CAPÍTULO

1 Sistemas de Equações Lineares

Muitos problemas da engenharia, física e matemática estão associados à solução de sistemas de equações lineares.

Dois exemplos bem ilustrativos e extremamente importantes são a tomografia computadorizada, que faz uso de sistemas de equações lineares para estimar, para cada pixel da imagem obtida no tomógrafo, a densidade do anteparo (no caso, paciente), descobrindo, assim, se esse anteparo tem material mais líquido, mais sólido, se trata-se de músculo, tumor, osso e por aí vai (esse artigo tem uma excelente descrição de como obter um sistema de equações lineares para esse caso) e os sistemas de recomendação, como, por exemplo, o usado pelo Netflix, que faz uso desses sistemas de equações considerando como incógnitas cada um dos parâmetros do sistema (esse artigo mostra de forma bem simplória como funciona o sistema de recomendações da Netflix, exemplificando sua implementação com Excel).

Nessa parte de nosso curso, trataremos de técnicas numéricas empregadas para obter a solução desses sistemas. Veremos como definir e classificar sistemas de equações lineares, as soluções para sistemas triangulares, métodos diretos de solução – Eliminação Gaussiana e fatoração LU, uma forma melhorada de solução.

Para o códigos a serem trabalhados nesse capítulo, consideraremos a importação da biblioteca `Numpy` e do submódulo `linalg` da biblioteca `Scipy`, que conterão as funções que implementam os algoritmos que serão discutidos.

In [ ]:
```
import numpy as np
import scipy.linalg as sla
```

1.1 – Introdução à Sistemas de Equações Lineares

Um **sistema de equações lineares** (abreviadamente, sistema linear) é um conjunto finito de equações lineares aplicadas em um mesmo conjunto, igualmente finito, de variáveis reais ou complexas (nos deteremos somente as variáveis reais).

Uma **solução** para um sistema linear é uma atribuição de números às incógnitas que

satisfazem simultaneamente todas as equações do sistema. A palavra sistema, nesse contexto, indica que as equações devem ser consideradas em conjunto, e não de forma individual, isto é, cada uma das equações precisa ser satisfeita pelo conjunto de soluções encontradas.

Tipicamente, sistemas de equações são compostos por m equações lineares, cada uma delas com n incógnitas, comumente descritas da forma

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n &= b_3 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{1.1}$$

Esse formato é chamado de *forma algébrica* do sistema de equações lineares. Esse mesmo sistema pode ser representado na sua forma matricial

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{1.2}$$

onde:

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix} \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{pmatrix}, \tag{1.3}$$

sendo $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ (isto é, com m linhas e n colunas) a *matriz dos coeficientes*, cujos valores a_{ij}, com $1 \leqslant i \leqslant m$ e $1 \leqslant j \leqslant n$ são os coeficientes das equações, $\boldsymbol{x} \in \mathbb{R}^n$ o *vetor das incógnitas ou vetor solução* e $\boldsymbol{b} \in \mathbb{R}^m$ o *vetor dos termos independentes*.

Exemplo 1.1

Consideremos o seguinte sistema linear

$$\begin{aligned} x + y + z &= 1 \\ 4x + 4y + 2z &= 2 \\ 2x + y - z &= 0. \end{aligned} \tag{1.4}$$

Escreva ela no formato matricial.

Na sua forma matricial, este sistema é escrito como

$$\boldsymbol{Ax} = \boldsymbol{b} \quad \Rightarrow \quad \underbrace{\begin{pmatrix} 1 & 1 & 1 \\ 4 & 4 & 2 \\ 2 & 1 & -1 \end{pmatrix}}_{\boldsymbol{A}} \cdot \underbrace{\begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}}_{\boldsymbol{b}} . \tag{1.5}$$

1.2 – Classificação de Sistemas de Equações Lineares

A solução de um sistema linear é a atribuição de valores às variáveis $x_1, x_2, \ldots, x_n$ de modo a satisfazer todas as equações simultâneamente. O grupo de todas as soluções possíveis é chamado de conjunto-solução.

Dependendo da relação entre o número de linhas m e o número de colunas n, podemos classificar os sistemas lineares em três tipos:

1. **sistemas lineares sobredeterminados:** nos quais há mais equações que incógnitas ($m > n$);
2. **sistemas lineares determinados:** em que o número de equações é igual ao de incógnitas ($m = n$, o que significa matrizes de coeficientes quadradas);
3. **sistemas lineares subdeterminados:** em que há mais incógitas que equações ($m < n$).

