IMAGE PROCESSING TECHNIQUES USING PYTHON

Image Processing and Artificial Vision

Grau de Física
Grau d'Enginyeria Electrònica de Telecomunicacions

Course notes and exercises
Academic year 2018-19

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

Updated Jan 30, 2019

Table of contents.

Lab #8: Radom Transforms and the Projection-Slide Theorem.

proyecto+trabajo escrito 40%

middterm 20%

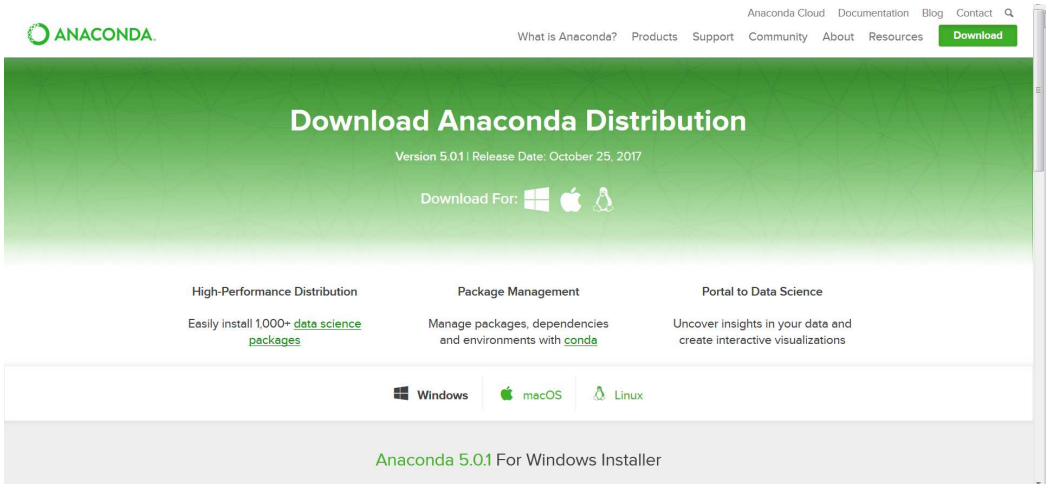
final 40%

(funció màxima)(mínim 5/10 in final & project)

Lab #1: Python concepts for image processing.

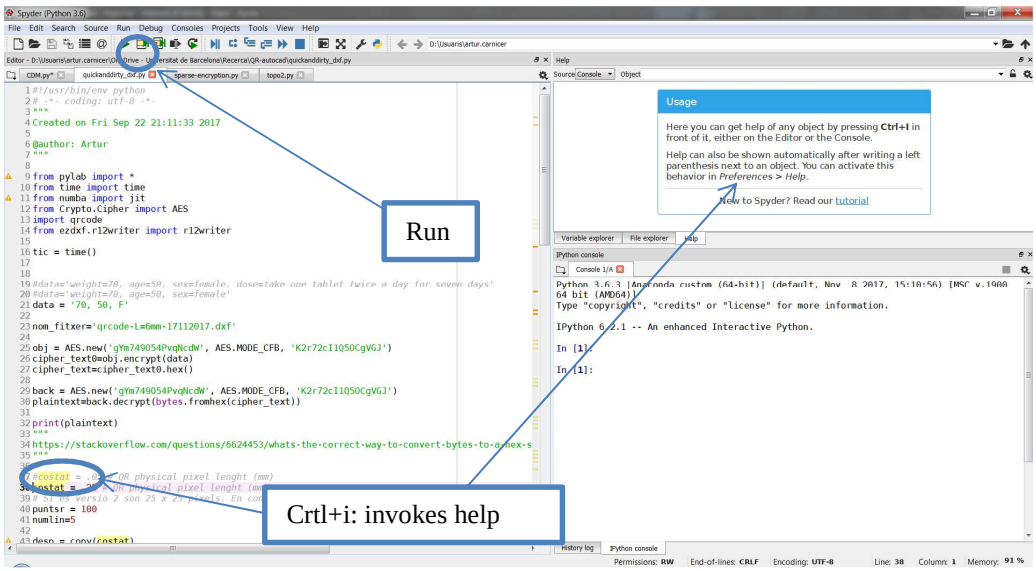
1.1. Download and install Python 3.7 + Spyder from the Anaconda website.

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



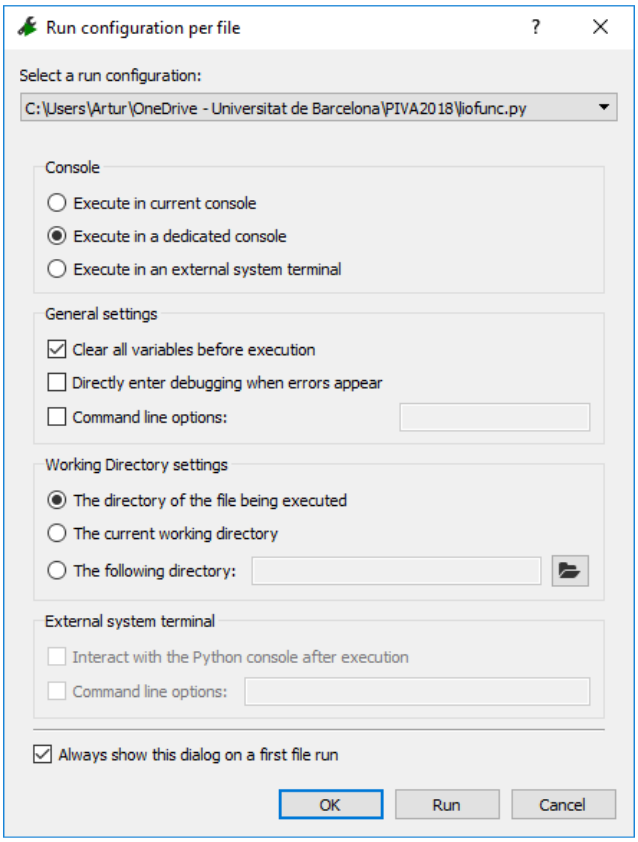
The browser will detect the OS of your computer. Otherwise, select the right choice by clicking on the proper icon. In general, download the 64 bit version; only old computer with less than 2MB memory require the 32 bits version.
For Mac OS users: you can download either the graphical or the command 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 in your computer, it is advisable to remove it (except you know how to manage virtual environments).
Please select the Python 3 flavor. Python 2 is going to be discontinued soon.

