NumPy

What is NumPy

NumPy is a scientific computation package

It offers many functions and utilities to work with N-Dimension arrays

Largely used by other libraries such as OpenCV, TensorFlow and PyTorch to deal with multi dimensional arrays (e.g., tensors or images)

How to install

We can easily install NumPy using pip by running

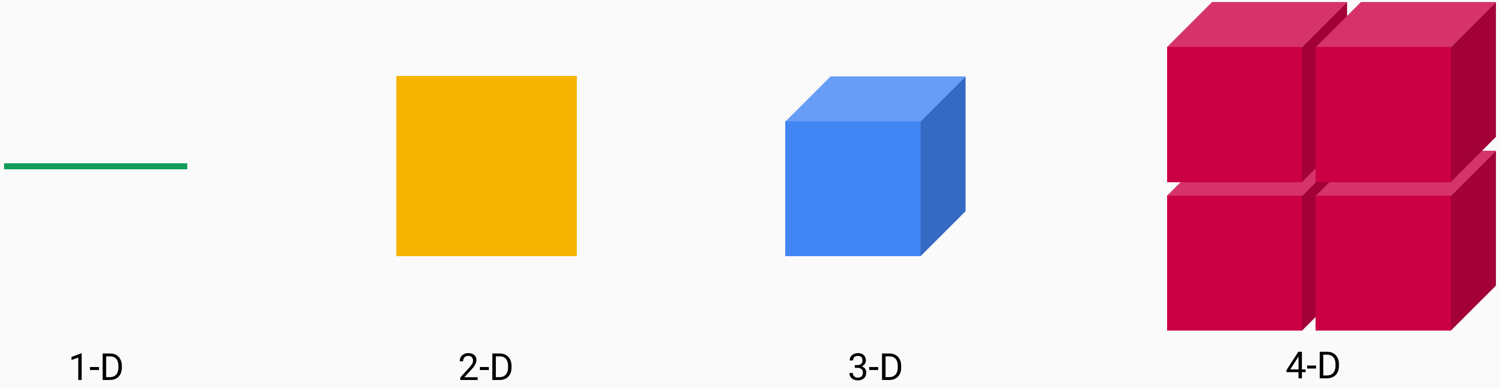
pip install numpy

Then, in our python script we have to import it

```
import numpy as np
```

N-Dimensional array

N-Dimensional array are, for instance, the followings:



NumPy array: how to create an array

In order to create an array, we can use the **array** function, passing a list of values and optionally the type of data

```
# create an array from a list of values
list_of_values = [20.,2.,5.]
x = np.array(list_of_values)

more_values = [[[20],[2],[5]]]
y = np.array(more_values, dtype=np.int32)
```

NOTE: NumPy arrays must be *homogeneous*, so each element must have the same type

NOTE: notice that if the type is not set, NumPy will decide the type for you. Default value for NumPy arrays is Float64

NumPy array: how to create an array

Moreover, NumPy offers standard functions to easily create arrays. Some of them are **ones**, **zeros**, **ones_like**, **zeros_like** and **eye**, but there are many more

We use the functions **zeros** and **ones** to create an array with given shapes in which each element is, respectively, 0 or 1

```
# create a int32 array of zeros with shape (20,2)
zeros = np.zeros((20,2), dtype=np.int32)

# create a float32 array of ones with shape (5,2,1)
ones = np.ones((5,2,1), dtype=np.float32)
```

NumPy array: how to create an array

Given a NumPy array with a certain shape, **zeros_like** and **ones_like** allow to create a 0 and 1 arrays with the same shape

```
x = np.array([5,5])
zeros = np.zeros_like(x) # [0 0]

ones = np.ones_like(x, dtype=np.float32) # [1. 1.]
```

NumPy array: how to create an array

Using the function **arange**(**start**, **stop**, **step**), we obtain a NumPy array containing all elements from **start** to **stop**, using a **step** spacing consecutive elements

```
# create an array containing [0,1,2,3,4]
x = np.arange(5)

# create an array containing [2,3,4]
y = np.arange(2,5)

# create an array containing [2,4]
z = np.arange(2,5,2)
```