Podemos também classificar os sistemas com relação a existência e quantidade de soluções. Para estebalecer esse critério de classificação, considere a definição da *matriz estendida* $\widetilde{\boldsymbol{A}} = [\boldsymbol{A}|\boldsymbol{b}]$ do sistema $\boldsymbol{Ax} = \boldsymbol{b}$, que é basicamente uma matriz criada com a inserção do vetor $\boldsymbol{b}$ à matriz $\boldsymbol{A}$.

$$\widetilde{\boldsymbol{A}} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} & b_3 \\ \vdots & \vdots & \ddots & \vdots & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} & b_m \end{pmatrix} \tag{1.6}$$

1. **Sistema Possível Determinado:** é o sistema que possui apenas uma única solução possível. Esse sistema pode ser chamado de sistema possível e suas equações de equações compatíveis. Para esse sistema, $rank(\widetilde{\boldsymbol{A}}) = rank(\boldsymbol{A}) = n$, sendo $rank(\cdot)$ o posto de uma matriz[1].
2. **Sistema Possível Indeterminado:** é o sistema que possui infinitas soluções, quando $rank(\widetilde{\boldsymbol{A}}) = rank(\boldsymbol{A}) < n$. Um caso especial é o *sistema possível homogêneo*, ocorrido quando $\boldsymbol{b} = 0$ e, portanto, tem sempre $x = 0$ como uma possível solução.

[1] Se não lembra a definição do posto de uma matriz, dá uma olhada no link.

3. **Sistema Impossível:** é o sistema que não admite uma solução precisa, sendo designado por sistema impossível e suas equações como equações incompatíveis. Para esse sistema, $rank(\widetilde{\boldsymbol{A}}) > rank(\boldsymbol{A})$ e a única forma de encontrar solução aproximada é usando o método dos minimos quadrados, que veremos mais a frente.

Essa classificação pode ser feita com a ajuda dos métodos `c_`, cuja documentação está nesse link e `matrix_rank`, cuja documentação se encontra nesse link, ambos da biblioteca `Numpy`, sendo o segundo presente no submódulo `linalg` (que é diferente do submódulo de mesmo nome da `Scipy`!).

O método `c_` concatena `arrays` ou pedaços deles ao longo do eixo das colunas, isto é, do eixo 1. Por exemplo, considere dois vetores

```
In [ ]: a = np.array([1,2,3])
        b = np.array([4,5,6])
```

O resultado do uso do método `c_` será

```
In [ ]: np.c_[a,b]
```

```
Out[ ]: array([[1, 4],
               [2, 5],
               [3, 6]])
```

que é a concatenação dos dois vetores no eixo das colunas.

Uma coisa é importante frisar: para que o método funcione, é preciso que os vetores tenham o mesmo número de linhas, podendo ter diferente número de colunas, ou um erro aparecerá. Considere, para um exemplo, um outro vetor

```
In [ ]: c = np.array([7,8,9]).reshape(1,3)
```

que é um vetor linha com 3 colunas, diferente de `a`, que tem 3 linhas.

```
In [ ]: print('a: ',a.shape)
        print('c: ',c.shape)
```

```
a: (3,)
c: (1, 3)
```

Se utilizarmos o método `c_` para esses vetores, obteremos um erro

```
In [ ]: np.c_[a,c]
```

```
Out[ ]: ValueError: all the input array dimensions for the
        concatenation axis must match exactly, but along dimension 0,
        the array at index 0 has size 3 and the array at index 1 has
        size 1
```

indicando, justamente, a diferença entre o número de linhas dos dois vetores.

É direto perceber que esse método pode ser usado para a criação da matriz aumentada $\widetilde{\boldsymbol{A}}$. Já o método `matrix_rank` retorna o posto da matriz. O exemplo a seguir ilustra o uso.

