5 – File I/O, Plotting with Matplotlib

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

Universität Bremen

Bremen Center for Computational Materials Science BCCMS

https://www.bccms.uni-bremen.de/cms/people/b-aradi/wissen-progr/python/2022

Installing some SciPy stack components

We will need several Scipy components for the exercises:

```
sudo apt-get install python3-scipy python3-matplotlib
```

Alternatively, in the Conda installer use:

```
conda install scipy matplotlib
```

File I/O workflow

- Open file
- Do read/write operations
- Close file

```
fp = open("test.txt", "r")
txt = fp.read()
fp.close()
```

- The closing of a file is optional (although recommended)
- Using context manager blocks (with … as …) closing the file can be automatic
- File would be closed as soon as the block is left

```
with open("test.txt", "r") as fp:
    txt = fp.read()
print("The file has been already closed")
```

Opening a file

- A file is opened by the **open()** function

```
open(filename, mode)
```

- It returns a file handler which can be used to manipulate the file content
- The file handler is valid until the file is closed with the close() statement
- Mode flag determines what can be done with the file and how the file content is handled (as text or binary data)

"r"	Open for **reading** (default)
"w"	Open for **writing** (**truncating** content if already present)
"a"	Open for **writing** (**appending** to existing content)
"b"	**Binary** mode
"t"	**Text** mode (default)
"+"	Open file for **updating** (reading and writing)

Reading from text file

```
fp = open("test.txt", "r")
```

- **Iterating** over file handler returns the lines in the file as strings (including the newline character a the line ends):

```
for line in fp:
    print(line)
```

- The **readlines()** method returns a list of the lines in the file:

```
lines = fp.readlines()
print(lines)
```

- The **readline()** method returns the next line in the file (and empty string if all lines had been read):

```
line = fp.readline()
while line:
    print(line)
    line = fp.readline()
```

- The **read()** method returns the entire file content as one string:

```
txt = fp.read()
print(txt)
```

Writing to text file

```
fp = open("test.txt", "w")
```

- The **write()** method writes a given string into a file
- The **writelines()** method writes a list of strings into a file

```
fp.write("Line 1\n")
```

```
lines = ["Line1\n", "Line2\n"]
fp.writelines(lines)
```

equiv.

```
lines = ["Line1\n", "Line2\n"]
for line in lines:
    fp.write(line)
```

equiv.

```
lines = ["Line1", "Line2"]
fp.write("\n".join(lines))
```

Reading / writing numerical data

- Numpy/Scipy have special routines to read/write data arrays in text form (and also in other formats)

numpy.loadtxt()	Reads data from a file into an array
numpy.savetxt()	Writes array data into a file

test.dat:

```
# Some comment
1  2
3 4
```

```python
data = np.loadtxt("test.dat")
data
```

```
array([[ 1.,  2.],
       [ 3.,  4.]])
```

```python
data2 = np.array([1, 2, 3])
np.savetxt("test2.dat", data2)
```

→ test2.dat:

```
1.000000000000000000e+00
2.000000000000000000e+00
3.000000000000000000e+00
```

Path manipulation

os.path module

- Module with very helpful functions for file name and path manipulations
- **os.path.join()**: Joining path names:

```
import os.path

directory = "schroedinger/harmonic"
fname = "energies.dat"
fname_full = os.path.join(directory, fname)
fname_full
'schroedinger/harmonic/energies.dat'
```

Plotting data with matplotlib

Matplotlib interfaces

- Fully object oriented interface
- Matlab-like simplified interface (**pyplot**)

