1. Image enhancement and reviewing Python/NumPy

Abstract
In this first lab, we will learn to modify the gray levels of an image with the main goal of enhancing its visual quality. This enhancement can of interest for human perception or for subsequent computational processes. We will also review Python programming and libraries such as NumPy, that are commonly used in scientific contexts, and in computer vision in particular, since we will be using them throughout this course.

Palabras clave
Brighten and darken images • Histogram equalization • Python • NumPy

Contents

1. Introduction

We assume a general and basic background in Python. Therefore, we will only mention some concepts that are good to review and take into account, mostly regarding image and matrix computations, using NumPy in particular. At least some parts of Array programming with NumPy may be useful.

Fundamentals. Among the most basic things to know, we need to create and initialise arrays, finding their sizes and performing matrix operations, both elementwise and matrix ones. For instance, if `a` and `b` are two 2D arrays (matrices) of the same size, what are we computing by writing `a*b`. Is it the matrix product or the elementwise product? Following some tutorial (example) will help us in these first and important steps. For loading and saving images we will use Python Imaging Library (PIL). Some introductory tutorial or short video may come handy. Note that we will not cover OpenCV.

Vectorization. Generally speaking, we must avoid writing explicit loops as much as possible, for two reasons: readability and efficiency. Regarding readability, a code without loops tends to have fewer lines of code. Regarding efficiency, vectorized versions of the code run significantly faster (speed up rates can be of one or even more orders of magnitude). How can we vectorize? Basically, by using appropriate syntax structures, and predefined (NumPy) functions that are vectorized and optimized (see example). Besides the code, the problem itself might be vectorized (see Problem vectorization), but this an advanced concept that we are not considering in our course. By now, it suffices that you get used to avoiding writing loops (the most obvious ones at least), and look for alternative vector forms of your code.

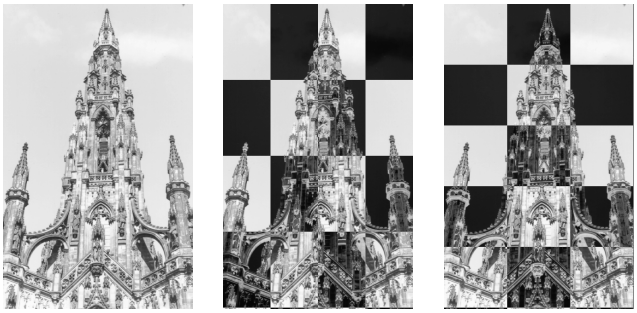
Indexing and slicing. It is an essential skill to know how to access elements or blocks of elements within an array of two or more dimensions. Somehow related to this, many NumPy functions include a parameter 'axis' to indicate the dimension the operation has to be performed on. Although this is an optional parameter and the default value may be good for the intended operation, it is not always the case. For instance, one may end up applying a function over the 1D flattened array when this is not what one expects. Therefore, it is important to know or read carefully the documentation to make sure the operation works as intended. Performing interactive tests on the Python console, before including the code in the Python program we are developing, is usually very useful.

Broadcasting. In the NumPy parlance, broadcasting refers to the treatment NymPy does of the arrays taking part in expressions so that operations involving arrays of different sizes are possible. For instance, can we sum an array of 10×8 with another of size 8×1? Or, how can we sum a constant to all elements of a matrix? It is important to be aware of this concept so that we can use it as we wish, or understand when it is being used, maybe unintentionally. Broadcasting is a vectorization form that in exchange of the temporal speed-up, it may require more memory, but this is usually less of a concern unless we are working with many arrays at the same time, or they are huge. Some clarifying schematics about broadcasting or an explanatory video can be useful.

Copying variables A possible source of problems or confusion has to do with assignments that do not correspond to actual copies, but to references. For instance, to copy a NumPy array should be done explicitly (copy). A review of assignment statements in Python may help clarify some related aspects.

2. Brightening and darkening images

The basic functions to lighten and darken images are simple. Compare their math expression with their implementation in the provided code. Check these functions by comparing both



(a) (b) (c)

Figure 1. Visualisation of the checkboard pattern (Exercise 4) applied to the original image (a) with 4×4 (b) y 5×3 (c) cells

the images and their histograms before and after applying them.

3. Histogram equalization

To better understand histogram equalization and know how to implement it in Python, we are writing our own version of this operation. Please, study and check the provided code. As before, visualise the images and their histograms before and after the equalization, and make sure you understand the effect of histogram equalization.

4. Exercises

The main goal of the following exercises is to become more fluent in Python, mostly regarding image and array manipulation with NumPy. Yo do not necessarily have to do all the exercises; just reflect on how much practice you need to get the skills required to solve, with moderate effort, exercises of this type and difficulty level.

1. Summarise the concepts and Python (and NumPy) functions used in this lab that are (more) novel for you. The purpose is that this helps you understand and internalise them, and it serves as a later reference as well. Try to organise this summary by category (e.g., NumPy, PIL, Matplotlib), and do it as much visual and quick to consult as possible.

2. In the provided code, we save an image to disk using several lines of code. Replace those lines by a single one with the same functionality by defining and calling a function `saveImg()`. Decide which parameters this function should have.

3. Generalize the brightening function, `brightenImg()` so that it works both with gray-level and color images. You can treat each color band as if it was a single gray-level image. Do not forget to display the resulting image and saving it to disk. You may try this function on images that are previously darkened with `darkenImg()`, which you may also generalise.

4. Write a function `checkBoardImg(im,m,n)` which, given a gray-level image `im`, it creates an image of the same size and same contents, but which inverts the pixels or not, alternatively, following a checkboard pattern where the image is partitioned into $m \times n$ cells, as illustrated in Fig. 1.

   Then check for the proper operation of this function with the assistance of an additional function, `testCheckBoard()` that calls `checkBoardImg()` for one of the available images and for some particular number of cells. You can use the provided `showInGrid()`, to display the results. Reuse the function `saveImg()` to save the resulting image.

5. It is often convenient to process or represent images in a multi-level tree-like fashion. For instance, the first level considers the full image; the second level considers the four subimages corresponding to partitioning the original image into four quadrants, and so on until reaching a given number of levels.

   Write a function `multiHist(im,n)` that returns a list of gray-level histograms of image `im` corresponding to n levels. Thus, if $n = 1$, the returned list will consist of a single histogram; if $n = 2$, the list will have 5 (1+4) histograms; if $n = 3$, it will have 21 (1+4+16) histograms, etc.

   Assume the input image is a 2D NumPy array (not an `Image` object from PIL module). Again, you can use our `showInGrid()` function to help you develop or test the function.

   As an example, with $n = 2$ and using 3-bin histograms (to simplify the console output), the result of our implementation of `multiHist()` on the image in Fig. 1a is

```
[array([ 25837,  53944, 193375]),
 array([ 1811,  3782, 62593]),
 array([ 3835,  7701, 56650]),
 array([ 9006, 19530, 39856]),
 array([10921, 21955, 35516])]
```

6. Write a function `expTransf(alpha,n,l0,l1,bInc)` that generates a gray-level transformation function $T(l) = a \cdot e^{-\alpha l^2} + b$ for n input values in the range $[l_0, l_1]$. The output should be a 1D array of n gray-level values, in the same range, increasingly or decreasing depending on the value of the boolean `bInc`. Set a and b to guarantee the range. Notice that you can compute the output assuming `bInc=True`, and just invert it if `bInc=False`. Display these functions for different gray level ranges and $\alpha > 0$ values. Numpy's linspace() function can be handy here. Then, apply these transformations to input images `im` via another function `transfImage(im,f)` for different `f` functions generated with `expTransf()`.