IMAGE PROCESSING TECHNIQUES USING PYTHON

Image Processing and Computer Vision

Course notes and exercises
Academic year 2019-20

Artur Carnicer, artur.carnicer@ub.edu
Departament de Física Aplicada
Universitat de Barcelona

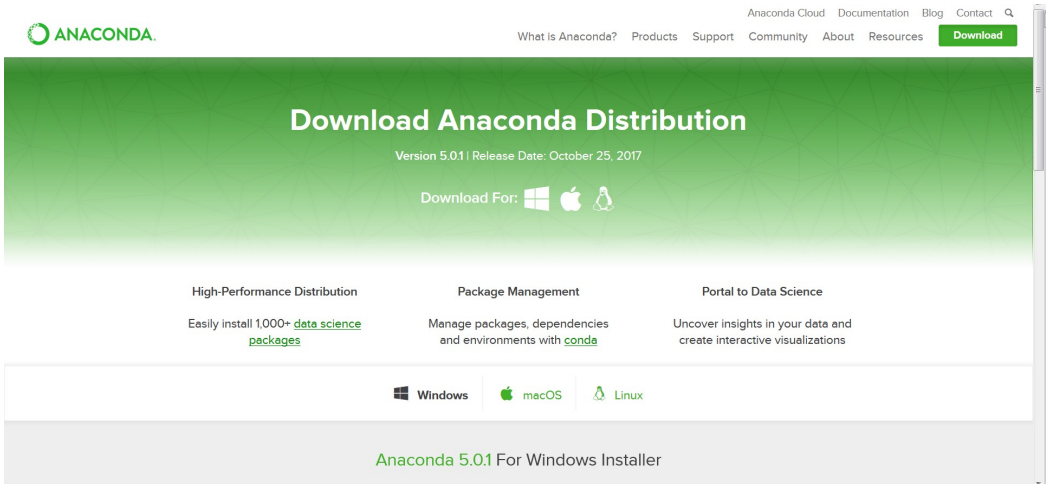
Updated May 22th, 2020

Table of contents.

Lab #1: Python concepts for image processing.

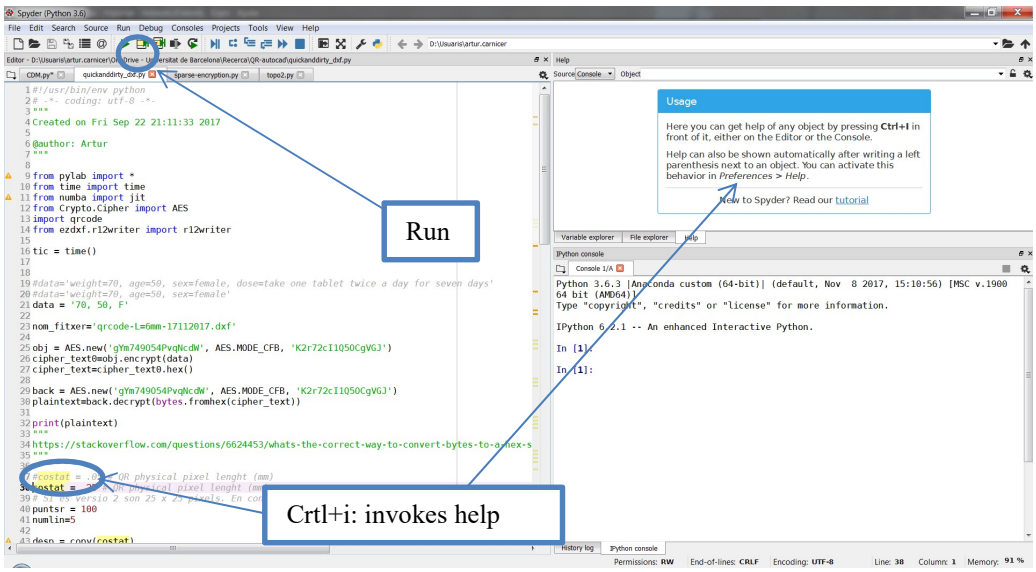
1.1. Download and install Python 3 + Spyder from the Anaconda website.

https://www.anaconda.com/download/



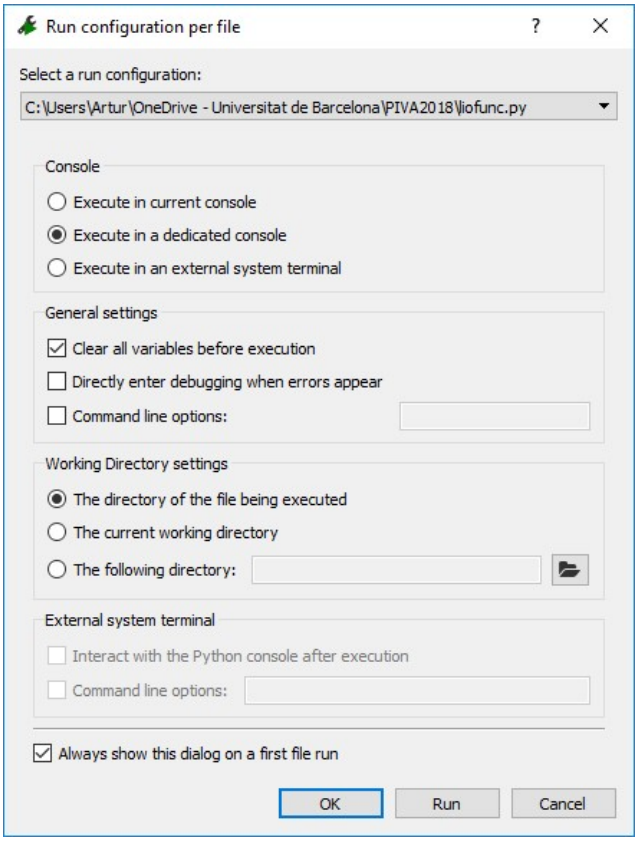
The browser should detect the OS on your computer; if it does not, select the system by clicking on the corresponding icon. In general, you should download the 64-bit version; only old computer with less than 2 MB of memory (RAM) require the 32-bit version.
For Mac OS users: you can download either the graphical or the command line installer; the former is more user-friendly.
Linux users: Select the x86 version if your computer is Intel or AMD-based.
If an old Python distribution is present on your computer, it is advisable to remove it (unless you know how to manage virtual environments).
Please select the Python 3 flavor. Python 2 is going to be discontinued soon.

1.2. Familiarize yourself with the editor button and panes.

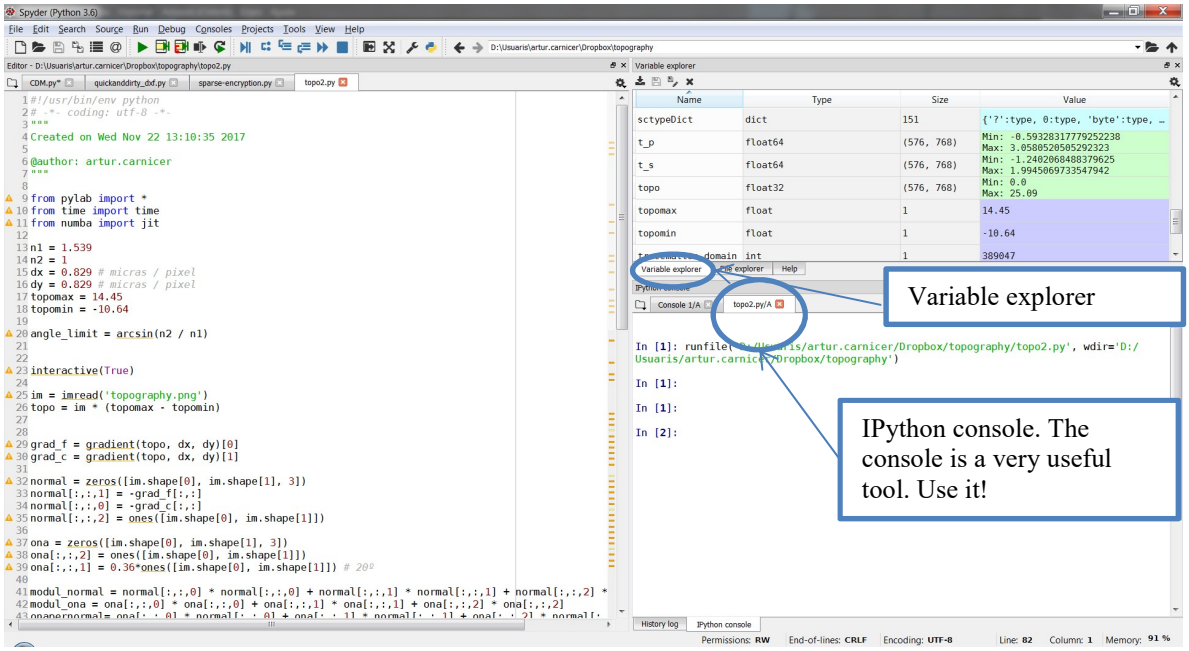


Save your script before the first run. Please, do not use directory or script filenames that contain spaces, or language-specific symbols such as á, è, ñ, ź, ę, etc.

