Python Científico

André Nepomuceno

Universidade Federal Fluminense

4 de novembro de 2021

O que você vai aprender ?

- Matplotlib Avançado
- Introdução ao Pandas
- Álgebra Linear com NumPy
- Interpolação
- Otimização
- Solução de Equações não lineares
- Solução Numérica de EDO
- Computação Simbólica

Matplotlib - Interface Avançada

O Matplotlib tem uma interface básica, semelhante ao MATLAB, que permite fazer vários gráficos simples (veja a aula 8 do minicurso Python Básico). 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
`num`	String identificador da figura
`figsize`	Tupla com as dimensões da figura (largura, comprimento), em polegadas
`dpi`	Resolução da figura (pts por polegada)
`facecolor`	Cor de fundo
`edge color`	Cor da borda

Example

```
fig = plt.figure("Figura1", figsize=(4.5,3),
                 facecolor='r')
```

Matplotlib - Fontes

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

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

Para usar a mesmas opções de fonte em todos os textos (título, labels), podemos construir um dicionário e usar o comando `rc`.

Matplotlib - Fontes

Parâmetros da Fonte

```
from matplotlib import rc
font_properties = {'name': 'Courier',
                   'weight': 'bold ', 'size': 13}

rc('font ', **font_properties )
```

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')
```

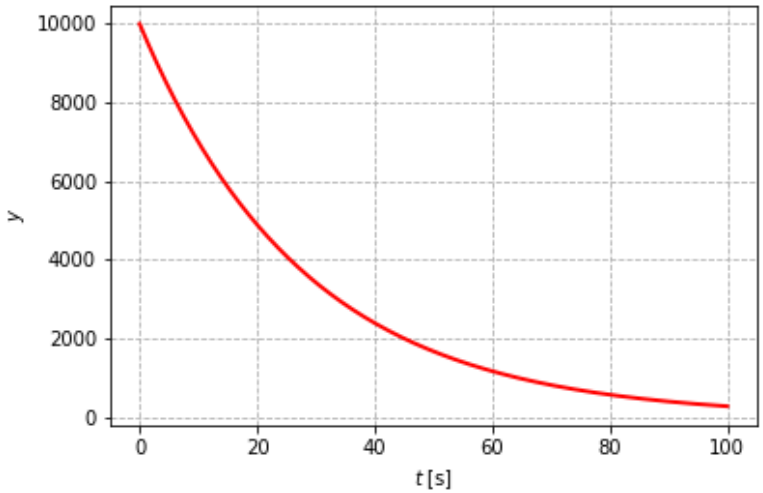
Matplotlib - Marcadores

Existem diversas opções para modificar marcadores no Matplotlib. Por exemplo, para definir marcadores para apenas algums valores do eixo, fazemos:

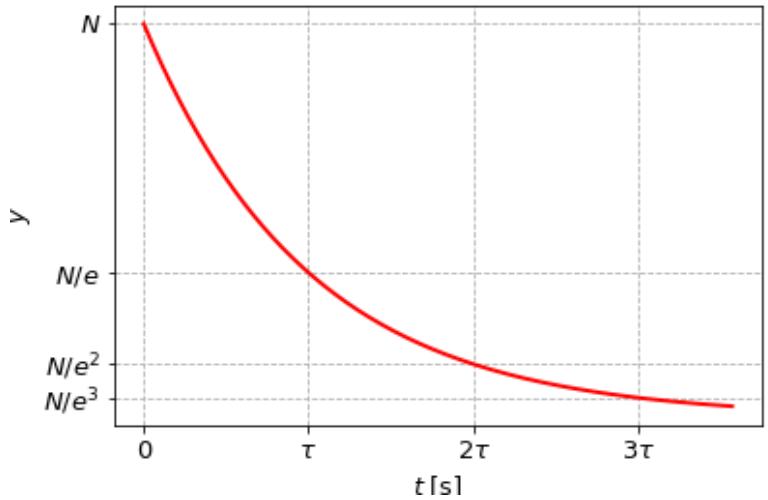
`ax.set_yticks([1, 2, 3.5, 4.])`.

É possível substituir os marcadores numéricos por *strings* usando os comandos `ax.set_xticklabels` e `ax.set_yticklabels`.

Veja o código `pcientifico_matplotlib.ipynb`



Marcadores padrão



Marcadores modificados

Matplotlib - Marcadores

Esconder Marcadores

`ax.set_yticks([])`: apaga marcadores e labels.

`ax.set_yticklabels([])`: apaga os labels, mantém os marcadores.

Adicionar submarcadores

`ax.minorticks_on()`

Adicionar marcadores aos eixos paralelos

`ax.xaxis.set_ticks_position('bottom')`

As opções são `'bottom'`, `'top'`,`'both'`, ou `'none'`

`ax.yaxis.set_ticks_position('both')`

As opções são `'left'`, `'right'`,`'both'`, ou `'none'`

Matplotlib - Marcadores

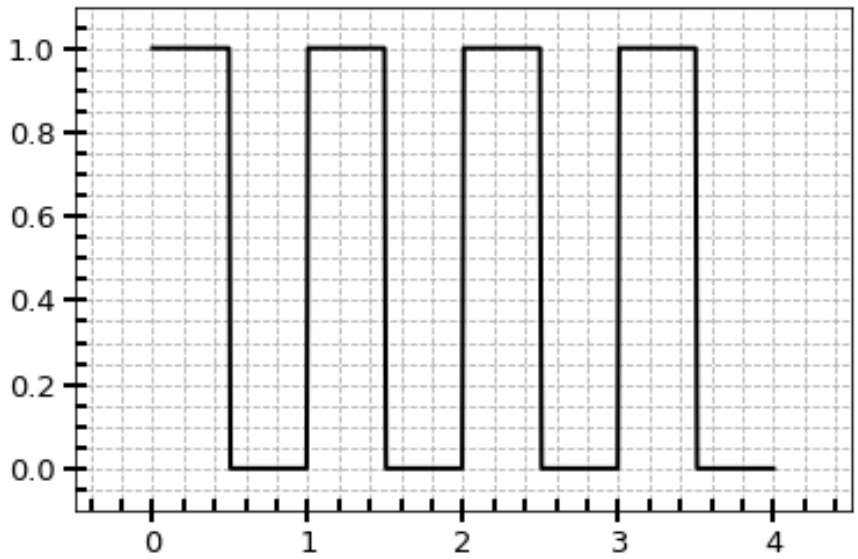
Opções mais avançadas de marcadores estão disponíveis na função `ax.tick_params()`, com os argumentos da tabela abaixo.

Argumento	Descrição
`axis`	Eixo que será alterado: 'x','y','both'
`which`	Marcador:'major','minor', 'both'
`direction`	'in', 'out','inout
`length`	Comprimento do marcador
`width`	Largura do marcador
`labelsize`	Tamanho do label
`color`	Cor do marcador
`labelcolor`	Cor do label
`labeltop/labelright`	True ou False

Matplotlib - Marcadores

Example

```
ax.minorticks_on()
ax.tick_params(which='major', length=10, width=2,
               labelsize=13, direction='inout')
ax.tick_params(which='minor', length=5, width=2,
               direction='in')
ax.grid(which='both',ls='--')
```



Matplotlib - Subplots

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 x 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>)`.

Matplotlib - Subplots

Para ajustar a distância entre os gráficos, usamos `fig.subplots_adjust(hspace=<float>, wspace=<float>)`.

Example

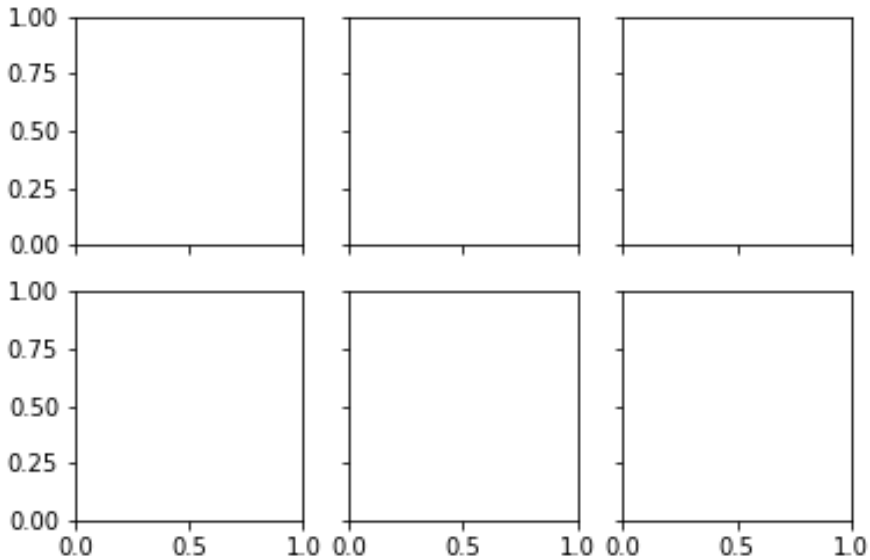
```
fig, axes = plt.subplots(2,1)
fig.subplots_adjust(hspace=0.05)
x = np.linspace(0,2*np.pi,100)
axes[0].plot(x,np.sin(x))
axes[0].set_xticks([])
axes[1].plot(x,np.cos(x))
```

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 - Subplots

Em algumas situações, é conveniente fazer um gráfico menor dentro de um gráfico maior, para realçar um resultado. Esse tipo de figura pode ser feito com o método `plt.axes()`

Métodos plt.axes()

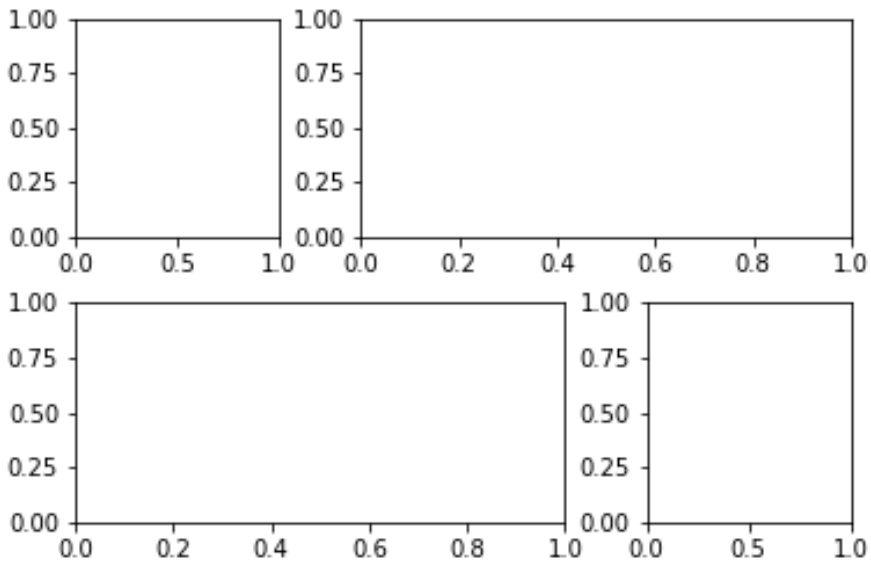
```
fig = plt.figure()
ax1 = plt.axes()
ax2 = plt.axes([0.65,0.65,0.2,0.2])
```

Os primeiros dois valores indicam que a posição do segundo plot começa em 65% da largura e 65% do comprimento de ax1, e seu tamanho é de 20% de ax1.

Matplotlib - Subplots

Métodos plt.GridSpec()

