

Image Processing Tutorials
with PYTHON®

Yann GAVET
Johan DEBAYLE

creative commons

Attribution 4.0 International

(CC BY 4.0)

This is a human-readable summary of (and not a substitute for) the license, available at:

https://creativecommons.org/licenses/by-sa/4.0/legalcode

Your are free to:

Share - copy and redistribute the material in any medium or format

Adapt - remix, transform or build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

Appropriate credit — You must mention the **authors** and their host institution (**MINES SAINT-ETIENNE**).

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Credits
Cover image: pixexid, pixabay.

Contributors:

- Léo Théodon
- Victor Rabiet
- Séverine Rivollier
- Valentin Penaud–Polge
- Place your name here and get special credit if you want to participate to this book.

Table of Contents

13 Part I
Fundamentals

1 Introduction to image processing 15

33 Part II
Image Transforms and Representations

2 2D Fourier Transform 35
3 DCT: Discrete Cosine Transform 43
4 Introduction to wavelets 53
5 Multiscale Analysis 67
6 Color Spaces 75

85 Part III
Enhancement and Restoration

7 Image Enhancement 87
8 Image restoration: an enhanced median filter 95
9 Image Filtering using PDEs 107
10 Image Restoration: deconvolution 115

129 Part IV
Image Formation and Reconstruction

11 Shape From Focus 131
12 Introduction to tomographic reconstruction 139

145 Part V
Alternative Algebraic Frameworks

13 Logarithmic Image Processing (LIP) 147
14 Color LIP 155
15 GANIP 163

173 Part VI
Mathematical Morphology

16 Binary Mathematical Morphology 175
17 Morphological Geodesic Filtering 185
18 Morphological Attribute Filtering 191
19 Morphological skeletonization 195
20 Granulometry 201

207 Part VII
Segmentation

21 Histogram-based image segmentation 209
22 Segmentation by region growing 219
23 Watershed 225
24 Active contours 231
25 Segmentation of follicles 239

245 Part VIII
Feature Detection and Description

26 Hough transform and line detection 247
27 Harris corner detector 253
28 Elliptic Fourier Descriptors 259
29 Local Binary Patterns 265
30 Image Registration 273
31 Phase Correlation 285
32 Machine Learning 295

303 Part IX
Geometry, Shape Description and Measurement

33 Convex Hull 305
34 Voronoï Diagrams and Delaunay Triangulation 311
35 Alpha Shapes 319
36 Freeman Chain Code 327
37 Topological Description 337
38 Integral Geometry 343
39 Image Characterization 349
40 Shape Diagrams 355

363 Part X
Stochastic Geometry and Random Fields

41 Stochastic Geometry / Spatial Processes 365
42 Boolean Models 377
43 Geometry of Gaussian Random Fields 385
44 Stereology 391
45 Bertrand's paradox: random chords and sections 397
46 Fractal dimension 405

411 Part XI
Exams

47 Practical exam 2016 413
48 Theoretical exam 2016 415
49 Theoretical exam 2017 417

About the authors

Yann GAVET

received his "Ingénieur Civil des Mines de Saint-Etienne" diploma in 2001. He then obtained a Master of Science and a PhD thesis on the segmentation of human corneal endothelial cells (in 2004 and 2008). He is now an assistant professor at the Saint-Etienne School of Mines, where he teaches computer science and image processing to engineering and master students. He is a member of the PMDM Department of the LGF Laboratory, UMR CNRS 5307, dedicated to granular media analysis and modelisation.

He is particularly interested in the world of free (LIBRE) software in computer science. His research interests include image processing and analysis, stochastic geometry and numerical simulations. He published more than 70 papers in international journals and conference proceedings. He is a member of the Institute of Electrical and Electronics Engineers (IEEE), the International Association for Pattern Recognition (IAPR), International Society for Stereology and Image Analysis (ISSIA). He has worked for CS-SI (Toulouse, France) as an IT engineer, and for Thalès-Angénieux (Saint-Héand, France) as an image processing expert.

Johan DEBAYLE

received his M.Sc., Ph.D. and Habilitation degrees in the field of image processing and analysis, in 2002, 2005 and 2012 respectively. Currently, he is a Full Professor at the Ecole Nationale Supérieure des Mines de Saint-Etienne (ENSM-SE) in France, within the SPIN Center and the LGF Laboratory, UMR CNRS 5307, where he leads the PMDM Department interested in image analysis of granular media. In 2015, he was a Visiting Researcher for 3 months at the ITWM Fraunhofer / University of Kaisersleutern in Germany. In 2017 and 2019, he was invited as Guest Lecturer at the University Gadjah Mada, Yogyakarta, Indonesia. He was also Invited Professor at the University of Puebla in Mexico in 2018 and 2019. He is the Head of the Master of Science in Mathematical Imaging and Spatial Pattern Analysis (MISPA) at the ENSM-SE.

His research interests include image processing and analysis, pattern recognition and stochastic geometry. He published more than 120 international papers in international journals and conference proceedings and served as Program committee member in several international conferences (IEEE ICIP, MICCAI, ICIAR…). He has been invited to give a keynote talk in several international conferences (SPIE ICMV, IEEE ISIVC, SPIE-IS&T EI, SPIE DCS…) He is Associate Editor for 3 international journals: Pattern Analysis and Applications (Springer), Journal of Electronic Imaging (SPIE) and Image Analysis and Stereology (ISSIA).

He is a member of the International Society for Optics and Photonics (SPIE), International Association for Pattern Recognition (IAPR), International Society for Stereology and Image Analysis (ISSIA) and Senior Member of the Institute of Electrical and Electronics Engineers (IEEE).

ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE SAINT-ÉTIENNE

One of the missions of École des Mines de Saint-Étienne, France, is scientific research at the highest level and contributions to companies' competitiveness. It aims at conveying the economical politics of the country and speeding up the sustainable industrial development by innovation and efficient contributions.

This high level scientific research leads to publications recognized by the international scientific community. Research and teaching are very closely interwoven and the consequence of this is the attractiveness of our Master's Degree courses and our renowned Doctoral School.

MINES
Saint-Étienne

Une école de l'IMT

Liminar considerations

This book is presented as a collection of tutorials. Each chapter presents different objectives, usually with practical applications, in order to develop the image processing and analysis skills by its own. For each tutorial, you will first find in the statement different questions that will guide you to the complete results. Our suggestion is to start by the first tutorials, that acts as a round-up stage. Then, choose the topic that interests and motivates you, and start to work on your code.

The statements does not always contain the necessary theoretical background that you need to answer to the questions. Find resources on the net. Wikipedia is usually a very good start. Go to the university library!

The correction gives you a proposition of solution: it is not unique and it might not be perfect (or it might unfortunately contain errors). This correction is also available in a dedicated website (links will be provided along with the tutorials), we suggest you have a look at it only when you encounter blocking problems or fastidious lines of code to program. Moreover, you can evaluate the processing time of your version of the code and compare it to our proposed version, which usually uses fast and optimal functions, except for readibility purposes.

With Python, you can use the following code:



```
import time

time_start = time.timeit()
# run your code
time_elapsed = (time.timeit() - time_start)
```

CC-By license?

If you manage to code a drastically faster method, if you notice errors or mistakes, if you want to add precisions, remember that this book as well as the code is published under a FREE CC-BY license (as in FREE speech). Contact us and we will be really happy to introduce modifications and insert you in the credits, or more...



The book is available in a high quality printed format, obviously not for free, but the pdf format is free (as in FREE beer). This is due to the fact that we (Johan Debayle and Yann Gavet) are employed by the French government, and we are paid to teach and disseminate informations to students. Our work is thus belongs to the society, and it seems a normal process to give the results back to the society. Feel free to use, modify and distribute these tutorials as you wish. Just remember to keep the appropriate credits (our names and MINES Saint-Etienne). If you want to express your gratitude, send us a postcard at:

```
Yann GAVET or Johan Debayle
MINES Saint-Etienne
CS 62362
42023 SAINT-ETIENNE cedex 2 - FRANCE
```

You can also buy the printed version of the book, which will make our editor really happy.

Images

The images belong to their authors, unless otherwise stated. If by mistake we used images without giving the appropriate credit, please contact us.

Softwares

This book is available for two programming languages, MATLAB® and Python. MATLAB® is a proprietary software dedicated to scientific computing. Functions are grouped into so-called toolboxes. The documentation

and the compatibility of the functions are very good, but the price is a consequence of this. Python is built upon open-source and Free softwares. Scientific modules are numerous, but other general purposes functions can also be found. Documentation and code may be of inequal qualities, but major scientific modules (numpy, scipy, opencv...) are really well presented and optimized.

Portions of code will be highlighted by the use of special boxes, like:



```
% this is matlab code
```



```
# this is python code
```

Time to spend on a tutorial

The tutorials are almost independent. To evaluate the difficulty of the tutorial, stars are present at the header of each one. Depending on your knowledge, you will have to spend between 2 hours and 10 hours per tutorial.

- One star tutorial should be a relatively easy tutorial,
- two stars tutorials introduce some difficulties, either in programming or in the theoretical concepts,
- three stars tutorials are difficult both in theory and in programming.

They are divided into 6 parts, namely:

- Enhancement and Restoration: dedicated to image processing methods employed to eliminate noise and improve vision of data.
- Mathematical Morphology: introduction to basic operations of mathematical morphology.
- Registration and Segmentation: methods to segment, i.e. detect objects in images.
- Stochastic Analysis: method based on random processes.
- Characterization and Pattern Analysis: measures performed on objects in order to perform analysis and recognition.
- Exams: uncorrected problems and questions.

Enjoy!

You will find online corrections on the editor website and at page: http://iptutorials.science. You will also find qrcodes for each correction that points to code and images.

Part I Fundamentals

1 Introduction to image processing

In this tutorial, you will discover the basic functions in order to load, manipulate and display images. The main informations of the images will be retrieved, like size, number of channels, storage class, etc. Afterwards, you will be able to perform your first classic filters.
Do not forget that one of the main objectives is finding by yourself (in the documentation of the different modules) the usage of the appropriate functions.

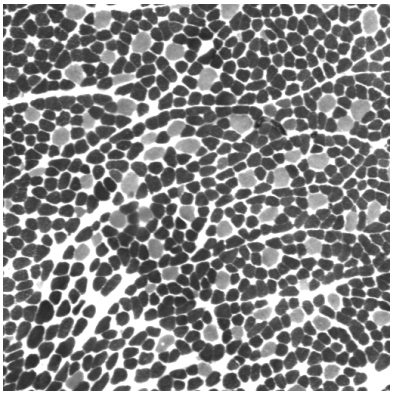
The different processes will be realized on the following images:

Figure 1.1: Image examples.

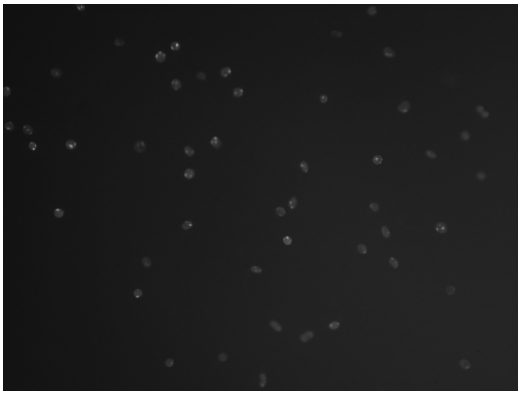
(a) Retinal vessels.



(b) Muscle cells.



(c) Cornea cells (BIIGC, Univ. Jean Monnet, Saint-Etienne, France).



1.1. First manipulations

- Image loading can be made by the use of the python function skimage.io.imread.
- The visualization of the image in the screen is realized by using the module matplotlib.pyplot.
- Also import the numpy module with `import numpy as np`. All images with be manipulated as a numpy array.

?

- Load and visualize the first image as below. Notice the differences.
- Look at the data structure of the image I such as its size, type....
- Visualize the green component of the image. Is it different from the red one? What is the most contrasted color component? Why?
- Enumerate some digital image file formats. What are their main differences? Try to write images with the JPEG file format with different compression ratios (0, 50 and 100), as while as the lossless compression, and compare.

See skimage.io.imsave

1.2. Color quantization

Color quantization is a process that reduces the number of distinct colors used in an image, usually with the intention that the resulting image should be as visually similar as possible to the original image. In principle, a color image is usually quantized with 8 bits (i.e. 256 gray levels) for each color component.

?

- By using the gray level image 'muscle', reduce the number of gray levels to 128, 64, 32, and visualize the different resulting images.
- Compute the different image histograms and compare.

You can take advantage of the floor division in order to get integer value of the division, for example:

```
>>> print (255//4)
63
```

1.3. Image histogram

An image histogram represents the gray level distribution in a digital image. The histogram corresponds to the number of pixels for each gray level. It is equivalent to a probability density function.

With numpy, you can use a the following code to plot the histogram:

```
# I is the 2D grayscale image
h, edges = np.histogram(I, bins=256)
plt.plot(edges[:-1], h)
plt.show()
```

You have to code a python function with the following prototype:

```
def histogram(I):
```

?

- Test the numpy version of the histogram.
- Code your own version of this function and visualize the histogram of the image 'muscle.jpg'.

1.4. Linear mapping of the image intensities

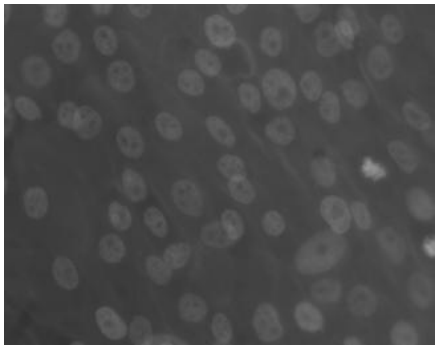
The gray level range of the image 'cellules_cornee.jpg' can be enhanced by a linear mapping such that the minimum (resp. maximum) gray level value of the resulting image is 0 (resp. 255). Mathematically, it consists in finding a function $f(x) = ax + b$ such that $f(min) = 0$ and $f(max) = 255$.

?

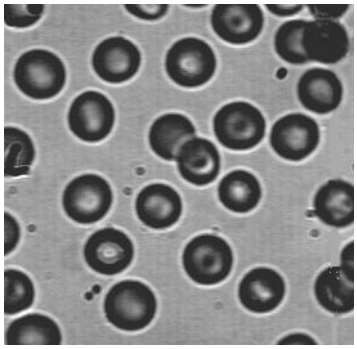
- Load the image and find its extremal gray level values.
- Adjust the intensities by a linear mapping into $[0, 255]$.
- Visualize the resulting image and its histogram.

1.5. Low-pass filtering

The different processes will be realized on the following images:



(a) osteoblast cells



(b) blood cells

Low-pass filtering aims to smooth the fast intensity variations of the image to be processed.

?

Test the low-pass filters 'mean', 'median', 'min', 'max' and 'gaussian' on the noisy image 'blood cells'.

The modules skimage. filters and skimage. filters .rank contain a lot of classical filters.

?

Which filter is suitable for the restoration of this image?

1.6. High-pass filtering

High-pass filtering aims to smooth the low intensity variations of the image to be processed.

?

- Test the high-pass filters HP on the two initial images in the following way: $HP(f) = f - LP(f)$ where LP is a low-pass filtering (see the previous exercise).
- Test the Laplacian (high-pass) filter on the two initial images.

1.7. Convolution algorithm

The general expression of the convolution, denoted by the operator $*$, is given by

$$g(x,y) = h * f(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} h(i,j) f(x-i, y-j)$$

with f the original image, g the filtered image and h the convolution mask (a.k.a. kernel).
The scipy.signal.convolve2d is used for 2D convolution.

?

Apply the convolution operation with the following convolution kernels, that define some classical filters:

- Mean filter

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- Laplacian filter

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & +8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

- Gaussian filter:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

1.8. Derivative filters

Derivative filtering aims to detect the edges (contours) of the image to be processed.

?

- Test the Prewitt and Sobel derivative filters (corresponding to first order derivatives) on the image 'blood cells' with the use of the following convolution masks:

$$\begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

- Look at the results for the different gradient directions.
- Define an operator taking into account the horizontal and vertical directions.

Remark : the edges could be also detected with the zero-crossings of the Laplacian filtering (corresponding to second order derivatives).

1.9. Enhancement filtering

Enhancement filtering aims to enhance the contrast or accentuate some specific image characteristics.

?

- Test the enhancement filter E on the image 'osteoblast cells' defined as: $E(f) = f + HP(f)$ where HP is a Laplacian filter (see also tutorial 7 for enhancement methods.).
- Parameterize the previous filter as: $E(f) = \alpha f + HP(f)$, where $\alpha \in \mathbb{R}$.

1.10. Aliasing effect

The following image is generated via a function g defined by: $g(x, y) = \sin(2\pi f_0 \sqrt{x^2 + y^2})$.

Informations

To generate a 2D realization of this function, the np.meshgrid function is strongly recommended. Its usage is as follows, with f_s being the (spatial) sampling frequency:

```
t = np.arange(0, 1, 1./fs)
xx, yy = np.meshgrid(t, t)
realization = g(xx, yy) # compute 2D values
```

?

- Create an image (as right) that contains rings, defined above. The function takes two input parameters: the sampling frequency f_s and the signal frequency f_0.
- Look at the influence of the two varying frequencies. What do you observe? Explain the phenomenon from a theoretical point of view.



1.11. Open question

Find an image filter for enhancing the gray level range of the image 'osteoblast cells'.



1.12. Python correction





```
# display images
import matplotlib.pyplot as plt
# ndimage defines a few filters
from skimage.filters.rank import mean
import skimage.filters # prewitt, gaussian
# numeric calculation
import numpy as np
# measure time
import time
# read and save images
from skimage.io import imread, imsave
# convolution method
from scipy.signal import convolve2d
# data
import skimage.data as data
```

1.12.1. First manipulations

Open, write images

The following file loads the camera image and display it. The `print` function is optional.



```
# load file camera
camera = data.camera()
# load file cerveau.jpg
brain = imread('cerveau.jpg')
print(type(brain))
print(brain.shape, brain.dtype)
# save file
imsave('test.png', brain)
```

Display images

You can modify to previous example to add the following lines:

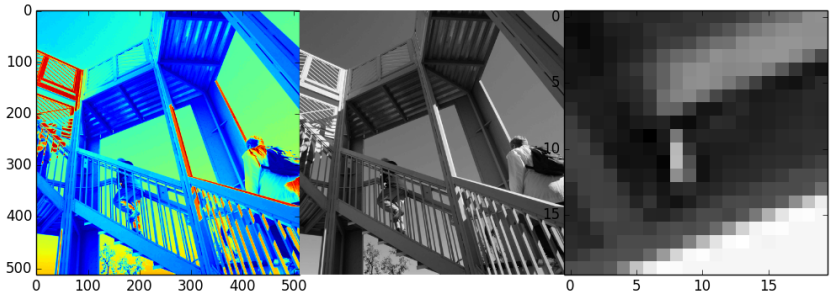


```
plt.imshow(camera)
plt.show()
```

Notice that you have to close the image window to write commands again. Also, the colormap is not the good one by default (see Fig. 1.2).

```
plt.figure(figsize=(10, 3.6))
# first subplot
plt.subplot(131)
plt.imshow(camera)
# second subplot
plt.subplot(132)
plt.imshow(camera, cmap=plt.cm.gray)
plt.axis('off')
# third subplot (zoom)
plt.subplot(133)
plt.imshow(camera[200:220, 200:220], cmap=plt.cm.gray, interpolation='nearest')
plt.subplots_adjust(wspace=0, hspace=0.,
                    top=0.99, bottom=0.01,
                    left=0.05, right=0.99)
plt.show()
```

Figure 1.2: Displaying images with an adapted colormap.



Color channels

A color image is constituted of (generally) three channels. This representation follows the human visual perception principles: in the human retina, the sensitive cells (the cones) react to specific wavelength that correspond to red, green and blue. The sensors technology adopted the *same* characteristics and a so-called Bayer filter has 2 green filters for 1 red and 1 blue. Consequently, the green channel presents a better resolution than the other channels.

Figure 1.3: The green channel presents the best contrasts in the case of retina images.

(a) Color image. (b) Red channel. (c) Green channel (d) Blue channel.

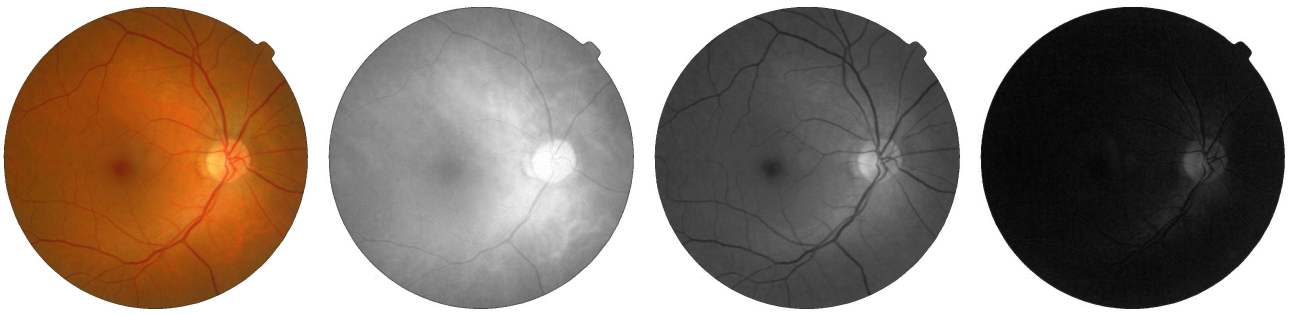
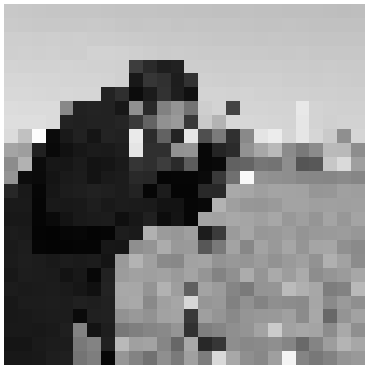


Image Resizing

The number of pixels is reduced by subsampling the image. Notice that there is no anti-aliasing filter applied to the image before reducing its size. The result is presented in Fig.1.4 with images of the same size, and in Fig.1.5 with images at the same resolution.

Figure 1.4: Reduction of the scale of the image on each axis, showing a so-called *pixellisation* effect. Represented at the same size, the density of pixels is thus reduced.

(a) Scale divided by 20.



(b) Scale divided by 10.



(c) Scale divided by 5.



Figure 1.5: At a constant resolution, images with different definitions are represented with different sizes.

(a)



(b)



(c)



Color quantization

The following code uses the properties of integer operations to round values to the nearest integer. Illustration is presented Fig. 1.6.

```
# Integer division operator //
q4 = camera // 4*4;
q16= camera //16*16;
q32= camera //32*32;
```

Figure 1.6: Reduction of the number of gray levels (quantization).

(a) q4.



(b) q16.



(c) q32.



JPEG file format

In order to test the effect of jpeg compression, one can use the parameter `quality`. For the lowest quality, the loss of informations is really important (see Fig.1.7).

```
# test jpeg quality
imsave("a_25.python.jpeg", camera, quality=25);
imsave("a_100.python.jpeg", camera, quality=100);
imsave("a_50.python.jpeg", camera, quality=50);
imsave("a_75.python.jpeg", camera, quality=75);
imsave("a_1.python.jpeg", camera, quality=1);
```

Figure 1.7: Different quality used to compress jpeg files.

