

# Python para Computação Numérica

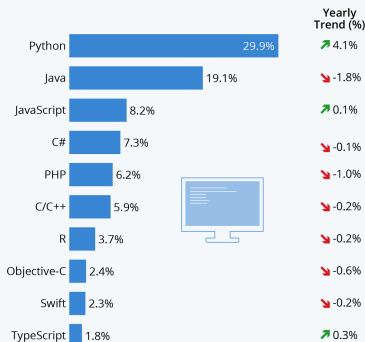
André Nepomuceno

Universidade Federal Fluminense

18 de outubro de 2021

## 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

## Pacotes abordados neste minicurso:

- NumPy
- Matplotlib
- SciPy

Você pode executar os códigos online:

- 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.

**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(início, 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  $N$ .

## 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]])
```

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  $a[\text{início}]$ ,  $a[\text{início} + \text{passo}]$ ,  $a[\text{início} + 2 * \text{passo}]$ ,  $\dots$ ,  $a[\text{início} + N * \text{passo}]$ , com a posição “ $\text{início} + N * \text{passo}$ ”  $< \text{fim}$ .

O array que retorna dessa operação **não** é uma 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.])
```

## 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`.

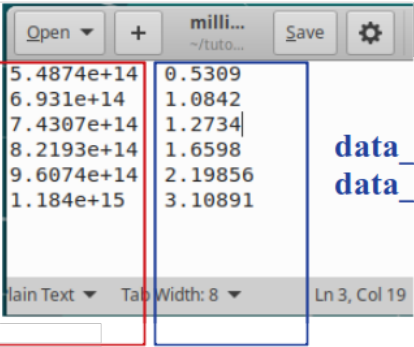
```
data_set = np.loadtxt("millikan.csv", delimiter=',')
```



# Importando 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]**

# Importando e Exportando Dados

Salvando dados em um arquivo.

## Usando o método `savetxt()`

```
x = np.linspace(0,1,100)
y = 3*np.sin(x)**3 - np.sin(x)
np.savetxt("x_valores.dat",x)
np.savetxt("xy_values.dat",list(zip(x,y)),fmt="%8.3f")
```

A última linha salva os valores 'x' e 'y' num mesmo arquivo.

## Usando *Loops*

```
my_file = open("xy_values.txt","w") #w=writing
for i in range(x.size):
    my_file.write("{:f}\t\t{:f}\n".format(x[i],y[i]))
my_file.close()
```

## 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.]])
```

## 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.

## 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**

## 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]
```



# 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  $\mathbf{Mx} = \mathbf{b}$

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

## Solução de Sistemas Lineares

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

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

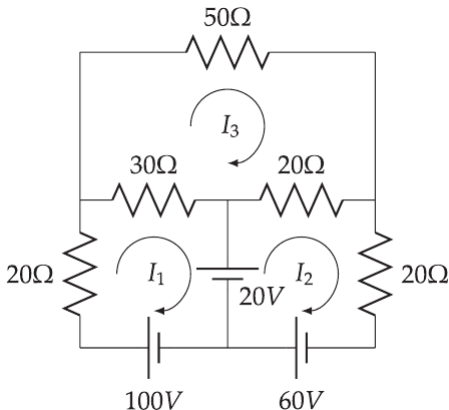
```
Out[x]: [[ 3. -2.  0.]  
         [-2.  1. -3.]  
         [ 4.  6.  1.]]
```

```
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
```

**Exercício.** 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:

$$50I_1 - 30I_3 = 80$$

$$40I_2 - 20I_3 = 80$$

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

Python tem uma poderosa biblioteca para produção de gráficos de boa qualidade: **Matplotlib**. Para gráficos simples, podemos usar o módulo **pyplot** que deve ser importado da seguinte forma:

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

No jupyter, para que o gráfico apareça numa célula do notebook, digite na primeira célula: `%matplotlib inline`. Para que o gráfico seja mostrado numa janela separada, digite na primeira célula: `%matplotlib`.