```
grid = plt.GridSpec(2, 3, hspace=0.3, wspace=0.4)
fig = plt.figure()
ax1 = fig.add_subplot(grid[0,0])
ax2 = fig.add_subplot(grid[0,1:])
ax3 = fig.add_subplot(grid[1,:2])
ax4 = fig.add_subplot(grid[1,2])
```



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

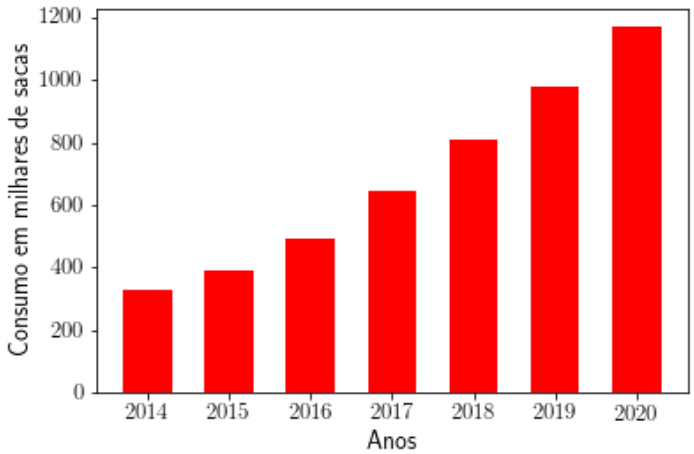
```
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000*np.random.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)
plt.colorbar()
```

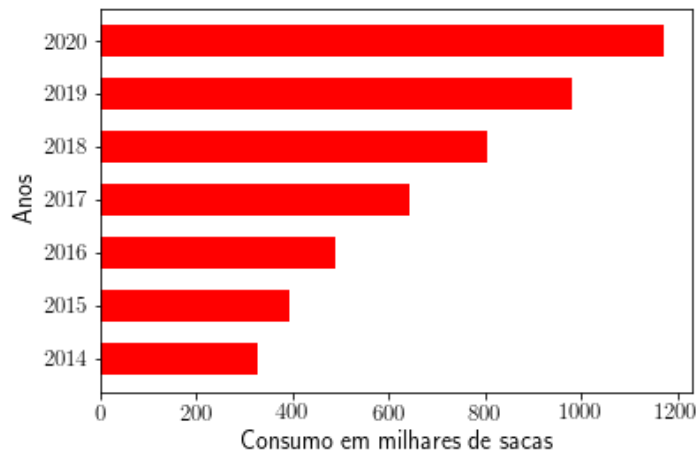
Matplotlib - Gráfico de Barras

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`

Example

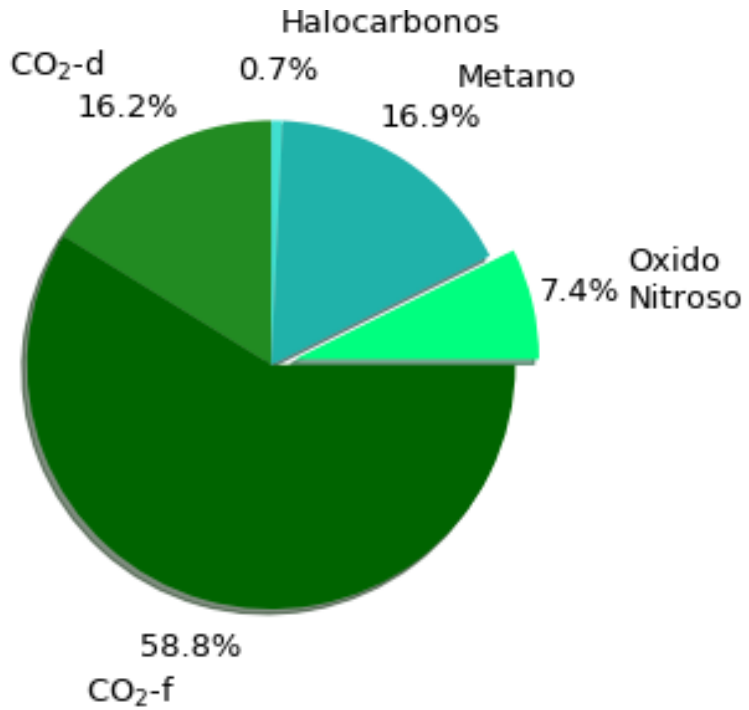
```
anos = np.arange(2014,2021)
consumo = np.array([327,392,490,643,806,981,1171])
plt.bar(anos, consumo, width=0.6,color='r')
plt.barh(anos, consumo, height=0.6,color='r')
```





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`
Veja exemplo em `pcientifico_matplotlib01.ipynb`

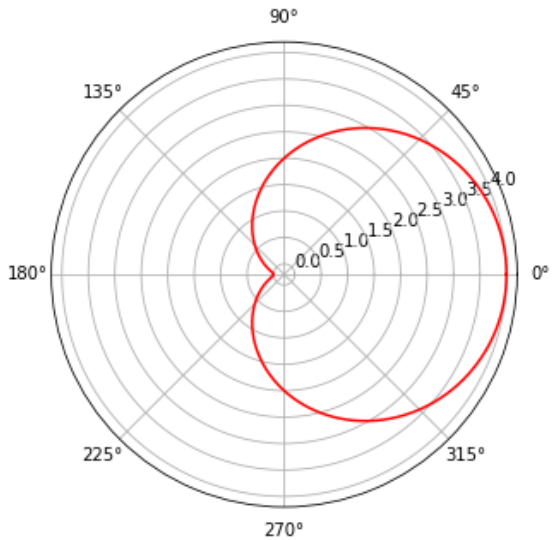


Matplotlib - Gráficos Polares

Gráficos polares (raio r em função de um ângulo θ) podem ser criados especificando o argumento `projection='polar'` na função `fig.add_subplot()`, ou usando a função `pyplot.polar()`. Como exemplo, vamos fazer o gráfico de $r = 2a(1 + cos\theta)$ (cardioide).

Example

```
theta = np.linspace (0, 2*np.pi, 1000)
r = 2 * (1. + np.cos(theta ))
plt.polar(theta,r,'r')
```



Matplotlib - Histogramas

Histograma é uma representação gráfica de uma distribuição discreta de probabilidade. Para plotar um histograma, basta passar um array para a função `plt.hist()`. O exemplo abaixo ilustra as várias opções disponíveis.

Example

```
data = np.random.randn(1000)
plt.hist(data, bins=30, density=True, alpha=0.5,
         histtype='bar', color='steelblue')
```

Para obter a contagem em cada bin, podemos usar o método `np.histogram()`:

Example

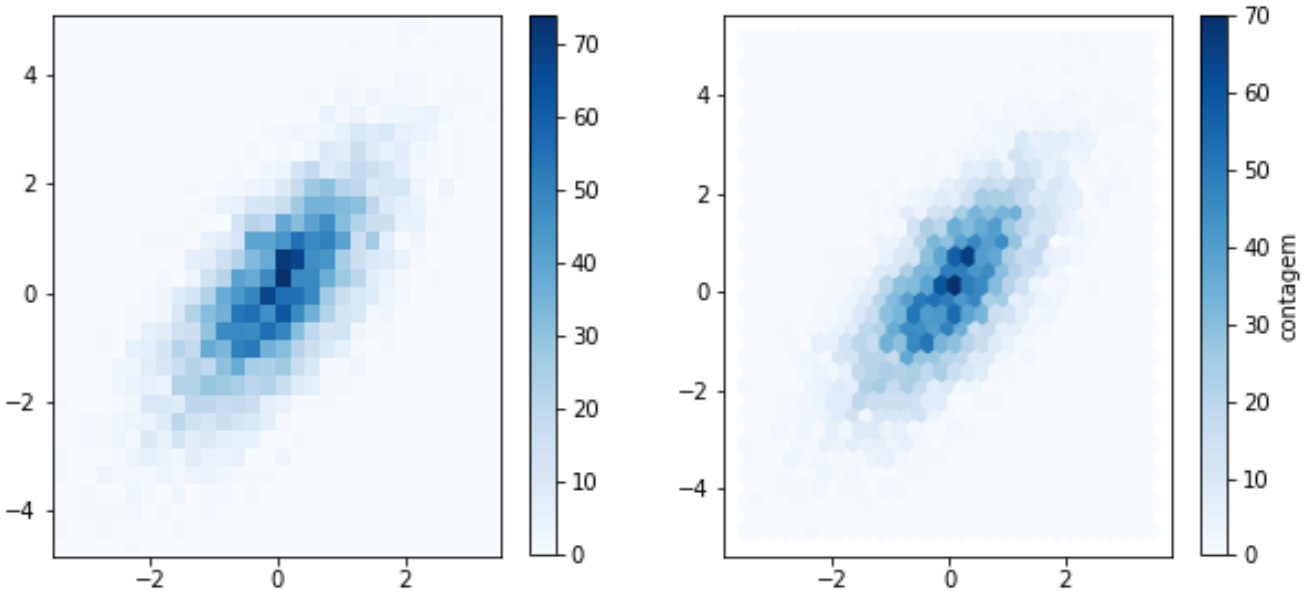
contagem, x_bin = np.histogram(data,bins=10)

Matplotlib - Histogramas 2D

Para criar histogramas em duas dimensões, devemos passamos dois arrays para a função `plt.hist2d()` ou `plt.hexbin()`.

Example

