

# Python para Físicos

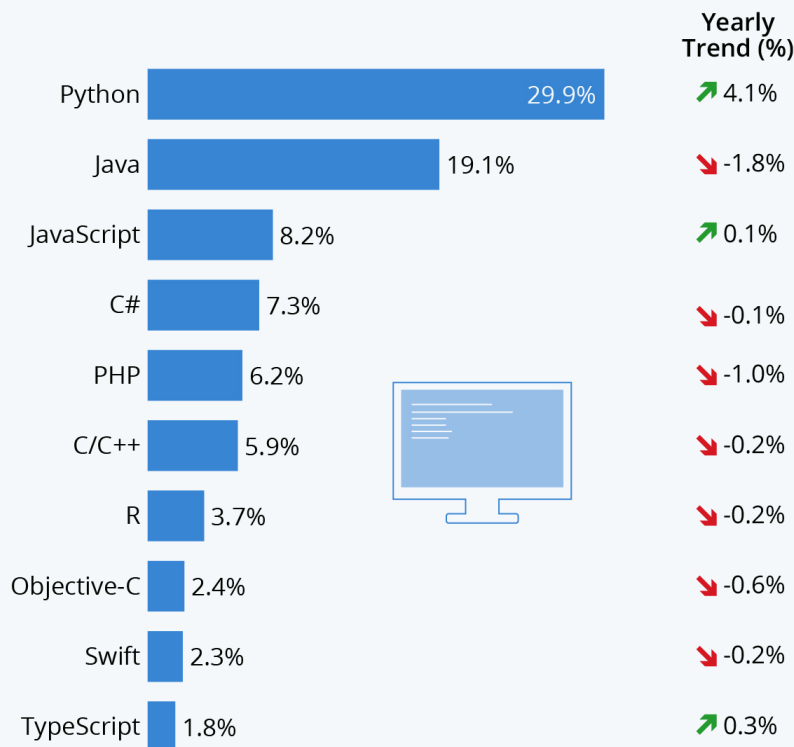
André Nepomuceno

Universidade Federal Fluminense

4 de setembro de 2023

## Python Remains Most Popular Programming Language

Popularity of each programming language based on share of tutorial searches in Google



Yearly trend compares percent change from Feb 2019 to Feb 2020  
Sources: GitHub, Google Trends



statista

Python é usado em diferentes áreas

- Ciência de Dados
- Inteligência Artificial
- Desenvolvimento Web
- Desenvolvimento de jogos
- Medicina e Farmacologia (AstraZeneca)
- Bioinformática
- Neurociência
- Física e Astronomia
- Business

## **Meu canal no YouTube: Python Para Cientistas**

`https://www.youtube.com/@python4scientists/videos`

## **Material do minicurso disponível em:**

`https:`

`//github.com/aanepomuceno/Mincurso-Python-Fisicos`

- Entre o site do Google Colab

`https://colab.research.google.com/notebooks/intro.ipynb?utm\_source=scs-index#recent=true`

- Escolha a opção **New Notebook**
- Renomeie o arquivo de `Untitled0` para um nome apropriado.

Em Python, uma **lista** é um conjunto ordenado de objetos que podem ser de vários tipos (inteiro, flutuante, complexo, booleano, string, etc.). Por exemplo, para criar uma lista, fazemos:

```
L = [1, 2.5, "Olá", True]
```

Cada entrada da lista é chamada de **elemento**, cada elemento tem uma **posição** na lista, e cada posição tem um inteiro associada a ela. Assim, o número (elemento) 1 está na posição zero da lista, o número 2.5 na posição um, e assim por diante. O **índice** que indica uma posição na lista sempre começa em zero.

Uma lista é um objeto **mutável**, e portanto podemos acrescentar ou retirar um elemento da lista.

Podemos também criar uma lista vazia: `L0 = []`.

# Listas

Um elemento da lista pode ser acessado pelo seu índice. O operador **in** pode ser usado para verificar se um dado elemento pertence a lista.