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)
```

Para nomear um gráfico, devemos atribuir um string ao argumento `label` da função `plot`. Para adicionar a legenda no gráfico, faça:

```
plt.legend()
```

## Exemplo 2 - Legenda

```
x = np.linspace(-3*np.pi, 3*np.pi, 100)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, label='sen(x)')
plt.plot(x, z, label='cos(x)')
plt.legend()
```

Para retirar a legenda da "caixa", use a opção `frameon=False`. Para seleccionar o tamanho da fonte, use `fontsize=<inteiro>`.

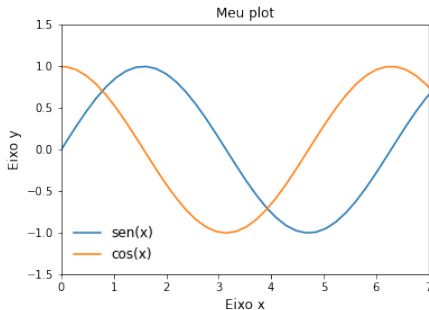
## Opções de localização da legenda

String	Inteiro
'best'	0
'upper right'	1
'upper left'	2
'lower left'	4
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'upper center'	10

# Gráficos - Descrição dos Eixos e Título

## Exemplo 3 - Eixos e Título

```
plt.title('Meu plot')  
plt.xlabel('Eixo x', fontsize=12)  
plt.ylabel('Eixo y', fontsize=12)  
plt.xlim(0, 7.)      #x-range  
plt.ylim(-1.5, 1.5)  #y-range
```





# Gráficos - Marcadores, Cores e Linhas

Existem diversas opções de marcadores, linhas e cores, que devem ser especificados por strings. Por exemplo, se quisermos linha vermelha tracejada, basta incluir 'r -' na função plot.

## Exemplo 3 - Cores e Linhas

```
plt.plot(x,y, 'r--', label='sen(x)')  
plt.plot(x,y, 'r--o', label='sen(x)')    #marcador 'o'
```

Também é possível passar os atributos explicitamente com `c` (color), `marker` (marcador) `ls` (estilo da linha) e `lw` (largura da linha)

```
plt.plot(x,y, c='r', marker='o', ls='--', lw=2)
```

Também é possível selecionar o tamanho do marcador (`markersize`), a cor (`markerfacecolor` ou `mfc`), e a cor da borda (`markeredgecolor` ou `mec`).

# Gráficos - Marcadores, Cores e Linhas

## Marcadores

Código	Marcador
.	Ponto
o	Círculo
+	Cruz
x	Cruzado
D	Diamante
v	Triângulo p/ baixo
^	Triângulo p/ cima
s	Quadrado
*	Estrela

## Cores Básicas

Código	Cor
r	Vermelho
g	Verde
b	Azul
c	ciano
m	magenta
y	Amarelo
k	Preto
w	Branco
brown	Marrom
gray	Cinza
purple	Roxo

**Estilos de linha:** (-) (—) (:) (-.)

O comando `plt.twinx()` cria um novo eixo *y* mantendo o mesmo eixo *x* (também existe a opção `plt.twiny()`).

## Example

```
line1 = plt.plot(tempo, divorcios, 'b-o')
plt.ylim(4, 5.2)
plt.ylabel('Divorcios por 100 mil')
plt.xlabel('Anos')
plt.twinx()
line2 = plt.plot(tempo, margarina, 'r-v')
plt.ylabel('Consumo de Margarina [lb]')
lines = line1 + line2
legendas = ['Divorcios', 'Consumo de Margarina']
plt.legend(lines, legendas, frameon=False)
```

# Matplotlib - Interface Avançada

O Matplotlib tem uma interface básica, semelhante ao MATLAB, que permite fazer vários gráficos simples. No entanto, para ter maior controle sobre os elementos do gráfico, existe uma interface orientada a objetos. Nessa interface, criamos um objeto chamado *figure*, que pode ser pensando como um contêiner que contém todos os objetos relacionados aos eixos, gráficos e descrições. Acoplamos ao *figure* o objeto *axes*, que contém todos os elementos do gráfico.

## Criando uma Figura

```
fig = plt.figure()  
ax = fig.add_subplot()
```

Também é possível criar os dois objetos numa única linha

```
fig, ax = plt.subplots()
```

# Matplotlib - Interface Avançada

O objeto *figure* tem vários argumentos opcionais, como identificador e tamanho da figura.

Argumento	Descrição
<code>num</code>	String identificador da figura
<code>figsize</code>	Tupla com as dimensões da figura (largura, comprimento), em polegadas
<code>dpi</code>	Resolução da figura (pts por polegada)
<code>facecolor</code>	Cor de fundo
<code>edge color</code>	Cor da borda

## Example