```
plt.hist2d(x,y,bins=30,cmap='Blues')
plt.hexbin(x, y, gridsize=30, cmap='Blues')
plt.colorbar(label='contagem')
```



Matplotlib - Anotações

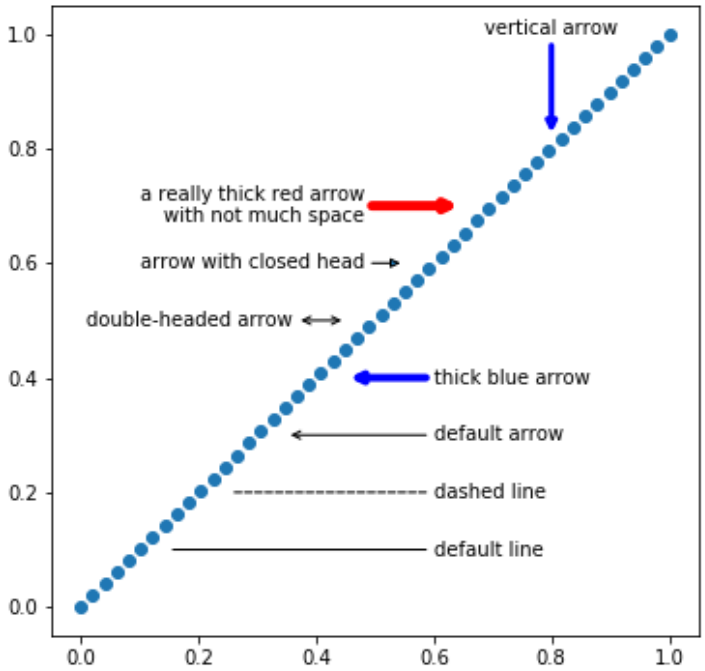
O método `ax.text(x,y,s)` permite incluir um *string* `s` na posição (x,y) dos **dados**. Para fixar o texto numa determinada posição independente do range dos dados, devemos usar o argumento `transform=ax.transAxes`. Nesse caso, a coordenada (0,0) representa o canto inferior esquerdo do eixo, e (1,1) o canto superior direito.

Para utilizar setas junto com texto, existe o método `ax.annotate()`. Os principais argumentos da função são:

- `s`: texto inserido (string)
- `xy`: tupla com as coordenadas para onde a seta *aponta*
- `xytext`: coordenadas do texto
- `xycoords`: string opcional que especifica o **tipo** de coordenada do argumento `xy` ('data' ou 'axes fraction')

Matplotlib - Anotações

- `textcoords`: mesmo que `xycoords`, mas referente ao argumento `xytext`. Para esse argumento existe um valor adicional, 'offset points', que dá uma separação entre `xytext` e `xy`
- `arrowprops`: **dicionário** com propriedades e estilo da seta.
- `ha/va`: alinhamento do texto em relação a posição `xytext` ('center','right','top','bottom', 'baseline').



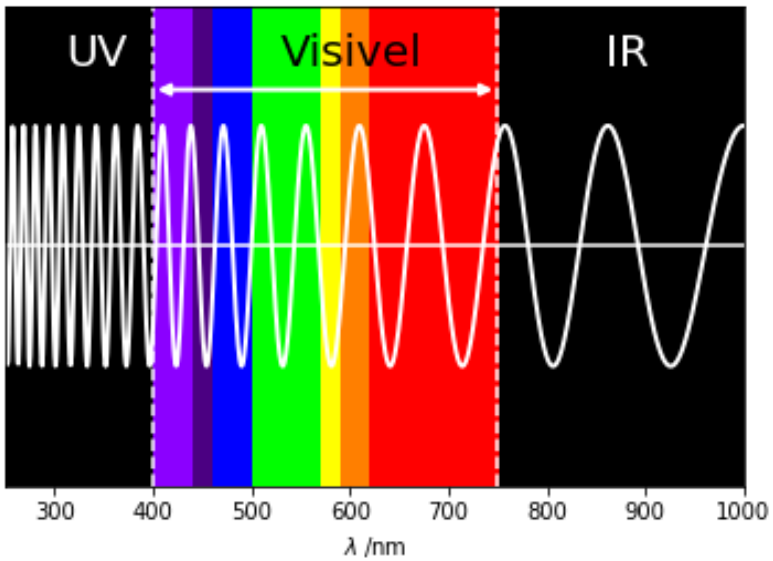
Matplotlib - Linhas

Para desenhar linhas horizontais e verticais, existem os métodos `ax.hlines()` `ax.vlines()`, respectivamente. Os argumentos do método `ax.hlines()` são `y`, `xmin` e `xmax`, que podem ser uma sequência ou um escalar. Para `ax.vlines()`, os argumentos são `x`, `ymin` e `ymax`.

Para desenhar linhas independentes do range dos eixos, podemos usar a `ax.axhline()` e `ax.axvline()`, mas nesse caso, os argumentos são necessariamente **escalares**, e as posições `xmin` e `xmax` (`ymin` e `ymax`) são frações da coordenadas, ou seja `xmin = 0` é o extremo esquerdo e `xmax = 1` é o extremo direito (no caso da linha vertical, 0 é a base e 1 é o topo).

Matplotlib - Retângulos

Retângulos horizontais ou verticais podem ser criados com os métodos `ax.axhspan()` e `ax.axvspan()`. Nesse caso, são passados quatro argumentos, sendo os dois primeiros as coordenadas dos **dados** e os dois últimos **frações** dos eixos. Se os dois últimos valores não são passados, os valores usados são `0` e `1`. O exemplo abaixo ilustra o uso de linhas, retângulos e texto.
Veja o código em `pcientifico_matplotlib01.ipynb`.



Matplotlib - Gráficos de Contorno

Uma função $h(x, y)$ pode ser interpretada como uma superfície em que cada ponto (x, y) tem um altura específica. Um gráfico de contorno é um gráfico em duas dimensões da função $h(x, y)$ em que linhas ou cores são usadas para representar a altura. Para fazer um gráfico desse tipo, precisamos especificar uma altura Z e um conjunto de pontos que formem uma grade no plano xy. Essa grade de pontos pode ser criada usando o método `numpy.meshgrid()`.

Example

```
x = np.linspace(-3,3,21)
y = np.linspace(0,10,11)
X,Y = np.meshgrid(x,y)
```

Os novos arrays X e Y tem forma 11×21 e $11 \times 21 = 231$ entradas.

Matplotlib - Gráficos de Contorno

Método `contour()`

Os argumentos são os 2D arrays X, Y e Z. Os níveis de contorno pode ser especificado por um número total `N` ou uma sequência que especifique os valores de Z que devem ser plotados. Para plotar contorno com um única cor, usamos o argumento `colors`.

Example

```
x = np.linspace(-3,3,21)
y = np.linspace(0,10,11)
X,Y = np.meshgrid(x,y)
Z =  np.sin(X)**10 + np.cos(10 + Y*X)*np.cos(X)
fig, ax = plt.subplots()
ax.contour(X,Y,Z,5,colors='k')
```

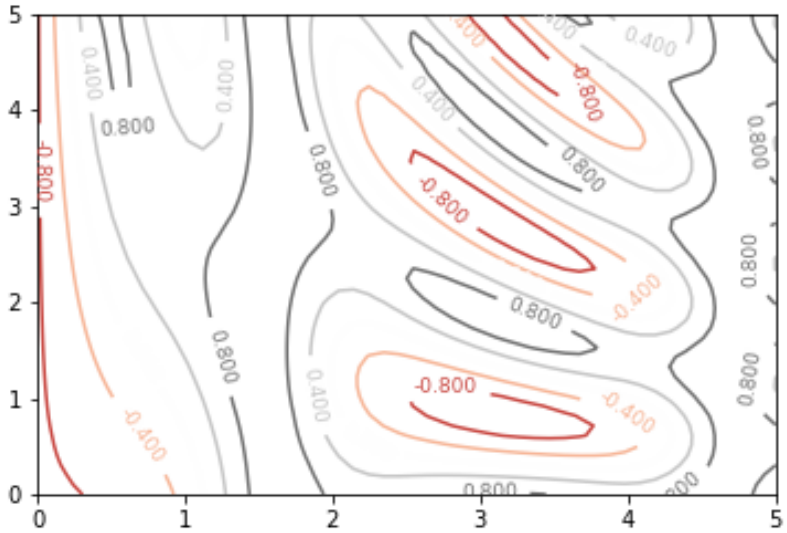
Cada nível de contorno também pode ser especificado por um mapa de cor com o argumento `cmap`. Exemplo: `cmap='RdGy'`.

Matplotlib - Gráficos de Contorno

Para adicionar os valores dos contornos (labels), usamos a função `ax.clabel()`. Nessa caso, o plot deve ser guardado numa variável e passado para a função `clabel()`.

Example

```
fig, ax = plt.subplots()
cp = ax.contour(X,Y,Z,5, cmap='RdGy')
ax.clabel(cp,fontsize=9)
```



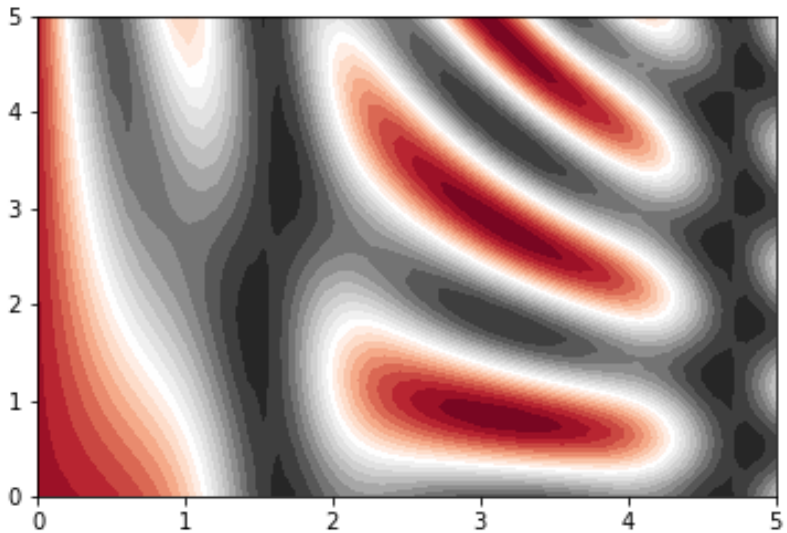
Matplotlib - Gráficos de Contorno

Método `contourf()`

O método `contourf()` tem os memos argumentos de `contour()`, mas produz contornos preenchidos.

Example

```
fig, ax = plt.subplots()
ax.contourf(X,Y,Z,20, cmap='RdGy')
```



Matplotlib - Mapa de Calor

Um mapa de calor é um imagem em que a cor de cada pixel é determinado por um valor correspondente numa sequência de dados. A função para produzir mapas de calor é `imshow()`. Alguns pontos importantes:

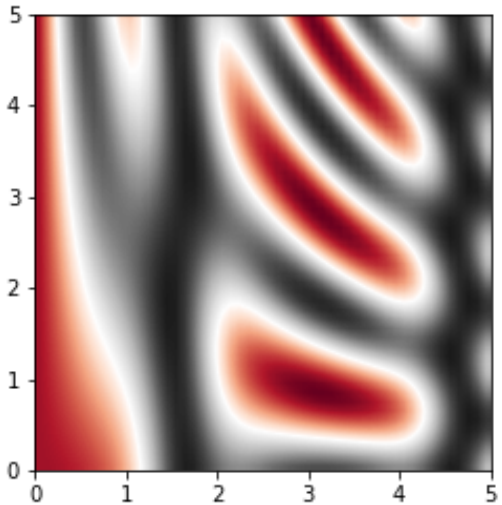
- `imshow()` não aceita um grid (x,y); o que é plotado nos eixos são os **índices** da matriz, sendo (0,0) o canto *superior* esquerdo.
- Para que a imagem seja mostrada nas coordenadas dos dados, devemos utilizar as opções `extent = [xmin, xmax, ymin, ymax]` e `origin='lower'`.
- pode-se aplicar interpolação com o argumento `interpolation`.

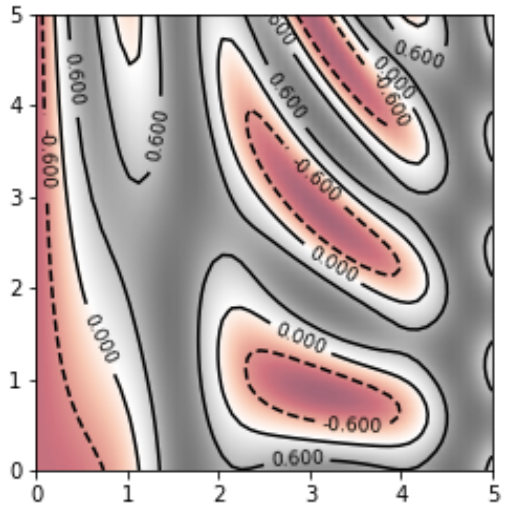
Matplotlib - Mapa de Calor

Podemos também combinar `imshow()` com `contour()`

Example