The *Run Configuration per file* dialog appears the first time the code runs (or by selecting Run -> Configuration per file). Select the following options (why? –because you should understand what they mean):

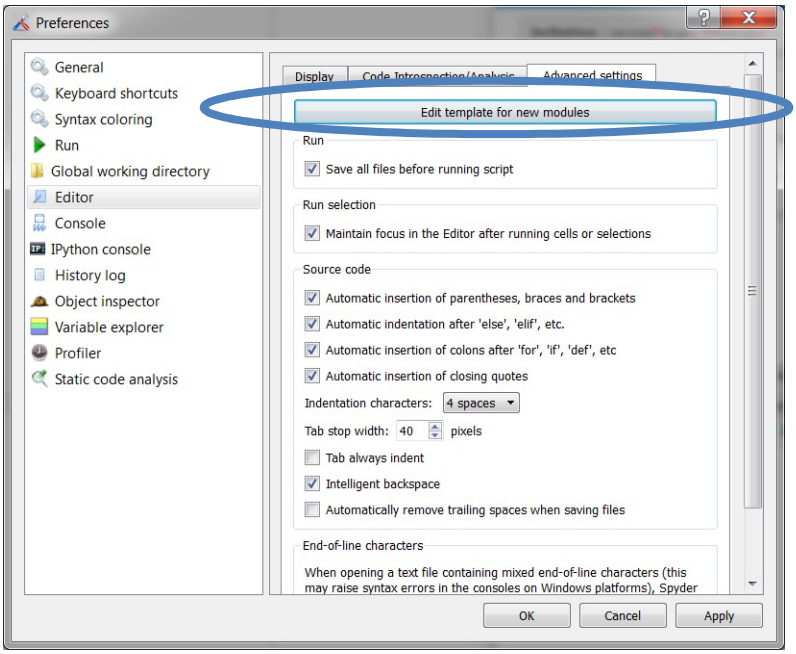


After execution, the variable explorer tab and the console are available for inspection.

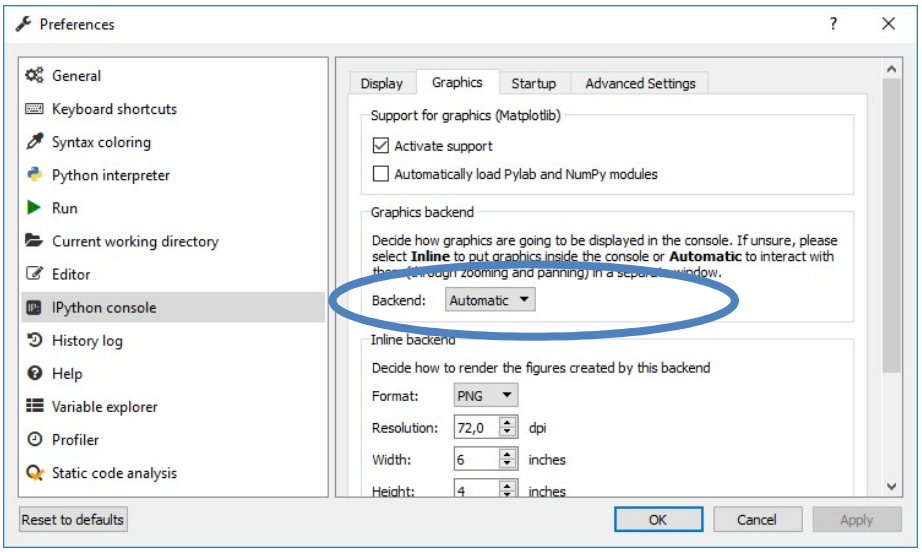


It is advisable to edit the template.py script.
Tools > Preferences > Editor > Advances > Edit template for new modules

We will now consider what to include here. From time to time you might want to modify the template script.



Also, it is handy to display images in a window instead of the IPython interface. In this case, select the 'Automatic' backend from the IPython Console > Graphics (see figure).



For MATLAB fans forced to use Python: please consider the information on the following page: https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

1.3. What you are expected to know.

Python is huge. However, you are only expected to know some parts of it:

- Numbers, strings, lists, tuples
- for and while loops
- functions
- input/output functions
- import modules

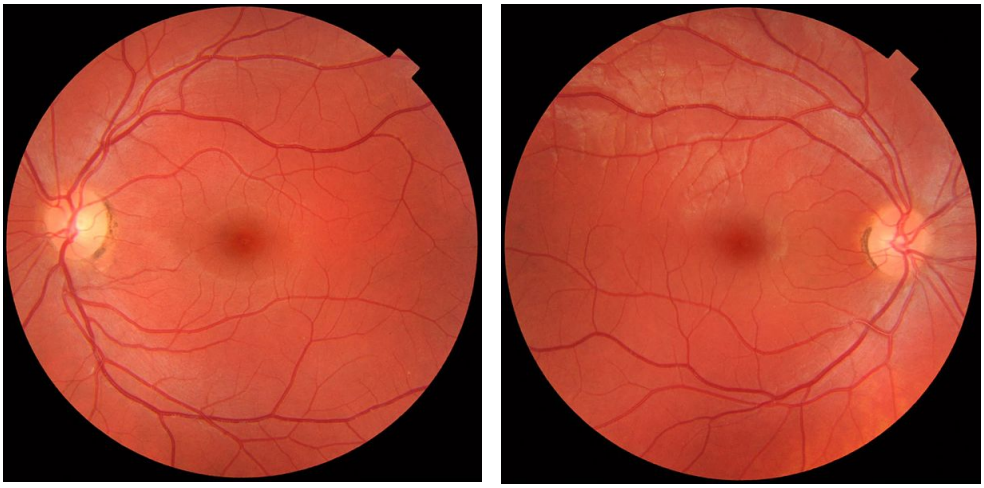
- how to handle arrays in numpy
- matplolib: figures, plot, images

You might want to have a look at the Python tutorial https://docs.python.org/3/tutorial/ sections 3 to 6 and the Numpy Quickstart tutorial https://docs.scipy.org/doc/numpy-1.15.0/user/quickstart.html.

