Image Processing and Computer Vision

Artur Carnicer
artur.carnicer@ub.edu
Updated February 8th, 2022

Program

Lab #1: Python concepts for image processing

Intensity transformations

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

Lab #3: Image binarization and the error diffusion algorithm.

Lab #4: More on color and channel transformations (Equalization, entropy, steganography, encryption).

Fourier-based filtering

Lab #5: Fourier transforms and spatial filtering.

Lab #6: Point–spread functions and image restoration filters.

Lab #7: Computer tomography. Radon transforms and the Projection-Slide theorem.

Machine learning

Lab #8: K-means clustering.

Project #10: Automatic diagnostic using an X-ray images dataset: image classification using machine learning.

Schedule

February: 14, 18, 21, 25, 28
March: 4, 7, 11, 14, 18, 21, 25, 28
Abril: 1, 4, 8, 22, 25, 29
May: 2, 6, 9, 13, 16, 20

3 ECTS: 75 (up to 90) hours
Lectures: 48 hours
Personal work ~ 27 - 42 hours

Grading

- Midterm exam: 20 % (April 1st, 2022).
- Final exam: 40% (June 2nd, 2022; September 7th, 2022).
- Computational project: 40% (July 4th, 2022; September 1st, 2022.).

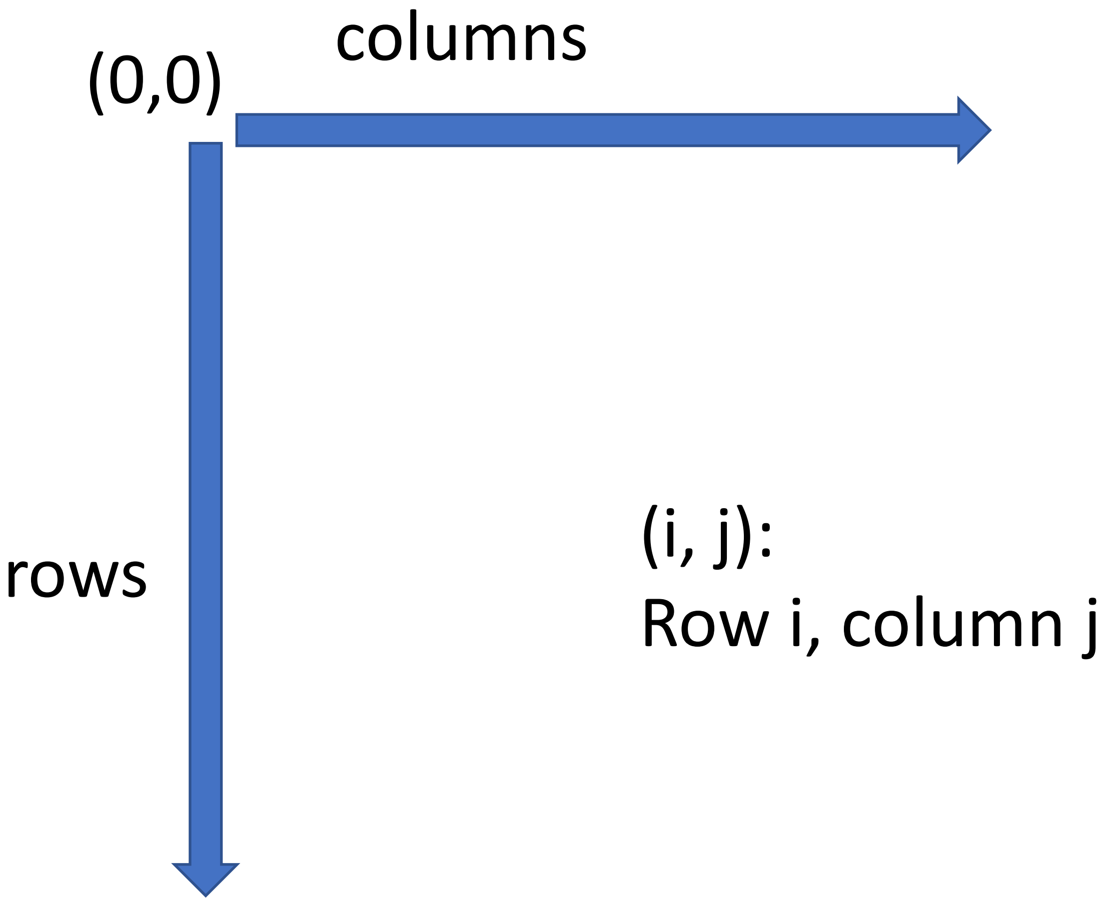
Case 1: Midterm exam (20%) + Final exam (40%) + Project (40%)

Case 2: Final exam (60%) + Project (40%)

Final grade = max(case1, case2)

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

Images:
coordinate convention



Use of the : operator

- within an array, you can select part of it using begin: end: step
- if begin =0, end = M-1 and step = 1, then begin: end: step   :
- if step = -1, the order of the array is inverted

Display

- Images are displayed using the RGB model
- Each channel (R, G or B) ranges from 0 (minimum) to 255 (maximum energy).
- A grey level image uses the three channels, but the information is the same for all of them

Memory. Let im be the stored image

MxN (gray level) images:

- np.uint8: The maximum value is 255
- np.double: The image is displayed taking im.max()

MxNx3 and MxNx4 (color images):

- np.uint8: The image is displayed as it is
- np.double: im **must** be normalized to 1. Otherwise, it will not be properly displayed
- Experiment with script `normalization_issues.py`

matplotlib.pyplot.imshow

`matplotlib.pyplot.imshow`(*X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=None, extent=None, shape=<deprecated parameter>, filternorm=1, filterrad=4.0, imlim=<deprecated parameter>, resample=None, url=None, *, data=None, **kwargs*) [source]

Display an image, i.e. data on a 2D regular raster.

Parameters:

X : array-like or PIL image

The image data. Supported array shapes are:

- (M, N): an image with scalar data. The data is visualized using a colormap.
- (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
- (M, N, 4): an image with RGBA values (0-1 float or 0-255 int), i.e. including transparency.

The first two dimensions (M, N) define the rows and columns of the image.

Out-of-range RGB(A) values are clipped.

cmap : str or `Colormap`, optional

The Colormap instance or registered colormap name used to map scalar

Quick search

Go

Table of Contents

Related Topics

Show Page Source

Gray level images from color images

- Average: $M = (R + G + B) / 3$

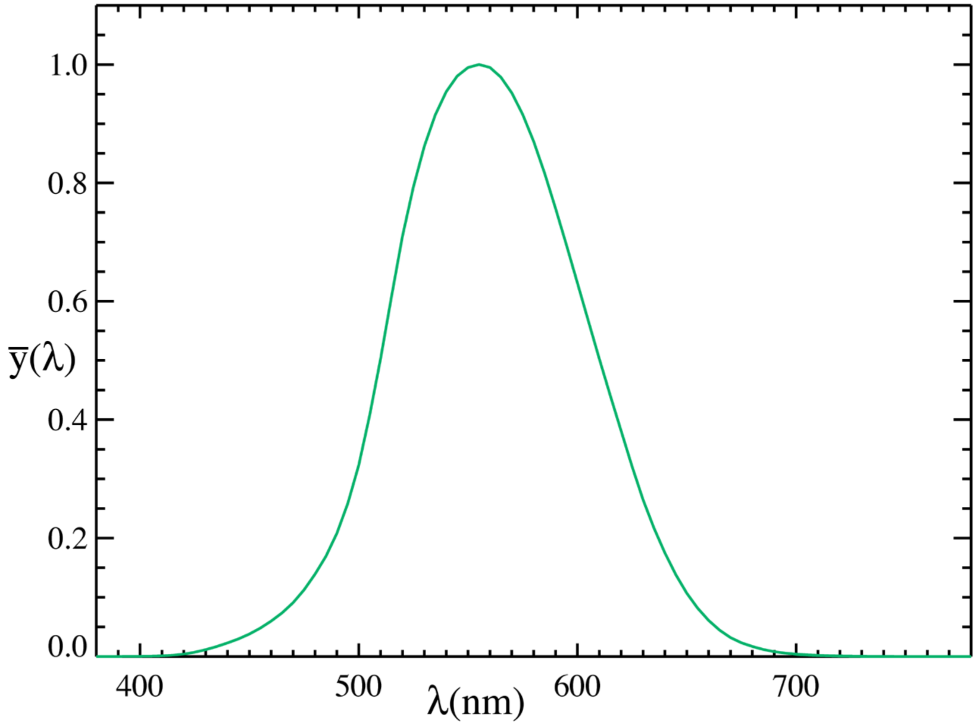
  Note that $R + G + B > 255$

In general, calculations should be carried out using real numbers (Why?)

- Luminance:

$L = 0.299 * R + 0.587 * G + 0.114 * B$

L is the gray level image obtained from a color one, according to the human luminosity function (in well-lit conditions)



1931 CIE photopic luminosity function. https://en.wikipedia.org/wiki/File:CIE_1931_Luminosity.png

Normalization

- np.uint8 images do not require normalization. But the 255 limit cannot be exceeded after calculations
- In general, real-valued arrays are unbounded. But they should be normalized to 1 before they can be displayed with `plt.imshow()`. Use

```
im = im / im.max()
```

- Conversion from np.uint8 (imc) to np.float64 (imf) can be tricky. Very often, integer-valued images (imc) use the full np.uint8 range. But to be on the safe side use the following rule of thumb:

```
imf = imc / 255.
```

- Maximum values for integers: `255 = 2**8 - 1, 65535 = 2**16 -1.`

Look-up tables (LUT).

- a LUT is a function that relates the processed gray level g' as a function of the original gray kevel g, i.e. gl' = f(gl). They are used to modify the contrast of an image.
- Colormap: an arbitrary color value is assigned to each g:

$$r = f_R(gl),\ g = f_G(gl),\ b = f_B(gl)$$

- LUTs might be used with color images as well:

$$r' = f_R(r),\ g' = f_G(g),\ b' = f_B(b).\ \text{Very often, } f_R = f_G = f_B.$$

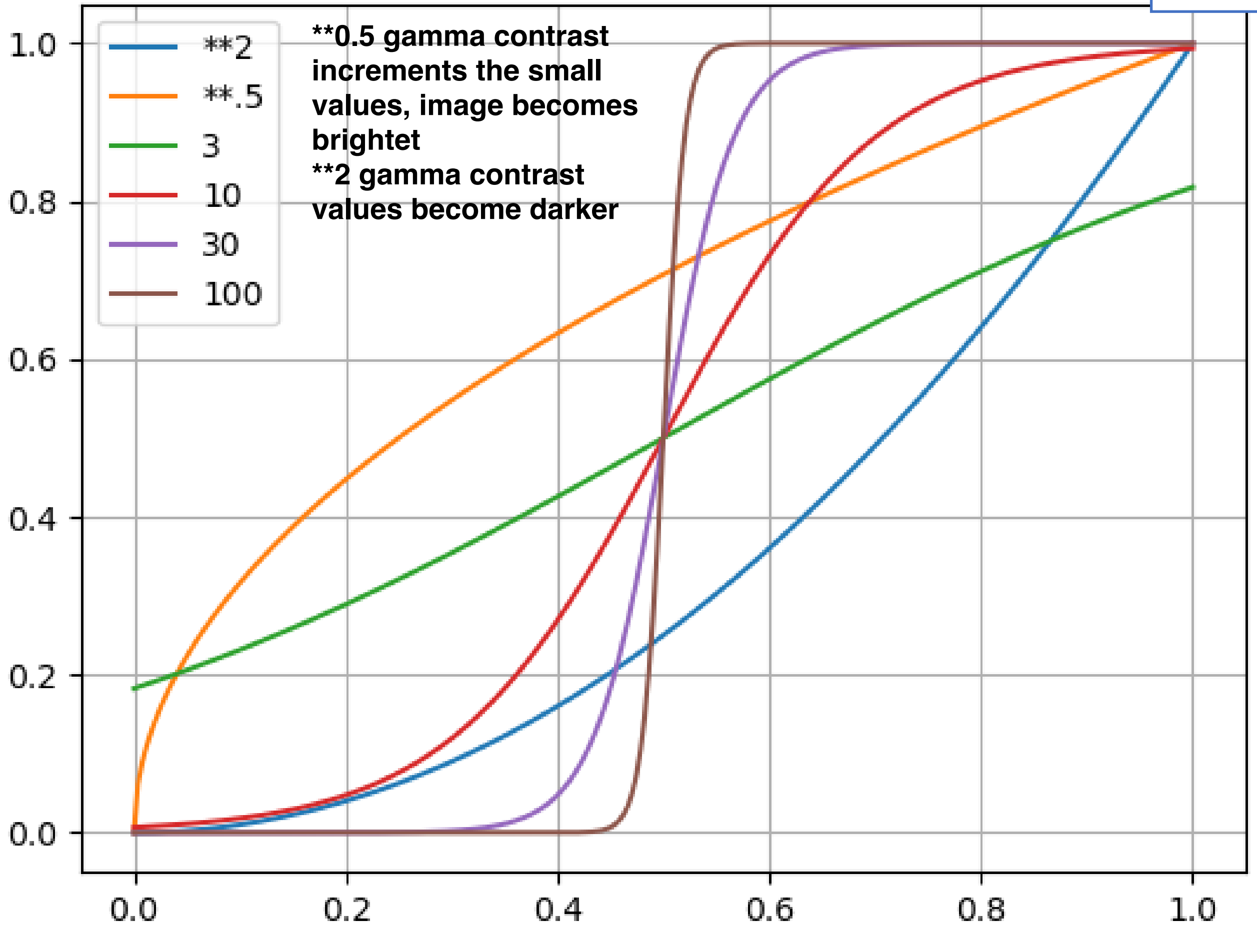
- Examples (change g by r, g or b when necessary)
  - Inversion of contrast: gl' = 255 – gl' (or 1 –gl')
  - Linear: gl' = m gl + n
  - Gamma: $gl' = gl^{Y}$. Cases: $0<Y<1$, $1<Y<\infty$ ($Y = 1$, g' = g)
  - Logistic function $gl' = gl_{max} / (1 + \exp(-k(gl-gl_0)))$
  - Binarization gl' = 255(1) if g > threshold, g'=0 otherwise.
- In general, calculations should be carried out using real numbers

Color map: gl ——> RGB

LUT: gl ——> gl'
RGB ——— R', G', B'

Look-up tables (LUT). Examples

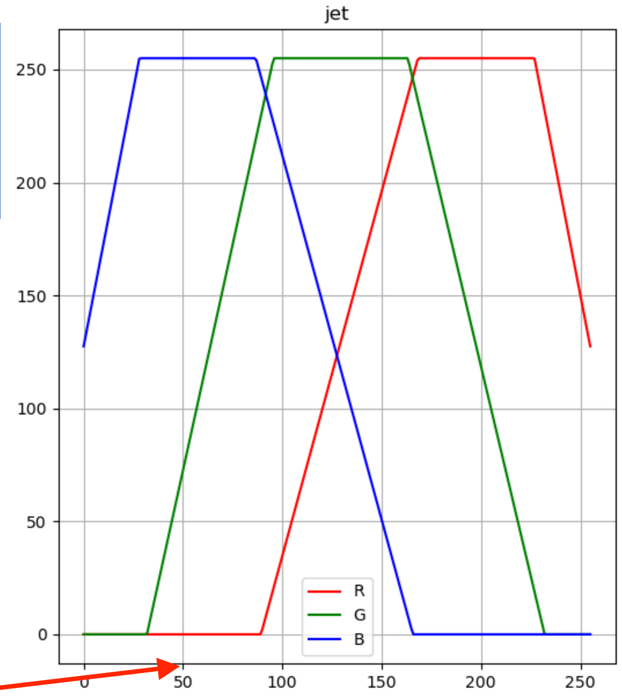
```
from matplotlib import cm
jetv = cm.jet(range(256))
virv = cm.viridis(range(256))
```



Gamma and logistic

LUT: transformation of an image on each 2d array channels. Or from gl (gray level) to gl'.

Jet

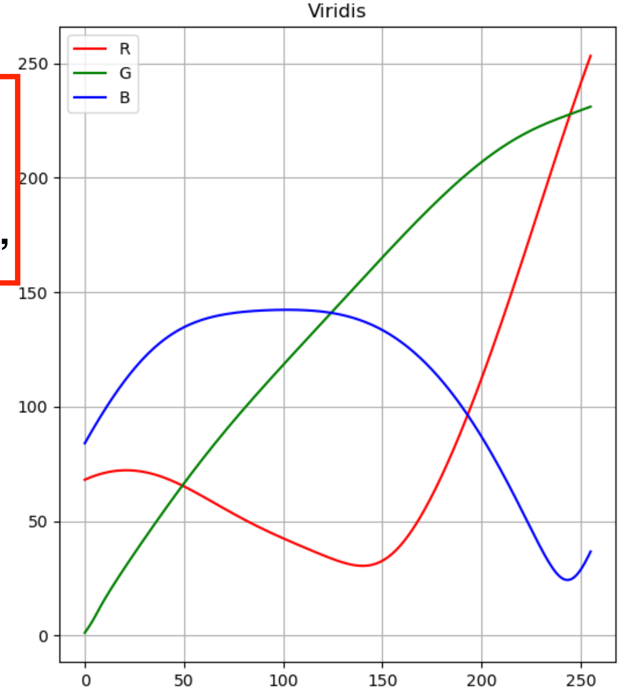


The value of 50 in a one channel (2d array) is mapped to 0R 75G 255B in order to implement a colormap

Color map: gl ——> RGB

LUT: gl ——> gl'
RGB ——— R', G', B'

Viridis



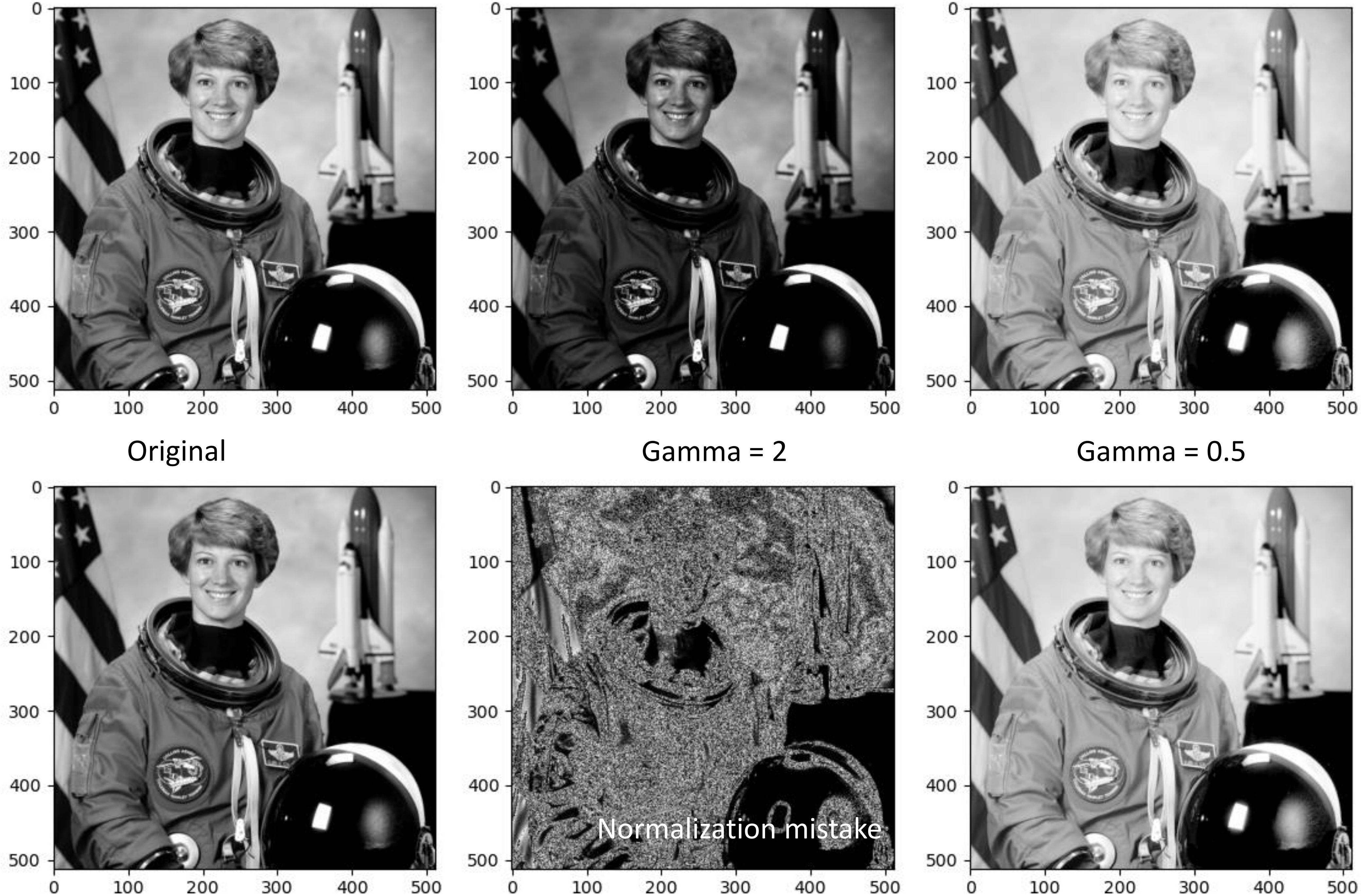

Original
Gamma = 2
Gamma = 0.5
Normalization mistake

Image comparison

Several assessment methods are provided in
https://scikit-image.org/docs/dev/api/skimage.metrics.html

skimage.metrics.mean_squared_error

- Not normalized. Simple to calculate.
- MSE=0 means both images are identical

```
MSE=((im1-im2)**2).sum()/im1.size
```

skimage.metrics.structural_similarity

- Is a perceptual quality measures, including connections to human visual neurobiology and perception, and direct validation of the index against human subject ratings. https://en.wikipedia.org/wiki/Structural_similarity
- Normalized 0 < SSIM < 1
- SSIM=1 means both images are identical

$$\mathrm{SSIM} = \frac{\left(2\mu_1\mu_2 + c_1\right)\left(2\sigma_{12} + c_2\right)}{\left(\mu_1^2\mu_2^2 + c_1^2\right)\left(\sigma_1^2 + \sigma_2^2 + c_2^2\right)}$$

$$c_1 = k_1\left(2^n - 1\right) \qquad \text{with } k_1 = 0.01 \text{ and } n = 8 \text{ (bits)}$$

$$c_2 = k_2\left(2^n - 1\right) \qquad \text{with } k_2 = 0.03 \text{ i and} = 8 \text{ (bits)}$$

Lab 3: Binarization

Pros:

- Very useful LUT for gray level images.
- Weight reduction. Just 1 bit per pixel.
- Straightforward calculation: imb = im > th

Cons:

- Loss of visual information

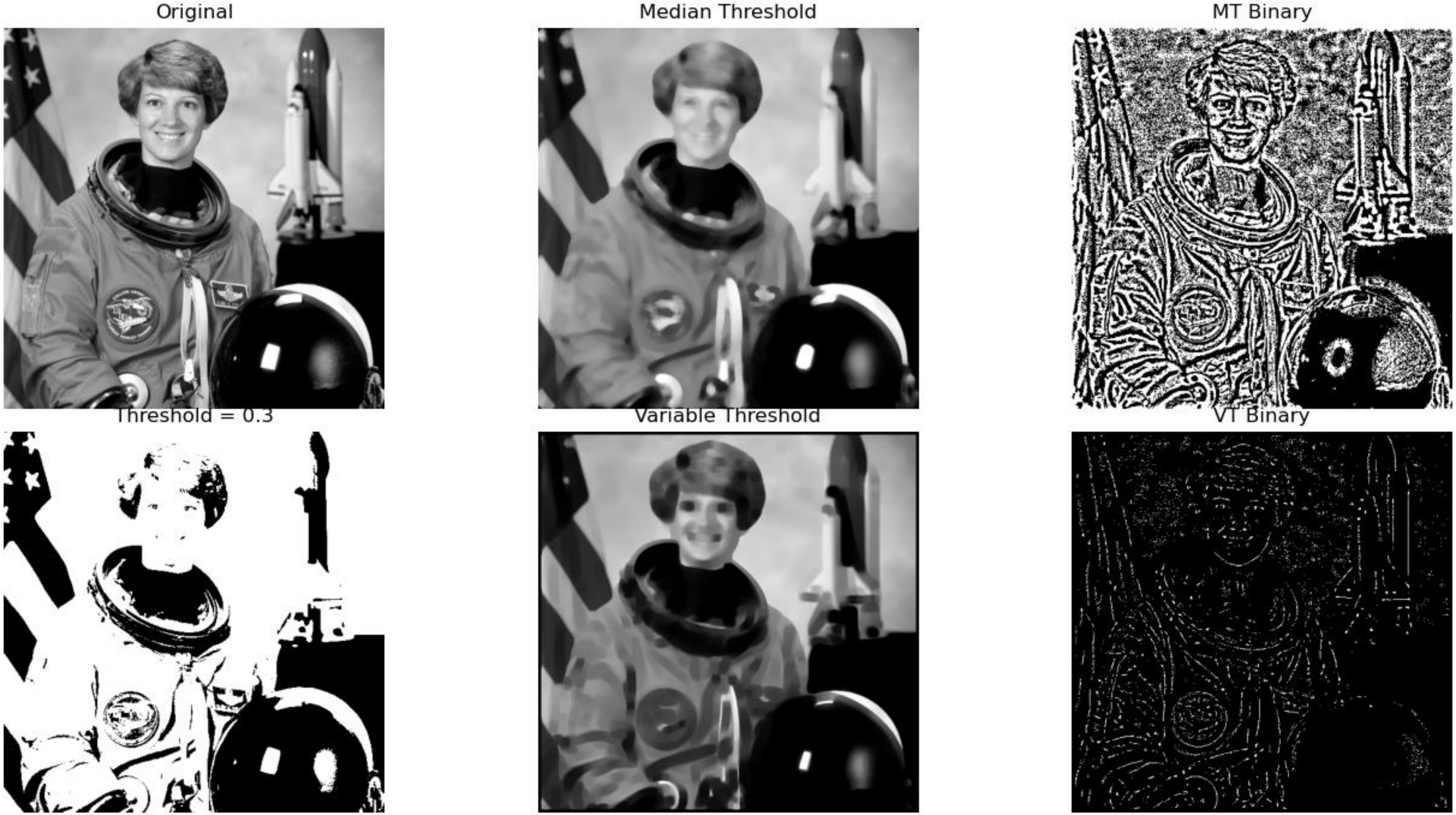
Possible approaches:

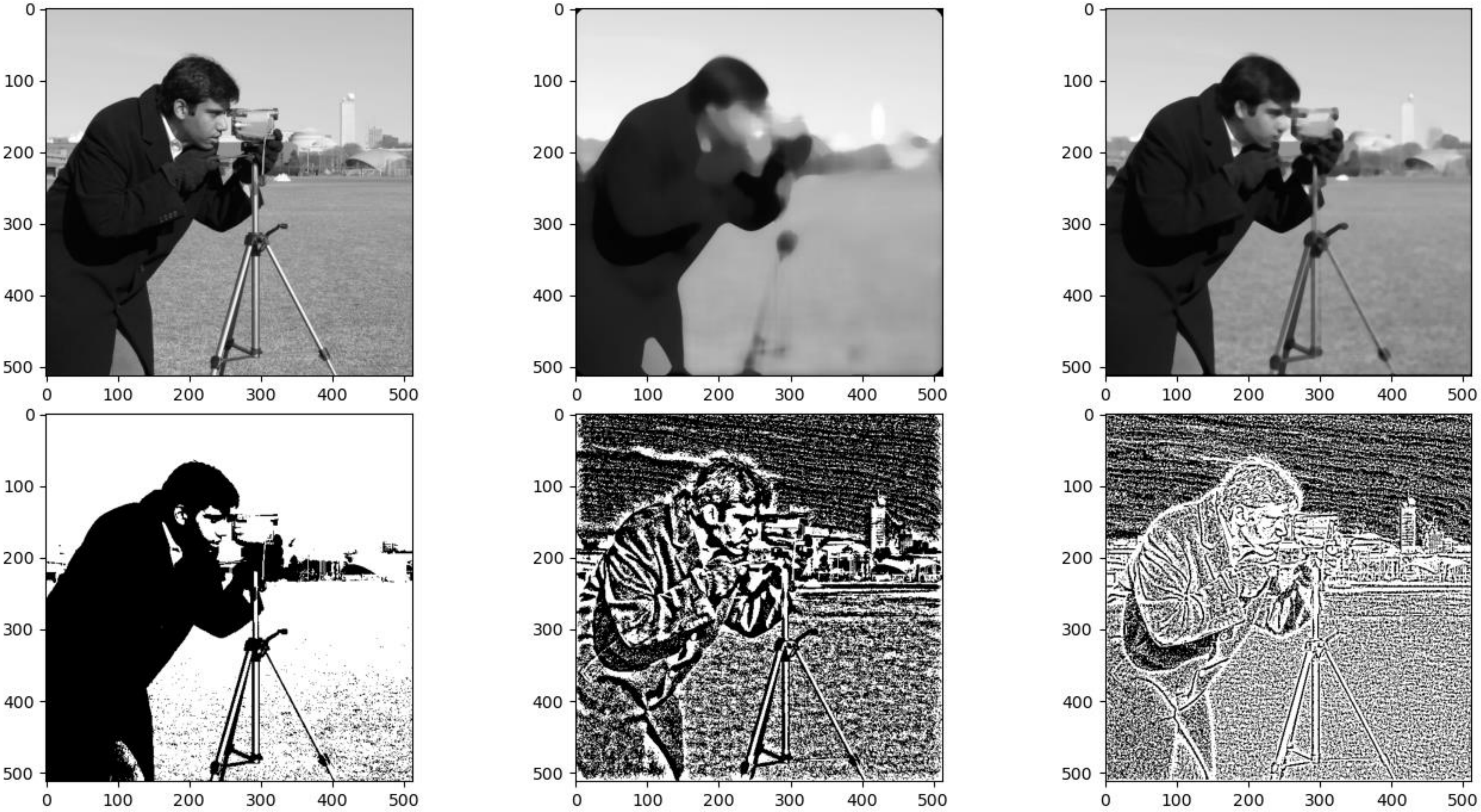
- Determine optimum threshold (e.g.: Otsu method)
- Use of local thresholds (calculate the median of a neighborhood)
- Error diffusion (Floyd-Steinberg) algorithm


Threshold = 0.1
Threshold = 0.2
Threshold = 0.3
Threshold = 0.4
Threshold = 0.5
Threshold = 0.6
Threshold = 0.7
Threshold = 0.8
Threshold = 0.9

Adaptive threshold

- A single threshold for a complete image cannot be appropriate.
- Thresholds are calculated according to the statistics of the neighbor pixels window.
- Possible windows sizes: 3x3, 5x5, 7x7, … The processed pixels is set in the center.
- We calculated the median / mean / … of the neighborhood window. This value becomes the local threshold
- The local median according to a certain window can be calculated using `scipy.signal.medfilt`
- `scipy.signal.order_filtre` can be used to set local variable thresholds.


Original
Median Threshold
MT Binary
Threshold = 0.3
Variable Threshold
VT Binary


0 100 200 300 400 500

mode : *{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*

The *mode* parameter determines how the input array is extended beyond its boundaries. Default is 'reflect'. Behavior for each valid value is as follows:

'reflect' (*d c b a | a b c d | d c b a*)

The input is extended by reflecting about the edge of the last pixel.

'constant' (*k k k k | a b c d | k k k k*)

The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

'nearest' (*a a a a | a b c d | d d d d*)

The input is extended by replicating the last pixel.

'mirror' (*d c b | a b c d | c b a*)

The input is extended by reflecting about the center of the last pixel.

'wrap' (*a b c d | a b c d | a b c d*)

The input is extended by wrapping around to the opposite edge.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.convolve.html

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.

We ignore those pixels that cannot be calculated (values are set to zero)

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):

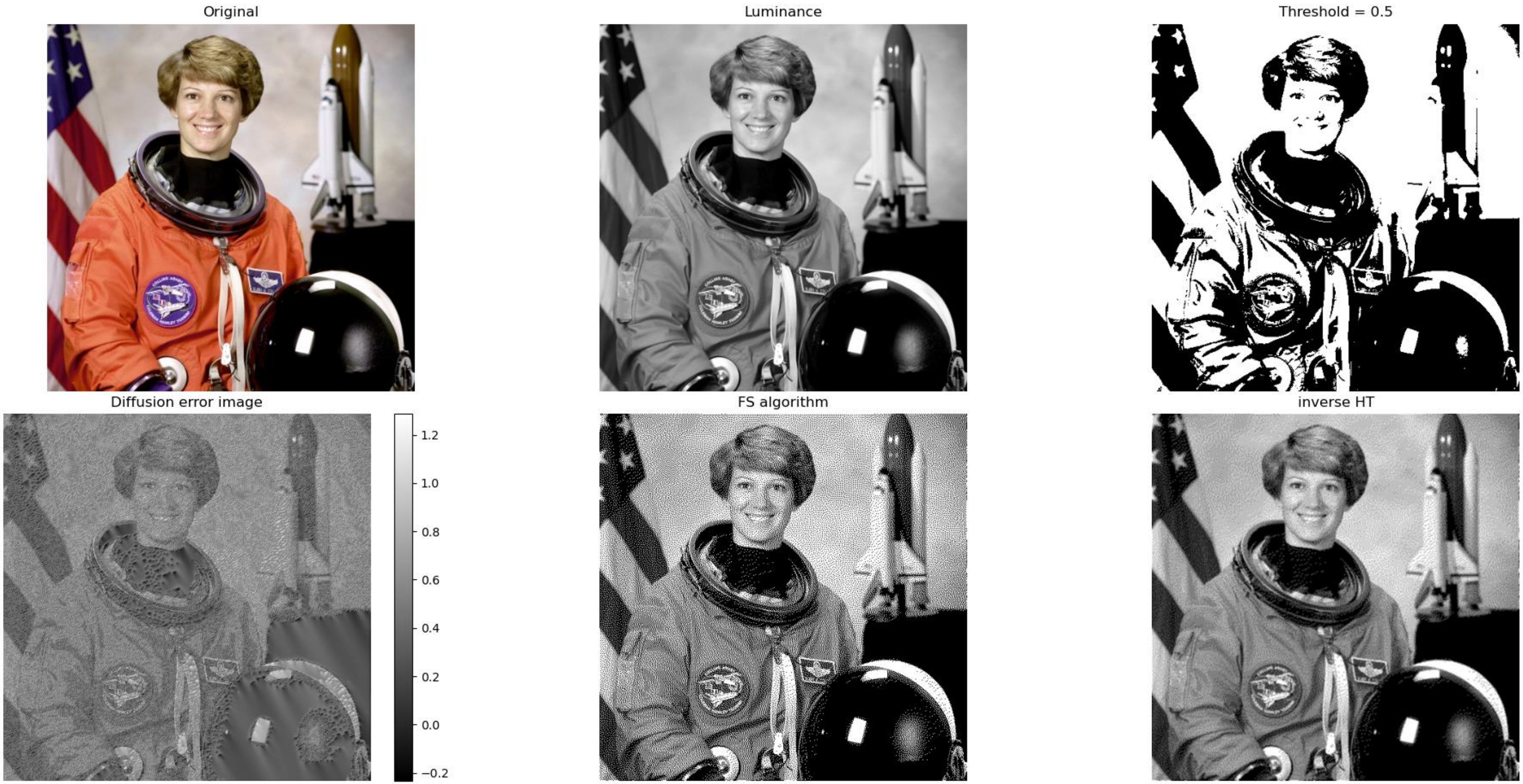
$$\mathbf{FS} = \frac{1}{16}\begin{pmatrix} 0 & 0 & 0 \\ 0 & p & 7 \\ 3 & 5 & 1 \end{pmatrix} \qquad \mathbf{JJN} = \frac{1}{48}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & p & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{pmatrix}.$$

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.


Original
Luminance
Threshold = 0.5
Diffusion error image
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
FS algorithm
inverse HT

Indexed color

Color reduction: from 256x256x256 to 6x6x6 colors

- Only several values are considered: (e.g.: 0, 51, 102, 153, 204, 255). Other possibilities might be considered as well.

- 6x6x6 color images can be stored in a single 8-bit channel because only 216 possible values are necessary.
- An index table is required to provide a four-column table that relates the integer values in the file (index) with the displayed color.
- this table can be understood as a LUT that describes the color-map used to transform the gray level distribution into an approximated true color image.

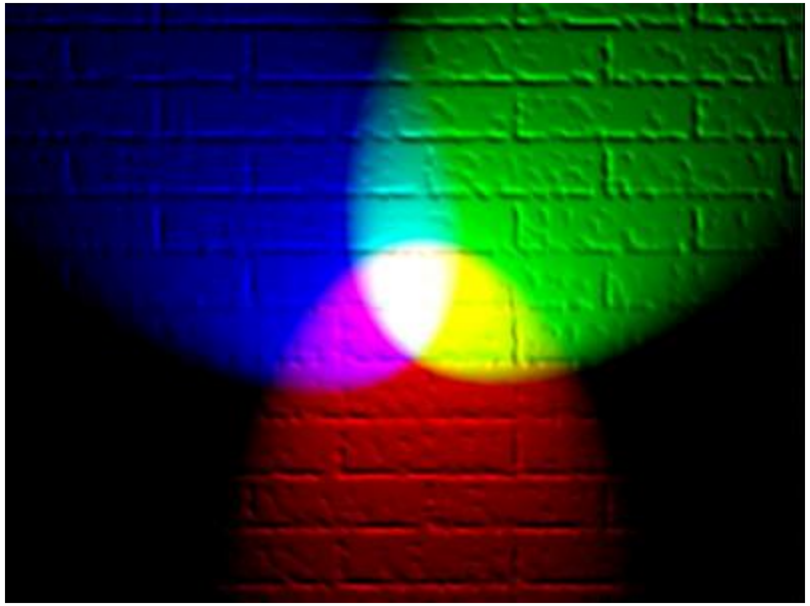
Indexed color table. Example.

File stored gray level	**Displayed colors** R	 G	 B
0	0	0	0
1	51	0	0
2	0	51	0
3	0	0	51
4	102	0	0
…			
215	255	255	255
216	0	0	0
…			
255	0	0	0

Lab #4. More on color and intensity transformations.

1. RGB coordinates from spectrum data. The CIE 1931 color model.
2. Histogram equalization.
3. Image entropy.
4. Least significant bit steganography.
5. Visual encryption.

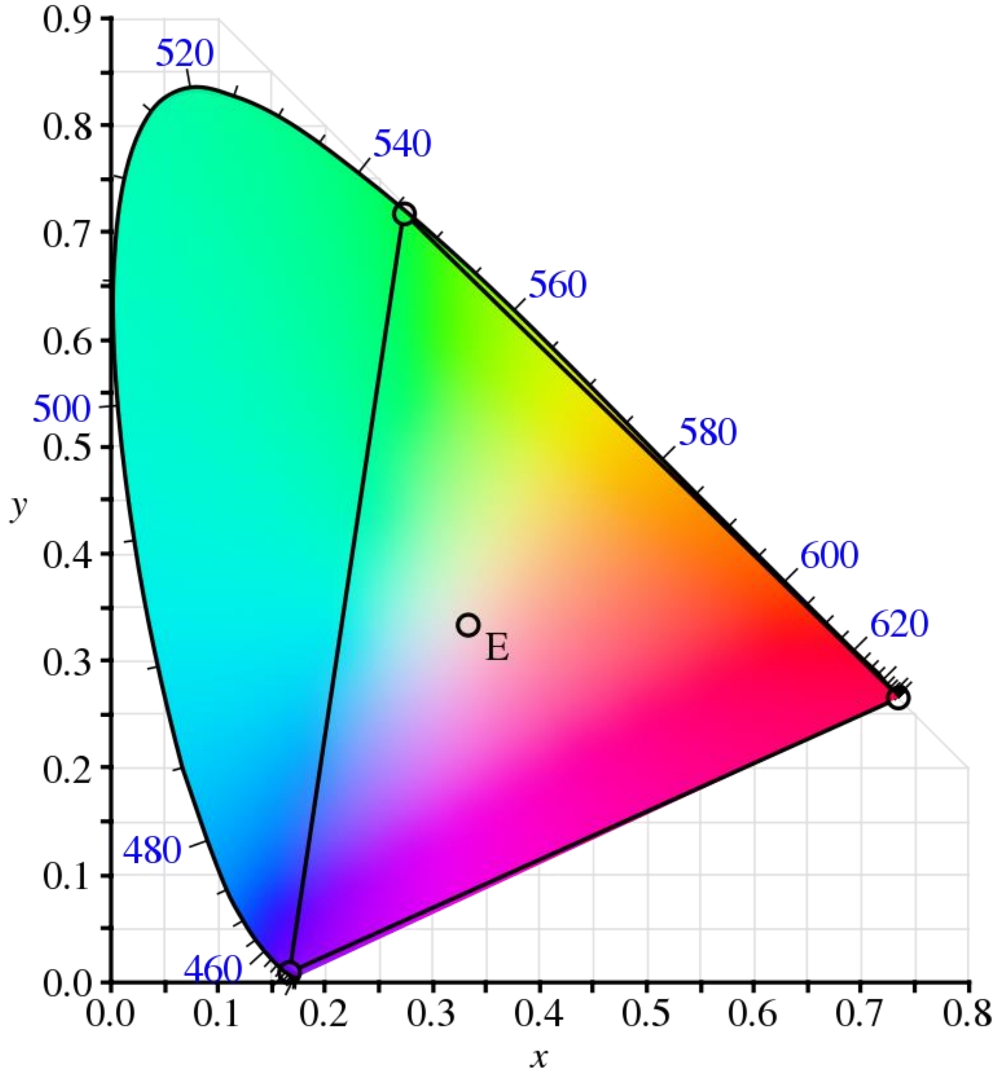
Color models: RGB, CIE 1931, HSV



https://en.wikipedia.org/wiki/RGB_color_model#/media/File:RGB_illumination.jpg

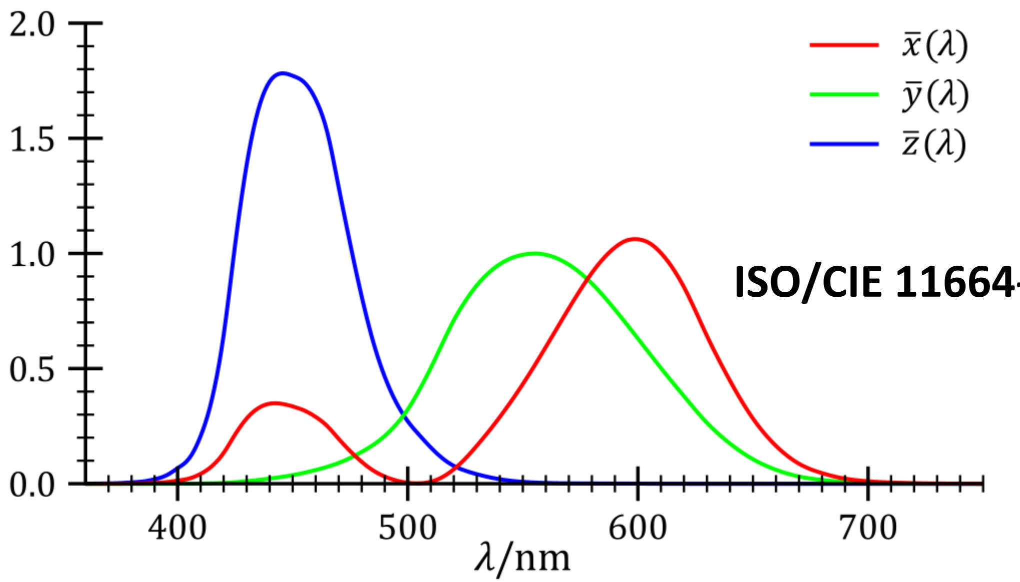
Each primary color ranges from 0 to 255
These values are related to the display intensity (e.g.: LED, LCD)

How RGB coordinates are related to physical measures?



https://en.wikipedia.org/wiki/CIE_1931_color_space#/media/File:CIE1931xy_CIERGB.svg

It is said that color is a property related to the wavelength λ (or the spectrum E(λ)). How are related E(λ) and RGB values?



ISO/CIE 11664-4:2019

The color matching functions x(λ), y(λ), and z(λ), relate the weight of the spectrum with the spectral sensitivity of the observer to calculate CIE coordinates X, Y and Z.

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

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

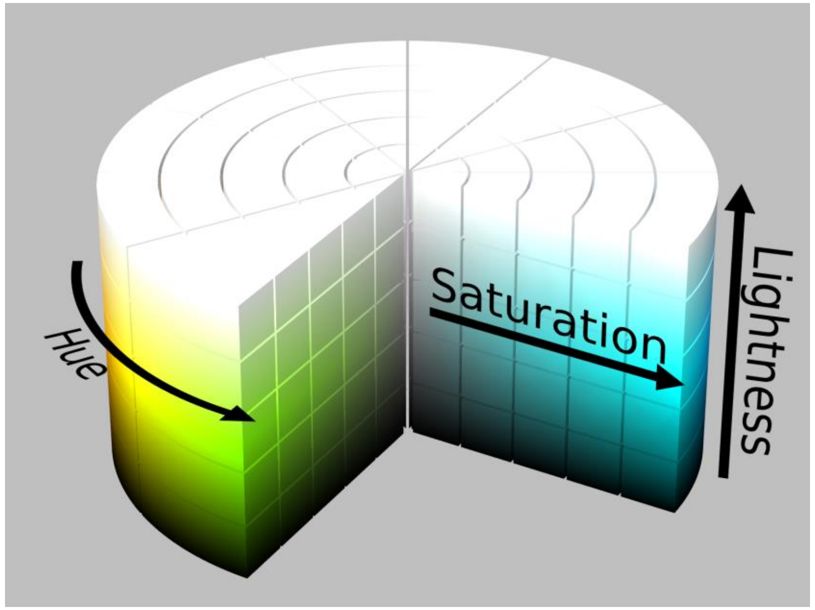
Use `scipy.integrate.simps`

Finally, XYZ and RGB are related by a linear transformation

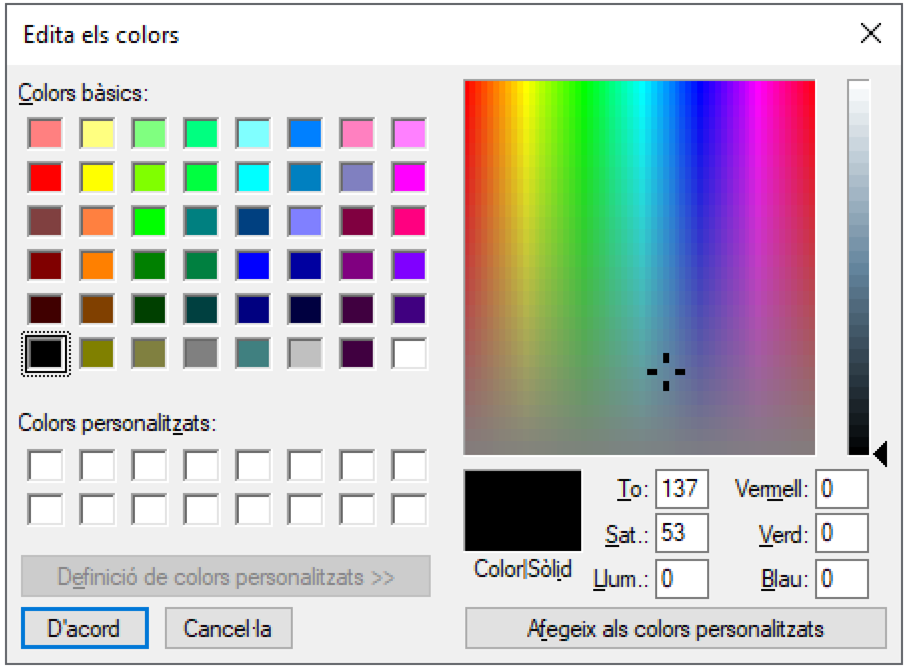
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \text{np.uint8}\left[255 * \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}\right]$$

Color models: Hue, Saturation, Luminance (HSL)

`L=0.299*R+0.587*G+0.114*B`



https://en.wikipedia.org/wiki/File:HSL_color_solid_cylinder_saturation_gray.png



HSV: V = max{R, G, B} (instead of L)

Python transformation functions:

```
matplotlib.colors.hsv_to_rgb.
matplotlib.colors.rgb_to_hsv
```

https://en.wikipedia.org/wiki/HSL_and_HSV#Hue_and_chroma

Now consider problems in section 3.3.

Image histogram. Equalization.

Objective: to take advantage of the full bandwidth of the image

- A very simple and effective processing suitable for (gray-level) images with unbalanced histograms.
- A perfectly balanced image displays a flat histogram (all gray levels are equiprobable). Accordingly, the cumulative histogram will be linear.
- Image equalization is the process to produce an image with a linear cumulative histogram
- Images must be `np.uint8`

- Calculation of the histogram:

  `scipy.ndimage.measurements.histogram`

  `scipy.ndimage.histogram`

- Calculation of the cumulative histogram:

  `numpy.cumsum`

Image histogram. Equalization.