## Example

```
>>> L = [1, 2.5, 5.69, "x"]
>>> L[0]
1
>>> L[3]
'x'
>>> L[-1]
'x'
>>> 2 in L
False
>>> 'x' in L
True
```

# Listas - objetos mutáveis

Como lista são mutáveis, é possível modificar itens da lista.

## Example

```
>>> L = [1, 'dois', 3.14, 0]
>>> L[2] = 2.6
>>> L
[1, 'dois', 2.6, 0]
```

Atenção ao exemplo abaixo

## Example

```
>>> q1 = [1, 2, 3]
>>> q2 = q1
>>> q1[2] = 'x'
>>> q1
[1, 2, 'x']
>>> q2
[1, 2, 'x']
```

Existem vários métodos que podem ser usados com listas. Exemplos de alguns métodos:

- `append()` - adiciona um elemento ao final da lista.
- `insert()` - semelhante ao `append()`, mas podemos escolher a posição onde o novo elemento será alocado. Exemplo: `L.insert(1,4.56)`
- `remove()` - remove um elemento específico que está na lista. Exemplo: `L.remove(4.56)`
- `pop()` - remove um elemento da lista, dado sua posição. Exemplo: `L.pop(1)` vai remover o elemento que está na posição “1” da lista, ou seja, o segundo elemento. `L.pop()` remove o último elemento da lista.
- `index()` - retorna o índice da primeira ocorrência de um elemento da lista (posição do elemento). Exemplo: `L.index(2.5)`
- `sort()` - ordena os elementos de uma lista em ordem crescente.
- `reverse()` - inverte a ordem dos elementos da lista.



# Listas - Exemplos

## Example

```
>>> import math
>>> L = []
>>> for i in range(5):
>>>     L.append( round(math.sqrt(i**2.5), 2) )
>>> L
[0.0, 1.0, 2.38, 3.95, 5.66]
>>> L.insert(1, 5.3)
>>> L
[0.0, 5.3, 1.0, 2.38, 3.95, 5.66]
>>> L.sort()
>>> L
[0.0, 1.0, 2.38, 3.95, 5.3, 5.66]
```

# NumPy Arrays

**NumPy** é o pacote padrão para programação científica em Python. O módulo NumPy implementa de forma eficiente operações matemáticas. Para usar os métodos do módulo, devemos importá-lo no início do programa:

```
import numpy as np
```

Os objetos do NumPy são **arrays**, que é um conjunto ordenado de valores, mas que possuem diferenças cruciais em relação a listas:

- O número de elementos de um array é **fixo**. Não se pode adicionar ou remover itens de um array.
- Os elementos de um array são todos do mesmo tipo.
- Arrays podem ter  $n$  dimensões. Por exemplo, arrays com  $n = 2$  são matrizes.
- Operações com arrays são **mais rápidas** do que com listas.

# Criando Arrays

Vamos ver diversas formas de criar um array.

## Array a partir de listas

```
>>> a = np.array([1., 2, 3.1])
>>> a
array([1. , 2. , 3.1])
>>> a[0]
1.0
>>> b = np.array([ [1., 2.], [3., 4.] ]) #2D array
>>> b
array([[1., 2.],
       [3., 4.]])
>>> b[0, 0]
1.0
>>> b[1, 0]
3.0
```

# Criando Arrays

## Array com todas as entradas iguais a zero

```
>>> np.zeros(5)
array([0., 0., 0., 0., 0.])
>>> np.zeros(5, dtype=int)
array([0, 0, 0, 0, 0])
```

## Array com todas as entradas iguais a um

```
>>> np.ones(6)
array([1., 1., 1., 1., 1., 1.])
>>> np.ones((3,4))      #matrix 3 x 4
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

# Criando Arrays

## Array com todas as entradas iguais a um dado valor

```
>>> np.full(4, 3.14)
array([3.14, 3.14, 3.14, 3.14])
>>> np.full((4, 3), 3.14)
array([[3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14]])
```

## Array como matrix identidade

```
>>> np.eye(3)          #mesmo que np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

# Criando Arrays

## Criando array com o método `arange()`

```
>>> np.arange(7)
array([0, 1, 2, 3, 4, 5, 6])
>>> np.arange(1.5, 3.0, 0.5)
array([1.5, 2. , 2.5])
>>> np.arange(6.5, 0, -1)
array([6.5, 5.5, 4.5, 3.5, 2.5, 1.5, 0.5])
```

A sintaxe do método **`arange()`** é `np.arange(inicio, fim, passo)`. Se apenas um número for dado, por exemplo, `np.arange(N)`, será criado um array de zero até o valor `N-1`, com passo de um.