Exemplo 2.1

Classificar o sistema linear abaixo, com relação à quantidade e existência de soluções.

$$\begin{aligned} 2x + 3y + 5z &= 10 \\ x - y + 10z &= 20 \\ -x + y - z &= 5 \end{aligned} \tag{1.7}$$

Representando a matriz de coeficientes e o vetor de termos independentes, temos

```
In [ ]: A = np.array([[2,3,5],[1,-1,10],[-1,1,-1]])
        b = np.array([10,20,5])
```

Assim, o $rank(\boldsymbol{A})$ é

```
In [ ]: np.linalg.matrix_rank(A)
Out[ ]: 3
```

o $rank(\widetilde{\boldsymbol{A}})$ é

```
In [ ]: np.linalg.matrix_rank(np.c_[A, b])
Out[ ]: 3
```

e o número de linhas n é

```
In [ ]: A.shape[1]
Out[ ]: 3
```

Assim, como $rank(\widetilde{\boldsymbol{A}}) = rank(\boldsymbol{A}) = n$, então o sistema é possível e determinado.

■ **Exercício 1.1** *Classifique os sistemas abaixo com relação a quantidade e existência de soluções.*

a)

$$\begin{aligned} x + 2y + 3z &= 1 \\ 4x + 5y + 6z &= 1 \\ 7x + 8y + 9z &= 1 \end{aligned} \tag{1.8}$$

b)

$$\begin{aligned} 2x + 3y &= 10 \\ -4x - 6y &= -10 \end{aligned} \tag{1.9}$$

1.3 – Soluções Triviais de Sistemas de Equações Lineares

Durante os anos de estudos básicos (ensino fundamental e médio) nós somos apresentados à sistemas de equações com duas incógnitas, cujas soluções podem ser feitas por substituição: basta isolar uma das incógnitas em uma equação, depois substituir esse incógnita isolada na equação seguinte e encontrar os valores individualmente.

Essa mesma abordagem pode ser utilizada com sistemas de ordem superior à dois, em dois casos especiais: quando a matriz de coeficientes for diagonal ou triangular.

Solução de sistemas lineares com matrix diagonal de coeficientes

Sistemas desse tipo possuem uma matriz de coeficientes diagonal, ou seja, uma matriz quadrada $n \times n$ em que $a_{ij} = 0$ quando $i \neq j$, da forma

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{pmatrix} \tag{1.10}$$

e o sistema, portanto, terá a forma matricial

$$\begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}. \tag{1.11}$$

Paa esse tipo de sistema, a solução é simples e dada por

$$x_i = \frac{b_i}{a_{ii}}, 1 \leqslant i \leqslant n. \tag{1.12}$$

Assim, o vetor solução do sistema será

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \\ \vdots \\ b_n/a_{nn} \end{pmatrix} \tag{1.13}$$

O algoritmo a seguir implementa essa solução

Algoritmo 1.1 ***Solução de sistemas lineares diagonais***

Entrada: matriz diagonal de coeficientes $\boldsymbol{A}$, vetor de termos independentes $\boldsymbol{b}$

Passo 1: criar um vetor com todos os valores iguais a zero: $\boldsymbol{x} = 0$

para i *variando de 1 à* n **faça**

Passo 2: calcular os valores do vetor solução $x_i = \dfrac{b_i}{a_{ii}}$

Passo 3: atualizar os valores do vetor $\boldsymbol{x}$

e o código abaixo implementa, em uma função, o algoritmo

```
In [ ]: def sist_lin_diagonal(A,b): return b/A.diagonal()
```

Na implementação, utilizou-se o método `diagonal` da biblioteca `Numpy` que retorna um vetor com os valores da diagonal principal de uma matriz quadrada (veja a documentação do metodo nesse link). Atenção para não confundir com o método `diag`, que pode ser usado para criar uma matriz diagonal. Isso permite aproveitar a vetorização nativa da `Numpy` sem precisar fazer laços explícitos na implementação.

O exemplo a seguir ilustra o uso da função `sist_lin_diagonal`.

Exemplo 3.1

Encontrar a solução do sistema de equações

$$\begin{aligned} x &= 4 \\ 4y &= 2 \\ 2z &= 2 \end{aligned} \tag{1.14}$$

Na forma matricial, o sistema pode ser definido pela matriz de coeficientes e pelo vetor de termos independentes

```
In [ ]: A = np.diag([1,4,2])
        b = np.array([4, 2, 2])
```

e usando a função implementada,

```
In [ ]: sist_lin_diagonal(A,b)
Out[ ]: array([4. , 0.5, 1. ])
```

obtém-se o vetor solução

$$\boldsymbol{x} = \begin{pmatrix} 4 \\ 0.5 \\ 1 \end{pmatrix}$$

Solução de sistemas lineares com matrix triangular de coeficientes

Sistemas de equações triangulares são sistemas cuja matriz dos coeficientes é uma matriz triangular superior ou inferior.