- A histogram measures how many times (counts) a gray level appears in the image. The size of the image is M x N.

$$h[g] = \text{counts}[g]$$

- Probability

$$P[g] = \frac{\text{counts}[g]}{\text{M x N}}$$

- Cumulative histogram

$$\text{ch}[g] = \sum_{i=0}^{g} \text{counts}[i] \qquad \text{ch}[g = 255] = \text{M x N}$$

- Cumulative probability

$$\text{cp}[g] = \frac{\sum_{i=0}^{g} \text{counts}[i]}{\text{M x N}} \qquad \text{cp}[g = 255] = 1$$

Image histogram. Equalization. Image credit: https://commons.wikimedia.org/wiki/File:Hamptonshump.PNG

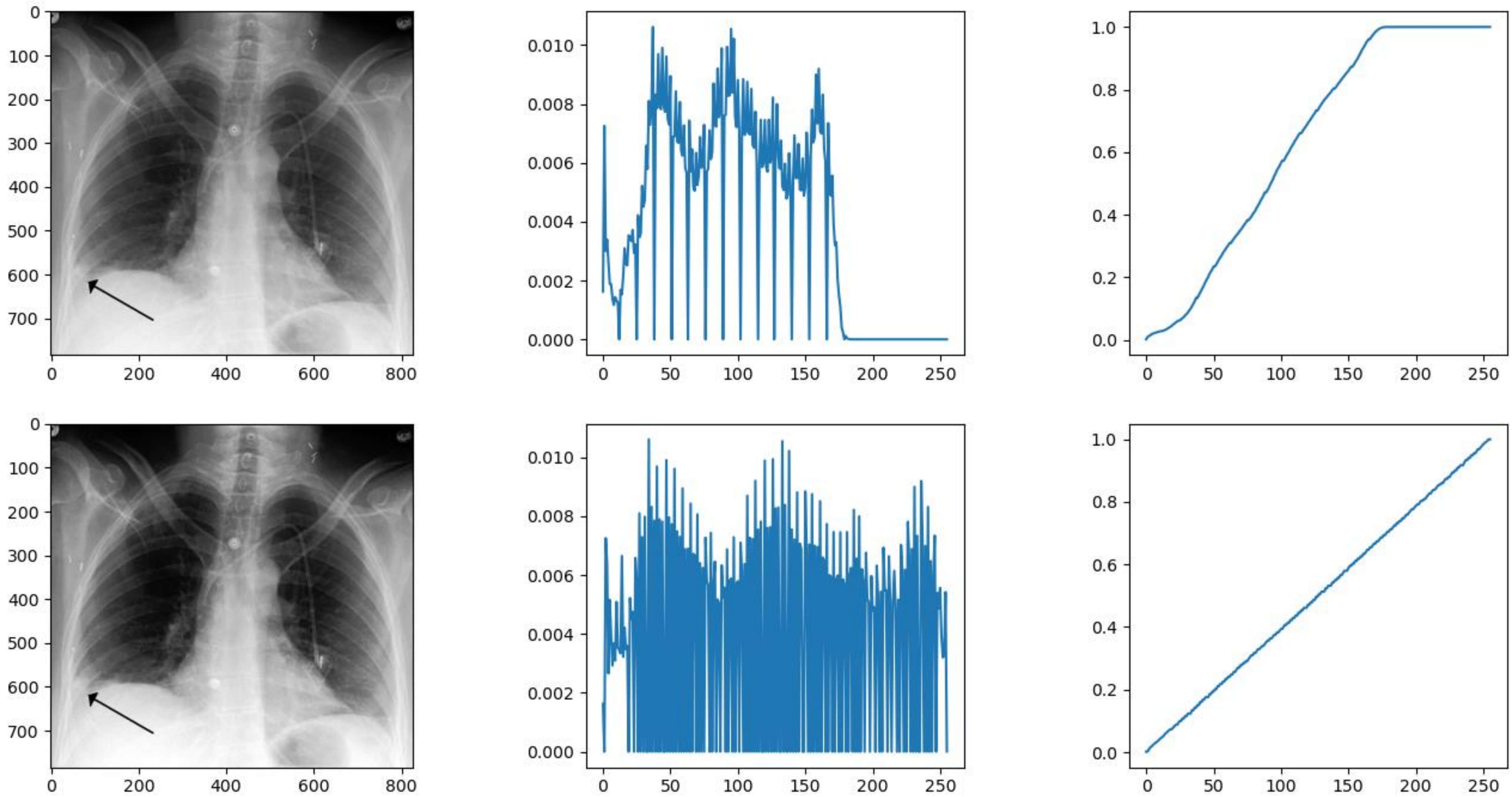
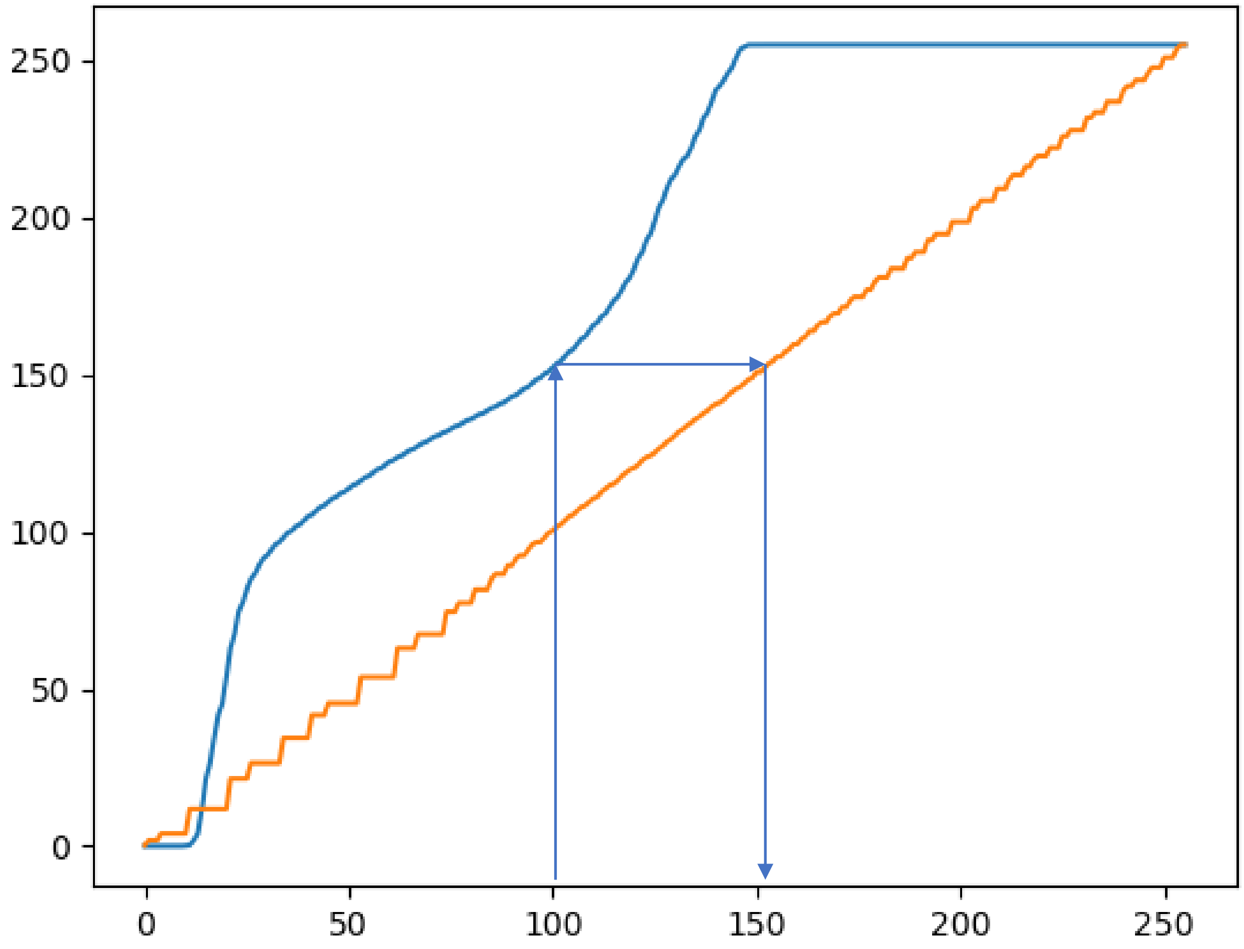


Image histogram. Equalization.

- Equalization formula $$\text{eqim} = \text{np.uint8}\left(\frac{255 * \text{ch[im]}}{\text{M x N}}\right)$$



Rescale the values of the cumulative histogram in the range 0-255 (y-axis)

What does ch[im] mean?

- ch[] is a 1d array;
- but im is a 2D array (!?)

Image entropy

Shannon's entropy formula:

$$S = -\sum_{g=0}^{255} P[g]\log_2 P[g] \qquad \text{bits/pixel} \qquad 0 < S < 8$$

Images are assumed to be `np.uint8`

This definition is general and can be applied to any communication channel, e.g.: texts.

S provides an idea of the theoretical compression limit.

The **local** entropy can be calculated using
`skimage.filters.rank.entropy`

For color images, the entropy should be calculated for every channel.

Cryptography and Steganography

- **Cryptography** is the science of writing in secret codes

    (from cryptos κρυπτός - hidden, secret)

- **Steganography** is the science of hiding information

    (from steganos στεγανός - covered, protected).

Whereas the goal of cryptography is to make data unreadable by a third party, the goal of steganography is to hide the data from a third party.

Applications

Digital watermarking (to prevent copyright infringement)

Data codification: time-stamps, serial numbers.

Combined with cryptographic methods improves security

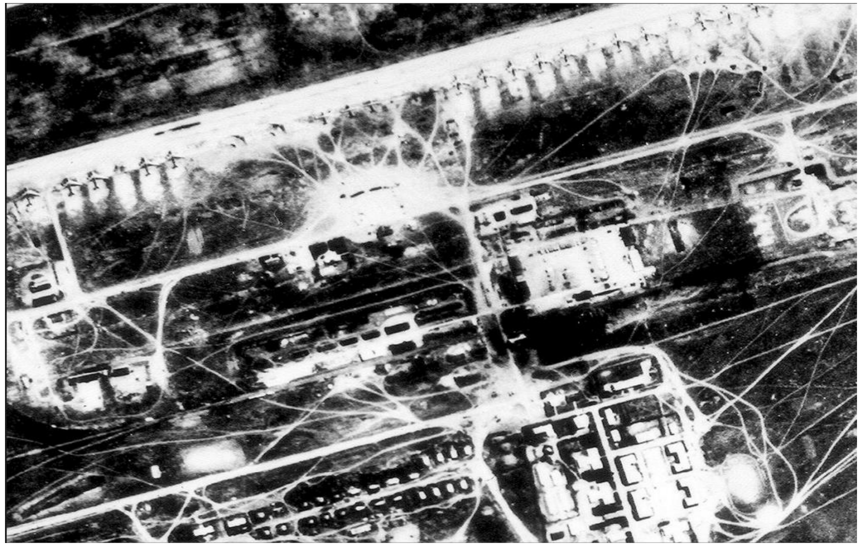
Steganography example

The secret image (cat) is hidden within the host image

Images from:
http://upload.wikimedia.org/wikipedia/commons/4/4e/StenographyOriginal.png
http://upload.wikimedia.org/wikipedia/commons/1/1b/StenographyRecovered.png

34

Another example



A satellite image of a Soviet strategic bomber base embedded in a Renoir painting

Images from:
www.jjtc.com/pub/r2026.pdf

Binary numbers

"There are only 10 types of people in the world:
Those who understand binary, and those who don't."

Most significant bits | Least significant bits

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

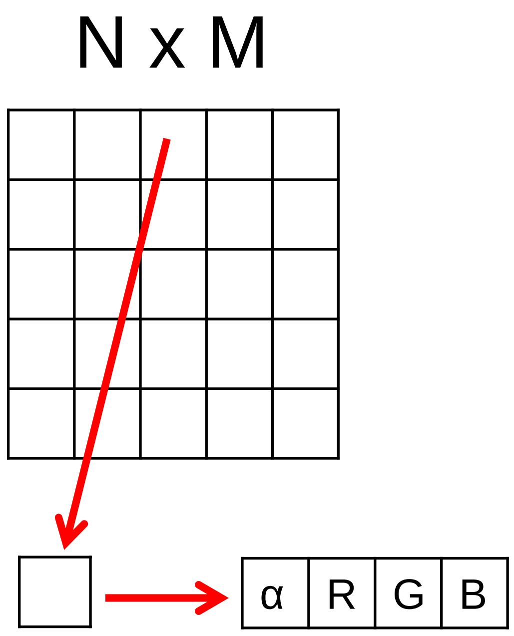
128	64	32	16	8	4	2	1

- Example
- 161=128+32+1

1	0	1	0	0	0	0	1

128	0	32	0	0	0	0	1

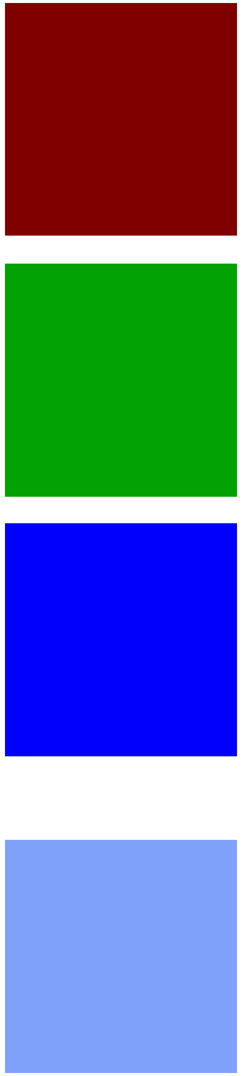
Images as matrices



A bit-mapped image is composed by pixels, arranged as a M (rows) x N (columns) matrix

- The color displayed in each pixel depends on the combination of four 8-bit values: R (red), G (green), B (blue) and α (transparency).
- α, R, G and B range from 0 to 255

Binary representation of color



R=128

1	0	0	0	0	0	0	0

G=161

1	0	1	0	0	0	0	1

B=250

1	1	1	1	1	0	1	0

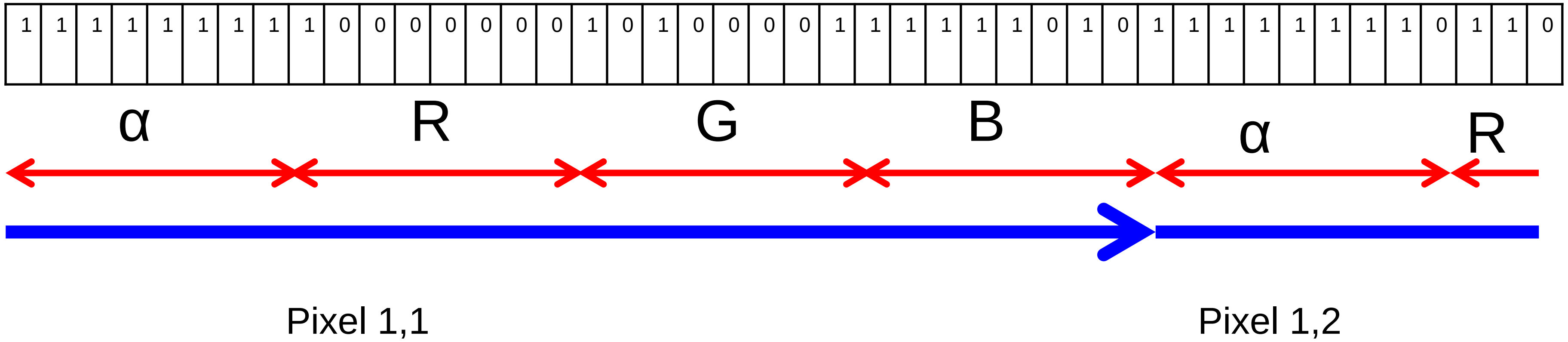
1	1	1	1	1	1	1	1

α=255 (opaque)

38

Bitmaps

A bitmap is a long strip containing just 1s and 0s:



Facts

“Color is often mistaken as a property of light when it really is a property of the brain. Our experience of color depends not only on the wavelength of the light rays that hit the retina, but also the context in which we perceive.”

http://hypertextbook.com/facts/2006/JenniferLeong.shtml

According to Calkins, human beings can distinguish al least 100000 colors.

D. J. Calkins, ”Mapping color perception to a physiological substrate” The Visual Neurosciences. The MIT Press, 1993.

https://mitpdev.mit.edu/library/erefs/chalupa/c064/section1.html

Least **Significant** Bits (LSB) steganography (I)

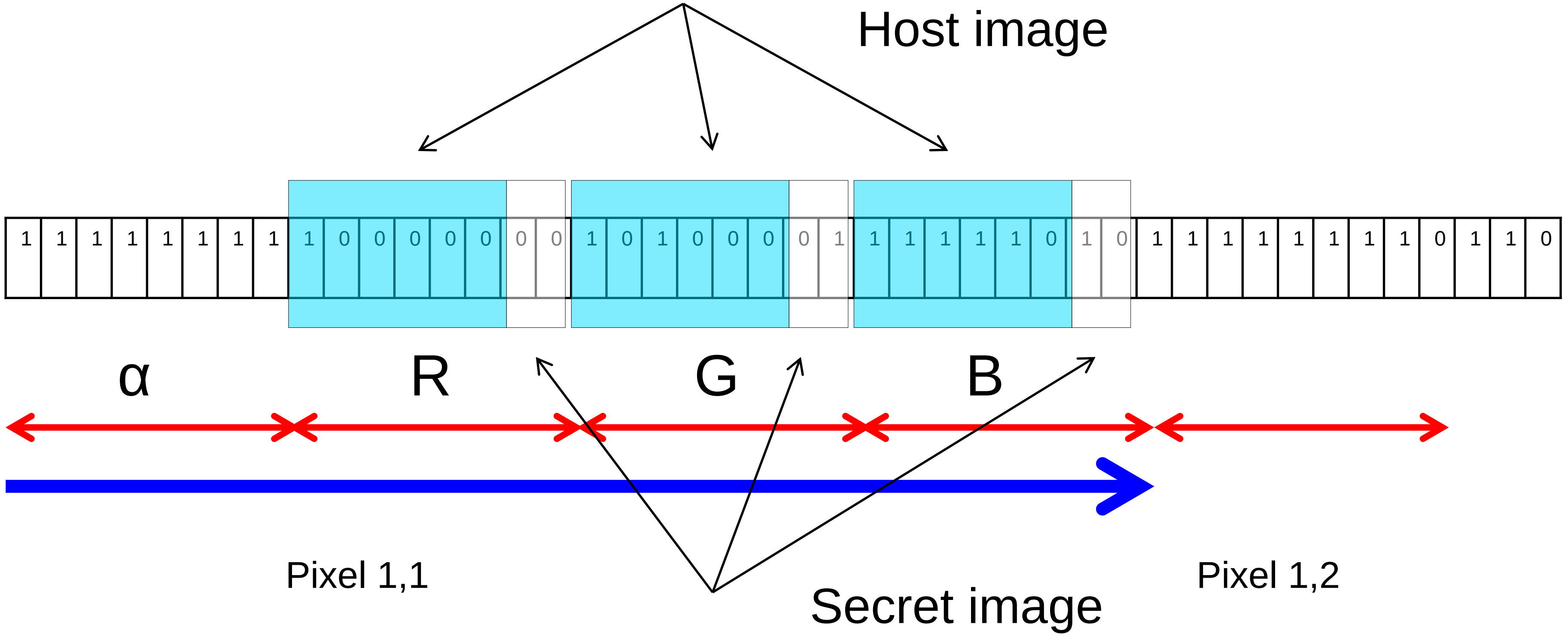
The 8-bit RGB (2^8) model can display 256 x 256 x 256 = 16.777.216 colors (far more than required)

Using a 6-bit RGB (2^6) model we would be able to display 64 x 64 x 64 = 262.144 color (enough!)

The remaining two bits are used to code the secret image using just 4 x 4 x 4 = 64 colors (low quality but not that bad!)

41

Least Significant Bits (LSB) steganography (II)



Since the host image is codified using the most significant bits (more intense), the secret image cannot be easily detected.

Further reading

Computing Practices

Exploring Steganography: Seeing the Unseen

Steganography is an ancient art of hiding information. Digital technology gives us new ways to apply steganographic techniques, including one of the most intriguing—that of hiding information in digital images.

Neil F. Johnson
Sushil Jajodia
George Mason University

Steganography is the art of hiding information in ways that prevent the detection of hidden messages. *Steganography*, derived from Greek, literally means "covered writing." It includes a vast array of secret communications methods that conceal the message's very

(with the exception of JPEG images). All color variations for the pixels are derived from three primary colors: red, green, and blue. Each primary color is represented by 1 byte; 24-bit images use 3 bytes per pixel to represent a color value. These 3 bytes can be represented as hexadecimal, decimal, and binary val-

Neil F. Johnson, Sushil Jajodia, "Exploring Steganography: Seeing the Unseen," Computer **31**, 26-34 (1998)

Programming tips:
Using np.unpackbits

```
a = np.array([[180, 200],[30, 50]], dtype=np.uint8)

a_bin = np.unpackbits(a)

a_bin0 = np.unpackbits(a, axis=0)

a_bin1 = np.unpackbits(a, axis=1)

a
Out[19]:
array([[180, 200],
      [ 30,  50]], dtype=uint8)

a_bin
Out[20]:
array([1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
      1, 0, 0, 0, 1, 1, 0, 0, 1, 0], dtype=uint8)
```

```
a_bin0
Out[21]:
array([[1, 1],
      [0, 1],
      [1, 0],
      [1, 0],
      [0, 1],
      [1, 0],
      [0, 0],
      [0, 0],
      [0, 0],
      [0, 0],
      [0, 1],
      [1, 1],
      [1, 0],
      [1, 0],
      [1, 1],
      [0, 0]], dtype=uint8)
```

```
a.shape
Out[37]: (2, 2)

a_bin0.shape
Out[38]: (16, 2)

a_bin1.shape
Out[39]: (2, 16)

a_bin.shape
Out[40]: (32,)
```

```
a_bin1
Out[23]:
array([[1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0],
      [0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0]], dtype=uint8)
```

Programming tips:
Using np.unpackbits

\# An unconventional binarization method:

```
imhidden = data.camera()
imhi_b = 255 * np.unpackbits(imhidden, axis= 0)[0::8,  :]
```

\# Select an arbitrary bitplane

```
bitplane7 = np.unpackbits(imhidden, axis= 0)[6::8,  :]
```

```
a_bin0
Out[21]:
array([[1, 1],
       [0, 1],
       [1, 0],
       [1, 0],
       [0, 1],
       [1, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 1],
       [1, 1],
       [1, 0],
       [1, 0],
       [1, 1],
       [0, 0]], dtype=uint8)
```

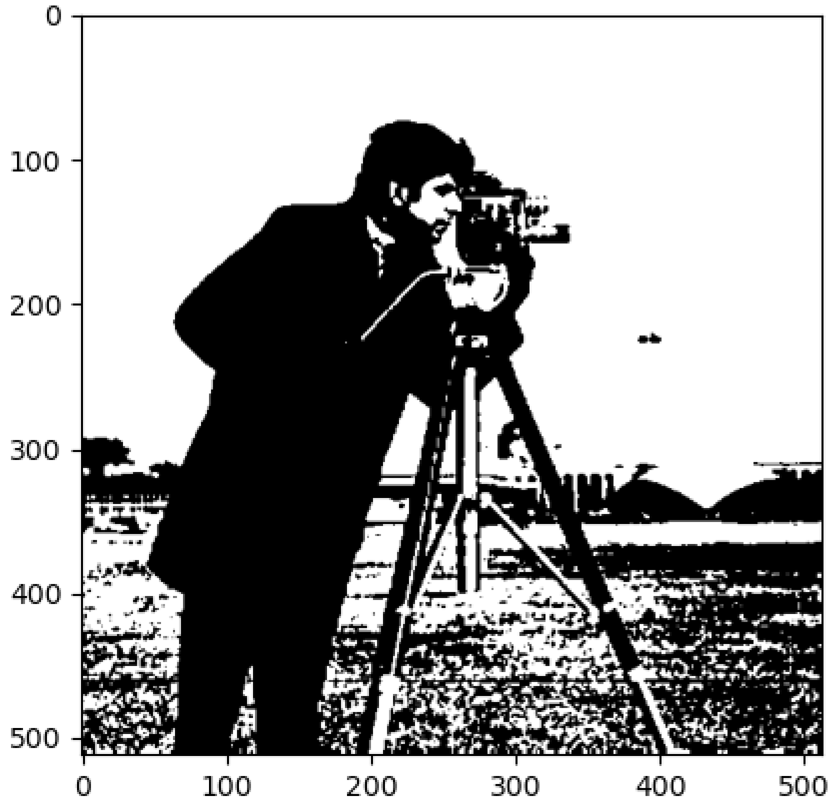
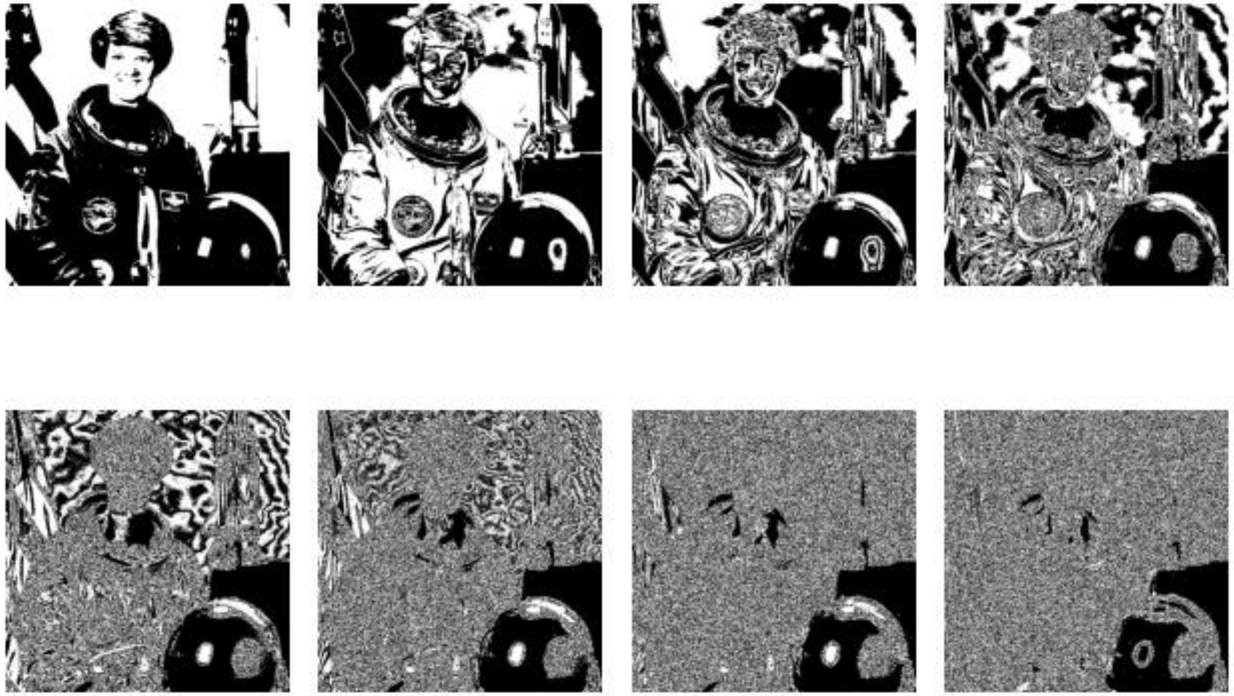
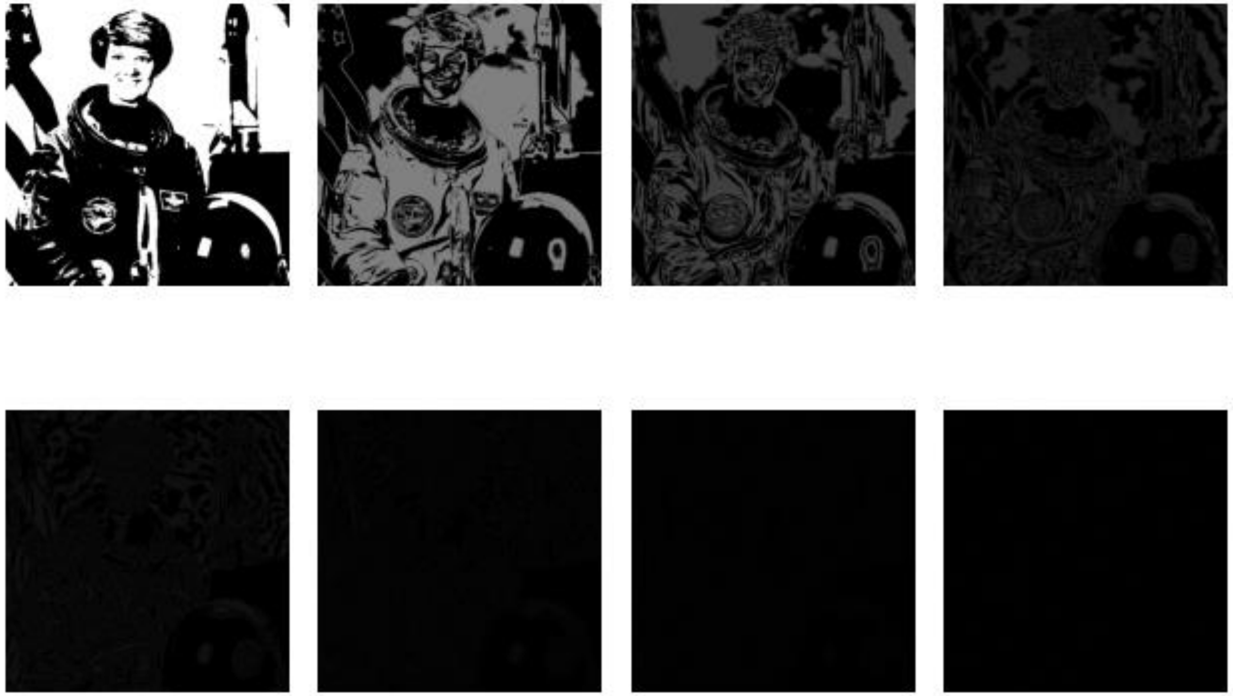


Image bitplanes

plt.imshow(…, cmap='gray')



plt.imshow(…, cmap='gray', vmax=255)



```
Graylevel_from_bitplanes =
128 * b0 + 64 * b1 + 32 * b2 + 16 * b3 + 8 * b4 + 4 * b5 +  2 * b6 + b7
```

Lab #5: Fourier transforms and spatial filtering.

