Pyplot, Numpy & SciPy Cheatsheet

Basic Plotting

Handy imports

```
import numpy as np # maths (esp. vectors and matrices)

import matplotlib.pyplot as plt # plotting
import matplotlib.image as mpimg # loading images

import vtk # the visualization toolkit

from pydicom import dcmread # reading DICOM files

from scipy import signal # signal processing (e.g., Fourier)
from scipy import ndimage # image processing (e.g., gradient)
from scipy import special # additional functions
```

Create a figure

```
# Figures are the container for multiple plots
fig = plt.figure(figsize=(9,4))

# Add some sub-plots (called axes)
axs1 = fig.add_subplot(1, 3, 1) # 1x3 grid, position 1
axs2 = fig.add_subplot(1, 3, 2) # 1x3 grid, position 2

# Define optional titles / legends
axs2.set_title('Title')
axs2.set_xlabel('Label (x)')
axs2.legend()
axs2.yaxis.set_visible(False) # sets the y-axis invisible

# Define what you want to draw
# ...
# ...

# Run this at the end of the file to display everything
plt.show()
```

Plot a line graph

```
# Arbitrary function (uses numpy)
def function(x):
    return np.exp(-x) * np.cos(2 * np.pi * x)

# Create a 1D sequence from 0 to 10 with 0.01 increments
x = np.arange(0.0, 10.0, 0.01)

# Apply a function to the whole array
y = function(x)

# Plot the results
axs1.plot(x, y, label='optional label')
```

To define the style of the plot, a variety of parameters can be used.

Scatterplots

```
# Positions are given in two arrays: x,y
ax.scatter(x, y)

# The point area can be individually set with another array
ax.scatter(x, y, s=areas)

# Other useful parameters include:
#   - c (scalar array for coloring points with a colormap)
#   - cmap (which colormap to use)
#   - vmin (minimal reference value for colormap)
#   - vmax (maximal reference value for colormap)
```

3D Plotting

```
# Meshgrid
# Given two axes meshgrid builds a matrix with
#   X = 2D array of all row indices
#   Y = 2D array of all column indices
X, Y = np.meshgrid(x, y)

# Example:
x = y = np.arange(0, 3, 1) # -> [0, 1, 2]
X, Y = np.meshgrid(x, y)
# This results in the grids/matrices:
# X         Y
# 0 1 2     0 0 0
# 0 1 2     1 1 1
# 0 1 2     2 2 2
```

```
# Arbitrary functions can be evaluated on a meshgrid
Z = np.sin(np.sqrt(X**2 + Y**2))

# Plot as a color-coded height field...
ax.imshow(Z)

# ...or plot as a 3D surface (requires projection='3d')
ax3D = fig.add_subplot(1, 3, 3, projection='3d')
ax3D.plot_surface(X,Y,Z)
```

3D Triangulations

A triangulated surface can be plottet with `plot_trisurf`. This requires the points of all triangles to be stored in 3 flat arrays, one for each component x, y, and z. Example for a single triangle:

```
x_coords = np.array([0.0, 1.0, 1.0])
y_coords = np.array([0.0, 0.0, 1.0])
z_coords = np.array([0.0, 0.0, 0.0])
```

The connectivity information needs to be stored in a matrix of shape $|\mathrm{triangles}| \times 3$. Each row contains the indices of the three points that make up one triangle, ordered anticlockwise.

```
# Connectivity matrix (for one triangle)
t = np.array([[0, 1, 2]])

# Plot
ax.plot_trisurf(x_coords, y_coords, z_coords, triangles=t)
```

> Note that if `triangles` is not specified, the *Delaunay triangulation* is calculated.

Quiverplots

Arrows can be used to visualize a vector field. Pyplot conveniently provides a function for just that, the *quiver plot*.

```
# This requires:
#   - positions X, Y (e.g., from np.meshgrid)
#   - arrow directions U, V
#   - an optional size parameter to scale the arrows
ax.quiver(X, Y, U, V, scale=0.5)
```

Images

1D Numpy arrays can be treated as vectors while 2D arrays can be treated as matrices. Note that:

- Images are just 2D arrays/matrices with values in [0,255]. If loaded with matplotlib this value range is per default scaled to [0.0, 1.0] (double precision).

