Minicurso de Python para Matemática

Pedro H A Konzen

26 de setembro de 2021

Sumário

1 Sobre a linguagem 2

1.1 Instalação e execução 3

1.2 Utilização 3

2 Elementos da linguagem 4

2.1 Operações aritméticas elementares 5

2.2 Funções e constantes elementares 7

2.3 Operadores de comparação elementares 7

2.4 Operadores lógicos elementares 8

2.5 Conjuntos 9

2.6 n-uplas 11

2.7 Listas 12

2.8 Dicionários 15

3 Função, ramificação e repetição 16

3.1 Definindo funções 16

3.2 Ramificação 18

3.3 Repetição 19

3.3.1 while 19

3.3.2 for 20

3.3.3 range 21

4 Elementos da computação matricial 22

4.1 NumPy array 22

4.1.1 Inicialização de um array 23
4.1.2 Manipulação de arrays 24
4.1.3 Operadores elemento-a-elemento 25
4.2 Elementos da álgebra linear 26
4.3 Vetores 26
4.3.1 Produto escalar e norma 27
4.3.2 Matrizes 28
4.3.3 Inicialização de matrizes 29
4.3.4 Multiplicação de matrizes 30
4.3.5 Traço e Determinante de uma matriz 31
4.3.6 Rank e inversa de uma matriz 31
4.3.7 Autovalores e autovetores de uma matriz 32

5 Gráficos **33**

Referências Bibliográficas **34**

Licença

1 Sobre a linguagem

Python é uma linguagem de programação de alto nível e multi-paradigma. Ou seja, é relativamente próxima das linguagens humanas naturais, é desenvolvida para aplicações diversas e permite a utilização de diferentes paradigmas de programação (programação estruturada, orientada a objetos, orientada a eventos, paralelização, etc.).

- Site oficial: https://www.python.org/

1.1 Instalação e execução

Para executar um código Python é necessário instalar um interpretador. No site oficial do Python estão disponível para *download* interpretadores gratuitos e com licença para uso livre. Neste minicurso, vamos utilizar Python 3 instalado em um sistema `Linux`. Para outros sistemas, pode ser necessário fazer algumas pequenas adequações.

1.2 Utilização

A execução de códigos Pythonpode ser feita de três formas básicas:

- em modo iterativa em um console Python;
- por execução de um código `.py` em um console Python;
- por execução de um cógido `.py` em um terminal;

Exemplo 1.1. Implemente o seguinte pseudocódigo.

```
s = "Ola, mundo!".
imprime(s). (imprime a string s)
```

a) em modo iterativo no console;

b) escrevendo o código `.py` e executando-o no console;

c) escrevendo o código `.py` e executando-o no terminal.

Resolução. Seguem as implementações em cada caso.

a) Em modo iterativo.

Iniciamos um console Python em terminal digitando

```
$ python3
```

Então, digitamos

```
>>> s = "Ola, Mundo!"
>>> print(s) #imprime a string s
Ola, Mundo!
```

Para encerrar o console, digitamos

```
>>> quit()
```

b) Executando *script* `.py` no console.

Primeiramente, escrevemos o código

```
s = "Ola, Mundo!"
print(s) # imprime a string s
```

em um editor de texto (ou no seu IDE de preferência) e salvamo-lo em `/pasta/codigo.py`. Então, executamo-lo no console Python com

```
>>> exec(open('/pasta/codigo.py').read())
Ola, mundo!
```

c) Executando o código em terminal.

Considerando que já temos o código salvo em `/pasta/codigo.py`, executamo-lo com

```
$ python3 /pasta/codigo.py
Olá, mundo!
```

2 Elementos da linguagem

Python é uma linguagem de programação dinâmica em que as variáveis são declaradas automaticamente ao receberem um valor. Por exemplo, consideremos as seguintes instruções

```
>>> x = 1
>>> y = x * 2.0
```

Na primeira instrução, a variável `x` é recebe o valor inteiro 1 e, então, é armazenado na memória do computador como um objeto da classe `int`. Na segunda instrução, `y` recebe o valor decimal 2.0 (resultado de 1×2.0) e é armazenado como um objeto da classe `float` (ponto flutuante de 64-*bits*). Podemos verificar isso, com as seguintes instruções

```
>>> print(x, y)
1 2.0
>>> print(type(x), type(y))
<class 'int'> <class 'float'>
```

Códigos Python admitem comentários e continuação de linha como no seguinte exemplo

```
>>> # isso eh um comentario
>>> s = "isso eh uma \
... string"
>>> print(s)
isso eh uma string
>>> type(s)
<class 'str'>
```

Observação 2.1. (Notação científica) O Python aceita notação científca, por exemplo 5.2×10^{-2} é digitado como

```
>>> 5.2e-2
0.052
```

Observação 2.2. Além dos tipos numéricos e *string*, Python também conta com os tipos de dados `list` (lista), `tuple` (n-upla) e `dict` (dicionário). Estudaremos estes tipos mais adiante neste minicurso.

Exercício 2.1. Antes de implementar, diga qual o valor `x` após as seguintes instruções.

```
>>> x = 1
>>> y = x
>>> y = 0
```

Justifique seu resposta e verifique-a.

Exercício 2.2. Implemente um código em que o usuário entre com valores para as variáveis `x` e `y`[1]. Então, os valores das variáveis são permutados entre si.

2.1 Operações aritméticas elementares

Os operadores aritméticos elementares são:

`+`: adição

[1] A entrada de valores via console pode ser feita com a função Python `input`. Consule Python Docs.