(a) Quality 1.



(b) Quality 25.



(c) Quality 75.

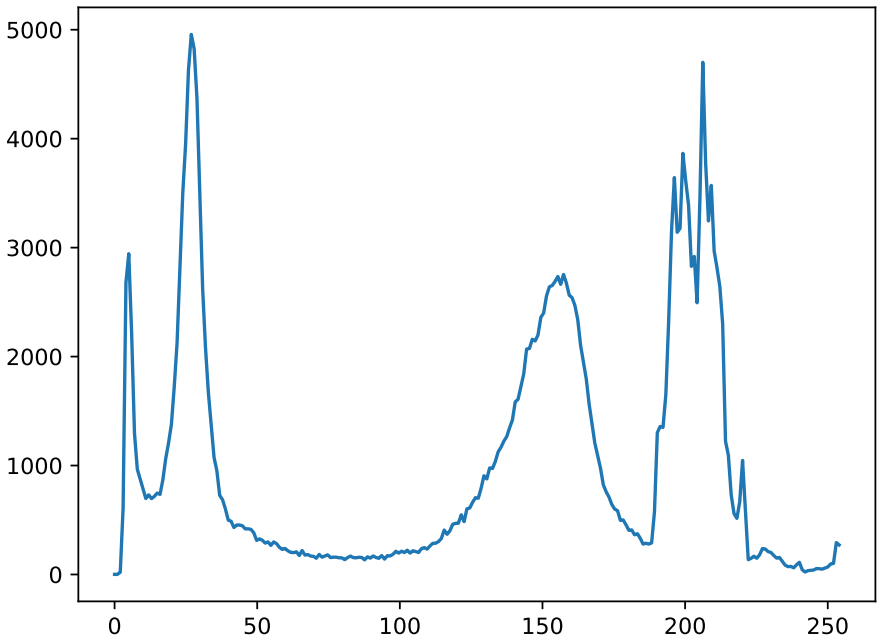


1.12.2. Histogram

To compute the histogram of an image, we recommand the numpy function `histogram` (see result in Fig. 1.8).

```
h, edges = np.histogram(camera, bins=256)
plt.plot(edges[:-1], h)
```

Figure 1.8: Histogram of camera image.



You can also write your own code. The execution time is lower for the numpy method, which is vectorized and optimized.



```
# Histogram function with 2D image
def compute_histogram(image):
    tab = np.zeros((256, ), dtype='I')
    X, Y = image.shape
    for i in range(X):
        for j in range(Y):
            tab[image[i,j]]+=1

    return tab
```



```
# Histogram function with flatten image (vector)
def compute_histogram2(image):
    im = image.flatten()
    tab = np.zeros((256, ), dtype='I')
    for i in im:
        tab[i]+=1
    return tab
```

The following code presents a comparison of the different method for histogram computation.



```python
# load camera image and compute histograms
camera = data.camera()
t0 = time.time()
h = compute_histogram(camera)
t1 = time.time()
h2 = compute_histogram2(camera)
t2 = time.time()

#  ....  plots
print(f"execution time 2D: {t1-t0:.2} s")
plt.subplot(131)
plt.plot(h)
plt.title('2D function')

print(f"execution time 1D: {t2-t1:.2} s")
plt.subplot(132)
plt.plot(h2)
plt.title('1D function')

# last plot: with numpy function
plt.subplot(133)
t3 = time.time()
h, edges = np.histogram(camera, bins=256)
plt.plot(edges[:-1], h)
t4 = time.time()
print(f"execution time numpy: {t4-t3:.2} s")

# display
plt.show()
```

The console outputs the following computation durations:



```
execution time 2D: 0.44 s
execution time 1D: 0.4 s
execution time numpy: 0.0025 s
```

1.12.3. Linear mapping of the image intensities

The linear mapping is a simple method that stretches linearly the histogram. If displayed with matplotlib, the images is linearly stretched, thus the modification cannot be observed.



```python
def image_stretch(image):
    # returns image with new maximum and minimum at 255 and 0
    I = I - np.min(I)
    I = 255 * I / np.max(I)
    return I.astype('int')
```

1.12.4. Low-pass filtering



The module scipy.ndimage.filters contains the usual filter functions.

The mean filter is illustrated in Fig. 1.9.



```
# mean on a 3x3 neighborhood
m3 = mean(camera, np.ones((3, 3))
m25= mean(camera, np.ones((25, 25))

plt.subplot(121)
plt.imshow(m3, cmap=plt.cm.gray)
plt.axis('off')
plt. title ('3x3 mean filter')

plt.subplot(122)
plt.imshow(m25, cmap=plt.cm.gray)
plt.axis('off')
plt. title ('25x25 mean filter')

plt.show()
```

Figure 1.9: Mean filters.

(a) Neighborhood of size 3x3.



(b) Neighborhood of size 25x25.



Gaussian filter

The gaussian filter is presented in Fig. 1.10.



```
# camera image
camera = data.camera()

# Gaussian filter
gaussian = skimage.filters.gaussian(camera, 5)
```

Figure 1.10: Gaussian filter of size 5.



1.12.5. High-pass filter

The computation of the high-pass filter is simply the subtraction of a low-pass filter from the original image. For example:

```
H = I-m25
```

1.12.6. Convolution algorithm

A very simple 2D convolution function, that does not handle sides, can be coded as:

```
def myconv2d(I, h):
    X, Y = I.shape
    N, M = h.shape
    G = I.copy()

    # loop on all pixels that can include the kernel
    # (this avoids side effects)
    for i in np.arange(0, X-N):
        for j in np.arange(0, Y-M):

            w = I[i:i+N, j:j+M]
            g = np.sum(w*h)
            G[i+N//2, j+M//2] = g

    return G
```

1.12.7. Derivative filters

Derivative filters (Prewitt, Sobel...) use a finite derivation approximation. They are very sensitive to noise (as every system using a derivation). The gradient is defined as a vector, and a norm should be used to display a resulting image. Notice that with these filters, the connexity of the contours is not preserved. Illustration is proposed in Fig. 1.12.



Figure 1.11: Illustration of a simple convolution filter, a mean filter of size 25×25. One can see the sides of the image have not been filtered.



```python
# camera image
camera = data.camera()
camera.astype('int32');

# Prewitt filter
prewitt0 = skimage.filters.prewitt(camera, axis=0)
prewitt1 = skimage.filters.prewitt(camera, axis=1)

# Sobel filter
dy = skimage.filters.sobel(camera, axis=0) # vertical
dx = skimage.filters.sobel(camera, axis=1) # horizontal
mag = np.hypot(dx, dy) # magnitude
sobel = mag * 255.0 / mag.max() # normalize (Q&D)

# display results
plt.subplot(131)
plt.imshow(prewitt0, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Prewitt filter axis 0')

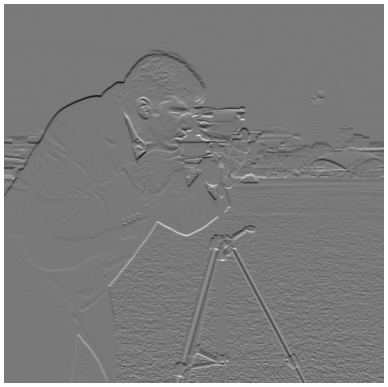
plt.subplot(132)
plt.imshow(prewitt1, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Prewitt filter axis 1')

plt.subplot(133)
plt.imshow(sobel, cmap=plt.cm.gray)
plt.axis('off')
plt.title('Sobel filter')

plt.show()
```

Figure 1.12: Prewitt filter.

(a) Prewitt filter for axis x.



(b) Prewitt filter for axis y.



The previous code uses filters proposed by the skimage module. If you want to write down the convolution matrix, this is the solution:

```
h = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
grad1 = convolve2d(camera, h);
plt.imshow(grad1, "gray")
plt.show()

grad0 = convolve2d(camera, h.transpose());
plt.imshow(grad0, "gray")
plt.show()

plt.imshow(np.sqrt(grad0**2 + grad1**2), "gray")
plt.show()
```

1.12.8. Enhancement filter

This functions adds the Laplacian filter to the original image.

```
def sharpen(I, alpha):

    h = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    L = convolve2d(I, h, mode='same')
    np.max(L)
    E = alpha * I + L
    E = skimage.exposure.rescale_intensity(E, out_range=(0,255))
    E = E.astype(np.uint8)

    return E
```

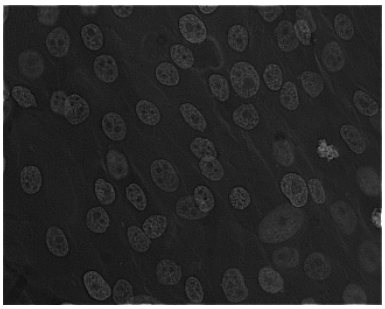
This constitutes a really simple and efficient edge sharpening method. To test it, you may write this example code:

```
I = imread('osteoblaste.png').astype(np.float)
I = I/255
A = [1, 0.5, 5]

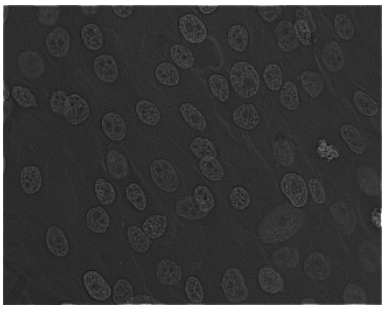
fig = plt.figure(figsize=(12,8))
plt.subplot(221)
plt.imshow(I, cmap=plt.cm.gray)
for i,alpha in enumerate(A):
    plt.subplot(222+i)
    E = sharpen(I, alpha)
    E = skimage.exposure.rescale_intensity(E, out_range=(0,255))
    plt.imshow(E, cmap=plt.cm.gray)
    plt.title('alpha='+str(alpha))

    imsave('osteoblaste_rehauss_'+str(alpha)+'.python.png', E)

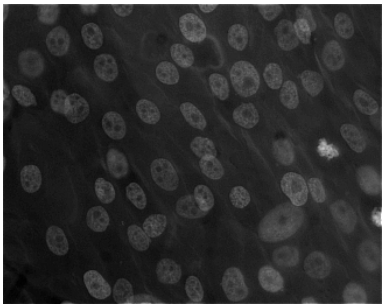
plt.show()
```



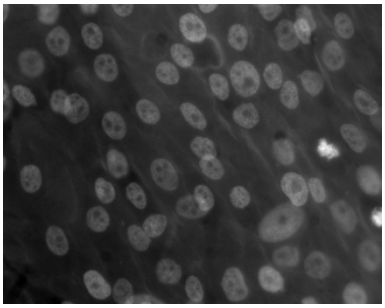
(c) $\alpha = 1$.



(d) $\alpha = 0.5$.



(e) $\alpha = 5$.



(f) Original image.

Figure 1.13: Image enhancement: $I = \alpha \cdot I + HP(I)$, where HP is a high-pass filter (the Laplacian filter in these illustrations).

1.12.9. Aliasing effect

```
# aliasing effect (Moire)
def circle(fs, f):
    # Generates an image with aliasing effect
    # fs: sample frequency
    # f : signal frequency
    t = np.arange(0,1,1./fs);
    ti, tj = np.meshgrid(t,t);
    C = np.sin(2*np.pi*f*np.sqrt(ti**2+tj**2));
    return C
```

The image of Fig. 1.14 is generated with the following code.

```
C = circle(300,50);
plt.imshow(C, cmap=plt.cm.gray);
plt.show()
imsave('moire.png', C);
```

Figure 1.14: Moiré effect, generated with $f_s = 300$ and $f = 50$.



Part II Image Transforms and Representations

2 2D Fourier Transform

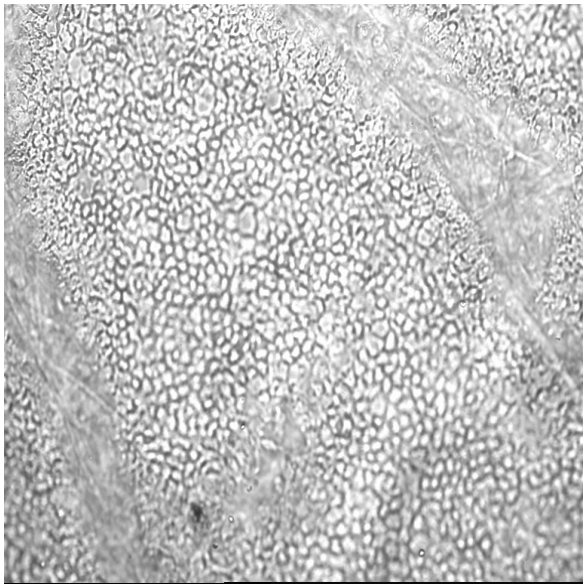
The main objective of this tutorial is to study image filters applied in the frequencial or spatial domain with the Fourier transform.

Use module np. fft for Fourier Transform functions (function fft2), angle and abs for phase and amplitude, fftshift for changing the representation).

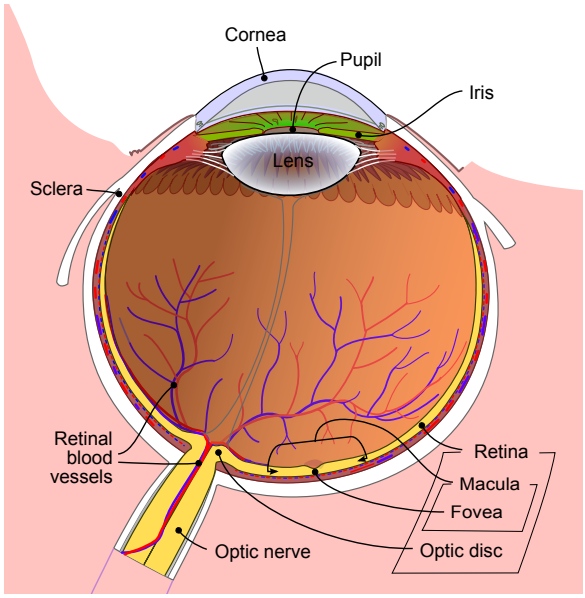
The image to be used comes from a human cornea endothelium observed ex vivo by optical microscopy.

Figure 2.1: Image of human cornea endothelium observed by optical microscopy. The ophthalmologists would like to know the cell density, without manually counting all the cells.

(a) Human corneal endothelium, extracted from a donor and observed here before grafting.



(b) Human eye (from Wikipedia, authors: Rhcastilhos and Jmarchn, CC-By-SA).



2.1. Fourier transform

?

1. Load an image and visualise it.
2. Compute the Fourier Transform by the fft algorithm.
3. Visualise the images of the phase and amplitude of the Fourier Transform.

2.2. Inverse Fourier transform

In this exercise, it can be interesting to consider different images, like the cameraman picture for example.

```python
from skimage.data import camera
I = camera()
```

?

1. Apply the inverse Fourier transform on the Fourier transform to find the original image.
2. Now, apply the inverse Fourier transform on the phase information only (without using the amplitude informations).
3. In a same spirit, apply the inverse Fourier transform on the amplitude informations only (without the phase).

2.3. Low-pass and high-pass filtering

The objective is to apply an ideal filter, that "perfectly" cuts the frequencies. You have to code this filter H in the Fourier domain so that $G = F \times H$ is the Fourier transform of the filtered image g.

?

1. Modify the Fourier transform of the image to
   - keep only low frequencies,
   - keep only high frequencies.
2. Apply the inverse Fourier transform on both and comment.

2.4. Application: evaluation of cellular density

The ophthalmologists would like to evaluate the cell density of the cornea endothelium observed in Fig. 2.1.

?

1. Compute the Fourier transform of the image. If one considers that the cells constitute a repeated pattern on the whole image, locate the repetition frequency on the amplitude image.
2. Can this frequency be linked to the cell density ?

The answer to this last question is obviously yes. Here follows a simple method to evaluate the cell density (or the mean cell radius), if these cells a considered as circular [38].

?

1. The amplitude information is very noisy. First of all, a gaussian filter should be applied.
2. Find a simple way to evaluate the mean radius of the cells.



Informations

See skimage. filters .gaussian.

2.5. Python correction

```
import numpy as np
import matplotlib.pyplot as plt # plots
```

2.5.1. Introduction and utilities

To display a spectrum, the following function can be useful:

```
def viewSpectrumPhase(amplitude, phase):
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.imshow(np.log(1+amplitude), plt.cm.gray)

    plt.subplot(1, 2, 2)
    mmax = np.max(phase)
    mmin = np.min(phase)
    if (mmax == mmin):
        B = 0
    else:
        B = 255*(phase-mmin)/(mmax-mmin)

    plt.imshow(B, plt.cm.gray)
    plt.show()
```

2.5.2. Fourier transform

Results (amplitude and phase) are represented in Figs.2.2 and 2.5.

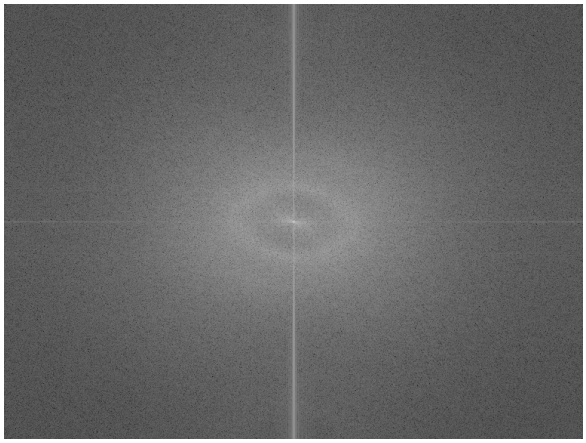
```
cornea = skimage.io.imread('cornee.png')
print type(cornea)
print cornea.shape, cornea.dtype

plt.subplot(131)
plt.imshow(cornea, 'gray')
plt.show()
```

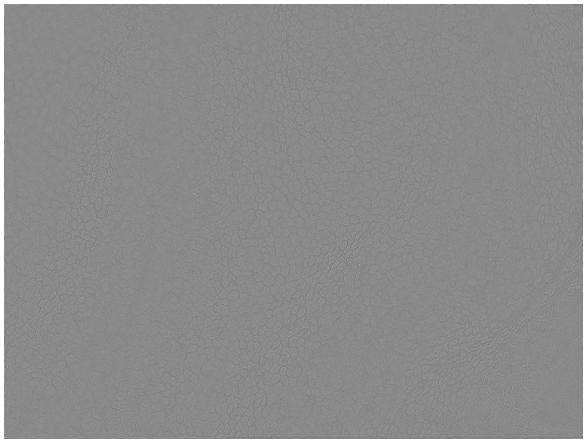
```
## Fourier transform
#  result is complex
#  fftshift is used by convention the get frequency 0 at center of image
spectre = np.fft.fftshift(np.fft.fft2(cornea));

A = abs(spectre);
G = np.angle(spectre);

viewSpectrumPhase(A, G);
```



(a) Logarithm of the amplitude: $\log(1 + A)$.



(b) Phase.

Figure 2.2: Amplitude and phase of the Fourier transform of the cornea. Remember that the information lays in the phase (see reconstruction by phase).

2.5.3. Inverse Fourier Transform

```
## inverse Fourier transform
cornee2 = np.real(np.fft.ifft2(np.fft.fftshift(spectre)))
plt.imshow(cornee2, 'gray')
plt.title('inverse FT')
plt.show()
```

The Fourier transform, although very powerful, is really difficult to interpret in 2D. The main information is not contained in the amplitude, but in the phase. The following reconstructions will illustrate it (see Fig.2.3).

```
## inverse Fourier transform, without the phase
cornee_amplitude = np.real(np.fft.ifft2(np.fft.fftshift(A)))

plt.imshow(cornee_amplitude, 'gray')
plt.title('inverse FT on amplitude')
plt.show()
```

```
## inverse Fourier transform on phase only
complex_phase = np.exp(1j*G)
cornee_phase=np.real(np.fft.ifft2(np.fft.fftshift(complex_phase)))

plt.imshow(cornee_phase, 'gray')
plt.title('inverse FT on phase')
plt.show()
```

2.5.4. Low-pass and high-pass filtering

The results are illustrated in Fig. 2.4. Two functions are defined.

A low pass-filter consists in the suppression of values for frequencies lower than a cut-off frequency.

Figure 2.3: Reconstruction of partial informations (phase or amplitude only).

(a) Reconstruction with amplitude only.



(b) Reconstruction with phase only.

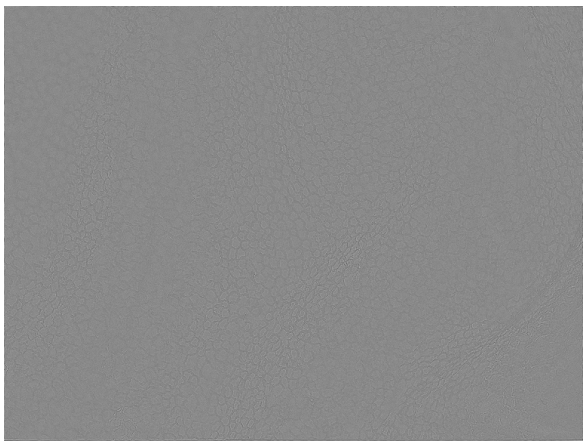
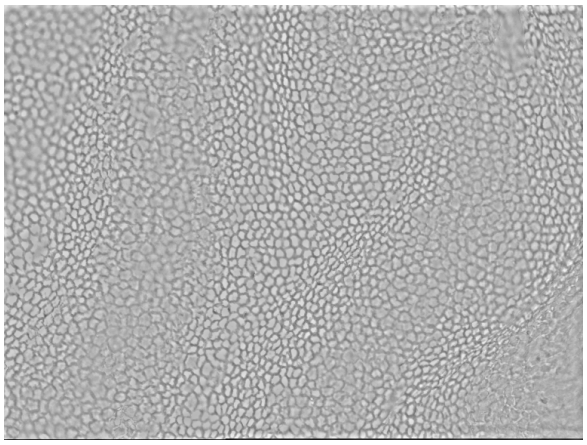
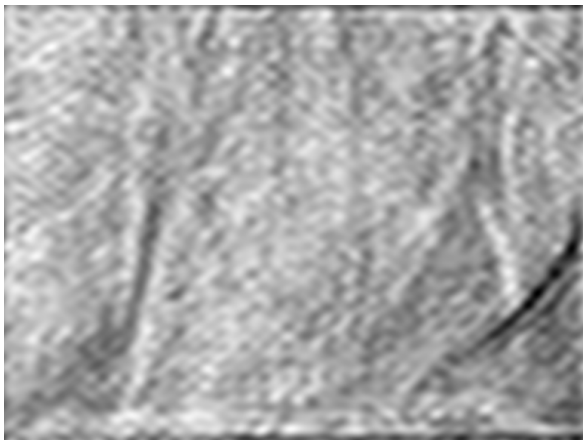


Figure 2.4: Fourier basic filtering of cornea image.

(a) High Pass filter.