1.2. Familiarize with the editor button and panes.

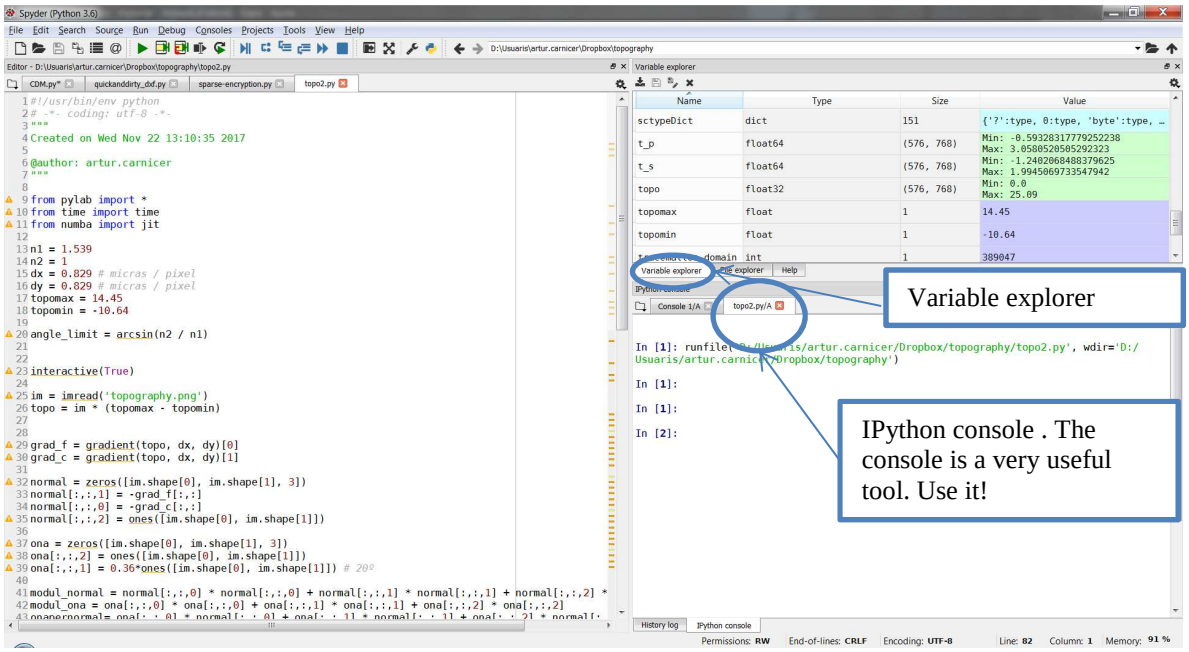


Save your script before the first run **Do not use directories and scripts filenames that contain spaces, or language-based extended symbols such as á, è, ñ, ź, ę, et cetera.**

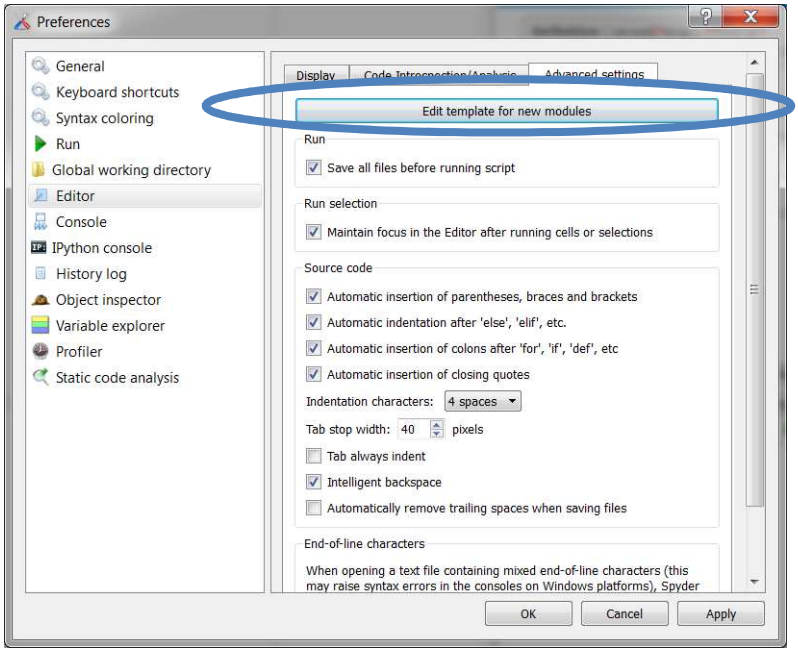
The *Run Configuration per file* dialog appears first time the code runs (or selecting Run -> Configuration per file). Select the following options (why? you should understand what they mean):



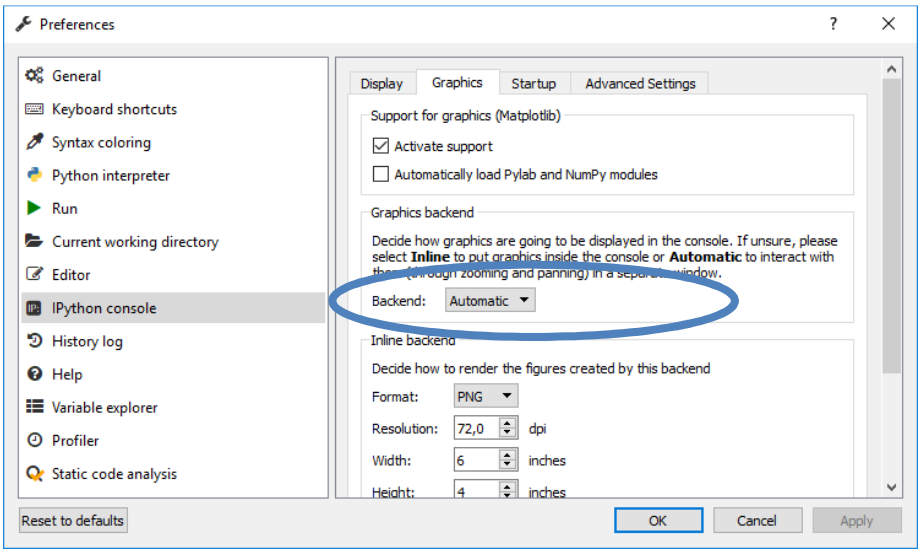
After execution, the variable explorer tab and the console are available for inspection.



It is advisable to edit the template.py script.
Tools > Preferences > Editor > Advances > Edit template for new modules
We discuss 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 in the IPython Console > Graphics (see figure).



For MATLAB fans forced to use Python: please consider the information included in 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 expected to know just some parts of it:

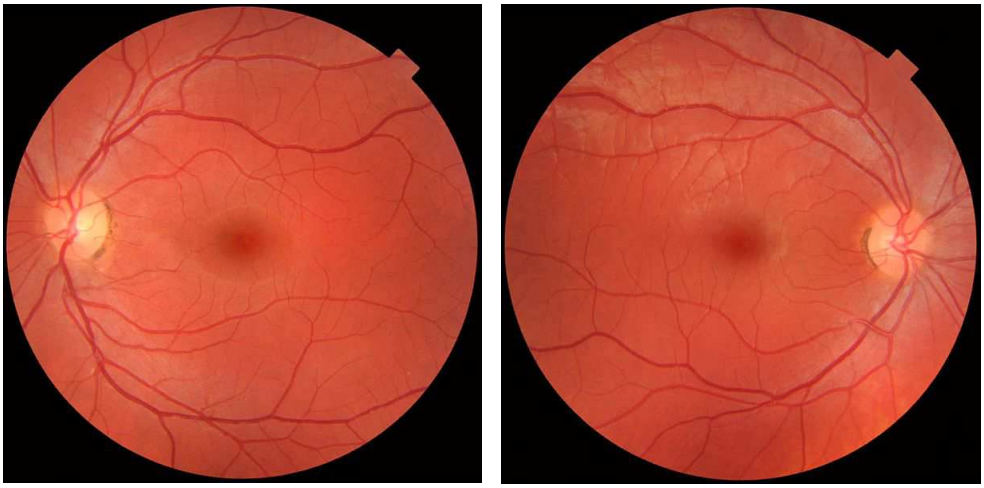
- Numbers, strings, lists, tuples
- for and while loops
- functions
- input/output functions
- import modules
- handle arrays in numpy
- matplolib: figures, plot, images