Fourier transforms (FT) arranges the information in object (image) space into the frequency (Fourier) domain (space).

The image information in the Fourier space is organized in a very convenient way that enables manipulation of the frequency information (image detail).

However, FTs are for defined mathematical (analogic) functions:

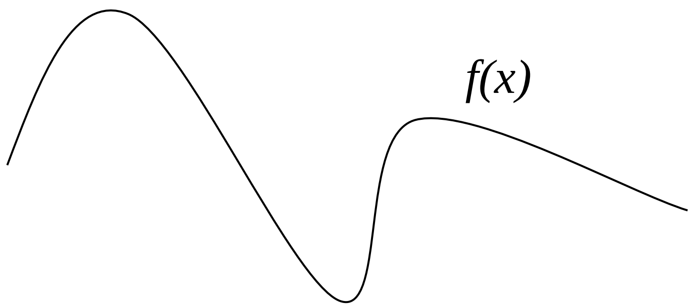
$$F(u,v) = \iint_{\mathbb{R}^2} f(x,y) \exp\left(-2\pi i\,(ux+vy)\right) dxdy$$

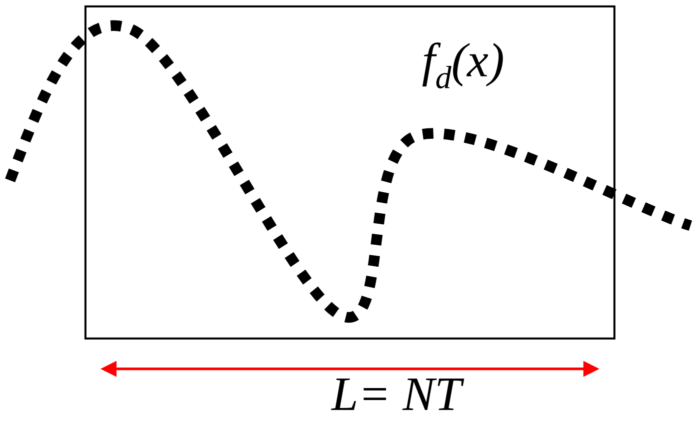
Can we extend this definition to sampled distributions?

The 2D Fast Fourier Transform algorithm enables us to perform this calculation

Digital Fourier Transforms?
Mathematical functions and sampled distributions

- $f(x)$ is a a continuous signal (a real or complex valued function) and $F(u)$ its Fourier transform
- $f(x)$ is converted into a band-limited discrete signal $f_d(x)$ through a sampling process. L is the signal length, T the sampling rate and N the number of samples, $L=NT$.
- Relationship between $f_d\,(x)$ and $F_d(u)$?





Digital Fourier Transform (DFT) properties that should be considered:

DFT produce very similar results to the analogical FT as long as very general conditions are hold:

- Signals are band limited: images are band limited!
  (*images are zero valued outside the image limits*)
- Sampling frequency (pixels per length unit) is high enough
- However, the convolution theorem should be used with care
  (circular convolution problem)

Shannon Sampling theorem:

- **Image space:** L is the signal length, T the sampling rate and N the number of samples, $L=NT$.
- **Fourier (frequency) space**: $1/T = N\ 1/L$.
- **Nyquist frequency (cut-off frequency):** $f_N = 1/\,2\,/\,T = N\,/\,2\,/\,L$

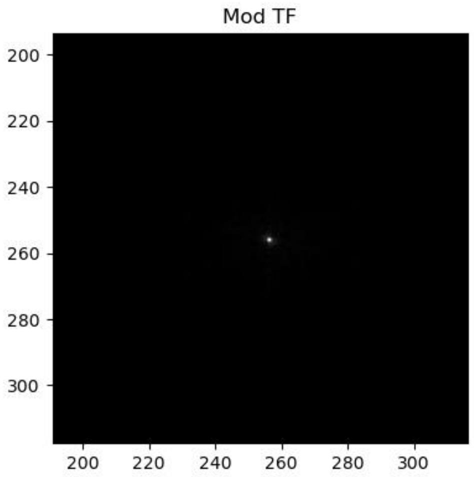
Fourier transforms: manipulation & interpretation

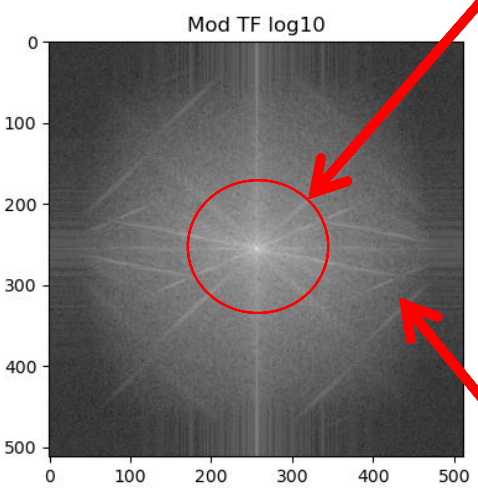
F(u, v) = FT[f(x,y)]

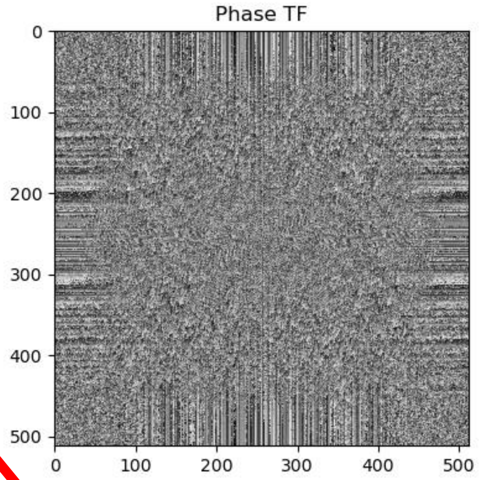
Low frequency area
High energy
Information describes by slow harmonics
(less details, less entropy)

```
F = fft.fftshift(fft.fft2(f)))
f = fft.ifft2(fft.ifftshift(fftshift(fft.fft2(f))))
```

```
np.real(F)
np.imag(F)
np.abs(F)
np.angle(F)
```








```
F = np.real(F) + 1j * np.imag(F)
F = np.abs(F)* np.exp(1j * np.angle(F))

Log LUT: np.log10(np.abs(F)+1)
```

High frequency area
Low energy
Information described by fast harmonics
(image details, more entropy)

Remarks and questions:

The amplitude (modulus) of the FT of real signals (such as images) is symmetrical, whereas the phase is anti-symmetrical.

What is the role of the `fft.fftshift / fft.ifftshift` pair ?

Try not to use this functions and investigate what happens.

Provided that `f` is real, `f2` is complex. Why?

```
F = fft.fftshift(fft.fft2(f)))
f2 = fft.ifft2(fft.ifftshift(fftshift(fft.fft2(f))))
```

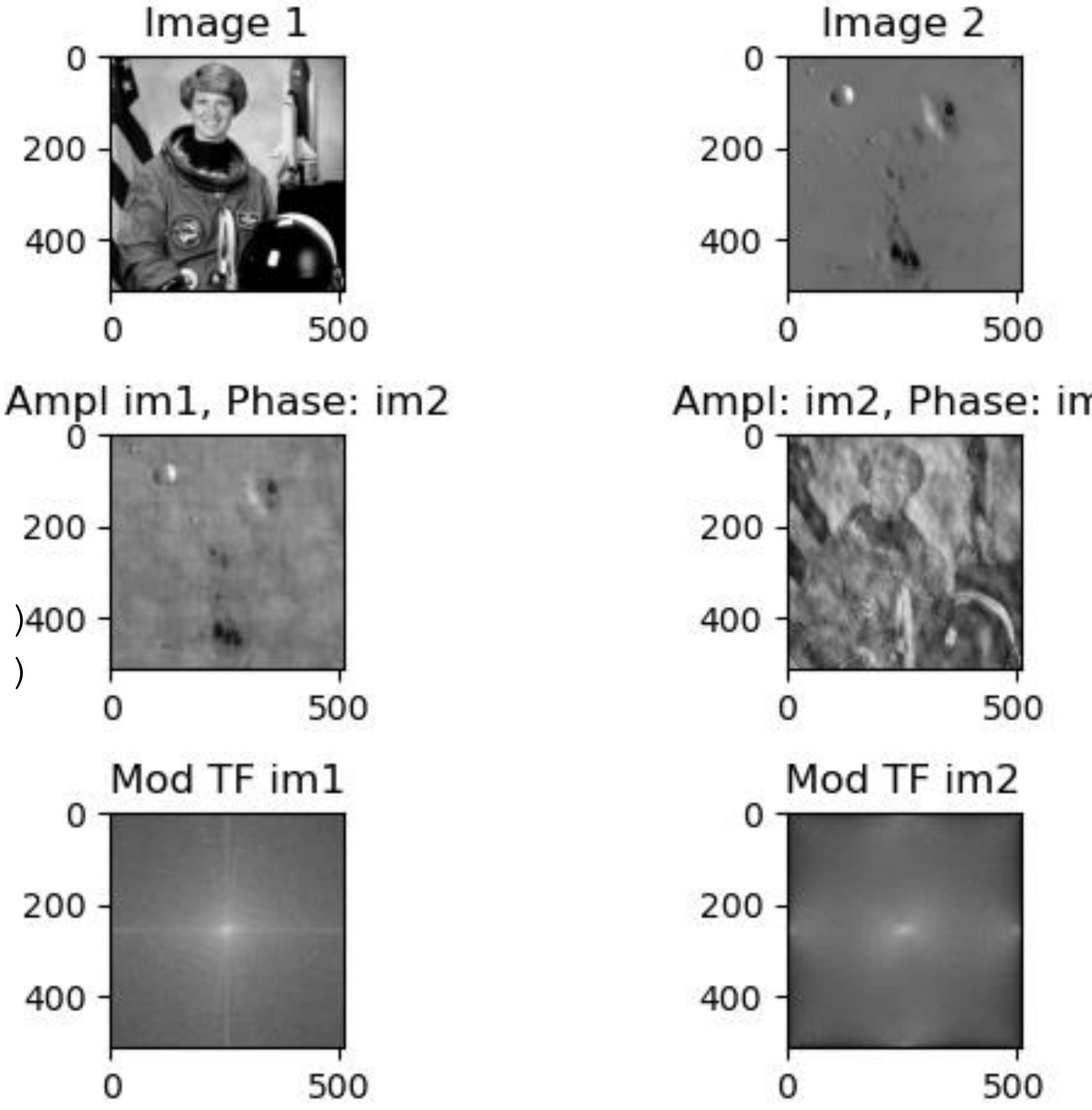
Spatial filtering targets the amplitude of the Fourier transform. Do not attempt to modify the phase. Why? (exercise 5.3)

Recall to use the Shannon theorem to determine the length of the signal $1/T$ in Fourier space (only if necessary) (exercise 5.2)

Color images: every channel with `fft.fft2` but not using a single `fft.fftn`

Oppenheim and Lim Experiment
Importance of phase in signals

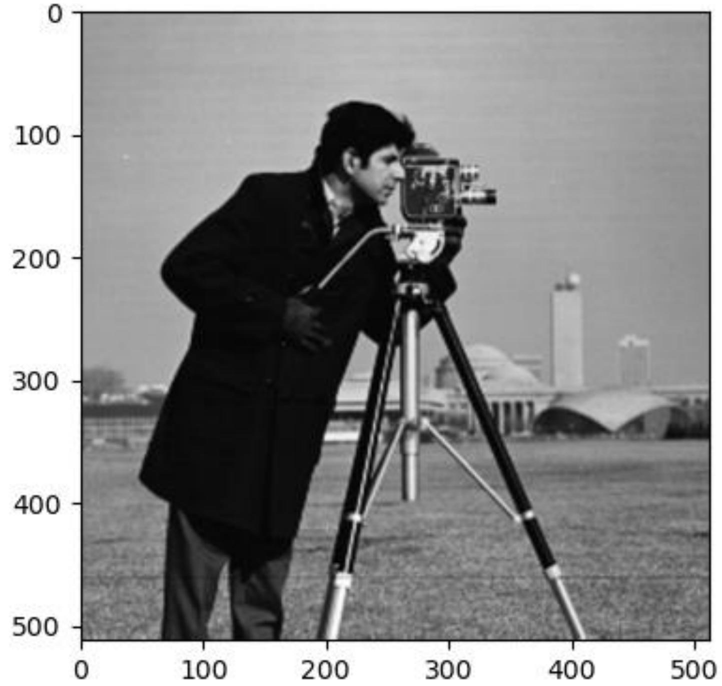
```
im1 = data.astronaut()[:, :, 1]
im2 = data.moon()

tf1 = fft.fftshift(fft.fft2(im1))
tf2 = fft.fftshift(fft.fft2(im2))

tf12 = np.abs(tf1) * np.exp(1j * np.angle(tf2))
tf21 = np.abs(tf2) * np.exp(1j * np.angle(tf1))

im12 = np.real(fft.ifft2(fft.ifftshift(tf12)))
im21 = np.real(fft.ifft2(fft.ifftshift(tf21)))
```



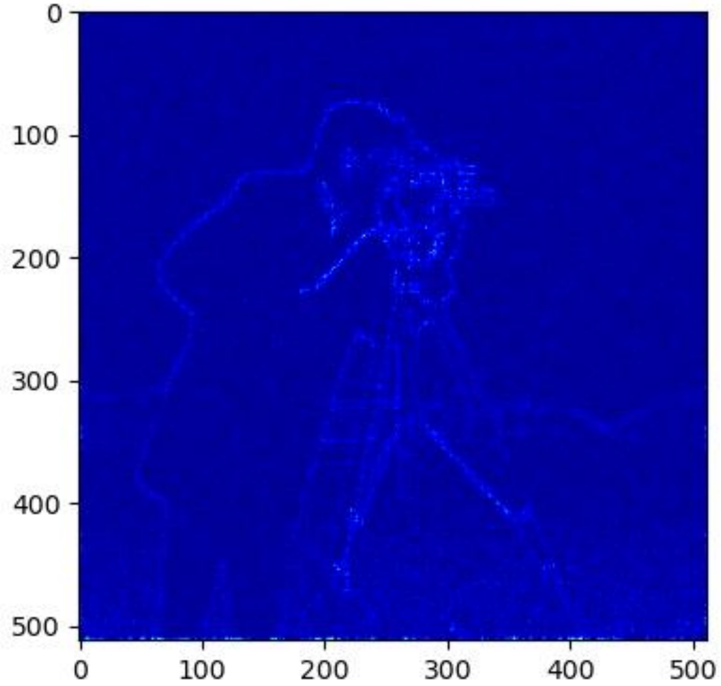
Phase Rules!
The phase characterizes the image.
Filters should modify the amplitude

An example of high pass filtering: Phase-only reconstruction

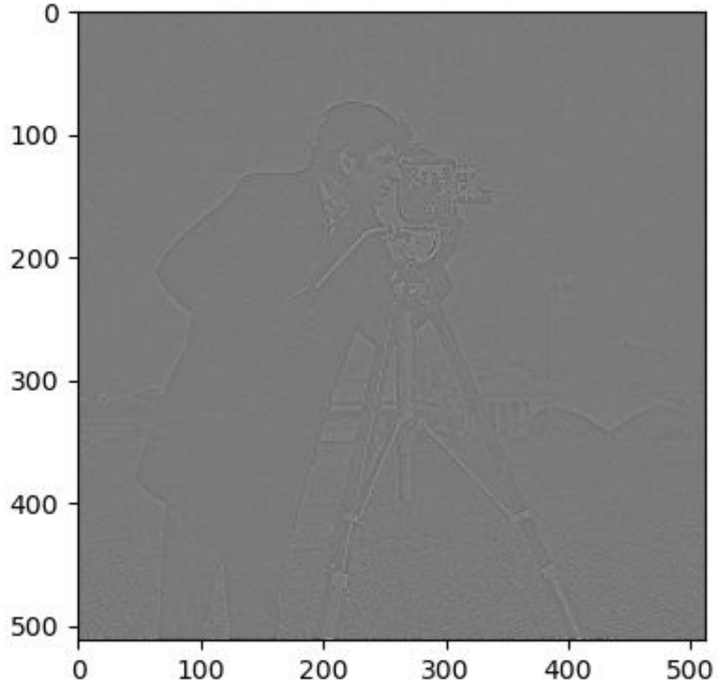
```
im = data.camera()
tfim = fft.fftshift(fft.fft2(im))
tfpo = tfim / np.abs(tfim)
poim = fft.ifft2(fft.fftshift(tfpo))
```



np.abs



np.real



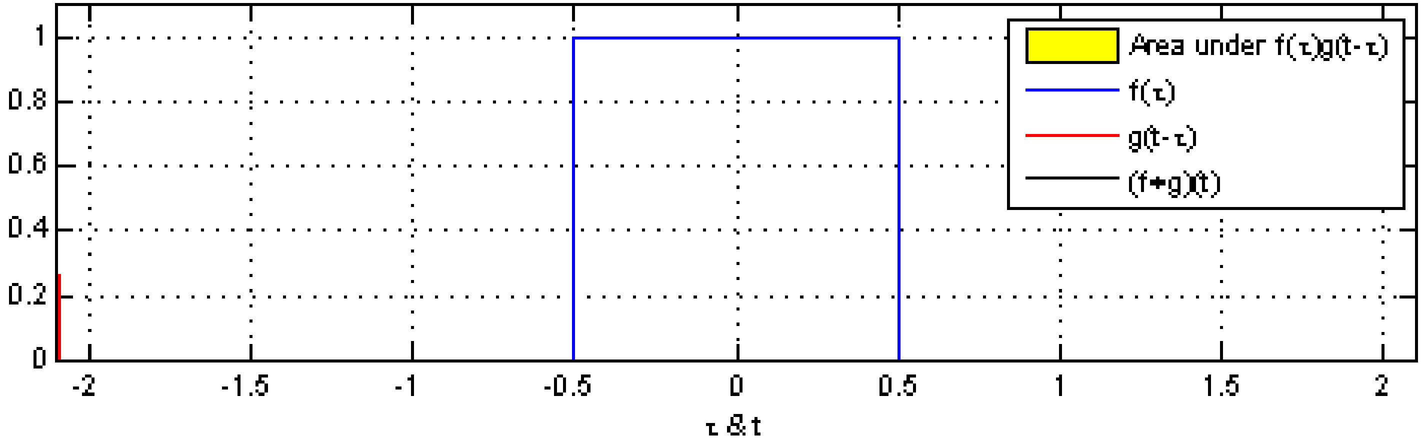
Convolution

$$\left[f*g\right](x)=\int_{\mathbb{R}} f(u)g(x-u)du$$

$$\left[f*g\right](x,y)=\int_{\mathbb{R}^2} f(u,v)g(x-t,y-v)dt$$

Computationally intensive: one integral for every point (x, y)

$$\left[f(x-a)*\delta(x)\right]=\delta(x-a)$$



This image is still in the pdf version

Image credit: Convolution_of_box_signal_with_itself.gif: Brian Amberg derivative work: Tinos (talk) - Convolution_of_box_signal_with_itself.gif, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11003835

Convolution theorem: $$f*g=\mathrm{FT}^{-1}\left[\mathrm{FT}\left[f\right]\mathrm{FT}\left[g\right]\right]$$

Spatial filtering using the convolution theorem

```
F = fft.fftshift(fft.fft2(f)))

U, V = np.meshgrid(np.linspace(-1, 1, NP),
                   np.linspace(-1, 1, NP))
```

Note arbitrary limits -1, 1

Low pass filter:
```
radius = 0.2 # e.g.
d = np.sqrt(U * U + V * V)
Lp = d < radius
```

Gaussian filter:
```
Gf = np.exp(-A(u * u + v * v))# set A
```

Laplacian filter:
```
Lf = u * u + v * v
```

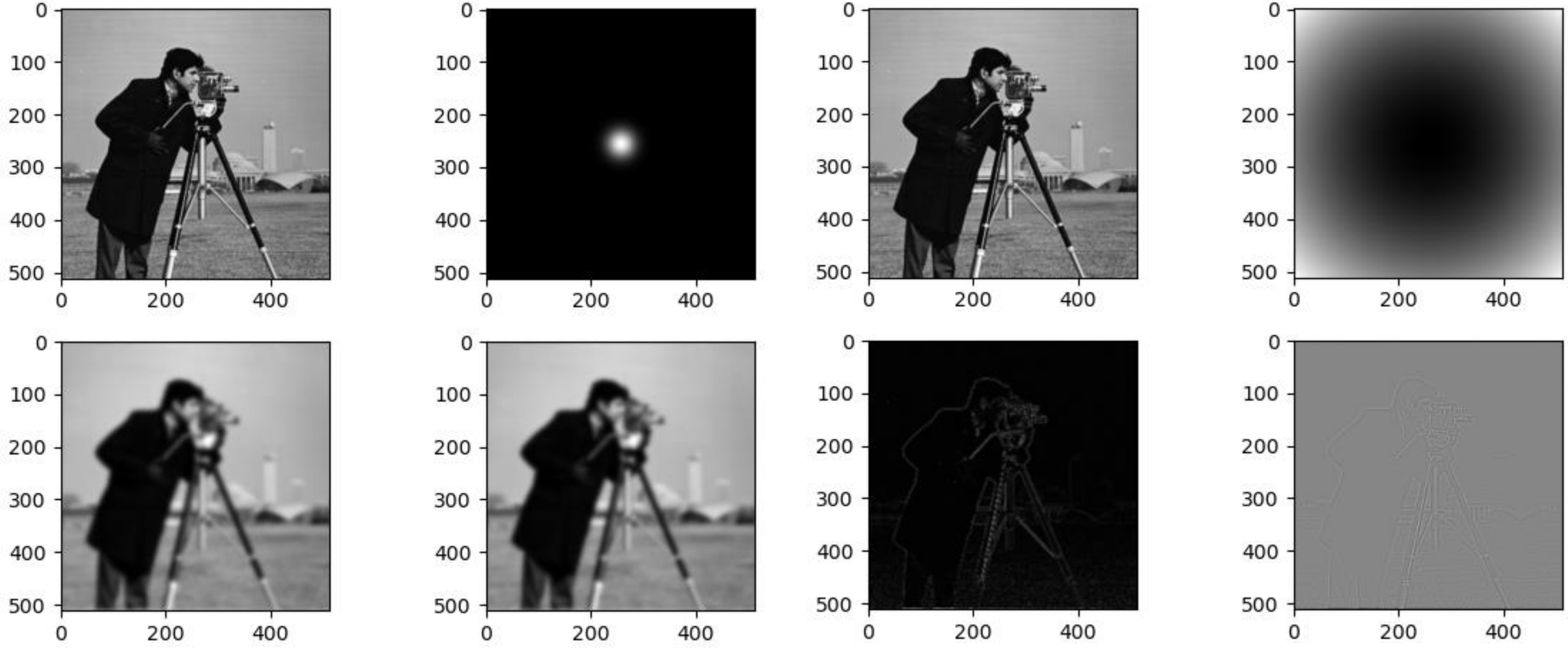
Convolution
```
C = fft.ifft2(fft.ifftshift(F * Lp))
C = fft.ifft2(fft.ifftshift(F * Gf))
C = fft.ifft2(fft.ifftshift(F * Lf))
```

Gaussian and Laplacian filters
Examples of low- and high-pass filters

Equivalent to calculate the second derivatives of the image (Laplacian operator)

Gaussian filter A=100

Laplacian filter



Low pass filters remove high frequency information = details

- Additive noise is high frequency. It can be removed with a low pass filter
- Nevertheless, this is a trade-off problem: after a low-pass filtering both noise and image information are removed as well.

High pass filters enhance details i.e., fast contrast changes.

- Close to the DC term (frequency (0,0)), the Fourier transform accumulates most of its energy. After a high-pass filter, the image looks mainly black.
- The real part looks grayish. Values are real but with positive and negative values.
- Zero is in-between. If displayed with imshow, zero is shown around gray level 128.

Convolution in object space

$$f * g = \mathrm{FT}^{-1}\left[\mathrm{FT}\left[f\right]\mathrm{FT}\left[g\right]\right]$$

$$\left[f * g\right](x, y) = \int_{\mathbb{R}^2} f(u,v)g(x-t, y-v)dt$$

- So far, we have considered the convolution theorem, but we can filter images in object space as well.
- Statement: # rows x # columns of f and g are expected to be the same.
- Well, not really.