Na forma algébrica, sistemas triangulares superiores tem o formato

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1(n-1)}x_{n-1} + a_{1n}x_n &= b_1 \\ a_{22}x_2 + a_{23}x_3 + \cdots + a_{2(n-1)}x_{n-1} + a_{2n}x_n &= b_2 \\ a_{33}x_3 + \cdots + a_{3(n-1)}x_{n-1} + a_{3n}x_n &= b_3 \\ &\vdots \\ a_{(n-1)(n-1)}x_{n-1} + a_{(n-1)n}x_n &= b_{n-1} \\ a_{nn}x_n &= b_n \end{aligned} \tag{1.15}$$

em que a matriz de coeficientes

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{nn} \end{pmatrix} \tag{1.16}$$

é uma matriz triangular superior, cujos elementos abaixo da diagonal principal são nulos, isto é, $a_{ij} = 0$ para todo $i > j$.

Observando a equação (1.15), é possível perceber que a incógnita da última equação,

x_n, pode ser encontrada diretamente, simplesmente fazendo

$$a_{nn}x_n = b_n \quad \Rightarrow \quad x_n = \frac{b_n}{a_{nn}}. \tag{1.17}$$

Para a penúltima equação, $a_{(n-1)(n-1)}x_{n-1} + a_{(n-1)n}x_n = b_{n-1}$, o valor de uma incógnita já é conhecido, x_n. Dessa forma, para encontrar a incógnita que falta, basta isolar a incógnita cujo valor ainda não se conhece

$$a_{(n-1)(n-1)}x_{n-1} + a_{(n-1)n}x_n = b_{n-1} \quad \Rightarrow \quad x_{n-1} = \frac{b_{n-1} - a_{(n-1)n}x_n}{a_{(n-1)(n-1)}} \tag{1.18}$$

substituindo o valor de x_n encontrado no passo anterior.

O leitor atento, pode perceber que o mesmo procedimento valerá para a antepenúltima equação, quando duas das três incógnitas já serão conhecidas e que esse raciocínio seguirá até que, para a primeira equação do sistema, que possue n incógnitas, se conheça o valor de $n-1$ de suas incógnitas, faltando apenas determinar uma.

Esse procedimento no qual os valores das incógnitas de uma equação são obtidos por meio de substituições feitas de "baixo para cima", isto é, da incógnita de maior índice para a de menor, é chamado de *substituição retroativa*, e é formalmente descrito no algoritmo 1.2 a seguir.

Algoritmo 1.2 ***Substituição retroativa para sistemas triangulares superiores***

Entrada: matriz triangular superior de coeficientes $\boldsymbol{A}$, vetor de termos independentes $\boldsymbol{b}$

Passo 1: criar um vetor com todos os valores iguais a zero: $\boldsymbol{x} = 0$

Passo 2: calcular a incógnita de maior índice $x_n = \dfrac{b_n}{a_{nn}}$

para i *variando de* $n-1$ *à 1* **faça**

Passo 3: calcular os valores restantes do vetor solução

$$x_i = \frac{b_i - \displaystyle\sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}} \tag{1.19}$$

Passo 4: atualizar os valores do vetor $\boldsymbol{x}$

Pensemos na implementação. Para o passo 1 do algoritmo, uma opção é criar uma matriz "vazia", usando o método `empty` da `Numpy`, por ele ser, em muitas situações, mais rápido que usar o método `zeros`, que cria um vetor ou matriz com todos os elementos iguais a zero. É importante ter em mente que usar `empty` é diferente de criar um `array` sem elementos, pois o primeiro cria uma estrutura com elementos, mas com valores não definidos previamente, enquanto o segundo realmente não tem nenhum elemento, nem forma, como ilustra o exemplo de código abaixo.

```
In [ ]: np.empty(3).shape
Out[ ]: (3,)
In [ ]: np.array([]).shape
Out[ ]: (0,)
```

Mais uma curiosidade que merece atenção no uso: uma vez que `empty` não requer inicialização, ele assume valores "aleatórios", como mostrado acima, ou pode ser inicializado com o valor de um dos vetores ou matrizes de mesmo tamanho já inicializados no código. Portanto, é preciso bastante cuidado ao usar o método para não verificar valores estranhos oriundos de inicializações que não estavam planejadas.