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

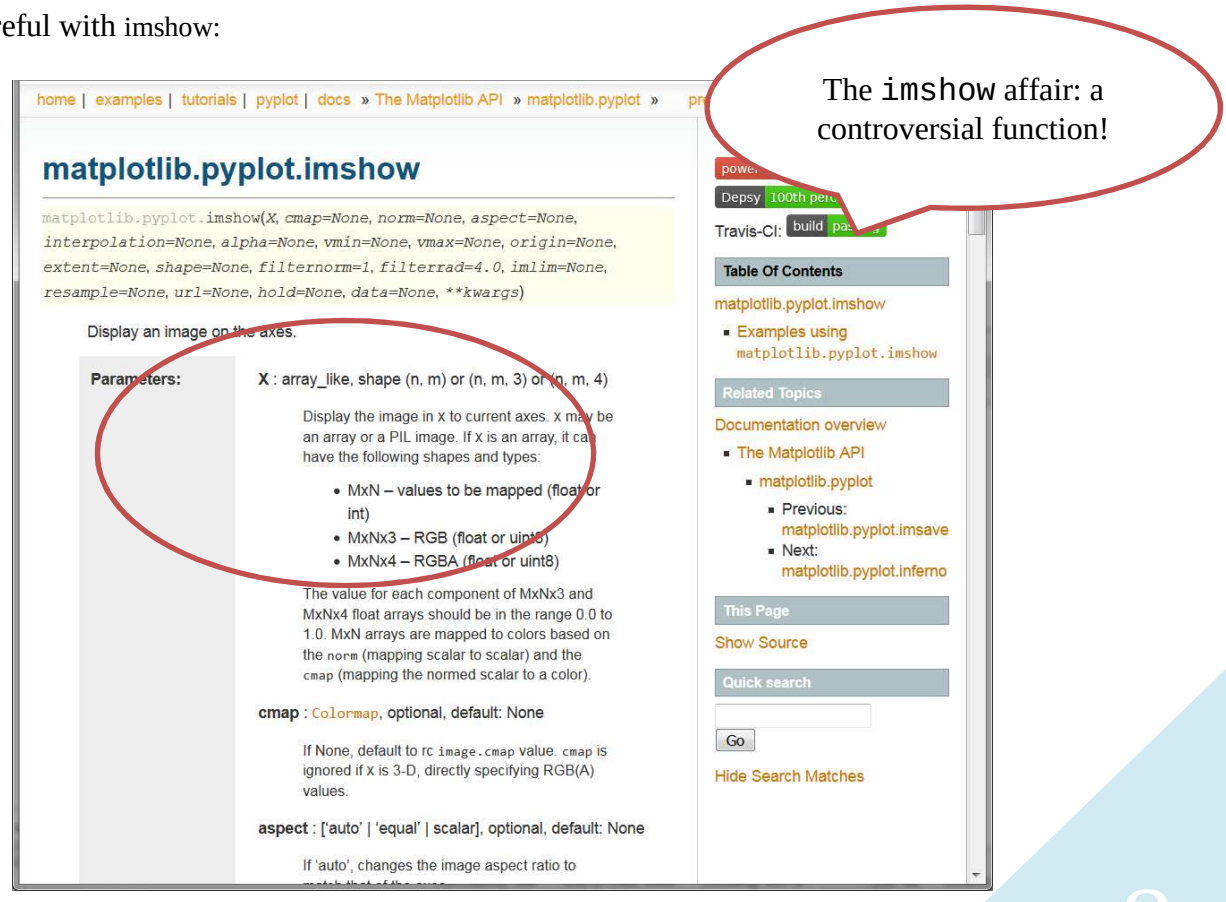
Lab #2: Basic image manipulation: channel processing, colormaps and cameras.

Previous 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 a MxN matrix of 8-bit unsigned integers. A RGB image is a MxMx3 cubic matrix of 8-bit unsigned integers
- Recall how the ':' operator works
- The examples area 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



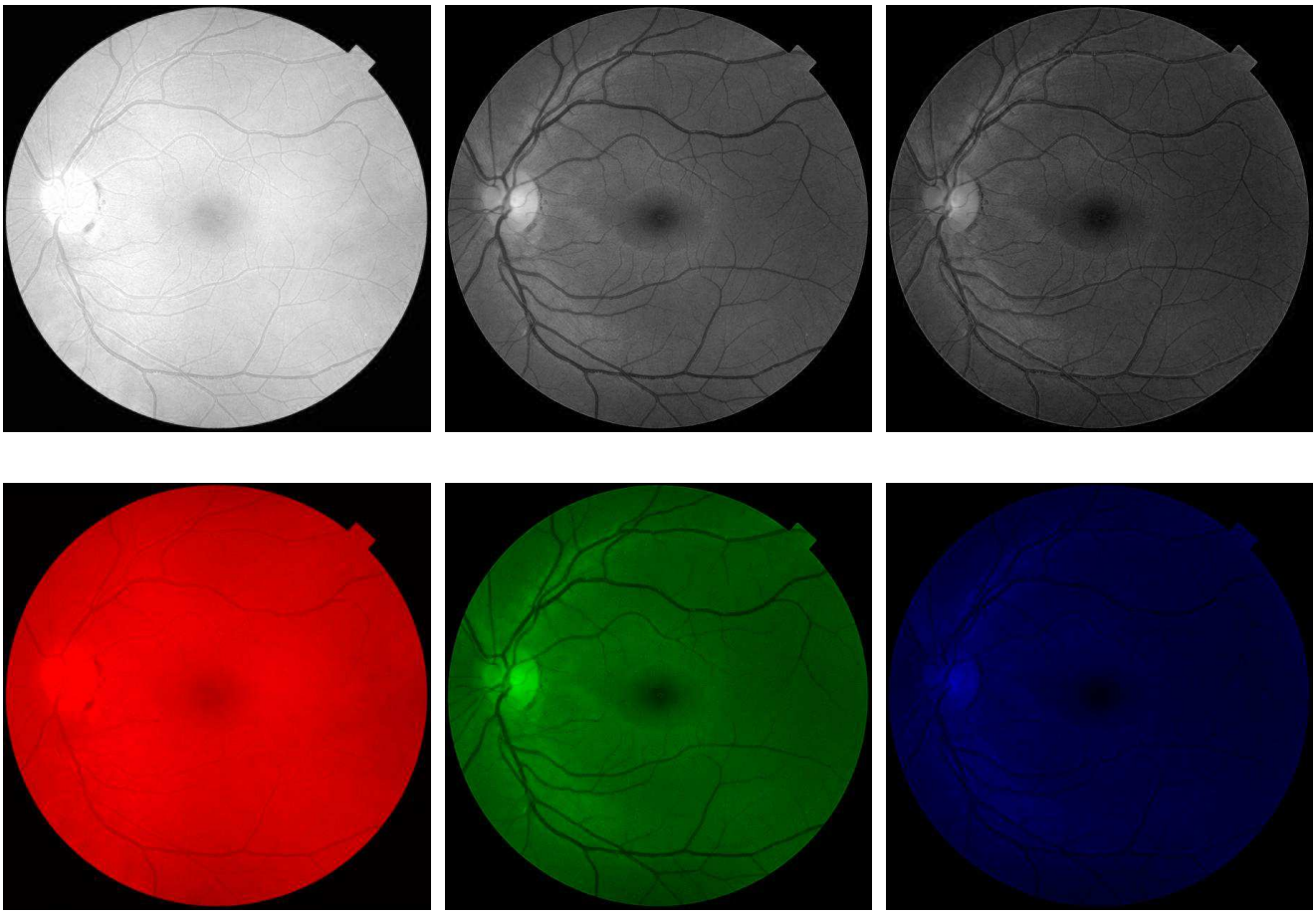
- Be careful with imshow:



2.1. Channel extraction

Develop a script able to

a) load a color image,
b) check whether the image is 24 or 8 bit. Print the size of the image on the command window,
c) calculate the following six images: gray level images of the R, G, and B channels and three color images containing each one 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 tree 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`.