Matplotlib render engines

- Embedding plots into the IPython/Jupyter notebook

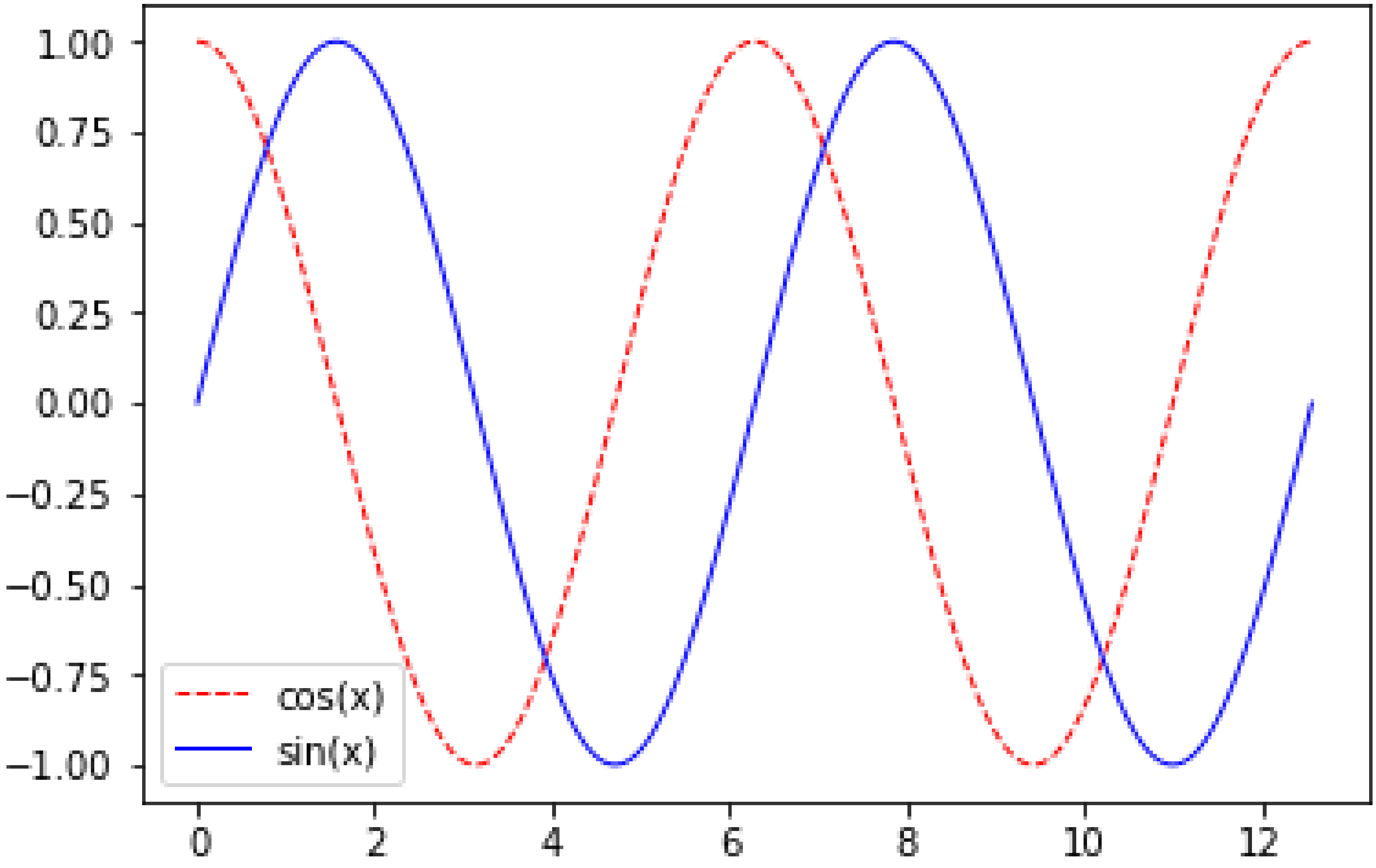
```
%matplotlib inline
```

- Showing plots in separate windows (when using from script or from IPython-console
- Creating graphical files (pdf, jpg, etc.)

Self-containing plotting example

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

xx = np.linspace(0.0, 4.0 * np.pi, 200, endpoint=True)

y1 = np.cos(xx)
y2 = np.sin(xx)

plt.plot(xx, y1, color='red', linewidth=1.0,
         linestyle="--", label='cos(x)')
plt.plot(xx, y2, color='blue', linewidth=1.0,
         linestyle="-", label='sin(x)')

plt.legend()

plt.show()
```

Self-containing plotting example



Plotting with pyplot

```
%matplotlib inline
```

Embed figures into Jupyter-notebook
(Leave this out if you do not work in a Jupyter notebook)

```
import numpy as np
import matplotlib.pyplot as plt
```

Use simplified (pyplot) interface