# Criando Arrays

A função `np.linspace(x, y, N)` gera  $N$  números entre  $x$  e  $y$ , com  $y$  **incluso**.

## Criando array com o método `linspace()`

```
>>> np.linspace(0, 10, 6)
array([ 0.,  2.,  4.,  6.,  8., 10.])
>>> z, dz = np.linspace(0., 2*np.pi, 100, retstep=True)
>>> dz
0.06346651825433926
```

A opção `retstep = True` retorna o tamanho do passo.

## Warning

Note a diferença entre `arange()` e `linspace()`. Use `linspace()` sempre que desejar um array de tamanho precisamente  $N$ .

# Arrays - Atributos

## Atributos de um array

```
>>> a = np.array([ [1,0,1], [1,2,2] ])
>>> a.shape
(2, 3)
>>> a.ndim
2
>>> a.size
6
>>> a.dtype
dtype('int64')
>>> a.nbytes
48
```



# Operações com Arrays

O grande poder do NumPy reside na realização de operações em todos os elementos de um array sem a necessidade de *loops* explícitos. Esse tipo de operação é chamada **vetorização**, e é muito mais rápida que *for loops*.

## Example

```
>>> a = np.array([1.3, 2.5, 10.1])
>>> b = np.array([9.3, 0.2, 1.2])
>>> a + b
array([10.6,  2.7, 11.3])
>>> a*b
array([12.09,  0.5, 12.12])
>>> a/b
array([0.13978495, 12.5,  8.41666667])
>>> a/b + 1
array([ 1.13978495, 13.5,  9.41666667])
>>> a**2
array([1.69,  6.25, 102.01])
```

# Operações com Arrays

## Produtos

```
>>> a = np.array( [1.,2.,3.])
>>> b = np.array( [2.,4.,5.])
>>> np.dot(a,b) # produto interno, (mesmo que a @ b)
25.0
>>> np.cross(a,b) #produto vetorial
array([-2.,  1.,  0.] )
```

## Operadores de comparação e lógica

```
>>> a = 2*np.linspace(1,6,6)
>>> a
array([ 2.,  4.,  6.,  8., 10., 12.])
>>> t = a > 10
>>> t
array([False, False, False, False, False,  True])
```

# Operações com Arrays

**Exemplo:** Vamos implementar o cálculo abaixo:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} + 2 \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} -3 & 5 \\ 0 & 2 \end{pmatrix}$$

## Código

```
>>> a = np.array([ [1,3], [2,4] ])
>>> b = np.array([ [4,-2], [-3,1] ])
>>> c = np.array([ [1,2], [2,1] ])
>>> r = np.dot(a,b) + 2*c
>>> r
array([[ -3,   5],
       [  0,   2]])
```

# Operações com Arrays - Funções

As funções disponíveis no módulo math também existem no NumPy. Teste os exemplos abaixo.

## Funções

```
theta = np.linspace(0.1,np.pi,4)
print("theta = ", theta)
print("sen(theta) = ",np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("ln(theta) = ",np.log(theta))
print("log(theta) = ",np.log10(theta))
print("exp(theta) = ", np.exp(theta))
print("modulo = ", np.absolute(np.log(theta)))
```

# Arrays - Funções de Agregação

Quando trabalhamos com um grande conjunto de dados, é comum calcularmos estatísticas para uma análise inicial. NumPy oferece vários métodos para essa tarefa.

## Example

```
#soma, media, max. e min.  
>>> data = np.random.random(100)  
>>> data.sum()  
51.72239489031435  
>>> data.mean()  
0.5172239489031435  
>>> data.max()  
0.9946152525979709  
>>> data.min()  
0.0023509304159052835
```

# Arrays - Funções de Agregação

Para arrays em n-dimensões ( $n > 1$ ), podemos escolher o *eixo* sobre o qual os valores serão agregados.