```
cp = ax.contour(X,Y,Z,3,colors='k')
ax.clabel(cp,fontsize=9)
ax.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
          interpolation='bilinear',alpha=0.6,
          cmap='RdGy')
```



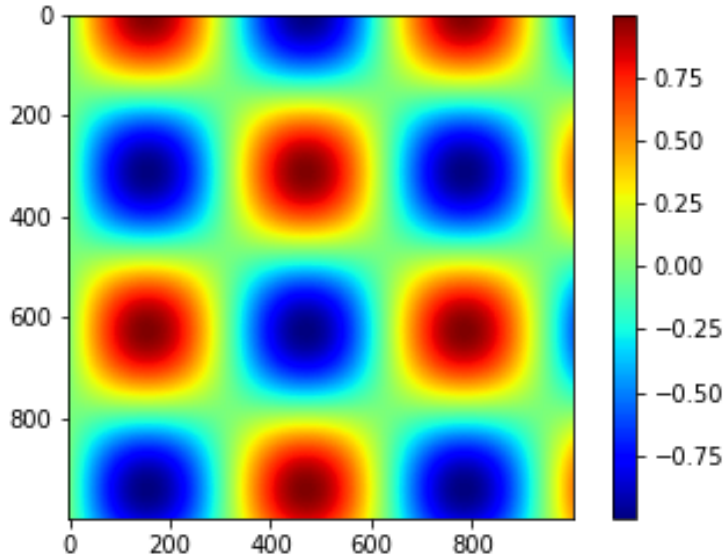


Matplotlib - *colorbar*

No Matplotlib, um *colorbar* é um eixo separado que indica como as cores do plot se relacionam aos valores do função $h(x, y)$. Para adicioná-lo a figura, chamamos o método `fig.colorbar(mappable = <obj>)`, onde `<obj>` é o objeto que guarda das informações do plot.

Example

```
x = np.linspace(0,10,1000)
y = np.linspace(0,10,1000)
X,Y = np.meshgrid(x,y)
I = np.sin(X) * np.cos(Y)
im= ax.imshow(I,cmap='jet')
fig.colorbar(im)
```



Matplotlib - *colormap*

A relação entre os valores do array e um determinado range de cores é definido pelo *colormap*. A escolha do *colormap* é passada pelo argumento `cmap` (veja exemplo anterior).
Veja lista de opções em `https://matplotlib.org/stable/tutorials/colors/colormaps.html`.
Alguns exemplos:

A escolha do *colormap* para a visualização dos dados depende de vários fatores, e **não é trivial**.

Matplotlib - Escolha do *colormap*

De forma geral, existem pelo menos três categorias de *colormap*:

1. **Sequencial**: sequência contínua de cor ou cores, usado para dados que variam de valores baixos para valores mais altos (ex., `viridis`, `Blues`).
2. **Divergente**: usualmente tem duas cores distintas, usado para mostrar desvios em relação a um valor médio (ex.,`RdBu` ou `PuOr`).
2. **Qualitativo**: mistura de cores com rápida variação, usado principalmente para dados discretos ou categorias (ex.,`rainbow` ou `jet`).

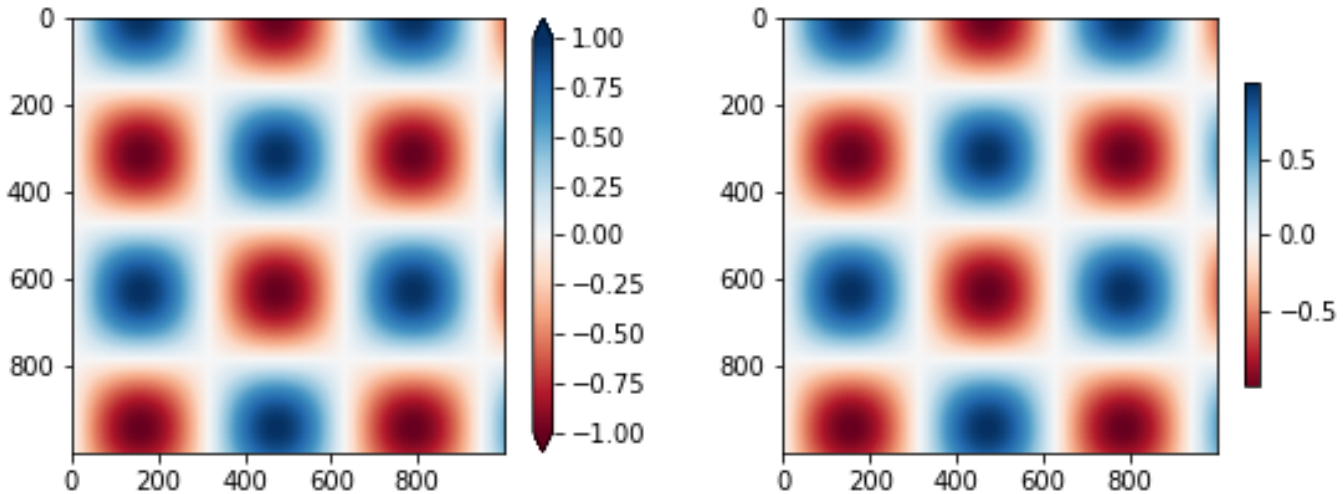
Para mais detalhes, veja:
Why We Use Bad Color Maps and What You Can Do About It
`https://www.osti.gov/servlets/purl/1338147`
Ten Simple Rules for Better Figures
`https://doi.org/10.1371/journal.pcbi.1003833`

Matplotlib - Ajustando o *colorbar*

Existem várias opções disponíveis para o eixo *colorbar* como ilustrado nos exemplos abaixo (mostrado parte do código apenas).

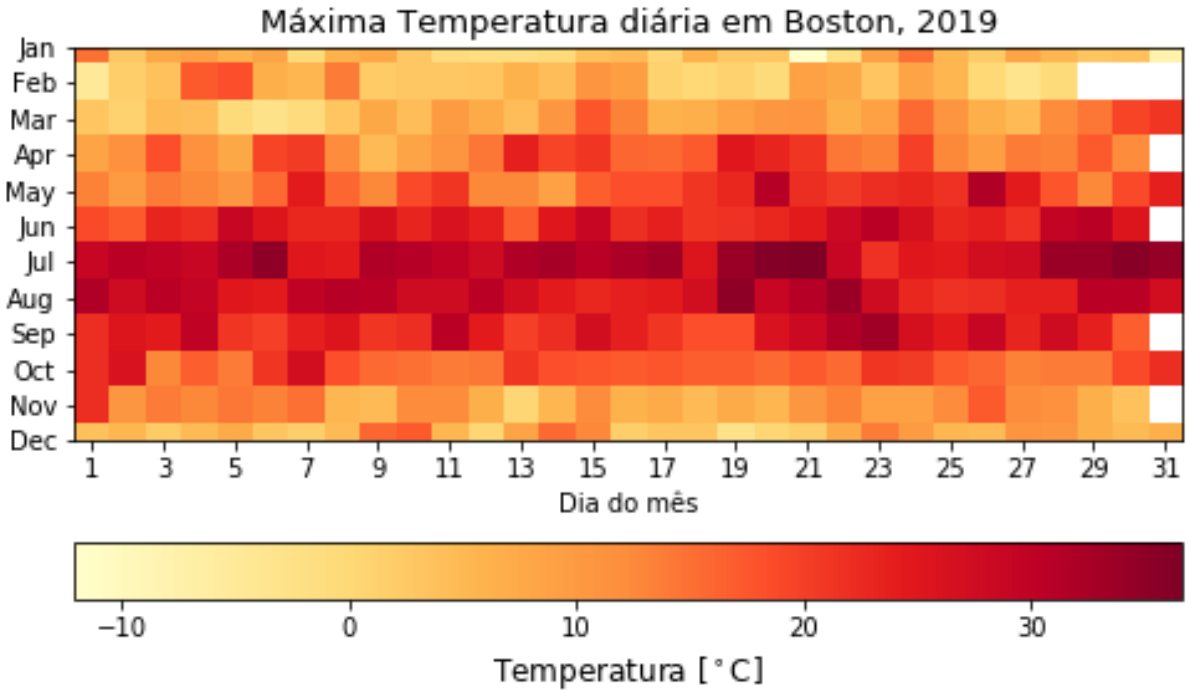
Example

```
fig, (ax1,ax2) = plt.subplots(figsize=(8,3),ncols=2)
......
fig.colorbar(im1,ax=ax1,orientation='vertical',
             extend='both')
fig.colorbar(im2,ax=ax2,shrink=0.7)
```



Matplotlib - Exemplo com `imshow()`

Exercício. O arquivo `boston_temp2019.dat` contém a máxima temperatura diária de Boston em 2019. Escreva um programa em Python para plotar um mapa de calor dos dados, incluindo uma legenda para o *colorbar*.
Veja a solução em `pcientifico_matplotlib02.ipynb`.



Matplotlib - Gráficos 3D

Para criar um plot tridimensional, devemos importar o função `Axes3D` a adicionar e argumento `projection =‘3d‘` ao subplot:

import Axes3D

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(projection ='3d')
```

Matplotlib - Gráficos 3D

Linhas e pontos podem ser plotados da mesma forma que 2D plots.

Linhas e Pontos

```
fig = plt.figure()
ax = fig.add_subplot(projection ='3d')
zline = np.linspace(0, 15, 500)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot(xline, yline, zline, 'gray')
zdata = 15*np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter(xdata, ydata, zdata, c=zdata, cmap='Blues')
```

Matplotlib - Gráficos 3D

Para plotar superfícies, devemos passar três arrays 2D. Os argumentos `rstride` e `cstride` indicam o passo em que linhas/colunas devem ser tomadas.

Wireframe e Superfícies

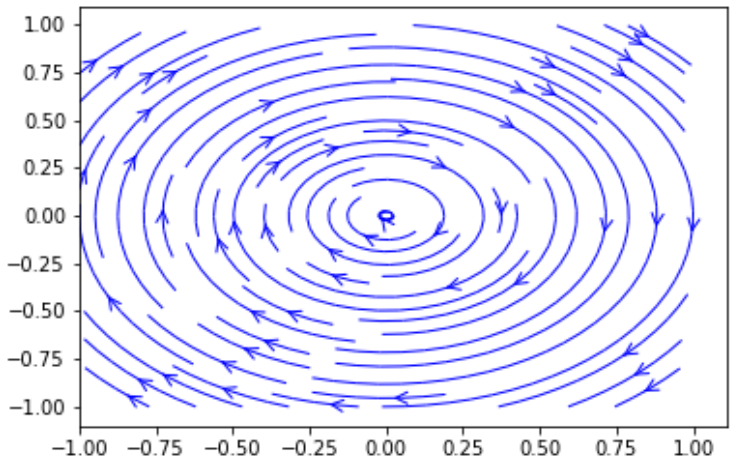
```
fig, ax = plt.subplots(figsize=(8,7),nrows=2,
          ncols=2, subplot_kw={'projection': '3d'})
ax[0,0].plot_wireframe(X, Y, Z, rstride=40,
                       cstride=40)
ax[0,1].plot_surface(X, Y, Z, rstride=40,
                     cstride=40, color='r')
ax[1,0].plot_surface(X, Y, Z, rstride=12,
                     cstride=12, color='m')
ax[1,1].plot_surface(X, Y, Z, rstride=20,
                     cstride=20, cmap='hot')
```

Matplotlib - Streamlines

Visualização de linhas de campo.