```
xx = np.linspace(0.0, 4.0 * np.pi, 200, endpoint=True)
```

Generate x-coordinates of the points to plot

200 points evenly distributed in the interval [0.0 to 4 * pi],

Including the upper bound

Plotting with pyplot

```
y1 = np.cos(xx)
y2 = np.sin(xx)
```

Generate the y-coordinates of the points to plot (two curves)

```
plt.plot(xx, y1, color='red', linewidth=1.0,
         linestyle="--", label='cos(x)')
plt.plot(xx, y2, color='blue', linewidth=1.0,
         linestyle="-", label='sin(x)')
```

Plot the points xx, y1 and xx, y2 (and connect them)

Set line color to red/blue

Set line width to 1.0 pixel

Set line style to dashed/solid

Set curve label to cos(x) / sin(x)

Plotting with pyplot

```
plt.legend()
```

Plot legend box

```
plt.show()
```

Render figure on screen

Alternative rendering into file:

```
plt.savefig('curves.pdf', format='pdf')
```

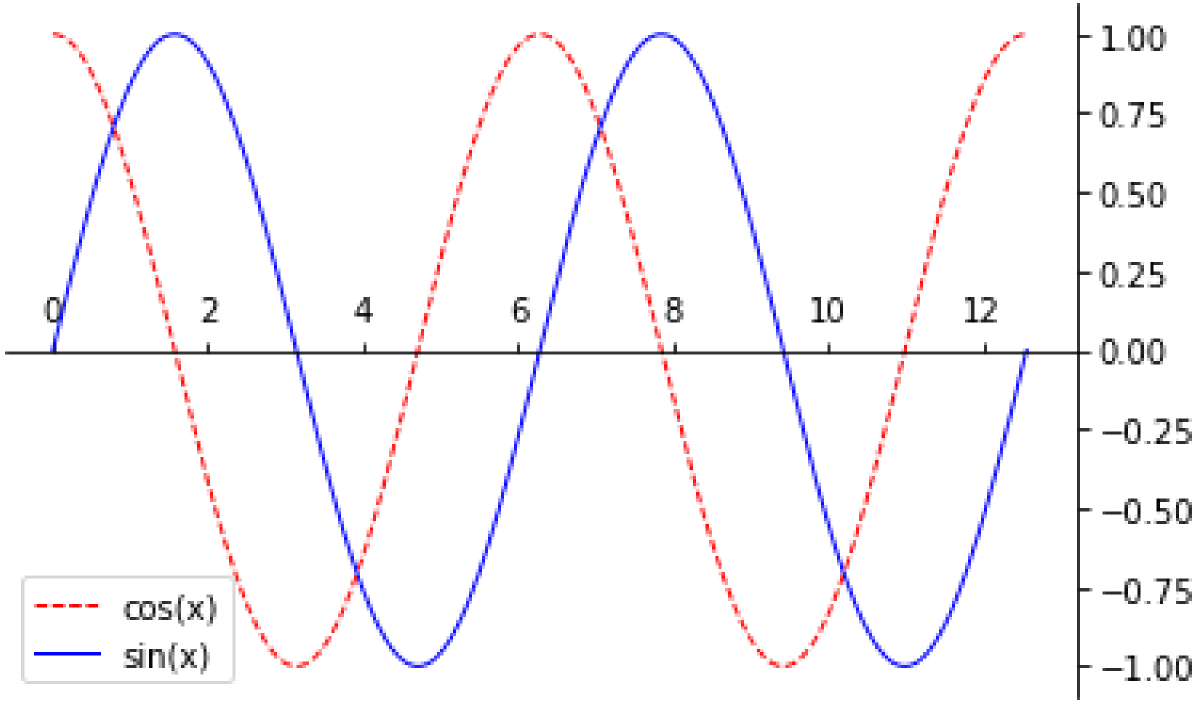
Render figure into file

Axis object

Axis objects enables access to several fine-tuning settings

```
ax = plt.gca()
ax.xaxis.set_ticks_position('top')
ax.yaxis.set_ticks_position('right')
ax.spines['top'].set_position(('data', 0))
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none)
```

Get current axis



Subplots

Plotting of multiple plots on a grid within one figure:

```
plt.subplot(2, 1, 1)
plt.plot(xx, y1, color='red', linewidth=1.0,
         linestyle="--", label='cos(x)')
plt.legend(loc='upper right')

plt.subplot(2, 1, 2)
plt.plot(xx, y2, color='blue', linewidth=1.0,
         linestyle="-", label='sin(x)')
plt.legend(loc='upper right')
plt.show()
```

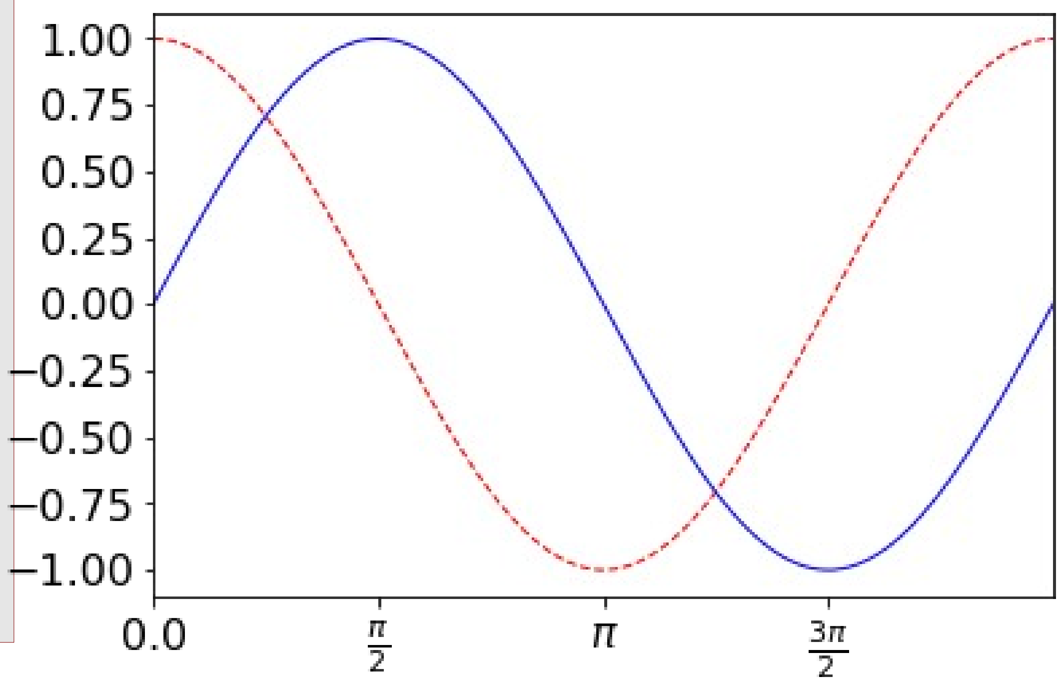
```
plt.subplot(nrow, ncol, iplot)
```

nrow	Number of grid rows
ncol	Number of grid columns
iplot	Current plot nr. (left to right, top to bottom)

Rendering TeX within plots

Matplotlib can render TeX sequences within the plots

```
plt.xticks(
    [0.0, np.pi / 2, np.pi,
     3 * np.pi / 2],
    [r'$0.0$',
     r'$\frac{\pi}{2}$',
     r'$\pi$',
     r'$\frac{3\pi}{2}$'],
    fontsize=16)
```



- TeX-sequences should be delimited by $
- It is advisable to put TeX-sequences into **raw-strings** (**r'something'**)
- In raw-strings, backslashes are interpreted literally and not as special Python commands (e.g. \n as "\" "n" and not as newline)
- Useful when passing backslash commands to various enginens (TeX-sequences in Matplotlib, regular expressions, ...)

Further useful Matplotlib functions

plt.xlim(), **plt.ylim()**	Setting/Querying x/y limits
plt.xticks(), **plt.yticks()**	Setting customized ticks (and tick labels)
plt.annotate()	Write text into the plot
plt.plot()	Curve plot
plt.scatter()	Scatter plot
plt.bar()	Bar plot
plt.contour()	Contour plot
plt.imshow()	Bitmap image
plt.pie()	Pie charts
plt.quiver()	Quiver plots
:	

See for example **Matplotlib: plotting** in **Scipy-lectures**