## Example

```
#2D array
>>> M = np.random.random( (3, 4) )
>>> M
array([[0.37019599, 0.15892146, 0.23032805, 0.37...],
       [0.17968684, 0.69242006, 0.51502879, 0.06...],
       [0.78280796, 0.63324658, 0.22553994, 0.94...]])
>>> M.sum()
5.168637614536063
>>> M.sum(axis=0)
array([1.33269079, 1.4845881 , 0.97089679, 1.380...])
>>> M.max(axis=1)
array([0.37897901, 0.69242006, 0.94005747])
```

# Arrays - Funções de Agregação

Método	Descrição
<code>np.sum</code>	soma dos elementos
<code>np.cumsum</code>	soma cumulativa dos elementos
<code>np.prod</code>	produto dos elementos
<code>np.mean</code>	valor médio
<code>np.std</code>	desvio padrão
<code>np.var</code>	variância
<code>np.min</code>	valor mínimo
<code>np.max</code>	valor máximo
<code>np.argmin</code>	<b>índice</b> do valor mínimo
<code>np.argmax</code>	<b>índice</b> do valor máximo
<code>np.conj</code>	complex. conjugado de todos elementos
<code>np.trace</code>	soma dos elementos da diagonal

Veja mais detalhes neste [LINK](#).

# Arrays - Slicing

Muitas vezes precisamos obter um “subarray” a partir de um array, ou seja, um array com apenas alguns elementos do array original. Para isso, existe uma técnica chamada **slicing**. A sintaxe é:

**[início:fim:passo]**

onde “início” é o índice (posição) da primeira entrada desejada, e “fim” o índice do último elemento, que **NÃO** entrará no novo array. Esse comando vai gerar um array com entradas  $a[\text{início}]$ ,  $a[\text{início} + \text{passo}]$ ,  $a[\text{início} + 2 * \text{passo}]$ , ...  $a[\text{início} + N * \text{passo}]$ , com a posição “ $\text{início} + N * \text{passo}$ ”  $< \text{fim}$ .

O array que retorna dessa operação **não** é um cópia, ou seja, não é um novo objeto.



# Arrays - Slicing

## Example

```
>>> a = np.linspace(1,6,6); a
array([1., 2., 3., 4., 5., 6.])
>>> a[:3]      #mesmo que a[0:3]
array([1., 2., 3.])
>>> a[1:4:2]
array([2., 4.])
>>> a[1:]
array([2., 3., 4., 5., 6.])
>>> a[3::-2]
array([4., 2.])
>>> a[::-1]
array([6., 5., 4., 3., 2., 1.])
```

# Arrays - Exemplo

**Exemplo:** Dados dois arrays de posição  $x$  e tempo  $t$  de uma partícula, calcule a velocidade média  $\bar{v}$  para cada intervalo de tempo, utilizando *slicing*.

```
x = np.array([0., 1.3, 5. , 10.9, 18.9, 28.7, 40.])  
t = np.array([0., 0.49, 1. , 1.5 , 2.08, 2.55, 3.2])
```

Com

$$\bar{v} = \frac{x_i - x_{i-1}}{t_i - t_{i-1}}$$

# Importando e Exportando Dados

## Abrindo arquivos com NumPy

Para abrir arquivos de dados dos tipos `.txt`, `.dat` ou `.csv`, podemos usar o métodos **`np.loadtxt()`**. Os dados serão transformados num array. Como default, é assumido que os dados estão separados por espaços ou tabulação.

```
import numpy as np
data_set = np.loadtxt("millikan.txt")
data_x = data_set[:, 0]
data_y = data_set[:, 1]
```