Notice that the values are generated within the half-open [**start**, **stop**[, so **stop** is not included

NumPy array: how to create an array

With **eye** we create an *identity* matrix, so a matrix full of zeros except for the diagonal

```
x = np.eye(5)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Sometimes we need random values. We can obtain an array filled with random values calling the **rand** function

```
# create an random matrix 3x5
x = np.random.rand(3,5)
```

NumPy array: how to create an array

Finally, we can create an array that contains a single scalar value using **full**

```
# create an matrix 2x2 matrix in which each element is 7
x = np.full((2,2), 7)
```

We are able to obtain the same result using **ones**/**ones_like** function

```
# create an 2x2 matrix in which each element is 7
x = np.ones((2,2)) * 7
```

NumPy attributes

Each array has got **attributes**, such as **dtype** or **shape.** Attributes contain important information related to that particular array

dtype allows to know the type of the array

```
x = np.array([2., 5., 3.])
y = np.array([2., 5., 3.], dtype=np.int32)

print(x.dtype) # float64
print(y.dtype) # int32
```

NumPy attributes

shape give you back the size of the array along each dimension

```
x = np.array([2., 5., 3.])
y = np.ones((2,4,1,2,3))

print(x.shape) # (3,)
print(y.shape) # (2, 4, 1, 2, 3)
```

Changing the shape of arrays

Given an array, we can add a new dimension using **expand_dims**

```
x = np.full((2,2,3),7)
print(x.shape)
x = np.expand_dims(x, 0)
assert x.shape == (1,2,2,3)
x = np.expand_dims(x,-1)
assert x.shape == (1,2,2,3,1)
```

Changing the shape of arrays

A common operation consist in changing the shape of a given array. For instance, we can turn a 10 elements array into a 2x5 using the **reshape** function

```
x = np.arange((10)) # [0 1 2 3 4 5 6 7 8 9]
y = np.reshape(x, (2,5)) #[[0 1 2 3 4],[5 6 7 8 9]]
assert y.shape == (2,5)
```

We can “ask” NumPy to complete by himself the shape, using **-1**

```
x = np.arange((20))
y = np.reshape(x, (2,-1,2))
assert y.shape == (2,5,2)
```

Changing the shape of arrays

Notice that this operation is valid for just 1 dimension. In fact, If more dimensions are unknown, NumPy will throw **ValueError**

```
x = np.arange((20))
y = np.reshape(x, (2,-1,-1)) # ValueError: can only specify one unknown dimension
```

Changing the shape of arrays

Using **squeeze**, we can remove all the single dimensional entries of the array

```
x = np.full((20,1,1), 5)
y = np.squeeze(x)
assert y.shape == (20,)
```

However, squeeze allows also to specify the axis to delete (**scalar**, **tuple** or **None**. Default is **None**)

```
x = np.full((20,1,1), 5)
y = np.squeeze(x, axis=1)
assert y.shape == (20,1)
```

Elements of arrays

Given an array, you can access to its elements by index notation

```
# get the 10th element of the array
x = np.arange(20)
element = x[10] #10
```

Elements can be retrieved also using **item** function

```
# get the 10th element of the array
x = np.arange(20)
element = x.item(10) #10
```

Elements of arrays

Slice notation (the same used for python strings) is valid also for arrays

```
# get elements with index in [10,15[
x = np.arange(20)
element = x[10:15] #[10,11,12,13,14]

# get elements with index in [10: len(array)-7[
more_elements = x[10:-7] #[10 11 12]
assert np.array_equal(more_elements, x[10:13])

# get every element whose index is multiple of 3, starting from index 0
array = x[::3] #[0 3 6 9 12 15 18]
```

Elements of arrays

Given two array, we can concatenate them together to obtain a single array as output thanks to the **concatenate** function