- `-`: subtração
- `*`: multiplicação
- `/`: divisão
- `**`: potenciação
- `%`: módulo
- `//`: módulo

Consideremos o seguinte exemplo

```
>>> 2+8*3/2**2-1
7.0
```

Observamos que as operações `**` tem precedência sobre as operações `*, /`, as quais têm precedência sobre as operações `+, -`. Operações de mesma precedência seguem a ordem da esquerda para direita, conforme escritas na linha de comando. Usa-se parênteses para alterar a precedência entre as operações, por exemplo

```
>>> (2+8*3)/2**2-1
5.5
```

Consulte mais informações sobre a precedência de operadores em Python Docs.

Exercício 2.3. Compute as raízes do seguinte polinômio quadrático

$$x^2 - x - 2 \tag{1}$$

usando a fórmula de Bhaskara[2]

O operador `%` módulo computa o resto da divisão e o operador `//` a divisão inteira, por exemplo

```
>>> 5 % 2
1
>>> 5 // 2
2
```

[2] Bhaskara Akaria, 1114 - 1185, matemático e astrônomo indiano. Fonte: Wikipédia.

Exercício 2.4. Use o Python para verificar se 14/21 é menor que 15/23. Então, compute o resto da divisão do maior quociente.

2.2 Funções e constantes elementares

O módulo Python math disponibiliza várias funções e constantes elementares. Para usá-las, precisamos importar o módulo para nossa seção. Fazemos isso com a instrução

```
>>> import math
```

Com isso, temos acesso a todas as definições e declarações contidas neste módulo. Por exemplo

```
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
>>> math.sqrt(2)
1.4142135623730951
>>> math.log(math.e)
1.0
```

Observação 2.3. Notemos que `math.log` é a função logaritmo natural, i.e. $\ln(x) = \log_e(x)$. A implementação Python para o logaritmo de base 10 é `math.log(x,10)` ou, mais acurado, `math.log10`.

Exercício 2.5. Compute $e^{\log_3(\pi)}$.

2.3 Operadores de comparação elementares

Os operadores de comparação elementares são

`==`: igual a

`!=`: diferente de

`>`: maior que

`<`: menor que

`>=`: maior ou igual que

`<=`: menor ou igual que

Estes operadores retornam os valores lógicos `True` (verdadeiro) ou `False` (falso).

Por exemplo, temos

```
>>> x = 2
>>> x + x == 4
True
```

Exercício 2.6. Atribua a variável `x` o valor $\sqrt{3}$. Então, verifique se o valor computado de x^2 é maior que 3. Em caso negativo, verifique se x^2 é menor que 3. Comente o resultado obtido.

2.4 Operadores lógicos elementares

Os operadores lógicos elementares são:

`and`: e lógico

`or`: ou lógico

`not`: não lógico

A tabela booleana[3] do "e" lógico é

Valor	Valor	Resultado
True	True	True
True	False	False
False	True	False
False	False	False

Podemos verificar isso no Python como segue

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
```

[3]George Boole, 1815 - 1864, matemático e filósofo britânico. Fonte: Wikipédia.

```
>>> False and False
False
```

Exercício 2.7. Construa as tabelas booleanas do operador `or` e do `not`.

Exercício 2.8. Use Python para verificar se $1.4 <= \sqrt{2} < 1.5$. E, também, verifique se $\sqrt{3} > 1.7$ ou $\sqrt{3} >= 1.7321$.

Exercício 2.9. Implemente uma instrução para computar o operador `xor` (ou exclusivo). Dadas duas afirmações `A` e `B`, `A xor B` é `True` no caso de uma, e somente uma, das afirmações ser `True`, caso contrário é `False`.

2.5 Conjuntos

Python tem conjuntos finitos como um tipo básico de variável. Um conjunto é uma coleção de itens **não ordenada** e **imutável** e **não admite itens duplicados**. Por exemplo,

```
>>> a = {1, 2, 3}
>>> type(a)
<class 'set'>
>>> b = set((2, 1, 3, 3))
>>> b
{1, 2, 3}
>>> a == b
True
>>> # conjunto vazio
>>> e = set()
```

aloca o conjunto $a = 1{,}23$. Note que o conjunto b é igual a a. Observamos que o conjunto vazio deve ser construído com a instrução `set()` e não com `{}`[4].

Observação 2.4. A função Python `len` retorna o número de elementos de um conjunto. Por exemplo,

```
>>> len(a)
3
```

[4]Isso constrói um dicionário vazio, como introduziremos logo mais.

- Operadores envolvendo conjuntos:

    `-`: diferença entre conjuntos;

    `|`: união de conjuntos;

    `&`: interseção de conjuntos;

    `^`: diferença simétrica;

Exemplo 2.1. Sejam os conjuntos

$$A = \{2, \pi, -0.25, 3, \text{'banana'}\} \tag{2}$$
$$B = \{\text{'laranja'}, 3, \arccos(-1), -1\} \tag{3}$$

Compute

a) $A \setminus B$

b) $A \cup B$

c) $A \cap B$

d) $A \Delta B = (A \setminus B) \cup (B \setminus A)$

Resolução. Começamos alocando os conjuntos como segue

```
>>> import math
>>> A = {2, math.pi, -0.25, 3, 'banana'}
>>> B = {'laranja', 3, math.acos(-1), -1}
```

a) $A \setminus B$