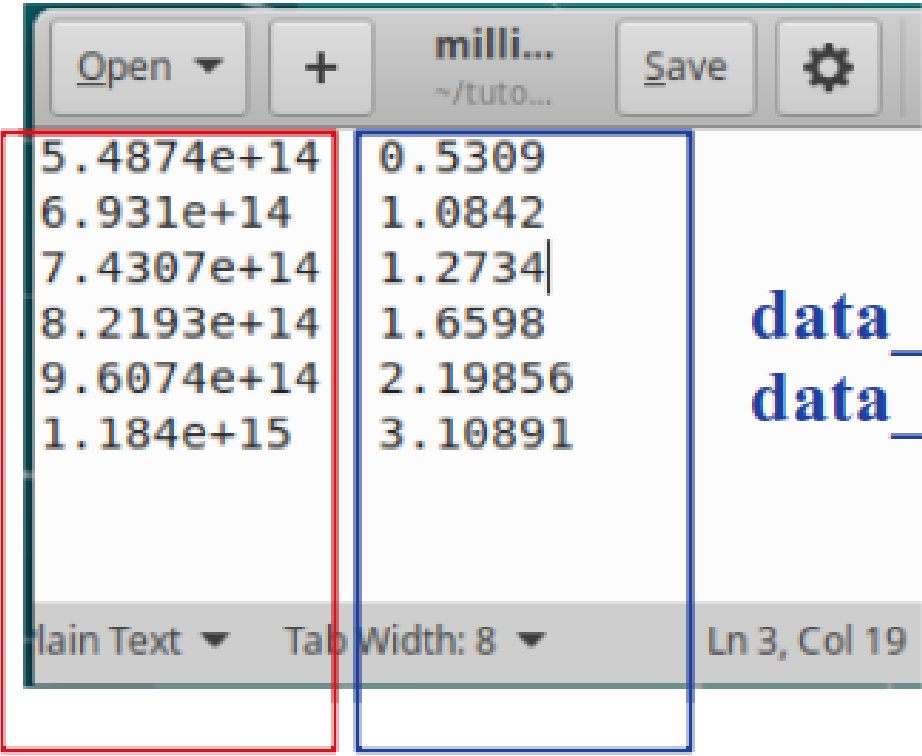
Se os valores estiverem separados por um caractere, ele dever ser especificado usando a palavra chave `delimiter`.

```
data_set = np.loadtxt("millikan.csv", delimiter=',')
```

# Importando e Exportando Dados

A figura ilustra o exemplo acima.

**data\_set**



5.4874e+14	0.5309
6.931e+14	1.0842
7.4307e+14	1.2734
8.2193e+14	1.6598
9.6074e+14	2.19856
1.184e+15	3.10891

**data\_x**  
**data\_set[:,0]**

**data\_y**  
**data\_set[:,1]**

# Interlúdio: Gráficos Simples

Importar o módulo

**import matplotlib.pyplot as plt**

Se quisermos fazer um gráfico de uma função, as entradas para o pyplot devem ser arrays (ou listas) correspondentes aos valores x e y. Exemplo:

## Exemplo 1 - Gráfico simples

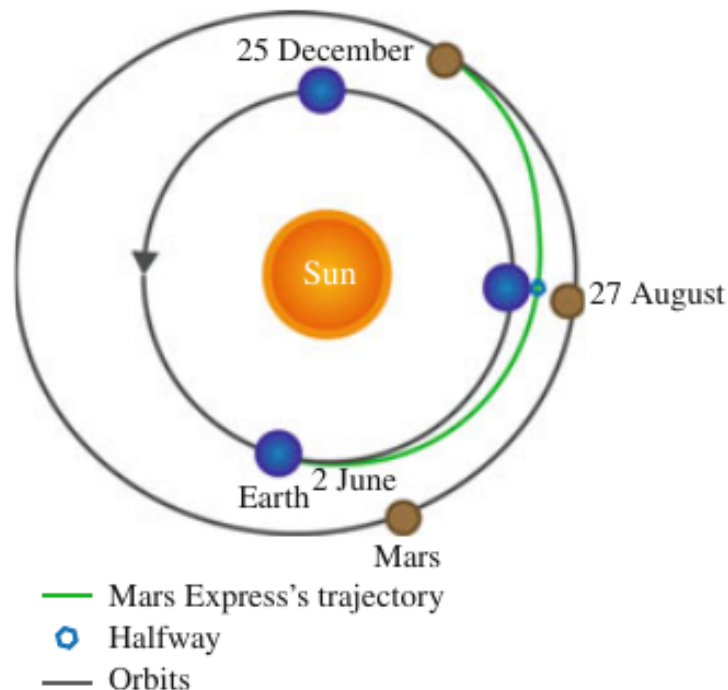
```
x = np.linspace(-3*np.pi, 3*np.pi, 100)
y = np.sin(x)      #vetorização
plt.plot(x, y)
```