Example

```
xx = np.linspace(-1,1,11)
X,Y = np.meshgrid(xx,xx)
Vx,Vy = Y,-X
plt.streamplot(X,Y,Vx,Vy,color='b',
               linewidth=1, density=1,
               arrowstyle='->', arrowsize=1.5)
```



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')
```

Já para salvar imagens com boa resolução (como mapas de calor), o melhor é especificar 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)
```

Pandas - Definindo os Termos

Pandas é uma biblioteca do Python para manipulação e análise de dados. O Pandas tem duas estrutura de dados chamadas `Series` e `DataFrame`, que representam, respectivamente, uma sequência de e uma tabela de dados. Diferentemente dos arrays do NumPy, esses objetos podem conter **diferentes** tipos de dados.

Instalação

```
conda install pandas
```

Devemos importar o módulo no início do código

```
import pandas as pd
```

Pandas - Series

Criando Series

```
In[x] tamanho_rios = pd.Series([6300,6650,6275,6400])
In[x] tamanho_rios
Out[x]:
0    6300
1    6650
2    6275
3    6400
dtype: int64
```

Atribuir nome e tipo

```
tamanho_rios = pd.Series([6300,6650,6275,6400],
                         name='Comprimento',
                         dtype = float )
```

Pandas - Series

Indexação Explícita

```
In[x]: tamanho_rios =
       pd.Series(data =[6300 ,6650,6275, 6400],
                 index =['Yangtze', 'Nilo',
                         'Mississippi', 'Amazonas'],
                 name='Comprimento')
In[x]: tamanho_rios
Out[x]: Yangtze          6300
        Nilo             6650
        Mississippi      6275
        Amazonas         6400
        Name: Comprimento, dtype: int64
In[x]:  tamanho_rios.index
Out[x]: Index(['Yangtze', 'Nilo', 'Mississippi',
               'Amazonas'], dtype='object')
```

Pandas - Series

Indices e *Slicing*

```
In[x]: tamanho_rios['Nilo']
Out[x]: 6650
In[x]: tamanho_rios[['Amazonas', 'Nilo']]
Out[x]: Amazonas    6400
        Nilo        6650
        Name: Comprimento, dtype: int64
In[x]: tamanho_rios['Nilo':'Amazonas']
Out[x]: Nilo            6650
        Mississippi     6275
        Amazonas        6400
        Name: Comprimento, dtype: int64
```

Note que no *slicing* explícito o último elemento é **incluído**.

Pandas - Series

Combinando Series

```
massas = pd.Series({'Ganymede': 1.482e23,
                    'Callisto': 1.076e23,
                    'Io': 8.932e22,
                    'Europa': 4.800e22,
                    'Moon': 7.342e22,
                    'Earth': 5.972e24},
                    name='massa')
raios = pd.Series({'Ganymede': 2.634e6,
                   'Io': 1.822e6,
                   'Moon': 1.737e6 ,
                   'Earth': 6.371e6}, name='raio')
```

Combinando Series

```
G = 6.674e-11
gravidade = G*massas/raios**2
gravidade.name = 'Gravidade em m/s2'
gravidade
Out[x]: Callisto           NaN
        Earth         9.819532
        Europa             NaN
        Ganymede      1.425617
        Io            1.795718
        Moon          1.624056
        Name: Gravidade em m/s2, dtype: float64
```

Note que nos índices que não tem correspondência, o resultado da combinação é 'Not a Number' (NaN). Exitem pelo menos dois métodos para tratar essa situação.

Pandas - Series

isnull() e dropna()

```
In[x]: gravidade.isnull()
Out[x]: Callisto     True
        Earth        False
        Europa       True
        Ganymede     False
        Io           False
        Moon         False
        Name: Gravidade em m/s2, dtype: bool
In[x]: gravidade.dropna()
Out[x]: Earth        9.819532
        Ganymede     1.425617
        Io           1.795718
        Moon         1.624056
        Name: Gravidade em m/s2, dtype: float64
```

Pandas - DataFrame

Um `DataFrame` é uma tabela de dados que pode ser pensanda como um conjunto ordenado de colunas de `Series` de mesmo índice. Um `DataFrame` pode ser criados a partir de dicionários ou `Series`.

Criando um `DataFrame`

```
data = {'mass': [1.482e23,1.076e23,8.932e22,
                 4.800e22,7.342e22],
        'radius': [2.634e6, None, 1.822e6,
                   None, 1.737e6],
        'planet': ['Jupiter', 'Jupiter', 'Jupiter',
                   'Jupiter', 'Earth']}
index = ['Ganymede', 'Callisto', 'Io', 'Europa',
         'Moon']
df = pd.DataFrame(data, index=index )
```

Acessando Colunas e Células

```
In[x]: df['mass']    #ou df.mass
Out[x]: Ganymede     1.482000e+23
        Callisto     1.076000e+23
        Io           8.932000e+22
        Europa       4.800000e+22
        Moon         7.342000e+22
        Name: mass, dtype: float64
In[x]: df['mass']['Io']  #ou df['mass'][2]
Out[x]: 8.932e+22
```

Warning I

Evite atribuições do tipo `df['mass']['Io'] = <valor>`.

Pandas - DataFrame

Método `loc`

```
In[x]: df.loc['Europa']
Out[x]: mass      4.8e+22
        radius    NaN
        planet    Jupiter
        Name: Europa, dtype: object
In[x]: df.loc['Europa',['mass','planet']]
Out[x]: mass     4.8e+22
        planet   Jupiter
        Name: Europa, dtype: object
In[x]: df.loc['Ganymede':'Io', ['mass', 'radius']]
Out[x]:
                mass            radius
Ganymede   1.482000e+23   2634000.0
Callisto   1.076000e+23   NaN
Io         8.932000e+22   1822000.0
```

Atribuição e Filtro com `loc`

```
In[x]: df.loc['Europa', 'radius'] = 1.561e6
In[x]: df.loc['Europa']
Out[x]: mass          4.8e+22
        radius     1.561e+06
        planet       Jupiter
        Name: Europa, dtype: object
In[x]: df.loc[df.radius < 2.e6]
Out[x]:
          mass             radius     planet
Io      8.932000e+22  1822000.0  Jupiter
Europa  4.800000e+22  1561000.0  Jupiter
Moon    7.342000e+22  1737000.0  Earth
```

Note que no método `loc` os elementos são acessados na forma df.loc[**linha**, **coluna**].

Pandas - DataFrame

Método iloc

```
In[x]: df.iloc[1]
Out[x]:
        mass       1.076e+23
        radius           NaN
        planet       Jupiter
        Name: Callisto, dtype: object
In[x]: df.iloc[:,[1,2]].dropna()
Out[x]:
              radius       planet
Ganymede    2634000.0   Jupiter
Io          1822000.0   Jupiter
Europa      1561000.0   Jupiter
Moon        1737000.0    Earth
In[x]: df.iloc[-1,1] #ultima linha, segunda coluna
Out[x]: 1737000.0
```

`DataFrame` para Arrays

```
In[x]: df.iloc[:,[0,1]].values
Out[x]: array([[1.482e+23, 2.634e+06],
               [1.076e+23,       nan],
               [8.932e+22, 1.822e+06],
               [4.800e+22, 1.561e+06],
               [7.342e+22, 1.737e+06]])
```

Warning II

Lembre-se que `loc` sempre se refere ao **índice explícito**, enquanto `iloc` se refere a **posição** do índice (começando em 0).

Pandas - Lendo Arquivos de Texto

O grande poder do Pandas é revelado no tratamento de dados provenientes de grandes arquivos. Para ler arquivos de texto com Pandas, o comando básico é `pd.read_csv()`. Além do nome do arquivo, a função tem 48 argumentos opcionais! Três desses argumentos são :

- `delimiter = <string>`, que indica o tipo de separador;
- `header = <inteiro>`, que indica qual linha deve ser usada para os nomes das colunas;
- `index_col = <inteiro>`, que indica qual coluna deve ser usada como índices.

Veja mais detalhes em `https://pandas.pydata.org/`.

Example

```
data = pd.read_csv('india-data.csv',index_col=0)
data.head()
```

Pandas - Web Scraping

A função `pd.read_html()` pode ser usada para obter dados da Web disponíveis em tabelas HTML. Será retornado uma **lista** de `DataFrames`. Pode ser necessário instalar alguns pacotes:

```
conda install lxml
conda install html5lib
conda install bs4
```

Example

```
url = 'endereço_da_pagina'
data_web = pd.read_html(url,header=1,index_col=0)
data_web[0].head()
```

Pandas - Indexação Hierárquica

Um `DataFrame` é um array de dados em 2D. Para representar dados em mais dimensões, podemos adotar **níveis** dentro de um dado índice. Esse tipo de indexação é chamada indexação hierárquica ou *multi-indexing*. Exemplo: considere um conjunto de dados da temperatura média e chuva mensais em diferentes cidades. Esses dados podem ser considerados tridimensionais, sendo a cidade a primeira dimensão, o mês a segunda e os dados (temperatura ou chuva) a terceira. No Pandas, podemos representar as duas primeiras dimensões como um índice hierárquico de dois nívies.

Example

```
cidades = ('Paris', 'Berlin', 'Viena', 'Londres',
           'Madri')
meses = ('Jan', 'Abr', 'Jul', 'Out')
index = pd.MultiIndex.from_product((cidades, meses))
index.names = ['Cidade', 'Mes']
index
```

Pandas - Indexação Hierárquica

Como são quatro meses, cinco cidades e dois dados (temp. e chuva), temos no total 40 entradas. Vamos criar um `DataFrame` passando uma array de forma (20,2).

Example

```
#array arr1 contém duas colunas de dados
In[x]: df3 = pd.DataFrame(arr1,index=index,
                  columns=['Temp Media','Chuva'])
Out[x]:
                  Temp Media      Chuva
Cidade  Mes
Paris   Jan          4.9          51.0
        Abr         11.5          51.8
        Jul         20.5          62.3
        Out         13.0          61.5
Berlin  Jan          0.1          37.2
        Abr          9.0          33.7
```

Pandas - Indexação Hierárquica

Método `loc` com `MultiIndex`

```
df3.loc['Londres']
df3.loc[('Londres','Abr')]
df3.loc[('Londres','Abr'),'Chuva']
```

Slicing

```
df3.sort_index(inplace=True)
df3.loc['Berlin':'Madri']
```

Método `xs`

```
df3.xs('Jul',level=1)
#no nível 1, procure todos os dados de índice 'Jul'
```

Método `unstack`

```
df3.unstack()
```

Pandas - Valores Inválidos

Dados experimentais ou da Web geralmente contém valores inválidos ou 'perdidos'. Em geral, esses dados perdidos são representados por `NaN`. Pandas possui métodos para detectar, remover ou substituir esses valores:

- `isnull()`: retorna um valor booleano indicando a presença de NaN
- `notnull()`: oposto a `isnull()`
- `dropna()`: retorna uma versão filtrada do conjunto de dados
- `fillna()`: retorna uma cópia dos dados com NaN substituídos por um valor escolhido.