```
>>> A - B
{-0.25, 2, 'banana'}
```

b) $A \cup B$

```
>>> A | B
{-0.25, 2, 3, 3.141592653589793, \
'laranja', 'banana', -1}
```

c) $A \cap B$

```
>>> A & B
{3, 3.141592653589793}
```

d) $A \Delta B$

```
>>> A ^ B
{-0.25, 2, 'laranja', 'banana', -1}
```

Observação 2.5. Python disponibiliza a sintaxe de compreensão de conjuntos. Por exemplo,

```
>>> {x for x in A if type(x) == type('')}
{'banana'}
```

Exercício 2.10. Considere o conjunto

$$Z = -3, -2, -1, 0, 1, 2, 3. \tag{4}$$

Faça um código Python para extrair o subconjunto dos números pares do conjunto Z.

2.6 n-uplas

Em Python n-uplas (*tuples*) é uma sequência de objetos, i.e. uma coleção ordenada, indexada e imutável. Por exemplo, na sequência temos um par, uma tripla e uma quadrupla

```
>>> a = (1, 2)
>>> a
(1, 2)
>>> b = -1, 1, 0
(-1, 1, 0)
>>> c = (0.5, 'laranja', {2, -1}, ['t', 0])
>>> c
(0.5, 'laranja', {2, -1}, ['t', 0])
>>> len(c)
4
```

A função `len` retorna o número de objetos da n-upla. Pode acessar um elemento, usando sua indexação. Por exemplo,

```
>>> c[2]
{2, -1}
```

Pode-se também extrair uma fatia (um subconjunto) usando-se a notação `:`. Por exemplo,

```
>>> c[1:3]
('laranja', {2, -1})
```

- Operadores básicos:

  `+`: concatenação

  ```
  >>> (1, 2) + (3, 4, 5)
  (1, 2, 3, 4, 5)
  ```

  `*`: repetição

  ```
  >>> (1, 2) * 2
  (1, 2, 1, 2)
  ```

  `in`: pertencimento

  ```
  >>> 1 in (-1, 0, 1, 2)
  True
  ```

Exercício 2.11. Aloque os conjuntos

$$A = \{-1, 0, 2\}, \tag{5}$$
$$B = \{2, 3, 5\}. \tag{6}$$

Então, compute o produto cartesiano $A \times B$.

2.7 Listas

O tipo Python `list` permite alocar em uma única variável uma lista de itens ordenada. Por exemplo, observemos as seguintes listas

```
>>> x = [-1, 2, -3, -5]
>>> type(x)
<class 'list'>
>>> y = ['a', 'b', 'a']
>>> y
['a', 'b', 'a']
>>> vazia = []
```

```
>>> len(x)
4
```

Os elementos de uma lista são indexados, o índice 0 corresponde ao primeiro elemento, o índice 1 ao segundo elemento e assim por diante. Desta forma é possível o acesso direto a um elemento de uma lista usando-se sua posição. Por exemplo,

```
>>> x[0]
-1
>>> y[2] = 'c'
>>> y
['a', 'b', 'c']
```

Pode-se fazer um corte de elementos de uma lista usando o operador `:`. Por exemplo,

```
>>> x = [1,2,-1,3,-2]
>>> x[1:3]
[2,-1]
```

- Operadores básicos:

  `+`: concatenação

  ```
  >>> [1, 2] + [3, 4, 5]
  [1, 2, 3, 4, 5]
  ```

  `*`: repetição

  ```
  >>> [1, 2] * 2
  [1, 2, 1, 2]
  ```

  `in`: pertencimento

  ```
  >>> 1 in [-1, 0, 1, 2]
  True
  ```

Observação 2.6. Listas contam com várias funções prontas para a execução de diversas tarefas práticas como, por exemplo, inserir/deletar itens, contar ocorrências, ordenar itens, etc. Consulte Python Docs.

Observação 2.7. (Alocação *versus* cópia) Estude o seguinte exemplo

```
>>> x = [2, 3, 1]
>>> y = x
>>> y[1] = 0
>>> x
[2, 0, 1]
```

Notamos que `y` aponta para o mesmo endereço de memória de `x`. Para copiar uma lista e alocá-la em um novo endereço de memória, deve-se usar a função `list.copy()`, como segue

```
>>> x = [2, 3, 1]
>>> y = x.copy()
>>> y[1] = 0
>>> x
[2, 3, 1]
>>> y
[2, 0, 1]
```

Exercício 2.12. Implemente uma lista para alocar os primeiros 5 elementos da sequência de Fibonacci[5].

Exercício 2.13. Uma aplicação do Método Babilônico[6] para a aproximação da solução da equação $x^2 - 2 = 0$, consiste na iteração

$$x_0 = 1, \tag{7}$$

$$x_{i+1} = \frac{x_i}{2} + \frac{1}{x_i}, \quad i = 0,1,2,\ldots \tag{8}$$

Implemente uma lista para alocar as quatro primeiras aproximações da solução, i.e. x_0, x_1, x_2, x_3.

Exercício 2.14. Aloque os seguintes vetores como listas no Python

$$x = (-1, 3, -2), \tag{9}$$

$$y = (4, -2, 0). \tag{10}$$

Então, compute

[5]Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: Wikipédia.

[6]Matemática Babilônica, matemática desenvolvida na Mesopotâmia, desde os Sumérios até a queda da Babilônia em 539 a.C.. Fonte: Wikipédia.

a) $x + y$

b) $x \cdot y$

Exercício 2.15. Uma matriz pode ser alocada como uma lista de listas Python, alocando cada linha como uma lista e a matriz como a lista destas listas. Por exemplo, a matriz

$$M = \begin{bmatrix} 1 & -2 \\ 2 & 3 \end{bmatrix} \tag{11}$$

pode ser alocada como a seguinte lista de listas