2.3. How to use the camera of your computer.

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

```
conda install opencv
```

For instance, the following snippet enables the camera of 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()
```

Lab #3: Image binarization. Dithering.

Image binarization is a process that converts an 8-bit image into a 1-bit black and white one. This procedure 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.)

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 processed one. One possible way to generate an adaptive threshold is to compute the median of an NxN mask around the considered pixel (3x3, 5x5, 7x7, etc.). Recall that the median is calculated by arranging *all the observations from lowest value to highest value and picking the middle one.* Alternatively, instead of using the median, the threshold can be selected by picking another value of this set.

Implement a function/script to calculate a binary image using a median-based adaptive threshold.

You might find useful the following functions: `scipy.signal.medfilt` and `scipy.signal.order_filter`

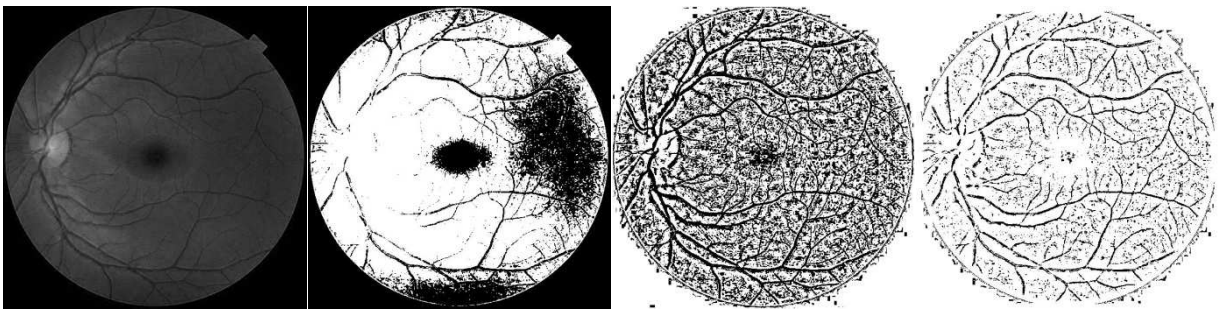


Figure 3.1: (a) Original image; (b) binary image using global thresholding; (c) binary image using median thresholding; (d) binary image using a selected value of the ordered set of the pixel neighborhood.

3.2. Error diffusion binarization (dithering)

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

The algorithm works as follows:

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

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

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

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

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

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

Then, according to the Floyd-Steinberg 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 global threshold th is applied.

Write the 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 to compare 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:
[have a look at https://en.wikipedia.org/wiki/Web_colors#Web-safe_colors].

Develop the code for generating web-safe color images: The original RGB image is converted into the HSV color model (Huge-Saturation-Value), with V= max{R, G, B}; component V is binarized according to the FS algorithm, whereas H and S are not modified (why?) Then, these HSV components are used to create an alternative RBG image that is reduced to 6 levels for channel. Compare the dithered image with an image obtained by reducing to 6 the 256 levels of the original image.

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

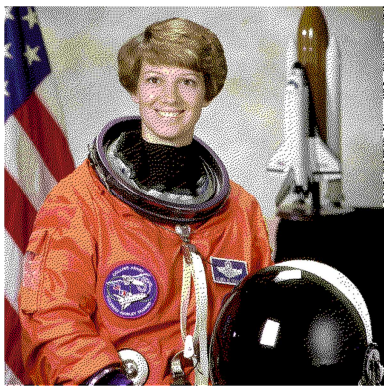


Figure 3.3: Color dithering

Lab #4: More on color and channel transformations

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

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

1. Calculate the following 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. Functions $x(\lambda), y(\lambda), z(\lambda)$ are available in a file posted on the course webpage. The color matching functions file is a space-separated[2] plain text with five columns: wavelength, $x(\lambda)$, $y(\lambda)$, $z(\lambda)$ and the emission spectrum of a lamp $E(\lambda)$. Function `numpy.loadtxt` may be useful. 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 means of the following linear relationship:

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

4. Finally, create a NxNx3 numpy array for displaying 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 histogram of the image 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 expected to develop a function for generating equalized images according to the following algorithm:

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

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

[2] Commas, semicolons, tabs are also used as delimeters

Write your code as a function for future reference.

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

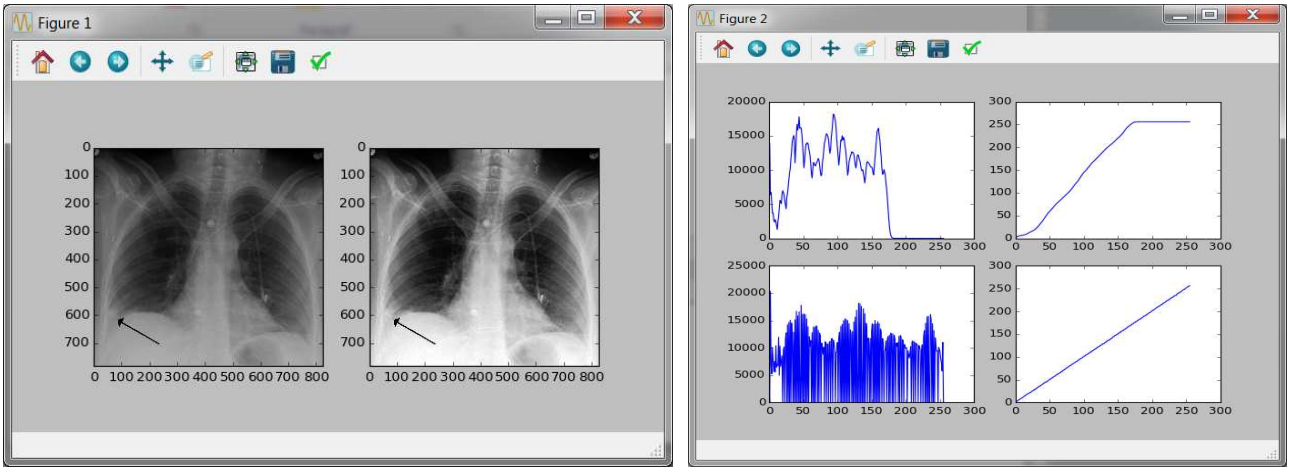
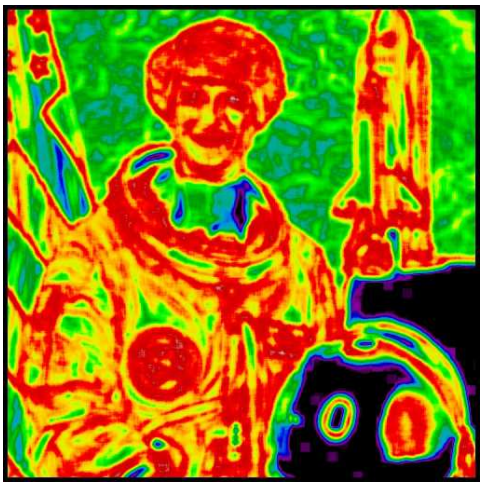


Figure 4.1: (a) original image, (b) equalized image. Figure 2: (a) histogram of the original image, (b) Cumulative histogram of the original image, (c) histogram of the equalized image and (d) cumulative histogram of the equalized image. Note the histogram of the equalized image is linear.

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 for encoding a signal. For a 256 gray-level image, the mathematical equation for entropy reads

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

where P_i is the probability associated to gray-level (i.e. times gray level i is present on the image / # pixels of the image. The local entropy at pixel [i,j], taking into account a MxM neighborhood 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 to 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 of 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 bits steganography

Steganography is a technique intended for hiding information within an image. This technique is used for watermarking in order to preserve copyright; used in combination with encryption provides an extra security layer. Please check the following paper for more insight about 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 steganographic technique: a secret message or image is encoded in the least significant bits of the host image

1. Let us first consider two 8-bit images. Ideally, both images are of the same # of pixels (this may help but is not compulsory – please consider function `scipy.misc.imresize`)[3]. 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 for encoding the secret image and the 4-most significant remain the same. The resulting image look very similar to the host image but a careful visual inspection may reveal the content of the secret image.
2. This technique may be refined easily. For instance, a color host image provides 24 bits of information. Using the two-least significant bits in each channel enables encoding 6-bits gray level secret images
3. If the # of pixels of the secret image is small in comparison with the size of the host image, the information can be scattered at random. Note that the seed of the random numbers generator is the key of this simple encryption.
4. This method can be easily expended to text messages since each character can be encoded as a 8 bit number (ASCII) or a 16-bit number (UTF-8). You may consider funcions `ord()` and `chr()`.

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

4.5. Visual encryption

A binary distribution B can be split in two random 2d-arrays using the XOR logic function. First, a random binary distribution A1 should be produced. Then, array A2 that fulfils A1^A2 = B is determined. Distributions A1 and A2 can be provided to two different recipients and thus, 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-planes sets in two random gray-level images (use `numpy.unpackbits` and `numpy.packbits`).
3. Generalize this method to 24-bit (color) images.

[3] Depending of the version used, this function might be deprecated.
Use `scipy.ndimage.zoom` or `skimage.transform.resize` instead.

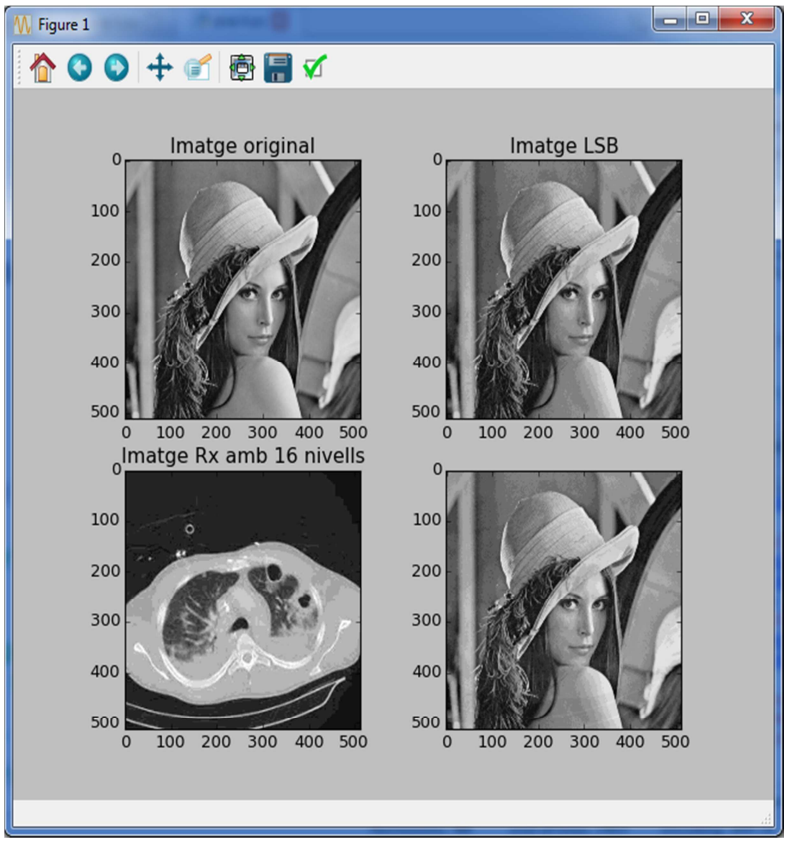


Figure 4.2: (a) 256 gray-level image original image, (b) 16 gray-level host image (c) 256 gray-level image to hide (d) host + hidden image.

Lab #5: Fourier transforms and spatial filtering.

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

5.1 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 Fourier transform 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 Fourier transform. 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 Fourier transform). Display the result.

5.2 Fourier series and filtering of spatial frequencies.

Using function `scipy.signal.square`, generate a 5Hz square wave with a sampling frequency of 500 samples per second (see the example in the documentation of the function). Calculate the 1D-TF (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 TF 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 in the interval `[-1/(2*T), 1/(2*T)].` Note that two cut-off frequencies can coexist if `N`≠`M`.

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

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))
```