Warning III

Tenha em mente que a retirada de alguns pontos do conjunto de dados poder enviesar a conclusão da sua análise.

Pandas - Valores Inválidos

Detectando Valores Inválidos

Example

```
    A   B    C    D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6  2.9
2  1.2  2.5  1.6  2.7
3  NaN  NaN  NaN  NaN
4  NaN  NaN  3.6  5.3
In[x]: df.isnull()
Out[x]:
    A      B      C      D
0  False  True   True   False
1  False  True   False  False
2  False  False  False  False
3  True   True   True   True
4  True   True   False  False
```

Pandas - Valores Inválidos

Excluindo Valores Inválidos

Example

```
In[x]: df.dropna()
Out[x]:      A    B    C      D
          2  1.2  2.5  1.6  2.7
In[x]: df.dropna(how='all')
Out[x]:
     A    B      C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6  2.9
2  1.2  2.5  1.6  2.7
4  NaN  NaN  3.6  5.3
In[x]: df.dropna(thresh=3, axis=0)
Out[x]:      A    B    C     D
          1  0.8  NaN  3.6  2.9
          2  1.2  2.5  1.6  2.7
```

Substituindo Valores Inválidos I

Example

```
#substitui todos NaN por 1.0
In[x]: df.fillna(1.0)
```

Example

```
#forward-fill (direção COLUNAS)
In[x]: df.fillna(method ='ffill',axis=0)
Out[x]:
    A   B       C   D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6  2.9
2  1.2  2.5  1.6  2.7
3  1.2  2.5  1.6  2.7
4  1.2  2.5  3.6  5.3
```

Substituindo Valores Inválidos II

Example

```
#backward-fill
In[x]: df.fillna(method ='bfill',axis=1)
Out[x]:
     A    B      C      D
0   1.1   10.3 10.3   10.3
1   0.8   3.6   3.6     2.9
2   1.2   2.5   1.6     2.7
3   NaN   NaN   NaN     NaN
4   3.6   3.6   3.6     5.3
```

Example

```
In[x]: df.fillna({'A': df['A'].mean(),
                  'C':df['C'].mean()})
```

Pandas - Valores Duplicados

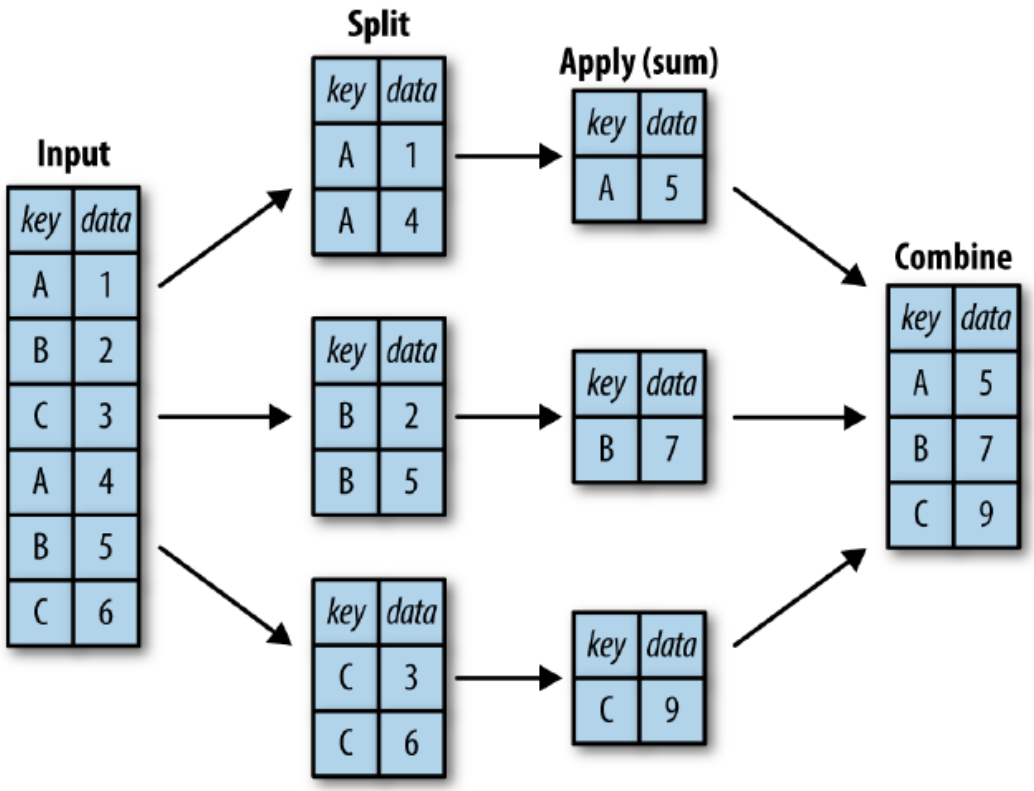
Valores duplicados num conjunto de dados podem ser identificados com o método `duplicates()`, que aponta se uma **linha** tem elementos duplicados em relação a linha anterior. Para remover linhas duplicadas, usamos o método `drop_duplicated()`.

Example

```
   Element     Symbol   Z   A    Abundance
0  Lithium     Li       3   6    0.075900
1  Lithium     Li       3   7    0.924100
2  Potassium   K       19   39   0.932581
3  Potassium   K       19   40   0.000117
In[x]: df.drop_duplicates(['Symbol','Z'])
Out[x]:
    Element    Symbol    Z    A    Abundance
1   Lithium    Li        3    6    0.075900
3   Potassium  K        19   39    0.932581
```

Pandas - Agrupamento

Pode-se agrupar dados com Pandas de acordo categorias utilizando identificadores de colunas ou linhas. O procedimento consiste em **separar**, **aplicar** e **combinar**, ou seja, os dados são separados em categorias, alguma operação é aplicada aos diferentes grupos, e os resultados são combinados. O exemplo abaixo ilustra uma operação de soma.



Pandas - Agrupamento

Example

```
  Letra  data1  data2
0   A      0      2.0
1   B      1      2.5
2   C      2      3.0
3   A      3      3.5
4   B      4      4.0
5   C      5      4.5
In[x]: grupo = df.groupby('Letra')
       grupo.sum()
Out[x]:        data1 data2
      Letra
      A        3     5.5
      B        5     6.5
      C        7     7.5
```

Pandas - Agrupamento

Example

```
In[x]: grupo['data1'].mean()
Out[x]:
Letra
A      1.5
B      2.5
C      3.5
In[x]: grupo.aggregate({'data1':'min','data2':'max'})
Out[x]:
         data1  data2
  Letra
  A          0    3.5
  B          1    4.0
  C          2    4.5
```

Pandas - Concatenação

Podemos concatenar `DataFrames` e `Series` usando o método `pd.concat()`. Dentre várias opções, o argumentos básicos são uma lista de `DataFrames` e o eixo de referência para a junção: `axis=0` (linha, default) ou `axis=1` (coluna).

Example

```
In[x]: pd.concat([df1,df2])
Out[x]:
    A   B
1   A1  B1
2   A2  B2
3   A3  B3
4   A4  B4
```

Álgebra Linear com NumPy - Operações com Matrizes

Diversas operações com matrizes podem ser realizadas no Python com NumPy e com o módulo **numpy.linalg**.

Multiplicação por um escalar

```
In[x]: A = np.array([[0,0.5], [-1,2]])
Out[x]: array([[ 0. ,  0.5],
               [-1. ,  2. ]])
In[x]: A*5
Out[x]: array([[ 0. ,  2.5],
               [-5. , 10. ]])
```

Multiplicação de Matrizes

```
In[x]:  B = np. array([[2, -0.5] , [3, 1.5]])
In[x]:  np.dot(A,B)     #ou A @ B
Out[x]: array([[1.5 , 0.75],
               [4.  , 3.5 ]])
```

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 $\boldsymbol{A}$, um *autovetor* $\boldsymbol{v}$ é um vetor que satisfaz

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

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

Para calcular autovetores e autovetores 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 dispões 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 $\boldsymbol{Mx} = \boldsymbol{b}$

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

Álgebra Linear com NumPy - Sistemas Lineares

Solução de Sistemas Lineares

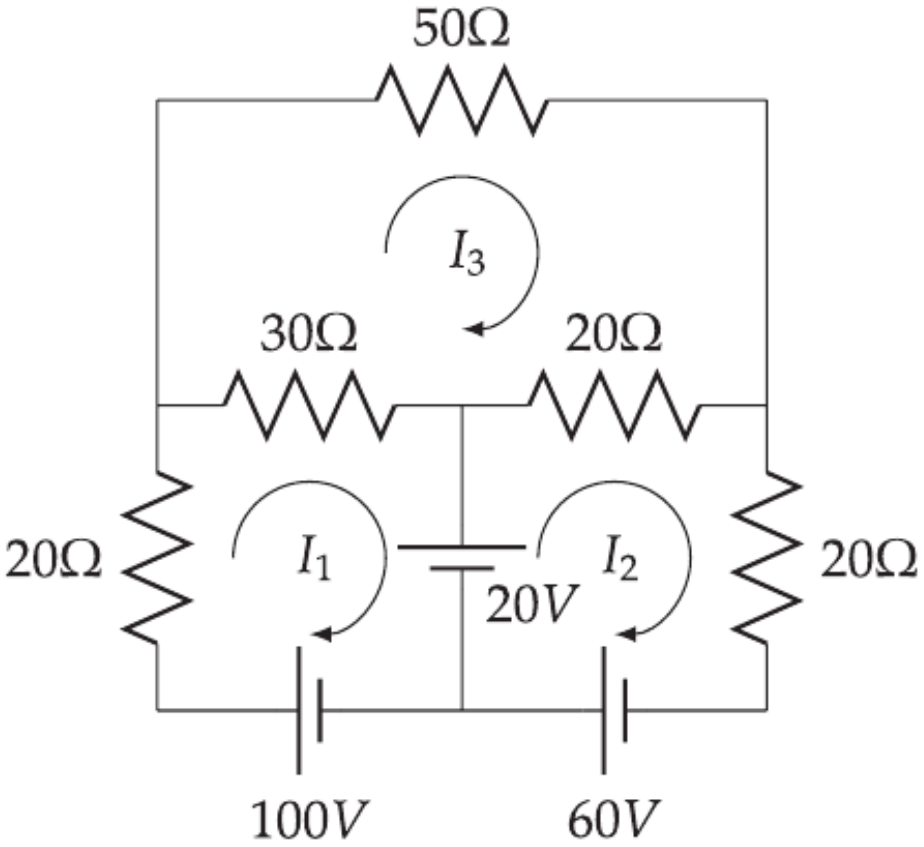
```
In[x]: M = np.array([ [3., -2, 0], [-2, 1, -3],
                      [4, 6, 1]])
In[x]: print(M)
Out[x]: [[ 3. -2.  0.]
         [-2.  1. -3.]
         [ 4.  6.  1.]]
In[x]: b = np.array([8, -20, 7])
In[x]: x, y, z = np.linalg.solve(M,b)
       print('x = {}, y = {}, z = {}'.format(x,y,z))
Out[x]: x = 2.0, y = -1.0, z = 5.0
```

Operadores do Numpy

Operator	Effect	Operator	Effect
dot(a, b[,out])	Dot product arrays	vdot(a, b)	Dot product
inner(a, b)	Inner product arrays	outer(a, b)	Outer product
tensordot(a, b)	Tensor dot product	einsum()	Einstein sum
linalg.matrix_power(M, n)	Matrix to power n	kron(a, b)	Kronecker product
linalg.cholesky(a)	Cholesky decomp	linalg.qr(a)	QR factorization
linalg.svd(a)	Singular val decomp	linalg.eig(a)	Eigenproblem
linalg.eigh(a)	Hermitian eigen	linalg.eigvals(a)	General eigen
linalg.eigvalsh(a)	Hermitian eigenvals	linalg.norm(x)	Matrix norm
linalg.cond(x)	Condition number	linalg.det(a)	Determinant
linalg.slogdet(a)	Sign and log(det)	trace(a)	Diagnol sum
linalg.solve(a, b)	Solve equation	linalg.tensorsolve(a, b)	Solve $ax = b$
linalg.lstsq(a, b)	Least-squares solve	linalg.inv(a)	Inverse
linalg.pinv(a)	Penrose inverse	linalg.tensorinv(a)	Inverse N–D array

Álgebra Linear com NumPy - Aplicação

Exercício. No circuito abaixo, determine os valores das correntes I_1, I_2, I_3.