- *Colored* images can be seen as 3 stacked matrices, one for each channel red, green, and blue. We will only use grayscale images.
- The result of a 2D function (a function with two inputs) will also be stored in a 2D array. This means it can be handled and displayed like an image.

Loading an image

```
img = mpimg.imread('image.jpg')

# If the image is loaded with three dimensions (red, green, blue)
# this will result in three stacked matrices containing the pixel values.
# The dimensions can be displayed with:
print(img.shape)

# If 3 channels are given extract one of these matrices (using numpy array
slicing).
# Assuming the rgb-values are similar, this is a conversion to grayscale
img_bw = img[:,:,0]
```

Displaying an image

```
# A 2D matrix can be displayed using 'imshow', rendering each entry as a pixel.
axs1.imshow(bw_img)

# The image is not in grayscale because the default colormap is applied.
# To display an image with gray levels, the 'gray' colormap can be used.
axs1.imshow(bw_img, cmap='gray')

# When reading the image, all values were scaled to floats in [0,1].
# To correctly scale the values in the output, this range can be specified.
axs1.imshow(bw_img, cmap='gray', vmin=0, vmax=1)
```

Numpy & SciPy

Creating arrays

```
# Conversion from python list to numpy array
a = np.array([0, 1, 2])

# Create a 1D sequence [0.0, 1.0, 2.0, 3.0]
x = np.arange(0.0, 4.0, 1.0) # sets the increment [start, stop)
# alternatively:
x = np.linspace(0.0, 3.0, 4) # sets the size [start, stop]

# Fill with defined value
a = np.zeros(100) # 100 zeros (1D)
```

```
a = np.zeros((100,20)) # 100x20 matrix
a = np.zeros(b.shape) # copy shape of b
a = np.ones(100) # [1,1,1...]
a = np.full((20, 10), 5) # a 20x10 matrix filled with 5
a = np.eye(n) # identity matrix size nxn

# Fill defined shape with random value
noise = np.random.random_sample(a.shape)

# Copy array data
a = b # a and b point to the same data
a = np.copy(b) # a and b point to different data
```

Accessing and Dimensions

```
# Sizes
shape = a.shape # shape of an array (returns a tuple)
size = a.size # number of total elements

# Slicing
a = b[0:3] # elements 0,1,2
a = b[4:]  # element 4 to last
a = b[:5]  # elements 0,1,2,3,4
a = b[:-1] # all elements except the last

# 2D access: select element at position i (row), j (column)
element = a[i,j]

# 2D Slicing
# Example: Select whole column/row from a matrix.
# The shortcuts from above work equivalently.
column = a[:,c]
row = a[r,:]
```

```
# Working with axes
# Example: sum of each column/row in matrix M
columns_sum = np.sum(M, axis=0)
rows_sum = np.sum(M, axis=1)
```

```
# Array masking
# Example: set values below t to 0
A[A < t] = 0

# Example: select the min but exclude 0
min_val_nonzero = np.min(A[np.nonzero(A)])

# Example: Compute A-B only where A>B, everywhere else write 0
result = np.zeros(A.shape)
mask = A > B
C[mask] = A[mask] - B[mask]
```

```
# Interweaved stack
# two 2D arrays -> 3D array
C = np.dstack((A, B))

# Example:
# A: [[0, 1, 2], [3, 4, 5]]
# B: [[6, 7, 8], [9, 10, 11]]
# C: [[[0,6], [1,7], [2,8]], [[3,9], [4,10], [5,11]]]
```

Minima / Maxima

```
# total min/max
min_val = np.min(a)
max_val = np.max(a)

# max values of each row/column
columns_max = np.max(M, axis=0)
rows_max = np.max(M, axis=1)

# Element-wise maximum of two arrays (with same shape)
max_values = np.maximum(a, b)
```

