# additional materials for SET 3

kacper.topolnicki@uj.edu.pl

The `set_3.py` script contains additional materials for the second set of exercises. This set is related mainly to the `numpy` library and in this tutorial I used some of the materials available in the official documentation:

- https://numpy.org/doc/stable/user/quickstart.html
- https://numpy.org/doc/stable/
- https://docs.opencv.org/master/

## Importing the necessary libraries

Just as last time we will begin by importing `numpy` [set_3.py line: 21] :

```
import numpy
```

This library has become the de-facto standard for numerical computations in python. The `numpy` array is used as the basic array type in many other libraries. In order to demonstrate this we will import the computer vision library *OpenCV* [set_3.py line: 24] :

```
import cv2
```

You can install this library from `pip` (`pip install cv2`). There is a standard python package, `cv2`, but it might not have all the fun methods of the full `c++` version. For this reason you might try to install the experimental `opencv-contrib-python` package that contains a more complete set of functions.

## Reading the sample image

The `zip` archive available on our web page contains the python script and a sample image `sample_image.png`. We will read this image and later point to it using the `image` variable [set_3.py line: 60] :

```
image = cv2.imread("./sample_image.png")
```

If you are using *ipython* then you can read the documentation for `cv2.imread` by using the `?` operator. Let's inspect the result. First we can ask python to print out the type: The result should be `<class 'numpy.ndarray'>`, indicating that we are dealing with a `numpy.ndarray` object. This is the basic type used in `numpy`.

The arrays used in `numpy` have a fixed number of elements and dimensions (referred to as axes). Further more each element of the array has the same type. This is a very useful assumption to have if you are going to be mapping the same function over each element of the array.

Some properties of the array are accessible directly through object variables. Let's start with the shape of the array [set_3.py line: 98] :

```
print("image.shape : " , image.shape)
```

The result, printed to the screen, is `(720, 1280, 3)` which means that we are dealing with an array that has three axes (dimensions). We can easily deduce that the first axes, the one that has 720 elements, corresponds to the rows of the image and that the second axes, the one that has 1280 elements, corresponds to the columns of the image. The third dimension (axes) of the image has only three elements. It is easy to guess that these three elements correspond to the three color channels of the image (blue, green and red).

What about the type of the elements in `image`. This is accessible through the `dtype` property [set_3.py line: 112] :

```
print("image.dtype : " , image.dtype)
```

The result `uint8` means that we are dealing with unsigned, 8 bit integers - a sensible choice to store 256 color values in each of the three color channels.

Displaying the image

In order to display the image we will be using the `cv2.imshow` function [set_3.py line: 130] :

```
cv2.imshow('image' , image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

This is immediately followed by `cv2.waitKey` which waits for the user to pres any key (the `0` argument means that the library will wait indefinitely). Finally after a key press has been registered all windows are closed using `cv2.destroyAllWindows`.

De-noising

What is meant by de-noising here is background removal. We would like to be left with nothing but the text and the pen outline on a white backdrop. Esthetic's aside, compression algorithms will do a much better job reducing the size of the de-noised image, compared to the original image.

After some experimentation I think that I finally found a good recipe to do this. It involves some statistics. Here it is ...

The first step is to convert the image to gray scale using the `cv2.cvtColor` function [set_3.py line: 167] :

```
gray = cv2.cvtColor(image , cv2.COLOR_BGR2GRAY)
```

Next, the gray scale image is inverted. We are mostly interested in the dark pixels (they contain the text) and this way dark pixels will have a high value associated with them [set_3.py line: 170] :

```
gray = 255 - gray
```

In the language of high energy physics experiments, the dark pixels (now having large values associated with them) will be treated as the "signal" and the light pixels (now having small values associated with them) will be treated as the background.

Next we will remove a 50 pixel white border around the image. Removing is done by replacing the border with the background value 0, My experiments show that this is also beneficial in that it takes care of any distortions on the image boundary [set_3.py line: 194] :

```
border = 5
gray[0:border , :] = 0
gray[gray.shape[0] - border : gray.shape[0] , :] = 0
gray[: , 0:border] = 0
gray[: , gray.shape[1] - border : gray.shape[1]] = 0
```

Notice that you can reference segments of the array (`0:border`) and assign the same value to a whole segment using a single command. Also notice that array indexing works slightly different then for standard python arrays. Please try these commands in *ipython* and modify them to get a better understanding. The result can be displayed by using the three familiar commands [set_3.py line: 202] :