```
x = np.full((5,2), 3)
y = np.full((5,1), 4)
z = np.concatenate([x,y], axis=-1)
print(z) #[[3 3 4],[3 3 4],[3 3 4],[3 3 4],[3 3 4]]
assert z.shape == (5,3)
```

NumPy math

Since NumPy is a scientific package that offers easy and even complex functions that you can apply to arrays.

In the following, we are going to see some of them

NumPy math: sum and subtraction

Given two arrays, you can sum or subtract them just using **+** and **-** operators

```
x = np.full((4,2,3), 8) # 4x2x3 array, full of 7
y = np.ones_like(x) # 4x2x3 array, full of 1

# sum two arrays
array_sum = x + y
assert np.array_equal(array_sum, np.ones_like(x)*9)

# subtract two arrays
array_sub = x - y
assert np.array_equal(array_sub, np.ones_like(x)*7)
```

In this case, both arrays have the same shape, so the operations are performed **element-wise**

NumPy math: broadcasting

Sometimes, our arrays have not the same shape, but NumPy is smart enough and try to “fit” the arrays. This operation is called **broadcasting**

```
x = np.full((4,2,3), 8) # 4x2x3 array, full of 7
y = 1

# sum two arrays
array_sum = x + y
assert np.array_equal(array_sum, np.ones_like(x)*9)

# subtract two arrays
array_sub = x - y
assert np.array_equal(array_sub, np.ones_like(x)*7)
```

Notice that **y** is a scalar, but both **array_sum** and **array_sub** have shapes (4,2,3)

NumPy math: broadcasting

Broadcasting can't work for all the cases: when operating on two arrays, NumPy looks at their shapes. The shapes are compatible if, in the element-wise comparison, they are equals or one dimension is 1. The resulting shape is the maximum shape along each dimension.

If broadcasting can't be applied, **ValueError** would be raised

```
x = np.full((4,2,3), 8)
y = np.full((4,3),3)

z = x + y # ValueError: operands could not be broadcast together with shapes (4,2,3) (4,3)

y2 = np.full((4),3)
z = x + y2  # ValueError: operands could not be broadcast together with shapes (4,2,3) (4,3)

y3 = np.ones(4) # shape is (4,)
y3 = np.expand_dims(1,0) # shape is (4,1)
z = x + y3 # it works!
```

NumPy math: multiplication

Given two NumPy arrays, we can perform **element-wise** multiplication using * or **multiply** function

```
x = np.full((4,2,3), 8) # 4x2x3 array, full of 8
y = np.ones_like(x)*2 # 4x2x3 array, full of 2

# multiplicate element-wise two arrays
mul = x * y
assert np.array_equal(mul, np.ones_like(x)*16)

# subtract two arrays
mul2 = np.multiply(x, y)
assert np.array_equal(mul2, np.ones_like(x)*16)
```

NumPy math: matrix multiplication

Given two NumPy arrays, we can perform **matrix** multiplication using **matmul** function

```
x1 = np.full((4,2,3), 8) # 4x2x3 array, full of 8
x2 = np.full((3,3), 7) # 3x3 array, full of 7
y = np.eye(3) # 3x3 diagonal array

# matrix multiplication of two arrays
mul = np.matmul(x1,y)
assert np.array_equal(mul, x1)

# matrix multiplication
mul = np.matmul(x2,y)
assert np.array_equal(mul, x2)
```

For N-Dimensional arrays (N>2), **matmul** applies broadcasting, treating the array as a stack of matrices

Conditions

We can use **where** function to apply a condition.Given an input array, a condition and two arrays x and y, **for each element** we sample from x if the condition is verified, from y otherwise

```
x = np.arange(5)
y = np.where( x < 2, 0, 255) [0,0,255,255,255]
```

Conditions

Notice that NumPy is quite optimized, so when possible try to used “**native**” NumPy way instead of other approaches

```
x = np.random.rand(4,640,480,3)
batch, height, width, channels = x.shape
start = time()
y = np.ones_like(x)
for b in range(batch):
    for h in range(height):
        for w in range(width):
            for c in range(channels):
                y[b,h,w,c] = 0 if x[b,h,w,c] < 0.05 else 255
duration = time() - start # 4.982 sec
```