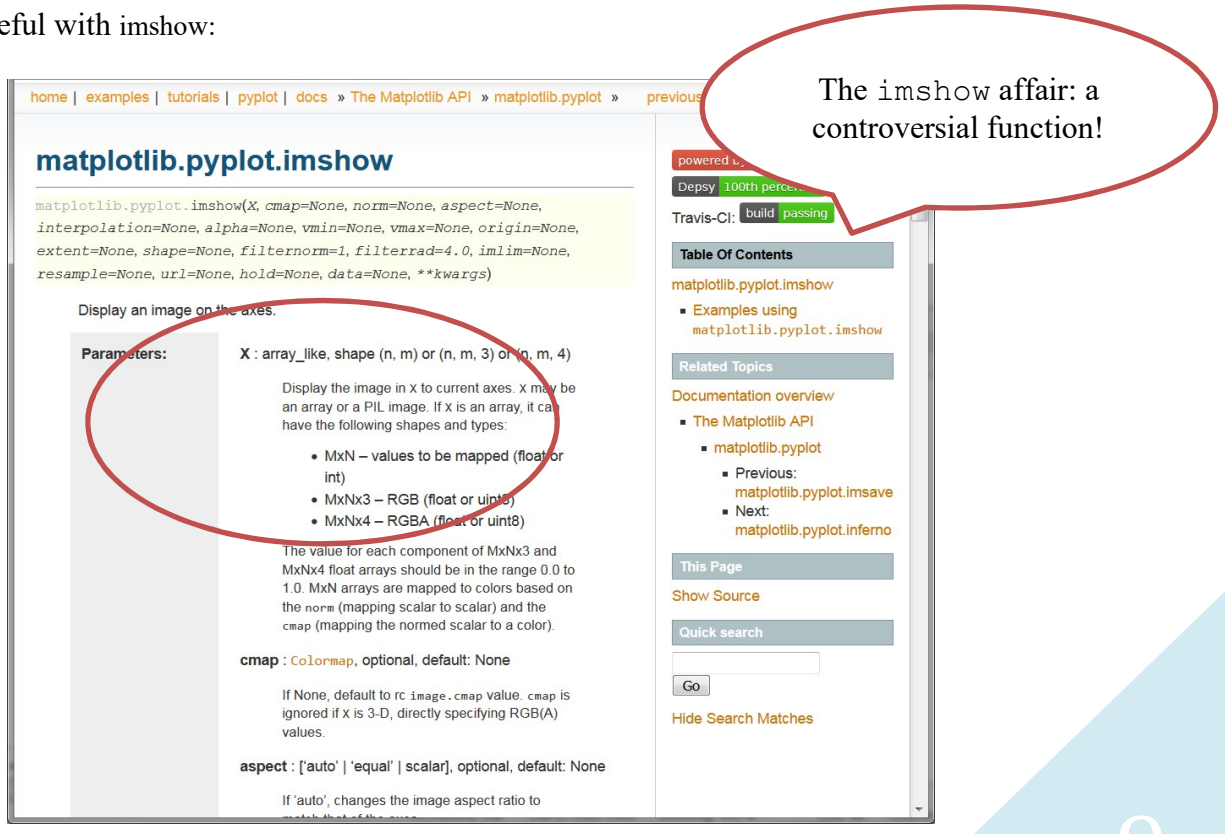
Lab #2: Basic image manipulation: channel processing, colormaps and cameras.

Opening remarks:

- Functions that you may need in this lab: `imshow, imread, imsave, savefig, figure, subplot`. These functions are part of the `matplotlib.pyplot` module.
- Note that a gray-level image is an MxN array of 8-bit unsigned integers (or unnormalized double values). An RGB image is an MxMx3 array of 8-bit unsigned integers (or double values ranging from 0 to 1). In addition, an RGBα image is described by an MxMx4 array (α stands for transparency).
- Recall how the ':' operator works
- The examples presented below are calculated using the following images:
  http://commons.wikimedia.org/wiki/File:Fundus_photograph_of_normal_right_eye.jpg
  http://commons.wikimedia.org/wiki/File:Fundus_photograph_of_normal_left_eye.jpg



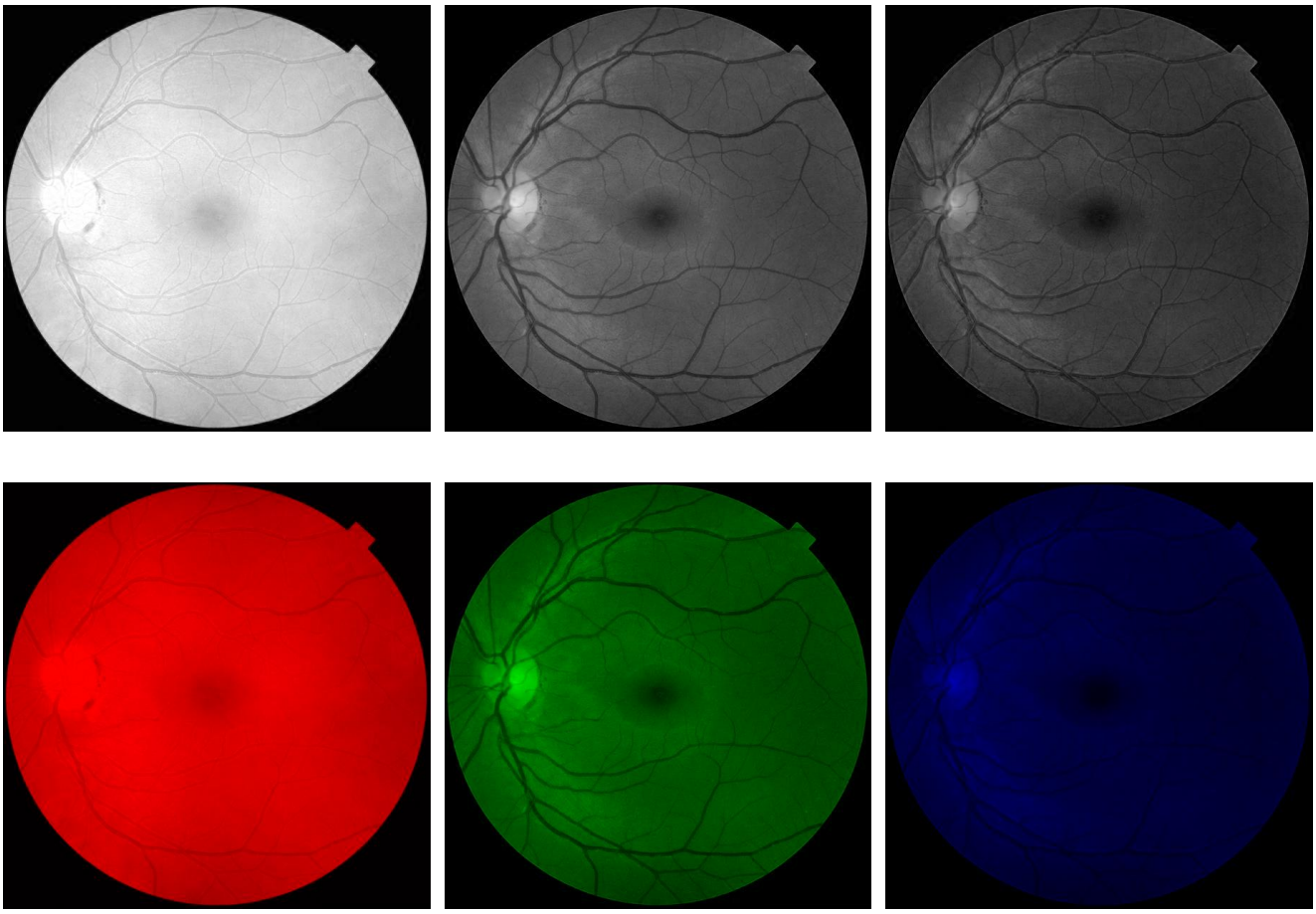
- Be careful with imshow:



2.1. Channel extraction.

Write script to:

a) load a color image.
b) check whether the image is 24 or 8 bit. Print the size of the image in the command window,
c) calculate the following six images: gray-level images of the R, G, and B channels, and three color images each one containing the following information: (R,0,0), (0,G,0) and (0,0,B).
d) write a script for displaying the six images in a single window.



2.2. Luminance, colormaps and false color, and gamma contrast.

a) Calculate the average of the three channels and display the result `M=0.333*(R+G+B)`. Be careful: Are we dealing with 8-bit integers or floating point numbers?
b) Calculate the luminance of the original color image as `L=0.299*R+0.587*G+0.114*B`. What is the physical meaning of luminance, L?
c) Display image L using different *colormaps.* See: http://matplotlib.org/examples/color/colormaps_reference.html
d) Implement a function for gamma contrast: `imm = im^g` with `g>0`. Display the resulting image. Consider (i) `0<g<1` and (ii) `g>1`. Be careful: in this case we deal with gray-level unnormalized double images.

