Università degli Studi di Pavia

Bachelor degree in Artificial Intelligence

`numpy` and `matplotlib` minimal tutorial

STEFANO FERRARI

Computer programming, Algorithms and Data str., Mod. 1

Contents

1 Matrix representation 4

2 Matrix visualization 5

2.1 Colors and colormaps 5

2.2 Visualization with `matplotlib` 6

2.3 Full color images visualization 8

3 Plot organization 9

4 Interactive display 10

References 11

date: January 24, 2023
version: 0.2

This document aims to provide a basic tutorial to help the study of the `numpy` [1] and `matplotlib` [2] Python libraries for the 2022/23 final project (Planisuss).

The project requires to develop a simulation which provides data that have to be properly diplayed. The user will use the graphical interface to analyze the evolution of the simulated system. The `matplotlib` library provides functionalities to operate data visualization and user interaction. The main datastructures used by `matplotlib` are arrays, matrices, and multidimensional arrayes provided by the `numpy` library.

1. Matrix representation

Matrices in Python can be natively represented as `list` of rows or columns vectors, which in turn can be also represented as `list`s.

For instance, the row vector $[1, 2, 3]$ can be represented in Python as:

```
v = [1, 2, 3]
```

and the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
m = [[1, 2, 3], [4, 5, 6]]
```

In the above implementation, the row-wise representation has been considered. In fact, also the following column-wise representation may be used:

```
m = [[1, 4], [2, 5], [3, 6]]
```

Clearly, the choice of the implementation depends on the usage (e.g., the computations) that will be done in the program. In addition, the programmer must take care of the the various aspects that make the matrix a well-shaped matrix. For instance, all the rows and columns must have the same number of elements, and the elements must be of a numeric type (not `str`, at least).

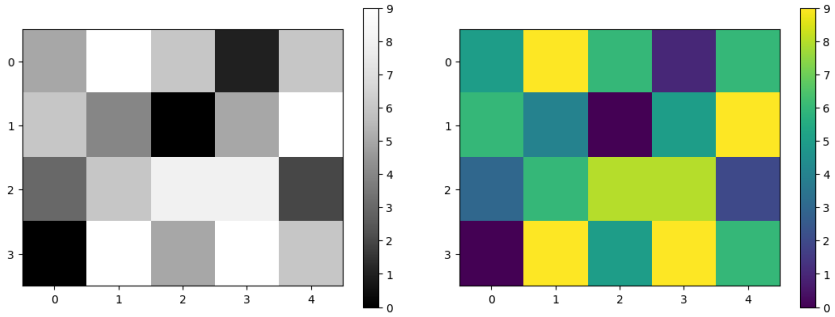
The `numpy` package [1] provides a handy implementation of the matrices. Actually, two classes can be used: `matrix` and `ndarray`. For strictly matricial computations, probably `matrix` is more suited. However, `ndarray` offers more flexibility. In fact, `matrix` can be used to represent arrays and (bidimensional) matrices, while `ndarray` can represent generic n-dimensional matrices [3] (a 3-dimensional matrix can be visualized as a volumetric matrix, i.e., as formed by same-shape matrices structured in layers).

For the following, `ndarray` will be considered. For instance, the above defined matrix `m` can be converted in `ndarray`:

```
m = [[1, 2, 3], [4, 5, 6]]
b = np.asarray(m)
```

The elements can be accessed using a convenient notation: the indices uses the same notation as for the list elements/slices, but can be provided in the same square bracket, as a comma-separated list. For instance, the second element of the first row can be accessed as `b[0,1]`, while the third column is `b[:, 2]`.

$$A = \begin{bmatrix} 5 & 9 & 6 & 1 & 6 \\ 6 & 4 & 0 & 5 & 9 \\ 3 & 6 & 8 & 8 & 2 \\ 0 & 9 & 5 & 9 & 6 \end{bmatrix}$$



(a) (b)

Figure 1: Visual representation of a matrix using two different colormaps.

2. Matrix visualization

When a matrix is too large to be analyzed by reading the value of its elements (or is physically impossible to print all of its elements on a single page), the visualization of the values can be realized using a graph and a suitable color-code to map values in a colors. The matrix is then visualized by plotting a colored point on a cartesian axis, where row and column of the element identify its position on the plane and the color of the point gives an approximate feeling of its value. The matrix is hence considered a digital image and, similarly, displayed.

For instance, the matrix A:

$$A = \begin{bmatrix} 5 & 9 & 6 & 1 & 6 \\ 6 & 4 & 0 & 5 & 9 \\ 3 & 6 & 8 & 8 & 2 \\ 0 & 9 & 5 & 9 & 6 \end{bmatrix}$$

can be displayed both as in Fig. 1a or Fig. 1b.

The rule that maps a number into a color is called "colormap". In the case in Fig. 1a, the map is a linear map that convert a number in the interval [0, 9] into a number in the interval [0, 1], where 0 is black and 1 is white. The case in Fig. 1b is more complex because requires to understand how colors are represented in digital systems.

2.1. Colors and colormaps

The colors are a personal feeling related to our visual perception. Simplifying, we can describe a color as a combination of the three primary colors: red, green, and blue. Different proportions of the primary colors result in all the colors that can be represented: the quantity of each color is a coordinate of a three-dimensional space (the RGB color space).

In digital devices, the color coordinates are also called "channels": the quantity of red is sent through the R-channel to be displayed. Tipically, the color channels support 256 different shades of the corresponding primary color. Hence, each color can be represented with three bytes ($2^8 = 256$), each representing integers in [0, 255]. For instance, yellow can be represented as [255, 255, 0], orange is [255, 69, 0], and brown is [139, 69, 19].

The color map in Fig. 1b is then a function that maps [0, 9] into $[0, 255]^3$: every number in the matrix corresponds to a triplet.

Although grey shades require only a single number (the intensity of white) to be represented, they also have a representation in the RGB (3D) color space: the black is [0, 0, 0], while white is [255, 255, 255]. All the shades are represented as a triplet having the same three coordinates.

2.2. Visualization with `matplotlib`

The `matplotlib` library provides several functionalities for data display and makes use of `numpy` for the data structures.

For instance, Figs 1a-b are actually generated using the following code:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # create the matrix
5  A = np.asarray([[5, 9, 6, 1, 6], [6, 4, 0, 5, 9],
6  [3, 6, 8, 8, 2], [0, 9, 5, 9, 6]])
7
8  # display the first version
9  plt.imshow(A, cmap='Greys_r')
10 plt.yticks([0, 1, 2, 3])
11 plt.colorbar()
12
13 # in a second figure, display another version
14 plt.figure(2)
15 plt.imshow(A)
16 plt.yticks([0, 1, 2, 3])
17 plt.colorbar()
18
19 plt.show()
```

After creating the matrix `A`, the graph is generated by the `imshow` function. The plots are hosted in separated windows, called "figures". The first one is created automatically by invoking `imshow` (line 9). The `imshow` parameters are the matrix and the name of the colormap. Then two commands to manage the appearance of the graph are applied: `yticks` to display only the integer values on the vertical axis, and `colorbar` to display the colormap.

The second figure is created using `figure(2)` (on line 14). The invocation of `imshow` with the second colormap (line 15) do not explicitly mention the colormap name, since the colormap used is actually the default colormap. At the end, to effectively display the figures, the `show()` function is invoked on line 19.

Similarly, the maps in Fig. 2 are represented in the two colormaps. It is apparent that the use of the color allows for a better perception of the details in the data.

The three maps can be combined to be shown in a single color image. In fact, each map can constitute the input for a channel. The result is reported in Fig. 3. In order to achieve it, the maps (actually three bidimensional matrices) have to be rescaled in the interval [0, 1] and then stacked toghether. The following section describes in detail the script to realize this processing.

