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[28]

$$J\left(x^{(k+1)}\right) = \min_{\alpha \in \mathbb{R}} J\left(x^{(k)} + \alpha r^{(k)}\right) \tag{1.95}$$

A fim de resolver este problema de minimização, vamos denotar

$$g(\alpha) = J\left(x^{(k)} + \alpha r^{(k)}\right). \tag{1.96}$$

Então, observamos que

$$\begin{aligned} g(\alpha) &= \frac{1}{2}\left(x^{(k)} + \alpha r^{(k)}\right)^T A\left(x^{(k)} + \alpha r^{(k)}\right) - \left(x^{(k)} + \alpha r^{(k)}\right)^T b \\ &= \frac{1}{2} x^{(k)^T} A x^{(k)} + \frac{\alpha}{2} x^{(k)^T} A r^{(k)} + \frac{\alpha}{2} r^{(k)^T} A x^{(k)} \\ &+ \frac{\alpha^2}{2} r^{(k)^T} A r^{(k)} - x^{(k)^T} b - \alpha r^{(k)^T} b \end{aligned} \tag{1.97}$$

Agora, usando o fato de A ser simétrica, obtemos

$$g(\alpha) = J\left(x^{(k)}\right) + \alpha r^{(k)^T} A x^{(k)} + \frac{\alpha^2}{2} r^{(k)^T} A r^{(k)} - \alpha r^{(k)^T} b \tag{1.98}$$

$$= J\left(x^{(k)}\right) - \alpha r^{(k)^T} r^{(k)} + \frac{\alpha^2}{2} r^{(k)^T} A r^{(k)} \tag{1.99}$$

[27] Iteração do Método do Máximo Declive.

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

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

$$0 = g'(\alpha) \tag{1.100}$$
$$= -r^{(k)^T} r^{(k)} + \alpha r^{(k)^T} b. \tag{1.101}$$

Logo, encontramos

$$\alpha = \frac{r^{(k)^T} r^{(k)}}{r^{(k)^T} A r^{(k)}} \tag{1.102}$$

Com isso, temos a iteração do Método do Gradiente

$$x^{(0)} = \text{aprox. inicial} \tag{1.103}$$
$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)}, \tag{1.104}$$
$$\alpha_k = \frac{r^{(k)^T} r^{(k)}}{r^{(k)^T} A r^{(k)}} \tag{1.105}$$

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

$$r^{(k+1)} = r^{(k)} - \alpha_k A r^{(k)}. \tag{1.106}$$

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 1.1.1 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[29] e do Método LU[30] nos itens anteriores.

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

$$x_0 = \text{aprox. inicial}, x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}, \tag{1.107}$$

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[31]

$$J(y) = \frac{1}{2} \langle y,y \rangle_A - \langle b,y \rangle, \tag{1.108}$$

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

$$\langle x,y \rangle_A := x^T A y \tag{1.109}$$

é o **produto interno induzido por** A, lembrando que A é positiva definida[32]. Associada a este produto interno, temos a norma

$$\|x\|_A := \sqrt{\langle x,x \rangle_A}, \tag{1.110}$$

chamada de **norma da energia**. O produto interno associado é também conhecido como **produto interno da energia**. Com isso, definimos que

[29] scipy.sparse.linalg.gmres
[30] scipy.sparse.linalg.spsolve
[31] Compare com o funcional J dado em (1.87).
[32] Mostre que $\langle \cdot,\cdot \rangle_A$ é de fato um produto interno.

dois vetores x e y são **conjugados**, quando eles são ortogonais com respeito ao produto interno da energia, i.e. quando

$$\langle x,y\rangle_A = 0. \tag{1.111}$$

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 [6, Seção 7.7].

Listing 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 1.1.1 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[33] e da implementação SciPy do Método do Gradiente Conjugado[34]

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

$$Ax = b \tag{1.112}$$

resolvemos o sistema equivalente

$$MAx = Mb, \tag{1.113}$$

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

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

adequadamente removidas, buscando-se uma boa esparsidade e ao mesmo tempo uma boa aproximação LU para A.