(b) Low Pass filter.



```python
def LowPassFilter(spectrum, cut):
    """Low pass filter of the FFT (spectrum)
    The shape of this filter is a square. fftshift has been applied so that
    frequency 0 lays at center of spectrum image
    @param spectrum: FFT2 transform
    @param cut     : cut value of filter (no physical unit, only number of pixels)
    """
    X,Y = spectrum.shape
    mask = np.zeros((X,Y), "int")
    mx = X/2; my = Y/2
    mask[mx-cut:mx+cut, my-cut:my+cut] = 1
    f = spectrum * mask
    plt.figure
    plt.imshow(np.log(1+abs(f)))
    plt.title('Low pass filter')
    return f
```

A high pass filter is exactly the opposite: suppression of the values under the cut-off frequency.



```
def HighPassFilter(spectrum, cut):
    """High pass filter of the FFT (spectrum)
    The shape of this filter is a square. fftshift has been applied so that
    frequency 0 lays at center of spectrum image
    @param spectrum: FFT2 transform
    @param cut     : cut value of filter (no physical unit, only number of pixels)
    """
    X,Y = spectrum.shape;
    mask = np.ones((X,Y), "int")
    mx = X/2; my = Y/2
    mask[mx-cut:mx+cut, my-cut:my+cut] = 0
    f = spectrum * mask

    plt.imshow(np.log(1+abs(f)))
    plt.title('High pass filter')
    return f
```

In the following application, the image is loaded and the effects of a low-pass and high-pass filters are illustrated.



```
# FT of original image
cornea = imageio.imread('cornee.png')
spectre = np.fft.fftshift(np.fft.fft2(cornea))
# low pass filter
L = LowPassFilter(spectre, 30)
viewSpectrumPhase(abs(L), np.angle(L))
corneaLP = np.real(np.fft.ifft2(np.fft.fftshift(L)))
# high pass filter
H = HighPassFilter(spectre, 30)
viewSpectrumPhase(abs(H), np.angle(H))
corneaHP = np.real(np.fft.ifft2(np.fft.fftshift(H)))

# display results and filters
plt.subplot(1,2,1)
plt.imshow(corneaLP, plt.cm.gray); plt.title('reconstruction after LP filtering')
plt.subplot(1,2,2)
plt.imshow(corneaHP, plt.cm.gray); plt.title('reconstruction after HP filtering')
plt.show()
```

2.5.5. Application: evaluation of cellular density

The cells can be considered as a pattern, and its frequency repetition can found in the Fourier transform. More precisions on this application can be found in [34, 14, 35, 38] on the relation between the real cellular density and the image pixels.

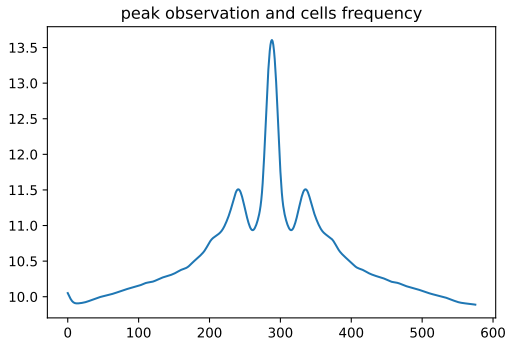
```
cornea = imageio.imread('cornee.tif')
# Fourier Transform
spectre = np.fft.fftshift(np.fft.fft2(cornea))
amplitude = abs(spectre)
# Filter amplitude
Blurred = skimage.filters.gaussian_filter(amplitude, 5)

plt.subplot(1,2,1)
plt.imshow(np.log(1+Blurred), 'gray')
plt.title('filtered amplitude')
# Observe frequency peaks
plt.subplot(1,2,2)
plt.plot(np.log(1+Blurred[:, Y/2]))
plt.title('peak observation and cells frequency')
plt.show()
```

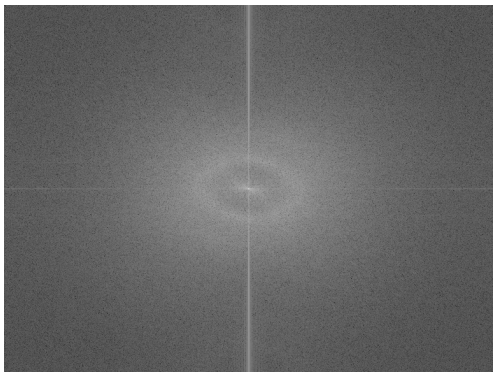
The results are illustrated in Fig. 2.5.

Figure 2.5: The two peaks represent the frequency of repetition of the cellular pattern, i.e. it can be used to compute the cellular density.

(a) Amplitude of Fourier Transform.



(b) Filtering of amplitude.



3 DCT: Discrete Cosine Transform

This tutorial introduces the discrete cosine transform (DCT), which is the core of the JPEG compression standard. The transform is first studied on a single small block, in order to observe the coefficients one by one, then it is applied to the whole image, block by block, together with the quantization step that actually produces the loss of information.

The different processes will be applied on the image of the cameraman, and on a small block extracted from it (Fig. 3.1).

Figure 3.1: The test image and the 16×16 block obtained by resizing it (displayed here with an enlargement of the pixels).

(a) Cameraman.



(b) 16×16 block.



3.1. Definition

The discrete cosine transform F of an image f of size $M \times N$ is an image of the same size. It is similar to the Fourier transform, but the complex exponentials are replaced by cosines, so that the result is **real**:

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f(m,n) \cos\left[\frac{(2m+1)u\pi}{2N}\right] \cos\left[\frac{(2n+1)v\pi}{2N}\right] \tag{3.1}$$

with

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0, \\ 1 & \text{if } u > 0. \end{cases}$$

As the transform is real, there is no need to distinguish a modulus and a phase as with the Fourier transform. The coefficient $F(0,0)$ is the continuous component (the mean value of the block); the low frequencies are located in the upper left corner, and the high frequencies in the lower right corner.

The coefficients have a very large dynamic range: the continuous component is several orders of magnitude larger than the others. The logarithm of the DCT is therefore usually displayed, otherwise nothing is visible.

Natural images (photographs, illustrations) have a frequency content that is mostly concentrated in the low frequencies. This is exactly the property exploited by the JPEG compression, which strongly compresses the high frequencies of the DCT and barely modifies the low frequencies.



The functions `dctn` and `idctn` of the module `scipy.fftpack` compute the n-dimensional direct and inverse transforms. The option `norm='ortho'` makes the transform orthonormal, so that `idctn(dctn(f))` returns f exactly.

3.2. DCT of a single block

In the JPEG standard, the image is processed by blocks of 8×8 pixels. In order to visualize the coefficients more comfortably, we start with a block of 16×16 pixels, obtained by resizing the whole 'cameraman' image.

?

1. Load the image of the cameraman and convert it into floating point values.
2. Resize it to a block of 16×16 pixels and display it: the cameraman should still be recognizable.
3. Compute and display the DCT of this block. Where is the continuous component located? Check that the inverse transform gives the initial block back.

Informations

The image is available in skimage.data.camera(), and the function resize comes from skimage.transform. The option anti_aliasing =False is used here to keep the raw subsampling.

```
I = skimage.data.camera().astype(np.float64)
n = 16
bloc = resize(I, (n, n), anti_aliasing=False)
```

3.3. Quantization

The transform alone does not compress anything: it is a bijection, and the coefficients are floating point values. The loss appears during the **quantization**, where each coefficient is divided by a weight and then rounded to an integer. The higher the weight, the more information is destroyed.

The quantization matrix $Q = (Q_{ij})$ used in this tutorial is given by the following rule, where K is a loss parameter:

$$Q_{ij} = 1 + K \cdot (1 + i + j) \tag{3.2}$$

The weights increase with $i + j$, that is to say with the frequency: the high frequencies are therefore quantized much more coarsely than the low ones.

?

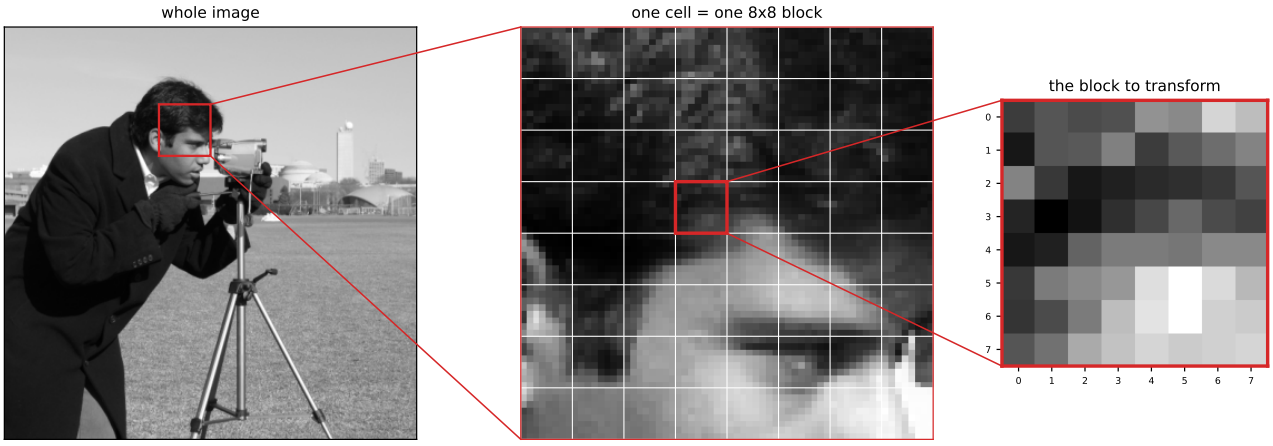
1. Write a function that builds the quantization matrix of size $n \times n$ for a given value of K.
2. Quantize the DCT of the block: divide it by Q and convert the result into integers. Count the number of zero coefficients, and comment.
3. Reconstruct the block: multiply by Q, then apply the inverse DCT. Compare the result with the initial block, and explain where the differences come from.

The function numpy.meshgrid generates the two arrays of the indices i and j, which makes it possible to build Q without any loop. The conversion into integers is done with astype(np.int32): it is this rounding, and only it, that destroys the information.

3.4. The whole image

We now process the entire image, with blocks of 8×8 pixels as in the JPEG standard (Fig. 3.2). The image is supposed to have dimensions that are multiples of 8, which avoids dealing with the incomplete blocks of the borders.

Figure 3.2: Decomposition of the image into blocks of 8×8 pixels. Each block is transformed and quantized independently of the others.



?

1. Write a function `quantization(I, K)` that loops over all the blocks of size 8×8 of the image, and returns the array of the quantized DCT coefficients.
2. Write the reciprocal function `reconstruction(C, K)` that rebuilds the image from these coefficients.
3. Display the array of the coefficients. Why is it hardly interpretable?
4. Display the reconstructed images for several values of K. At which point do the blocks become visible?
5. Plot the reconstruction error (for example the RMSE) and the proportion of zero coefficients as functions of K. Comment on the compromise between the compression rate and the quality.

Informations

The blocks are extracted by slicing the array: the block of indices (i, j) is `I[8*i:8*(i+1), 8*j:8*(j+1)]`.

The real JPEG standard uses a quantization matrix that is tabulated (and stored in the file), not computed by a formula, and it completes the chain with a zig-zag reordering of the coefficients followed by an entropy coding, which is the step that actually reduces the size of the file. The long sequences of zeros produced by the quantization are precisely what this coding compresses so efficiently.

3.5. Python correction

This correction uses scipy.fftpack for the transform itself, and skimage for the image and its resizing.

```
import numpy as np
import matplotlib.pyplot as plt
import skimage
from skimage.transform import resize
from scipy.fftpack import dctn, idctn
```

3.5.1. DCT of a single block

The image is converted into floating point values before any computation, and reduced to a block of 16×16 pixels. The cameraman is still recognizable (Fig. 3.3).

```
I = skimage.data.camera().astype(np.float64)
n = 16
bloc = resize(I, (n, n), anti_aliasing=False)
```

The option norm='ortho' makes the transform orthonormal: the energy is preserved, and above all the reconstruction is exact, which is verified here up to the numerical precision.

```
D = dctn(bloc, norm='ortho')
print("continuous component: {:.1f}".format(D[0, 0]))
print("reconstruction error: {:.2e}".format(
    np.abs(idctn(D, norm='ortho') - bloc).max()))
```

```
continuous component: 2066.8
reconstruction error: 8.53e-14
```

The continuous component is located in $(0, 0)$ and is much larger than all the other coefficients. The logarithm is therefore displayed instead of the raw values (Fig. 3.3).

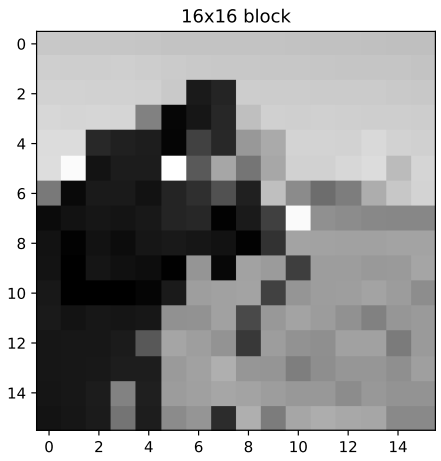
```
plt.imshow(np.log(1 + np.abs(D)), cmap='gray')
```

3.5.2. Quantization of the block

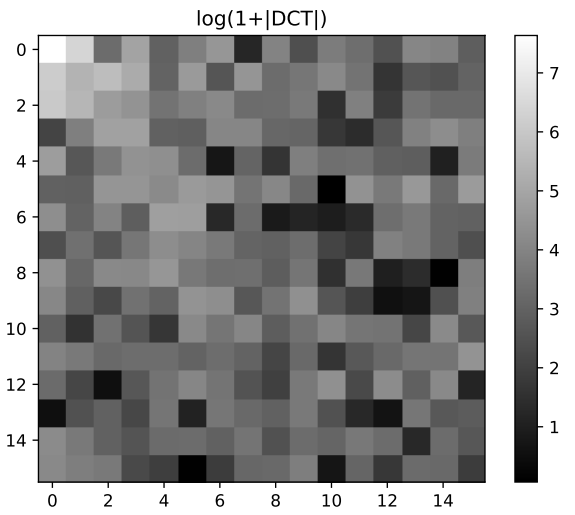
The quantization matrix is built without any loop, thanks to numpy.meshgrid.

Figure 3.3: The 16×16 block and its discrete cosine transform.

(a) Block.



(b) $\log(1 + |DCT|)$: the energy is concentrated in the upper left corner.




```python
def quantization_matrix(n, K=0):
    """
    Square matrix of the quantization coefficients.

    n: size of the (square) matrix
    K: loss parameter. K=0 means no quantization at all.

    returns: numpy array of size (n, n)
    """
    i, j = np.meshgrid(range(n), range(n))
    Q = 1 + K*(1 + i + j)
    return Q
```

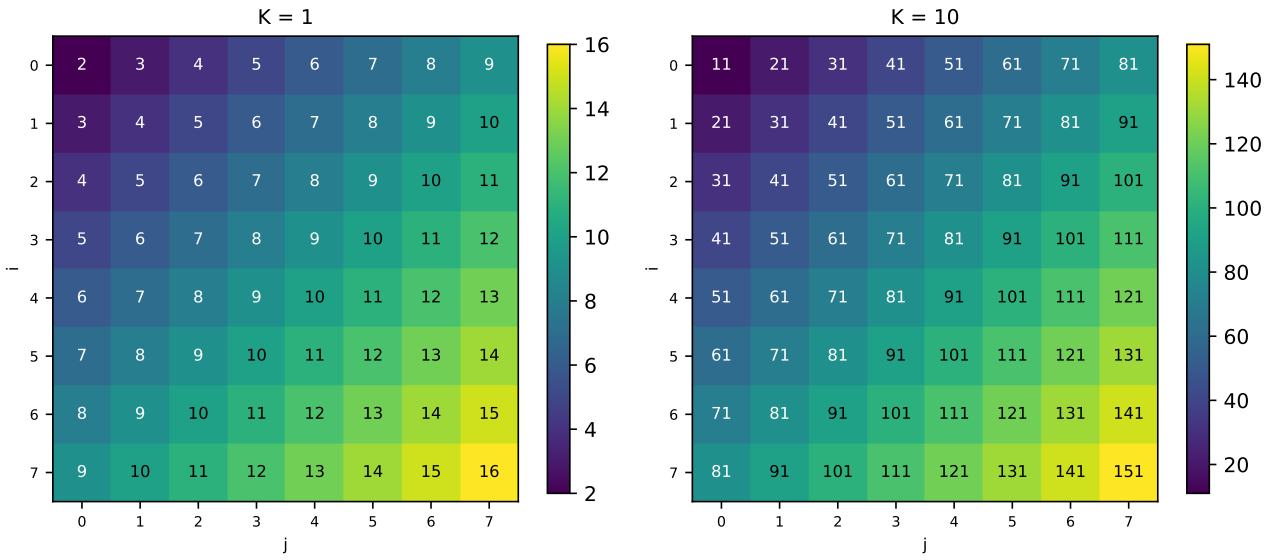
The weights are constant along the anti-diagonals $i + j = \text{cst}$, and they grow linearly with K (Fig. 3.4): the coefficients of the upper left corner, *i.e.* the low frequencies, are almost preserved, while those of the lower right corner are divided by a large number and therefore collapse to zero.

The DCT is then divided by this matrix and converted into integers. **It is this rounding, and only it, that destroys the information**: everything else in the chain is reversible.

```python
K = 2
Q = quantization_matrix(n, K=K)
D2 = (D/Q).astype(np.int32)
print("number of zeros: {0} / {1}".format(np.sum(D2 == 0), n*n))
```

```
number of zeros: 154 / 256
```

Figure 3.4: Quantization matrices of size 8×8 for two values of the loss parameter. The weight of a coefficient only depends on $i + j$, that is to say on its frequency.

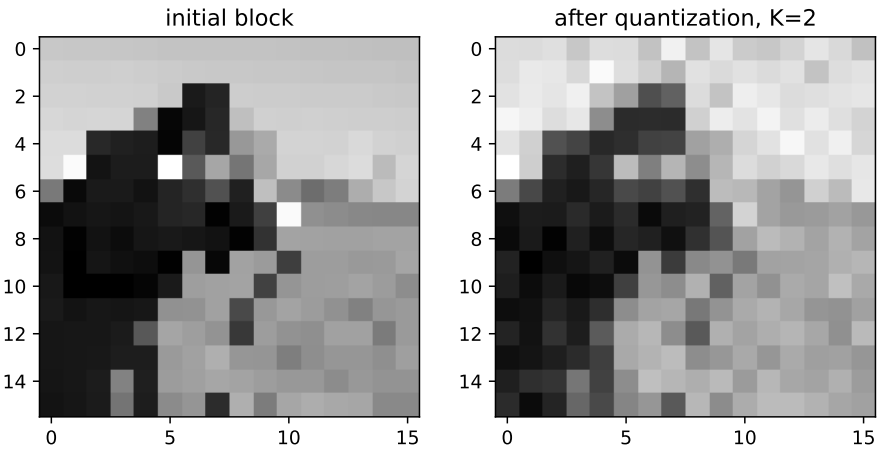


More than half of the coefficients are now equal to zero, and they are all grouped in the lower right corner, *i.e.* in the high frequencies. This is what makes the compression efficient: a long sequence of zeros costs almost nothing to encode.

The block is reconstructed by the reciprocal operations, and the loss is clearly visible on the finest details (Fig. 3.5).

```
bloc_jpeg = idctn(D2*Q, norm='ortho')
```

Figure 3.5: Effect of the quantization on the block, with $K = 2$.



3.5.3. The whole image, block by block

The image is traversed by blocks of 8×8 pixels, and the previous operation is applied to each of them. The quantization matrix is computed only once.



```
def quantization(I, K=0):
    """
    Block DCT and quantization of the whole image.
    The size of the image is supposed to be a multiple of 8.

    I: grayscale image, float
    K: loss parameter of the quantization matrix

    returns: array of the same size as I, containing the quantized
             DCT coefficients of each 8x8 block, as 32 bits integers
    """
    Q = quantization_matrix(8, K=K)

    C = np.zeros_like(I, dtype=np.int32)
    for i in range(I.shape[0]//8):
        for j in range(I.shape[1]//8):
            bloc = I[8*i:8*(i+1), 8*j:8*(j+1)]
            D = dctn(bloc, norm='ortho')
            C[8*i:8*(i+1), 8*j:8*(j+1)] = (D/Q).astype(np.int32)
    return C
```

The reconstruction follows exactly the same structure, with the reciprocal operations.



```
def reconstruction(C, K=0):
    """
    Inverse operation: multiplication by the quantization matrix,
    then inverse DCT, block by block.

    C: quantized DCT coefficients (integers)
    K: loss parameter used for the quantization

    returns: reconstructed image, float
    """
    Q = quantization_matrix(8, K=K)

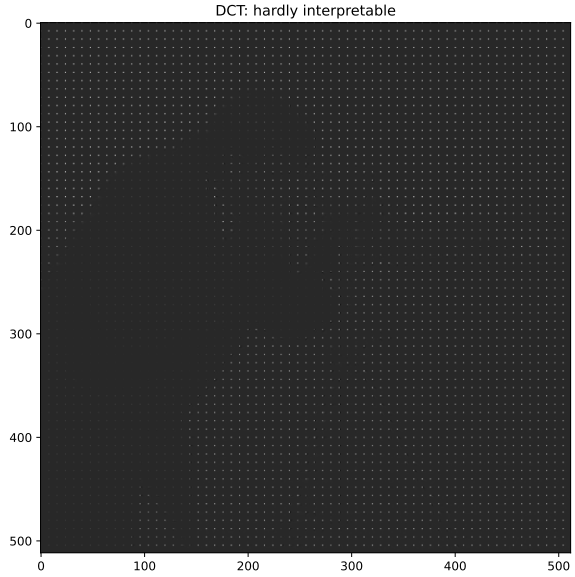
    D = np.zeros(C.shape)
    for i in range(C.shape[0]//8):
        for j in range(C.shape[1]//8):
            bloc = C[8*i:8*(i+1), 8*j:8*(j+1)]
            D[8*i:8*(i+1), 8*j:8*(j+1)] = idctn(bloc*Q, norm='ortho')
    return D
```

Displaying the coefficients gives an almost black image (Fig. 3.6): in each block, only a few values in the upper left corner are non zero, and the continuous component is so large compared to the others that the grayscale is entirely used by it. This display is of little practical interest, but it shows the structure in 8×8 blocks very well.



```
C = quantization(I, K=3)
plt.imshow(C, cmap='gray')
```

Figure 3.6: Quantized DCT coefficients of the whole image, $K = 3$. Only the upper left corner of each 8×8 block carries information.



3.5.4. Influence of the loss parameter

The reconstructions are compared for increasing values of K (Fig. 3.7). For $K = 1$ the image is visually identical to the original one; from $K = 10$ the grass and the sky lose their texture, and for $K = 50$ the 8×8 blocks become clearly visible: this is the well known blocking artifact of a JPEG image compressed too much.

```
for K in [1, 10, 50]:
    C = quantization(I, K)
    D = reconstruction(C, K)
```

Figure 3.7: Reconstructions for three values of the loss parameter.



The compromise is quantified by plotting, as functions of K, the root mean square error between the reconstruction and the original image, and the proportion of coefficients that have been set to zero (Fig. 3.8).