```
fig = plt.figure("Figural", figsize=(4.5,3),  
                 facecolor='r')
```

Os elementos de texto de um plot podem ser modificados com as opções da tabela abaixo.

Argumento	Descrição
<code>fontsize</code>	Tamanho da fonte
<code>fontname</code>	Nome (ex. 'Arial')
<code>family</code>	Família (ex. 'cursive')
<code>fontweight</code>	Peso (ex. 'normal', 'bold')
<code>fontstyle</code>	Estilo (ex. 'normal', 'italic')
<code>color</code>	Cor da fonte

# Matplotlib - Gridlines e Escala Log

Podemos adicionar linhas de grade aos eixos vertical e/ou horizontal. É possível escolher o estilo da linha e cor (`linestyle`, `linewidth`, `color`, etc.).

## Gridlines

```
ax.grid(True) #linhas horizontais e verticais  
ax.yaxis.grid(True) #apenas horizontal
```

Escala logarítmica pode ser escolhida com os comandos abaixo. Por padrão, logaritmo decimal é usado, mas podemos escolher outra base com os argumentos `basex`, `basey`.

## Escala Log

```
ax.set_xscale('log')  
ax.set_yscale('log')
```

*Subplots* são grupos de gráficos que pertencem a uma mesma figura. Existem diferentes formas de criar subplots no Matplotlib.

## Métodos `plt.subplots()`

```
fig, axes = plt.subplots(nrows=3, ncols=2)
fig.tight_layout()
```

Será criada uma figura com seis gráfico ( $3 \times 2$ ), e cada um pode ser acessado com índices semelhantes a elementos de matrizes:

```
ax1 = axes[0, 0]    #superior esquerdo
ax2 = axes[2, 1]    #inferior direito
```

É possível escolher o tamanho da figura passando para a função `plt.subplots()` o argumento `figsize=(<int>, <int>)`.



Para ajustar a distância entre os gráficos, usamos

```
fig.subplots_adjust(hspace=<float>, wspace=<float>).
```

## Example

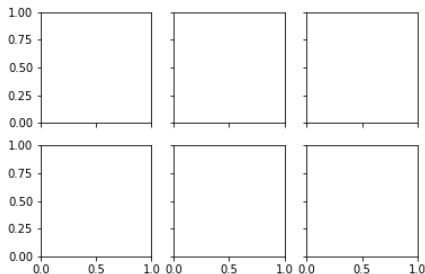
```
fig, axes = plt.subplots(2,1)
fig.subplots_adjust(hspace=0.05)
x = np.linspace(0,2*np.pi,100)
axes[0].plot(x,np.sin(x))
axes[0].set_xticks([])
axes[1].plot(x,np.cos(x))
```

# Matplotlib - Subplots

Podemos ocultar automaticamente os labels internos dos gráficos usando os argumentos `sharex` e `sharey` da função `plt.subplots()`.

## Example

```
fig, ax = plt.subplots(2, 3, sharex='col',  
                        sharey='row')
```



# Matplotlib - Scatter Plots

A função `pyplot.scatter()` permite criar gráficos de dispersão onde as propriedades de cada ponto (cor, tamanho) podem ser controladas. Além dos valores `x` e `y`, podemos passar uma sequência de valores para os argumentos `s` e `c`, que controlam o tamanho e a cor da cada ponto. Esse tipo de gráfico é útil para visualizar dados multidimensionais.

## Example

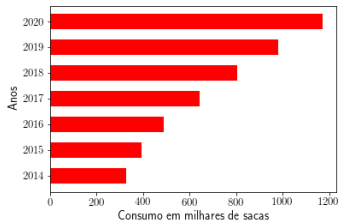
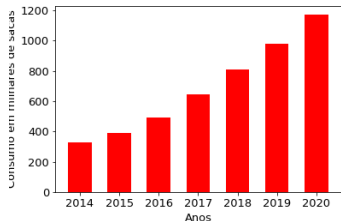
```
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000*np.random.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)
plt.colorbar()
```

# Matplotlib - Gráfico de Barras

A função para plotar gráficos de barras é `plt.bar()`. Para barras horizontais, usamos `plt.barh()`. Veja as opções de argumento em [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.bar.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html)

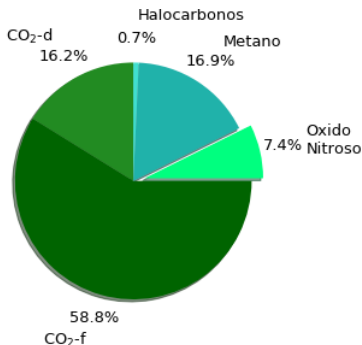
## Example