```
>>> M = [[1,-2],[2,3]]
>>> M
[[1, -2], [2, 3]]
```

Use listas para alocar a matriz

$$A = \begin{bmatrix} 1 & -2 & 1 \\ 8 & 0 & -7 \\ 3 & -1 & -2 \end{bmatrix} \tag{12}$$

e o vetor coluna

$$x = (2, -3, 1), \tag{13}$$

então compute Ax e $x^T A$.

2.8 Dicionários

Em Python um dicionário é um mapeamento objeto a objeto, cada par (chave:valor) é separado por uma vírgula. Por exemplo,

```
>>> a = {'nome': 'triangulo', 'perimetro': 3.2}
>>> a
{'nome': 'triangulo', 'perimetro': 3.2}
>>> b = {1: 2.71, (1,2): 2, 'a': {1,0,-1}}
>>> b
{1: 2.71, (1, 2): 2, 'a': {0, 1, -1}}
>>> d = {1: 'a', 'b': 2, 1.4: -1, 2: 'b'}
>>> d
{1: 'a', 'b': 2, 1.4: -1, 2: 'b'}
```

O acesso a um item do dicionário é feito usando-se sua **chave**. Por exemplo,

```
>>> d['b']
2
>>> d[1.4] = 1
>>> d
{1: 'a', 'b': 2, 1.4: 1, 2: 'b'}
```

Pode-se adicionar um novo par, simplesmente, atribuindo valor a uma nova chave. Por exemplo,

```
>>> d[1.5] = 0
>>> d
{1: 'a', 'b': 2, 1.4: 1, 2: 'b', 1.5: 0}
```

Observação 2.8. Consulte sobre mais sobre dicionários em Python Docs.

Exercício 2.16. Considere a função afim

$$f(x) = 3 - x. \tag{14}$$

Implemente um dicionário para alocar a raiz da função, a interseção com o eixo y e seu coeficiente angular.

Exercício 2.17. Considere a função quadrática

$$g(x) = x^2 - x - 2 \tag{15}$$

Implemente um dicionário para alocar suas raízes, vértice e interseção com o eixo y.

3 Função, ramificação e repetição

Nesta seção, vamos introduzir funções e estruturas de ramificação e de repetição. Estes são procedimentos fundamentais na programação estruturada.

3.1 Definindo funções

Em Python, uma função é definida com a palavra-chave `def` seguida de seu nome e seus parâmetros encapsulados entre parênteses e por dois-pontos `:`. Suas instruções formam o corpo da função, iniciam-se na linha abaixo e devem estar indentadas. A indentação define o escopo da função. Por exemplo, a seguinte função imprime o valor da função

$$f(x) = 2x - 3 \tag{16}$$

```
>>> def f(x):
...     y = 2*x - 3
...     print(y)
...
>>> f(2)
1
```

Você pode protestar que `f` não é uma função e, sim, um procedimento, pois não retorna valor. Para uma função retornar um objeto, usamos a instrução `return`. Por exemplo,

```
>>> def f(x):
...     y = 2*x - 3
...     return y
...
>>> z = f(2)
>>> z
1
```

Observação 3.1. Para funções pequenas, pode-se utilizar a instrução `lambda` de funções anônimas. Por exemplo,

```
>>> f = lambda x: 2*x - 3
>>> f(3)
3
```

Observação 3.2. Consulte mais informações sobre a definição de funções em Python Docs.

Exercício 3.1. Implemente uma função para computar as raízes de uma polinômio de grau 1 $p(x) = ax + b$.

Exercício 3.2. Implemente uma função para computar as raízes de uma polinômio de grau 2 $p(x) = ax^2 + bx + c$.

Exercício 3.3. Implemente uma função que computa o produto escalar de dois vetores

$$x = (x_0, x_1, x_2), \tag{17}$$

$$y = (y_0, y_1, y_2). \tag{18}$$

Use listas para representar os vetores no Python.

Exercício 3.4. Implemente uma função que computa o determinante de matrizes 2×2. Use lista de listas para representar as matrizes.

Exercício 3.5. Implemente uma função que computa a multiplicação matrix-vetor Ax, com A 2×2 e x um vetor coluna de dois elementos.

3.2 Ramificação

Uma estrutura de ramificação é uma instrução para a tomada de decisões durante a execução de um programa. No Python, está disponível a instrução `if`. Consultemos o seguinte exemplo.