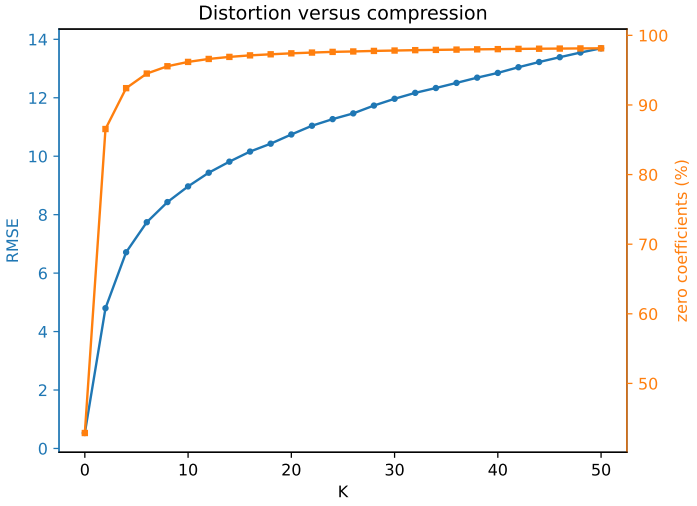
```python
Ks = np.arange(0, 51, 2)
rmse = []
zeros = []
for K in Ks:
    C = quantization(I, K)
    D = reconstruction(C, K)
    rmse.append(np.sqrt(np.mean((D - I)**2)))
    zeros.append(100*np.mean(C == 0))
```

```
K= 0  RMSE= 0.529  zeros= 42.9%
K= 2  RMSE= 4.800  zeros= 86.5%
K= 10  RMSE= 8.967  zeros= 96.2%
K= 50  RMSE= 13.692  zeros= 98.1%
```

The two curves are very dissymmetric: the proportion of zeros grows extremely fast for the small values of K, then saturates, whereas the error keeps increasing. The interesting working point is therefore located in the first part of the curve, around $K \simeq 2$ to 5: most of the compression is already obtained there, for an error that remains small.

Even for $K = 0$ the error is not exactly zero, and $43\,\%$ of the coefficients are already null: the quantization matrix is then filled with ones, but the conversion into integers still rounds the coefficients.

Figure 3.8: Distortion (RMSE) and proportion of zero coefficients as functions of the loss parameter K.



4 Introduction to wavelets

This tutorial introduces practically the basic wavelet decomposition and reconstruction algorithms. The objectives are to code some basic programs that will decompose and reconstruct 1D and 2D signals.

4.1. Introduction

Wavelets are based on a mother wavelet Ψ (s is a scale parameter, τ is the time translation factor $(s,\tau) \in \mathbb{R}_+^* \times \mathbb{R}$):

$$\forall t \in \mathbb{R}, \psi_{s,\tau}(t) = \frac{1}{\sqrt{s}}\Psi\left(\frac{t-\tau}{s}\right)$$

The continuous wavelet transform is written as follows, where ψ^* means the complex conjugate of ψ and $< .,. >$ is the complex scalar product (Hermitian form):

$$g(s,\tau) = \int_{-\infty}^{\infty} f(t)\psi_{s,\tau}^*(t)\mathrm{d}s\mathrm{d}\tau = < f, \psi_{s,\tau} >$$

The reconstruction is defined by:

$$f(t) = \frac{1}{C}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{1}{|s|^2}g(s,\tau)\psi_{s,\tau}(t)\mathrm{d}s\ \mathrm{d}\tau$$

with

$$C = \int_{-\infty}^{\infty}\frac{|\hat{\Psi}(\omega)|^2}{|\omega|}\mathrm{d}\omega$$

and $\hat{\Psi}$ is the Fourier Transform of Ψ.

The discrete wavelet transform corresponds to a sampling of the scales. To compute the different scales, one has to introduce a "father" wavelet, that similarly defines a family of functions orthogonal to the family $\psi_{s,\tau}$.

4.2. Fast discrete wavelet decomposition / reconstruction

A simple algorithm (cascade algorithm, from Mallat) is defined as two convolutions (by a lowpass ld filter for the projection on the ψ family, and a high pass hd filter for the orthogonal projection) followed by a subsampling (see Fig. 4.1).

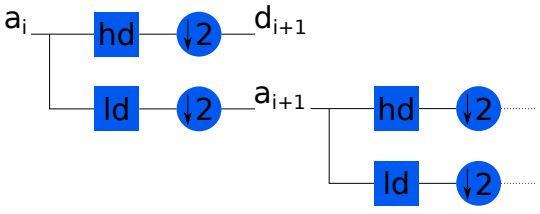


Figure 4.1: Algorithm for wavelet decomposition. First, a convolution is performed (square node), then, a subsampling. a_i stands for decomposition at scale i, d_i stands for detail at scale i.

4.2.1. Simple 1D example

Let's consider the signal $[4; 8; 2; 3; 5; 18; 19; 20]$. We will use the Haar wavelets defined by $ld = [1; 1]$ and $hd = [-1; 1]$. Basically, these filters perform a mean and a difference (see Table 4.1). The result of the decomposition in 3 scales with these wavelets is the concatenation of the details and the final approximation

$$C = \{[-4; -1; -13; -1]; [7; -16]; [-45]; [79]\}$$

Scale i	Approximation a_i	Details d_i
0 (original signal)	$[4; 8; 2; 3; 5; 18; 19; 20]$	
1	$[12; 5; 23; 39]$	$[-4; -1; -13; -1]$
2	$[17; 62]$	$[7; -16]$
3	$[79]$	$[-45]$

Table 4.1: Illustration of Haar decomposition in a simple signal.

?

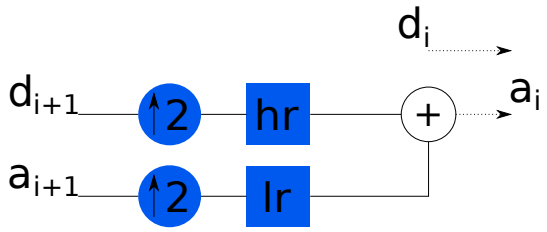
In the case of these Haar wavelets, code a function that performs the decomposition for a given number of scales. The prototype of this function will be:

```
def simpleWaveDec(signal, nb_scales):
    """
    wavelet decomposition of <signal> into <nb_scales> scales
    This function uses Haar wavelets for demonstration purposes.
    """
```

4.2.2. Reconstruction

To reconstruct the original signal (see Fig. 4.2), we need the definition of two reconstruction filters, hr and lr. For the sake of simplicity, we use $lr = ld/2$ and $hr = -hd/2$. These filter will perform an exact reconstruction of our original signal.

Figure 4.2: Reconstruction algorithm. The oversampling is done by inserting zeros.



Using this algorithm, you can now go on to the next exercise.

?

Code a function that performs the reconstruction of the signal. The prototype of this function will be, with the previous definition of C:

```python
def simpleWaveRec(C):
    """
    wavelet simple reconstruction function of a 1D signal
    C: Wavelet coefficients
    """
```

4.3. 2D wavelet decomposition

Let A be the matrix of an image. We consider that A is of size $2^n \times 2^n$, $n \in \mathbb{N}$. We consider, as for the 1D transform, the filters ld and hd.

The wavelet decomposition is as follows:

- Apply ld and hd on rows of A. Results are denoted $ld_r A$ and $hd_r A$, of size $2^n \times 2^{n-1}$, with r standing for row.
- Then, apply ld and hd again, to get the four new matrices: $ld_c ld_r A$, $ld_c hd_r A$, $hd_c ld_r A$ and $hd_c hd_r A$, of sizes $2^{n-1} \times 2^{n-1}$, with c standing for column. The matrix $ld_c ld_r A$ is the approximation, and the other matrices are the details.

?

- Code a function to perform the wavelet decomposition and for a given number of scales (lower than n). Test it on images of size $2^n \times 2^n$.
- Code the reconstruction function.

4.4. Built-in functions

Let us explore some built-in functions.

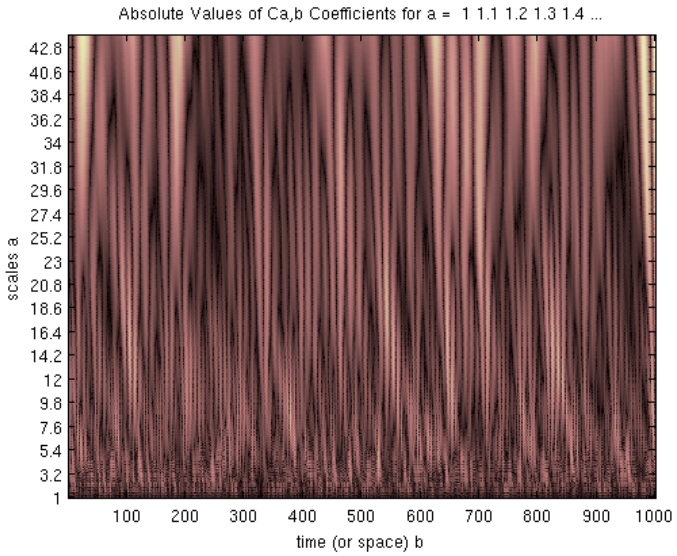
4.4.1. Continuous wavelet decomposition

One has to make the difference between the continuous wavelet transform and the discrete transform.

Informations

In scipy.signal you can find function to perform wavelets decomposition, and among them cwt for the continuous wavelet transform. There is also the module pywt that may be more developped (see also cwt).

Figure 4.3: Continuous wavelet transform.



4.4.2. Discrete decomposition

?

1. Generate the signal $\sin(2\pi * f_1 * t) + \sin(2\pi * f_2 * t)$, with $f_1 = 3$Hz and $f_2 = 50$Hz.
2. Apply the wavelet transformation for a given wavelet (like 'db4').
3. Display the 5th approximation of the wavelet decomposition.
4. Display the 4th detail coefficients of the wavelet decomposition.

Look at pywt.downcoef for the decomposition at a given level and pywt.wavedec for the entire decomposition.



4.5. Python correction



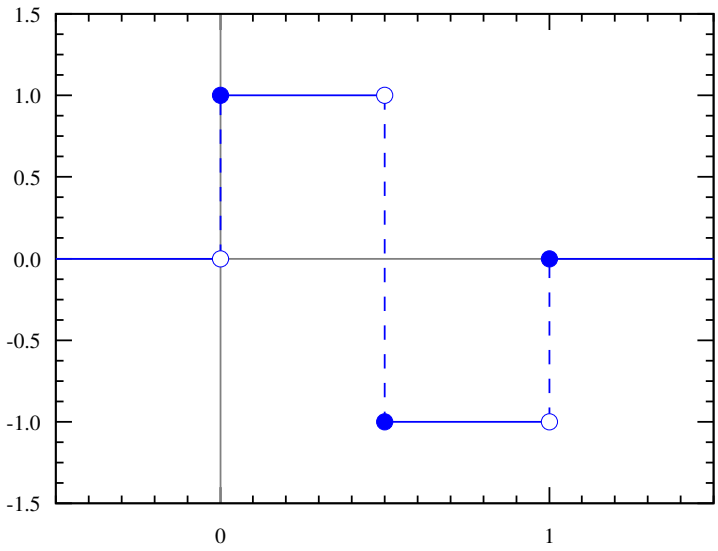


```
import numpy as np
from scipy import misc
import matplotlib.pyplot as plt
```

4.5.1. 1D signals

Two functions are required: a function (simpleWaveDec) that loops over the different scales and calls the second function (waveSingleDec) that performs the single step wavelet decomposition. Notice that the Haar wavelet is defined here with integer values (see Fig.4.4), so that the mental computation can be done easily.

Figure 4.4: Haar wavelets. From wikipedia, author Omegatron.



Simple 1D decomposition



```
def simpleWaveDec(signal, nb_scales):
    """
    wavelet decomposition of <signal> into <nb_scales> scales
    This function uses Haar wavelets for demonstration purposes.
    """
    # Haar Wavelets filters for decomposition and reconstruction
    ld = [1, 1];
    hd = [-1, 1];

    # transformation
    C=[];
    A = signal; # approximation
    for i in range(nb_scales):
        A, D = waveSingleDec(A, ld, hd);
        #get the coefficients
        C.append(D);

    C.append(A);
    return C;
```



```
def waveSingleDec(signal, ld, hd):
    """
    1D wavelet decomposition into
    A: approximation vector
    D: detail vector
    ld: low pass filter
    hd: high pass filter
    """
    # convolution
    A = np.convolve(signal, ld);
    D = np.convolve(signal, hd);
    # subsampling
    A = A[1::2];
    D = D[1::2];
    return A, D;
```

Notice that in this case, the use of the 'same' option is not required.

Simple 1D reconstruction

The reconstruction starts from the highest scale and computes the approximation signal with the given details.



```
def simpleWaveRec(C):
    """
    wavelet simple reconstruction function of a 1D signal
    C: Wavelet coefficients
    The Haar wavelet is used
    """
    ld = np.array([1, 1]);
    hd = np.array([-1, 1]);
    lr = ld/2;
    hr = -hd/2;

    A = C[-1];
    for scale in reversed(C[:-2]):
        A = waveSingleRec(A, scale, lr, hr);
    return A;
```



```
def waveSingleRec(a, d, lr, hr):
    """
    1D wavelet reconstruction at one scale
    a: vector of approximation
    d: vector of details
    lr: low pass filter defined by wavelet
    hr: high pass filter defined by wavelet
    This is Mallat algorithm.
    NB: to avoid side effects, the convolution function does not use the
    'same' option
    """
    approx = np.zeros((len(a)*2,));
    approx[::2] = a;
    approx = np.convolve(approx, lr);

    detail = np.zeros((len(a)*2,));
    detail[::2] = d;
    detail = np.convolve(detail, hr);
    # sum up approximation and details to reconstruct signal at lower scale
    approx = approx + detail;
    #get rid of last value
    approx = np.delete(approx, -1)
    return approx
```

Results

This is the result for the decomposition of the vector with 3 scales.



```
s = [4, 8, 2, 3, 5, 18, 19, 20];
print(s)
C = simpleWaveDec(s, 3);
print(C)
srec = simpleWaveRec(C);
print(srec);
```

```
[4, 8, 2, 3, 5, 18, 19, 20]
[array([ -4,  -1, -13,  -1]), array([  7, -16]), array([-45]), array([79])]
```

4.5.2. 2D signals

Decomposition

The simpleImageDec function is the main interface. It takes the image as first parameter, and the number of scales of decomposition. It makes a call to decWave2D for the decomposition at one given scale. The latter uses the previous 1D decomposition method.

```python
def decWave2D(image, ld, hd):
    """
    % wavelet decomposition of a 2D image into four new images.
    % The image is supposed to be square, the size of it is a power of 2 in the
    % x and y dimensions.
    """

    # Decomposition on rows
    sx, sy = image.shape;

    LrA = np.zeros((sx, int(sy/2)));
    HrA = np.zeros((sx, int(sy/2)));

    for i in range(sx):
        A, D= waveSingleDec(image[i,:], ld, hd);
        LrA[i,:]= A;
        HrA[i,:]= D;

    # Decomposition on cols
    LcLrA = np.zeros((int(sx/2), int(sy/2)));
    HcLrA = np.zeros((int(sx/2), int(sy/2)));
    LcHrA = np.zeros((int(sx/2), int(sy/2)));
    HcHrA = np.zeros((int(sx/2), int(sy/2)));
    for j in range(int(sy/2)):
        A, D= waveSingleDec(LrA[:,j], ld, hd);
        LcLrA[:,j] = A;
        HcLrA[:,j] = D;

        A, D= waveSingleDec(HrA[:,j], ld, hd);
        LcHrA[:,j] = A;
        HcHrA[:,j] = D;

    return LcLrA, HcLrA, LcHrA, HcHrA
```



```
def simpleImageDec(image, nb_scales):
    """
    wavelet decomposition of <image> into <nb_scales> scales
    This function uses Haar wavelets for demonstration purposes.
    """

    #Haar Wavelets filters for decomposition and reconstruction
    ld = [1,1];
    hd = [-1, 1];

    #transformation
    C=[];
    A = image; # first approximation

    coeffs = [];
    for i in range(nb_scales):
        [A, HcLrA, LcHrA, HcHrA] = decWave2D(A, ld, hd);
        coeffs.append( HcLrA);
        coeffs.append(LcHrA);
        coeffs.append(HcHrA);
        #set the coefficients
        C.append(coeffs.copy());
        coeffs.clear();
    C.append(A);
    return C;
```

2D reconstruction

The simpleImageRec function performs the reconstruction of a multiscale wavelet decomposition. The recWave2D performs the reconstruction of one scale.



```
def recWave2D(LcLrA, HcLrA, LcHrA, HcHrA, lr, hr):
    """
    Reconstruction of an image from lr and hr filters and from the wavelet decomposition.
    A: resulting (reconstructed) image

    NB: This algorithm supposes that the number of pixels in x and y dimensions is a power of 2.
    """

    sx, sy = LcLrA.shape;

    # Allocate temporary matrices
    LrA = np.zeros((sx*2, sy));
    HrA = np.zeros((sx*2, sy));
    A   = np.zeros((sx*2, sy*2));

    #Reconstruct from cols
    for j in range(sy):
        LrA[:,j] = waveSingleRec(LcLrA[:,j], HcLrA[:,j], lr, hr);
        HrA[:,j] = waveSingleRec(LcHrA[:,j], HcHrA[:,j], lr, hr);

    # Reconstruct from rows
    for i in range(sx*2):
        A[i,:] = waveSingleRec(LrA[i,:], HrA[i,:], lr, hr);

    return A;
```

```python
def simpleImageRec(C):
    """
    wavelet reconstruction of an image described by the wavelet coefficients C
    """
    #The Haar wavelet is used
    ld = np.array([1, 1]);
    hd = np.array([-1, 1]);
    lr = ld/2;
    hr = -hd/2;
    A = C[-1];
    for scale in reversed(C[:-1]):
        A = recWave2D(A, scale[0], scale[1], scale[2], lr, hr);

    return A;
```

Results

The illustration Fig. 4.5 is obtained by the following code. The useful functions are presented below.

```
I = imageio.imread('camera256.png');
C = simpleImageDec(I, 3);
Irec = simpleImageRec(C);
plt.imshow(Irec);
plt.show()
```

The image is recursively split into 4 areas, where the left upper corner is the approximation, and the three others are the details. As the details can have negative values, the intensities are adjusted in order to display the image correctly.

```python
def displayImageDec( C ):
    """
    Construct a single image from a wavelet decomposition
    C: the decomposition
    """
    n, m = C[0][0].shape;
    A = np.zeros((2*n, 2*m));
    prev = C[-1];
    for s, scales in reversedEnumerate(C[:-1]):

        ns = n / 2**(s-1);
        ms = m / 2**(s-1);
        T = imdec2im(prev, scales);
        A[0:int(ns), 0:int(ms)] = T;
        prev = A[0:int(ns), 0:int(ms)];
    return prev
```

These two functions adjust and reversedEnumerate are used to simplify the notations. The first one performs a linear stretching of the image intensities, and the second one allows an enumeration of a list in a reverse order.



```
def adjust(I):
    """
    simple image intensity stretching
    return I
    """
    I = I - np.min(I);
    I = I / np.max(I);
    return I;

def reversedEnumerate(l):
    """
    Utility function to perform reverse enumerate of a list
    returns zip
    """
    return zip(range(len(l)-1, -1, -1), l[::-1]);
```



```
def imdec2im(LcLrA, lvlC):
    """
    constructs a single image from:
    LcLrA: the approximation image
    lvlC: the wavelet decomposition at one level

    for display purposes
    """

    HcLrA=lvlC[0];
    LcHrA=lvlC[1];
    HcHrA=lvlC[2];
    n, m = HcLrA.shape;

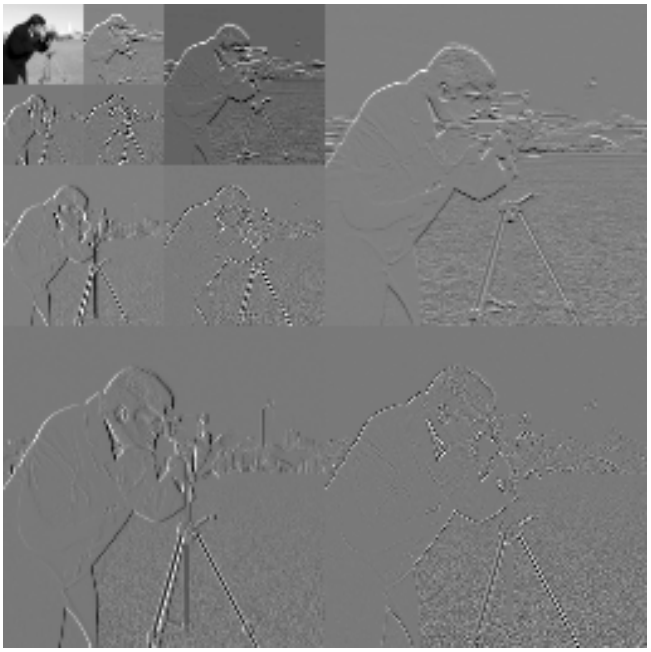
    A = np.zeros((2*n, 2*m));

    # Approximation image can be with high values when using Haar coefficients
    A[0:n, 0:m] =adjust(LcLrA);

    # details are low, and can be negative
    A[0:n, m:2*m] = adjust(HcLrA);
    A[n:2*n, 0:m] = adjust(LcHrA,);
    A[n:2*n, m:2*m] = adjust(HcHrA);

    return A;
```

Figure 4.5: (Haar) Wavelet decomposition of the cameraman image.



4.5.3. Built-in functions

An interesting module is pywt. This is illustrated by the following example. This gives the same results as previously, with the multiplication by $1/\sqrt{2}$.

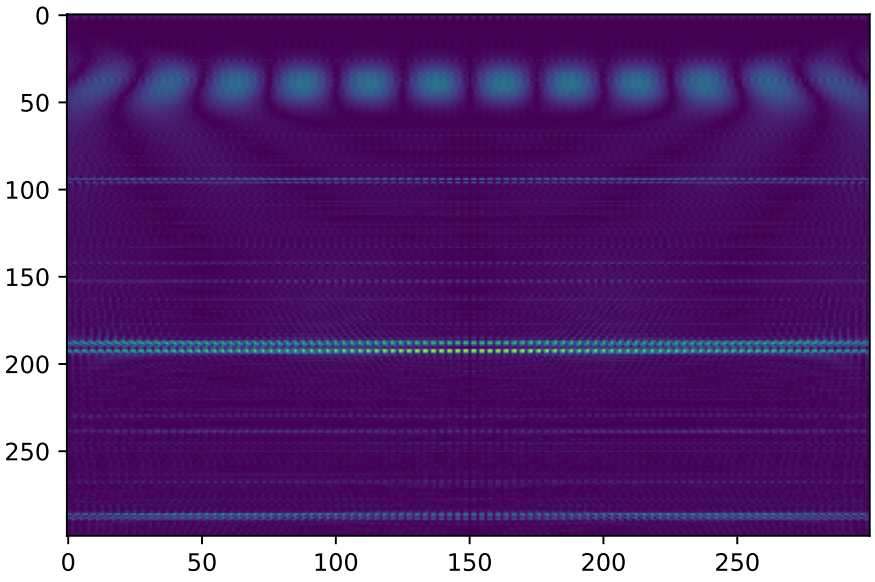
```
import pywt
cA, cD = pywt.dwt(s, 'haar');
print(cA, cD);
```

```
[ 8.48528137  3.53553391 16.26345597 27.57716447]
[-2.82842712 -0.70710678 -9.19238816 -0.70710678]
```

The continuous wavelet transform is applied in the following code and the result is displayed in Fig.4.6.

```
t = np.linspace(-1, 1, 1000, endpoint=False)
f1 = 3;
f2 = 50;
sig  = np.sin(2 * np.pi * f1 * t) + np.sin(2*np.pi * f2 * t);
widths = np.arange(1, 129)
coef, freqs=pywt.cwt(sig,widths,'morl')
fig = plt.figure();
plt.imshow(coef, extent=[-1, 1, 1, 31], cmap='PRGn', aspect='auto',
           vmax=abs(coef).max(), vmin=-abs(coef).max())
plt.show()
fig.savefig('cwt.python.pdf', bbox_inches='tight')
```

Figure 4.6: Continuous wavelet decomposition of the sum of sinusoids.