ILU(0)

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

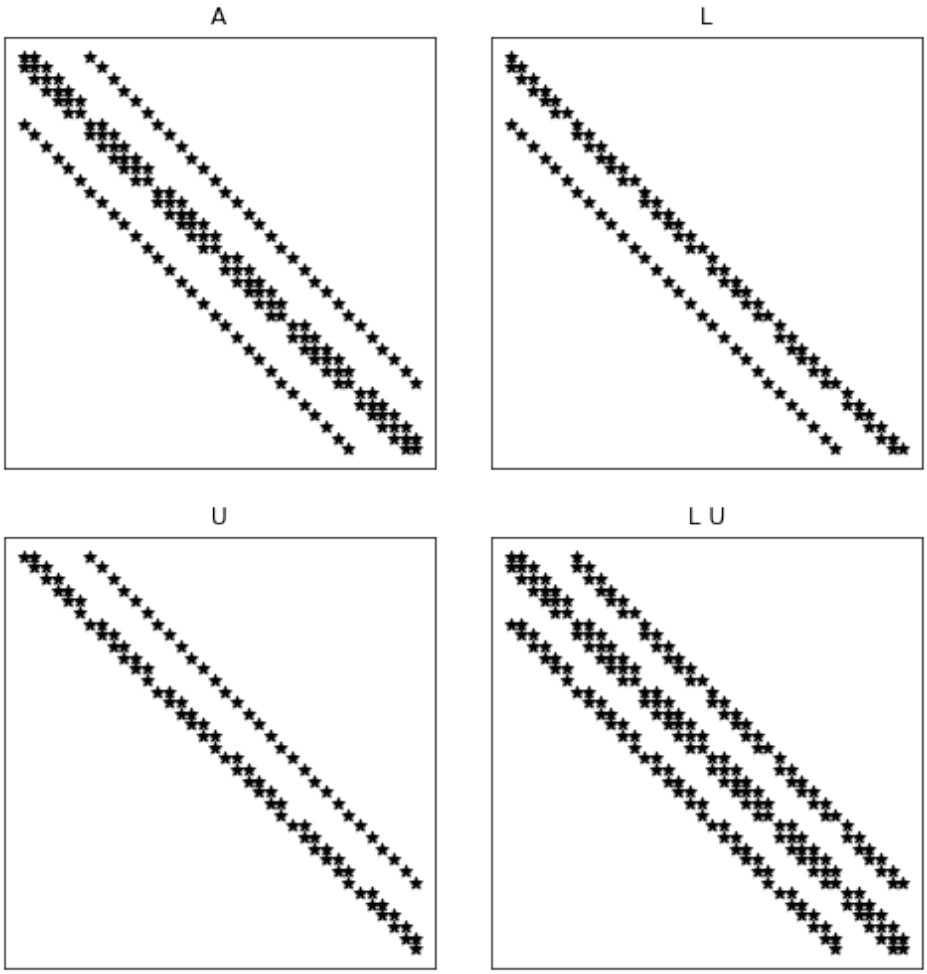

Figura 1.5: Representação das matrizes ILU(0).

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*.

Listing 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[3] contínua. Consulte [3, Seção 7.1] para mais detalhes.

Exemplo 2.1.1. Consideremos a **Equação de Burgers**[4]

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

com $\nu > 0$, condição inicial

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

e condições de contorno de Dirichlet[5] homogêneas

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

Aplicando o **Método de Rothe**[6] com aproximação de Euler[7] implícita, obtemos

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

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

[3]Rudolf Otto Sigismund Lipschitz, 1832 - 1903, matemático alemão. Fonte: Wikipédia.

[4]Johannes Martinus Burgers, 1895 - 1981, físico holandês. Fonte: Wikipedia.

[5]Johann Peter Gustav Lejeune Dirichlet, 1805 - 1859, matemático alemão. Fonte: Wikipédia.

[6]Erich Hans Rothe, 1895 - 1988, matemático alemão. Fonte: Wikipédia.

[7]Leonhard Paul Euler, 1707 - 1783, matemático e físico suíço. Fonte: Wikipédia.

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

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

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

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[8] que buscam reduzir o custo computacional. Há estratégias para simplificar as computações da matriz jacobiana[9] e para reduzir o custo nas computações das atualizações de Newton.

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

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

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 (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.

Referências Bibliográficas

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

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

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

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

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

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