```
def paridade(n):
    if (n%2 == 0):
        print('par')
```

Aqui, a função `paridade` recebe o valor `n`. Se (`if`) o resto da divisão de `n` por 2 é igual a zero (condição), então (`:`) imprime a *string* `par`.

Observação 3.3. A indentação determina o escopo de cada instrução `if`.

Também está disponível a instrução `if-else`. Por exemplo,

```
def paridade(n):
    if (n%2 == 0):
        print('par')
    else:
        print('impar')
```

Agora, se (`if`) a condição `(n%2 == 0)` for verdadeira (`True`), então imprime `par`, senão (`else`) imprime `impar`.

Ainda, é possível ter instruções `if-else` encadeadas. Por exemplo,

```
def paridade(n):
    if (n%2 == 0):
        print('eh divisivel por 2')
    elif (n%3 == 0):
        print('eh divisivel por 3')
    else:
        print('nao eh divisivel por 2 e 3')
```

Observe que `elif` deve ser utilizado no lugar de `else if`.

Exercício 3.6. Implemente uma função que recebe dois números n e m e imprime o maior deles.

Exercício 3.7. Implemente uma função que recebe os coeficientes de um polinômio

$$p(x) = ax^2 + bx + c \tag{19}$$

e classifique-o como um polinômio de grau 0, 1 ou 2.

3.3 Repetição

Estruturas de repetição são instruções que permitem que a execução repetida de uma região do código. São duas instruções disponível `while` e `for`.

3.3.1 while

A sintaxe da instrução `while` é

```
while expressao:
    comando 0
    .
    .
    .
    comando n
```

Isto é, enquanto (`while`) a expressão (`expressao`) for verdadeira, os comandos `comando 0` a `comando n` serão repetidamente executados em ordem. Por exemplo, o seguinte código computa a soma dos 10 primeiros números naturais e, então imprime-a.

```
n = 0
s = 0
while (n < 10):
    s += n
    n += 1
print(s)
```

Observação 3.4. As instruções de controle `break`, `continue` são bastante úteis em várias situações. A primeira, encerra as repetições e, a segunda, pula para uma nova repetição. Consulte mais em Python Docs.

Exercício 3.8. Use `while` para imprimir os dez primeiros números ímpares.

Exercício 3.9. Uma aplicação do Método Babilônico[7] para a aproximação da solução da equação $x^2 - 2 = 0$, consiste na iteração

$$x_0 = 1, \tag{20}$$

$$x_{i+1} = \frac{x_i}{2} + \frac{1}{x_i}, \quad i = 0,1,2,\ldots \tag{21}$$

Faça um código com `while` para computar aproximação x_i, tal que $|x_i - x_{i-1}| < 10^{-5}$.

3.3.2 for

A estrutura `for` tem a sintaxe

```
for i in iteravel:
    escopo
```

onde, `iteravel` pode ser qualquer objeto de uma classe iterável (conjunto, n-upla, lista, dicionário, *string*). Os comandos dentro do escopo (determinado pela indentação) são repetidos para cada iterada `i`. Por exemplo,

```
>>> for i in [0,1,2]:
...     print(i)
...
0
1
```

[7] Matemática Babilônica, matemática desenvolvida na Mesopotâmia, desde os Sumérios até a queda da Babilônia em 539 a.C.. Fonte: Wikipédia.

```
2
```

3.3.3 range

A função Python `range([start,]stop[,sep])` é particularmente útil na construção de instruções `for`. Ela cria um objeto de classe iterável de `start` (incluído) a `stop` (excluído), de elementos igualmente separados por `sep`. Por padrão, `start=0`, `sep=1` caso omitidos. Por exemplo,

```
>>> for i in range(1,6,2):
...     print(i)
...
1
3
5
```

ou

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

Exercício 3.10. Escreva uma função que retorne o n-ésimo termo da função de Fibonacci[8], $n \geq 1$.

Exercício 3.11. Implemente uma função para computar o produto escalar de dois vetores de n elementos. Assuma que os vetores estão alocados em listas.

E 3.1. Implemente uma função para computar a multiplicação de uma matriz A $n \times n$ por um vetor coluna x de n elementos. Assuma que o vetor está alocada como uma lista e a matriz como uma lista de listas por linhas.

E 3.2. Implemente uma função para computar a multiplicação de uma matriz A $n \times m$ por uma matriz B de $m \times n$. Assuma que as matrizes estão

[8]Leonardo Fibonacci, 1170 - 1250, matemático italiano. Fonte: Wikipédia.

alocadas como listas de listas por linhas de cada matriz.

4 Elementos da computação matricial

Nesta seção, vamos explorar a NumPy (Numerical Python), biblioteca para tratamento numérico de dados. Ela é extensivamente utilizada nos mais diversos campos da ciência e da engenharia. Aqui, vamos nos restringir a introduzir algumas de suas ferramentas para a computação matricial.

Usualmente, a biblioteca é importada como segue

```
>>> import numpy as np
```

4.1 NumPy array

Um `array` é uma tabela de valores (vetor, matriz ou multidimensional) e contém informação sobre os dados brutos, indexação e como interpretá-los. **Os elementos são todos do mesmo tipo** (diferente de uma lista Python), referenciados pela propriedade `dtype`. A indexação dos elementos pode ser feita por um `tuple` de inteiros não negativos, por booleanos, por outro `array` ou por números inteiros. O `rank` de um `array` é seu número de dimensões (chamadas de `axes`[9]). O `shape` é um `tuple` de inteiros que fornece seu tamanho (número de elementos) em cada dimensão. Sua inicialização pode ser feita usando-se listas simples ou encadeadas. Por exemplo,

```
>>> a = np.array([1,3,-1,2])
>>> print(a)
[ 1  3 -1  2]
>>> a.dtype
dtype('int64')
>>> a.shape
(4,)
>>> a[2]
-1
>>> a[1:3]
array([ 3, -1])
```

[9]Do inglês, plural de *axis*, eixo.

temos um `array` de números inteiros com quatro elementos dispostos em um único `axis` (eixo). Podemos interpretá-lo como uma representação de um vetor linha ou coluna, i.e.

$$a = (1, 3, -1, 2) \tag{22}$$

vetor coluna ou a^T vetor linha.

Outro exemplo,