5 Multiscale Analysis

This tutorial aims to study some multiscale image processing and analysis methods, based on pyramidal and scale-space representations.

The different processes will be applied on the following MR image.

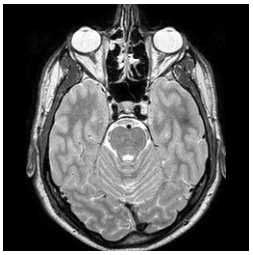


Figure 5.1: Brain MR image.

5.1. Pyramidal decomposition and reconstruction

5.1.1. Decomposition

Data: image A_0
Result: pyramid of approximations $\{A_i\}_i$, pyramid of details$\{D_i\}_i$
for *i=1 to 3* **do**
  filtering: $F = filt(A_{i-1})$;
  subsampling: $A_i = ech(F, 0.5)$;
  details: $D_i = A_{i-1} - ech(A_i, 2)$;
end

Algorithm 1: The algorithm of the pyramidal decomposition

The initial image (with the highest resolution), denoted A_0, represents the level 0 of the pyramid. To build the pyramid, we successively carry out the following steps:

1. a Gaussian filtering,
2. a subsampling,
3. a calculation of the details (residues).

?

Make a pyramidal decomposition with 4 levels of the image 'brain'.

5.1.2. Reconstruction

To reconstruct the original image, we carry out the following steps at each level of the pyramid:

1. an oversampling,
2. an addition of the details.

Data: image A_3, pyramid of details$\{D_i\}_i$
Result: reconstructed pyramid $\{B_i\}_i$
initialization : $B_3 = A_3$;
for *i=3 to 1* **do**
1 oversampling: $R = ech(B_i, 2)$;
2 adding details: $B_{i-1} = R + D_i$
end

Algorithm 2: The algorithm of the pyramidal reconstruction

?

1. Reconstruct the original image from the last level of the pyramid.
2. Make the same reconstruction without adding the details.
3. Calculate the resulting error between the reconstruted image and the original image.

5.2. Scale-space decomposition and multiscale filtering

We are going to decompose (without any sampling) the image with the highest resolution with a morphological operator, dilation or erosion. The resulting images will have the same size.

5.2.1. Morphological multiscale decomposition

?

Build the two scale-space decompositions of the 'brain' image, with a disk of increasing radius as a structuring element

Informations

See the functions morphology.erosion and morphology.dilation from skimage.

5.2.2. Kramer and Bruckner multiscale decomposition

Now, we are going to use the iterative filter of Kramer and Bruckner [20] defined as:

$$MK_B^n(f) = K_B(MK_B^{n-1}(f)) \tag{5.1}$$

where B denotes a disk of a fixed radius r, and:

$$K_B(f)(x) = \begin{cases} D_B(f)(x) & \text{if } D_B(f)(x) - f \leq f - E_B(f)(x) \\ E_B(f)(x) & \text{otherwise} \end{cases} \tag{5.2}$$

where $D_B(f), E_B(f)$ represent respectively the dilation and the erosion of the image f with the structuring element B.

?

Implement and test this filter on the image 'brain' for different values of n.



5.3. Python correction



5.3.1. Pyramidal decomposition and reconstruction

Decomposition

The following function makes the decomposition of the Laplacian and Gaussian pyramids at the same time. The Laplacian pyramid can be reconstructed without any additional information. This is illustrated in Fig. 5.2.



```
from skimage.transform import rescale, resize

def LaplacianPyramidDecomposition(Image, levels):
    """
    Laplacian / Gaussian Pyramid
    The last image of the laplacian pyramid allows a full reconstruction of the original image.
    Image: original image, float32
    levels : number of levels of decomposition

    returns: pyrL, pyrG: Laplacian and Gaussian pyramids, respectively, as a list of arrays
    """

    pyrL = [];
    pyrG = [];

    sigma = 3.;
    for l in range(levels):
        prevImage = Image.copy();
        g = ndimage.gaussian_filter(Image, sigma);
        print(g.dtype)

        Image = rescale(g, .5);
        primeImage= resize(Image, prevImage.shape);

        pyrL.append(prevImage - primeImage);
        pyrG.append(prevImage);

    pyrL.append(Image);
    pyrG.append(Image);
    return pyrL, pyrG;
```



Informations

rescale and resize functions have the same goal but use a scale or a shape as a parameter.

Reconstruction

The reconstruction is straighforward and exact because of the construction of the residue. The details can be filtered (removed for example), thus giving the following result Fig. 5.3.

Figure 5.2: Gaussian and Laplacian pyramids, for 3 levels of decomposition. The Laplacian pyramid in addition to the last level of the Gaussian pyramid is required to exactly reconstruct the original image.

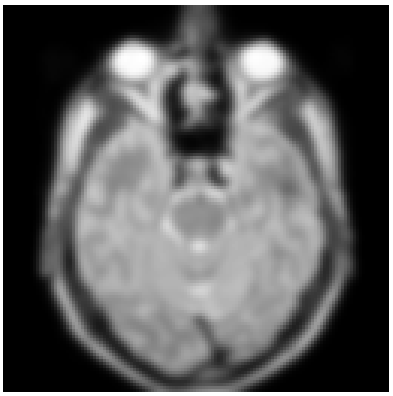
(a) Original image.



(b) Gaussian pyramid level 1.



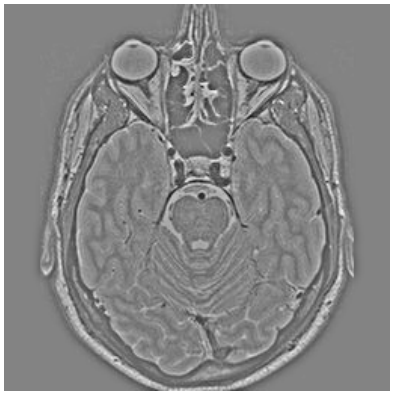
(c) Level 2.



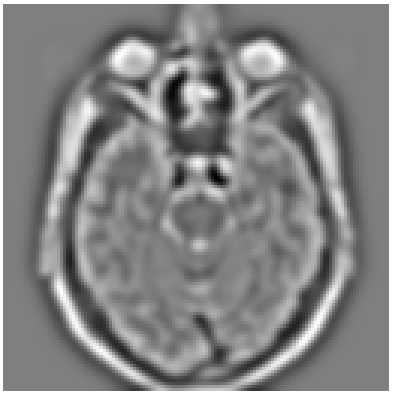
(d) Level 3.



(e) Laplacian pyramid level 1.



(f) Level 2.



(g) Level 3.



```python
def LaplacianPyramidReconstruction(pyr, interp='bilinear'):
    """
    Reconstruction of the Laplacian pyramid, starting from the last image
    pyr: pyramid of images (list of arrays)
    interp: interpolation mode, for upsizing the image
    returns: Image, reconstructed image
    """

    Image = pyr[-1];
    for i in range(len(pyr)-2, -1, -1):
        Image = pyr[i] + resize(Image, pyr[i].shape);

    return Image;
```

Figure 5.3: Reconstruction of the Laplacian pyramid.

(a) Reconstruction of the pyramid without any detail.



(b) Reconstruction of the pyramid with all the details.



5.3.2. Scale-space decomposition and multiscale filtering

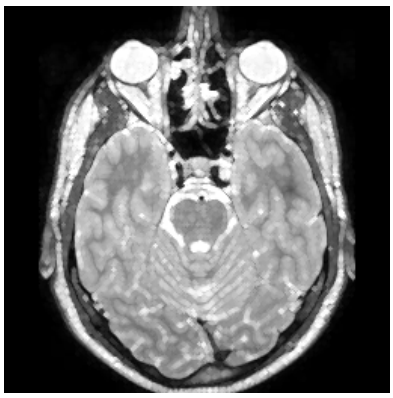
The pyramid of erosions and dilations is illustrated in Fig.5.4.

```python
def morphoMultiscale(I, levels):
    """
    Morphological multiscale decomposition
    I: original image, float32
    levels: number of levels, int

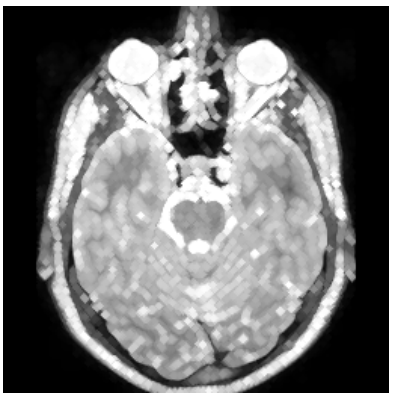
    returns: pyrD, pyrE: pyramid of Dilations/Erosions, respectively
    """
    pyrD=[];
    pyrE=[];
    for r in np.arange(1, levels):
        se = morphology.disk(r);
        pyrD.append(morphology.dilation(I, selem=se));
        pyrE.append(morphology.erosion(I, selem=se));
    return pyrD, pyrE;
```

Figure 5.4: Morphological multiscale decomposition by dilation and erosion.

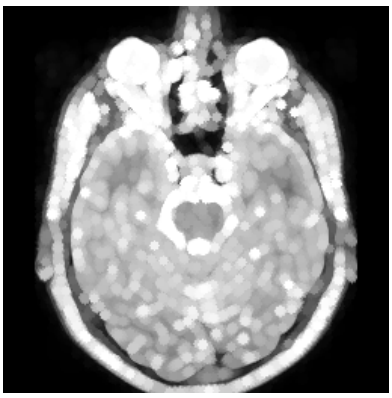
(a) Dilation scale 1.



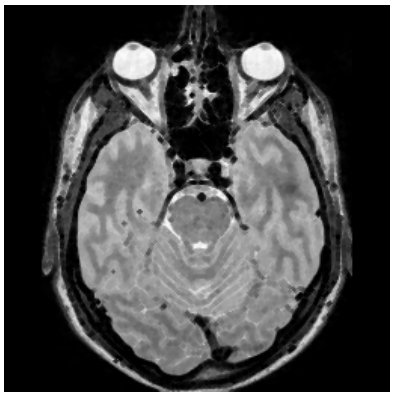
(b) Dilation scale 2.



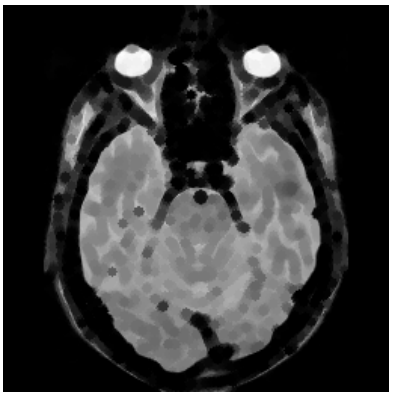
(c) Dilation scale 3.



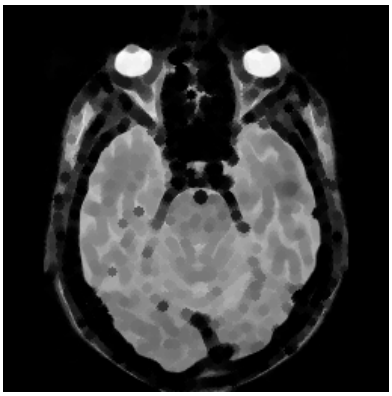
(d) Erosion scale 1.



(e) Erosion scale 2.



(f) Erosion scale 3.



5.3.3. Kramer and Bruckner multiscale decomposition

The results are illustrated in Fig.5.5.

```
def kb(I, r):
    """
    Elementary Kramer/Bruckner filter. Also called toggle filter.
    I: image
    r: radius of structuring element (disk), for max/min evaluation
    """
    se = morphology.disk(r);
    D=morphology.dilation(I, selem=se);
    E=morphology.erosion(I, selem=se);
    difbool = D-I < I-E;
    k = D*difbool + E * (~difbool);
    return k;
```

Figure 5.5: Kramer and Bruckner multiscale decomposition, with $r = 5$.

(a) MK_B^1.



(b) MK_B^2.



(c) MK_B^3.



```
def KBmultiscale(I, levels, r=1):
    """
    Kramer and Bruckner multiscale decomposition

    I: original image, float32
    pyrD: pyramid of Dilations
    pyrE: pyramid of Erosions

    returns: MKB: Kramer/Bruckner filters
    """
    MKB = [];
    MKB.append(I);
    for i in range(levels):
        MKB.append(kb(MKB[i-1], r));
    return MKB
```

6 Color Spaces

A color image can be represented in many different spaces, each one adapted to a particular use. This tutorial first observes the components of an image in two of them, $L^*a^*b^*$ and HSV, then goes back to the origin of all of them: the CIE 1931 chromaticity diagram, built from the colour matching functions, on which the MacAdam ellipses show why a Euclidean distance in this diagram is not a perceptual distance.

The first two sections use the following color image (Fig. 6.1), the image astronaut of skimage.

Figure 6.1: The test image.



```
from skimage import data
import matplotlib.pyplot as plt

I = data.astronaut() / 255.0
plt.imshow(I)
plt.show()
```

The conversions between the usual spaces are in skimage.color. For the colorimetric part, the module colour (*colour-science*, installed with pip install colour-science) gives the standard observer data and the MacAdam ellipses.

6.1. Components of an image in $L^*a^*b^*$

The $L^*a^*b^*$ space was built to be *perceptually uniform*: it separates the lightness L^* (from 0, black, to 100, white) from two opponent chromatic axes, a^* from green to red, and b^* from blue to yellow. The two chromatic components are centred on 0, which corresponds to a neutral grey.

1. Convert the image into $L^*a^*b^*$.
2. Display the three components in grayscale, one by one, and save each of them in a separate image file.
3. Which component looks like the grayscale version of the image? Where are the colored objects of the scene visible?


Informations

Use skimage.color.rgb2lab. To write a component as an image file, skimage.io.imsave expects integer values: skimage.exposure.rescale_intensity(C, out_range=np.uint8) brings an array of any range back between 0 and 255. The result is an image of exactly the size of the original one, with no axis and no colorbar.

6.2. Components of an image in HSV

HSV is nothing but the polar coordinates of the RGB cube around its grey axis: the hue H is an angle, the saturation S a distance to the axis, and the value V is $\max(R, G, B)$. It is widely used in image processing because the hue is easy to threshold.

?

1. Convert the same image into HSV, display and save its three components.
2. Look at the hue in the dark areas and in the grey areas of the image. What do you observe, and how do you explain it?
3. The hue is an *angle*: what problem does a grayscale display raise for this component?


skimage.color.rgb2hsv returns the three components normalized between 0 and 1.

6.3. Colour matching functions and the spectrum locus

All these spaces come from the same experiment: the matching of a monochromatic light by a mixture of three primaries. The result is the *standard observer*, that is to say the three colour matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$. The tristimulus values of a light of spectrum $S(\lambda)$ are:

$$X = \int S(\lambda)\bar{x}(\lambda)\,\mathrm{d}\lambda, \quad Y = \int S(\lambda)\bar{y}(\lambda)\,\mathrm{d}\lambda, \quad Z = \int S(\lambda)\bar{z}(\lambda)\,\mathrm{d}\lambda \tag{6.1}$$

The chromaticity coordinates are obtained by normalizing, which removes the intensity and keeps only the *color*:

$$x = \frac{X}{X+Y+Z}, \qquad y = \frac{Y}{X+Y+Z} \tag{6.2}$$

?

1. Plot the three colour matching functions of the CIE 1931 standard observer as functions of the wavelength.
2. Apply Eq. 6.2 to these functions and plot the result in the plane (x, y): this curve is the **spectrum locus**, the set of the monochromatic colors.
3. The locus is not closed. Join its two ends by a segment: this is the **line of purples**. Why do these colors not correspond to any wavelength?
4. Place a few wavelengths on the locus, as well as the point E of coordinates $(1/3, 1/3)$. What does this point represent?
5. Add the **Planckian locus**: the chromaticity of a black body as a function of its temperature, from 1000 to 25000 K. Mark a few temperatures, as well as the standard illuminants A and D_{65}. Where are the "warm" colors, and where are the "cold" ones?

Informations

The data of the standard observer are obtained with:

```
cmfs = colour.MSDS_CMFS["CIE 1931 2 Degree Standard Observer"]
```

cmfs.wavelengths gives the wavelengths, and cmfs.values the three functions, in the columns of an array. The spectrum of a black body at the temperature T is colour.sd_blackbody(T, shape); it is converted into tristimulus values by colour.sd_to_XYZ, exactly like any other light source. The chromaticities of the standard illuminants are tabulated in colour.CCS_ILLUMINANTS.

To color the inside of the diagram, each point (x, y) is converted back into sRGB (with $Y = 1$, then a normalization of the maximum of the three channels). The background is computed on the whole rectangle, then *clipped* by the locus: it is enough to build the polygon of the locus with matplotlib.patches.Polygon and to give it to the method set_clip_path of the image.

6.4. MacAdam ellipses

In 1942, MacAdam measured, around 25 colors of the diagram, the set of the chromaticities that an observer cannot distinguish from the central color. These regions are ellipses, of very different sizes and orientations depending on where they are in the diagram.

?

1. Draw the MacAdam ellipses in the diagram (x, y), magnified by a factor 10 so that they are visible.
2. Comment on their sizes according to the region of the diagram. What does that imply for a Euclidean distance computed in the plane (x, y)?
3. Transform the ellipses into the CIE 1976 UCS diagram (u', v') and compare:

$$u' = \frac{4x}{-2x + 12y + 3}, \qquad v' = \frac{9y}{-2x + 12y + 3}$$

colour.DATA_MACADAM_1942_ELLIPSES gives, for each ellipse, the centre (x_0, y_0), then the two semi-axes (in units of 10^{-3}) and the angle of the major axis in degrees.

This non-uniformity of the (x, y) diagram is exactly the reason why the CIE proposed, in 1976, the UCS diagram and the space $L^*a^*b^*$ of the first section: spaces in which an Euclidean distance is, at last, an acceptable approximation of a perceived difference.



6.5. Python correction



This correction uses skimage for the image and the conversions, and the module colour for the colorimetric data of the CIE.



```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

from skimage import data
from skimage.color import rgb2lab, rgb2hsv
from skimage.io import imsave
from skimage.exposure import rescale_intensity

import colour
```

6.5.1. Components in $L^*a^*b^*$

The three components are handled one by one. The components have very different ranges of values (L^* from 0 to 100, a^* and b^* negative or positive), so rescale_intensity brings each of them back between 0 and 255 before writing it: the image then has exactly the size of the original one, with no axis and no colorbar.



```
I = data.astronaut() / 255.0
imsave("astronaut.python.png", rescale_intensity(I, out_range=np.uint8))

Lab = rgb2lab(I)

L = Lab[..., 0]
plt.imshow(L, cmap='gray')
plt.title("$L^*$ (lightness)")
plt.show()
imsave("lab_L.python.png", rescale_intensity(L, out_range=np.uint8))

a = Lab[..., 1]
plt.imshow(a, cmap='gray')
plt.title("$a^*$ (green-red)")
plt.show()
imsave("lab_a.python.png", rescale_intensity(a, out_range=np.uint8))

b = Lab[..., 2]
plt.imshow(b, cmap='gray')
plt.title("$b^*$ (blue-yellow)")
plt.show()
imsave("lab_b.python.png", rescale_intensity(b, out_range=np.uint8))
```

L^* is indeed the grayscale version of the image. The two chromatic components are centred on zero and have a much smaller amplitude: the red suit stands out on a^*, the skin and the hair on b^*, while the white helmet and the grey background are dark on both — they are neutral colors, close to $a^* = b^* = 0$ (Fig. 6.2).

Figure 6.2: Components of the image in $L^*a^*b^*$. Each image is scaled between its own minimum and maximum.

(a) RGB image. (b) L^*. (c) a^*. (d) b^*.



6.5.2. Components in HSV

```
hsv = rgb2hsv(I)

H = hsv[..., 0]
plt.imshow(H, cmap='gray')
plt.title("hue $H$")
plt.show()
imsave("hsv_H.python.png", rescale_intensity(H, out_range=np.uint8))

S = hsv[..., 1]
plt.imshow(S, cmap='gray')
plt.title("saturation $S$")
plt.show()
imsave("hsv_S.python.png", rescale_intensity(S, out_range=np.uint8))

V = hsv[..., 2]
plt.imshow(V, cmap='gray')
plt.title("value $V$")
plt.show()
imsave("hsv_V.python.png", rescale_intensity(V, out_range=np.uint8))
```

The interesting observation is in the dark and grey areas (Fig. 6.3): the hue is completely noisy there. This is not a defect of the image but of the space itself: when the saturation tends to zero, the point gets close to the grey axis of the RGB cube and its angle is no longer defined. A hue-based segmentation must therefore always be weighted, or masked, by the saturation.

Figure 6.3: Components of the image in HSV. The hue is noisy wherever the saturation is low.

(a) RGB image. (b) hue H. (c) saturation S. (d) value V.





The hue is an *angle*: the values 0 and 1 are the same red. A grayscale display therefore creates an artificial discontinuity in the reds, which appear either black or white. It is the price to pay for a display that remains comparable with the two other components; a circular colormap (’hsv’) would be more faithful, but no longer a grayscale.

6.5.3. Colour matching functions and spectrum locus

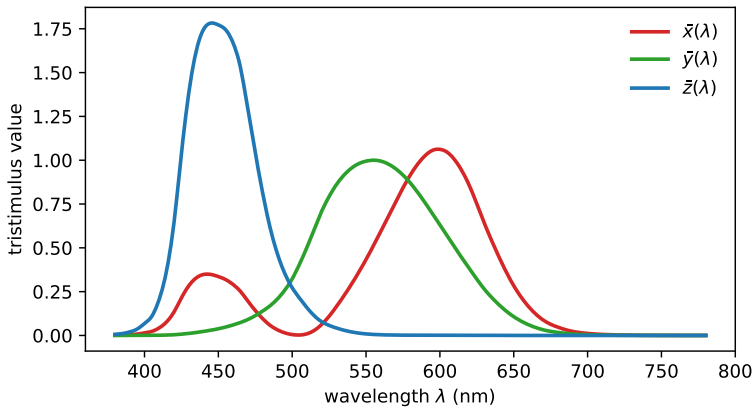
The standard observer is given directly by the module colour, as a table of the three functions sampled every nanometer (Fig. 6.4).



```
cmfs = colour.MSDS_CMFS["CIE 1931 2 Degree Standard Observer"]
sel = (cmfs.wavelengths >= 380) & (cmfs.wavelengths <= 780)
wl = cmfs.wavelengths[sel]
XYZ = cmfs.values[sel]

for k, name in enumerate([r"$\bar{x}$", r"$\bar{y}$", r"$\bar{z}$"]):
    plt.plot(wl, XYZ[:, k], label=name)
```

Figure 6.4: The colour matching functions of the CIE 1931 standard observer.