```
x = np.random.rand(4,640,480,3)
start = time()
y = np.where( x < 0.05, 0, 255)
duration = time() - start # 0.0283 sec
```

NumPy Input/Output

NumPy provides a set of functions to write and read directly from the filesystem

Plain text (.txt) and csv (.csv) can be loaded using **loadtxt** function, providing the path to the file, the data type and the delimiter

```
data = np.loadtxt('your_file.txt',dtype=np.float32, delimiter=',')
```

NumPy Input/Output

We can store NumPy arrays in two ways:

- binary file: using **np.save** we are able to serialize our arrays in the local file system. We will obtain a **.npy** file containing the array
- txt: using **savetxt** function we will store our 1D or 2D array in a new txt

Finally, **.npy** files can be read using **np.load** function

Plots

Sometimes, we want to display values in a human readable form. Suppose for instance you have got temperature values collected by a sensor, one measurement per hour for a week. Display such values in a chart would simplify largely the reading!

We can visualize NumPy arrays using some chart libraries, like Matplotlib

Matplotlib

[Matplotlib](#) is an open source plotting library able to produce high quality graphs and charts. Easy to install using pip (just “*pip install matplotlib*”)

It offers a large set of plot types (e.g., histogram, scatter, line, 3D and more), and uses NumPy arrays to handle data

It can run also on interactive environments such as Jupyter

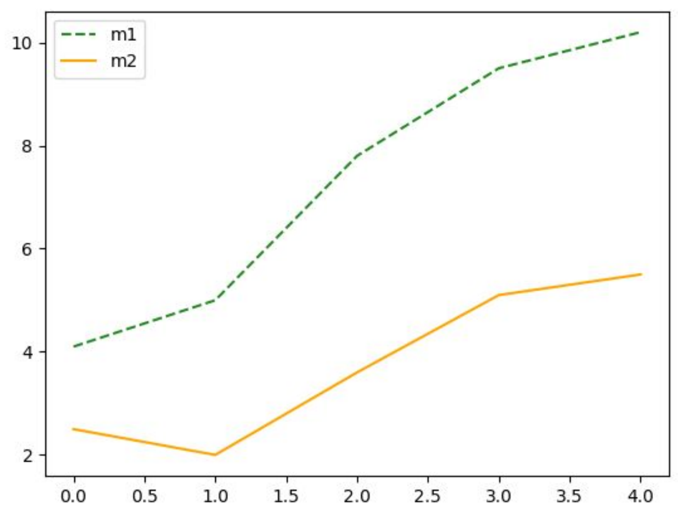
Matplotlib

Given two collection of values, **m1** and **m2**, we can visualize them using Matplotlib

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

labels = ['7', '8', '9', '10', '11']
x = np.arange(5)
m1 = np.array([4.1,5.0,7.8,9.5,10.2])
m2 = np.array([2.5,2.,3.6,5.1,5.5])

plt.plot(x,m1, color='forestgreen',label='m1', linestyle='dashed')
plt.plot(x,m2, color='orange', label='m2')
plt.legend()
plt.savefig('chart.png')
```



OpenCV

OpenCV

OpenCV is an open source Computer Computer Vision library. It allows to develop complex Computer Vision and Machine Learning applications fast, offering a wide set of functions.

Originally developed in C/C++, now OpenCV has handlers also for Java and Python

it can be exploited also in iOS and Android apps.

In Python, OpenCV and NumPy are strictly related

OpenCV

In particular, some of the functions offered by OpenCV are:

- Image Handling (read an image, write an image etc)
- Corner Detection (Harris, Shi-Tomasi etc)
- Camera Calibration
- Features Detection and Description (ORB, SIFT, SURF etc)
- K-Nearest Neighbour
- Depth estimation (Block Matching, SGM etc)
- Optical Flow (Lucas-Kanade)

and many more

OpenCV: installation

We can install OpenCV directly by pip, calling

pip install opencv-python

Then, in our Python script, we can import it as follows:

```python
import cv2
```

OpenCV: Image Handling

As we said, OpenCV offers functions to read and write images.