```
>>> a = np.array([[1.0,2,3],[-3,-2,-1]])
>>> a.dtype
dtype('float64')
>>> a.shape
(2, 3)
>>> >>> a[1,1]
-2.0
```

temos um `array` de números decimais (`float`) dispostos em um arranjo com dois `axes` (eixos). O primeiro `axis` tem tamanho 2 e o segundo tem tamanho 3. Ou seja, podemos interpretá-lo como uma matriz de duas linhas e três colunas. Podemos fazer sua representação algébrica como

$$a = \begin{bmatrix} 1 & 2 & 3 \\ -3 & -2 & -1 \end{bmatrix} \tag{23}$$

4.1.1 Inicialização de um array

O NumPy conta com úteis funções de inicialização de `array`. Vejam algumas das mais frequentes:

- `np.zeros()`: inicializa um `array` com todos seus elementos iguais a zero.

```
>>> np.zeros(2)
array([0., 0.])
```

- `np.ones()`: inicializa um `array` com todos seus elementos iguais a 1.

```
>>> np.ones((3,2), dtype='int')
array([[1, 1],
       [1, 1],
       [1, 1]])
```

- `np.empty()`: inicializa um `array` sem alocar valores para seus elementos[10].

```
>>> np.empty(3)
array([4.9e-324, 1.5e-323, 2.5e-323])
```

- `np.arange()`: inicializa um `array` com uma sequência de elementos[11].

```
>>> np.arange(1,6,2)
array([1, 3, 5])
```

- `np.linspace(a, b[, num=n])`: inicializa um `array` como uma sequência de elementos que começa em `a`, termina em `b` (incluídos) e contém `n` elementos igualmente espaçados.

```
>>> np.linspace(0, 1, num=5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

4.1.2 Manipulação de arrays

Outras duas funções importantes no tratamento de `arrays` são:

- `arr.reshape()`: permite a alteração da forma de um `array`.

```
>>> a = np.array([-2,-1])
>>> a
array([-2, -1])
>>> a.reshape(2,1)
array([[-2],
       [-1]])
```

O `arr.reshape()` também permite a utilização de um coringa `-1` que será dinamicamente determinado de forma obter-se uma estrutura adequada. Por exemplo,

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
```

[10] Atenção! Os valores dos elementos serão dinâmicos conforme "lixo" da memória.
[11] Similar a função Python `range`.

```
>>> a.reshape((-1,1))
array([[1],
       [2],
       [3],
       [4]])
```

- `arr.transpose()`: computa a transposta de uma matriz.

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
```

- `np.concatenate()`: concatena `arrays`.

```
>>> a = np.array([1,2])
>>> b = np.array([2,3])
>>> c = np.concatenate((a,b))
>>> c
array([1, 2, 2, 3])
>>> a = a.reshape((1,-1))
>>> a.ndim
2
>>> b = b.reshape((1,-1))
>>> b
array([[2, 3]])
>>> d = np.concatenate((a,b), axis=0)
>>> d
array([[1, 2],
       [2, 3]])
```

4.1.3 Operadores elemento-a-elemento

Os operadores aritméticos disponível no Python atuam elemento-a-elemento nos `arrays`. Por exemplo,

```
>>> a = np.array([1,2])
```

```
>>> b = np.array([2,3])
>>> a+b
array([3, 5])
>>> a-b
array([-1, -1])
>>> b*a
array([2, 6])
>>> a**b
array([1, 8])
>>> 2*b
array([4, 6])
```

O NumPy também conta com várias funções matemáticas elementares que operam elemento-a-elemento em `arrays`. Por exemplo,

```
>>> a = np.array([np.pi, np.sqrt(2)])
>>> a
array([3.14159265, 1.41421356])
>>> np.sin(a)
array([1.22464680e-16, 9.87765946e-01])
>>> np.exp(a)
array([23.14069263,  4.11325038])
```

Observação 4.1. O NumPy contém um série de outras funções práticas para a manipulação de `arrays`. Consulte NumPy: the absolute basics for beginners.

4.2 Elementos da álgebra linear

O NumPy conta com um módulo de álgebra linear

```
>>> from numpy import linalg
```

4.3 Vetores

Um vetor podem ser representado usando um `array` de um eixo (dimensão) ou um com dois eixos, caso se queira diferenciá-lo entre um vetor linha ou coluna. Por exemplo, os vetores

$$a = (2, -1, 7), b = (3, 1, 0)^T \tag{24}$$

podem ser alocados com

```
>>> x = np.array([2,-1,7])
>>> y = np.array([3,1,0])
```

Caso queira-se que x siga um arranjo em coluna, pode-se modificado como segue

```
>>> a = a.reshape((-1,1))
>>> a
array([[ 2],
       [-1],
       [ 7]])
```

Como já vimos, o NumPy conta com operadores elemento-a-elemento que podem ser utilizados na álgebra envolvendo `arrays`, logo também aplicáveis a vetores (consulte a Subseção 4.1.3). Vamos, aqui, introduzir outras operações próprias deste tipo de objeto.