- The information of the filter is concentrated in the center of the array: 3x3, 5x5, et cetera. The rest of the array is zero. The contribution of the pixels with zero values to the convolution is zero

```
filt = = np.ones([5, 5])
Out[4]:
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
im[0:12, 0:12]
Out[6]:
array([[147, 103,  58,  51,  76, 100, 121, 135, 141, 134, 115,  90],
       [171, 141, 114, 107, 128, 144, 158, 165, 167, 150, 124,  86],
       [194, 178, 165, 157, 165, 180, 185, 186, 176, 154, 115,  69],
       [213, 206, 198, 191, 194, 198, 202, 194, 174, 142, 101,  45],
       [223, 220, 215, 210, 211, 206, 206, 187, 161, 123,  76,  25],
       [229, 228, 224, 218, 216, 210, 198, 175, 143, 101,  49,  13],
       [229, 230, 226, 224, 215, 205, 184, 159, 121,  73,  30,  19],
       [226, 226, 223, 219, 212, 200, 182, 149, 102,  49,  26,  22],
       [223, 223, 220, 216, 211, 200, 179, 145,  95,  41,  29,  21],
       [221, 222, 218, 213, 210, 201, 178, 151, 110,  63,  31,  22],
       [222, 221, 219, 217, 211, 203, 188, 165, 138,  98,  57,  27],
       [219, 222, 220, 219, 212, 209, 197, 181, 159, 131,  90,  45]],
      dtype=uint8)
```

Pixel under consideration

```
[[147, 103,  58,  51,  76, 100, 121, 135, 141, 134, 115,  90],
 [171, 141, 114, 107, 128, 144, 158, 165, 167, 150, 124,  86],
 [194, 178, 165, 157, 165, 180, 185, 186, 176, 154, 115,  69],
 [213, 206, 198, 191, 194, 198, 202, 194, 174, 142, 101,  45],
 [223, 220, 215, 210 x 1, 211 x 1, 206 x 1, 206 x 1, 187 x 1, 161, 123,  76,  25],
 [229, 228, 224, 218 x 1, 216 x 1, 210 x 1, 198 x 1, 175 x 1, 143, 101,  49,  13],
 [229, 230, 226, 224 x 1, 215 x 1, 205 x 1, 184 x 1, 159 x 1, 121,  73,  30,  19],
 [226, 226, 223, 219 x 1, 212 x 1, 200 x 1, 182 x 1, 149 x 1, 102,  49,  26,  22],
 [223, 223, 220, 216 x 1, 211 x 1, 200 x 1, 179 x 1, 145 x 1 ,  95,  41,  29,  21],
 [221, 222, 218, 213, 210, 201, 178, 151, 110,  63,  31,  22],
 [222, 221, 219, 217, 211, 203, 188, 165, 138,  98,  57,  27],
 [219, 222, 220, 219, 212, 209, 197, 181, 159, 131,  90,  45]],
```

Convolution:
210 x 1 + 211 x 1 + 206 x 1 + … + 179 x 1 + 145 x 1 >> 255 !!!

scipy.ndimage.convolve

scipy.ndimage.convolve(*input*, *weights*, *output=None*, *mode='reflect'*, *cval=0.0*, *origin=0*)

Multidimensional convolution. [source]

The array is convolved with the given kernel.

Parameters: **input** : *array_like*

The input array.

weights : *array_like*

Array of weights, same number of dimensions as input

output : *array or dtype, optional*

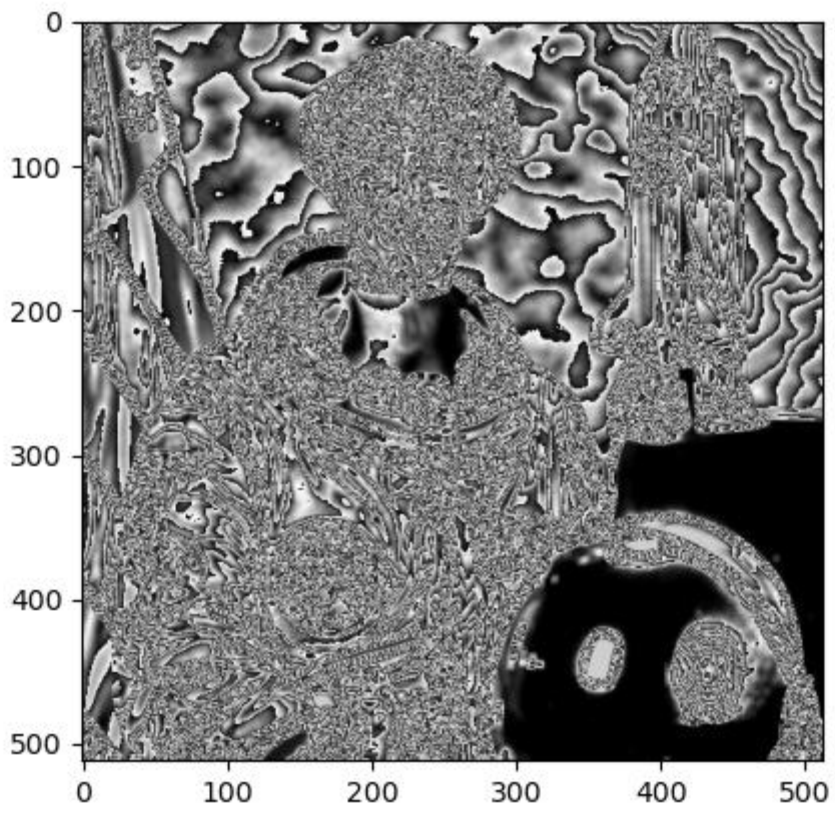
The array in which to place the output, or the dtype of the returned array. By default an array of the same dtype as input will be created.

mode : *{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*

The *mode* parameter determines how the input array is extended beyond its boundaries. Default is 'reflect'. Behavior for each valid value is as follows:

'reflect' (*d c b a | a b c d | d c b a*)

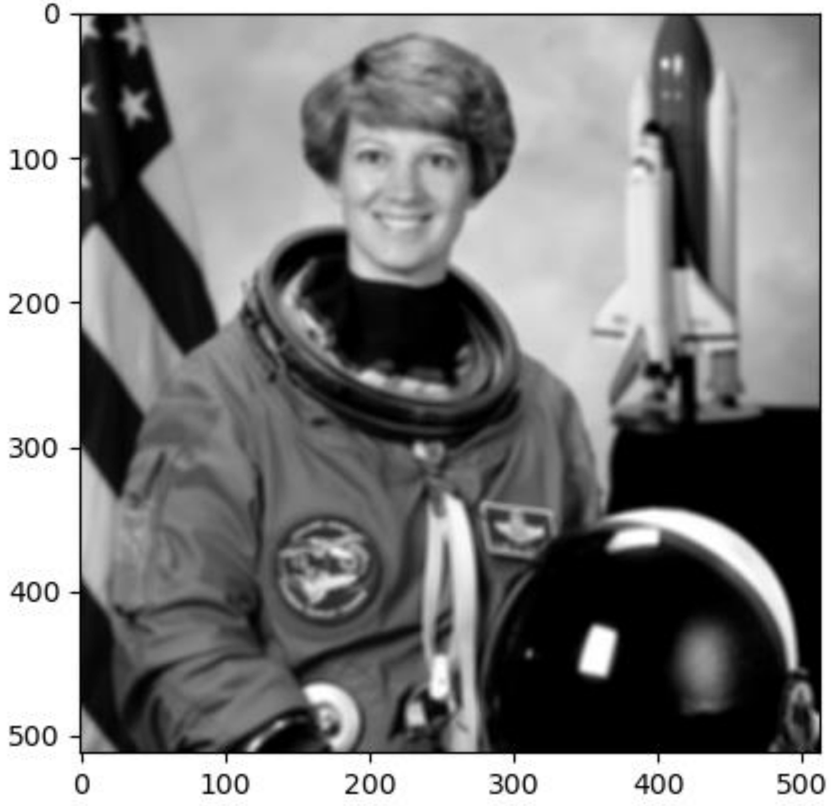
Name	Type	Size	Value
conv	Array of uint8	(512, 512)	Min: 0 Max: 255
filt	Array of float64	(5, 5)	Min: 1.0 Max: 1.0
im	Array of uint8	(512, 512)	Min: 0 Max: 255




```
im = data.astronaut()[:, :, 1]
filt = np.ones([5, 5])
conv = ndimage.convolve(im, filt)
plt.figure()
plt.imshow(conv, cmap='gray')
```

```
im = np.double(data.astronaut()[:, :, 1])
filt = np.ones([5, 5])
conv = ndimage.convolve(im, filt)
plt.figure()
plt.imshow(conv, cmap='gray')
```

Nan	Type	Size	Value
conv	Array of float64	(512, 512)	Min: 0.0 Max: 6356.0
filt	Array of float64	(5, 5)	Min: 1.0 Max: 1.0
im	Array of float64	(512, 512)	Min: 0.0 Max: 255.0



The resulting image looks slightly defocused

How are filters in image and Fourier space related?

- Promote a n x n filter to a N x N array (padding)
- Calculate the Fourier transform
- Compare how they look

```
filt = np.array(
    [[-1., -1., -1.],
     [-1., 8., -1.],
     [-1., -1., -1.]])
```

Image space Laplacian filter.

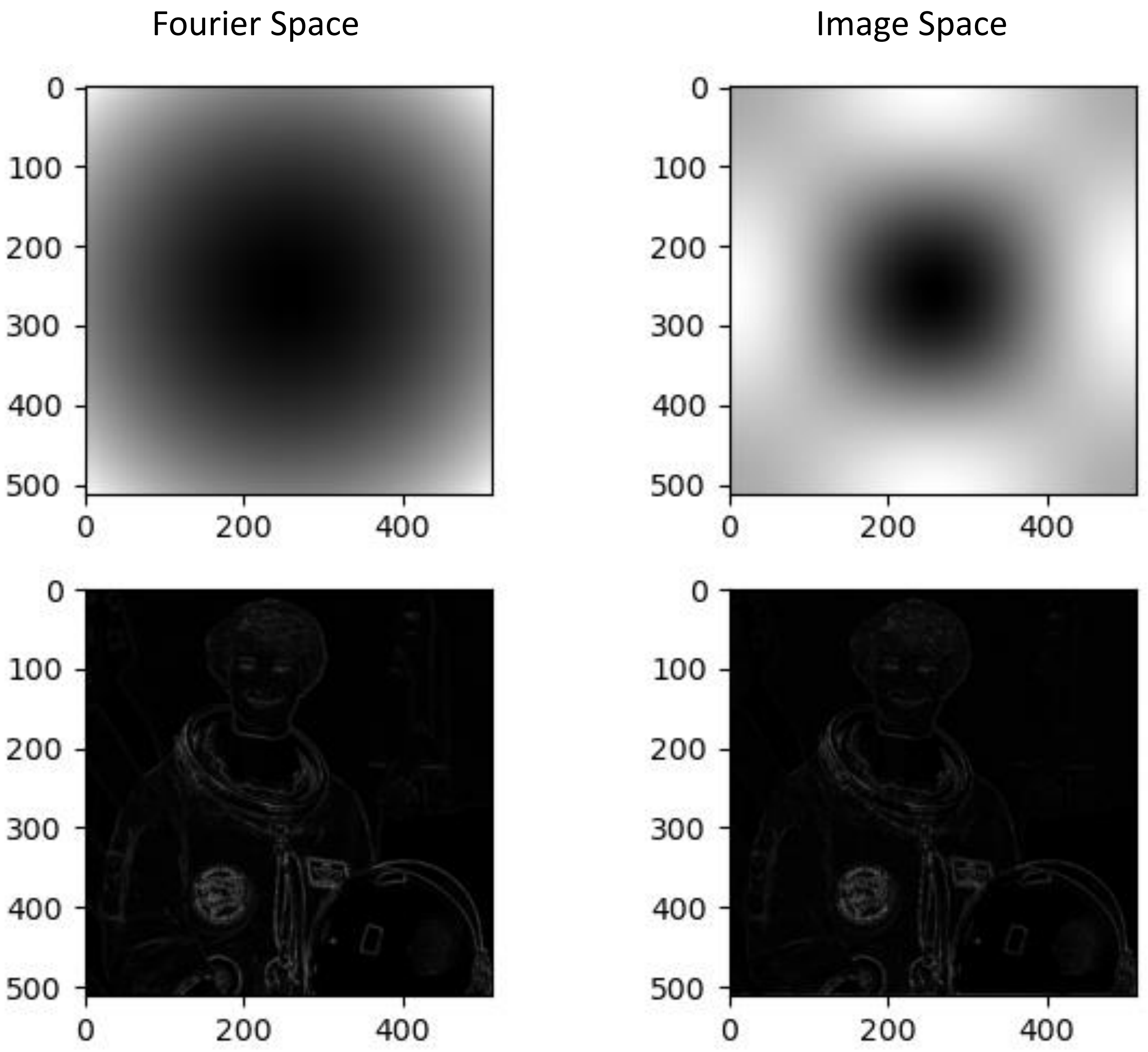
Wait. What?

```
filt512 = np.zeros([512, 512])
filt512[255:258, 255:258] = filt
```

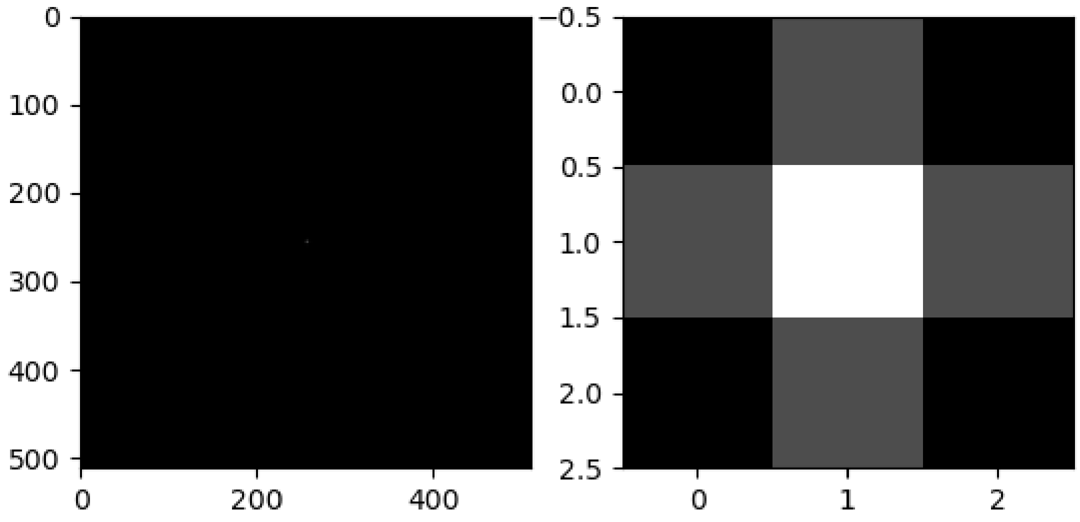
Cast a 3x3 array into a 512x512 array

```
ftfilt = fft.fftshift(fft.fft2(filt512))

u, v = np.meshgrid(np.linspace(-1, 1, 512), np.linspace(-1, 1, 512))
lap = u**2 + v**2
```

```
filt = np.array(
     [[0., -1., 0.],
      [-1., 4., -1.],
      [0., -1., 0.]])

filt512 = np.zeros([512, 512])
filt512[255:258, 255:258] = filt

ftfilt = fft.fftshift(fft.fft2(filt512))

u, v = np.meshgrid(np.linspace(-1, 1, 512),
                              np.linspace(-1, 1, 512))
lap = u**2 + v**2

ftim = fft.fftshift(fft.fft2(im))
res = np.abs(fft.ifft2(fft.ifftshift(ftim * lap)))

conv = np.abs(ndimage.convolve(im, filt))
```



```
lap = u**2 + v**2
tflap = np.real(fft.ifftshift(fft.fft2(lap)))
plt.figure()
plt.subplot(1,2,1)
plt.imshow(tflap, cmap='gray')
plt.subplot(1,2,2)
plt.imshow(tflap[255:258, 255:258], cmap='gray')

print(4 * tflap[255:258, 255:258] / tflap.max())
```




```
print(4 * tflap[255:258, 255:258] / tflap.max())
[[-9.17640509e-18  1.21582832e+00 -2.60796478e-19]
 [ 1.21582832e+00  4.00000000e+00  1.21582832e+00]
 [-1.04229685e-17  1.21582832e+00  2.60796478e-19]]
```

Signs are not properly recovered, because the definition of the Lapacian filter skips this information:
Lap = u**2 + v **2

Laplacian-like (edge extraction) filters

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

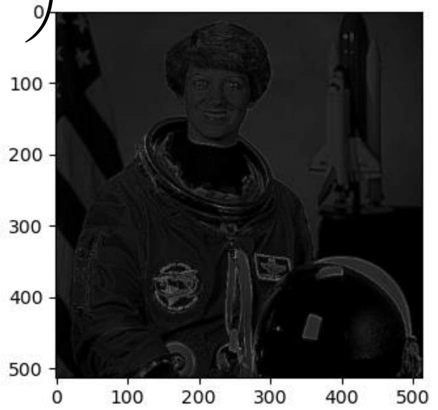
Average and gaussian like filters

$$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

Edge enhancer filters

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad \begin{pmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$



How to calculate the Laplacian of an image
(i) Fourier Space

$f, \; F = FT(f)$

$FT(\Delta f)$?

$FT\left(\frac{\partial f}{\partial x}\right) \propto uF \quad FT\left(\frac{\partial f}{\partial y}\right) = vF$

$FT\left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right) \propto (u^2+v^2)\, F$

Laplacian filter

How to calculate the Laplacian of an image
(ii) Image Space

$$\lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

How to calculate derivatives with sampled information?

$x - x_0 = 1 \quad \Rightarrow \quad f'_i = f_i - f_{i-1}$

$f_{i+1} - f_i$

Ambiguous definition:
Left and right derivatives

$f''_i = f'_i - f'_{i-1}$

Second derivative:

$= f_{i+1} - f_i - f_i + f_{i-1} =$

$-(- f_{i+1} + 2 f_i - f_{i-1})$

Laplacian operator for sampled functions

$$\Delta f = \frac{\partial^2}{\partial x^2} f + \frac{\partial^2}{\partial y^2} f$$

$$f''_{i,j} = -(-f_{i+1,j} + 2f_{ij} - f_{i-1,j})$$

$$-(-f_{i,j+1} + 2f_{ij} - f_{i,j-1}) =$$

$$= \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} * f_{ij}$$

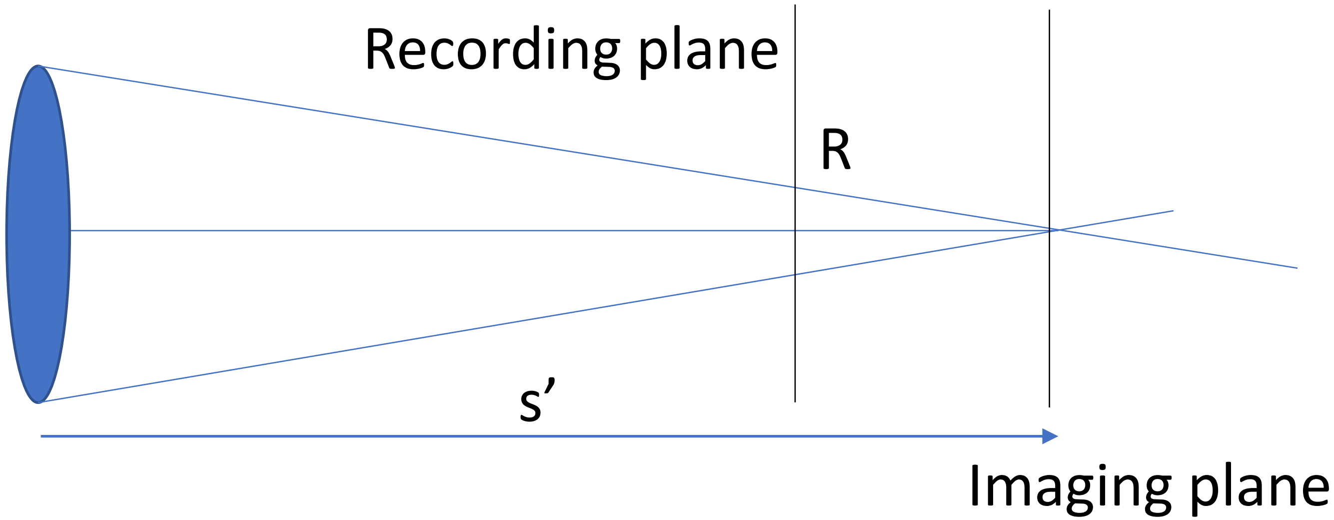
Lab #6: Point–spread functions and image restoration filters.

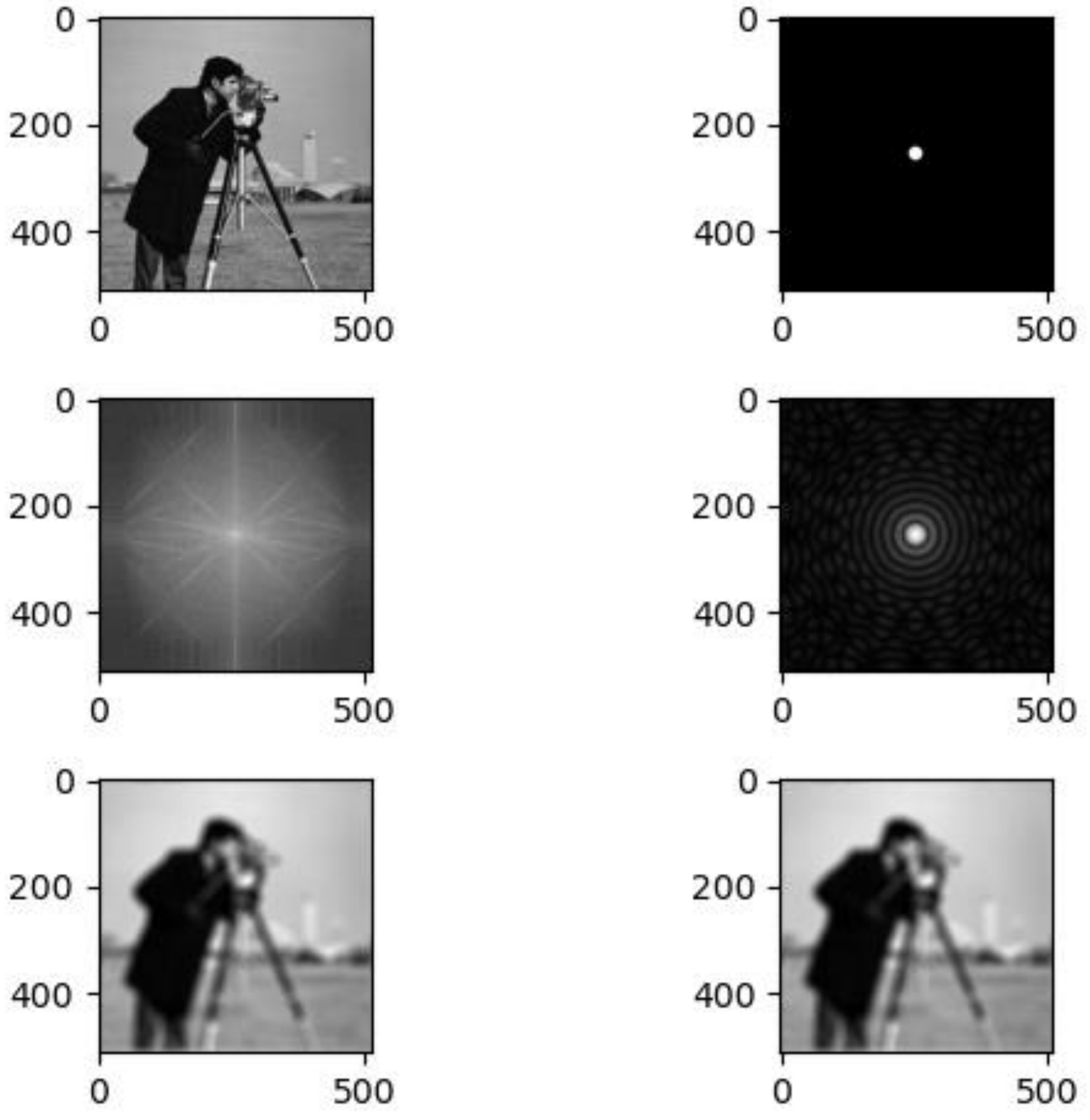
1. Geometrical model for defocus

$$\left[f(x,y) * \mathrm{circ}\frac{r}{R} \right] =$$

$$\mathrm{FT}^{-1}\left[F(u,v)\frac{R}{\rho} J_1(2\pi R\rho) \right]$$

$$r = \sqrt{x^2 + y^2} \qquad \rho = \sqrt{u^2 + v^2}$$





A defocused imaging system behaves as a physical low pass filter

2. Image formation model in aberration-free systems

Perfect optical system: the image of a single point (in object space) is another point (in image space)

Despite this definition, optical systems are always affected by diffraction due the limited extent of lenses.

The Airy disc (with radius r_A) describes the *actual point* (point-spread function, PSF) for a perfect system

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

$$\left[f * \mathrm{FT}_{\lambda s}\left[\mathrm{circ}\frac{r}{R} \right] \right] =$$

$$\mathrm{FT}^{-1}\left[\mathrm{FT}\left[f\left(\frac{x}{\beta}, \frac{y}{\beta} \right) \right] \frac{R}{\rho} J_1\left(\frac{2\pi R \rho}{\lambda s} \right) \right]$$

$r = \sqrt{x^2 + y^2}$ $\rho = \sqrt{u^2 + v^2}$ β geometric magnification

In an aberration-free system, the PSF is approximated as

$$\mathrm{PSF}(x,y) = \left| \mathrm{TF}_{\lambda s}\left[\mathrm{circ}\left(\frac{R}{R_0} \right) \right] \right|^2$$

Where the exit pupil is a circle with radius R_0); s is the distance between the exit pupil and the image plane and λ is the wavelength of the illuminating source. Note that the coordinates of the PSF are scaled according to the product λs.

For a defocused system, the PSF is $\mathrm{PSF}(x,y) = \mathrm{circ}\left(\frac{R}{R} \right)$

Then, the recorded image $d(x,y)$ *is* modeled as the convolution between the ideal, perfect geometrical image $i(x,y)$ and the PSF.

$$d(x,y) = i(x,y) * \mathrm{PSF}(x,y)$$

Dealing with scales: Use the Shannon theorem

Input space: $L = N * T$
Where L length of the window side of the image, N number of pixels (in one of the axis) and T is the sampling period.