Assim, para o passo 1, poderíamos usar o código

```
In [ ]: x = np.empty(n)
```

Para o passo 2, onde calcula-se o valor da incógnita de maior índice, basta usar os últimos valores do vetor `b` e da matriz `A`, invocados usando o índice `-1`.

```
In [ ]: x[-1] = b[-1]/A[-1, -1]
```

Uma vez que a incógnita x_n foi encontrada, é preciso implementar uma abordagem iterativa que considere os índices de $n-1$ à 1, nessa ordem, pois o cálculo é do maior para o menor. Isso pode ser feito por um laço, com os índices sendo gerados pelo método `range` gerando uma lista decrescente de valores, com valor inicial `n-2`, valor final `-1` (lembre-se que Python indexa a partir do 0 e que o método range não considera o valor final incluso) e passo `-1`.

Dentro do laço, o desafio é a implementação do cálculo iterativo das incógnitas, descrito pela equação 1.19 no passo 3 do algoritmo.

Focando no somatório da equação, podemos perceber que

$$\sum_{j=i+1}^{n} a_{ij}x_j = a_{i(i+1)}x_{(i+1)} + a_{i(i+2)}x_{(i+2)} + a_{i(i+3)}x_{(i+3)} + \ldots + a_{in}x_n, \tag{1.20}$$

é, basicamente, a soma dos produtos termo a termo entre os vetores

$$\boldsymbol{a}_{i,(i+1:n)} = \begin{bmatrix} a_{i(i+1)} & a_{i(i+2)} & a_{i(i+3)} & \cdots & a_{in} \end{bmatrix}, \tag{1.21}$$

formado pelos elementos que estão na linha i e entre as colunas $i+1$ e n de $\boldsymbol{A}$ e

$$\boldsymbol{x}_{(i+1:n)} = \begin{bmatrix} x_{(i+1)} & x_{(i+2)} & x_{(i+3)} & \cdots & x_n \end{bmatrix}, \tag{1.22}$$

formado pelos elementos que estão entre os índices $i+1$ e n de $\boldsymbol{x}$.

Ora, como a soma dos produtos termo a termo entre os vetores é a definição de produto interno, então pode-se afirmar que

$$\sum_{j=i+1}^{n} a_{ij}x_j = \langle \boldsymbol{a}_{i,(i+1:n)}, \boldsymbol{x}_{(i+1:n)} \rangle \tag{1.23}$$

em que o operador $\langle\,,\rangle$ indica o produto interno entre dois vetores.

Seguindo essa abordagem, o vetor $\boldsymbol{a}_{i,(i+1:n)}$ pode ser implementado usando o fatiamento `A[i,i+1:]` da matriz `A` e $\boldsymbol{x}_{(i+1:n)}$ usando o fatiamento `x[i+1:]` do vetor `x`, e o produto interno entre os dois, equivalente ao somatório da equação, pode ser implementado como

In [ ]:
```
np.sum(A[i,i+1:]*x[i+1:]))
```

Dessa forma, a equação 1.19 pode ser implementada como

In [ ]:
```
x[i] = (b[i] - np.sum(A[i,i+1:]*x[i+1:]))/A[i,i]
```

Juntando tudo isso numa única função, temos o algoritmo 1.2 é implementado na função `sist_lin_tri_sup`, mostrada abaixo.

In [ ]:
```
def sist_lin_tri_sup(A,b):
    n = len(b)
    x = np.empty(n)
    x[-1] = b[-1]/A[-1, -1]
    for i in range(n-2, -1, -1):
        x[i] = (b[i] - np.sum(A[i,i+1:]*x[i+1:]))/A[i,i]
    return x
```

O exemplo a seguir ilustra o uso da implementação.