Exercício 4.1. Aloque cada um dos seguintes vetores como um NumPy `array`:

a) $x = (1.2, -3.1, 4)$

b) $y = x^T$

c) $z = (\pi, \sqrt{2}, e^{-2})^T$

4.3.1 Produto escalar e norma

Dados dois vetores,

$$x = (x_0, x_1, \dots, x_{n-1}), \tag{25}$$
$$y = (y_0, y_1, \dots, y_{n-1}) \tag{26}$$

define-se o **produto escalar** por

$$x \cdot y = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1} \tag{27}$$

Com o NumPy, podemos computá-lo com a função `np.dot()`. Por exemplo,

```
>>> x = np.array([-1, 0, 2, 4])
>>> y = np.array([0, 1, 1, -1])
```

```
>>> np.dot(x,y)
-2
```

A norma (euclidiana) de um vetor é definida por

$$\|x\| = \sqrt{\sum_{i=0}^{n-1} x_i^2}. \tag{28}$$

O NumPy conta com a função `np.linalg.norm()` para computá-la. Por exemplo,

```
>>> np.linalg.norm(y)
1.7320508075688772
```

Exercício 4.2. Faça um código para computar o produto escalar $x \cdot y$ sendo

$$x = (1.2, \ln(2), 4), \tag{29}$$

$$y = (\pi^2, \sqrt{3}, e) \tag{30}$$

4.3.2 Matrizes

Uma matriz pode ser alocada como um NumPy `array` de dois eixos (dimensões). Por exemplo, as matrizes

$$A = \begin{bmatrix} 2 & -1 & 7 \\ 3 & 1 & 0 \end{bmatrix}, \tag{31}$$

$$B = \begin{bmatrix} 4 & 0 \\ 2 & 1 \\ -8 & 6 \end{bmatrix} \tag{32}$$

podem ser alocadas como segue

```
>>> A = np.array([[2,-1,7],[3,1,0]])
>>> A
array([[ 2, -1,  7],
       [ 3,  1,  0]])
>>> B = np.array([[4,0],[2,1],[-8,6]])
>>> B
array([[ 4,  0],
       [ 2,  1],
       [-8,  6]])
```

Como já vimos, o NumPy conta com operadores elemento-a-elemento que podem ser utilizados na álgebra envolvendo `arrays`, logo também aplicáveis a matrizes (consulte a Subseção 4.1.3). Vamos, aqui, introduzir outras operações próprias deste tipo de objeto.

Exercício 4.3. Aloque cada uma das seguintes matrizes como um Numpy `array`:

Exercício 4.4.

a)

$$A = \begin{bmatrix} -1 & 2 \\ 2 & -4 \\ 6 & 0 \end{bmatrix} \tag{33}$$

b) $B = A^T$

4.3.3 Inicialização de matrizes

Além das inicializações de `arrays` já estudadas na Subseção 4.1.1, temos mais algumas que são particularmente úteis no caso de matrizes.

- `np.eye(n)`: retorna a matriz identidade $n \times n$.

```
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

- `np.diag(v)`: retorna uma matriz diagonal formada pela `list v`.

```
>>> np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Exercício 4.5. Aloque a matriz escalar $C = [c_{ij}]_{i,j=0}^{99}$, sendo $c_{ii} = \pi$ e $c_{ij} = 0$ para $i \neq j$.

4.3.4 Multiplicação de matrizes

A multiplicação da matriz $A = [a_{ij}]_{i,j=0}^{n-1,l-1}$ pela matriz $B = [b_{ij}]_{i,j=0}^{l-1,m-1}$ e a matriz $C = AB = [c_{ij}]_{i,j=0}^{n-1,m-1}$ tal que

$$c_{ij} = \sum_{k=0}^{l-1} a_{ik}b_{k,j} \tag{34}$$

O NumPy tem a função `np.matmul()` para computar a multiplicação de matrizes. Por exemplo,

```
>>> C = np.matmul(A,B)
>>> C
array([[-50,  41],
       [ 14,   1]])
```

Observação 4.2. É importante notar que `np.matmul(A,B)` é a multiplicação de matrizes, enquanto que `*` consiste na multiplicação elemento a elemento.

Exercício 4.6. Aloque as matrizes

$$C = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 2 & 1 \\ 0 & -2 & -3 \end{bmatrix} \tag{35}$$

$$D = \begin{bmatrix} 2 & 3 \\ 1 & -1 \\ 6 & 4 \end{bmatrix} \tag{36}$$

$$E = \begin{bmatrix} 1 & 2 & 1 \\ 0 & -1 & 3 \end{bmatrix} \tag{37}$$

Então, se existirem, compute e forneça as dimensões das seguintes matrizes

a) CD

b) D^TE

c) D^TC

d) DE

4.3.5 Traço e Determinante de uma matriz

O NumPy tem a função `arr.trace()` para computar o **traço** de uma matriz (soma dos elementos de sua diagonal). Por exemplo,

```
>>> A = np.array([[-1,2,0],[2,3,1],[1,2,-3]])
>>> A.trace()
-1
```

Já, o **determinante** é fornecido no módulo `np.linalg`. Por exemplo,

```
>>> A = np.array([[-1,2,0],[2,3,1],[1,2,-3]])
>>> np.linalg.det(A)
25.000000000000007
```

Exercício 4.7. Compute e verifique os traços e os determinantes das seguintes matrizes

$$C = \begin{bmatrix} -2 & 3 \\ 1 & 4 \end{bmatrix} \tag{38}$$

$$D = \begin{bmatrix} 3 & 1 & -1 \\ 1 & 0 & 2 \\ 4 & 2 & -1 \end{bmatrix} \tag{39}$$