2.3. How to use the camera on your computer.

The `opencv` library enables us to perform interesting tasks. It can be installed using the *Command Line Interface* (Anaconda Prompt) by typing the following command:

```
conda install opencv
```

For instance, the following snippet enables the camera on the computer, as explained in the following document:
http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_gui/py_video_display/py_video_display.html

Following this tutorial, study and modify the code to save the recorded images as a video file.
Note: this script has only been tested in Windows.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from cv2 import VideoCapture, imshow, waitKey, destroyAllWindows

cap = VideoCapture(0)
while(True):
    ret, frame = cap.read()
    imshow('Camera',frame)
    if waitKey(1)==ord('q'):
        break

cap.release()
destroyAllWindows()
```

Modify this script to produce gray-level (luminance, average, and gamma contrast) real-time video. Use a colormap of your choice.

Lab #3: Image binarization. Dithering.

Image binarization is the process of converting an 8-bit image into a 1-bit black-and-white one. It is often used in image segmentation, dithering, character recognition, etc. Binarization can be understood as a step-like look-up table. Thresholding can be performed using the following order: `imb=255*np.uint8(im>th)` where `im` and `imb` are the gray-level and the binary images respectively, and `th` is the threshold value ranging from 0 to 255 (or 0. to 1. when dealing with double-valued images.)

3.1. Adaptive thresholding.

In general, global thresholds are not very useful for image segmentation purposes, since images can be illuminated in a non-uniform way. A simple alternative is to program an adaptive threshold that takes into account the values of the pixels surrounding the one being considered. One possible way to generate an adaptive threshold is to compute the median of an NxN mask around the pixel being considered (3x3, 5x5, 7x7, etc.). Recall that the median is calculated by arranging *all the observations from lowest value to highest value and selecting the middle one.* Alternatively, instead of using the median, the threshold can be selected by selecting another parameter for the set.

Implement a function/script to calculate a binary image using a median-based adaptive threshold. You might find the following functions useful:
`scipy.signal.medfilt` and `scipy.signal.order_filter`

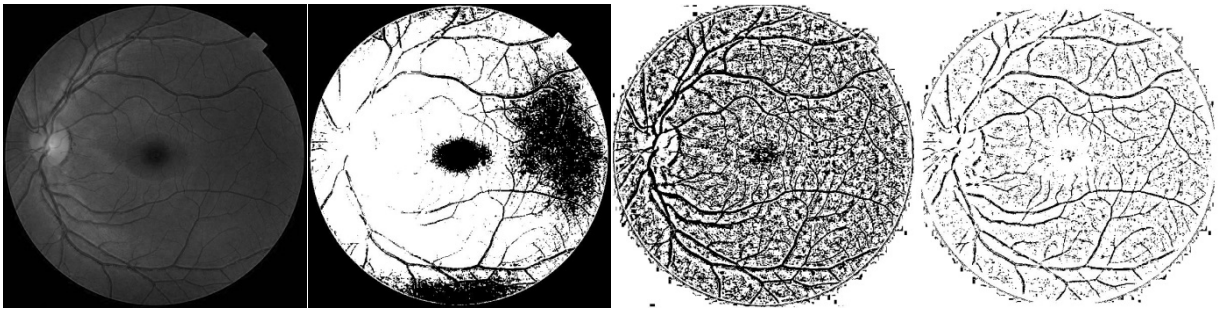


Figure 3.1: (a) Original image; (b) binary image using global thresholding; (c) binary image using median thresholding (`medfilt`); (d) binary image using a different value from the ordered set of the pixel neighborhood (`order_filter`).

3.2. Error diffusion binarization (dithering).

Error diffusion is a half-toning method used in printing and displaying technologies. The binarized image has to be similar to the original gray-level one. The underlying idea is simple: thresholding residual errors are distributed among neighboring pixels.

The algorithm works as follows:

- The program processes an NxM gray-level image using a pixel-by-pixel approach. Starting from pixel [0,0], the algorithm scans every row starting from the first column. (Alternatively, in the snake approach, when pixel [0,M-1] is reached[1], scanning of pixel [1,M-1] follows and then the process continues until pixel [1,0] is reached.)

[1] Boundaries are tricky since column M-1 and row N-1 cannot be processed.

- Let p_{ij}, th and e be the pixel value, the threshold and the quantization error, respectively. Error e at pixel p_{ij} is:

$$e = \begin{cases} p_{ij} - \max\{\text{image}\} & \text{if } p_{ij} > th \\ p_{ij} & \text{if } p_{ij} < th \end{cases}$$

where $\max\{\text{image}\}$ is 255 (or 1.) depending on the numerical class the image belongs to (`uint8` or `float64`, respectively). The threshold, th, is usually set to 128 or 0.5. Of the different possibilities, the following two dithering kernels are widely used: (i) Floyd and Steinberg (FS) and (ii) Jarvis, Judice, and Ninke (JJN):

$$\mathbf{FS} = \frac{1}{16}\begin{pmatrix} 0 & 0 & 0 \\ 0 & p & 7 \\ 3 & 5 & 1 \end{pmatrix} \qquad \mathbf{JJN} = \frac{1}{48}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & p & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{pmatrix}.$$

Then, according to the FS kernel, the error diffusion transformation induced on the neighborhood of p_{11} is:

$$\begin{pmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & p_{22} \end{pmatrix} \Rightarrow \begin{pmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} + e \cdot 7/16 \\ p_{20} + e \cdot 3/16 & p_{21} + e \cdot 5/16 & p_{22} + e \cdot 1/16 \end{pmatrix}.$$

Note that values p_{00}, p_{01}, p_{01}, p_{10} and p_{11} are not changed in this step.

- Quantization error is diffused across the image and finally the global threshold th is applied.

Write code to implement the error diffusion algorithm using the FS and JJN kernels.

Figure 3.2: (a) Original color image; (b) binary dithering of (a) using the JJN kernel; (c) binary dithering using the JJN kernel on the eye fundus image.

Note: The Structural Similarity Index provides a way of comparing processed images with a reference. Use `skimage.measure.compare_ssim`:

http://scikit-image.org/docs/dev/auto_examples/transform/plot_ssim.html and
http://scikit-image.org/docs/dev/api/skimage.measure.html#skimage.measure.compare_ssim .

3.3. Color dithering. The HSV color model.

The present algorithm can be extended to color images. This is useful for producing color images that can be encoded in a single 8-bit unit (instead of 24-bit) or even less (1-bit per channel).
[Have a look at https://en.wikipedia.org/wiki/Web_colors#Web-safe_colors].

a) Write code to reduce to six the number of levels available in each channel (0, 51, 102, 153, 205 and 255). Note that a 6 x 6 x 6 = 216 color is obtained, that can be fit in 8 bits. An index file that relates the present RGB values with the actual encoded 8-bit one is required. This is how GIF images are encoded.

b) Use the FS algorithm to dither the three channels. Produce a new color image with only 8 colors. Compare the dithered image with the previous result. Note that the procedures presented in sections a) and b) are independent.

c) Write code to generate web-safe color images: The original RGB image is converted into the HSV color model (Huge-Saturation-Value), with V= max{R, G, B}; the component V is binarized according to the FS algorithm, whereas H and S are not modified. These HSV components are then used to create an alternative RGB image that is reduced to 6 levels per channel.

   Note: use `matplotlib.colors.rgb_to_hsv` and `matplotlib.colors.hsv_to_rgb`.

   Display the original and the three color-reduced images. Use the `ssim` metric to compare the resulting images with the original one.

Figure 3.3: Color reduction example

Lab #4: More on color and channel transformations

4.1. RGB coordinates from spectrum data. The CIE 1931 XYZ color space.

Design a function that calculates the RGB coordinates from the spectrum data $E(\lambda)$. You need to execute the following steps:

1. Calculate the integrals:

$$x = \int E(\lambda)x(\lambda)d\lambda$$
$$y = \int E(\lambda)y(\lambda)d\lambda$$
$$z = \int E(\lambda)z(\lambda)d\lambda$$

where $x(\lambda), y(\lambda), z(\lambda)$ are the color matching functions. The integrals can be calculated using `scipy.integrate.simps` or similar. The functions $x(\lambda), y(\lambda), z(\lambda)$ are available in a `.npy` file posted on the course webpage. The color matching functions file has five columns: wavelength, $x(\lambda)$, $y(\lambda)$, $z(\lambda)$, and the emission spectrum of a lamp $E(\lambda)$. Plot the color matching function and the spectrum in a single plot.

2. Then, normalize *x, y* and *z*:

$$X = \frac{x}{x+y+z} \quad Y = \frac{y}{x+y+z} \quad Z = \frac{z}{x+y+z}$$

where *X, Y, Z* are the coordinates in the CIE 1931 XYZ color space.

3. RGB and CIE coordinates are related by the following linear relationship:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \text{np.uint8}\left[255 * \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}\right]$$

4. Finally, create an NxNx3 `numpy` array to display the color corresponding to the lamp spectrum.

4.2. Histogram equalization.

Sometimes, images do not use all the available dynamic range. Histogram equalization is a technique that involves modifying the image histogram in such a way that the frequency of appearance of gray levels is constant. Thus, the cumulative histogram becomes linear.

In this exercise you are required to design a function that generates equalized images according to the following algorithm:

Let `im` be an MxN pixel 8-bit gray-level image. Histogram h[g] of image `im` is easily calculated using the function `scipy.ndimage.measurements.histogram`. The cumulative histogram `ch(g)` of `im` is obtained with the help of method `.cumsum()`. Plot both the histogram `h[g]` and the cumulative histogram `ch[g]`. Then equalize the image using:

```
eq = np.uint8(255 * ch[im] / (M * N))
```

Write your code as a function for future reference.

More information: http://en.wikipedia.org/wiki/Histogram_equalization.

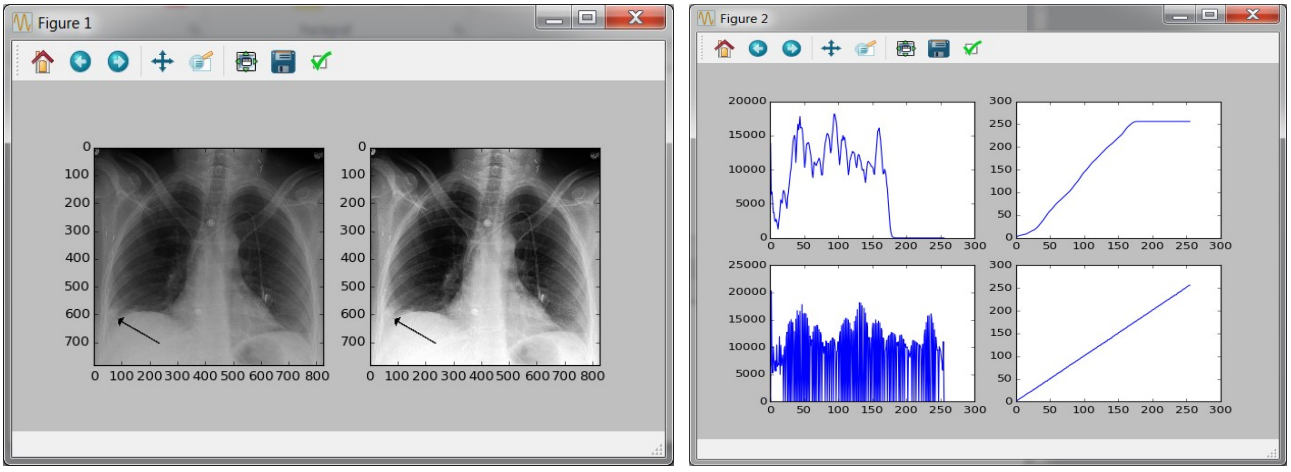
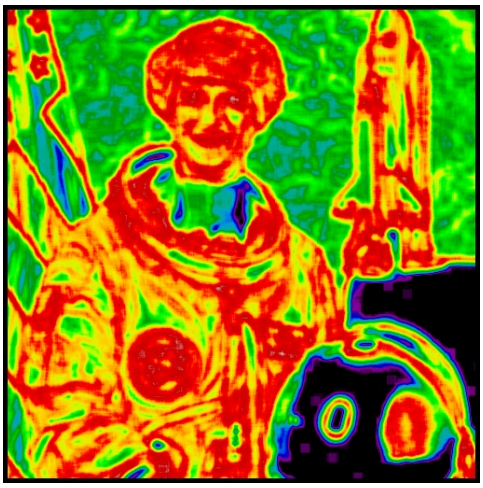


Figure 4.1: Left: (a) original image, (b) equalized image. Right: (a) histogram of the original image, (b) cumulative histogram of the original image, (c) histogram of the equalized image, (d) cumulative histogram of the equalized image. Note, the histogram of the equalized image is linear.

Image credit: https://upload.wikimedia.org/wikipedia/commons/8/83/Hamptonshump.PNG.

4.3. Image entropy.



Information entropy is defined as the amount of information (in bits) required to encode a signal. For a 256 gray-level image, the mathematical equation for the entropy is:

$$S = -\sum_{i=0}^{255} P_i \log_2 P_i \quad \text{(bits/pixel)},$$

where P_i is the probability associated with the gray level (i.e., the number of times gray level i is present in the image / number of pixels in the image. The local entropy at pixel [i,j] is:

$$S[i,j] = -\sum_{k=0}^{255} P_k[i,j] \log_2 P_k[i,j] \quad \text{(bits/pixel)}$$

where $S[i,j]$ is the local entropy at $[i,j]$ and $P_k[i,j]$ is the local probability associated with gray level k within the MxM neighborhood around pixel $[i,j]$. In this way, an entropy image can be produced.

Using the green channel of `skimage.data.astronaut`:
a) determine the global entropy of the image.
b) calculate the local entropy image using an 11x11 pixel neighborhood. You might want to display the result in combination with a `colorbar()`.
Note, this procedure is implemented as a function in:
http://scikit-image.org/docs/dev/api/skimage.filters.rank.html#skimage.filters.rank.entropy.

4.4. Least significant bit steganography.

Steganography is a technique intended to hide information within an image. It is used for watermarking in order to preserve copyright; when used in combination with encryption, it provides an extra layer of security. Please check the following paper for more insight into this technique:

N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen", *Computer* ***31***(2), 26-34 (1998) http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4655281&tag=1.

In this exercise we are going to implement a simple steganography technique: encoding a secret message or image in the least significant bits of a host image

1. Let us first consider two 8-bit images. Ideally, both images have the same number of pixels; this can help, but is not compulsory. You might use `scipy.misc.imresize`[2]. First, the dynamic ranges of both the secret and the host images are reduced to 4 bits. The 4 least significant bits (gray levels 0 to 16) of the host image are used to encode the secret image and the 4 most significant remain the same. The resulting image looks very similar to the host image but careful visual inspection may reveal the content of the secret image.
2. This technique can easily be refined. For instance, a color host image provides 24 bits of information. Using the two least significant bits in each channel enables us to encode 6-bit gray-level secret images.
3. In order to improve the security of the method, the pixels of the secret image can be scattered at random. Note that the seed of the random number generator becomes the key to this simple encryption.

Write code to encode secret images of arbitrary resolution within a color image.
Note that if you plan to write your code using binary numbers, the following functions may help you: `numpy.unpackbits` and `numpy.packbits`.

4.5. Visual encryption.

A binary plaintext B can be split into two random 2D arrays using the XOR logic function. First, a random binary distribution A1 should be produced. Then, array A2 that fulfills A1^A2 = B is determined. Distributions A1 and A2 can be given to two different recipients: the information B is only accessible when both arrays are used together. The XOR logical function is described by the following table:

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

1. Binarize an image using the error-diffusion algorithm. Split the resulting image using the procedure explained above.
2. Implement this method using gray-level images: first produce 8 bit planes. Then use the XOR strategy explained above for each bit plane; and finally, join the two bit plane sets into two random gray-level images (use `numpy.unpackbits` and `numpy.packbits`).
3. Generalize this method for use with 24-bit (color) images.

[2] Depending of the version used, this function might not be available.
Use `scipy.ndimage.zoom` or `skimage.transform.resize` instead.

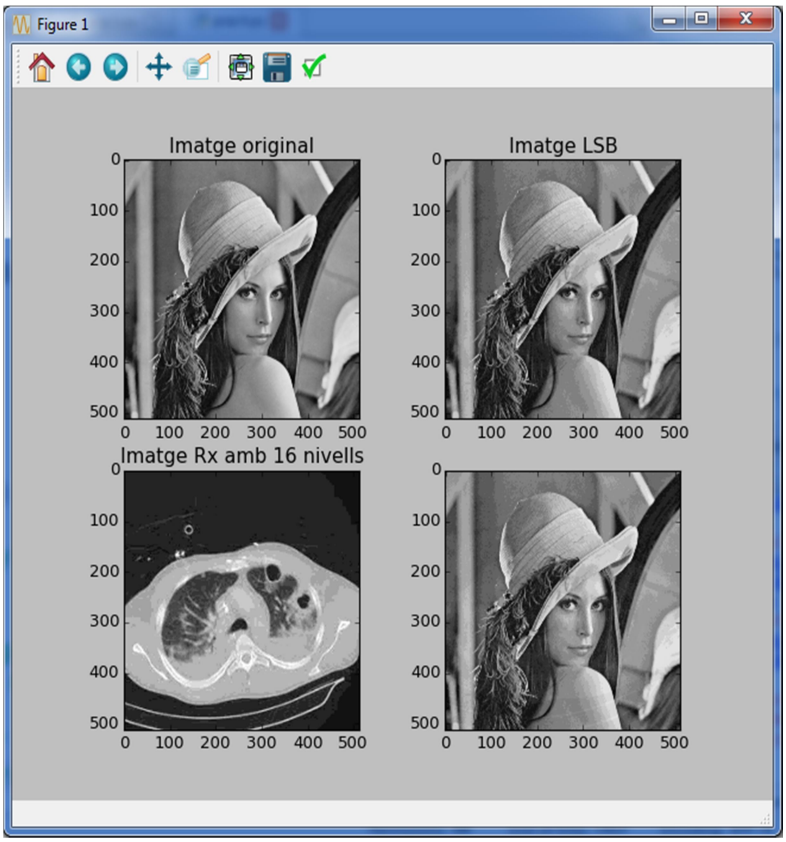


Figure 4.2: (a) 256 gray-level original image, (b) 16 gray-level host image, (c) 256 gray-level image to be hidden, (d) host + hidden image.

Lab #5: K-means clustering in remote sensing imaging

Clustering algorithms are used to form groups of data points that share some specific properties by which we can group them. Very often, in order to increase the amount of accessible information, images from different sources and spectral bands are used.
Let us assume a group of data points: $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\}$, where $\boldsymbol{x}_L \in \mathbb{R}^d$, $L \in \{1,2,\dots,N\}$, N is the number of data points and d is the dimensionality of the space in which the data points reside. A clustering of X is defined as ***hard clustering*** if all the points are grouped into A_i clusters, $i = 1, \dots, K$ such that:

$$\bigcup_i A_i = X$$

$$\forall \{i,j\} \in k; i \neq j \Longrightarrow A_i \bigcap A_j = \emptyset$$

$$\forall i \in k \Longrightarrow \emptyset \neq A_i \neq X$$