5.3. Relative importance of amplitude and phase of the Fourier Transform

Now, explore how the information of 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 classical Oppenheim and Lim experiment [4] which consists in exchanging phase and amplitude of the FT of both images and

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

then inverse Fourier transform them. This exercise suggests how to design spatial filters in the Fourier domain.

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

5.4. Spatial filtering

Linear processing is based on the integral convolution product. Let *i(x,y)* and *f(x,y)* be two (discrete) functions that represents the image and the filter respectively. The processed image is obtained after 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 area 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, small and capital letters are used for distributions in image and Fourier planes respectively, i.e. $I=FT[i]$, $F=FT[f]$ and $P=FT[p]$.

5.4.1. 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()` and do not forget to normalize the resulting distribution before displaying the final image with `plt.imshow()`.

5.4.2. Laplacian filter[5]: Implement a function for $L(\rho) = \rho^2$. Display $L(\rho)$. Calculate $i * l = FT^{-1}[FT[i]L(\rho)]$.

5.4.3. 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 noise to your test image *i*(*x*,*y*). Use `numpy.random.rand`. Test the Gaussian filter with noisy images.

5.4.4. Butterworth filters.

Implement de Butterworth filter G(ρ);

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

$$G(\rho;\rho_c,n)=\sqrt{\frac{1}{1+(\rho/\rho_c)^{2n}}} \quad \text{with } 0\le\rho\le 1$$

where ρ is the radial coordinate in frequency space, and ρ_c ($0< \rho_c <1$) and n are parameters to be selected. Normalize and plot $G(\rho;\rho_c, n)$ for different values of n and ρ_c. Note that this filter can perform as a low-pass or a high-pass filter depending on the values of n. What is the meaning of ρ_c ? Calculate $i * b = FT^{-1}[FT[i]G(\rho;n)]$.

5.4.5. 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.

1. Detect the undesired frequencies. Calculate the FT of the image and display the amplitude in log scale.
2. Generate the filter and calculate the inverse Fourier transform.

You may use the following images:
http://img-service.com/overview/pics/chap8/clown.jpg or
https://www8.cs.umu.se/kurser/5DV015/VT09/handouts/images/mit_noise_periodic.jpg
http://www.upstate.edu/radiology/images/education/rsna/radiography/artifact/index_clip_image002_0010.jpg

5.5. Spatial filtering in image domain.

5.5.1. Linear convolution kernels

Calculate $c=i * f$ in the image domain.
Python provides the function `scipy.ndimage.filters.convolve`[6]. Note that the size of the kernel f can be 3x3, 5x5, 7x7, et cetera

Select an image and use the following filters:

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

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

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

pad the kernel to NxM pixels; take the modulus, normalize and display the result. Do the same for f_1 *and* f_2 . How do these filters relate to the Laplacian filter?