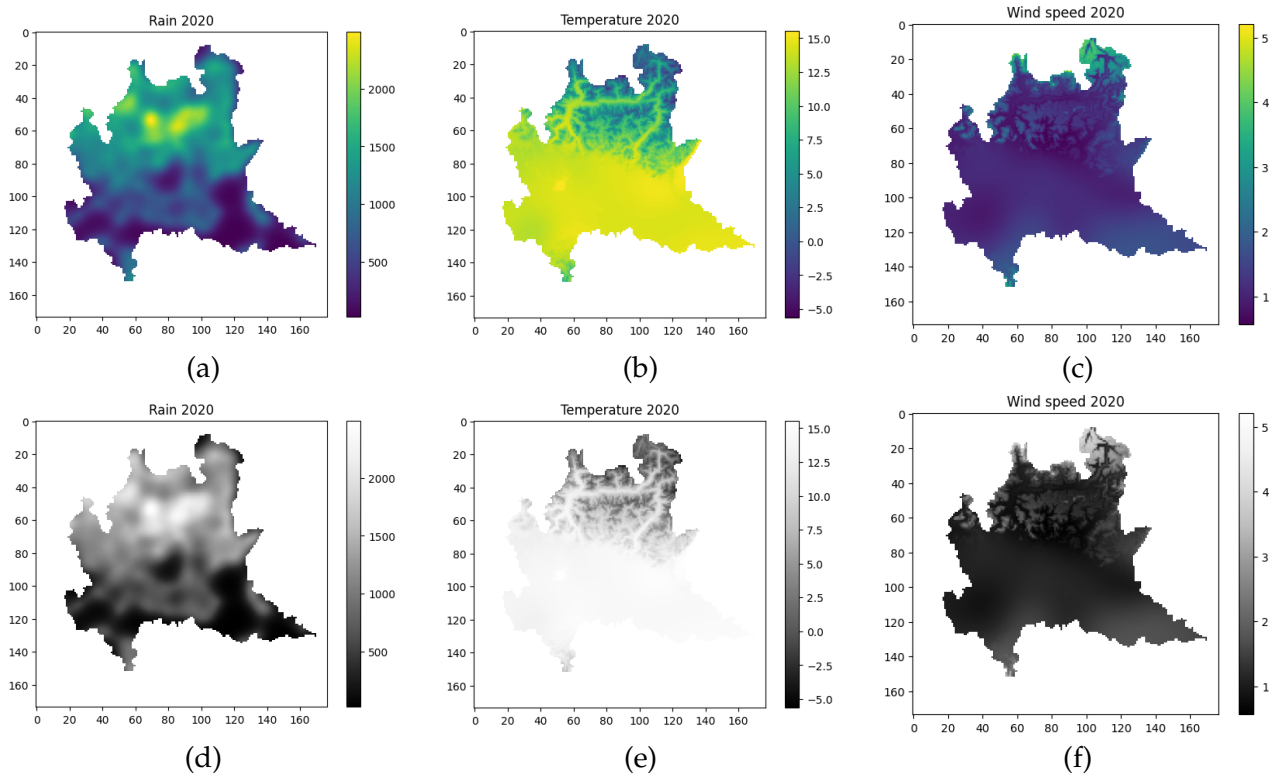


Figure 2: Visual representation of climatic data of Lombardy (Italy) in 2020. (a) rainfall, (b) average temperature, and (c) average wind speed magnitude. In (d)–(f) the same data in (a)–(c) are visualized using a grey-shades colormap.

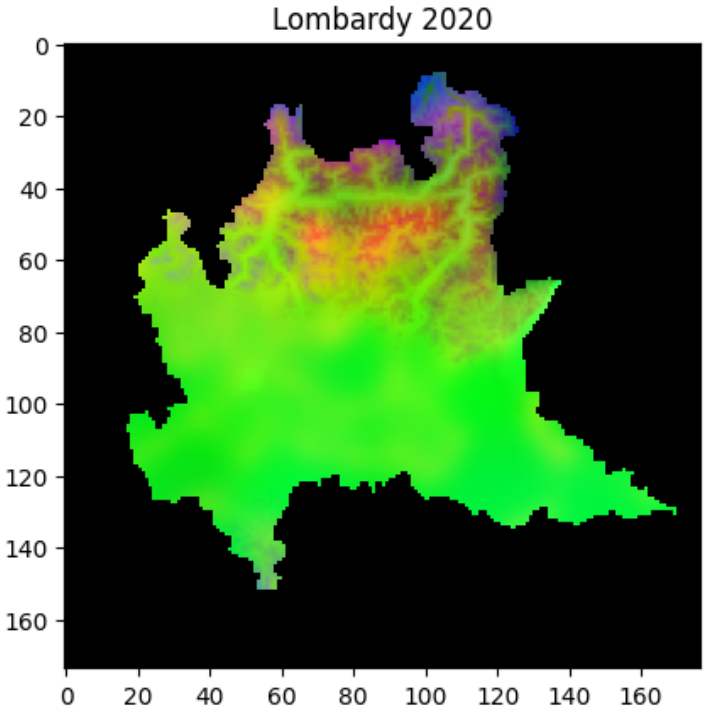


Figure 3: Visual representation of climatic data of Lombardy (Italy) in 2020 in Fig 2. The three maps have been combined in a single RGB image, where RGB channels are used to represent rainfall, temperature, and wind speed respectively.