In other words, the pixels of the data images can be grouped into several disjointed sets according to certain predetermined rules. For instance, in this lab we will use two different images of an aerial view of Barcelona airport[3]. The first is a conventional RGB image; the second, a three-channel near-infrared picture. The two images cover the same area (they are correlated pixel-by-pixel). The three channels of the false-color infrared image are infrared, red and green. Accordingly, we will use the three RGB channels of the conventional image and the IR of the second image. The goal of this lab is to group (cluster) the RGBI information into an arbitrary number of clusters.

Figure 5.1: (a) RGB and (b) IRG aerial images of Barcelona airport.

K-means[4] is a type of clustering algorithm that produces hard clustering of the data points. It is based on the minimization of a so-called *functional* or *merit figure* that aims to minimize the distance between each data point in a cluster and a point in that cluster called the *centroid* (or *representative*). The K-means method consists of the following steps:

1. First, K points are selected as the initial cluster centroids $\{\boldsymbol{c}_1(1), \boldsymbol{c}_2(1), \dots, \boldsymbol{c}_K(1)\}$
2. At the k-th iteration, each data point, $\boldsymbol{x}_L$, is assigned to the cluster $\boldsymbol{c}_j$ for which the following applies:

$$\|\boldsymbol{x}_L - \boldsymbol{c}_j(k)\| \leq \|\boldsymbol{x}_L - \boldsymbol{c}_i(k)\|; \forall \{i,j\} = 1,2,\dots,K, i \neq j$$

[3] The orthoimages were obtained from Sentinel Copernicus, modified by the Institut Cartogràfic i Geologic de Catalunya: http://www.icgc.cat/Administracio-i-empresa/Descarregues/Imatges-aeries-i-de-satel-lit/Ortoimatges-Sentinel-2 . The two images used were recorded in July 2018, and are available at
http://auriga.icgc.cat/descarregues2/dl.php?t=sen2rgb8bv10tf0f04s1_201807_0.zip&f=04&l=cat
http://auriga.icgc.cat/descarregues2/dl.php?t=sen2irc8bv10tf0f04s1_201807_0.zip&f=04&l=cat.
If you want to extract the airport images, consider the following figures [5800:5800+256, 3050:3050+512]

[4] Richard O. Duda, Peter E. Hart, & David G. Stork, *Pattern Classification* (2nd Edition), John Wiley & Sons, 2001.

The coordinates of the new centroids are obtained by minimizing the following functional:

$$J_j = \sum_{\boldsymbol{x}_L \in c_j(k)} \left\| \boldsymbol{x}_L - \boldsymbol{c}_j(k+1) \right\|^2, j = 1,2,\dots,K$$

3. The coordinates of the new centroids are then given by:

$$\boldsymbol{c}_j(k+1) = \frac{1}{N_j} \sum_{\boldsymbol{x}_L \in \boldsymbol{c}_j} \boldsymbol{x}_L\,, \qquad j = 1,2,\dots,K$$

4. The method stops when $\left\| \boldsymbol{c}_j(k+1) - \boldsymbol{c}_j(k) \right\| \leq \epsilon; j = 1,2,\dots,K$, or when a specific number of iterations has been reached.

K-means is implemented in the sklearn library: `sklearn.cluster.KMeans` described at https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

Please take into account the different variables of the function. They affect the behavior (convergence, reproducibility, precision, speed, etc.) of the algorithm. Note that methods `fit()` and `predict()` should be used as well.

The algorithm labels every pixel with a certain number. The function `skimage.color.label2rgb` is useful to display the resulting image. Use a proper colormap; note that `label2rgb()` allows you to generate your own colormap.

In the example provided, K = 5 clusters have been obtained: they approximately represent water, forests, buildings, roads, and others. Finally, display the six scatterplots of coordinate vs. coordinate (G vs. R, B vs. R, etc.) with the function `matplotlib.pyplot.scatter`. For example, the IR vs. R scatterplot is shown below:

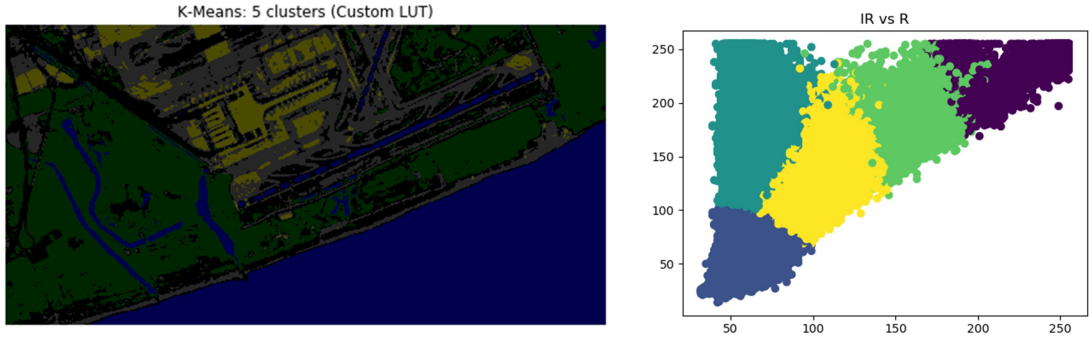


Figure 5.2: (a) K-means processed image, (b) IR vs. R scatterplot.

This lab has been possible thanks to the help of Dr. Pedro Latorre, Universitat Jaume I.

Lab #6: Fourier transforms and spatial filtering.

The Fourier transform (FT) is a mathematical operator used to send information contained in an original signal into the frequency domain. In this way, information is arranged in such a way that it can be conveniently processed. In this lab, we will study how to calculate the 2D FT of an image. Some basic filtering procedures related with FTs are initially reviewed.

6.1 Basic operations.

Starting from an 8-bit test image, perform the following:

1. Calculate the 2D Fast Fourier Transform (FFT) using `numpy.fft.fft2()` and `numpy.fft.fftshift()`.
2. Display the amplitude and phase of the FT. Use `numpy.abs()` and `numpy.angle()` (real and imaginary part are not interesting. Why?)
3. Inverse FT the result (use `numpy.fft.ifft2()` and `numpy.fft.ifftshift()`). Try to recover the original signal.
4. Note that the amplitude can take very high values at the center. Calculate the base-10 logarithm of the amplitude of the FT. Alternatively, saturate the values over a certain arbitrary threshold (e.g., 0.1%, or 0.01% of the maximum value of the amplitude of the FT). Display the result.

6.2 Fourier series and filtering of spatial frequencies.

Using the function `scipy.signal.square`, generate a 5 Hz square wave with a sampling frequency of 500 samples per second (see the example in the documentation of the function). Calculate the 1D FT (with `numpy.fft.fft()` and `numpy.fft.fftshift()`) and filter the high-frequency harmonics except those that correspond to a sinusoidal signal. Compute the inverse FT and show the result.

Determine the size of Fourier space using the Nyquist-Shannon theorem: Let L, N and T be the length, the number of pixels and the distance between pixels, i.e.: `L = NT`. Then, the Nyquist frequency (a.k.a. cut-off frequency) is f_N=`1/(2*T)`. This means that accessible frequencies span the interval `[-1/(2*T), 1/(2*T)]`. Note that two cut-off frequencies can coexist if `N≠M`.

Generalize this calculation to 2D. Produce a 500x500 pixel 5 Hz 2D bar test (tip: use `scipy.signal.square` and `numpy.meshgrid`). Display the amplitude of the FT and plot the central row. Filter the FT as in the previous exercise. Display the inverse FT.

2D distributions can be generated by means of `np.meshgrid()`. This function can be understood as a 2D or 3D generalization of `np.linspace()`. In order to understand how this function works, analyze what the following command does:

```
u, v = np.meshgrid(np.linspace(-1, 1, 5), np.linspace(-1, 1, 5))
```

6.3. Relative importance of amplitude and phase of the Fourier transform.

Now explore how the information in the image is distributed: What is more important, the phase or the amplitude of the FT? An interesting way to do this is to reproduce the classic Oppenheim and Lim experiment[5] which consists in exchanging the phase and amplitude of the FT of both images and then inverse Fourier transforming them. This exercise suggests how to design spatial filters in

[5] A. V. Oppenheim, and J. S. Lim, "The importance of phase in signals," Proc. IEEE **69** (5), 529-541 (1981).

the Fourier domain.

1. Take two gray-level images (which must have the same number of pixels; otherwise adjust their sizes).
2. Calculate their FTs. Swap amplitudes and phases and then inverse transform these mixed distributions.
3. Now take the test images and calculate their respective phase-only FTs by setting the amplitudes to a constant value equal to 1. Inverse FT these phase-only distributions. Show the results. What conclusion can be drawn?

6.4. Spatial filtering.

Linear processing is based on the integral convolution product. Let *i(x,y)* and *f(x,y)* be two (discrete) functions that represent the image and the filter respectively. The processed image is obtained by performing *p(x,y) = i(x,y) * f(x,y)*, where * stands for convolution, i.e.:

$$[i * f](x.y) = \int i(x', y')f(x - x', y - y')dx'dy'$$

Convolution produces a modified version of *i(x,y)* giving the overlap between *i(x,y)* and *f(x,y)* as a function of the amount that *f(x,y)* is translated. Calculation of the convolution product is straightforward in the frequency domain using the convolution theorem:

$$FT[i * f] = FT[i]FT[f] \Rightarrow i * f = FT^{-1}[FT[i]FT[f]]$$

In what follows, lower-case and capital letters are used for distributions in image and Fourier planes respectively, i.e., $I=FT[i]$, $F=FT[f]$ and $P=FT[p]$.

6.4.1. The sharp cut-off low-pass filter.

Implement a function for a sharp cut-off low-pass filter $C(\rho;R) = \mathrm{circ}(\rho/R)$ where:

$$\mathrm{circ}(\rho/R) = 1 \ \rho \le R$$
$$\mathrm{circ}(\rho/R) = 0 \ \rho > R,$$

$\rho = \sqrt{u^2 + v^2}$ and R is the cut-off frequency of the filter. For simplicity, take $[u, v] \in [-1,1]$ and select R within the range [0,1].

Calculate $i * c = FT^{-1}[FT[i]C(\rho;R)]$. Use different values of R and show the resulting images. Note that $i * f$ are, in general, complex-valued: use `np.abs()`.

6.4.2. The Laplacian filter[6].

Implement and display function $L(\rho) = \rho^2$. Calculate $i * l = FT^{-1}[FT[i]L(\rho)]$.

6.4.3. The Gaussian filter.

Implement a function for $G(\rho;\sigma) = \exp(-\rho^2/(2\sigma^2))$. Again, calculate $i * g = FT^{-1}[FT[i]G(\rho;\sigma)]$. Use appropriate values of σ. Add a large amount of zero-mean additive Gaussian noise to your test image *i(x,y)*. Use `numpy.random.randn`. Test the Gaussian filter with noisy images.

6.4.4. Quasi-periodic noise filtering.

Sometimes, images are recorded in such a way that a periodic signal is superimposed. This annoying effect can be removed by blocking the frequencies responsible of the periodic signal.

[6] Fourier transform of the second derivative: $TF[\nabla^2 f(x,y)] \propto (u^2 + v^2)TF[f(x,y)] = (u^2 + v^2)F(u,v)$.

1. Detect the undesired frequencies. Calculate the FT of the image and display the amplitude in a log scale.
2. Generate the filter that removes the periodic signal and calculate the inverse FT.

You may use one of the following images:

- http://img-service.com/overview/pics/chap8/clown.jpg
- https://www8.cs.umu.se/kurser/5DV015/VT09/handouts/images/mit_noise_periodic.jpg

6.5. Spatial filtering in the image domain.

6.5.1. Linear convolution kernels.

Calculate $c=i * f$ in the image domain.
Python provides the function `scipy.ndimage.filters.convolve`[7]. Note that the size of the kernel f can be 3x3, 5x5, 7x7, etc.

Select an image and use the following filters:

$$f_1 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad f_2 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad f_3 = \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

What do the resulting images look like? Why?
Take filter f_1 *(or* f_2 *or* f_3*)*. Calculate:

$$FT[f_3] = FT\left[\begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}\right],$$

(i) pad the kernel f_3 to NxM pixels, (ii) calculate the Fourier transform and finally, (iii) take the modulus, normalize and display the result. Do the same for f_2 *and* f_3. How do these filters relate to the Laplacian filter?

Convolve the following filters with your test images:

$$f_1 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad f_2 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad f_3 = \begin{pmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

What can you deduce?

Add a large amount of zero-mean Gaussian additive noise to your test image $i(x,y)$. Then calculate the convolution of the noisy image with:

$$f_1 = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

6.5.2. The **Kirsch compass kernel** is a non-linear edge detector that finds the maximum edge strength in some predetermined directions. The operator takes a single kernel mask and rotates it

[7] Be careful with this function. If the image is `unit8`, it is very likely that the convolution calculation will surpass the 255 limit. Cast the image value to `double` to avoid normalization problems.