Convolve the following filters with your test images:

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

[6] Be careful with this function. If the image is `unit8`, it is very likely that convolution calculation surpasses the 255 limit. Please, cast the image value to double to avod normalization problems.

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

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

5.5.2. 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 in 45 degree increments through 8 compass directions: 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, | | is the absolute value, and $\mathbf{g}^{(z)}$ is a 3x3 convolution matrix defined as:

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

et cetera. Develop a script that implements this filter.

5.5.3. Salt and Pepper noise

Add a large amount of salt & pepper noise to your test image $i(x,y)$. Note that you may need to write a code to implement this kind of noise. Calculate the convolution of the corrupted image with $\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$. Instead of using the Gaussian filter use the median filter in a similar way as the adaptive threshold function you develop in Lab #3.

5.5.4. Roberts, Sobel and Prewitt filters

Non-linear edge detection operators 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 #6: K-means clustering in remote sensing

Clustering algorithms are aimed at forming 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 whee the data points are. 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:

$$\bigcup_i A_i = X$$

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

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

In other words, the pixels of the data images can be grouped in several disjointed sets according to certain predetermined rules. For instance, in this lab we are using two different images of an aerial view of the Barcelona airport[7]. The first one is a conventional RGB image whereas the second one is 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 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 in an arbitrary number of clusters.

Figure 6.1: (a) RGB and (b) IRG aerial images of the Airport.

K-Means[8] is a type of clustering algorithm that makes a hard clustering of the data points. It is based on the minimization of a so-called *functional* or *merit figure* aimed at minimizing the distance between each data point in a cluster and a point in it called *centroid* (or *representative*). 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:

[7] The orthoimages have been 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 on 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
Should you want to extract the airport images, consider the following figures [5800:5800+256, 3050:3050+512]

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

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

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

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

3. The coordinates of the new centroids would be 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 $\|\boldsymbol{c}_j(k+1) - \boldsymbol{c}_j(k)\| \leq \epsilon; j = 1,2,\dots,K$, or when a specific number of iterations have 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, et cetera) of the algorithm. Note that methods `fit()` and `predict()`should be used as well.

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

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

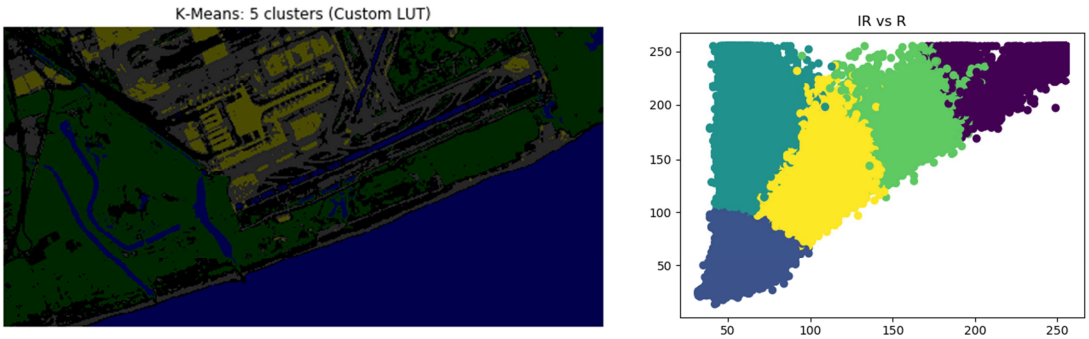


Figure 6.2: (a) K-means processed image, (b) IR vs R scatterplot.

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

Lab #7: Point spread function: spherical aberration and out-of-focus images. Image restoration filters.

A defocused image can be modelled as the convolution between the ideal, perfect image and the Point Spread Function (PSF). The figure shows different examples of PSFs combining out-of-focus and spherical aberration.

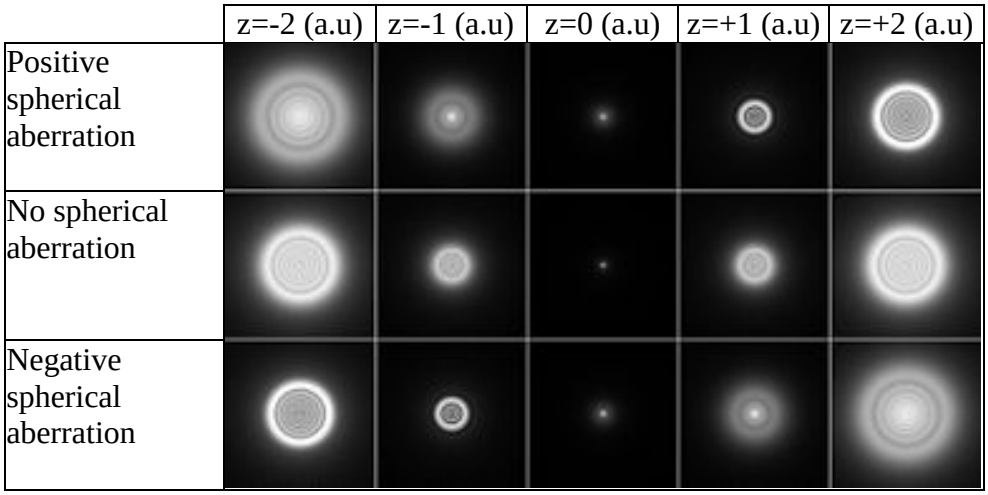


Figure 6.1: Source image: http://en.wikipedia.org/wiki/File:Spherical-aberration-disk.jpg

In paraxial geometrical optics, the out-of-focus PSF is approximated as the Fourier transform of a constant circle (the Airy disc), i.e. $h(x,y) = TF_{\lambda s}\left[\text{circ}\left(\frac{R_0}{R}\right)\right]$, where R is the polar coordinate and R_0 is the radius of the exit pupil of the instrument; s is the distance between the exit pupil and the image plane. The generalized theory of optical instruments shows how to calculate the exact PSF $h(x,y)$ at the image plane of the system when the device is affected by spherical aberration:

$$h(x,y) \propto \int_{PS} \exp\left(ikA_S\left(x_0^2+y_0^2\right)^2\right)\exp\left(2\pi i\left(x_0\frac{x}{\lambda s}+y_0\frac{y}{\lambda s}\right)\right)dx_0dy_0 = \\ = \text{TF}_{\lambda s}\left[\exp\left(ikA_S\left(x_0^2+y_0^2\right)^2\right)\text{circ}\, r_0\right], \quad (1)$$

where $W\left(x_0,y_0\right) = \exp\left(ikA_S\left(x_0^2+y_0^2\right)^2\right)$ is the wave aberration, $r_0 = \sqrt{x_0^2+y_0^2}$, and $k=2\pi/\lambda$ is the wavenumber. In this equation, coordinates $\left(x_0,y_0\right)$ are normalized to 1 at the radius of the exit pupil, i.e. $r_0 = \frac{R_0}{R}$. Note that for an aberrated-free system (A_S=0), Eq. (1) becomes $h(x,y) = TF_{\lambda s}\left[\text{circ}\left(\frac{R_0}{R}\right)\right]$. Finally, if the system is illuminated with incoherent (natural) light, the defocused image is calculated by means of Eq. (2):

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

7.1. Calculation of the PSF of an optical system with spherical aberration

The table summarizes the variables describing the optical system:

Spherical aberration A_s	26 λ
Radius of the exit pupil R_{PS}	17 mm
Length the window describing the exit pupil L	34 mm
Image distance s	316 mm
Wave-length λ	633 nm

1. Determine PSF $h(x,y)$ for a perfect system A_s=0 at the image formation plane:

- Use N=512 pixels; in order to improve resolution, calculate the Fourier transforms using *zero-padding* up to 2048x2048 pixels and then extract the 512x512 central part area of the Fourier transform. Why do we get a better resolution calculating the FT in this way?
- Using the Shannon theorem, demonstrate that the visualized area is $L_v = \lambda s \frac{N}{4L}$. Since the radius of the Airy disc is $r_A \simeq 0.61 \frac{\lambda s}{R_{\mathrm{PS}}}$, show that both results are compatible.