Exemplo 3.2

Encontre o vetor solução do sistema

$$\begin{aligned} 3x_1 + 4x_2 - 5x_3 + x_4 &= -10 \\ x_2 + x_3 - 2x_4 &= -1 \\ 4x_3 - 5x_4 &= 3 \\ 2x_4 &= 2 \end{aligned} \tag{1.24}$$

Definindo $\boldsymbol{A}$ e $\boldsymbol{b}$ usando a biblioteca `Numpy`, temos

```
In [ ]: A = np.array([[3,4,-5,1],[0,1,1,-2],[0,0,4,-5],[0,0,0,2]])
        b = np.array([-10,-1,3,2])
```

e usando a função `sist_lin_tri_sup` para calcular a solução, obtemos

```
In [ ]: x = sist_lin_tri_sup(A,b)
        x
Out[ ]: array([ 1., -1., 2., 1.])
```

Podemos verificar se o resultado de fato é correto resolvendo o sistema, isto é, se ao multiplicarmos $\boldsymbol{A}$ por $\boldsymbol{x}$ e, se obtivermos $\boldsymbol{b}$, então o resultado está certo. Para isso, basta fazer

```
In [ ]: A.dot(x)
Out[ ]: array([-10., -1., 3., 2.])
```

que é igual a $\boldsymbol{b}$, garantindo a corretude da solução.

■ **Exercício 1.2** *Repita o exemplo 3.2 acima, porém fazendo cada passo da execução de forma explícita, comparando como o algoritmo 1.2 e a função `sist_lin_tri_sup` funcionam. Sugestão: faça isso de forma manuscrita. Ajudará a entender melhor cada passo.*

Os sistemas triangulares inferiores tem a forma algébrica do tipo

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{1.25}$$

em que a matriz de coeficientes

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \tag{1.26}$$

é uma matriz triangular inferior, cujos elementos acima da diagonal principal são nulos, isto é, $a_{ij} = 0$ para todo $i < j$.

A ideia da substituição usada nos sistemas triangulares superiores é exatamente a mesma para sistemas triangulares inferiores, diferenciando-se apenas que, ao invés de começar pela incógnita de maior índice, inicia-se pela de menor índice, sendo esse método, por esse motivo, referido como *substituição progressiva*.

Dessa forma a primeira incógnita encontrada será

$$x_1 = \frac{b_1}{a_{11}} \tag{1.27}$$

cujo valor pode ser usado na segunda equação para encontrar a segunda incógnita, fazendo

$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}, \tag{1.28}$$

e esses dois valores podem ser usados na terceira equação para encontrar o termo x_3 e assim por diante, até que, na última equação, os $n-1$ primeiros termo da solução serão conhecidos e possibilitarão encontrar o termo x_n.

O algoritmo da substituição progressiva para a solução de sistemas lineares triangulares inferiores é descrito a seguir

Algoritmo 1.3 ***Substituição progressiva para sistemas triangulares inferiores***

Entrada: matriz triangular inferior de coeficientes $\boldsymbol{A}$, vetor de termos independentes $\boldsymbol{b}$

Passo 1: criar um vetor com todos os valores iguais a zero: $\boldsymbol{x} = \boldsymbol{0}$

Passo 2: calcular a incógnita de menor índice $x_1 = \dfrac{b_1}{a_{11}}$

para i *variando de 2 à* n **faça**

Passo 3: calcular os valores do vetor solução

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}} \tag{1.29}$$

Passo 4: atualizar os valores do vetor $\boldsymbol{x}$

---

e a função `sist_lin_tri_inf` implementa esse algoritmo

In [ ]:
```
def sist_lin_tri_inf(A,b):
    n = len(b)
    x = np.empty(n)
    x[0] = b[0]/A[0, 0]
    for i in range(1,n):
        x[i] = (b[i] - np.sum(A[i,:i]*x[:i]))/A[i,i]
    return x
```

Aqui, três pontos a considerar: primeiro, que o índice do cálculo da primeira incógnita,

obviamente, precisa ser 0, pois estamos tratando da primeira equação; segundo, que o laço agora vai do menor ao maior índice, ao contrário do algoritmo anterior, que contava de forma decrescente; e, por fim, a lógica de implementar a equação (1.29) como um produto interno é a mesma do algoritmo anterior, com

$$\sum_{j=1}^{i-1} a_{ij}x_j = a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \ldots + a_{i(i-1)}x_{(i-1)} = \langle \boldsymbol{a}_{i,(1:i-1)}, \boldsymbol{x}_{(1:i-1)} \rangle, \tag{1.30}$$

sendo $\boldsymbol{a}_{i,(1:i-1)}$ o vetor composto pelos elementos que estão na linha i e entre as colunas 1 e $i-1$ de $\boldsymbol{A}$, implementado como `A[i,:i]` e $\boldsymbol{x}_{(1:i-1)}$ o vetor formado pelos elementos que estão entre os índices 1 e $i-1$ de $\boldsymbol{x}$, implementado como `x[:i]`.