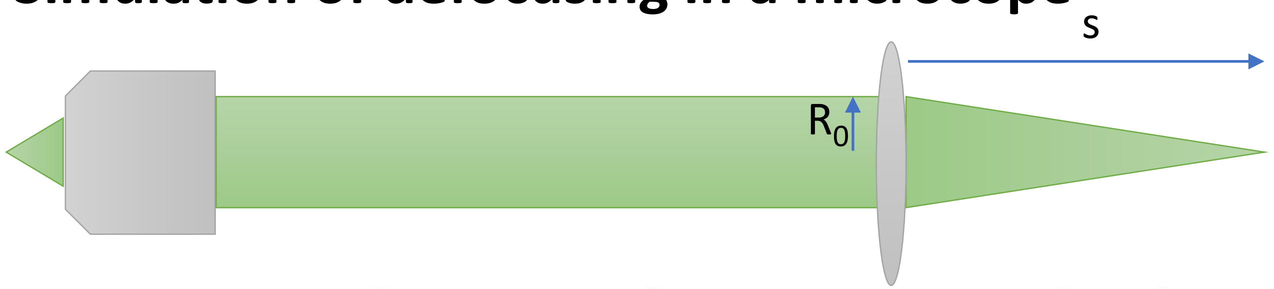
Transformed space: $1 / T = N \; * 1 / L$

The scale of diffracted wavefront is scaled λs.

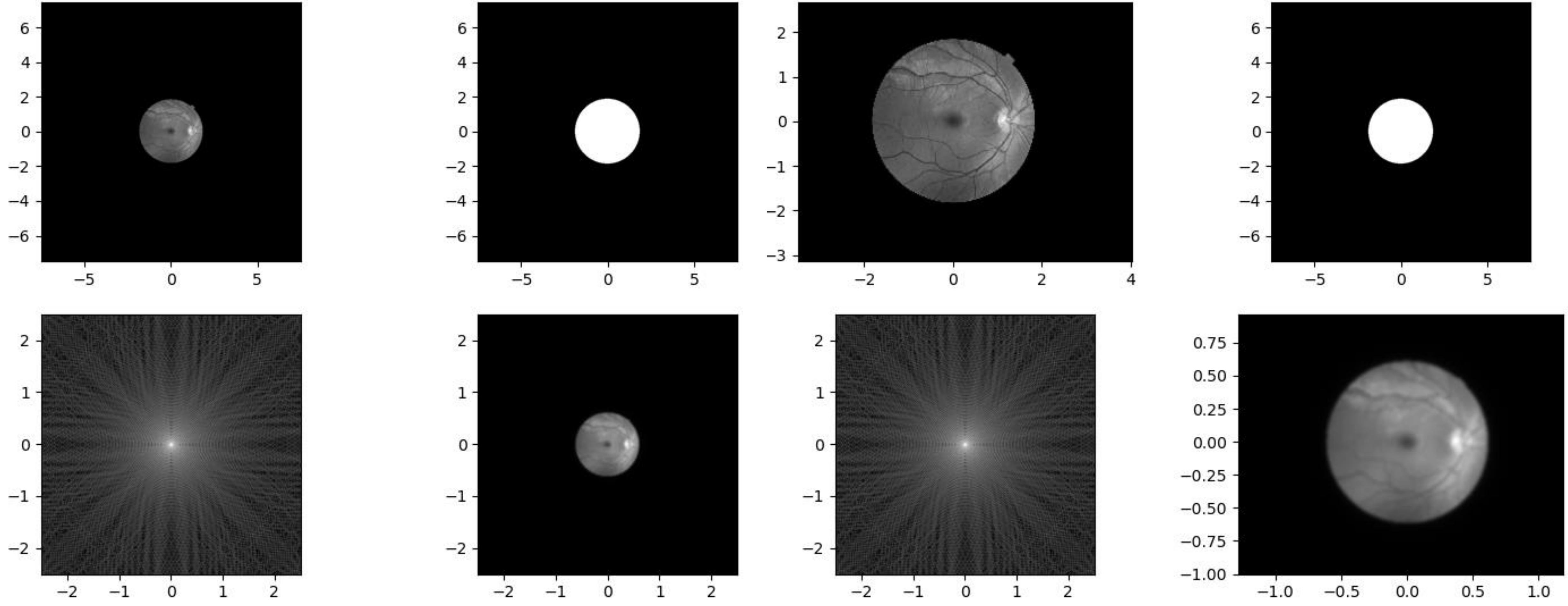
Using the scale property of the Fourier transform, the length of the window side in the image plane $L' = \lambda s \; N / L$

Exercise: check the Airy radius r_A is in agreement with the scale criteria.

Simulation of defocusing in a microcope



Length of the window describing the exit pupil L	15 mm
Radius of the exit pupil R_0	up to L/4
Image distance s	250 mm
Wavelength λ	500 nm



Reconstruction filters

$$D = \mathrm{FT}[d] \qquad H = \mathrm{FT}[\mathrm{PSF}]$$

Inverse filter $d_r = \mathrm{TF}^{-1}[DF_{\mathrm{I}}] = \mathrm{TF}^{-1}\left[D\frac{1}{H+k}\right]$

Pseudo-analytical solution

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

Filter that produces the minimal distance $\|d_r - d\|$

Lucy- Richardson $d_{k+1} = \left| d_k \left[\frac{d}{d_k * |\mathrm{PSF}|^2}\right] * |\mathrm{PSF}|^2 \right|$ with $d_0 = d$

$$\mathrm{PSF}(x,y) = \mathrm{circ}\left(\frac{R}{R}\right) \Rightarrow d_1 = \left| d_0 \left[\frac{d_0}{d_0 * \mathrm{circ}(r/R)}\right] * \mathrm{circ}(r/R) \right|$$

Circular convolutions

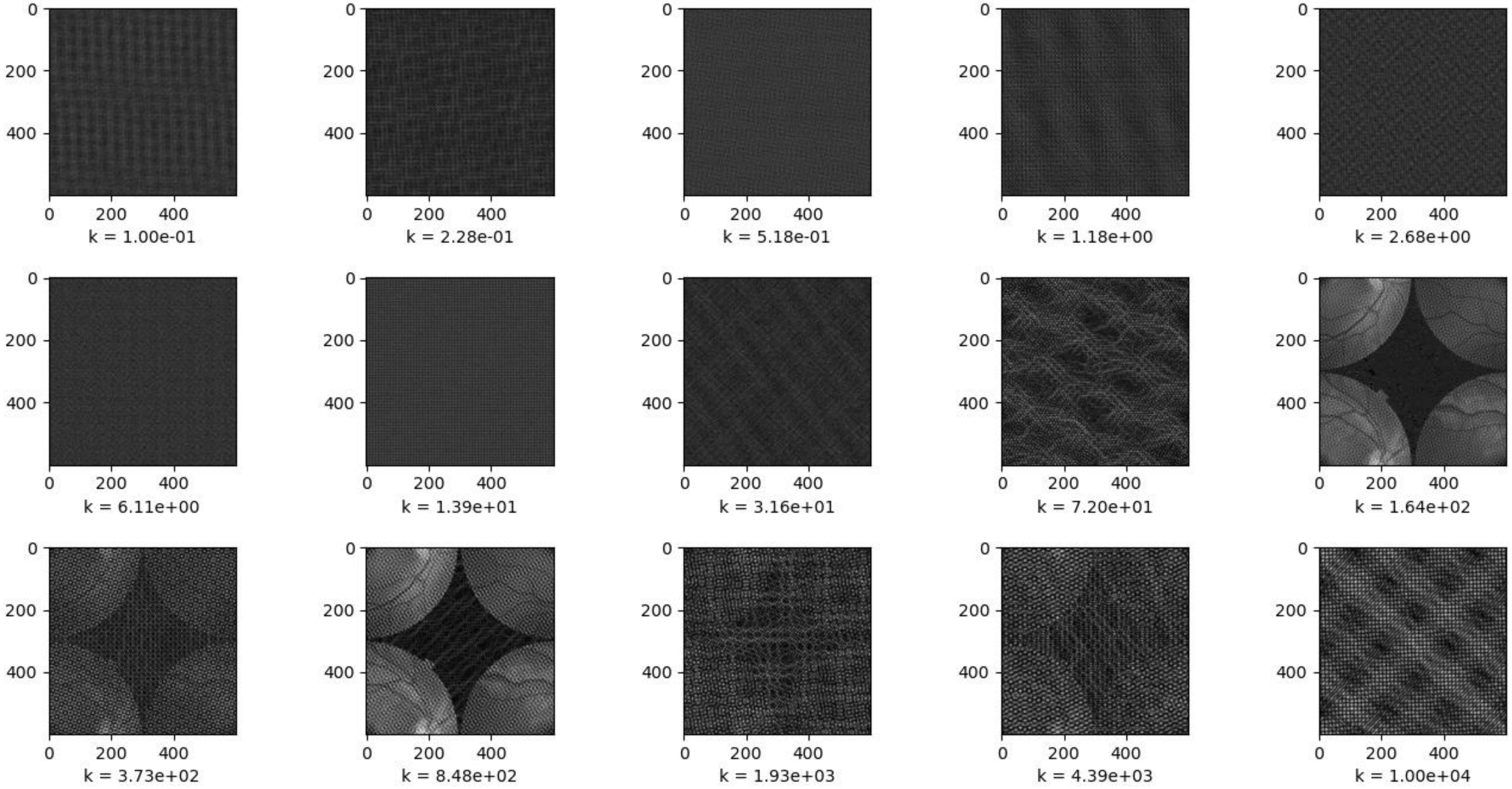
Circular convolution

From Wikipedia, the free encyclopedia

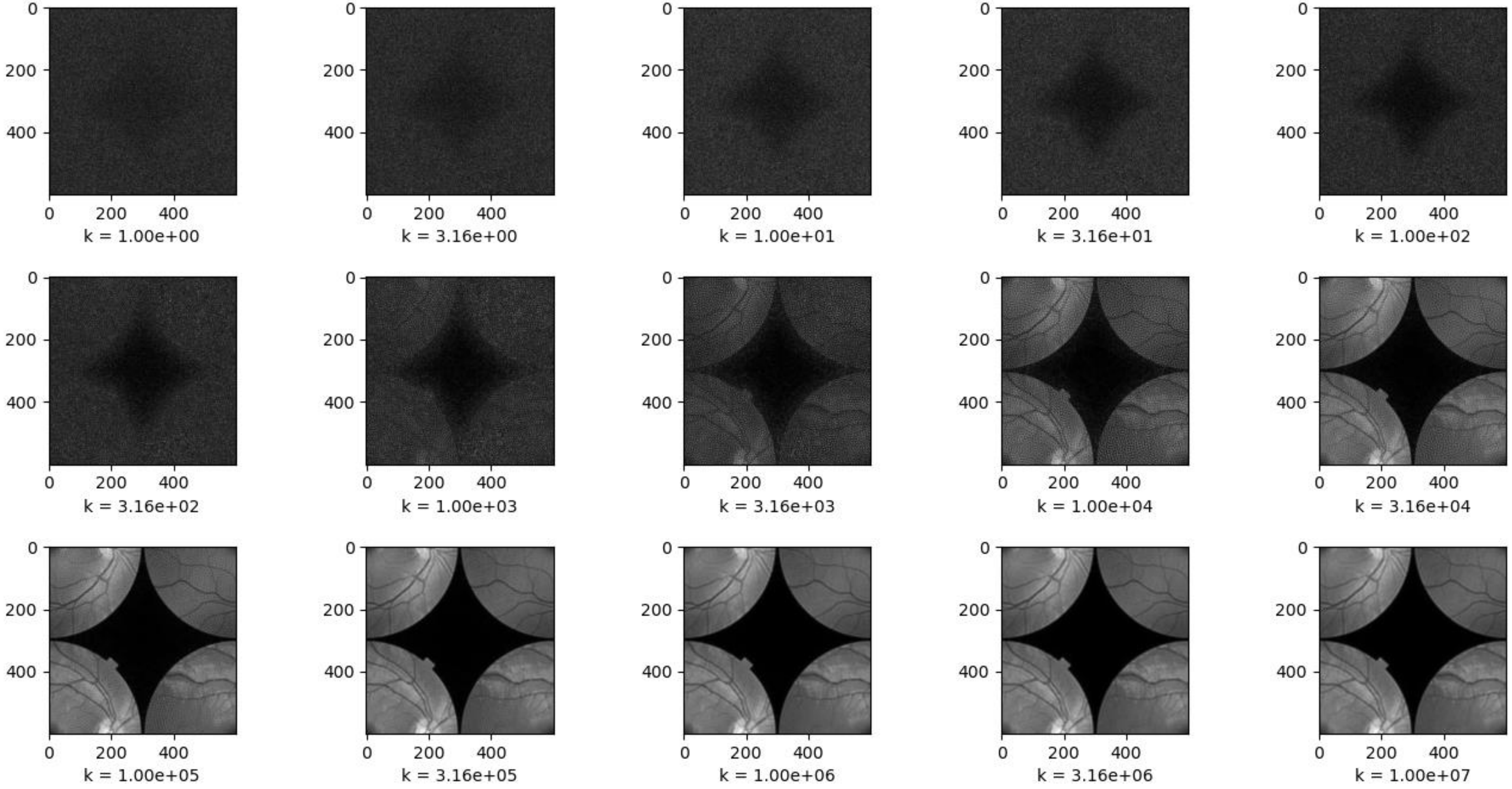
Circular convolution, also known as **cyclic convolution**, is a special case of **periodic convolution**, which is the convolution of two periodic functions that have the same period. Periodic convolution arises, for example, in the context of the discrete-time Fourier transform (DTFT). In particular, the DTFT of the product of two discrete sequences is the periodic convolution of the DTFTs of the individual sequences. And each DTFT is a periodic summation of a continuous Fourier transform function (see DTFT § Definition). Although DTFTs are usually continuous functions of frequency, the concepts of periodic and circular convolution are also directly applicable to discrete sequences of data. In that context, circular convolution plays an important role in maximizing the efficiency of a certain kind of common filtering operation.

Excerpt from https://en.wikipedia.org/wiki/Circular_convolution

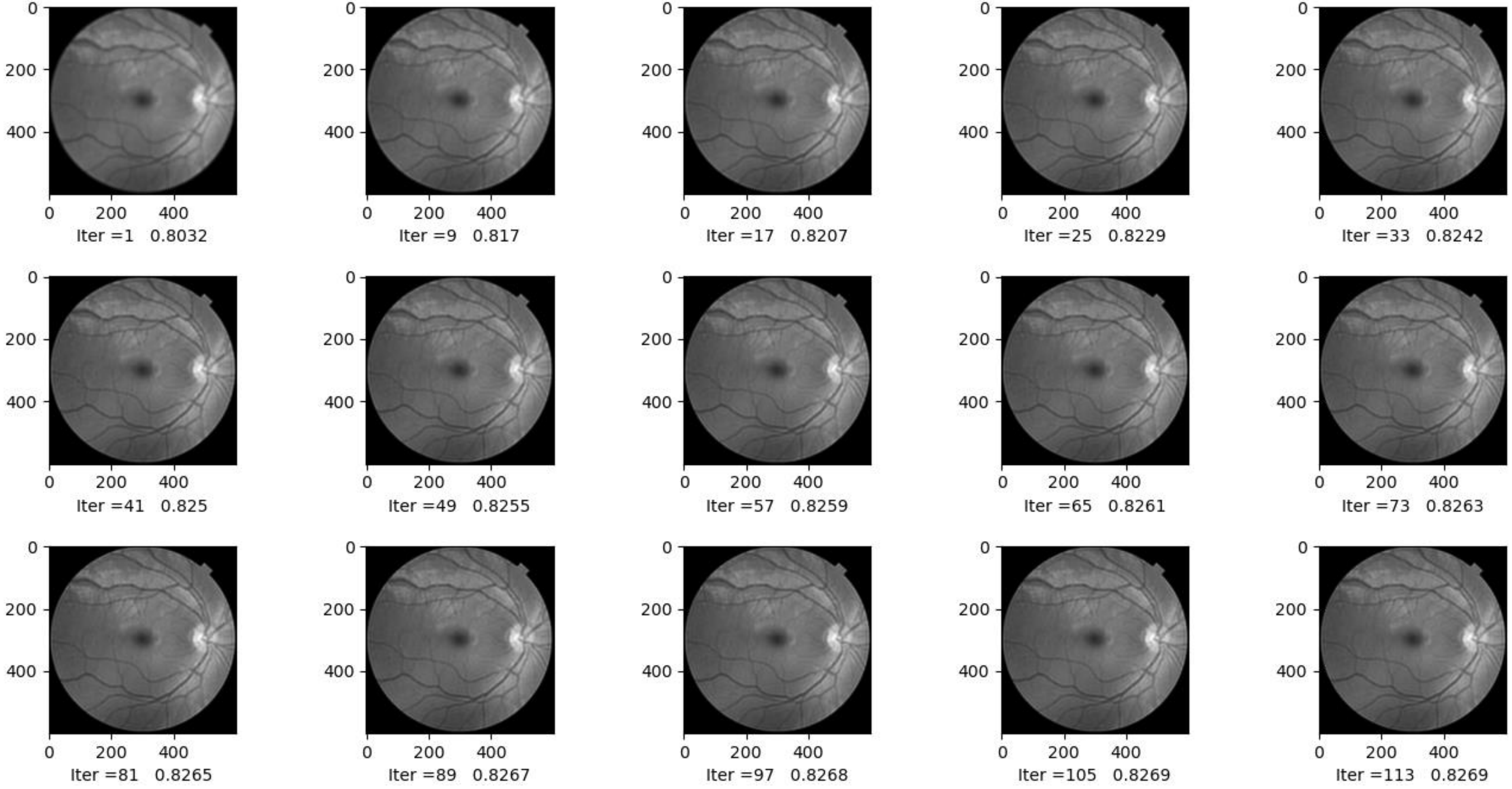
This effect can be simple removed by using an extra `fft.ifftshift`
The problem is dependent on the extent of the FT of the images involved.



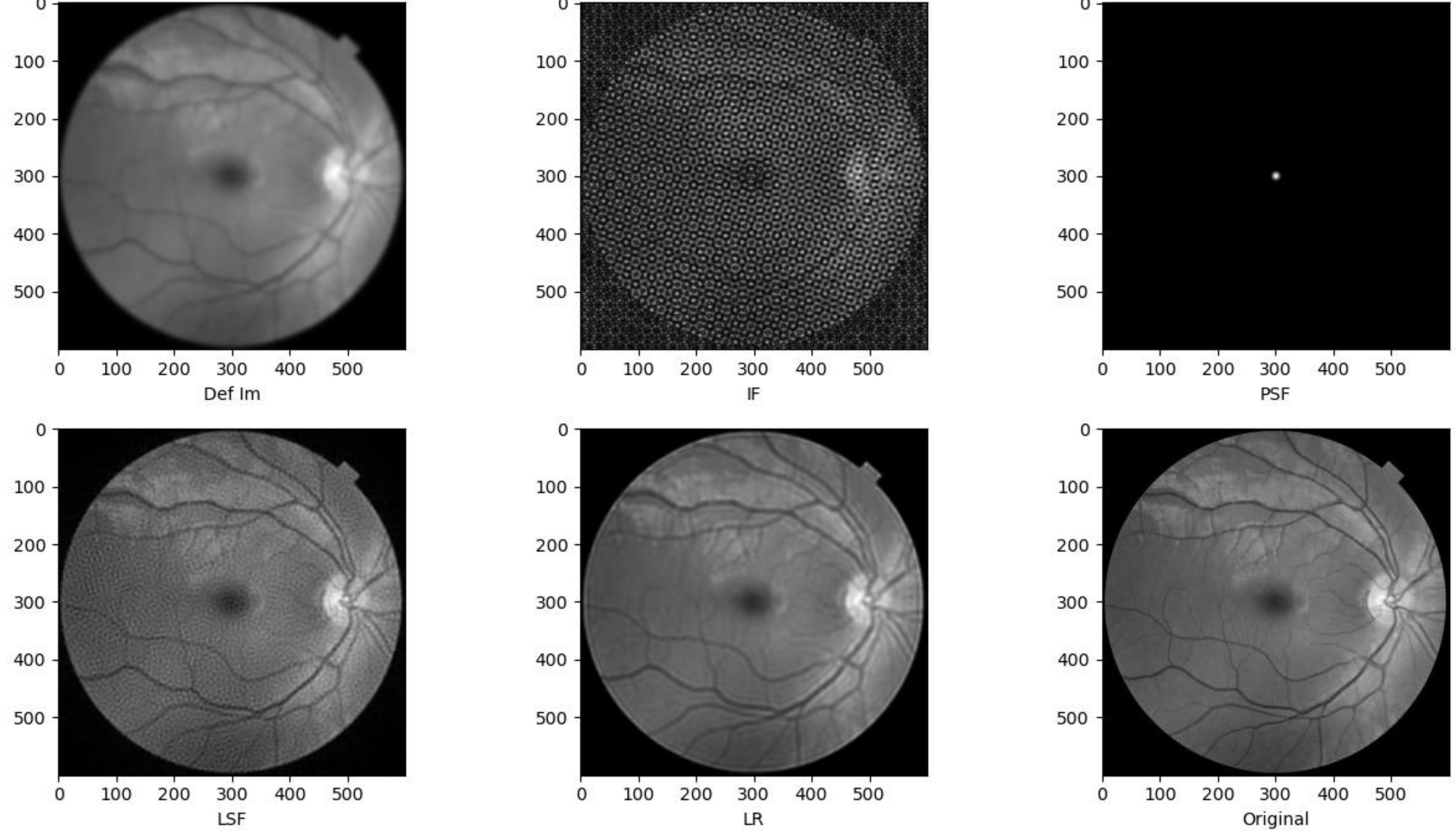
Inverse filter (without the extra `fft.ifftshift`)



Least squares filter (without the extra `fft.ifftshift`)



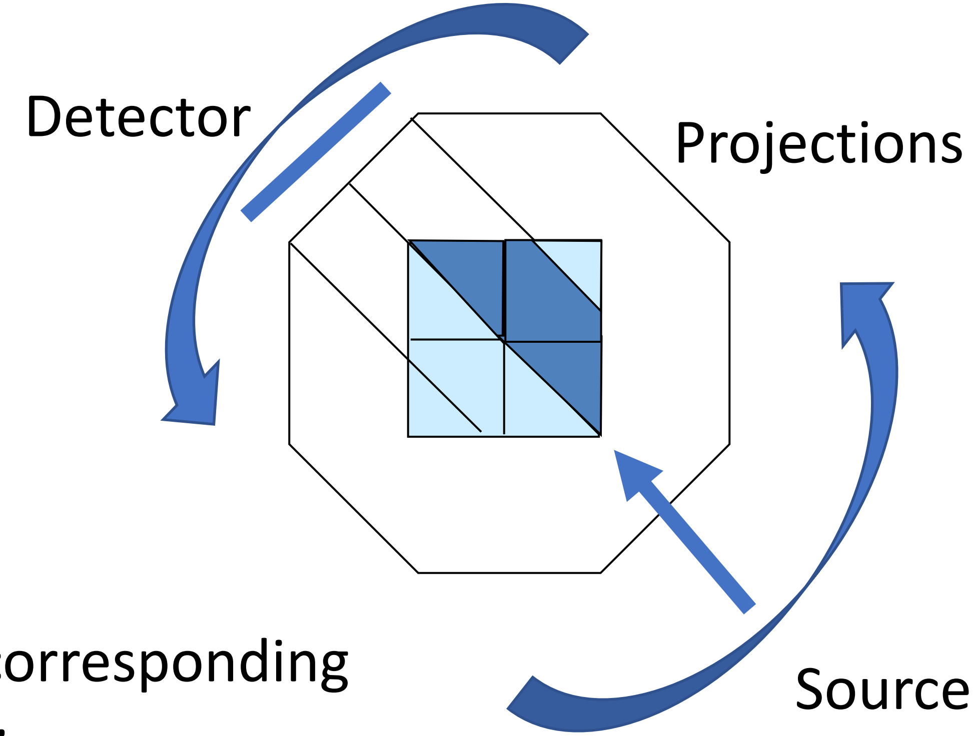
Lucy-Richardson filter


Def Im
IF
PSF
LSF
LR
Original

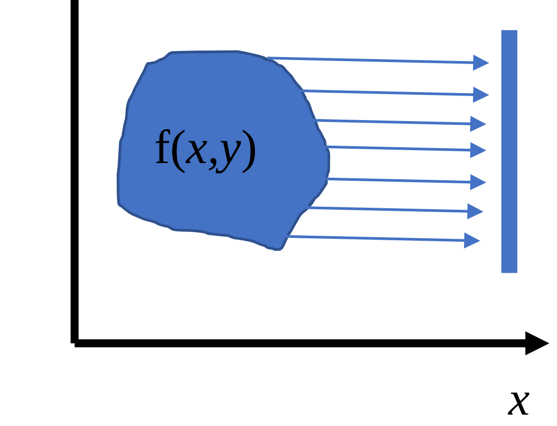
Lab #7: Radon transforms and the Projection-Slide Theorem.

A 2D object $f(x,y)$ can be determined from a set of 1D projection.
This problem is related with computer tomography: a rotating (X-ray) source + detectors measure the project

We calculate all possible projections $p_\theta(y)$
this is equivalent to rotate the object
while the detector is parallel to axis y.



$$p_{\theta=0}(y) = \int f(x,y)dx$$



We introduce the corresponding Fourier transforms:

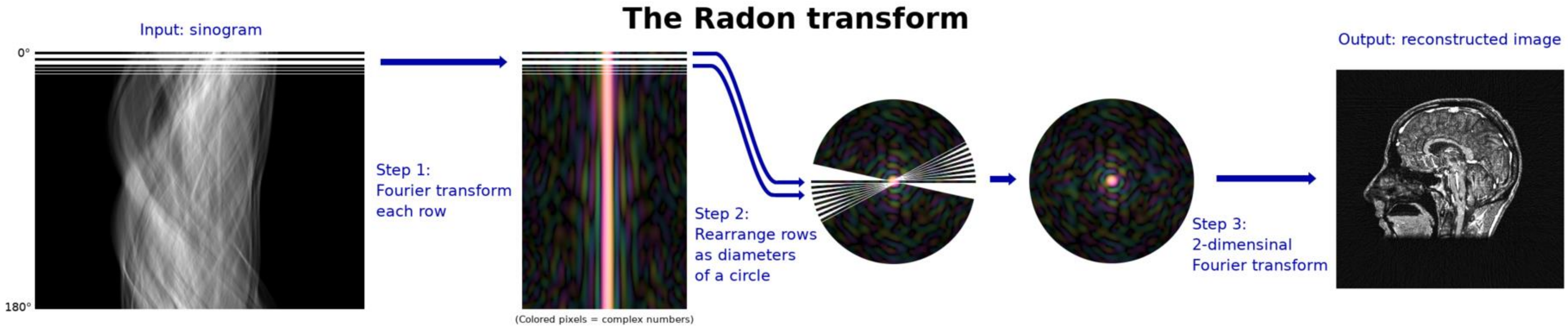
$$P_\theta(v) = \mathrm{FT}\left[p_\theta(y)\right]$$

The set of projections $p_{\theta=0}(y) = \int \mathrm{f}(x, y)dx$ is used to produce the so-called sinogram

Some noise (Poison) can be added to simulate the effect of real detectors.