2. Now, determine the PSF $h(x,y)$ for an aberrated system A_s=26λ at the image formation plane

3. Calculate images d using the PSFs obtained in the previous sections using incoherent light. For the sake of brevity take the same L_v for both the PSFs and the image; otherwise it would be required to adapt the size of the image and the PSF (why?). Save the defocused distribution image as an 8-bit image.

7.2. Image reconstruction: inverse and least squares filters

4. Image reconstruction filters can be used in order to minimize the effects of aberrations or defocusing. Note that Eq. (2) is written as IH in Fourier space, where H=TF[$|h|^2$]. It seems the non-aberrated image I could be restored if the calculation IH/H is performed. Nevertheless, $1/H$ is not well behaved in some frequencies. To avoid division-by-zero errors, the inverse filter is defined as $F_{\mathrm{I}} = 1/(H+k)$ where k is a constant value selected in such a way that minimizes noise in the reconstructed image. The restored image d_r is obtained by means of Eq. (3):

$$d_r = \mathrm{TF}^{-1}\left[IHF_{\mathrm{I}}\right] = \mathrm{TF}^{-1}\left[IH\frac{1}{H+k}\right] \qquad (3)$$

Use this filter with the defocused images obtained in the previous sections.

5. The least-squares filter is defined as

$$F_{\mathrm{MQ}} = \frac{H^*}{|H|^2 + k\left(u^2+v^2\right)} \qquad (4)$$

where k is a constant to be determined and u and v are the spatial frequencies. Use the defocused images obtained in the previous section. Compare these results with those obtained with the inverse filter.

7.3. Out-of-focus images

Defocusing can be modelled using Fresnel diffraction theory. The PSF at plane z is calculated by means of Eq. (5)

$$h(x,y,z) = \mathrm{TF}^{-1}\left[\mathrm{TF}\left[h(x,y)\right]\exp\left(i\pi\lambda z\left(u^2+v^2\right)\right)\right] \qquad (5)$$

where $h(x,y)$ was calculated by means of Eq. (1). Again, the Shannon theorem has to be used for calculating the corresponding Nyquist frequency. Note that in Eq, (1) the coordinates are λs – scaled, but in the present case this is not required.

Determine $h(x,y,z=2\text{ mm})$ for $A_s=0$ and $A_s=26\lambda$ and calculate the defocused images in these cases. Restore these images with the inverse and the least-squares filter.

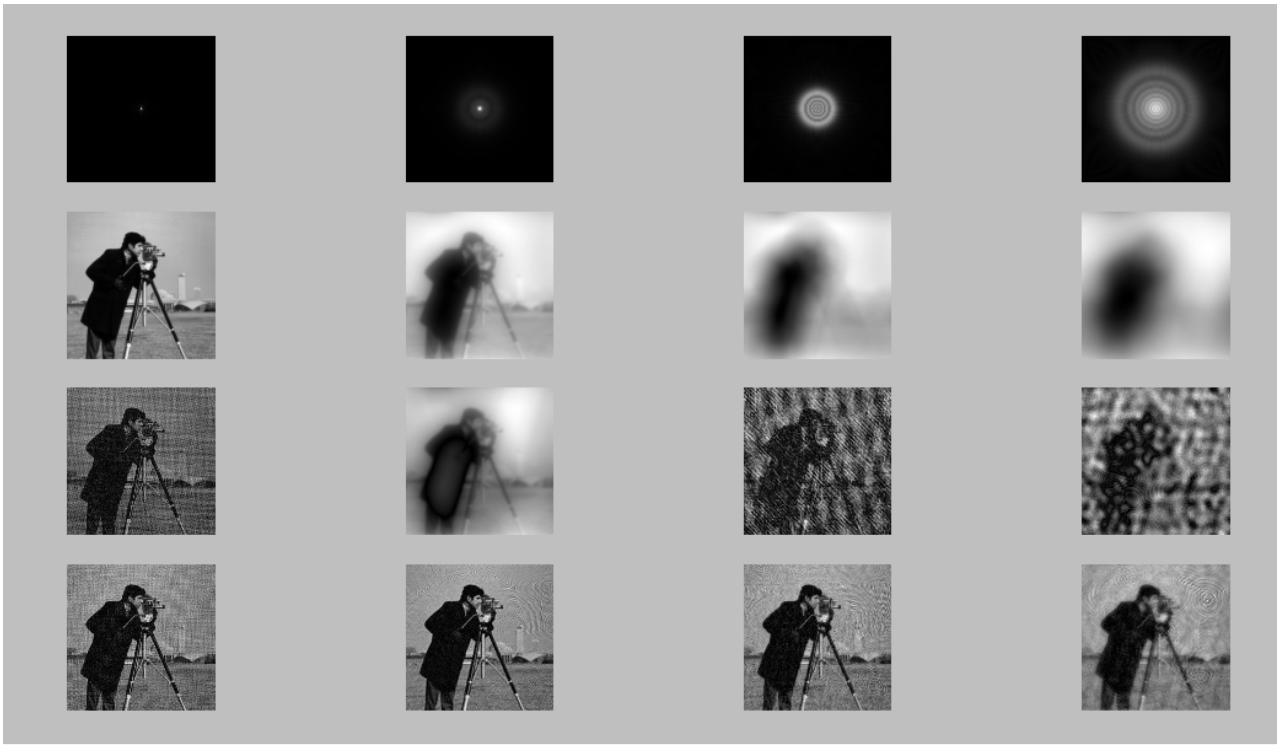
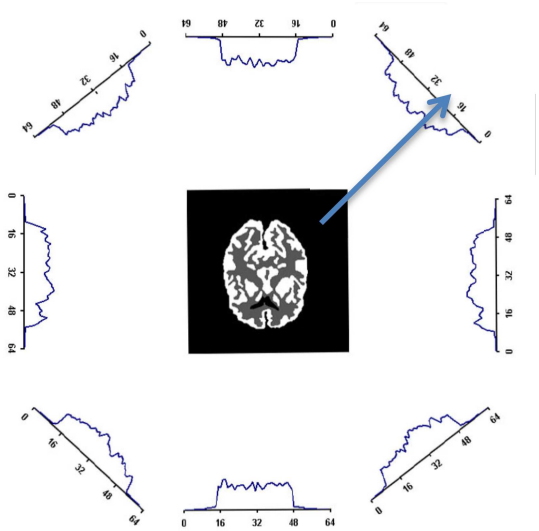


Figure 7.2.: Summary of results: first row, PSFs: (from left to right): ideal system, with spherical aberration, with defocusing, combined effect aberration + defocusing. Second row: convolved images. Third row: reconstructed images with the inverse filter. Fourth row: reconstructed images with the least-squares filter.

Lab #8: Radom Transforms and the Projection-Slide Theorem.