Exemplo 3.3

Encontre a solução do seguinte sistema de equações lineares

$$\begin{aligned} 3x_1 &= 4 \\ 2x_1 + x_2 &= 2 \\ x_1 + x_3 &= 4 \\ x_1 + x_2 + x_3 + x_4 &= 2 \end{aligned} \tag{1.31}$$

Definindo $\boldsymbol{A}$ e $\boldsymbol{b}$ usando a biblioteca `Numpy`, temos

```
In [ ]: A = np.array([[3,0,0,0],[2,1,0,0],[1,0,1,0],[1,1,1,1]])
        b = np.array([4,2,4,2])
```

e usando a função `sist_lin_tri_inf` para calcular a solução, obtemos

```
In [ ]: x = sist_lin_tri_inf(A,b)
        x
Out[ ]: array([ 1.33333333, -0.66666667, 2.66666667, -1.33333333])
```

Podemos verificar o resultado, basta fazer

```
In [ ]: A.dot(x)
Out[ ]: array([4., 2., 4., 2.])
```

verificando que é igual ao vetor $\boldsymbol{b}$.

Acerca das soluções, é possível notar que, nas equações (1.19) e (1.29):

i. se $a_{ii} \neq 0$ para qualquer valor de i, o sistema é possível e determinado;

ii. se $a_{ii} = 0$ para alguma valor de i, há dois casos a analisar:

- se o numerador for igual a zero, o sistema é possível, mas indeterminado;
- se o numerador for diferente de zero, o sistema é impossível.

Solução de sistemas triangulares usando `Scipy`

No submódulo `linalg` da biblioteca `Scipy`, existe a função `solve_triangular`, que, conforme diz sua descrição na documentação (veja o link), resolve a equação $\boldsymbol{Ax} = \boldsymbol{b}$ para $\boldsymbol{x}$, assumindo que $\boldsymbol{A}$ é uma matriz triangular. Da documentação da função, os principais parâmetros da função são:

- `a`: a matriz triangular dos coeficientes do sistema;
- `b`: o vetor de termos independentes;
- `lower`: encontra a solução considerando o sistema como triangular superior (`lower = False`) ou inferior (`lower = True`), sendo este último o valor *default* do parâmetro.

Os demais parâmetros são para casos muito particulares, não tratados aqui. A função retorna, então, um `array` contendo o vetor de incógnitas `x`, possuindo a mesma forma do vetor `b`.

O exemplo a seguir ilustra o uso da função `solve_triangular`.

Exemplo 3.4

Resolver os exemplos 3.2 e 3.3 usando a função `solve_triangular`.

Para o exemplo 3.2, que trabalha um sistema triangular superior, tem-se

```
In [ ]: A1 = np.array([[3,4,-5,1],[0,1,1,-2],[0,0,4,-5],[0,0,0,2]])
        b1 = np.array([-10,-1,3,2])
```

```
In [ ]: sla.solve_triangular(A1,b1, lower=False)
```

```
Out[ ]: array([ 1., -1., 2., 1.])
```

e para o exemplo 3.3, que trabalha um sistema triangular inferior, tem-se

```
In [ ]: A2 = np.array([[3,0,0,0],[2,1,0,0],[1,0,1,0],[1,1,1,1]])
        b2 = np.array([4,2,4,2])
```

```
In [ ]: sla.solve_triangular(A2,b2, lower=True)
```

```
Out[ ]: array([ 1.33333333, -0.66666667, 2.66666667, -1.33333333])
```

que são os mesmos resultados encontrados anteriormente.

■ **Exercício 1.3** *Repita o exemplo 3.3 acima, porém fazendo cada passo da execução de forma explícita, comparando como o algoritmo 1.3 e a função* `sist_lin_tri_inf` *funcionam. Use a mesma sugestão do exercício anterior.*

■ **Exercício 1.4** *Implemente uma função, cujos parâmetros de entrada sejam somente a matriz de coeficientes e o vetor de termos independentes e que calcule o vetor solução para qualquer um dos casos triviais vistos aqui. Atente para o fato de que a própria função terá de verificar se a matriz de coeficientes é diagonal, triangular superior ou inferior e não o usuário.*