Para adicionar um segundo plot, basta chamar `plt.plot` novamente:

```
z = np.cos(x)
plt.plot(x, z)      #ou plt.plot(x, y, x, z)
```

# Exemplo de Aplicação I

**Exemplo 1** A sonda *Mars Express* foi lançada em junho de 2003 e chegou em Marte em Dezembro de 2003. Os valores da posição da sonda em função do tempo estão no arquivo `marsexpress.dat`, com o tempo dado em dias e as posições em quilômetros. Vamos utilizar unidades astronômicas (AU), sendo  $1 \text{ AU} = 149\,598\,000 \text{ km}$ .



A partir dos dados, vamos calcular aproximadamente os vetores velocidade e aceleração da sonda:

$$\mathbf{v}(t_i) \approx \frac{\mathbf{r}(t_{i+1}) - \mathbf{r}(t_i)}{\Delta t}$$

$$\mathbf{a}(t_i) \approx \frac{\mathbf{r}(t_{i+1}) - 2\mathbf{r}(t_i) + \mathbf{r}(t_{i-1}))}{\Delta t^2}$$

# Álgebra Linear com NumPy - Operações com Matrizes

## Multiplicação dos **Elementos** das Matrizes

```
In[x]: A * B
Out[x]: array([[ 0. , -0.25],
               [-3.,  3.  ]])
```

## Matriz Transposta

```
In[x]: A.T    #ou A.transpose()
Out[x]: array([[ 0. , -1. ],
               [ 0.5,  2. ]])
```

## Matriz Identidade

```
In[x]: np.eye(3,3)
Out[x]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

# Álgebra Linear com NumPy - Operações com Matrizes

## Potência de Matrizes

```
In[x]: np.linalg.matrix_power(A, 3)
Out[x]: array([[ -1.   ,  1.75],
               [-3.5   ,  6.   ]])
```

## Potência dos Elementos

```
In[x]: A**3
Out[x]: array([[ 0.   ,  0.125],
               [-1.   ,  8.   ]])
```



# Álgebra Linear com NumPy - Normas e *Rank*

Normas são calculadas com o módulo `np.linalg.norm`. O *rank* (posto) é obtido pelo método `np.linalg.matrix_rank`.

## 1. Norma de um Vetor

$$\|a\| = \left( \sum_i |z_i|^2 \right)^{1/2}$$

## 2. Norma de Frobenius

$$\|A\| = \left( \sum_{i,j} |a_{ij}|^2 \right)^{1/2}$$

## 3. *Rank*: número de colunas linearmente independentes.

# Álgebra Linear com NumPy - Normas e *Rank*

## Cálculo de Normas

```
In[x]: np.linalg.norm(A)
Out[x]: 2.29128784747792
In[x]: c = np.array([1, 2j, 1-1j])
        np.linalg.norm(c)
Out[x]: 2.6457513110645907
```

## Cálculo do Rank

```
In[x]: np.linalg.matrix_rank(A)
Out[x]: 2
In[x]: D = np.array([[1, 1], [2, 2]])
Out[x]: array([[1, 1],
               [2, 2]])
In[x]: np.linalg.matrix_rank(D)
Out[x]: 1
```

# Álgebra Linear com NumPy - Determinante e Inversa

## Determinante