The spectrum locus is nothing but these three functions, normalized point by point: each wavelength gives one point of the plane (x, y).



```
locus = XYZ[:, :2] / XYZ.sum(axis=1, keepdims=True)
```

The inside of the diagram is colored by converting each chromaticity back into sRGB, then by making transparent what lies outside the locus.

```python
def chromaticity_background(n=400):
    """
    Background of the diagram: the sRGB color of the chromaticity (x, y) of
    each point of the plane. The values are normalized so that the brightest
    channel is saturated, since only the color matters here.
    """
    x, y = np.meshgrid(np.linspace(0, .8, n), np.linspace(0, .9, n))
    Y = np.ones_like(x)
    XYZg = np.stack([x*Y/np.maximum(y, 1e-6), Y,
                     (1-x-y)*Y/np.maximum(y, 1e-6)], -1)
    rgb = colour.XYZ_to_sRGB(XYZg)
    return np.clip(rgb / np.maximum(rgb.max(-1, keepdims=True), 1e-6), 0, 1)
```

The background is computed on the whole rectangle, then *clipped* by the locus: there is no need to test, point by point, whether a chromaticity is inside the polygon.

```python
im = ax.imshow(chromaticity_background(), origin='lower',
               extent=[0, .8, 0, .9])

polygon = Polygon(locus, closed=True, facecolor='none', edgecolor='k')
ax.add_patch(polygon)
im.set_clip_path(polygon)

ax.plot([locus[-1, 0], locus[0, 0]], [locus[-1, 1], locus[0, 1]],
        '--', color='#B0198E', label="line of purples")
```

The locus is open because the two extreme wavelengths, around 380 and 780 nm, are not neighbours in the plane. The segment that joins them, the **line of purples**, carries colors that *no monochromatic light can produce*: the purples only exist as mixtures of the two ends of the spectrum. The point E of coordinates $(1/3, 1/3)$ is the equal energy white.

The **Planckian locus** is obtained in the same way as any other light source: the spectrum of a black body at the temperature T is integrated against the colour matching functions.

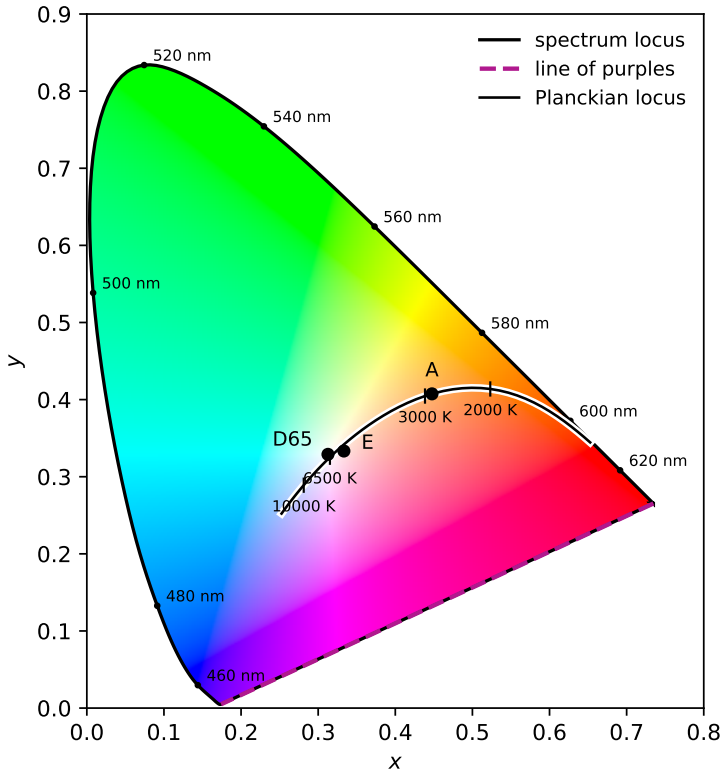
```python
shape = colour.SpectralShape(360, 830, 5)
temperatures = np.geomspace(1000, 25000, 60)
planck = np.array([colour.XYZ_to_xy(colour.sd_to_XYZ(
    colour.sd_blackbody(T, shape), cmfs)) for T in temperatures])

ax.plot(planck[:, 0], planck[:, 1], 'w', lw=3.5)        # casing
ax.plot(planck[:, 0], planck[:, 1], 'k', lw=1.2, label="Planckian locus")

illuminants = colour.CCS_ILLUMINANTS["CIE 1931 2 Degree Standard Observer"]
```

It is the curve followed by the color of a heated body, from the red of an ember at 1000 K to the blue-white of a hot star: the "warm" colors of everyday language are therefore the *low* temperatures, on the right of the diagram, and the "cold" ones the high temperatures, on the left. The incandescent lamp A (2856 K) and the daylight D_{65} (6500 K) are both very close to this curve, which is the whole point of characterizing an illuminant by its *color temperature* (Fig. 6.5).

Figure 6.5: The CIE 1931 chromaticity diagram: spectrum locus, line of purples, Planckian locus and standard illuminants.



6.5.4. MacAdam ellipses

The data give the centre of each ellipse, its two semi-axes in units of 10^{-3} and the angle of its major axis. The magnification by 10 is only there to make them visible.

```python
def macadam_ellipses(magnification=10, nb_points=72):
    """
    MacAdam ellipses (1942), as a list of polygons in the (x, y) plane.
    The data give the semi-axes in units of 1e-3, and the angle in degrees.
    """
    t = np.linspace(0, 2*np.pi, nb_points)
    ellipses = []
    for x0, y0, _, _, _, a, b, theta in colour.DATA_MACADAM_1942_ELLIPSES:
        a, b = a*1e-3*magnification, b*1e-3*magnification
        theta = np.radians(theta)
        ex = x0 + a*np.cos(theta)*np.cos(t) - b*np.sin(theta)*np.sin(t)
        ey = y0 + a*np.sin(theta)*np.cos(t) + b*np.cos(theta)*np.sin(t)
        ellipses.append(np.stack([ex, ey], axis=1))
    return ellipses

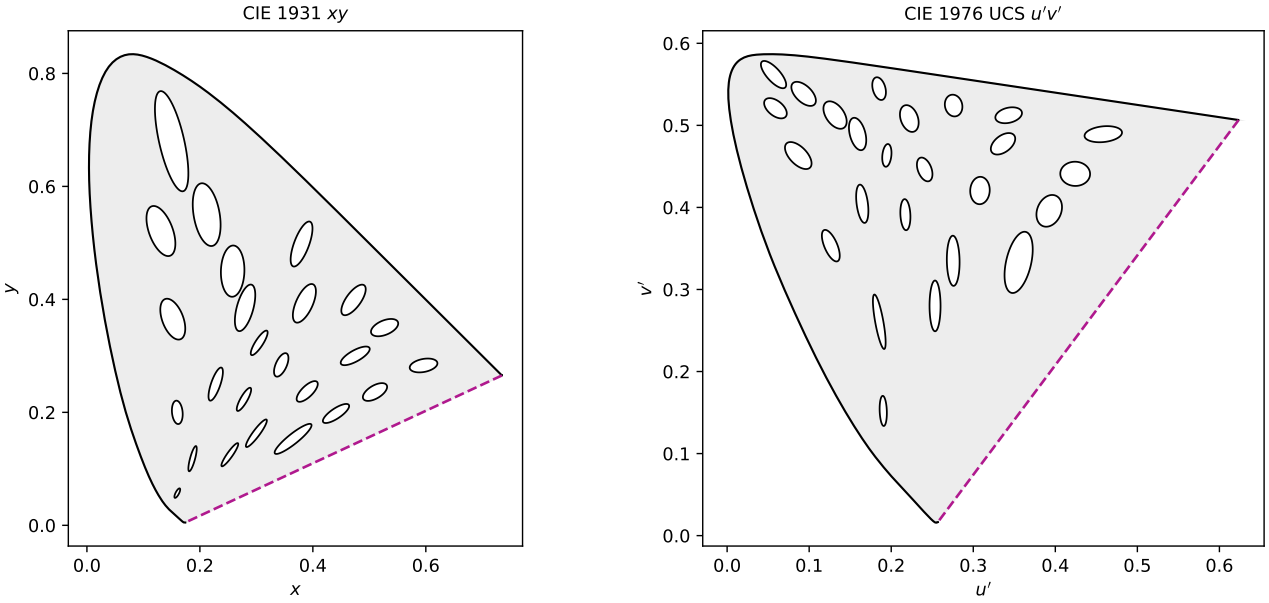

def xy_to_uv(xy):
    """ CIE 1976 UCS coordinates (u', v') from the (x, y) ones """
    x, y = xy[..., 0], xy[..., 1]
    d = -2*x + 12*y + 3
    return np.stack([4*x/d, 9*y/d], axis=-1)
```

In the diagram (x, y), the ellipses of the green region are about ten times larger than those of the blue region, and their orientations vary (Fig. 6.6). Two colors separated by the same Euclidean distance are therefore

clearly different in the blues, and *indistinguishable* in the greens: a distance computed in this plane has no perceptual meaning.

The transformation into the UCS diagram (u', v'), which is a simple homography, makes them much closer to circles of equal size. The uniformity is not perfect — the ellipses of the blue region remain smaller — but the approximation becomes usable, which is the whole purpose of this diagram and of the $L^*a^*b^*$ space.

Figure 6.6: The MacAdam ellipses (magnified 10 times) in the two diagrams.



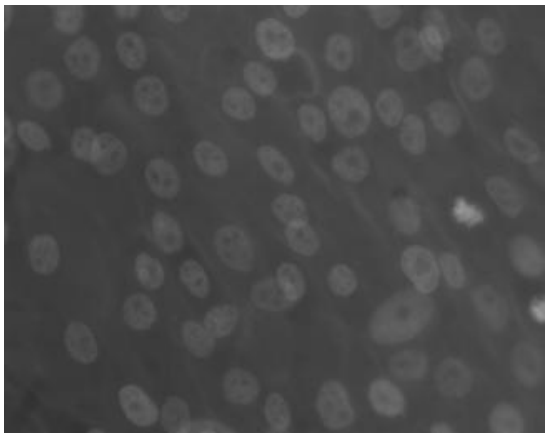
Part III Enhancement and Restoration

7 Image Enhancement

The objective of this tutorial is to implement some image enhancement methods, based on intensity transformations or histogram modifications. It will make use of statistical notions like probability density functions or cumulative distribution functions.

Figure 7.1: The different processes of this tutorial will be applied on these images.

(a) Osteoblasts.

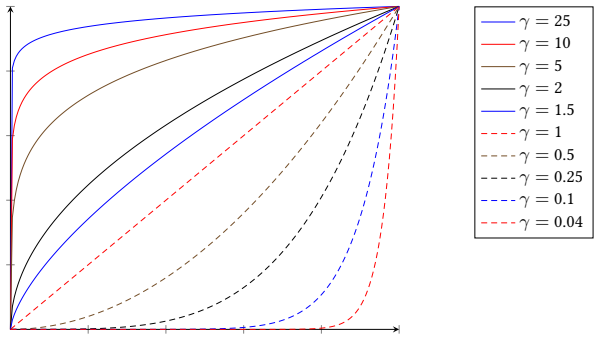


(b) Phobos (ESA/DLR/FU Berlin, CC-By-SA).

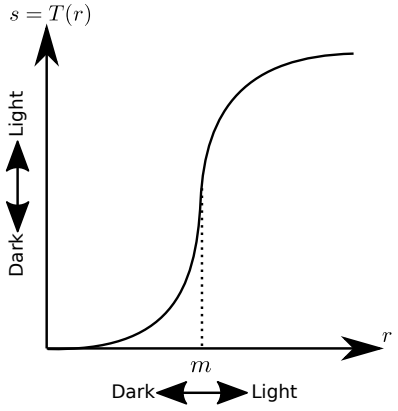


7.1. Intensity transformations (LUT)

Two transformations will be studied: the γ correction and the contrast stretching. They enable the intensity dynamics of the gray tone image to be changed. These two operators are based on the following Look Up Tables (LUT):



(c) γ correction LUT.



(d) Contrast stretching LUT.

?

1. Test the transformation 'γ correction' on the image 'osteoblast'.
2. Implement the operator 'contrast stretching' with the following LUT (also called cumulative distribution function cdf), with m being the mean gray value of the image, and r being a given gray value:

$$s = T(r) = \frac{1}{1 + (m/r)^E}$$

3. Test this transformation with different values of E on the image 'osteoblast'.

The module skimage.exposure contains different methods for contrast enhancement, among them adjust_gamma.

7.2. Histogram equalization

The objective is to transform the image so that its histogram would be constant (and its cumulative distribution function would be linear). The notations are:

- I is the image of n pixels, with intensities between 0 and L (for 8-bits images, $L = 255$).
- h is the histogram, defined by:

$$h_I(k) = \mathbb{P}(x = k) = \frac{n_k}{n}, \quad 0 \leq k \leq L,$$

where $\mathbb{P}$ denotes the probability. The following transformation $T(I)$ is called histogram equalization.

$$T(x_k) = L \cdot \mathrm{cdf}_I(k)$$

where

$$\mathrm{cdf}_I(k) = \sum_{j=0}^{k} \mathbb{P}(x_j)$$

is the cumulative distribution function (cumulative histogram).

?

1. Compute and visualize the histogram of the image 'osteoblast'.
2. Test this histogram equalization transformation on the image 'osteoblast' (with builtin functions) and visualize the resulting histogram.
3. Code your own function.
4. The corresponding LUT to this transformation is the cumulative sum of the normalized histogram. Evaluate and visualize this intensity transformation.

See the numpy.histogram and skimage.exposure functions.

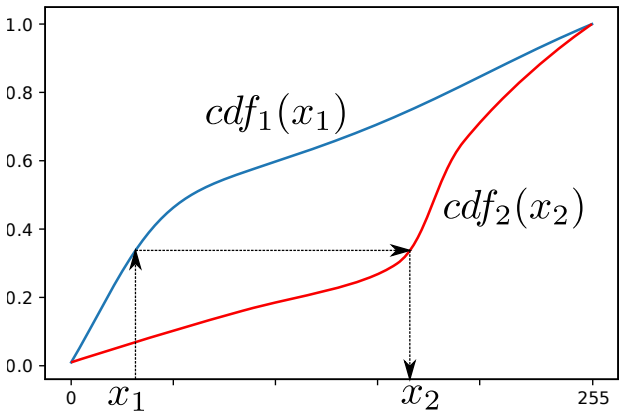
7.3. Histogram matching

The objective is to enhance the original image by matching its histogram with a modeled one. The principle is to transform the gray value x_1 of first image into x_2: $T(x_1) = x_2$ (see Fig.7.2). Based on the fact that $\mathrm{cdf}_1(x_1) = \mathrm{cdf}_2(x_2)$, the formula to find x_2 is:

$$x_2 = \mathrm{cdf}_2^{-1}\left(\mathrm{cdf}_1(x_1)\right).$$

As x_1 and x_2 are discrete values, this requires an interpolation.

Figure 7.2: Histogram matching principle.



?

1. Visualize the histogram of the image 'phobos'.
2. Make the histogram equalization and visualize the resulting image.
3. Construct a bi-modal histogram (for example) and code your own function for histogram matching.



See numpy.interp for interpolation and LUT application.

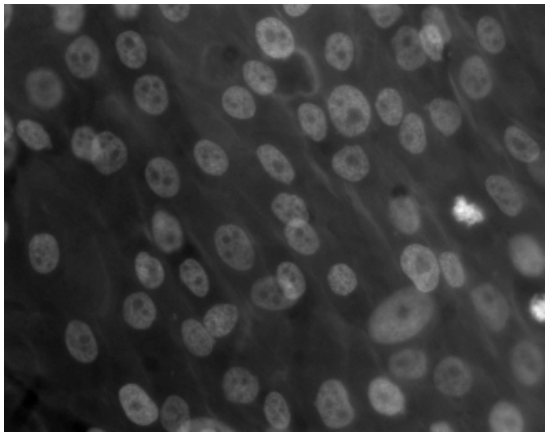


7.4. Python correction

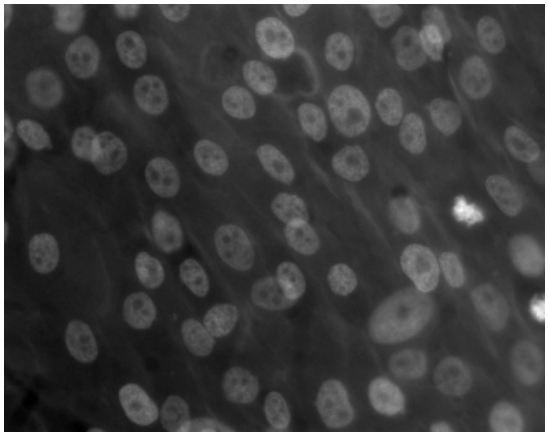


Figure 7.3: Gamma transforms.

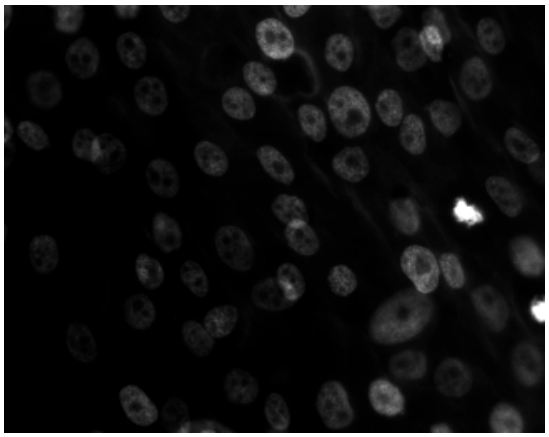
(a) Original image.



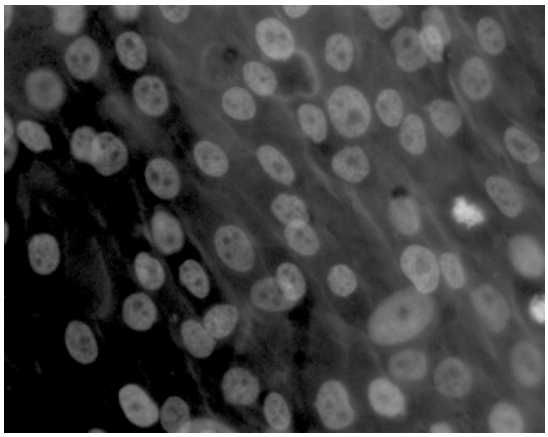
(b) $\gamma = 1.$



(c) $\gamma = 2.$



(d) $\gamma = 0.5.$





```
from scipy import misc
import matplotlib.pyplot as plt
from skimage import exposure
import numpy as np
import sys
```

7.4.1. Intensity transformations

γ correction

The function skimage.exposure.adjust_gamma is used in order to adjust the gamma of the image. It is before normalized (values are float between 0 and 1). The results are illustrated in Fig.7.3.

```
I=imageio.imread(" osteoblaste .png")
I = I / np.max(I)

gamma=2
I2= exposure.adjust_gamma(I, gamma)
plt .imshow(I2)
plt .show()
```

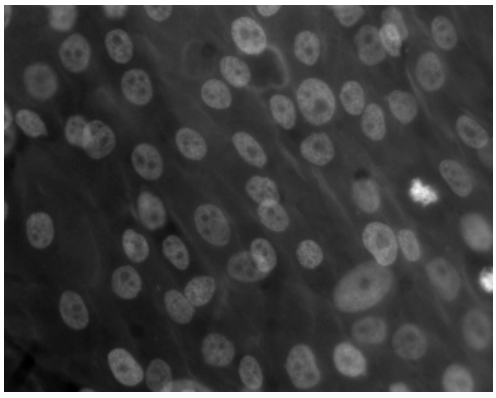
Contrast stretching

The LUT look-up-table is simply a function applied to the original image. In order to avoid division by zero, the smallest float value is introduced. Fig.7.4 illustrates the results.

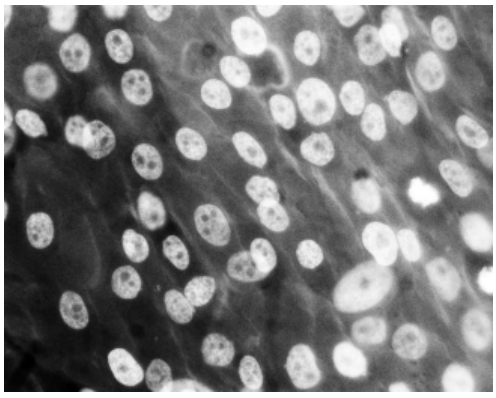
```
def contrast_stretching (I, E):
    epsilon = sys. float_info .epsilon
    m = np.mean(I)
    I = I.astype(" float ")
    Ar = 1. / (1.+( m/(I+epsilon))**E)
    return Ar
```

Figure 7.4: Constrast stretching.

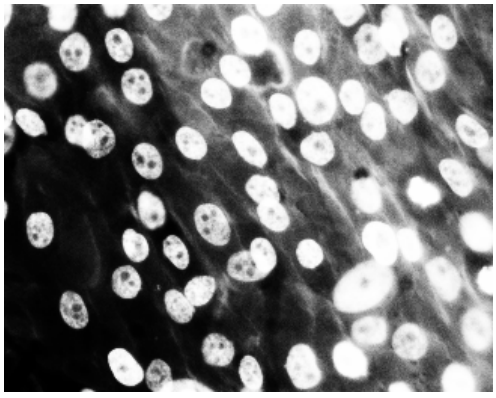
(a) Original image.



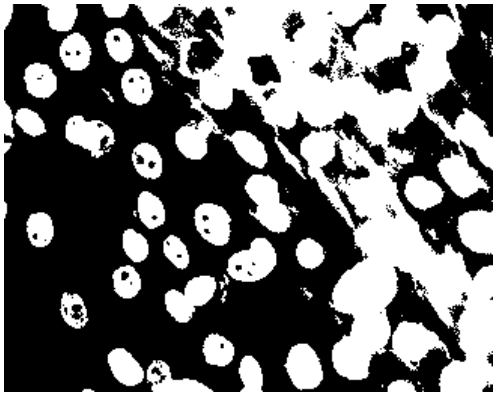
(b) $E = 10$.



(c) $E = 20$.



(d) $E = 1000$.



7.4.2. Histogram equalization

The histogram is displayed with the following function:

```
def displaySaveHisto(I, filename=None):
    """
    Display and save pdf (if filename provided) of histogram of image I
    If values are between 0 and 1, they are multiplied by 255
    """
    if np.max(I)<=1:
        I = 255 * I
    hist, bins = np.histogram(I, 256, range=(0,255))
    fig = plt.figure()
    plt.bar(bins[:-1], hist, width=1)
    plt.show()
    if filename!=None:
        fig.savefig(filename, bbox_inches='tight')
```

The histogram equalization is done via the skimage.exposure. equalize_hist function. The results are illustrated in Fig.7.5.

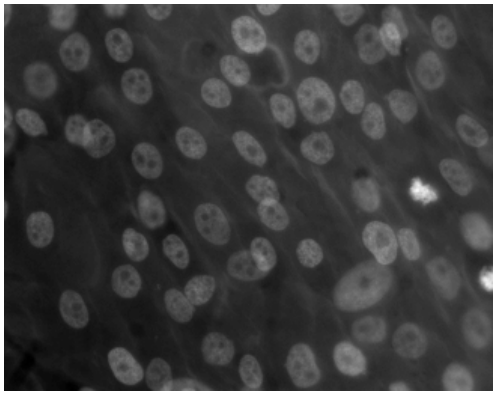
```
# histogram equalization
I2 = exposure.equalize_hist(I)
plt.imshow(I2)
plt.show()
```

The look-up-table (LUT) or cumulative distribution function (cdf) is computed and used in the next function:

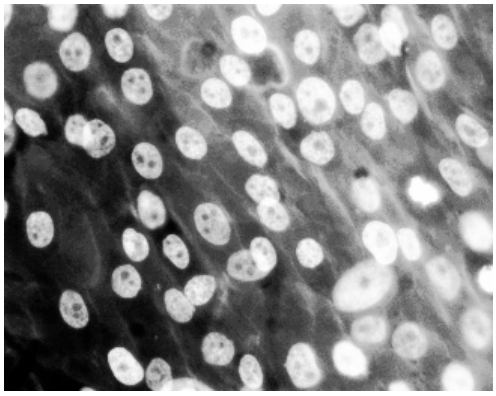
```
def histeq(I):
    """
    histogram equalization, version with look-up-table
    I: original image, with values in 8 bits integer
    """
    hist,bins=np.histogram(I, 256, range=(0,255))
    cdf = hist.cumsum()
    cdf = (cdf / cdf[-1])
    return cdf[I]
```

Figure 7.5: Histogram equalization.