We can open an image using the **imread** function:

```
img = cv2.imread('img_path')
```

Moreover, it is able to handle various image format (png, jpeg etc) and data types (8bit, 16 bit etc)

OpenCV: Image Handling

Once opened, OpenCV **returns a numpy array** that stores the image (each value of the array is a pixel)

OpenCV default format is **BGR**, so we have to swap the first and the last channels in order to manage a **RGB** image. We can do it manually or invoking the cvtColor function

```
img = cv2.imread('path_to_your_image')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

cvtColor helps in converting colored images (**BGR** or **RGB**) to grayscale just using as options **cv2.BGR2GRAY** or **cv2.RGB2GRAY**

OpenCV: Image Handling

(A)

(B)

In figure (A), the original **RGB** image, while in figure (B) the same picture saved using **BGR** format

OpenCV: Image Handling

(A)

(B)

We can use cv.cvtColor function also to obtain grayscale images

```
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

OpenCV: Image Handling

Instead, we can write an image in our file system using the **imwrite** function

```
cv2.imwrite('saving_path', img)
```

However, remember that OpenCV expects a **BGR** image, so if img is a **RGB** you must convert to BGR using cv2.cvtColor

```
bgr_img = cv2.cvtColor(rgb_img, cv2.COLOR_RGB2BGR)
```

OpenCV: Image Handling

OpenCV allows to resize images using the **resize** function. It takes the image and the new shape

```
img = cv2.imread('image/img.png')
resized_img = cv2.resize(img, (320, 240))
```

(A) Original image

(B) Image resized at 320x240

OpenCV: filters

2D Convolution in OpenCV is straightforward: you have just to call the **filter** function, passing as input the image and the kernel

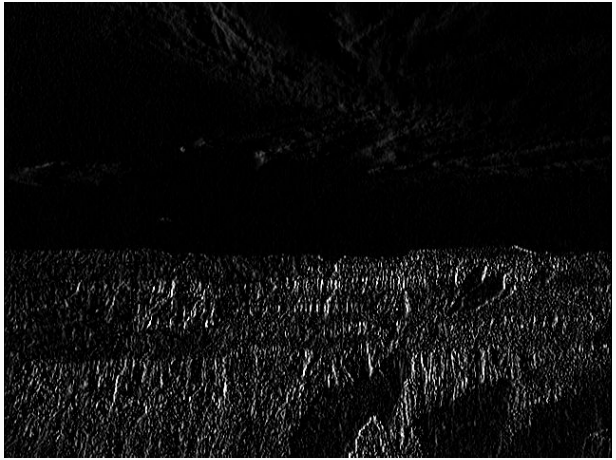
```
kernel = np.ones((9,9),np.float32)/81
dst = cv2.filter2D(img,-1,kernel)
```

OpenCV: filters

Changing the filter, we would obtain different results. For instance, **high-pass** filter can be obtained through a zero-sum kernel

```
kernel = np.array([[-1,0,1], [-2,0,2], [-1,0,1]], dtype=np.float32)
dst = cv2.filter2D(img,-1,kernel)
```

-1	0	1
-2	0	2
-1	0	1

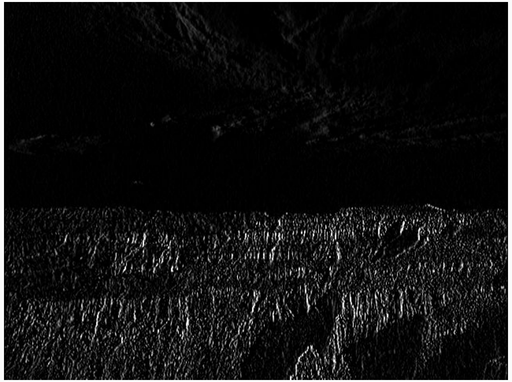


OpenCV: filters

We can obtain the same result* using **separable** filters

```
kernel1 = np.array([[-1,0,1]], dtype=np.float32)
kernel2 = np.array([[1],[2],[1]], dtype=np.float32)
dst = cv2.filter2D(img,-1, kernel1)
dst = cv2.filter2D(dst,-1, kernel2)
```