```
anos = np.arange(2014, 2021)
consumo = np.array([327, 392, 490, 643, 806, 981, 1171])
plt.bar(anos, consumo, width=0.6, color='r')
plt.barh(anos, consumo, height=0.6, color='r')
```



# Matplotlib - Gráfico de Pizza

Gráficos de pizza são feitos com o método `plt.pie()`. Os valores de cada entrada são automaticamente normalizados pela soma. As várias opções estão descritas em [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.pie.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.pie.html)  
Veja exemplo em `pcientifico_matplotlib01.ipynb`



# Matplotlib - Salvando Figuras

Para salvar gráficos de boa qualidade (para publicações, por exemplo), os melhores formatos são *Encapsulated Encapsulated* (EPS) ou PDF:

```
plt.savefig('my_figure.eps')
```

Também é possível especificar a resolução com o argumento `dpi` (*dots per inch*) tanto na função `plt.figure()` como na função `plt.savefig()`, e usar o formato PNG:

```
fig = plt.figure (figsize =(3.5, 3), dpi=300)
ax = fig.add_subplot()
.....
plt.savefig('my_figure.png', dpi=300)
```

O pacote `scipy.optimize` implementa vários métodos para calcular raízes de funções. Os argumentos passados devem ser uma função contínua,  $f(x)$ , e um intervalo  $[a, b]$  dentro do qual a raiz será encontrada, tal que  $\text{sgn}[f(a)] = -\text{sgn}[f(b)]$ . Alguns dos métodos disponíveis:

- Método de Brent (`scipy.optimize.brentq`)
- Método da bisseção (`scipy.optimize.bisect`)
- Método de Newton (`scipy.optimize.newton`)

No caso do método Newton–Raphson, deve-se passar um ponto inicial,  $x_0$  (próximo a raiz), e opcionalmente, a primeira derivada da função, `fprime`. Note que nesse método, temos menos controle sobre a raiz encontrada se a função tem várias raízes.

Métodos numéricos devem ser utilizados com cuidado. Verifique se a raiz  $x$  encontrada produz  $f(x) \approx 0$ .

## Raízes - Método de Brent

Vamos encontrar uma das raízes da função abaixo pelo método de Brent

$$f(x) = \frac{1}{5} + x \cos\left(\frac{3}{x}\right).$$

```
In[x]: from scipy.optimize import brentq
In[x]: f = lambda x: 0.2 + x*np.cos(3/x)
In[x]: print(np.sign(f(-0.7)))
        print(np.sign(f(-0.5)))
In[x]: brentq(f, -0.7, -0.5)
Out[x]: -0.5933306271014237
```



## Raízes - Método de Newton

Vamos encontrar a raiz da função abaixo pelo método de Newton

$$f(x) = e^x - 2$$

```
In[x]: from scipy.optimize import newton
In[x]: f = lambda x: np.exp(x) - 2
In[x]: fprime = lambda x: np.exp(x)
In[x]: x0 = 2
In[x]: xsol = newton(f, x0, fprime=fprime)
        print(xsol)
Out[x]: 0.6931471805599453
In[x]: np.isclose(f(xsol), 0, atol=1e-10)
Out[x]: True
```

A solução de sistemas de equações não lineares é bem mais complicada do que a solução de uma única equação, e não existe um método que garanta a convergência da solução. No SciPy, uma das possibilidades implementadas para esse tipo de problema é o método `optimize.fsolve`. O sistema de equações deve ser passado como um **array de funções**, e um conjunto de pontos iniciais, também como arrays, deve ser fornecido. Opcionalmente, pode-se passar o *jacobiano* da função com o argumento `fprime`. Note que diferentes pontos iniciais podem levar a diferentes soluções. Como exemplo, vamos resolver o seguinte sistema:

$$\begin{aligned}y - x^3 - 2x^2 + 1 &= 0, \\ y + x^2 - 1 &= 0\end{aligned}$$

Para codificar esse sistema, devemos escrever uma função na forma

$$f[x_1, x_2] = [x_2 - x_1^3 - 2x_1^2 + 1, x_2 + x_1^2 - 1].$$

No código,  $x_1 = x[0]$  e  $x_2 = x[1]$ .

## Example