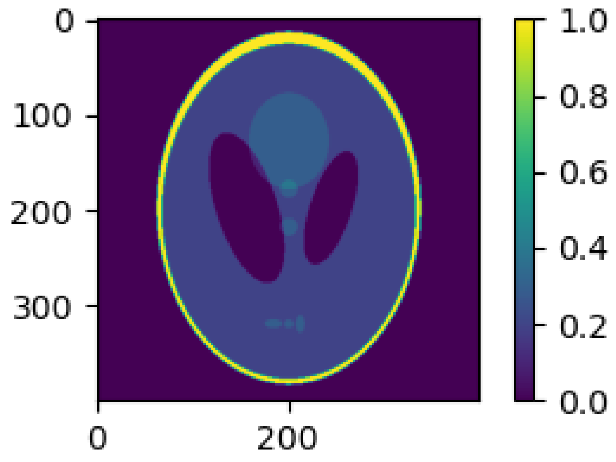
In summary, the forward Radon transform is calculated just by rotating the object a certain angle (form 0 to 180º) and calculating the projection on the x direction.

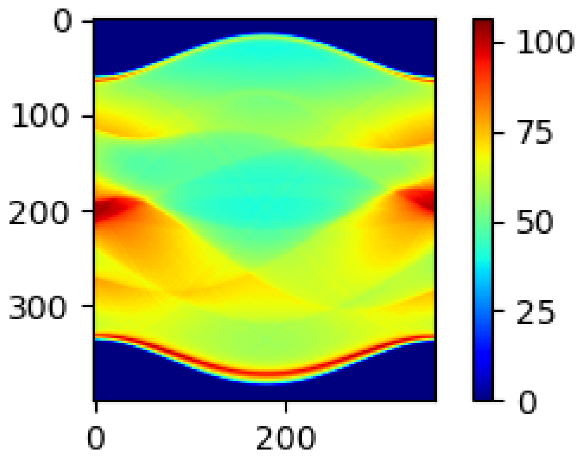
The problem arises when trying to calculate the inverse Radon transform (from projections to the 2D object).

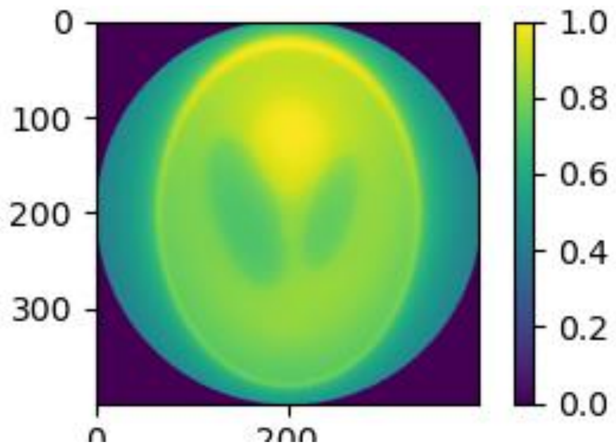


Filtered Back Projection algorithm

1. Calculate the 1D Fourier transform of every slice of the sinogram $P_\theta(v) = \mathrm{FT}\left[p_\theta(y)\right]$
2. Rearrange the Fourier transforms according a polar geometry
3. Calculate the 2D inverse Fourier transform.







Forward RT

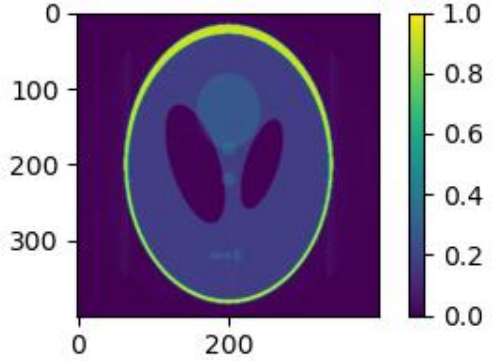
Inverse RT

The result is very poor. Why?

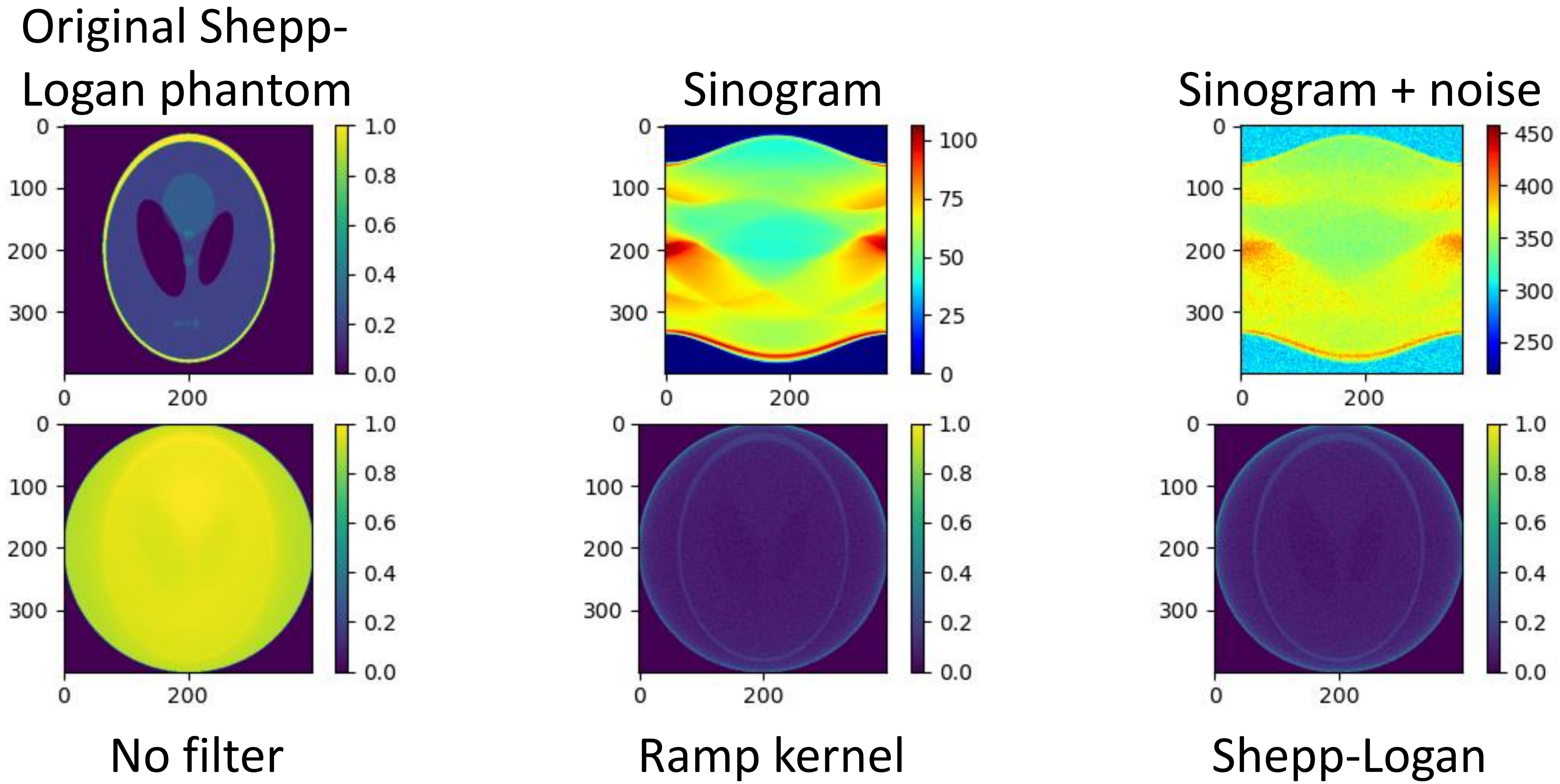
In the calculation of the IRT we missed the Jacobian term for the integration, i.e.:

$$dxdy = r\, drd\theta$$

We have to multiply the distribution obtained from the FT of the sinogram by a term r (a.k.a., the ramp kernel). Now, reconstruction is undistinguishable from the original.



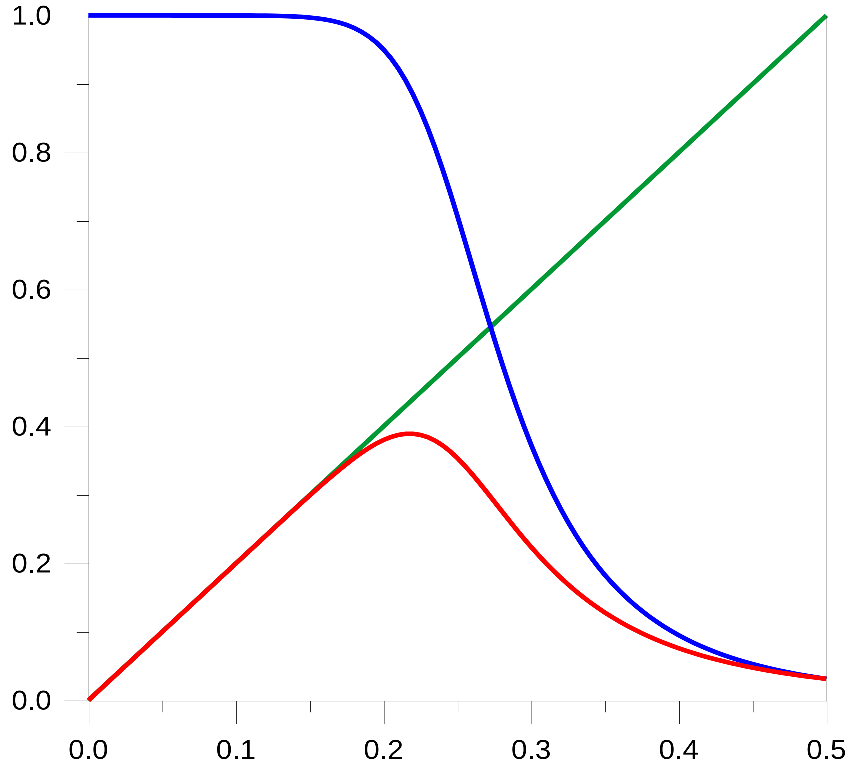
In realistic conditions, projections are affected by noise. Multiplying the FT term by r enhances high frequency information and thus, the effect of noise.
Accordingly, other filters [pass band] have been suggested.
See https://scikit-image.org/docs/dev/_images/sphx_glr_plot_radon_transform_002.png



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.



Other widely used kernels 1. Shepp-Logan $k(w) = \operatorname{sinc}\dfrac{w}{2w_c}$

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

Forward and Inverse Radon transforms implemented in skimage

https://scikit-image.org/docs/dev/auto_examples/transform/plot_radon_transform.html

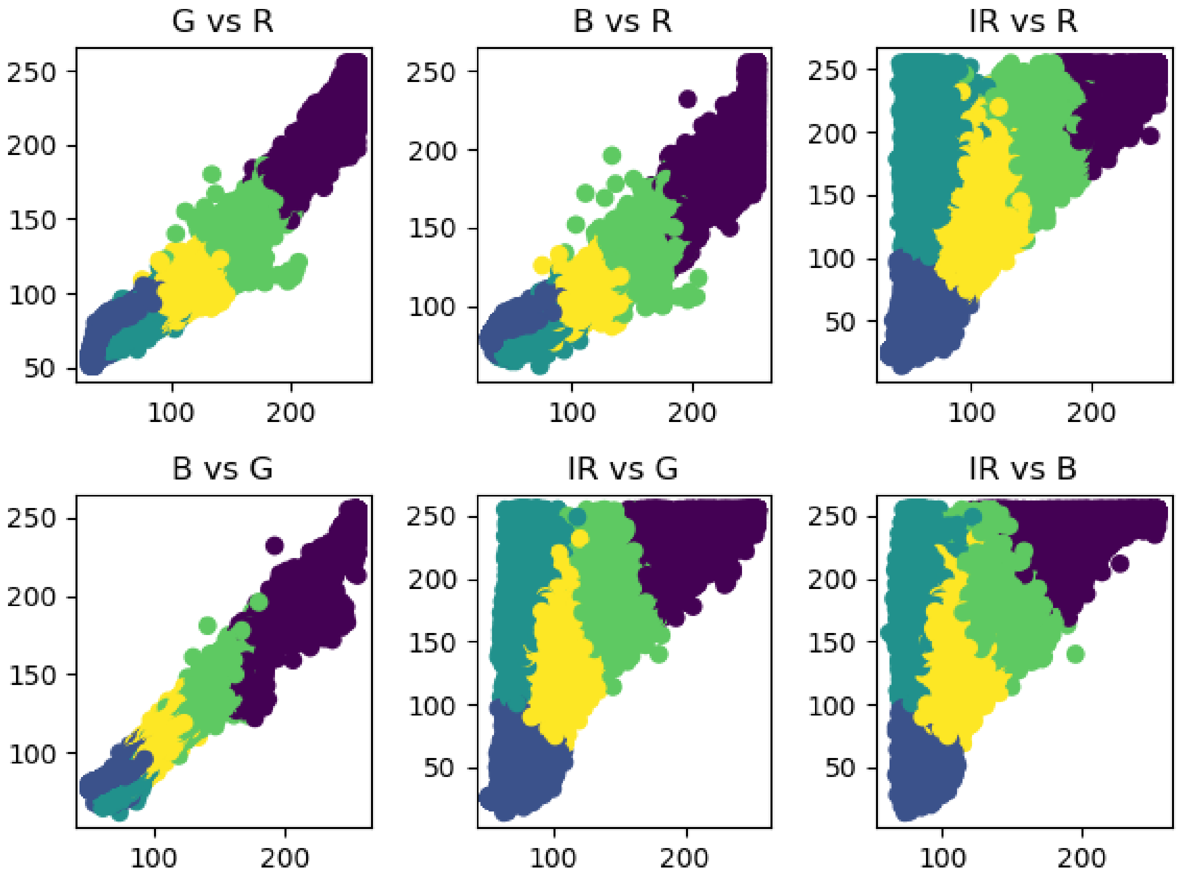
https://scikit-image.org/docs/dev/api/skimage.transform.html#skimage.transform.radon

https://scikit-image.org/docs/dev/api/skimage.transform.html#skimage.transform.iradon

Lab #8: K-means clustering

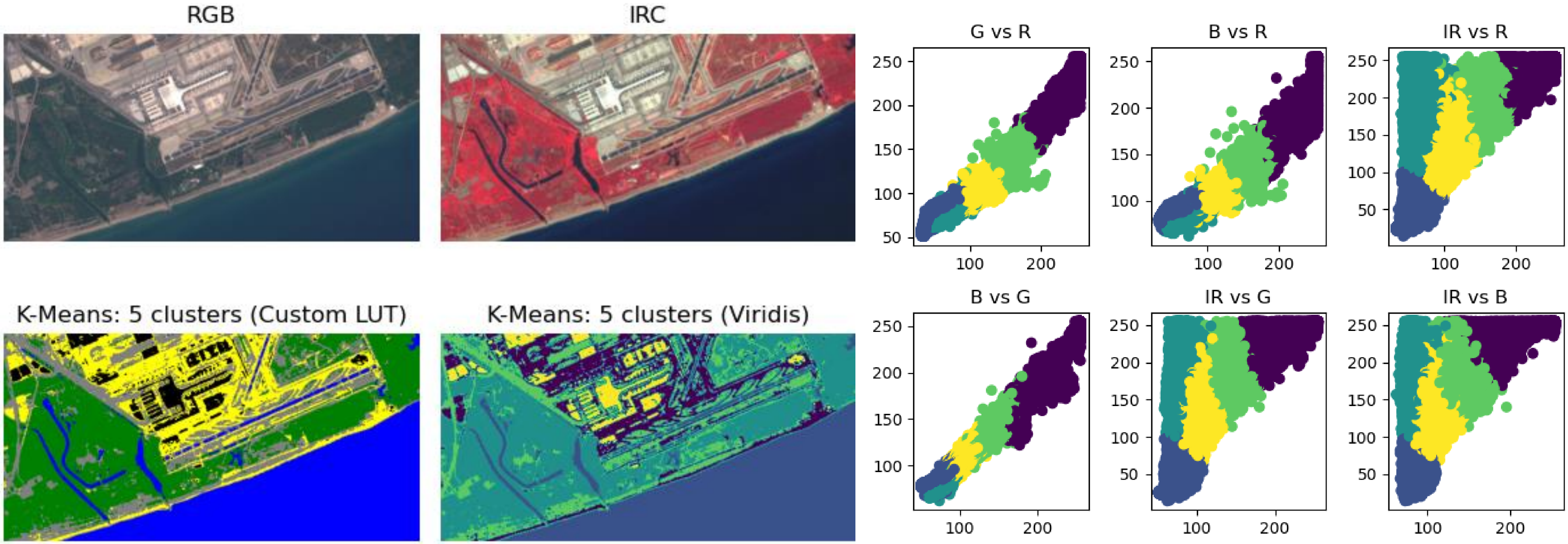
Goal: Segmentation of an image according to the color (I, R, G and B).

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



The number of clusters is selected by the programmer (e.g.: 5).

K-means clustering



Infrared-red-green: the red channel is replaced with near infrared. This combination is often used to detect vegetation (because of it is highly reflective in near IR).

- **Machine learning** algorithms build a model based on sample data, known as training data, in order to make predictions or decisions. https://en.wikipedia.org/wiki/Machine_learning

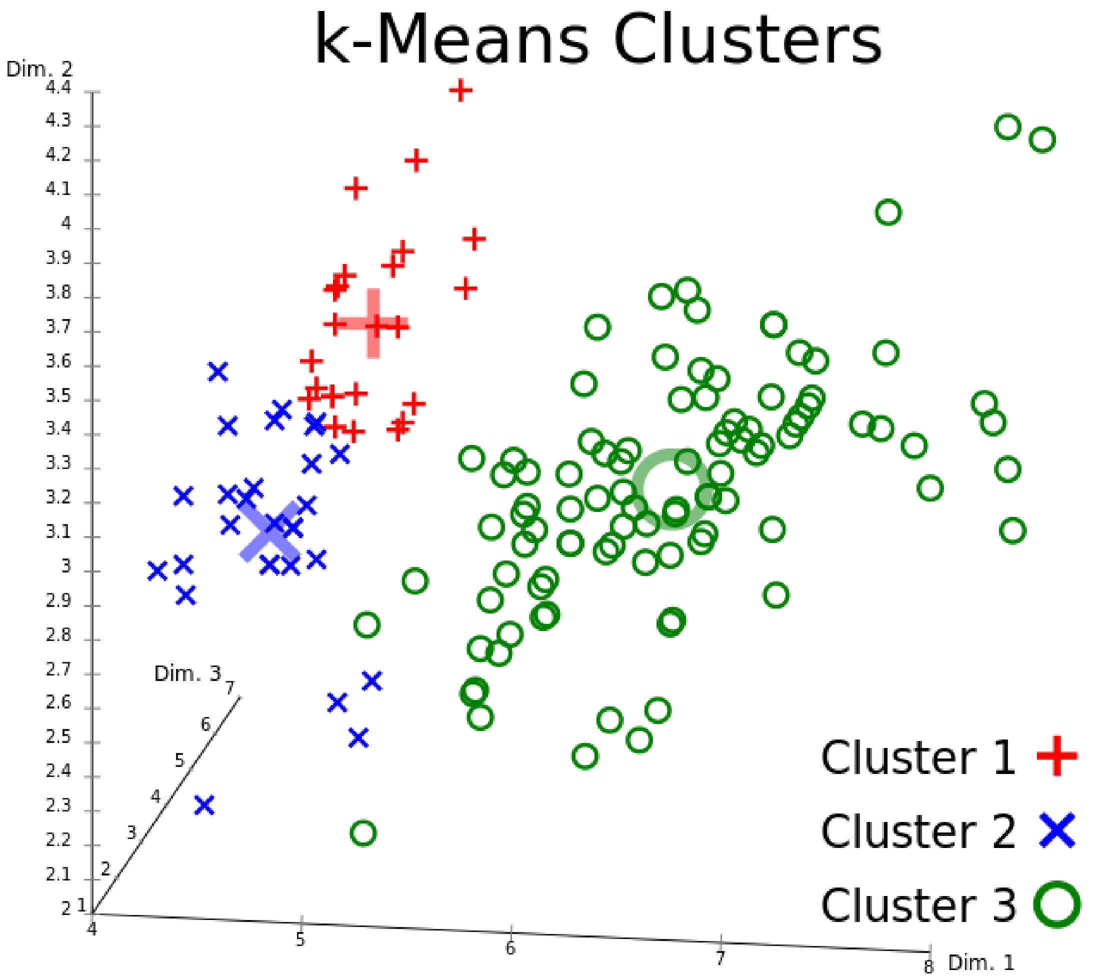
- The K-means algorithm clusters data by trying to separate samples in K groups, minimizing a criterion known as the ***within-cluster sum-of-squares***.
- **This algorithm requires the number of clusters to be specified**.
- It is an unlabeled, **not-supervised method**.
- It scales well to large number of samples.

- The K-means algorithm divides a set of N samples **X** (points or pixels of the image) into K disjoint clusters **C**, each described by the mean μ of the samples in the cluster. These means are commonly called the “cluster centroids”
- Note that the coordinates of the centroids are not, in general, points from **X.**

https://scikit-learn.org/stable/modules/clustering.html#k-means

K-means workflow

1. Determine how many clusters K are required
2. Set, at random, the position of the K centroids.
3. Calculate the Euclidian distance between the N points of the dataset and the K centroids (***within-cluster sum-of-squares)***. The point belongs to the cluster to the centroid is closer centroid.
4. A new centroid is determined using the set of points that belongs to each cluster. Repeat 3 until no changes in the position of the centroid is detected.



https://en.wikipedia.org/wiki/K-means_clustering#/media/File:Iris_Flowers_Clustering_kMeans.svg

How to arrange the data to use K-means

- `dataK` is an array containing N rows describing the samples and M columns describing the dimensionality of the problem
- In our problem the images are: 250 x 500 píxels, 4 channels.
  `dataK` is a 2D array: 250 x 500 = 125000 (rows) and 4 (columns)
- use `.flatten(), .reshape()` or `np.reshape()`

```
from sklearn.cluster import KMeans
# system training
kmn = KMeans(n_clusters=5, init='k-means++',
                   random_state=0).fit(dataK)
# a label (0 to 4) is assigned to e11ach sample (row)
labels = kmn.predict(dataK) # size = 125000
# centroids
centroids = kmn.cluster_centers_
# from 1d-array to 2d-array
imRes = np.reshape(labels, [250, 500])
```

Custom colors

```
Import skimage.color as color
newImage = color.label2rgb(imRes, colors=['yellow', 'blue’,
                                   'green', 'gray', 'black'])
```

How to produce a scatterplot

```
# a plt.scatter ≠ plt.plot
labels = kmn.predict(dataK)
plt.scatter(dataK[:,0], dataK[:,1], c = labels, cmap='jet')
```


dataK - NumPy object array

	0	1	2	3
0	100	95	101	113
1	97	90	97	129
2	102	95	102	118
3	105	95	103	107
4	114	104	112	123
5	129	122	129	140
6	128	123	129	133
7	120	115	121	126
8	108	105	112	136

Format Resize Background color Save and Close

labels - NumPy object array

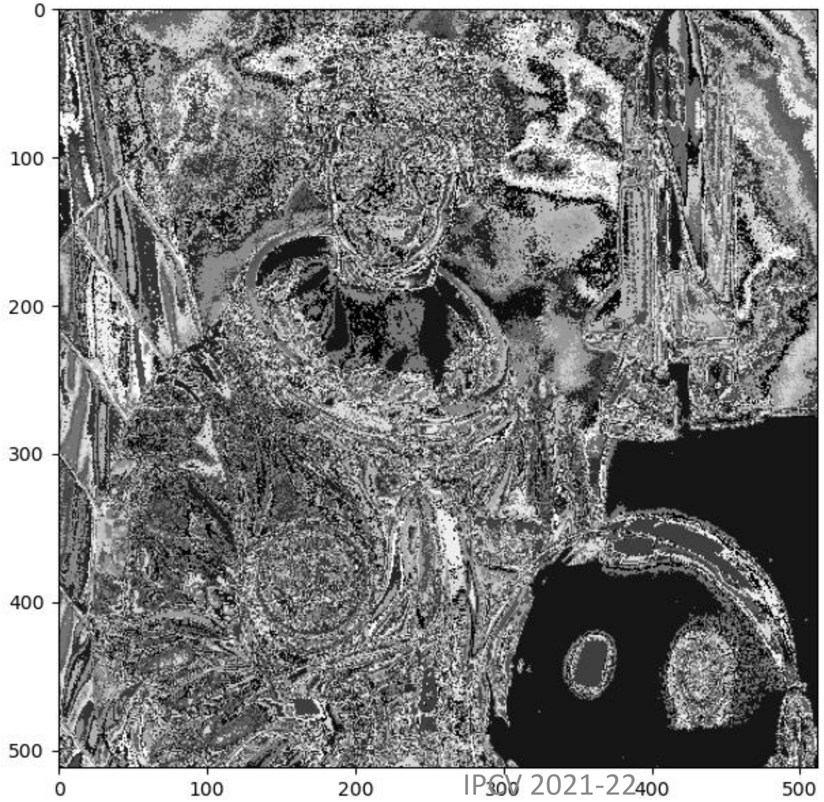
	0
0	3
1	3
2	3
3	3
4	3
5	3
6	3
7	3
8	3

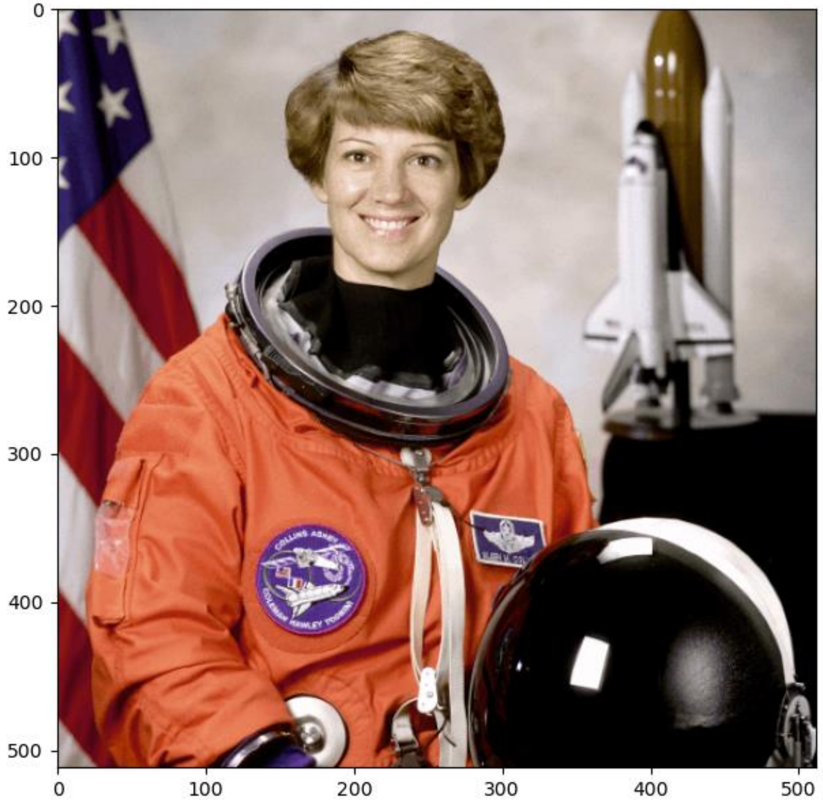
Format Resize Background color Save and Close Close