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

$$\begin{aligned} 50I_1 - 30I_3 &= 80 \\ 40I_2 - 20I_3 &= 80 \\ -30I_1 - 20I_2 + 100I_3 &= 0 \end{aligned}$$

Veja solução em `pcientifico_algLinear.ipynb`

SciPy - Constantes Físicas

SciPy é uma poderosa biblioteca de módulos do Python para computação científica, onde estão implementados diversos algoritmos usados em problemas de física e engenharia. Para instalar:
`conda install scipy`
O pacote `scipy.constants` contém várias constantes físicas tabeladas.

Example

```
In[x]:  import scipy.constants as pc
In[x]:  pc.physical_constants['electron mass']
Out[x]: (9.10938356e-31, 'kg', 1.1e-38)
In[x]:  pc.value('electron mass')
Out[x]: 9.10938356e-31
In[x]:  pc.physical_constants['Planck constant']
Out[x]: (6.62607004e-34, 'J s', 8.1e-42)
```

SciPy - Funções Especiais

O pacote `scipy.special` contém a implementação numérica de várias funções que aparecem constantemente em ciência.
Veja uma lista completa em
`docs.scipy.org/doc/scipy/reference/special.html`

Função Gama

A função $\Gamma(x)$, para x real, é definida como:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

Exemplo:

```
In[x]: from scipy.special import gamma
In[x]:  x = [0.5,1,2.5,4]
        gamma(x)
Out[x]: array([1.77245385, 1., 1.32934039, 6.])
```

Função Beta

A função beta $B(a, b)$ é definida como:

$$B(a, b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt, \quad a > 0,\ b > 0$$

Exemplo:

```
In[x]:  from scipy.special import beta, betainc
In[x]:  beta(0.5,0.25)
Out[x]: 5.244115108584239
In[x]:  betainc(0.5,0.25,1.) #razao B(a,b,x)/B(a,b)
Out[x]: 1.0
```

Integrais Elípticas

Integrais elípticas completas de primeira ordem são definidas, para $0 \leq m \leq 1$, como:

$$K(m) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - m\,\mathrm{sen}^2\theta}}, \quad E(m) = \int_0^{\pi/2} \sqrt{1 - m\,\mathrm{sen}^2\theta}\, d\theta.$$

E as integrais elípticas incompletas:

$$K(\phi, m) = \int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\,\mathrm{sen}^2\theta}}$$

$$E(\phi, m) = \int_0^{\phi} \sqrt{1 - m\,\mathrm{sen}^2\theta}\, d\theta.$$

Os valores dessas integrais são calculados pelas funções `ellipk(m)` `ellipe(m)`,`ellipkinc(phi,m)` e `ellipeinc(phi,m)`, respectivamente.

Exemplo. O período de um pêndulo simples de massa m e comprimento ℓ, com amplitude de oscilação θ_0, é dado por:

$$\tau = 2\sqrt{\frac{\ell}{g}} \int_0^{\theta_0} \left[\mathrm{sen}^2(\theta_0/2) - \mathrm{sen}^2(\theta/2)\right]^{-1/2} d\theta$$

Vamos utilizar as integrais elípticas implementas no scipy para calcular o período do pêndulo e comparar com a aproximação harmônica $2\pi\sqrt{\ell/g}$. Primeiro, sendo $n = 1/\mathrm{sen}(\theta_0/2)$ e $n\,\mathrm{sen}(\theta/2) = \mathrm{sen}\beta$, vamos escrever τ como:

$$\tau = 4\sqrt{\frac{\ell}{g}} \int_0^{\pi/2} \frac{d\beta}{\sqrt{1-(1/n^2)\mathrm{sen}^2\beta}} = 4\sqrt{\frac{\ell}{g}}\, K(\pi/2, 1/n^2)$$

Veja solução em `pcientifico_scipy_special.ipynb`

SciPy - Interpolação

O pacote `scipy.interpolate` contém várias funções para interpolação, tanto em uma como em mais dimensões.

Interpolação em 1D

Dado dois arrays de pontos x e y, o método `scipy.interpolate.interp1d` retorna uma função que pode ser usada para gerar valores interpolados em valores intermediários de x.

Métodos de Interpolação

```
In[x]: from scipy. interpolate import interp1d
....
....
In[x]: f_linear  = interp1d(x, y)
       f_nearest = interp1d(x, y, kind='nearest')
       f_cubic   = interp1d(x, y, kind='cubic')
```

Interpolação em 2D

O método `scipy.interpolate.interp2d` requer uma array z de **duas** dimensões e dois arrays de coordenadas correspondentes, x e y (ambos 1D).

Métodos de Interpolação

```
In[x]: from scipy.interpolate import interp2d
In[x]: x = np. linspace (0, 4, 13)
       y = np.array ([0, 2, 3, 3.5, 3.75 , 3.875 ,
                      3.9375 , 4])
       X, Y = np.meshgrid (x, y)
       Z = np.sin(np.pi*X/2)*np.exp(Y/2)
In[x]: x2 = np.linspace (0, 4, 65)
       y2 = np.linspace (0, 4, 65)
       f = interp2d(x, y, Z, kind='cubic')
       Z2 = f(x2, y2)
```

SciPy - Raízes de Funções

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 sgn$[f(a)]$ = - 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, `x0` (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, newton
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 raíz 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
```

Raízes de Polinômio

Podemos encontrar todas as raízes de um polinômio (reais e complexas), usando a função do NumPy `np.roots()`. Os argumentos são os coeficientes do polinômio.

Raízes Raízes de Polinômio

Exemplo:

$$f(x) = x^4 + x + 1$$

```
In[x]: coef = [1,0,0,1,1]
       np.roots(coef)
Out[x]:
       array([ 0.72713608+0.93409929j,
               0.72713608-0.93409929j,
              -0.72713608+0.43001429j,
              -0.72713608-0.43001429j])
```

SciPy - Sistemas de Eq. não Lineares

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 métido `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:

$$y - x^3 - 2x^2 + 1 = 0,$$
$$y + x^2 - 1 = 0$$

SciPy - Sistemas de Eq. não Lineares

Para codificar esse sistema, devemos escrever uma função na forma $f[x, y] = [y - x^3 - 2x^2 + 1, y + x^2 - 1]$.
No código, `x = x[0]` e `y = 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)
```

SciPy - Otimização

Além de raízes de funções, o pacote `scipy.optimize` conta com diversos algorítimos de para minimizar uma função de uma ou mais variáreis $f(x_1, x_2, ..., x_n)$. Para maximizar uma função, minimizamos $-f(x_1, x_2, ..., x_n)$.
Alguns algorítimos de minimização requerem, além da função, um array com o **jacobiano**:

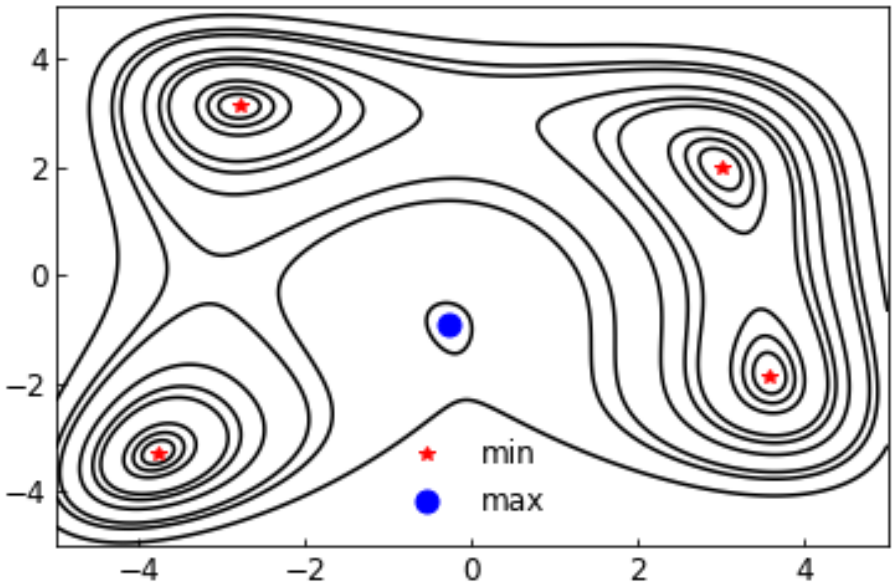
$$J(f) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n} \right)$$

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

SciPy - Otimização sem vínculo

O algorítimo 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:

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



SciPy - Otimização sem vínculo

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])
```

SciPy - Otimização sem vínculo

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])
```

SciPy - Otimização sem vínculo

Algorítimo	**Descrição**
`BFGS`	Broyden-Fletcher-Goldfarb-Shanno (default)
`Nelder-Mead`	Algorítimo de Nelder-Mead
`CG`	Método do gradiente conjugado
`Powell`	Método de Powell
`dogleg`	Necessita Jacobiano e Hessiano
`TNC`	Algorítimo de Newton Truncado (contornos)
`l-bfgs-b`	Algorítimo 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`

SciPy - Otimização sem vínculo

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: 39
success: True
x: array([-0.27087419, -0.9230486 ])
```

SciPy - Otimização sem vínculo

Para usar o método `doglet`, precisamos calcular o Jacobiano e Hessiano

$$J(f) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$$

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial y \partial x} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Maximização - Exemplo 3

```
In[x]: def df(X):
           x, y = X
           f1 , f2 = x**2 + y - 11, x + y**2 - 7
           dfdx = 4*x*f1 + 2*f2
           dfdy = 2*f1 + 4*y*f2
           return np.array ([dfdx, dfdy ])
```

SciPy - Otimização sem vínculo

Exemplo 3 - Continuação

```
In[x]: def ddf(X):
           x, y = X
           d2fdx2 = 12*x**2 + 4*y - 42
           d2fdy2 = 12*y**2 + 4*x - 26
           d2fdxdy = 4*(x + y)
           return np.array ([[d2fdx2, d2fdxdy],
                             [d2fdxdy , d2fdy2 ]])
In[x]: mdf = lambda X: -df(X)
       mddf = lambda X: - ddf(X)
In[x]: minimize(mf, (0,0),jac=mdf, hess=mddf,
                method='dogleg')
```

SciPy - Otimização com vínculo

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 sequencia 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])
```

SciPy - Otimização com vínculo

Para otimizar uma função sujeita a um vínculo (por exemplo, $x = y$), devemos passar um dicionário com as chaves `'type'` e `'fun'`. O valor da chave `'type'` pode ser `'eq'`, se o vínculo for um igualdade (ex. $x = y$), ou `'ineq'`, no caso de uma inigualdade (ex. $x > 2y$). O valor da chave `'fun'` deve ser a função que implementa o vínculo. Os métodos disponíveis são `cobyla` e `slsqp`, mas `cobyla` **não** aceita igualdades. A implementação da função que representa a igualdade deve retornar **zero**, e da desigualdade um valor positivo. Por exemplo, para minimizar uma função sujeita ao vínculo $x = y$, a função de vínculo deve ser $x - y$.