```
cv2.imshow('image' , gray)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Now, we will convert each element of the gray scale image into a floating point number [set_3.py line: 219] :

```
gray = gray.astype("float32")
```

We will be using this array in some statistical calculations and floating point numbers are more suitable. The argument `"float64"` can alternatively be replaced by `numpy.float64`. Both alternatives will have the same effect and the `astype` method will turn `gray` into an array of 64 bit floating point numbers. Ok, now let's take care of the uneven lighting. We will first construct a **convolution matrix** that will be used to construct an estimate of lighting conditions in the image [set_3.py line: 244] :

```
kernelSize = 300
blur_kernel = numpy.ones((kernelSize , kernelSize) ,
    dtype = numpy.float64)
```

The `numpy.ones` function, just as the name suggests, constructs an array filled with 1.0. In our case we will have a 300 by 300 matrix filled with 64 bit floating point - ones. We will be using the matrix convolution operation with `blur_kernel` in order to calculate the average pixel value in a 300 by 300 pixel wide square surrounding a pixel. In order to calculate the average we need to normalize the `blur_kernel` and divide it by the sum of elements inside [set_3.py line: 249] :

```
blur_kernel = blur_kernel / numpy.sum(blur_kernel.flatten())
```

Where `flatten` turns a multidimensional array into a 1D list and `numpy.sum` calculates the sum of elements in the list (to be fair the same can be achived by simply dividing by `kernelSize * kernelSize`). Kernel convolution is available in the `cv2.filter2D` function (see the documentation using `?cv2.filter2D` in *ipython*). and the lighting conditions are obtained in [set_3.py line: 261] :

```
blured_gray = cv2.filter2D(gray , -1 , blur_kernel)
cv2.imshow('image' , blured_gray.astype("uint8"))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Now comes the statistical part. We will calculate the standard deviation (are we going to be using a decent estimator?) of each pixel value from the mean [set_3.py line: 280] :

```
stdv = numpy.sqrt(
    numpy.mean(
        ((gray - blured_gray) * (gray - blured_gray)).flatten()))
```

Where we us a combination of `flatten`, `numpy.mean` and `numpy.sqrt`. Please break this line into smaller pieces and try them out in *ipython*. What is the shape of `gray - blured_gray`, `(gray - blured_gray) * (gray - blured_gray)`, `((gray - blured_gray) * (gray - blured_gray)).flatten()`. Notice that some operations are automatically mapped over all elements of the arrays.

We will classify all pixels above one standard deviation from the mean as the text and other pixels as the background (is this the best approach?). But first some magic. We will be converting our our original image from the BGR (blue, green, red) color space to **HLS** [set_3.py line: 306] :

```
hls = cv2.cvtColor(image , cv2.COLOR_BGR2HLS)
```

and we will only modify the L channel. This way we retain the original color of the pixel and only change the perceived brightness. We set the initial value of the L channel to 255 [set_3.py line: 309] :

```
l_res = 255.0 * numpy.ones(
                    shape = gray.shape ,
                    dtype = numpy.float64)
```

using `numpy.ones`. Next, using `numpy.where`, we replace all values that are one standard deviation (`stdv`) above the mean (`blured_gray`) with the actual value of the L channel (`hls[: , : , 1]`) [set_3.py line: 314] :

```
l_res = numpy.where((gray - blured_gray) > stdv ,
            hls[: , : , 1] ,
            l_res)
```

The final step is to put the three H, L, S channels back togather using `cv2.merge`, convert the HLS image back to BGR and display it [set_3.py line: 328] :

```
h_res = hls[: , : , 0]
s_res = hls[: , : , 2]
result = cv2.cvtColor(
            cv2.merge(
                (
                    h_res.astype("uint8") ,
                    l_res.astype("uint8") ,
                    s_res.astype("uint8"))
                )
                , cv2.COLOR_HLS2BGR)
cv2.imshow('image' , result.astype("uint8"))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

That's it. Pleas try to run the commands from this tutorial in *ipython*. You can dissect the more complex pieces of code into smaller bits ans see how they work. The `numpy` library is a very useful and popular tool so You are additionally encouraged to read the documentation and tutorials that are available online for this library.