```
In[x]: def f(x):  
        return [ x[1] - x[0]**3 - 2*x[0]**2 +1,  
                x[1]+x[0]**2 - 1 ]  
  
In[x]: fsolve(f,[1,1])  
Out[x]: array([0.73205081, 0.46410162])  
#jacobiano  
In[x] def f_jacobian(x):  
        return [[-3*x[0]**2 - 4*x[0], 1],  
                [2*x[0], 1]]  
In[x]: fsolve(f,[1,1], fprime=f_jacobian)
```

Além de raízes de funções, o pacote `scipy.optimize` conta com diversos algoritmos de para minimizar uma função de uma ou mais variáveis  $f(x_1, x_2, \dots, x_n)$ . Para maximizar uma função, minimizamos  $-f(x_1, x_2, \dots, x_n)$ .

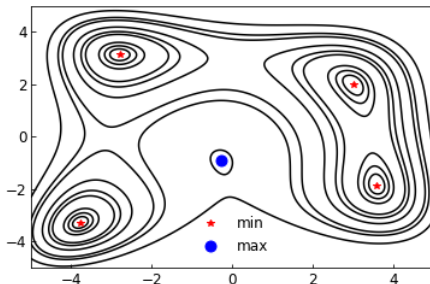
Alguns algoritmos de minimização requerem, além da função, um array com o **jacobiano**:

$$J(f) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

Alguns algoritmos mais sofisticados também requerem uma matriz simétrica com as derivadas parciais de segunda ordem da função (Hessiano).

O algoritmo geral para minimizar funções de várias variáveis é `scipy.optimize.minimize`. Pelo menos dois argumentos devem ser passados: a função a ser minimizada (`fun`), que deve ter como argumentos obrigatórios um **array** `X` e um array de pontos iniciais (`x0`). Como exemplo, vamos minimizar a função:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$



## Minimização

```
In[x]: def f(X):  
        x,y = X  
        return (x**2+y- 11)**2 + (x+y**2-7)**2  
  
In[x]: x0 = [0,0]  
        minimize(f,x0)  
  
Out[x]:  
fun: 1.3782261326630835e-13  
hess_inv: array([[ 0.01578229, -0.0094806 ],  
                 [-0.0094806 ,  0.03494937]])  
jac: array([-3.95019832e-06, -1.19075540e-06])  
message: 'Optimization terminated successfully.'  
nfev: 64  
success: True  
x: array([2.99999994, 1.99999999])
```

## Maximização - Exemplo 1

```
In[x]: mf = lambda X: -f(X)
        minimize(mf, [0.1, -0.2])

Out[x]:
fun: -1.2100579056485772e+35
hess_inv: array([[ 0.254751   , -0.43222419],
                 [-0.43222419,  0.83976276]])
jac: array([0., 0.])
message: 'Optimization terminated successfully.'
nfev: 68
nit: 2
njev: 17
status: 0
success: True
x: array([ 3.45579856e+08, -5.71590777e+08])
```

Algoritmo	Descrição
BFGS	Broyden-Fletcher-Goldfarb-Shanno (default)
Nelder-Mead	Algoritmo de Nelder-Mead
CG	Método do gradiente conjugado
Powell	Método de Powell
dogleg	Necessita Jacobiano e Hessiano
TNC	Algoritmo de Newton Truncado (contornos)
l-bfgs-b	Algoritmo para ser utilizado com contornos
slsqp	Utilizado com contornos e vínculos
cobyla	Método para minimização com vínculos

Veja mais detalhes em

<https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html>



## Maximização - Exemplo 2