```
In[x]: np.linalg.det(A)  
Out[x]: 0.5
```

## Traço

```
In[x]: np.trace(A)  
Out[x]: 2
```

## Matriz Inversa

```
In[x]: np.linalg.inv(A)  
Out[x]: array([[ 4., -1.],  
               [ 2.,  0.]])
```

Se a matriz não tiver inversa, será retornado o erro

**LinAlgError: Singular matrix**

# Álgebra Linear com NumPy - Autovalores e Autovetores

## Problema de autovalor

Para uma matriz quadrada  $\mathbf{A}$ , um *autovetor*  $\mathbf{v}$  é um vetor que satisfaz

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

onde  $\lambda$  são chamados *autovalores*. Para um matriz  $N \times N$ , existem  $N$  autovetores e  $N$  autovalores.

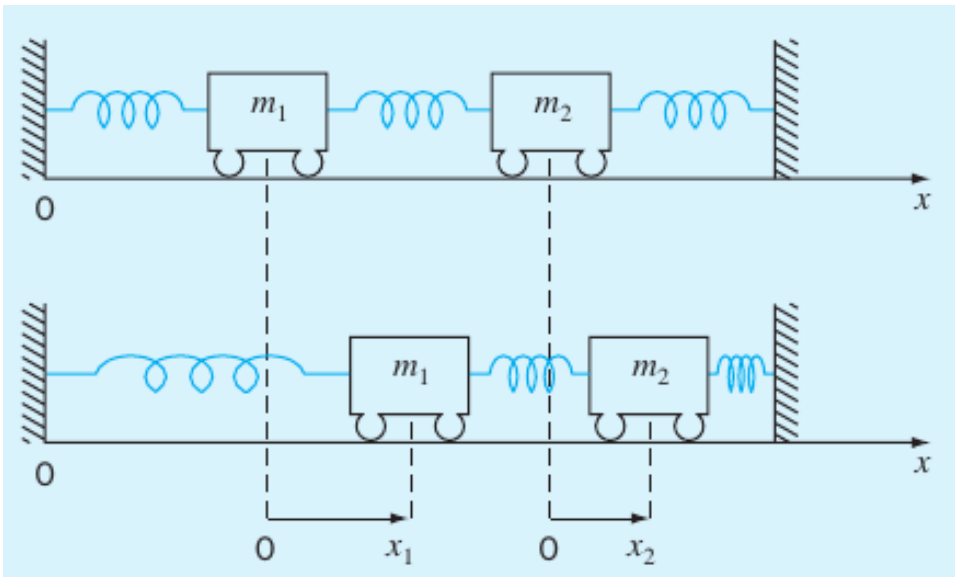
Para calcular autovetores e autovalores existe o módulo `np.linalg.eig`, que retorna os autovalores como um array de forma  $(n,)$  e os autovetores como **colunas** de um array de forma  $(n, n)$ . Use `np.linalg.eigval` para calcular os autovalores apenas.

## Autovalores e Autovetores

```
In[x]:  vals, vecs = np.linalg.eig(A)
        print(vals)
Out[x]: [0.29289322  1.70710678]
```

# Exemplo de Aplicação II

**Exemplo 2** No sistema massa-mola abaixo, vamos assumir que as molas tem os mesmos comprimentos naturais e as mesmas constantes  $k$ . O deslocamento de cada mola é medido em relação ao seu próprio sistema de coordenada.



Aplicando a segunda lei de Newton:

$$m_1 \frac{dx_1^2}{dt^2} = -kx_1 + k(x_2 - x_1)$$

$$m_2 \frac{dx_2^2}{dt^2} = -k(x_2 - x_1) - kx_2$$

# Exemplo de Aplicação II

A solução é dada por

$$x_i = X_i \sin(\omega t)$$

Substituindo nas equações anteriores:

$$\left( \frac{2k}{m_1} - \omega^2 \right) X_1 - \frac{k}{m_1} X_2 = 0$$

$$-\frac{k}{m_2} X_1 + \left( \frac{2k}{m_2} - \omega^2 \right) X_2 = 0$$

Vamos considerar o caso  $m_1 = m_2 = 40$  kg, e  $k = 200$  N/m.

# Álgebra Linear com NumPy - Sistemas Lineares

NumPy dispõe de um método eficiente e estável para resolver sistemas de equações lineares: `np.linalg.solve`. Exemplo: o sistema abaixo

$$\begin{aligned}3x - 2y &= 8, \\ -2x + y - 3z &= -20, \\ 4x + 6y + z &= 7\end{aligned}$$

pode ser escrito como uma equação matricial  $\mathbf{M}\mathbf{x} = \mathbf{b}$

$$\begin{pmatrix} 3 & -2 & 0 \\ -2 & 1 & -3 \\ 4 & 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -20 \\ 7 \end{pmatrix}$$

## Solução de Sistemas Lineares