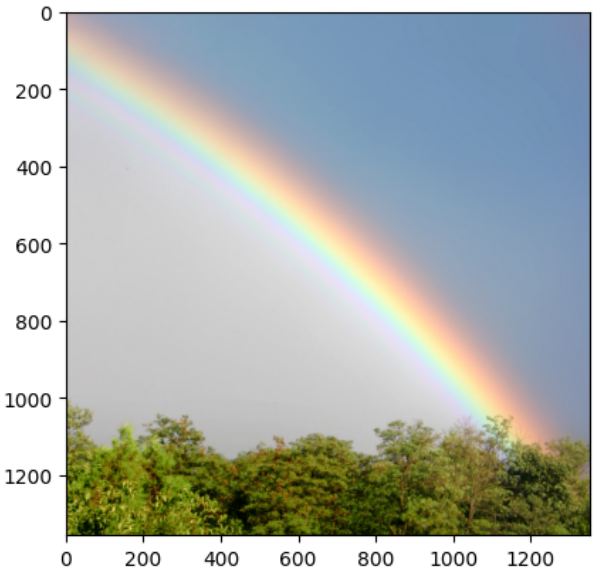


Figure 4: Digital image read from a file and displayed through `matplotlib`. Image from [4].

2.3. Full color images visualization

The `matplotlib` provides functions to load and display images. The function `pyplot.imread` reads an image from a file and returns the image as an `ndarray`.

The following script displays the figure reported in Fig. 4.:

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

rainbow = plt.imread('Rainbow_in_Budapest.jpg')

plt.imshow(rainbow)

plt.show()
```

The variable returned by `imread` is a (1356, 1356, 3)-`ndarray`, which (first) elements are:

```
[[[183 166 174]
  [182 166 176]
  [182 166 176]
  ...
  [122 138 172]
```

Hence it can be imagined as a list of lists which elements are 1×3 arrays (the RGB coordinates of the pixel).

`numpy` allows to combine three matrices of the same size such that each element goes to one of the three coordinates. Hence, each coordinate constitutes one of the channels of the RGB image.

Using the matrices containing the maps displayed in Fig. 2, this procedure is applied in the following script:

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