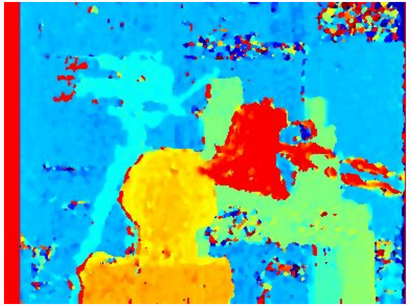
$$* \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$



Example: Stereo Matching

```
left = cv2.cvtColor(cv2.imread('image/left.png'),cv2.COLOR_BGR2GRAY)
right = cv2.cvtColor(cv2.imread('image/right.png'), cv2.COLOR_BGR2GRAY)
stereo_matcher = cv2.StereoSGBM_create(numDisparities = 16, blockSize = 5)
disparity = stereo_matcher.compute(left, right)
disparity = (disparity).astype(np.uint8)
disparity = cv2.applyColorMap(disparity, cv2.COLORMAP_JET)
cv2.imwrite('./disparity.png',disparity)
```

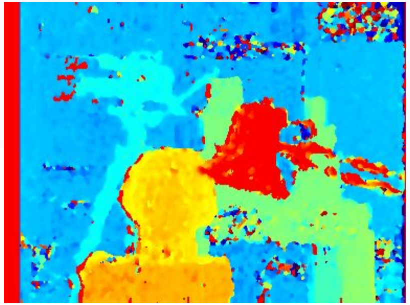
from Middlebury Dataset



FAR  CLOSE

Example: 3D reconstruction

point cloud can be visualized using MeshLab



from Middlebury Dataset

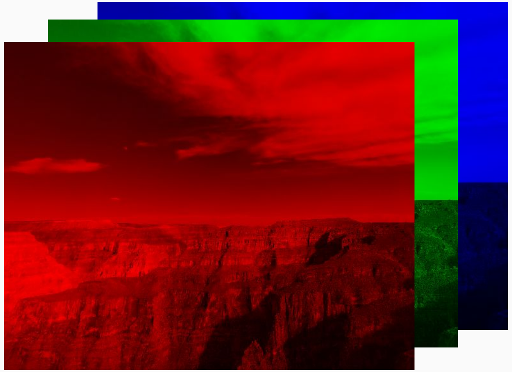
```
img = cv2.imread('image/left.png')
left = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
right = cv2.cvtColor(cv2.imread('image/right.png'), cv2.COLOR_BGR2GRAY)
stereo_matcher = cv2.StereoSGBM_create(numDisparities = 16, blockSize = 5)
disparity = stereo_matcher.compute(left, right)
focal_length=1.
ppm =  np.float32([[1,0,0,0],[0,-1,0,0], [0,0,focal_length,0],[0,0,0,1]])
points_3D = cv2.reprojectImageTo3D(disparity, ppm)
colors = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
generate_pointcloud(colors,points_3D,'3D.ply')
```

Exercises

Exercise 1

Given the image *canyon.png* load it using OpenCV, split the channels and save each channel in a new image called as the channel.

For instance, the red channel have to be saved as *red.png*



Exercise 2

Using the same image of the previous exercise, load it as gray-scale and replace all pixels with intensity lower than 80 with 0, 1 otherwise. Save it both as a new image, called *mask.png*, and as npy. Finally, apply the mask to the original image, keeping the original value where the mask is 1, 0 otherwise

Exercise 3

Using Matplotlib, display intensity values of *canyon.png* (loaded as grayscale image) in a bar chart. In particular, for each intensity value, the height of the column is the number of pixels that have such intensity value