through 45 degree increments to 8 compass bearings: N, NW, W, SW, S, SE, E, and NE. The edge magnitude of the Kirsch operator is calculated at every pixel as the maximum magnitude across all directions:

$$h = \max_{z=1,\ldots,8} \left\{ \left| f * \mathbf{g}^{(z)} \right| \right\}$$

where z enumerates the compass direction kernels, max is the maximum operator, f is the image to be processed, * stands for convolution, | | indicates the absolute value, and $\mathbf{g}^{(z)}$ is a 3x3 convolution matrix defined as:

$$\mathbf{g}^{(1)} = \begin{bmatrix} +5 & +5 & +5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(2)} = \begin{bmatrix} +5 & +5 & -3 \\ +5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(3)} = \begin{bmatrix} +5 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(4)} = \begin{bmatrix} -3 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & +5 & -3 \end{bmatrix}$$

etc. Write script that implements this filter.

6.5.3. Salt and pepper noise.

Add a large amount of salt & pepper noise to your test image $i(x,y)$ (note: this kind of noise is not implemented in Python). Alternatively, use the following image:
https://en.wikipedia.org/wiki/Salt-and-pepper_noise#/media/File:Noise_salt_and_pepper.png .

Calculate the convolution of the corrupted image with $\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$. Did you get a good result? Instead, use the median filter in a similar way as the adaptive threshold function you designed in Lab #3.1.

6.5.4. Roberts, Sobel and Prewitt filters.

Non-linear edge detection operators are defined as:

$$f_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad f_y = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad c = \sqrt{|f_x * i|^2 + |f_y * i|^2}$$

$$f_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad f_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} c \quad = \sqrt{|f_x * i|^2 + |f_y * i|^2}$$

$$f_x = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \quad f_y = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \quad c = \sqrt{|f_x * i|^2 + |f_y * i|^2}$$

Use these filters to detect the edges of the test image.

Lab #7: Defocused images and image restoration filters.

7.1. Calculation of the PSF of an optical system

A defocused image can be modeled as the convolution between the ideal, perfect image and the point–spread function (PSF). In paraxial geometrical optics, the out-of-focus PSF is approximated as the Fourier transform of the exit pupil (a circle), $h(x,y) = TF_{\lambda s}\left[\text{circ}\left(\frac{R}{R_0}\right)\right]$, where R is the polar coordinate and R_0 is the radius of the exit pupil of the instrument; and s is the distance between the exit pupil and the image plane. Recall that $h(x,y)$ is the so-called Airy disc with radius r_A:

$$r_A = 1.22\frac{\lambda s}{2R_0}. \tag{1}$$

Those instruments with a PSF described by an Airy disc are named *diffracted limited*. If the system is illuminated with incoherent (natural) light, the defocused image $d(x,y)$ is calculated by means of :

$$d(x,y) = i(x,y) * \left|h(x,y)\right|^2 \tag{2}$$

Now, produce several diffracted limited defocused images (use different values for R_0).Use the convolution theorem. Note that both $d(x,y)$ and $|h(x,y)|^2$ have to be converted to `np.uint8` values to simulate the behavior of the recording process.

The following table summarizes the variables that describe an optical system:

Radius of the exit pupil R_0	up to 17 mm
Length of the window describing the exit pupil L	34 mm
Image distance s	316 mm
Wavelength λ	550 nm

Using Eq. (1), calculate the radius of the Airy disc. Determine the length (in pixels) of r_A using the numerical calculation. Use the Shannon theorem adapted for optical systems: note that the visualized area in the image plane is $L_f = \lambda s\frac{N}{L}$.

7.2. Image restoration filters.

Image reconstruction filters can be used to minimize the effects of aberration or defocusing. Note that Eq. (2) is written as $D=IH$ in Fourier space, where D=FT[d], H=FT[$|h|^2$] and I=FT[i]. It would seem that image I could be restored if we perform the calculation D/H.

Nevertheless, $1/H$ is not well behaved at some frequencies. To avoid division-by-zero errors, the *inverse filter* is defined as $F_I = 1/(H + k)$, where k is a constant value selected in such a way as to minimize noise in the reconstructed image. The restored image, d_r, is obtained by means of Eq. (3):

$$d_r = \text{TF}^{-1}\left[DF_I\right] = \text{TF}^{-1}\left[D\frac{1}{H+k}\right] \tag{3}$$

Use this filter with the defocused image obtained in the previous section.

The *least-squares filter* is defined as:

$$d_r = \mathrm{TF}^{-1}\left[DF_{\mathrm{MQ}}\right] = \mathrm{TF}^{-1}\left[D\frac{H^*}{|H|^2 + k\left(u^2+v^2\right)}\right] \qquad (5)$$

where k is a constant to be determined, and u and v are the spatial frequencies. Use the defocused image obtained in the previous section.

The *Lucy-Richadson (LR) algorism* is described by the following iterative equation:

$$i_{k+1} = \left| i_k \left[\frac{d}{i_k * |h|^2} \right] * |h|^2 \right| \qquad \text{with } i_0 = d \qquad (6)$$

where symbols * and | | stand for convolution and modulus, respectively. Use the defocused image obtained in the previous section and compare results with the inverse and the Least Square filters. Use quality metrics such as the SSIM or the correlation coefficient (`scipy.stats.pearsonr`) to assess the quality of the reconstruction.

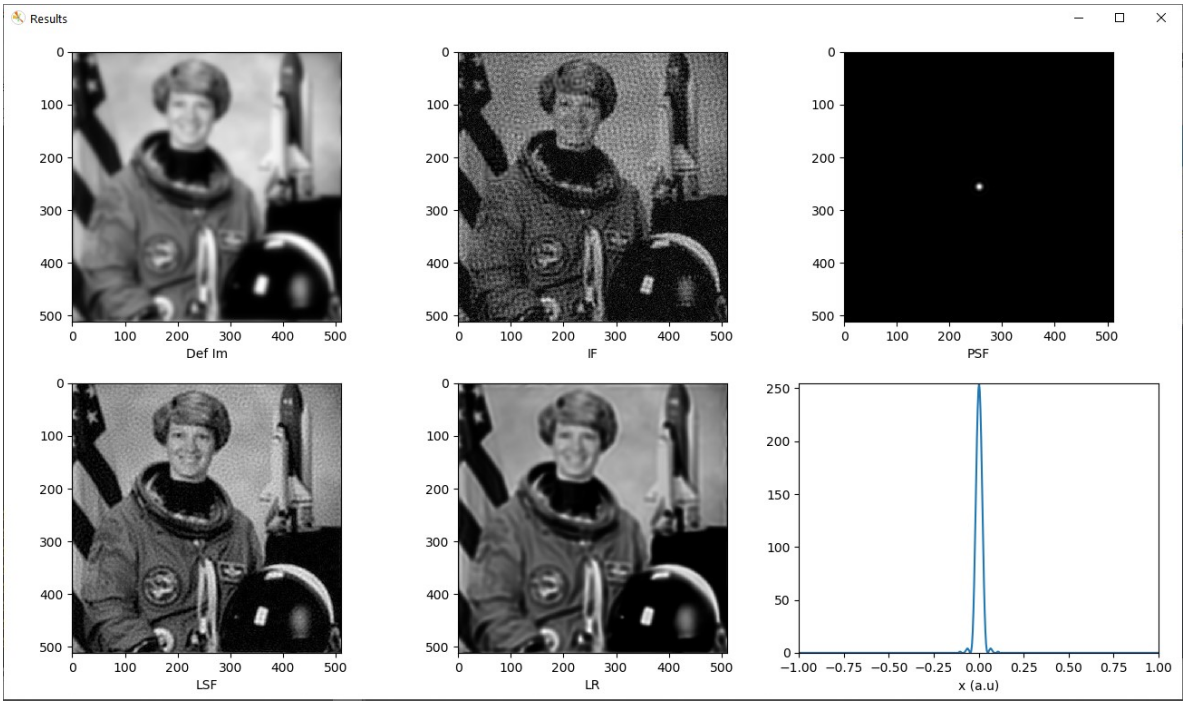


Figure 7.1: an example of restored images using inverse, least-squares and Lucy-Richardson filters.