```
In[x]: minimize(mf, [0.1, -0.2], method='Nelder-Mead')
Out[x]:
final_simplex: (array([[ -0.27087419,  -0.9230486 ],
                      [ -0.27089208,  -0.92298798],
                      [ -0.27077447,  -0.92304541]]),
                array([ -181.6165215 , -181.61652146,
fun: -181.61652150067573
message: 'Optimization terminated successfully.'
nfev: 77
nit: 39c
success: True
x: array([ -0.27087419,  -0.9230486 ])
```

Podemos também otimizar uma função em uma dada região (contorno) e sujeita a algum vínculo. Os métodos `l-bfgs-b`, `tnc` e `slsqp` podem ser usados com contornos. Os contornos devem ser passados como uma sequência de tuplas, com uma tupla  $(\min, \max)$  para cada variável. Para colocar limite em apenas uma direção, usamos a palavra `None`. Como exemplo, vamos minimizar nossa função  $f(x, y)$  na região  $x \leq 0$  e  $y \leq 0$ .

## Minimização com contorno

```
In[x]: xlimites = (None, 0)
        ylimites = (None, 0)
        contorno = (xlimites, ylimites)
        x0 = (-0.5, -0.5)

In[x]: minimize(f, x0, bounds=contorno, method='slsqp')
Out[x]: ....
        x: array([-3.77933774, -3.28319868])
```

O método `scipy.optimize.curve_fit` é o mais direto e permite passarmos de forma mais transparente os erros da variável `y` e obter as incertezas nos parâmetros ajustados. O método é chamado da seguinte forma:

```
curve_fit(f, xdata, ydata, p0, sigma, absolute_sigma).
```

- `f`, `xdata`, `ydata` são, respectivamente, a função a ser ajustada aos dados (`xdata`, `ydata`);
- `p0` é um valor inicial para os parâmetros;
- `sigma` é um array com as incertezas de `ydata`, de mesmo tamanho de `ydata`;
- `absolute_sigma` é uma variável booleana. Se **True**, os valores absolutos de `sigma` são usados. Essa deve ser a opção usada para obter os valores absolutos nas incertezas dos parâmetros. Se escolhermos a opção **False**, os valores de `sigma` são tratados como valores relativos.

O método `curve_fit` retorna o array `popt`, com o valor dos parâmetros ajustados, e o array 2D `pcov`, a matriz de covariância dos parâmetros. A incerteza nos parâmetros é dada pela raiz quadrada da diagonal de `pcov`:

```
np.sqrt(np.diag(pcov))
```

Para ilustrar o uso deste método, vamos ajustar a função Lorentziana a um conjunto de pontos:

$$f(x) = \frac{A\gamma^2}{\gamma^2 + (x - x_0)^2},$$

onde  $A$ ,  $\gamma$  e  $x_0$  serão os parâmetros ajustados.

Veja solução no código.

# SciPy - Integração Numérica

O pacote `scipy.integrate` contém funções para o cálculo numérico de integrais definidas próprias (limites finitos) e impróprias (limites infinitos). A rotina está implementada em `scipy.integrate.quad`, que é baseada na biblioteca QUADPACK (FORTRAN 77). Os argumentos básicos são o integrando (`func`), e os limites de integração `a` e `b`. O resultado será um flutuante com o valor da integral e outro com uma estimativa do erro absoluto.

## Example

$$I = \int_1^4 x^{-2} dx$$

```
In[x]: from scipy.integrate import quad
```

```
In[x]: f = lambda x: 1/x**(2)
```

```
In[x]: quad(f, a=1, b=4)
```

```
Out[x]:
```

```
(0.75000000000000002, 1.913234548258995e-09)
```

Para integrar funções com singularidades, devemos passar uma lista de pontos onde ocorrem as divergências usando o argumento `points`.

## Example

$$I = \int_{-1}^1 \frac{dx}{\sqrt{|x|}}$$