Useful build-in functions

```
a = np.abs(array) # absolute values, can convert complex numbers (!)
a = np.sqrt(array) # square roots
a = np.sin(array) # trigonometric functions
a = np.sum(array) # sum of all entries
a = np.cumsum(array) # cumulative sum along the (flat) array
a = np.power(array, b) # array^b (shorthand array**b)
a = np.exp(array) # exponential function
a = np.math.factorial(x) # x!
a = np.sign(x) # -1 if x < 0, 0 if x==0, 1 if x > 0

# Clips the array to the specified range
a = np.clip(array, min_val, max_val)

# Image histogram, assuming values in [0,255]
# This returns a tuple of two arrays:
# hist = (bins, bin_bounds)
hist = np.histogram(img_bw, bins=256, range=(0,255))

# binomial coefficient
a = scipy.special.binom(n, k)

# Constants
np.pi
np.e
```

Vectors and Matrices

```
# Addition / Subtraction
A = A + B # element-wise addition (requires same shape)
A = A + b # adds a scalar b to every element in A

# Multiplication
np.multiply(A, B) # element-wise multiplication (shorthand: A*B)
np.matmul(A, B) # matrix multiplication (shorthand: A@B)
np.dot(v, w) # dot product (use with vectors)
np.outer(v, w) # outer product (creates a matrix given two vectors)
np.cross(v, w) # cross product

# Norm
norm = np.linalg.norm(v) # length
dist = np.linalg.norm(v - w) # distance

# Other operations
trace = np.trace(A) # trace of a matrix
A = np.transpose(A) # transpose the matrix

# Solving the system Ax = b
x = np.linalg.solve(A,b)

# Eigenvalues/vectors of A (square matrix)
# eig_vals[i] corresponds to eig_vecs[:,i]
eig_vals, eig_vecs = np.linalg.eig(A)

# sort by increasing eigenvalues
idx = eig_vals.argsort()
eig_vals = eig_vals[idx]
eig_vecs = eig_vecs[:,idx]
```

Convolution

```
# 2D convolution (image filtering)
#  - the image is mirrored at the boundary ('symm')
#  - the dimension is retained ('same')
img_filtered = scipy.signal.convolve2d(img, kernel, boundary='symm',
mode='same')

# Default Gaussian filter
img_gauss = scipy.ndimage.gaussian_filter(img, sigma)

# Gradient (Sobel filter)
grad_x = scipy.ndimage.sobel(img, axis=1)
grad_y = scipy.ndimage.sobel(img, axis=0)

# Gradient magnitude (Gauss + Sobel + magnitude)
grad_mag = scipy.ndimage.gaussian_gradient_magnitude(img, sigma)
```

Fourier

```
img_spectrum = np.fft.fft2(img) # Fast 2D Fourier Transform
img_spectrum_shifted = np.fft.fftshift(img_spectrum) # shift to center
re = np.fft.ifft2(np.fft.ifftshift(img_spectrum_shifted)) # reverse shift and
FFT
```

General Guidelines

Vectorized Operations

Note that due to Python being an *interpreted* language, some operations take a lot of time when executed on a top level. This is especially true for nested for-loops. Take, for example, the computation of the gradient magnitude from the two directional gradients, which could be written as:

```
# BAD EXAMPLE (nested Python loops)
grad_mag = np.zeros(img.shape)
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        grad_mag[i,j] = sqrt(grad_x[i,j]**2 + grad_y[i,j]**2)
```

When using Numpy, most operations can be carried out in a *vectorized* style. The above could be written as:

```
# GOOD EXAMPLE (vectorized)
grad_mag = np.sqrt(grad_x**2 + grad_y**2)
```

This is not only easier to read but also much faster, as all loop executions are referred to Numpy, which internally uses highly efficient C code instead of Python.

For this reason, you should make adequate use of vectorized operations to avoid nested for-loops if possible. Sometimes, array slicing and array masking are powerful tools that allow vectorization even when it first seems like a nested loop is required.