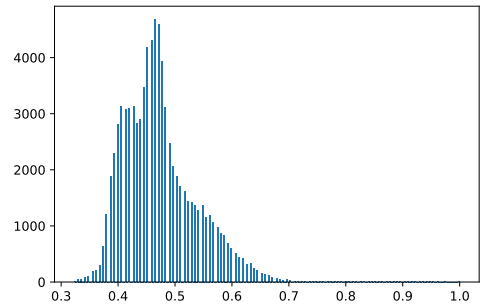
(a) Original image.



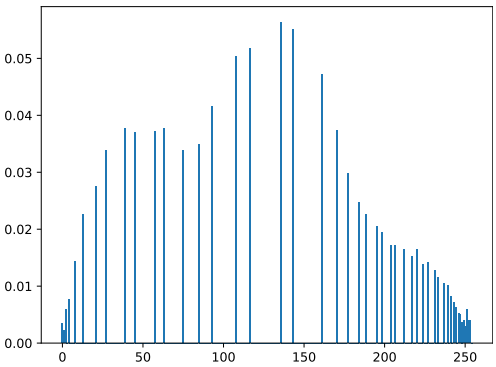
(b) Histogram equalization.



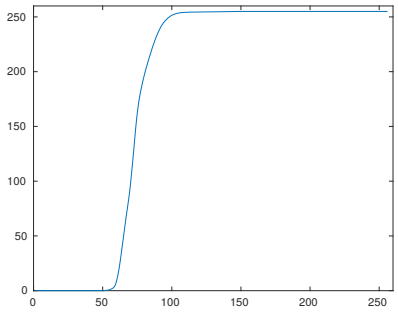
(c) Histogram of original image.



(d) Histogram after equalization.



(e) LUT (cumulative distribution function).



7.4.3. Histogram matching

This method is only an approximate method. It requires an interpolation of the cumulative distribution functions in order to find the transformation. Results are shown in Fig.7.6

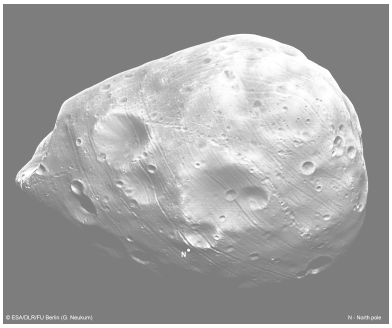
```python
def hist_matching(I, cdf_dest):
    """
    Histogram matching of image I, with cumulative histogram cdf_dest
    This should be normalized, between 0 and 1.
    This version uses interpolation
    """
    imhist,bins = np.histogram(I, len(cdf_dest), density=True)
    cdf = imhist.cumsum() #cumulative distribution function
    cdf = (cdf / cdf[-1]) #normalize between 0 and 1
    # first : histogram equalization
    im2 = np.interp(I.astype(np.float64), bins[:-1], cdf)
    # 2nd: reverse function
    imres = np.interp(im2, cdf_dest, bins[:-1])
    return imres
```

Figure 7.6: Histogram matching.

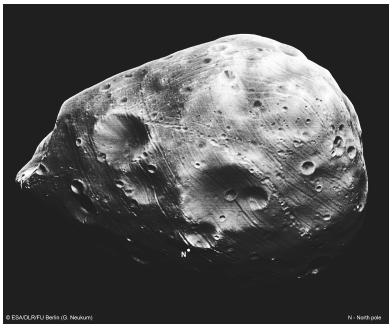
(a) Original image.



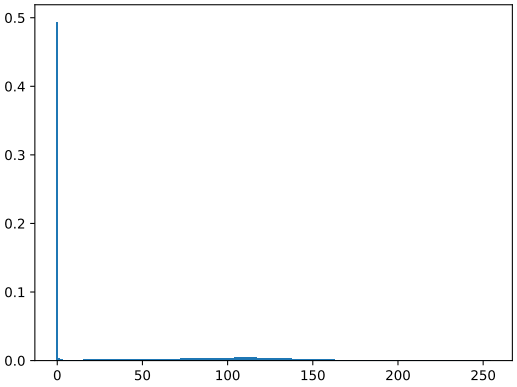
(b) Histogram equalization.



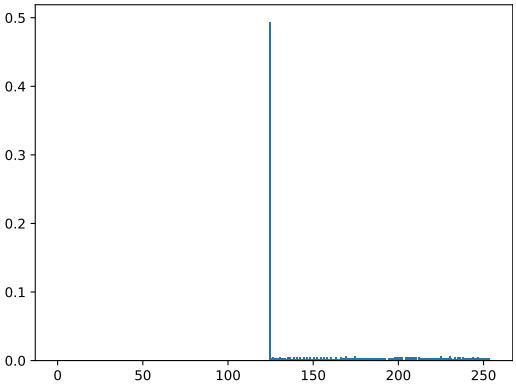
(c) Histogram matching.



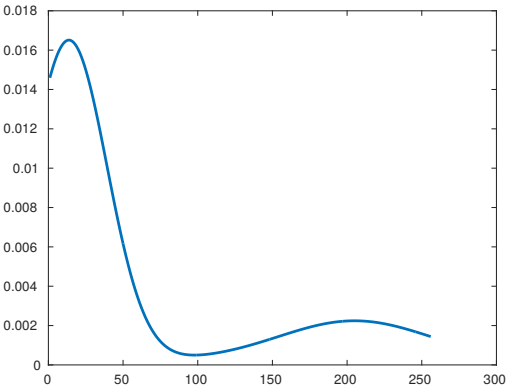
(d) Histogram of original image.



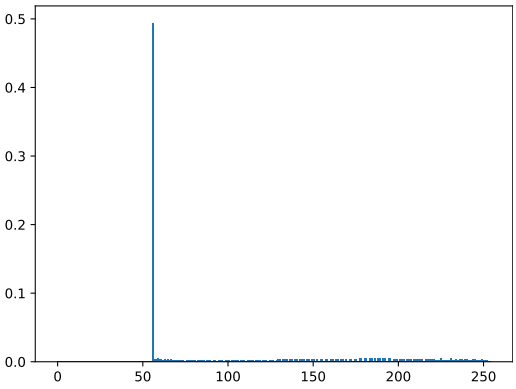
(e) Histogram after equalization.



(f) Target histogram (cumulative distribution function).



(g) Histogram after histogram matching.



8 Image restoration: an enhanced median filter

This tutorial aims to study some random noises and to test different image restoration methods (image denoising).

The different processes will be applied on the following MR image.



Figure 8.1: Leg.

8.1. Generation of random noises

Some random noises are defined with the given functions (corresponding to specific distributions):

- uniform noise: $R(p) = a + (b - a) * \mathcal{U}(0, 1)$.
- Gaussian noise: $R(p) = a + b * \mathcal{N}(0, 1)$.
- Salt and pepper noise: $R : \begin{cases} 0 \leq \mathcal{U}(0,1) \leq a & \mapsto & 0 \\ a < \mathcal{U}(0,1) \leq b & \mapsto & 0.5 \\ b < \mathcal{U}(0,1) \leq 1 & \mapsto & 1 \end{cases}$
- Exponential noise : $R = -\frac{1}{a} * \ln(1 - \mathcal{U}(0, 1))$

?

Generate sample images with the four kinds of random noise and visualize their histograms with the built-in functions. Pay attention to the intensity range of the resulting images when calculating the histograms

Use np.random.rand or np.random.randn to generate random numbers with uniform or normal laws.

8.2. Noise estimation

The objective is to evaluate the characteristics of the noise in a damaged/noisy image (in a synthetic manner in this tutorial).

?

1. Visualize the histogram of a Region Of Interest (ROI) of the image of Fig.8.1. The ROI should be extracted from a uniform (intensity) region.
2. Add an exponential noise to the original image and visualize the histogram of the selected ROI.
3. Add a Gaussian noise to the original image and visualize the histogram of the selected ROI.

8.3. Image restoration by spatial filtering

?

1. Add a salt-and-pepper noise to the image of Fig.8.1.
2. Test the 'min', 'max', 'mean' and 'median' image filters.

With skimage, look at skimage. filters .rank for the maximum, minimum and mean filters, and at skimage. filters for the median filter.

The median filter is efficient in the case of salt-and-pepper noise. However, it replaces every pixel value (first problem) by the median value determined at a given scale (second problem). In order to avoid these two problems, Gonzalez and Woods proposed an algorithm [11].

Data: Input (noisy) image I
Data: Maximal scale S_{max}
Result: Filtered image F

Main Function `amf` (I, S_{max})**:**
  forall *pixels* (i, j) **do**
    $S \leftarrow 1$
    while `isMedImpulseNoise`(I, i, j, S) *AND* $S \leq S_{max}$ **do**
      $S \leftarrow S + 1$
    end
    if $I(i,j) =$`Min`(I, i, j, S) OR $I(i,j) =$`Max`(I, i, j, S) OR $S = S_{max}$ **then**
      $F(i,j) \leftarrow$ `Med`(I, i, j, S)
    else
      $F(i,j) \leftarrow I(i,j)$
    end
  end

Algorithm 3: Adaptive median filter [11]. The main function takes two arguments: the image I to be filtered and the maximal size of neighborhood S_{max}. For each pixel (i, j), the good scale is the scale where the median value is different from the min and the max (median value is thus not an impulse noise). Then, if the pixel value is an impulse noise or the maximal scale has been reached, it must be filtered. Otherwise, it is kept untouched. Min, Max and Med functions compute the minimum, maximum and the median value in a neighborhood of size S centered at a pixel (i, j) in image I.

?

Implement the adaptive median filter from the following algorithm (Alg.3) based on two steps (the operator acts on an operational window of size $k \times k$ where different statistics are calculated: median, min or max).

Function `isMedImpulseNoise`(I, i, j, S)**:**
  $med \leftarrow$ `Med`(I, i, j, S)
  if med=`Max`(I, i, j, S) OR med=`Min`(I, i, j, S) **then**
    return True
  else
    return False
  end

Algorithm 1: Test if the pixel corresponds to an impulse noise, i.e. is the median a min or a max for a given size.



8.4. Python correction





```
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage
from scipy import misc
```

8.4.1. Generation of random noises

In order to convert stretch the images into the range [0;1], the following function can be used:



```
def hist_stretch(I):
    # histogram stretching
    # returns values of I linearly stretched to range [0;1]
    I = I - np.min(I);
    I = I / np.max(I);
    return I;
```

Random noise illustrations are proposed in Figs.8.2 and 8.3, and a size $S = 32$ is used to generate the noisy images.

Uniform noise

The python rand function generates values between 0 and 1 with uniform distribution.



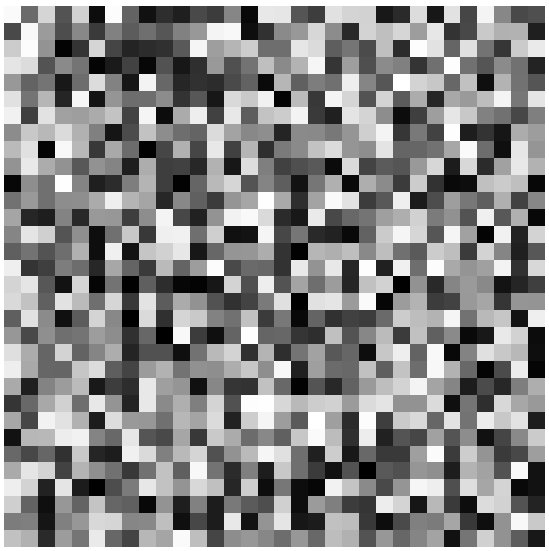
```
S=32
a=0; b=255;
R1 = a + (b-a) * np.random.rand(S, S);
```

Gaussian noise

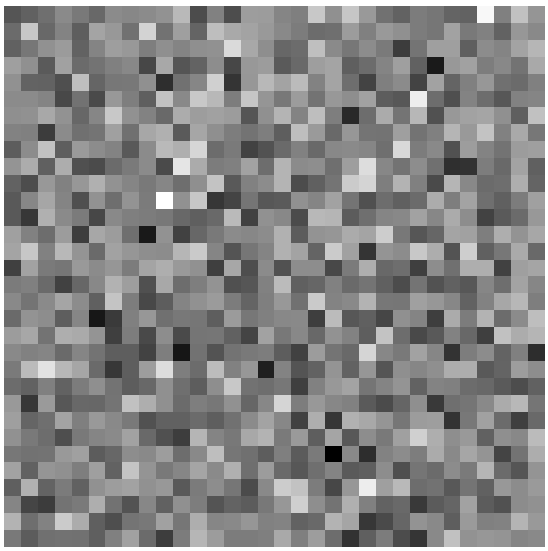
The python randn function generates values with normal centered distribution.

Figure 8.2: Resulting noise images.

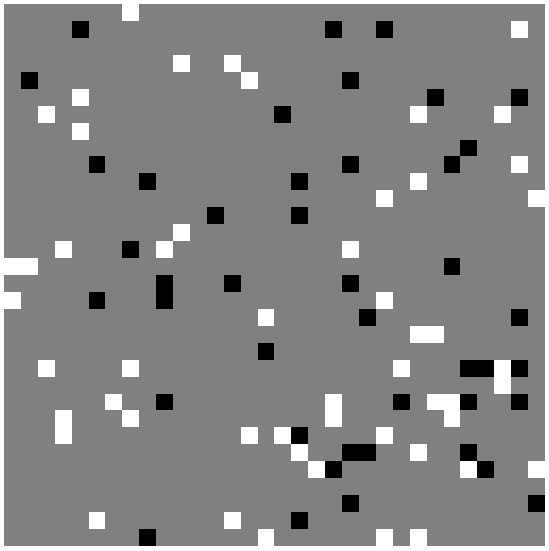
(a) Uniform noise.



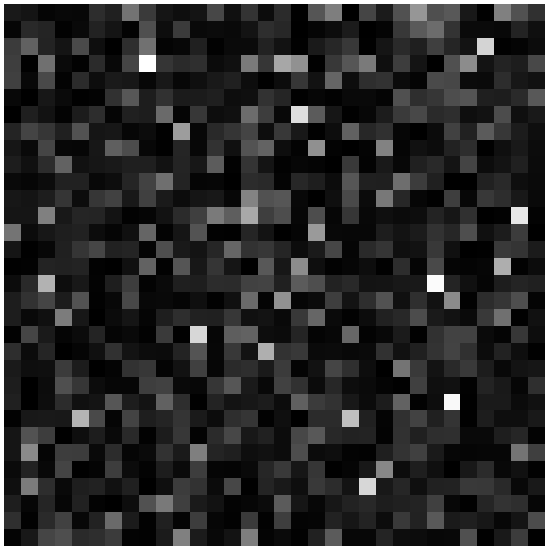
(b) Gaussian noise.



(c) Salt and pepper noise.



(d) Exponential noise.





```
a=0; b=1;
R2 = a + (b-a)*np.random.randn(S, S);
```

Salt and pepper noise



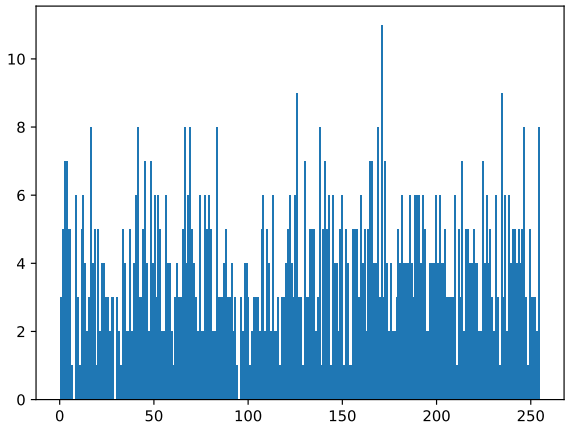
```
a = 0.05; b = 0.1;
R3 = 0.5 * np.ones((S,S));
X = np.random.rand(S,S);
R3[X<=a] = 0;
R3[(X>a) & (X <= b)] = 1;
R3 = hist_stretch (R3);
```

Exponential noise

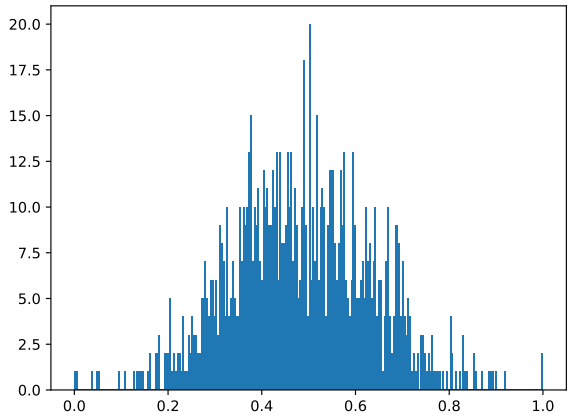
```
a=1;
R4 = -1/a * np.log(1-np.random.rand(32, 32));
R4 = hist_stretch (R4);
```

Figure 8.3: Histograms of the noise images generated with 32×32 pixels.

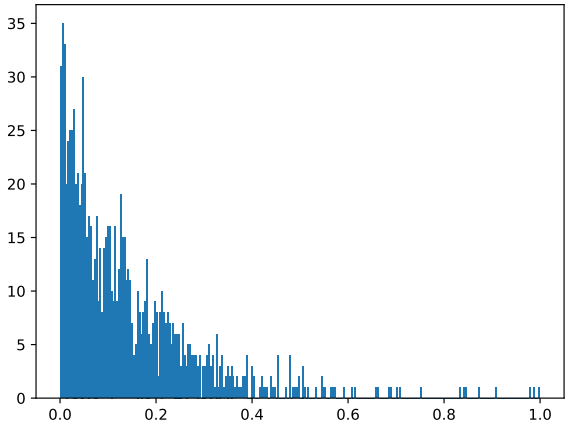
(a) Uniform noise.



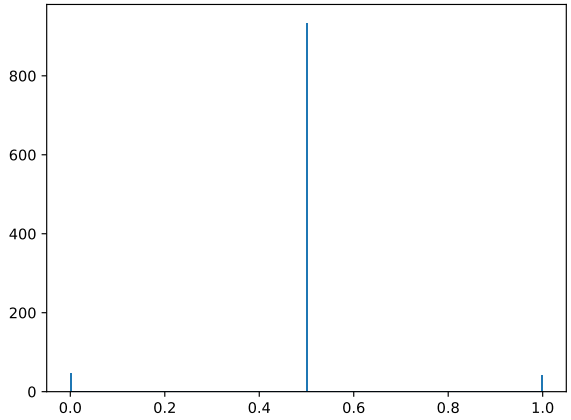
(b) Gaussian noise.



(c) Exponential noise.



(d) Salt and pepper noise.



8.4.2. Noise estimation

In order to estimate the noise, a ROI of visually constant gray level is chosen, and its histogram is displayed. This is simulated by the following code, the result is displayed in Fig.8.4:

```
# noise estimation
fig = plt.figure()
A = imageio.imread('jambe.png');
roi = A[160:200, 200:240];
plt.hist(roi.flatten(),255);
fig.savefig("histo_roi_leg.pdf", bbox_inches='tight');
```

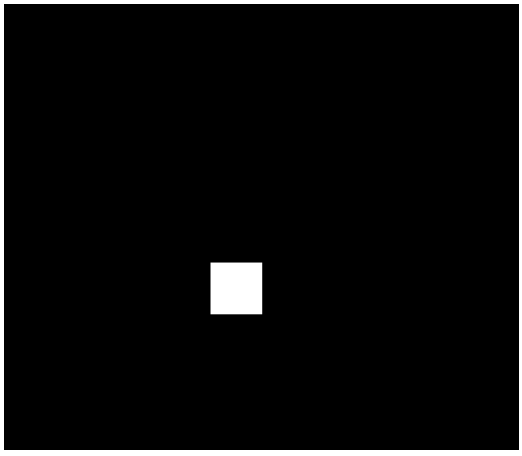
Then, some noise is added to the image, and the histogram of the ROI is displayed. The results are observed in Figs.8.5 and 8.6.

```
# add exponential noise to image
nx, ny = A.shape;
expnoise = -1/.5 * np.log(1-np.random.rand(nx, ny));
expnoise = expnoise / np.max(expnoise);
B = A + 255*expnoise;
B = hist_stretch (B);
fig = plt.figure ()
plt.imshow(B, cmap='gray');
imageio.imwrite("leg_exponential.png", B);
#plt.show()
```

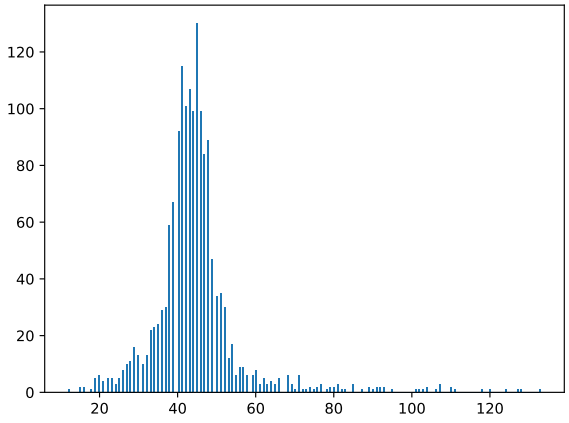
```
# add gaussian noise to image
nx, ny = A.shape;
gaussnoise = 50*np.random.rand(nx, ny);
B = A + gaussnoise;
B = hist_stretch (B);
fig = plt.figure ()
plt.imshow(B, cmap='gray');
imageio.imwrite("leg_gaussian.png", B);
```

Figure 8.4: Histogram of the Region of Interest.

(a) ROI.



(b) Histogram.



In the case of exponential and Gaussian noise added to the image, The histograms are displayed in Figs.8.5 and 8.6.

Figure 8.5: Exponential noise.

(a) Addition of exponential noise to the original image of the leg.



(b) Histogram of the ROI.

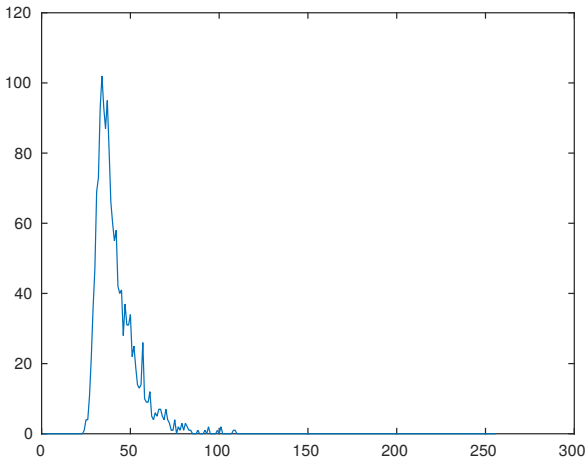
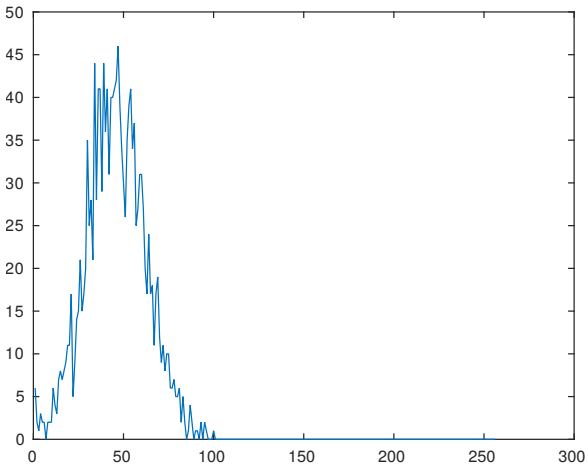


Figure 8.6: Gaussian noise.