```
In[x]: f5 = lambda x: 1/np.sqrt(np.abs(x))
```

```
In[x]: quad(f5,-1,1)
```

```
Out[x]:
```

```
RuntimeWarning: divide by zero encountered in  
double_scalars
```

```
(inf, inf)
```

```
In[x]: quad(f5,-1,1,points=[0,])
```

```
Out[x]:
```

```
(3.99999999999999813, 5.684341886080802e-14)
```

Integrais duplas, triplas e múltiplas ( $n > 3$ ) podem ser calculadas, respectivamente, com os métodos `dblquad`, `tplquad` e `nquad`. O método `dblquad` calcula integrais do tipo:

$$I = \int_a^b \int_{g(x)}^{h(x)} f(y, x) dy dx.$$

O integrando deve ser definido como uma função de pelo menos duas variáveis, `func(y, x...)`, tomando, **necessariamente**, `y` como primeiro argumento e `x` como segundo. Os limites de integração devem ser passados como flutuantes, `a` e `b`, para a integral na variável `x`, e como **funções** de `x` para a variável `y`.

Equações diferenciais ordinárias (EDOs) podem ser resolvidas numericamente com `scipy.integrate.odeint` ou `scipy.integrate.solve_ivp`. Esses métodos resolvem equações da forma:

$$\frac{dy}{dt} = \mathbf{F}(\mathbf{y}, t)$$

onde  $\mathbf{y}$  é um vetor de componentes  $y_i(t)$ , e  $\mathbf{F}$  um vetor de componentes  $F(y_i, t)$ .

Para resolver EDOs de ordem  $n > 1$ , devemos transformá-las em um sistema de EDOs de primeira ordem (exemplos nos próximos slides).

O método `scipy.integrate.solve_ivp` toma pelo menos três argumentos: uma função que retorna  $dy/dt$ , os pontos iniciais e finais da variável  $t$ , e um conjunto de condições iniciais  $y_0$ .



## Exemplo 1:

$$\frac{dy}{dt} = -ky$$

- 1 Primeiro, definimos  $dy/dt$  (note a ordem das variáveis!)  

```
def dydt(t, y):  
    return -k*y
```
- 2 Os tempos iniciais e finais devem ser passados como tuplas para o argumento `t_span`: `t_span = (t0, tf)`.
- 3 Os valores iniciais `y0` devem ser passados como sequência (lista, array), **mesmo que só tenha um valor**.
- 4 A solução será um objeto `soln` com os arrays `soln.y`, `soln.t` e `soln.success` (booleano).

## EDOs Acopladas

$$\begin{aligned}\frac{dy_1}{dt} &= f_1(y_1, y_2, \dots, y_n; t), \\ \frac{dy_2}{dt} &= f_2(y_1, y_2, \dots, y_n; t), \\ &\dots \\ \frac{dy_n}{dt} &= f_n(y_1, y_2, \dots, y_n; t).\end{aligned}$$

Nesse caso, a função a ser passada para o método `solve_ivp()` deve retornar uma sequência com as funções  $f_i(y_1, y_2, \dots, y_n; t)$ .

## EDOs Acopladas - Implementação

```
# y = [y1, y2, y3, ...]
# (sequencia de variáveis independentes)
def deriv(t, y):
    dy1dt = f1(y, t)
    dy2dt = f2(y, t)
    #....
    return dy1dt, dy2dt, ..., dyndt
solve_ivp(deriv, (t0, tf), y0 )
```

Note que agora,  $y_0$  será um sequência de  $n$  elementos.

## Exemplo 2: Evolução de uma Epidemia (Modelo SIR)

$$\frac{dS}{dt} = -\beta SI,$$

$$\frac{dI}{dt} = \beta SI - \gamma I,$$

$$\frac{dR}{dt} = \gamma I,$$

onde as constantes  $\beta$  e  $\gamma$  são, respectivamente, a taxa de transmissão e a taxa de recuperação. Queremos resolver esse sistema para  $S(t)$ ,  $I(t)$  e  $R(t)$ . Para uma população de tamanho fixo  $N$ , temos, em qualquer instante,  $N = S(t) + I(t) + R(t)$ .

Veja solução no código.

## Exemplo 3: EDO de segunda ordem

Para resolver uma EDO de ordem  $n > 1$ , primeiro devemos reduzi-la a um sistema de EDOs de primeira ordem:

$$\frac{d^2x}{dt^2} = -\omega^2 x$$

$$\frac{dx_1}{dt} = x_2,$$

$$\frac{dx_2}{dt} = -\omega^2 x_1,$$

onde  $x_1 = x$  e  $x_2 = dx/dt$ .

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

<https://www.youtube.com/user/HeavyState>

## **Material dos minicursos disponíveis em:**

<https://github.com/aanepomuceno>