Computer tomography (CT) is an imaging technique based on taking images of a 3D object from different slices. Then, the 2D objects are projected and recorded on linear detectors. Later on, this information is subsequently recombined to produce a 3D model of the object. X-rays are the illumination source in CT. The same technology principle is used in PET (positron emission tomography) or SPECT (single-photon emission computed tomography).



In this introductory lab we focus on 2D objects to be projected in 1D-arrays. The combination of the information projected in a single 2D-array is known as the sinogram. The Radom Transform (RT) is the operator that converts a conventional image in a sinogram.

CT aims to determine the original object from the information stored in the sinogram. The inversion reconstruction procedure (inverse Fourier transform) can be carried out by means of the Projection-Slide Theorem and the Filtered Back-Propagation method. The latter is implemented in functions `skimage.transform.radon` and `skimage.transform.iradom`. Other well-known inversion techniques are the Simultaneous Algebraic Reconstruction Technique (SART, also implemented in `skimage.transform`) and the Maximum Likelihood Estimation algorithm. In this lab we analyze the first two methods.

Let f(x,y) be a 2D gray level distribution representing the object to be processed. For testing purposes, standard images such as the Shepp-Logan phantom are used. At $\theta=0$, the projected information of $f(x,y)$ on the detector is $p_{\theta=0}(y) = \int f(x,y)dx$. Since the orientation of axis x-y is arbitrary, this result is valid for any rotation of axis x-y around the origin. The projection on direction θ reads $p_\theta(r)$. Then, the Fourier transform of $f(x,y)$ on the axis $u=0$ is

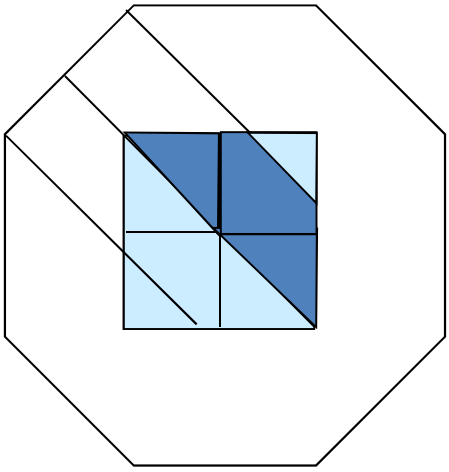
$$F(0,v) = \iint f(x,y)\exp(-2\pi ivy)dxdy = \int p_{\theta=0}(y)\exp(-2\pi ivy)dy = P_{\theta=0}(v)$$

i.e., the Fourier transform of the projection $p_{\theta=0}(y)$. Again, this result is independent of direction θ, and thus $P_\theta(R)$ = FT[$p_\theta(r)$]. At the end of the day, the Fourier transform F(u,v) can be inferred from the projections set $\{P_\theta(R)\}$. This result is known as the Projection-Slide theorem (PST). In summary, the steps for calculating the Inverse Radom transform using the PST are:

1. The Fourier set $\{P_\theta(R)\}$ is obtained from $\{p_\theta(r)\}$ [a 1D-FT has to be calculated for every $p_\theta(r)$].
2. Fourier transform $F(u,v)$ is generated from the angular-based set $\{P_\theta(R)\}$ (polar to Cartesian transformation).
3. $f(x,y)$ is reconstructed from the inverse Fourier transform $F(u,v)$.

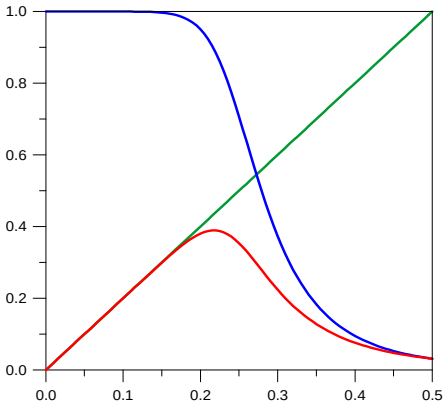
Despite the procedure is simple, conversion form the polar-based geometry sinogram to Cartesian coordinates is not simple due to interpolation and sampling issues.

The Filtered Back-Propagation algorithm takes also into account the contribution of each pixel (voxel) of the object during the propagation (direct Radom transform) and back-propagation (inverse Radom transform). Since the number of detectors used in CT is small, CT images display a low number of pixels.. For these reasons the contribution of each pixel of the object has to be weighted according to the polar geometry of the problem.



Note that since the final Fourier transform is carried in polar coordinates, the Jacobian R has to be taken into account, i.e. $dxdy = R\,drd\theta$. This term is known as the *ramp kernel*.

Usually, projections are affected by clutter. Since noise can be considered as high frequency information, it is amplified by the Jacobian term. For this reason, in addition to the ramp kernel (green line), filters that remove high frequency contributions while keeping low frequency information are used (blue curve). The resulting filter is shown in red.



Among different possible kernels, the following are widely used

1. Shepp-Logan $k(w) = \text{sinc}\dfrac{w}{2w_c}$

2. cosine / Hanning / Hamming $k(w) = C_1 + C_2 \cos\dfrac{\pi w}{w_c}$

Have a look at the documentation of function `skimage.transform.iradom` for details.

Task: Using the `radom` and `iradom` pair, propagate and back-propagate the Shepp-Logan phantom using different kernels. Add a certain amount of Poisson noise to the sinogram. Display the results.

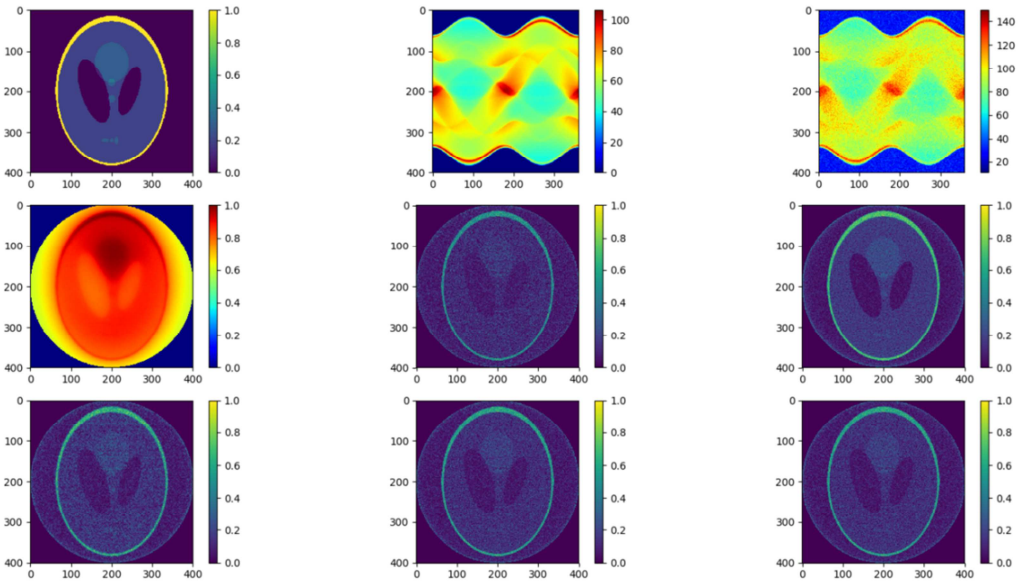


Figure 8.1: Examples of 2D-computer tomography simulations. (a) Shepp-Logan phantom; (b) Sinogram of (a); (c) sinogram affected by Poisson noise; (d) Inverse Radom Transform without using any kernel; (e-i) Inverse Radom Transform using the following kernels: ramp, Shepp-Logan, cosine, Hamming, Hanning.