```
In[x]: M = np.array([ [3., -2, 0], [-2, 1, -3],  
                     [4, 6, 1]])
```

```
In[x]: print(M)
```

```
Out[x]: [[ 3. -2.  0.]  
         [-2.  1. -3.]  
         [ 4.  6.  1.]
```

```
In[x]: b = np.array([8, -20, 7])
```

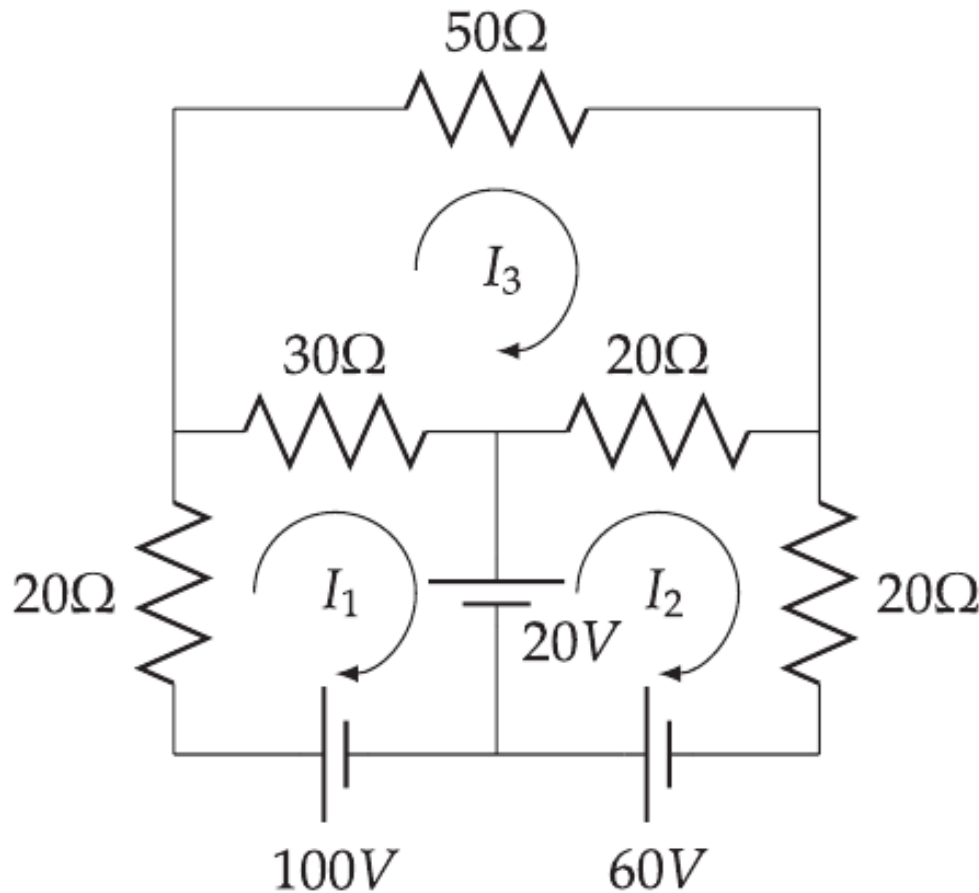
```
In[x]: x, y, z = np.linalg.solve(M,b)  
       print('x = {}, y = {}, z = {}'.format(x,y,z))
```

```
Out[x]: x = 2.0, y = -1.0, z = 5.0
```



# Exemplo de Aplicação III

**Exemplo 3** No circuito abaixo, determine os valores das correntes  $I_1$ ,  $I_2$ ,  $I_3$ .



Vamos aplicar a 2ª lei de Kirchhoff ( $\sum_k V_k = 0$ ) e a lei de Ohm ( $V = RI$ ) ao circuito:

$$50I_1 - 30I_3 = 80$$

$$40I_2 - 20I_3 = 80$$

$$-30I_1 - 20I_2 + 100I_3 = 0$$