4.3.6 Rank e inversa de uma matriz

O **rank** de uma matriz é o número de linhas ou colunas linearmente independentes. O NumPy conta com a função `matrix_rank()` para computá-lo. Por exemplo,

```
>>> np.linalg.matrix_rank(np.eye(3))
3
>>> A = np.array([[1,2,3],[-1,1,-1],[0,3,2]])
>>> np.linalg.matrix_rank(A)
2
```

A inversa de uma matriz **full rank** pode ser computada com a função `np.linalg.inv()`. Por exemplo,

```
>>> A = np.array([[1,2,3],[-1,1,-1],[1,3,2]])
>>> np.linalg.matrix_rank(A)
```

```
3
>>> Ainv = np.linalg.inv(A)
>>> np.matmul(A,Ainv)
array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 1.11022302e-16,  1.00000000e+00,  2.22044605e-16],
       [-2.22044605e-16,  0.00000000e+00,  1.00000000e+00]])
```

Exercício 4.8. Compute, se possível, a matriz inversa de cada uma das seguintes matrizes

$$B = \begin{bmatrix} 2 & -1 \\ -2 & 1 \end{bmatrix} \tag{40}$$

$$C = \begin{bmatrix} -2 & 0 & 1 \\ 3 & 1 & -1 \\ 2 & 1 & 0 \end{bmatrix} \tag{41}$$

Verifique suas respostas.

4.3.7 Autovalores e autovetores de uma matriz

Um auto-par (λ, v), λ um escalar chamado de autovalor e $v \neq 0$ é um vetor chamado de autovetor, é tal que

$$A\lambda = \lambda v. \tag{42}$$

O NumPy tem a função `np.linalg.eig()` para computar os auto-pares de uma matriz. Por exemplo,

```
>>> np.linalg.eig(np.eye(3))
(array([1., 1., 1.]), array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]]))
```

Observamos que a função uma dupla, sendo o primeiro item um `array` contendo os autovalores (repetidos conforme suas multiplicidades) e o segundo item é a matriz dos autovetores, onde estes são suas colunas.

Exercício 4.9. Compute os auto-pares da matriz

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 3 & 2 & -1 \\ 2 & -1 & 1 \end{bmatrix}. \tag{43}$$

Então, verifique se, de fato, $A\lambda = \lambda v$ para cada auto-par (λ, v) computado.

5 Gráficos

Matplotlib é uma biblioteca Python livre e gratuita para a visualização de dados. É muito utilizada para a criação de gráficos estáticos, animados ou iterativos. Aqui, vamos introduzir alguma de suas ferramentas básicas para gráficos.

Para utilizá-la, é necessário instalá-la. Pacotes de instalação estão disponíveis para os principais sistemas operacionais, consule a sua loja de *apps* ou Matplotlib Installation. Para importá-la, usamos

```
>>> import matplotlib.pyplot as plt
```

Observação 5.1. Se você está usando um console Python remoto, você pode querer adicionar a seguinte linha de comando para que os gráficos sejam visualizados no próprio console.

```
>>> %matplotlib inline
```

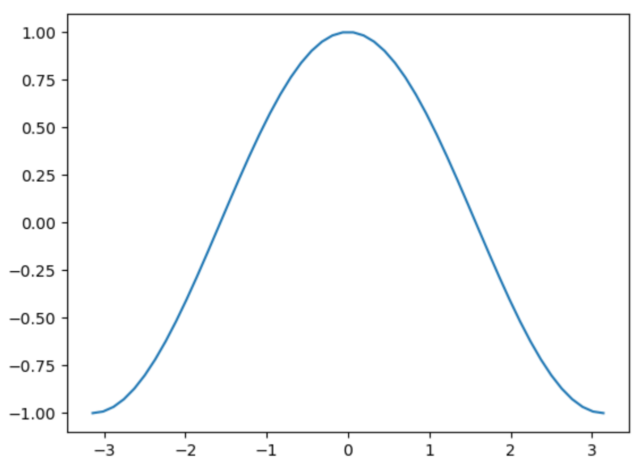

Figura 1: Esboço do gráfico da função $y = \text{sen}(x)$ no intervalo $[-\pi,\pi]$.

Gráficos bidimensionais podem ser criados com a função `plt.plot(x,y)`, onde `x` e `y` são `arrays` que fornecem os pontos cartesianos (x_i, y_i) a serem plotados. Por exemplo,

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-np.pi, np.pi)
>>> y = np.cos(x)
>>> plt.plot(x,y)
[<matplotlib.lines.Line2D object at 0x7f99f578a370>]
>>> plt.show()
```

produz o seguinte esboço do gráfico da função $y = \operatorname{sen}(x)$ no intervalo $[-\pi,\pi]$. Consulte a Figura 1.

Observação 5.2. Matplotlib é uma poderosa ferramenta para a visualização de gráficos. Consulte a galeria de exemplos no seu site oficial

`https://matplotlib.org/stable/gallery/index.html`

Exercício 5.1. Crie um esboço do gráfico de cada uma das seguintes funções no intervalo indicado:

a) $y = \cos(x)$, $[0, 2\pi]$

b) $y = x^2 - x + 1$, $[-2, 2]$

c) $y = \operatorname{tg}\left(\frac{\pi}{2}x\right)$, $(-1, 1)$

Referências

[1] **NumPy**. https://numpy.org/, 2021.

[2] **The Python Tutorial**. https://docs.python.org/3/tutorial/index.html, 2021.

[3] **Learn Python: simply easy learning**. https://www.tutorialspoint.com/python/index.htm, 2021.

[4] **SciPy**. https://www.scipy.org/, 2021.