```
3
4   def normalize_matrix(data):
5       """ Rescale a matrix in [0, 1] """
6       return (data-np.nanmin(data))/(np.nanmax(data)-np.nanmin(data))
7
8
9   rain = normalize_matrix(np.load('PR_2020.npy'))
10  temp = normalize_matrix(np.load('TEMP2m_2020.npy'))
11  wind = normalize_matrix(np.load('VU_2020_VV_2020.npy'))
12
13  meteo = np.dstack((rain, temp, wind))
14
15  plt.imshow(meteo)
16  plt.title('Lombardy 2020')
17  plt.savefig('Figure_4.png', bbox_inches='tight')
18
19  plt.show()
```

The matrices are read from files in line 9–11 (by means of `np.load`) and normalized through a simple rescaling as defined in `normalize_matrix`. Here, the maximum and minimun values of a matrix are computed (`np.nanmax` and `np.nanmin`) and used to rescale the matrix elements in [0, 1]. These operations are expressed in matricial notation (no nested for loops to access to each element of the matrix) and are provided by `numpy`. In particular, the minimum of the matrix is first subtracted to all the elements of the matrix (`data-np.nanmin(data)`) and then divided by the span of the data (`np.nanmax(data)-np.nanmin(data)`). Then, in line 13, the resulting matrices are combined through `numpy.dstack` in a 3-D `ndarray`, which is then displayd using `imshow`.

A last detail: the floating point representation has a particular value (`'nan'`, not-a-number) to represent a value that cannot be represented. For instance, the result of a division by zero can assume this value. Any combination of `'nan'` with a legal number (e.g., through addition) results in a `'nan'`. This value is used for the elements of the maps that represent a location outside the Lombardy region (that geographically do not belong to Lombardy). Hence, in the computation of minimum and maximum, these elements have to be excluded. The functions `numpy.nanmin` and `numpy.nanmax` do exactly this. Whereas `'nan'` have to be considered, the functions `numpy.min` and `numpy.max` have to be used instead.

3. Plot organization

The `matplotlib` allows for a great flexibility in structuring the data visualization graphs in a figure. Besides allowing a complete customization of the plot features (axis, lines, ticks, labels, title, to mention some), `matplotlib` (in particular through the subpackage `matplotlib.pyplot`) provides the mechanisms to arrange several graphs on the same figure:

- `axes`: a class that describes a reference system that can be positioned in the graph container (the figure); this allows to position plots arbitrarily, e.g., plots in plot.
- `subplots`: allow to arrange the graphs on a regular grid.
- `GridSpec`: a class that defines a semi-regular tessellation of the figure and allows to assign one or more adjacent portions to a plot.

A good introduction to these tools is given in [5].

4. Interactive display

The interactiveness with the displayed data can be useful to inspect a particular section of the data or the change the visual properties of the plot. This can be realized through `matplotlib` using the event handling mechanisms provided by `FigureCanvasBase.mpl_connect`. The attribute `Figure.canvas` is the area onto which the figure is drawn and allows to bind actions (i.e., functions) to events.

The following script is a minimal example of this functionality. It captures the mouse clicking and key pressing events, and run the corresponding handling functions. In particular, the mouse clicking prints the mouse location. Similarly, the keyboard pressing prints the pressed key. Besides this, if the pressed key is in `('R', 'r', 'G', 'g', 'B', 'b')`, the figure displays only the corresponding channel of the image in Fig. 3. If `'a'` or `'A'` is pressed, the full color image is displayed again.

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

def normalize_matrix(data):
    """ Rescale a matrix in [0, 1] """
    return (data-np.nanmin(data))/(np.nanmax(data)-np.nanmin(data))

def onclick(event):
    print('%s click: button=%d, x=%d, y=%d, xdata=%f, ydata=%f' %
          ('double' if event.dblclick else 'single', event.button,
           event.x, event.y, event.xdata, event.ydata))

def on_key(event):
    print('you pressed', event.key, event.xdata, event.ydata)
    if event.key == 'r' or event.key == 'R':
        meteo = np.dstack((rain, np.zeros_like(rain), np.zeros_like(rain)))
        title = 'Rain - Lombardy 2020'
    if event.key == 'g' or event.key == 'G':
        meteo = np.dstack((np.zeros_like(temp), temp, np.zeros_like(temp)))
        title = 'Temperature - Lombardy 2020'
    if event.key == 'b' or event.key == 'B':
        meteo = np.dstack((np.zeros_like(wind), np.zeros_like(wind), wind))
        title = 'Wind speed - Lombardy 2020'
    if event.key == 'a' or event.key == 'A':
        meteo = np.dstack((rain, temp, wind))
        title = 'Lombardy 2020'
    if event.key in 'rRgGbBaA':
        fig = plt.figure(1)
        plt.imshow(meteo)
        plt.title(title)
        fig.canvas.draw()

# main
```

```
34 rain = normalize_matrix(np.load('PR_2020.npy'))
35 temp = normalize_matrix(np.load('TEMP2m_2020.npy'))
36 wind = normalize_matrix(np.load('VU_2020_VV_2020.npy'))
37
38 meteo = np.dstack((rain, temp, wind))
39
40 fig = plt.figure(1)
41 plt.imshow(meteo)
42 plt.title('Lombardy 2020')
43
44 cid = fig.canvas.mpl_connect('button_press_event', onclick)
45 cid = fig.canvas.mpl_connect('key_press_event', on_key)
46
47 plt.show()
```

The statements in lines 44 and 45 assign a callback function to the events `'button_press_event'` (which occurs when a mouse button has been clicked) and `'key_press_event'` (which occurs at a keystroke). In particular, the function `on_key` contains the code for reassign the channels of the image and the title of the figure and then redraw the figure.

Running the code, pressing `'g'` or `'G'` having the mouse on the plotted image, a grid appears (cycling through different setups). This is the effect of a default callback in `matplotlib`. It can be disabled putting the following code before invoking `plt.show()` (line 47):

```
for key in ('keymap.all_axes', 'keymap.grid', 'keymap.grid_minor'):
    plt.rcParams[key] = ''
```

Similarly, default bindings can be changed simply assigning a value to the `matplotlib.rcParams` entries.

Interactivity can be provided also using other functions such as `ginput` and `waitforbuttonpress`. More information and examples can be found in [6].

References

[1] "NumPy website." https://numpy.org/.

[2] "Matplotlib website." https://matplotlib.org/.

[3] "'array' or 'matrix'? Which should I use?." https://numpy.org/devdocs/user/numpy-for-matlab-users.html#array-or-matrix-which-should-i-use.

[4] I. Takacs, "Rainbow in Budapest." https://commons.wikimedia.org/wiki/File:Rainbow_in_Budapest.jpg.

[5] M. Calderini, "Plot organization in matplotlib — your one-stop guide." https://towardsdatascience.com/plot-organization-in-matplotlib-your-one-stop-guide-if-you-are-re 2019.

[6] "Event handling — matplotlib documentation." https://matplotlib.org/stable/gallery/event_handling/index.html.