Minimização com vínculo

```
In[x]: fvinc = lambda X: X[0] - X[1]
       vinc = {'type': 'eq','fun': fvinc}
In[x]: minimize(f,(0,0),constraints=vinc,
                method='slsqp')
```

SciPy - Otimização com vínculo

Maximização com vínculo - Ex. 1

```
In[x]: minimize(mf,(0,0),constraints=vinc,
                    method='slsqp')
Out[x]:
 fun: -3.182605330060369e+68
 jac: array([0., 0.])
 message: 'Singular matrix C in LSQ subproblem'
 nfev: 16
 njev: 4
 status: 6
 success: False
 x: array([-1.12315113e+17, -1.12315113e+17])
```

A outra opção disponível, `cobyla`, não aceita igualdade. Neste caso particular, devemos passar **dois** dicionários, com as condições $x - y > 0$ e $y - x > 0$.

SciPy - Otimização com vínculo

Maximização com vínculo - Ex. 2

```
In[x]:
 vinc1 = {'type':'ineq','fun':lambda X: X[0] - X[1]}
 vinc2 = {'type':'ineq','fun':lambda X: X[1] - X[0]}
In[x]: minimize(mf,(0,0),constraints=(vinc1,vinc2),
                method='cobyla')
Out[x]:
 fun: -179.12499987327624
 maxcv: 0.0
 message: 'Optimization terminated successfully.'
 nfev: 34
 status: 1
 success: True
 x: array([-0.49994148, -0.49994148])
```

SciPy - Otimização de Função de uma Variável

Para minimizar uma função de uma variável, o algorítimo mais rápido é `scipy.optimize.minimize_scalar`. Idealmente, devemos passar, com o argumento `bracket`, valores de x como uma tupla (a, b, c), tal que $f(a) > f(b)$ e $f(c) > f(b)$. Se não for possível, passamos um intervalo de valores de x como pontos iniciais. Se nenhum valor inicial é passado, o intervalo inicial tomado é (0,1). Vamos minimizar o polinômio:

$$f(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$$

Minimização - Função 1D

```
In[x]: fp = np.polynomial.Polynomial((48.,28.,
                                      -24.,-3.,1.))
In[x]: minimize_scalar(fp)
```

SciPy - Ajuste de Curvas

SciPy tem diversas rotinas para ajustar uma função a um conjunto de dados. O método `scipy.optimize.leastsq` (mínimos quadrados) é chamado como `scipy.optimize.leastsq(func,x0,args=())`, com os argumentos descritos abaixo:

- `func` é a uma função que deve retornar a **diferença** entre os valores de y e a função a ser ajustada (resíduo);
- `x0` é um valor inicial para os parâmetros ajustados;
- `args` é uma sequência de argumentos com o array de dados, y, e as variáveis independentes.

Como exemplo, vamos ajustar a função $y(t) = Ae^{-\delta t}\cos(\omega t)$ a um conjunto de dados de um experimento que mediu a posição em função do tempo de um oscilador amortecido. Os parâmetros ajustados serão A, δ e ω.

SciPy - Ajuste de Curvas

Quando definimos a função `f` a ser ajustada, a primeira entrada deve ser a variável independente. Neste exemplo, os arrays com os dados são `t` e `y`.

Ajuste com leastsq

```
In[x]: def f(t,A,omega,delta):
         return A*np.exp(-delta*t)*np.cos(omega*t)
.....
In[x]: def residuo(p,y,t):
         A,omega,delta = p
         return y - f(t,A,omega,delta)
In[x]: x0 = [5,5,1]
       result1 = leastsq(residuo,x0,args=(y,t))
       result1[0]
Out[x]:
array([9.91672816, 25.09976487,  1.90909462])
```

SciPy - Ajuste de Curvas

O método `scipy.optimize.curve_fit` é 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.

SciPy - Ajuste de Curvas

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, γ e x_0 serão os parâmetros ajustados.

Veja solução em `pcientifico_otimizacao.ipynb`

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 intergração `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]:
(0.7500000000000002, 1.9132345482589 95e-09)
```

SciPy - Integração Numérica

O argumento `epsabs` permite estabelecer a tolerância absoluta (um limite superior). Vamos integrar a função $f(x) = e^{-|x|}\text{sen}^2 x^2$ entre -1 e 2:

Example

```
In[x]:
f2 = lambda x: np.exp(-np.abs(x))*np.sin(x**2)**2
In[x]: quad(f2,-1,2,epsabs=0.1)
Out[x]:
(0.29551455828969975, 0.0015295571827909423)
In[x]: quad(f2,-1,2,epsabs=1.49e-8)
Out[x]:
(0.29551455505239044, 4.4497633167455447e-10)
```

SciPy - Integração Numérica

Se uma função depende de outros parâmetros além da variável independente, estes devem ser passados como **tuplas** para o argumento `args`.

Example

$$I = \int_{-\pi/2}^{\pi/2} \text{sen}^n x \cos^m x \, dx$$

```
In[x]: def f3(x,n,m):
           return np.sin(x)**n*np.cos(x)**m
In[x]: quad(f3,-np.pi/2,np.pi/2,args=(2,1))
Out[x]:
(0.6666666666666666, 1.6257269518146785e-13)
```

Note que os parâmetros `n` e `m` devem aparecer como argumentos do integrando **depois** da variável de integração `x`.

SciPy - Integração Numérica

Para integrais com limites infinitos, usamos `np.inf`

Example

$$I = \int_0^{\infty} e^{-x^2} dx$$

```
In[x]: f4 = lambda x: np.exp(-x**2)
In[x]: quad(f4,0,np.inf)
Out[x]:
(0.8862269254527579, 7.101318390472462e-09)
```

SciPy - Integração Numérica

Para integrar funções com singularidades, devemos passar um 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]:
(3.9999999999999813, 5.6843418860808e-14)
```

SciPy - Integração Numérica - Integrais Múltiplas

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:

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

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`.

Example

$$I = \int_1^4 \int_0^2 x^2 y \, dydx$$

```
In[x]: f6 = lambda y,x: x**2*y
       g = lambda x: 0
       h = lambda x: 2
In[x]: dblquad(f6,1,4,g,h)
Out[x]:
(42.000000000000001, 4.662936703425658e-13)
```

SciPy - Integração Numérica - Integrais Múltiplas

O método `tplquad` toma como argumentos uma função `func(z,y,x)` e mais seis argumentos para os limites: `a` e `b` (limites de x), `gfun(x)` `hfun(x)` (limites de y), e `qfun(x,y)` e `rfun(x,y)` (limites de z).

Example

$$V = \int_0^{2\pi} \int_0^{\pi} \int_0^{1} r^2 \text{sen}\theta \, dr d\theta d\phi$$

```
In[x]: def f8(phi,theta,r):
           return r**2*np.sin(theta)
In[x]:
gf = lambda r: 0 ; hf = lambda r: np.pi
qf = lambda r,theta: 0; rf = lambda r,theta: 2*np.pi
In[x]: tplquad(f8,0,1,gf,hf,qf,rf)
Out[x]:
(4.188790204786390 5, 1.389095079694993e-13)
```

SciPy - Integração Numérica - Integral de Arrays

Para integrar arrays (*samples*), usamos o método `simpson`:
`simpson(y, x=None, dx=1.0, axis=- 1)`.
O argumento `dx` é o espaçamento entre os pontos, usado apenas quando o array `x` não é dado.

Example

```
In[x]: x = np.arange(0,10)
       y = np.arange(0,10)
In[x]: simps(y,x)
Out[x]: 40.5
```

SciPy - Equações Diferenciais

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

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{F}(\boldsymbol{y}, t)$$

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

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

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

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, `y0` será um sequência de n elementos.

Exemplo 2: Evolução de uma Epidemia (Modelo SIR)

$$\begin{aligned} \frac{dS}{dt} &= -\beta SI, \\ \frac{dI}{dt} &= \beta SI - \gamma I, \\ \frac{dR}{dt} &= \gamma I, \end{aligned}$$

onde as constantes β e γ 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 mais detalhes em
`www.professores.uff.br/andrenepomuceno/artigos`.

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 $x_2 = dx/dt$.

SciPy - Equações Diferenciais

A função `solve_ivp()` pode ser configurada para utilizar diferentes algorítimos com o argumento `method`. O método default é o método de Runge-Kutta (`'RK45'`). Para problemas 'severos' (*stiff problems*), ou seja, problemas onde termos da EDO apresentam variações em escalas muito diferentes, as opções mais adequadas são `'Radau'`, `'BDF'` ou `'LSODA'` (ODEPACK).

Veja uma descrição mais detalhada em
`https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`

Exemplo 4: Problema Severo

Reação química de Robertson:

$$\begin{aligned}
\dot{x} \equiv \frac{dx}{dt} &= -0.04x + 10^4 yz, \\
\dot{y} \equiv \frac{dy}{dt} &= -0.04x - 10^4 yz - 3 \times 10^7 y^2, \\
\dot{z} \equiv \frac{dz}{dt} &= 3 \times 10^7 y^2.
\end{aligned}$$

Vamos resolver esse conjunto de EDO com os métodos `'RK45'` e `'Radau'`.

SciPy - Equações Diferenciais

É possível interromper a integração quando uma determinada condição, definida pelo usuário e chamada *evento*, é atingida. O evento deve ser definido como uma função que retorne zero quando a condição desejada é satisfeita. Como exemplo, considere um carro que se desloca com velocidade inicial $v_0 = 20$ m/s e aceleração $a = dv/dt =$ - 3 m/s^2. Qual o tempo necessário para o carro parar ? Devemos nesse caso definir uma função **carro_parado** como:

```
def carro_parado(t, y):
    return y[0]
```

e passar essa função ao argumento `events`. Se quisermos que a integração seja interrompida nesse ponto, devemos fazer: `carro_parado.terminal = True`.

Veja exemplo em `pcientifico_eqDiferenciais.ipynb`

Para Saber Mais

Livros

- Kinder, J.; Nelson, P. *A Student's Guide to Python for Physical Modeling*.
- Hill, Christian. *Learning Scientific Programming with Python*.
- Landau, Rubin; Paez, Manuel, *et al*. *Computational Physics.*
- Johansson, Robert. *Numerical Python*.
- VanderPlas, Jake. *Python Data Science Handbook*.

Na web

- NumPy
- Matplotlib
- SciPy User Guide
- SymPy Tutorial
- SageMath