Revisiting indexed color

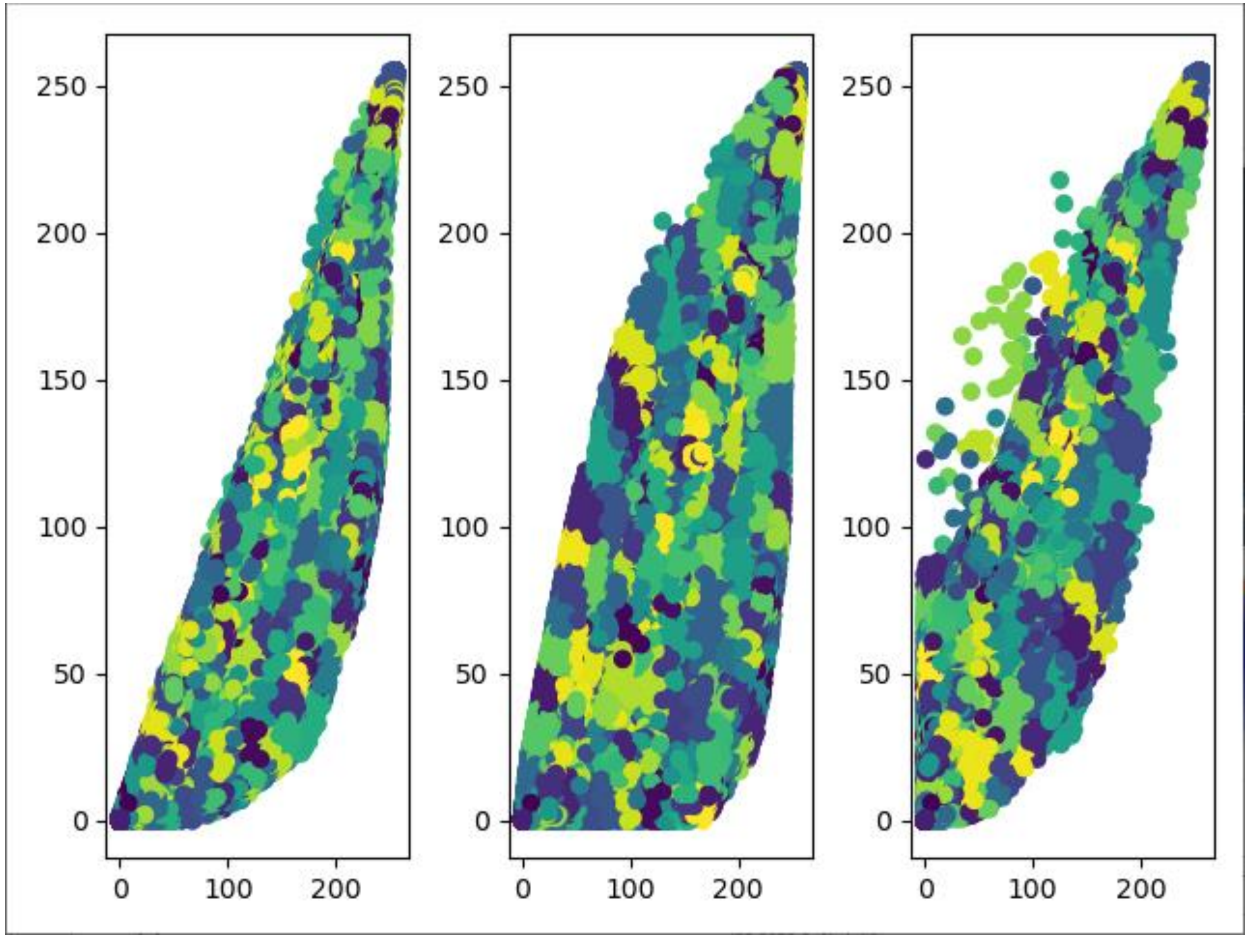
Goal: to produce a 216-level indexed image using K-means

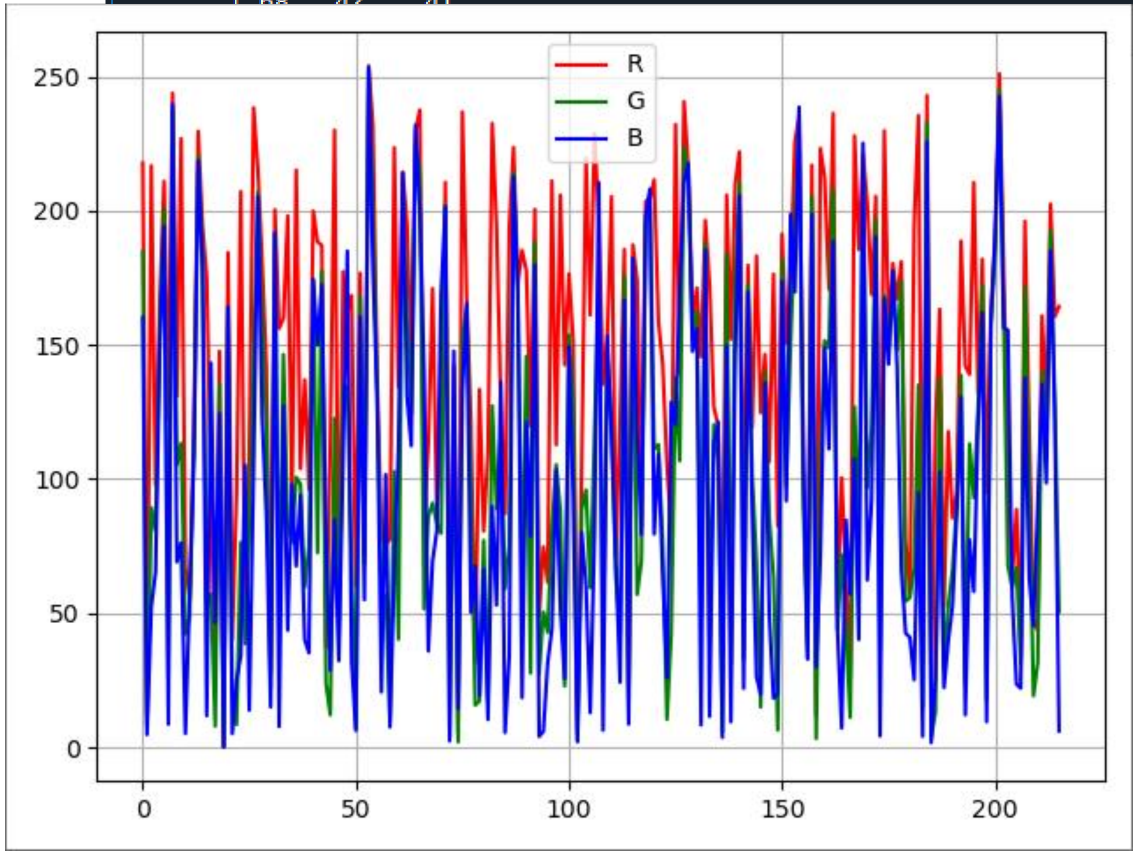
`sklearn.utils.shuffle` is great to speed up the process
`.cluster_centers_` provides de coordinates of the centroids. This array might be considered as the colormap that transforms the index of the 2d array into a color image.





```
In [10]: index = np.linspace(0, 215, 216, dtype=np.uint8)
    ...: lut = np.uint8(kmeans_res.cluster_centers_[index])

In [11]: lut
Out[11]:
array([[217, 184, 160],
       [ 11,   8,   4],
       [216,  89,  52],
       [ 97,  80,  65],
       [167, 157, 149],
       [210, 200, 194],
       [122,  25,   8],
       [243, 237, 239],
       [130, 105,  69],
       [226, 113,  76],
       [ 68,  42,   4],
```





Project #9. Analysis of a wildfire burned area.
(instead of developing short exercises describing a single technique, we introduce new concepts within the context of a more complex problem).

- Multispectral imaging
- Geometrical transformations
- Fourier transforms and correlation
- Mathematical morphology.

Target: determine the surface temperature after the wildfire and the surface burned using image analysis.

Landsat 8 spectral bands

	OLI	
	Spectral Band (Wavelength)	Spatial Resolution
1 - Coastal- Aerosol	0.433 – 0.453 µm	30 m
2 - Blue	0.450 – 0.515 µm	30 m
3 - Green	0.525 – 0.600 µm	30 m
4 - Red	0.630 – 0.680 µm	30 m
5 - Near Infrared (NIR)	0.845 – 0.885 µm	30 m
6 - Short Wavelength Infrared (SWIR1)	1.560 – 1.660 µm	30 m
7 - Short Wavelength Infrared (SWIR2)	2.100 – 2.300 µm	30 m
8 - Panchromatic	0.500 – 0.680 µm	15 m
9 - Cirrus	1.360 – 1.390 µm	30 m
	TIRS	
10 - Long Wavelength Infrared (LWIR1)	10.30 – 11.30 µm	100 m
11 - Long Wavelength Infrared (LWIR2)	11.50 – 12.50 µm	100 m

Project #10. Automatic diagnostic using an X-ray images dataset: Image classification using machine learning

In this lab we analyze image classification using Support Vector Machine. We use a large (1 GB!) X-Ray dataset that comprises 5826 images divided in four subsets:

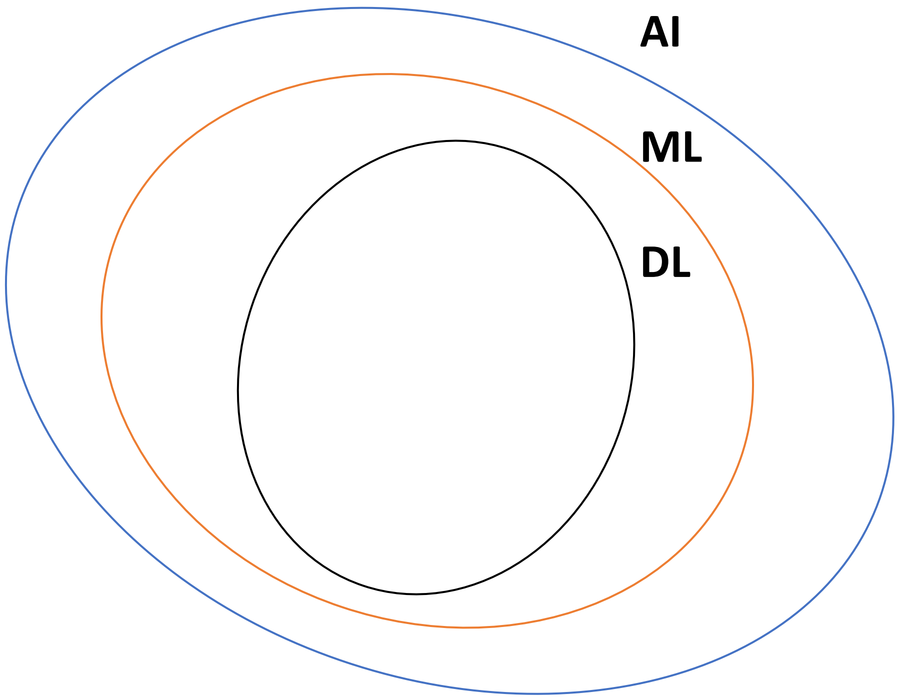
Train – normal: 1275 images
Train – pneumonia: 3883 images
Test – normal: 234
Test – pneumonia: 390

The image set is not split
Train and test sets are selected at random
Use `sklearn.utils.shuffle`

The target of this lab is to automatically determine whether an X-Ray image displays signs of pneumonia or not.

Glossary

Artificial intelligence is intelligence demonstrated by machines
https://en.wikipedia.org/wiki/Artificial_intelligence



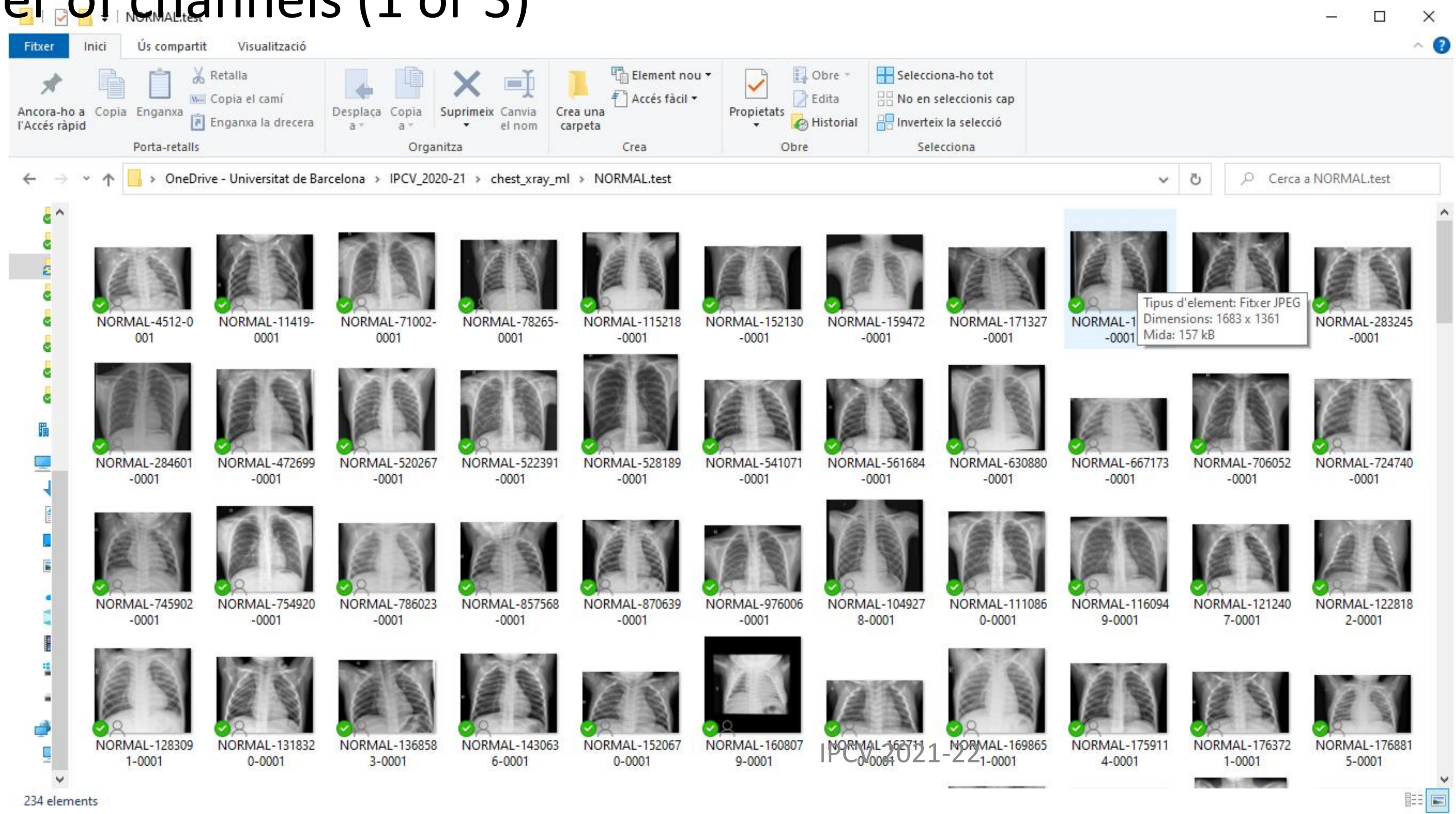
Machine learning is the study of computer algorithms that improve automatically through experience and by the use of data. It is seen as a part of artificial intelligence. Machine learning algorithms build a model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so
https://en.wikipedia.org/wiki/Machine_learning

Deep learning is part of a broader family of machine learning methods based on artificial neural networks. Learning can be supervised, semi-supervised or unsupervised.
Artificial neural networks were inspired by information processing and distributed communication nodes in biological systems. The adjective "deep" refers to the use of multiple layers in the network
https://en.wikipedia.org/wiki/Deep_learning

Data mining is a process of extracting and discovering patterns in large data sets involving methods at the intersection of machine learning, statistics, and database systems.
https://en.wikipedia.org/wiki/Data_mining

The full dataset can be downloaded from https://www.kaggle.com/tolgadincer/labeled-chest-xray-images

Registration is required to download the dataset.
The dataset is LABELLED! We can use a supervised strategy.
Inhomogeneous dataset: Images of different sizes and, despite they look gray-level, number of channels (1 or 3)



Search

Data Tasks Code (16) Discussion Activity Metadata Download (1 GB) New Notebook

Context

Pneumonia is an infection that inflames the air sacs in one or both lungs. It kills more children younger than 5 years old each year than any other infectious disease, such as HIV infection, malaria, or tuberculosis. Diagnosis is often based on symptoms and physical examination. Chest X-rays may help confirm the diagnosis.

Content

This dataset contains 5,856 validated Chest X-Ray images. The images are split into a training set and a testing set of independent patients. Images are labeled as (*disease*:NORMAL/BACTERIA/VIRUS)-(*randomized patient ID*)-(*image number of a patient*). For details of the data collection and description, see the referenced paper below.

According to the paper, the images (anterior-posterior) were selected from retrospective cohorts of pediatric patients of one to five years old from Guangzhou Women and Children's Medical Center, Guangzhou.

A previous version (v2) of this dataset is available here: https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia. Note that the files names are irregular in v2, but they are fixed in the new version (v3).

Inspiration

This data will be useful for developing/training/testing classification models with convolutional neural networks.

Acknowledgements

Training and test datasets.

The train set is used to produce a model whereas the test sets help us determine the reliability of its classification ability.

The dataset should be prepared in such a way that sklearn functions can be able to handle it (i.e. images as 1-dim arrays) : variables X_train, Y_train, X_test, Y_test.

The training dataset X for N samples is described by means of a N x M array: row i represents a M-dimensional vector containing the features that describe sample (image) i. Classes are labelled as integer numbers and arranged as a column vector of dimension N;

$$\mathbf{X}\_\text{train} = \begin{pmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1m} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2m} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3m} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{nm} \end{pmatrix} \quad \mathbf{Y}\_\text{train} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}.$$

Labels y_i take values 0 (for normal images) or 1 (for pneumonia images).

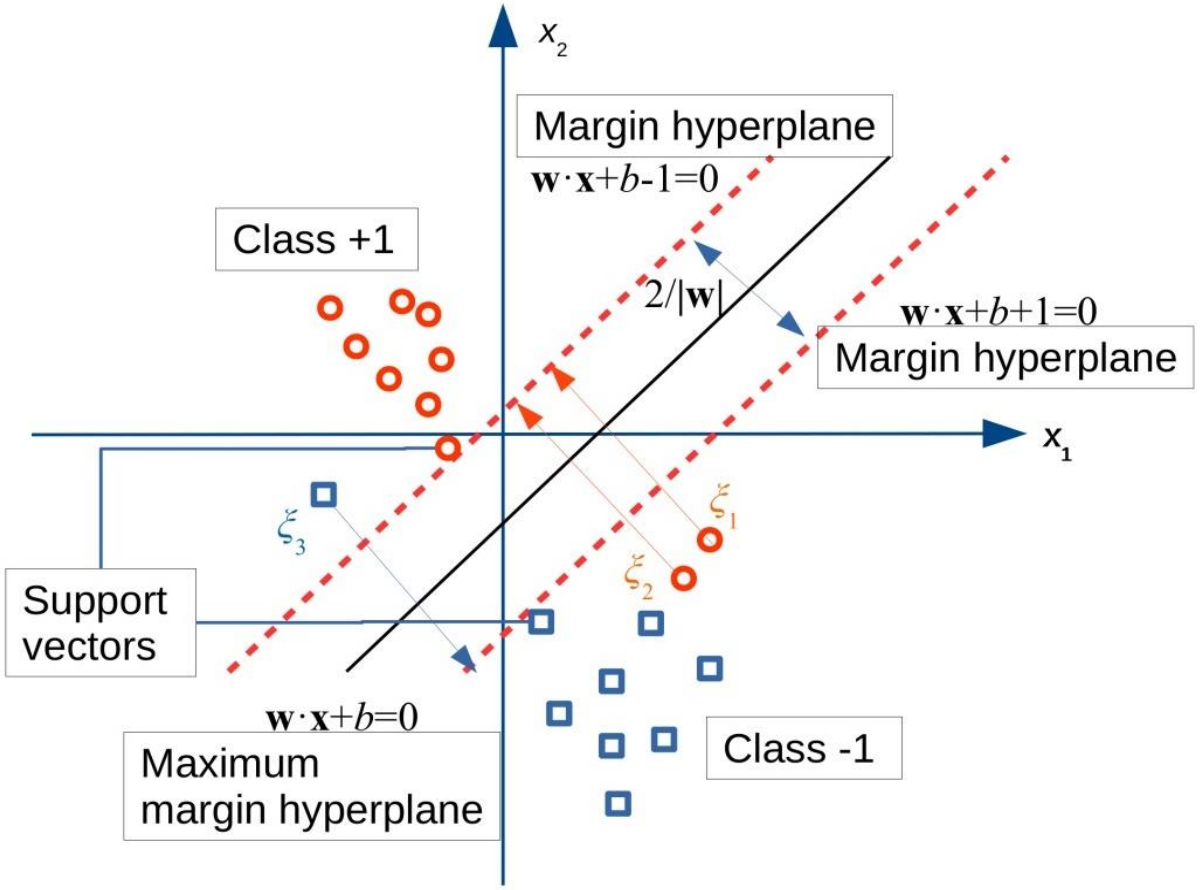
Data preprocessing

- In order to produce matrix **X**_train, all images are zoomed to the same size (e.g. 100x100 pixels) and then, flatten to 1 dimension.
- Every image is described by a vector of 10000 features.
- This set of 1-dim image arrays are combined to form the 2D-array **X**_train.

Support Vector Machine (SVM)

- SVM is a classification method based on determining the best hyper-surface able to distinguish among two classes.
- The use of SVM is suggested when the number of samples in the datasets is small but they present high dimensionality (dimensions higher that the number of samples) (4000 training images with 10000 feature each).
- However, this might produce overfitting (classification work for the training set but not necessarily for the test set).
- It is memory efficient and flexible (different classification kernels)
- SVM is a supervised method (uses labeled datasets to train the algorithm).

SVM searches for the hyperplanes that verify that the distance between the hyperplane and the nearest point of either group is maximized. These points are the so-called *support vectors.* The hyperplanes that contains the support vectors are named *margin hyperplanes.*



The objective of the algorithm is to find the hyperplane whose distance between margin hyperplanes is maximized, with a minimum number of samples not properly classified

1. Prepare matrix X_train

- Use `import glob` to generate a list of files in a folder
- Images are gray level: some are 8 bit, and some others, 24 bits.
- Use `scipy.ndimage.zoom` to produce arrays of the same number of pixels
- Normalize image values to 1 (`/ 255.`)

2. Train the system using SVM.

Training with SVM in sklearn is not complicated. Note the combined use of functions `svm.SVC()` and `clf.fit()`. Very similar to k-means.

```
from sklearn import svm
clf = svm.SVC(kernel=…)                  # creating an SVM Classifier
clf.fit(X_train, Y_train)                # train the model using the
                                         # training set
Y_pred = clf.predict(X_test)    # predict the response for the
                                # test dataset
```

```
from sklearn.metrics import accuracy_score, confusion_matrix
```

Performance metrics: sensivity and specificity.

How good these results are? Calculate the confusion matrix, the accuracy, and the sensivity (true positive rate) and the specificity (true negative rate).

- Confusion matrix:

TP: True Positive People correctly detected as pneumonia	**FP: False Positive** People wrongly detected as pneumonia
FN: False Negative People wrongly detected as healthy	**TN: True Negative** People correctly detected as healthy

- **Accuracy**: proportion of people properly identified among the total population tested.

$$\text{Accuracy}=\frac{\text{TP+TN}}{\text{TP+TN+FN+ FP}}$$

- **True positive rate (TPR) or sensitivity**: proportion of people with pneumonia who test positive among those who have the disease.

$$\text{TPR}=\frac{\text{TP}}{\text{TP+FN}}$$

- **True negative rate (TNR) or specificity:** proportion of healthy people who test negative among those who are healthy.

$$\text{TNR}=\frac{\text{TN}}{\text{TN+FP}}$$

Test – normal: 234
Test – pneumonia: 390

Confusion Matrix
[[132 102]
[4 386]]
Accuracy: 83%
True Positive ratio 97%
True Negative ratio 79%

Principal component analysis looks for an alternative base for reducing the dimensionality of dataset $\boldsymbol{X}$; this base corresponds to the eigenvectors of $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{X}^T$. Eigenvectors associated with high eigenvalues provide more information than those directions with small eigenvalues: at the end of the day, less relevant directions are avoided and the dataset dimensionality is reduced.