(a) Addition of Gaussian noise to the original image of the leg.



(b) Histogram of the ROI.



8.4.3. Image restoration by spatial filtering

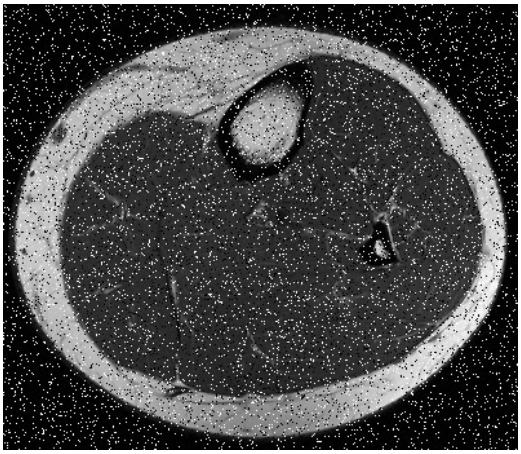
The following code is used to filter the images. The results are displayed in Fig.8.7. Salt and Pepper noise is added to the image.

Figure 8.7: Different filters applied to the noisy image. The median filter is particularly adapted in the case of salt and pepper noise (impulse noise), but still destroy the structures observed in the images.

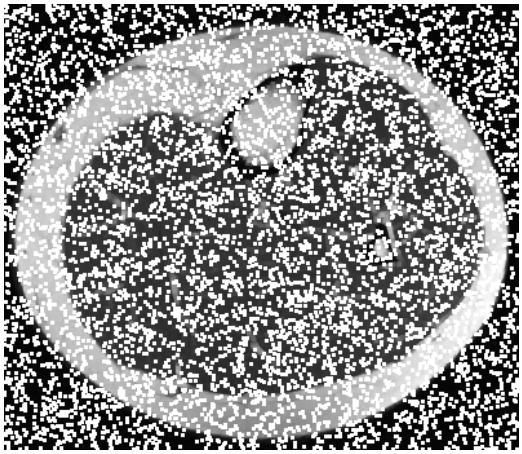
(a) Original image.



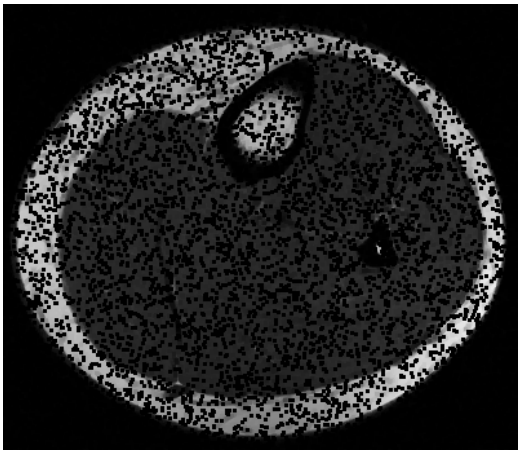
(b) Noisy image (salt and pepper).



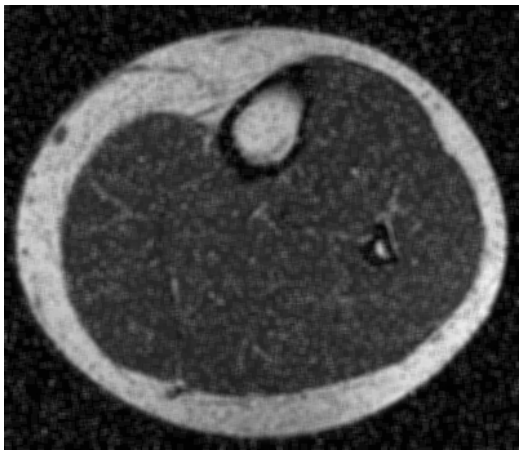
(c) Maximum filter.



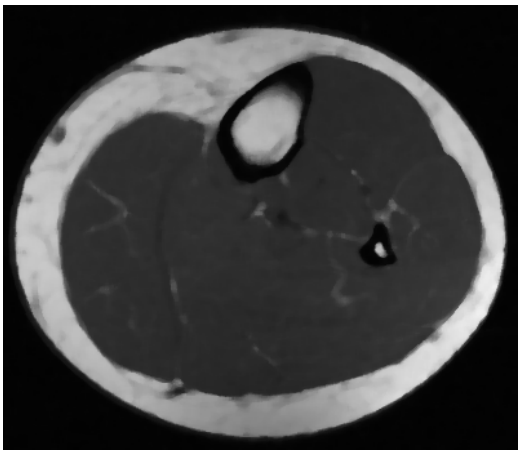
(d) Minimum filter.



(e) Mean filter.



(f) Median filter.





```
a = 0.05; b = 0.05;
spnoise = 0.5 * np.ones((nx,ny));
X = np.random.rand(nx,ny);
B = A.copy();
B[X<=a] = 0;
B[(X>a) & (X <= (a+b))] = 255;
fig = plt.figure()
plt.imshow(B, cmap='gray');
imageio.imwrite("leg_sp.png", B);
#plt.show()
```

Then, different filters are illustrated. Notice that the contours informations are somehow damaged. The median filter corresponds to the best filter in this case.



```
### filtering by convolution
#average filtering
B1 = ndimage.uniform_filter(B,5);
imageio.imwrite("leg_uniform.png", B1);
B2 = ndimage.minimum_filter(B,3);
imageio.imwrite("leg_minimum.png", B2);
B3 = ndimage.maximum_filter(B,3);
imageio.imwrite("leg_maximum.png", B3);
B4 = ndimage.median_filter(B, 7);
imageio.imwrite("leg_median.png", B4);
B5 = amf(B, 7);
plt.imshow(B5, cmap='gray');
imageio.imwrite("leg_amf.png", B5);
```

What can be noticed is that min and max filters are unable to restore the image (opening and closing filters, from the mathematical morphology, could be a solution to explore). The mean filter is a better solution, but an average value is highly modified by an impulse noise. The median filter is the optimal solution is order to suppress the noise, but fine details are lost. An interesting solution is to apply an adaptive median filter.

Adaptive median filter

The following code is a simple implementation of the algorithm previously presented. It has not been optimized in order to have a simple presentation. A more sophisticated version can be found in [11]. The results are illustrated in Fig.8.8 for $S_{max} = 7$.



```
def amf(I, Smax):
    """
    Adaptive median filter
    I: grayscale image
    Smax: maximal size of neighborhood. Limits the effect of median filter
    """
    f = np.copy(I);
    nx, ny = I.shape;
    sizes = np.arange(1, Smax, 2);
    zmin = np.zeros((nx, ny, len(sizes)));
    zmax = np.zeros((nx, ny, len(sizes)));
    zmed = np.zeros((nx, ny, len(sizes)));
    for k,s in enumerate(sizes):
        zmin[:,:,k] = ndimage.minimum_filter(I, s);
        zmax[:,:,k] = ndimage.maximum_filter(I, s);
        zmed[:,:,k] = ndimage.median_filter (I, s);
    isMedImpulse = np.logical_or(zmin==zmed,zmax==zmed);

    for i in range(nx):
        for j in range(ny):
            k = 0;
            while k<len(sizes)-1 and isMedImpulse[i,j,k] :
                k+=1;

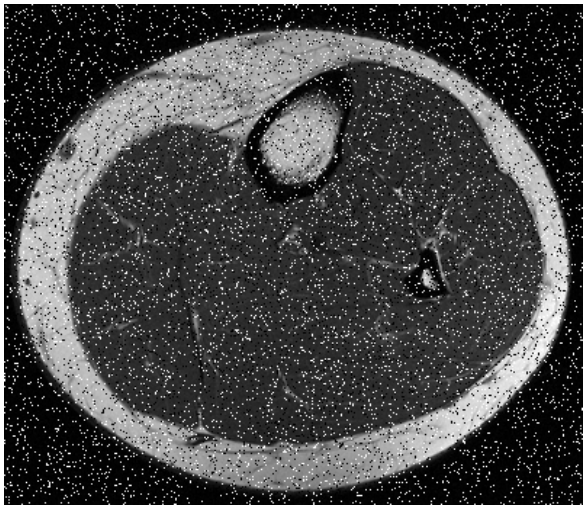
            if I[i,j] == zmin[i,j,k] or I[i,j]==zmax[i,j,k] or k==len(sizes):
                f[i,j] = zmed[i,j,k];
    return f;
```

Figure 8.8: Illustration of the adaptive median filter.

(a) Original image.



(b) Noisy image (salt and pepper).



(c) Median filter of size 7×7.



(d) Adaptive median filter, $S_{max} = 7$.



⋆ 9 Image Filtering using PDEs

This tutorial aims to process images with the help of partial differential equations.

The different processes will be applied on the following MR image Fig. 9.1.



Figure 9.1: MRI image of a human brain.

Notations

The different operators used here are:

$$\vec{A} = \begin{pmatrix} A_x \\ A_y \end{pmatrix} \tag{9.1}$$

$$\operatorname{div} \vec{A} = \frac{\partial A_x}{\partial x} + \frac{\partial A_y}{\partial y} \tag{9.2}$$

$$\vec{\operatorname{grad}}\, u(x,y) = \nabla u = \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial y}{\partial y} \end{pmatrix} \tag{9.3}$$

The Laplacian operator is denoted $\Delta u = \nabla^2 u = \operatorname{div} \overrightarrow{\operatorname{grad}}\, u$. A numerical approximation can be used, but is not recommanded in this tutorial.

9.1. Linear diffusion

The heat equation is defined as follows:

$$\begin{cases} \dfrac{\partial u}{\partial t}(x,y,t) &= \operatorname{div}(\nabla u(x,y,t)) \\ &= \dfrac{\partial^2 u}{\partial x^2}(x,y,t) + \dfrac{\partial^2 u}{\partial y^2}(x,y,t) \\ u(x,y,0) &= f(x,y) \end{cases} \tag{9.4}$$

If $f(x,y)$ is an image (with $(x,y) \in D \subset \mathbb{R}^2$, D is the spatial support), this defines a filtering method that is equivalent to the convolution of f by a Gaussian function [19]. This equation can be solved by a finite difference numerical method.

9.1.1. Numerical scheme

The following notations are employed. N, S, E, W stand for north, south, east and west.

$$\begin{array}{rcl} +\delta^N u & = & \frac{u(x,y+h)-u(x,y)}{h} \\ -\delta^S u & = & \frac{u(x,y-h)-u(x,y)}{h} \\ -\delta^E u & = & \frac{u(x-h,y)-u(x,y)}{h} \\ +\delta^W u & = & \frac{u(x+h,y)-u(x,y)}{h} \end{array} \tag{9.5}$$

Thus, the numerical scheme is (h has value 1):

$$\frac{u^{t+1}-u^t}{\delta t} = \frac{1}{h}\left\{\delta^N u - \delta^S u - \delta^E u + \delta^W u\right\} \tag{9.6}$$

?

1. Code the numerical scheme. It should use two parameters: the number of iterations n, and the step time δt.
2. Filter the original image by varying the parameters of the discrete diffusion process.
3. Comment the filtering results. Compare the results with the Gaussian filter.

Informations

Use the python function `np.roll` in order to efficiently code the discrete different. The convolution operator could also be used but is less efficient in this case.

9.2. Nonlinear diffusion

Image filtering by nonlinear diffusion reduces noise in a controled way. The diffusion coefficient is locally adapted, becoming negligible as object boundaries are approached.
The heat equation is replaced by the following PDE:

$$\left\{\begin{array}{rcl} \frac{\partial u}{\partial t}(x,y,t) & = & \operatorname{div}\left(c(\|\nabla u(x,y,t)\|)\cdot\nabla u(x,y,t)\right) \\ u(x,y,0) & = & f(x,y) \end{array}\right. \tag{9.7}$$

with c the diffusion fonction satisfying the following properties:

- $c(0)=1$,
- $\lim\limits_{s\to+\infty} sc(s)=0$,
- $\forall s>0, c'(s)\leq 0$.

Perona and Malik have proposed 2 diffusion coefficients [28, 5]:

- $c_1 : \exp\left(-\left(\frac{s}{\alpha}\right)^2\right)$,
- $c_2 : \frac{1}{1+\left(\frac{s}{\alpha}\right)^2}$.

9.2.1. Numerical scheme

The numerical scheme is the following:

$$\frac{u^{t+1}-u^t}{\delta t} = \frac{1}{h^2}\Big\{c(|\delta^N u|)\cdot\delta^N u + c(|\delta^S u|)\cdot\delta^S u + c(|\delta^E u|)\cdot\delta^E u + c(|\delta^W u|)\cdot\delta^W u\Big\} \tag{9.8}$$

?

1. Code the numerical scheme
2. Filter the original image by varying the parameters of the discrete nonlinear diffusion process.
3. Comment and compare the filtering results with the previous scheme.

9.3. Degenerate diffusion

The following degenerate PDEs have been shown to be equivalent to the morphological operators of dilation and erosion (using a disk as structuring element, see tutorial on mathematical morphology):

$$\begin{cases} \frac{\partial u}{\partial t}(x,y,t) &= +\|\nabla u(x,y,t)\| \\ u(x,y,0) &= f(x,y) \end{cases} \tag{9.9}$$

$$\begin{cases} \frac{\partial u}{\partial t}(x,y,t) &= -\|\nabla u(x,y,t)\| \\ u(x,y,0) &= f(x,y) \end{cases} \tag{9.10}$$

9.3.1. Numerical schemes

The previous numerical scheme is easy to find, but the results present large differences with the dilation and erosion operators (shocks due to discontinuities in the original image). This is why the following numerical scheme is preferred [23]:

For the erosion:

$$\frac{u^{t+1}-u^t}{\delta t} = -\frac{1}{h^2}\sqrt{max^2(0,-\delta^N u)+min^2(0,-\delta^S u)\ldots + max^2(0,-\delta^W u)+min^2(0,-\delta^E u)} \tag{9.11}$$

For the dilation:

$$\frac{u^{t+1}-u^t}{\delta t} = \frac{1}{h^2}\sqrt{min^2(0,-\delta^N u)+max^2(0,-\delta^S u)\ldots + min^2(0,-\delta^W u)+max^2(0,-\delta^E u)} \tag{9.12}$$

?

1. Code the numerical scheme.
2. Filter the original image while varying the parameters of the discrete degenerate diffusion process.
3. Comment the morphological filtering results and compare the numerical scheme to the approach with operational windows.



9.4. Python correction





```
import numpy as np
from scipy import ndimage
import matplotlib.pyplot as plt
```

9.4.1. Linear diffusion

The linear diffusion filter is equivalent to the classical gaussian filter. The gradients are decomposed into the 4 directions so that the next developments (non linear diffusion filters) are made easy.



```
def linearDiffusion(I, nbIter, dt):
    # linear diffusion  # I: image  # nbIter: number of iterations  # dt: step time
    hW = np.array([[1, -1, 0]]);
    hE = np.array([[0, -1,1]]) ;
    hN = np.transpose(hW);
    hS = np.transpose(hE);
    Z = I.copy();
    for i in range(nbIter):
        gW = ndimage.convolve(Z, hW, mode='constant');
        gE = ndimage.convolve(Z, hE, mode='constant');
        gN = ndimage.convolve(Z, hN, mode='constant');
        gS = ndimage.convolve(Z, hS, mode='constant');
        Z = Z + dt*(gW+gE+gN+gS);
    return Z
```

Notice that this first function uses a convolution, which is correct but not efficient. Another proposition is to use the np. roll function (4 times faster than the version using the convolution).



```
def linearDiffusion2(I, nbIter, dt):
    Z = I.copy()
    for i in range(nbIter):
        gW = np.roll(Z, 1, axis=1) - Z
        gE = np.roll(Z, -1, axis=1) - Z
        gN = np.roll(Z, 1, axis=0) - Z
        gS = np.roll(Z, -1, axis=0) - Z
        Z = Z + dt*(gW+gE+gN+gS)
    return Z
```

The code to filter images is presented as follows. Results are in Fig. 9.2.



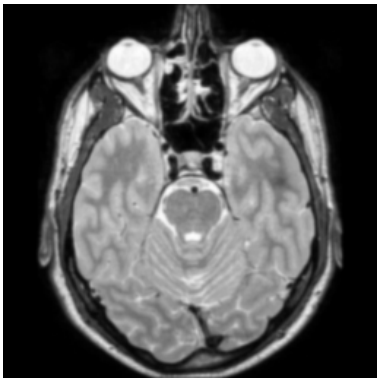
```
I = imageio.imread("cerveau.png")/255.
F = linearDiffusion(I, 50, .05)
F2 = linearDiffusion(I, 200, .05)
plt.subplot(1,3,1)
plt.imshow(I, 'gray')
plt.subplot(1,3,2)
plt.imshow(F, 'gray')
plt.subplot(1,3,3)
plt.imshow(F2, 'gray')
```

Figure 9.2: Linear diffusion filter. Contours are not preserved: this is equivalent to a Gaussian filter.

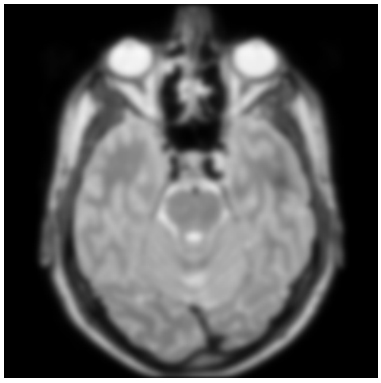
(a) Original image.



(b) Linear diffusion filter, with $dt = 0.05$ and 10 iterations.



(c) Linear diffusion filter, with $dt = 0.05$ and 50 iterations.



9.4.2. Nonlinear diffusion

The nonlinear diffusion is an adaptation of the diffusion to the informations contained in the images. These informations are high frequency components. The following diffusion coefficient is proposed by Perona and Malik.

```python
def c(I, alpha):
    # diffusion coefficient
    # I: image
    # alpha: diffusion parameter
    return np.exp(-(I/alpha)**2);
```

The gradients are evaluated in the 4 directions, a specific coefficient is thus applied to each one.

```python
def nonlinearDiffusion(I, nbIter, alpha, dt):
    # optimized version of nonlinear diffusion, using np.roll
    Z = I.copy()
    for i in range(nbIter):
        gW = np.roll(Z, 1, axis=1) - Z
        gE = np.roll(Z, -1, axis=1) - Z
        gN = np.roll(Z, 1, axis=0) - Z
        gS = np.roll(Z, -1, axis=0) - Z
        Z = Z + dt*(c(np.abs(gW), alpha)*gW + c(np.abs(gE), alpha)*gE
                    + c(np.abs(gN), alpha)*gN + c(np.abs(gS), alpha)*gS)
    return Z
```

The script to filter the images, for $\alpha = 0.1$, d$t = 0.05$ and for 10 and 50 iterations, is the following. Results are presented in Fig. 9.3.

```
alpha = 0.1
dt = .05
I = imageio.imread("cerveau.png")/255.

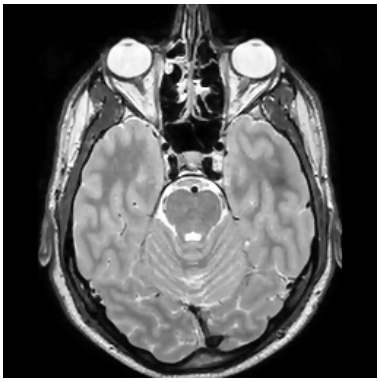
F = nonlinearDiffusion(I, 10, alpha, dt)
F2= nonlinearDiffusion(I, 50, alpha, dt)
```

Figure 9.3: Nonlinear diffusion filter. Contours are preserved.

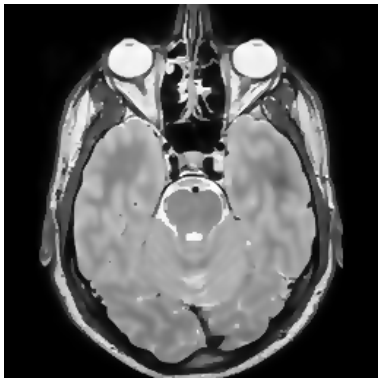
(a) Original image.



(b) Non linear diffusion filter, with $\alpha = 0.1$, $dt = 0.05$ and 10 iterations.



(c) Non linear diffusion filter, with $\alpha = 0.1$, $dt = 0.05$ and 50 iterations.



9.4.3. Degenerate diffusion

Erosion and dilation can theoretically be coded with this numerical scheme. However, some numerical problems appear in the first version, which are corrected in the second version.

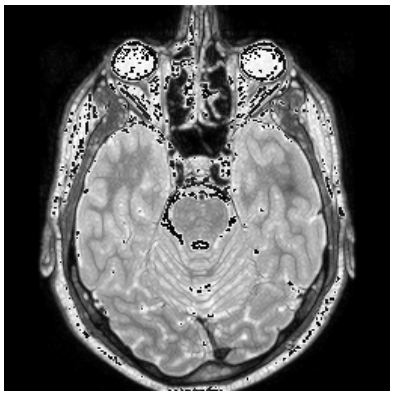
First version

This first version is the direct transcription of the numerical scheme. As observed in Fig.9.4, shocks (peaks) appear after a few iterations.

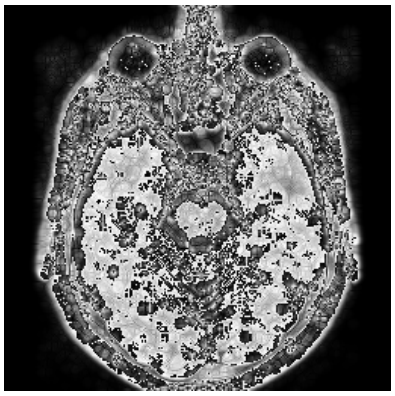
```
def degenerateDiffusion1(image, nbIter, dt):
    # degenerate diffusion
    # I: image
    # nbIter: number of iterations
    # dt: step time
    h = np.array([[-1,0,1]])
    ht= np.transpose(h)
    Zdilation = image.copy()
    Zerosion = image.copy()
    for i in range(nbIter):
        gH = ndimage.convolve(Zdilation, h, mode='constant')
        gV = ndimage.convolve(Zdilation, ht, mode='constant')
        jH = ndimage.convolve(Zerosion, h, mode='constant')
        jV = ndimage.convolve(Zerosion, ht, mode='constant')
        Zdilation = Zdilation + dt*np.sqrt(gV**2 + gH**2)
        Zerosion = Zerosion + dt*np.sqrt(jV**2 + jH**2)
    return (Zdilation, Zerosion)
```

Figure 9.4: Mathematical morphology operations by diffusion. This illustrate the unstability of this first numerical scheme.

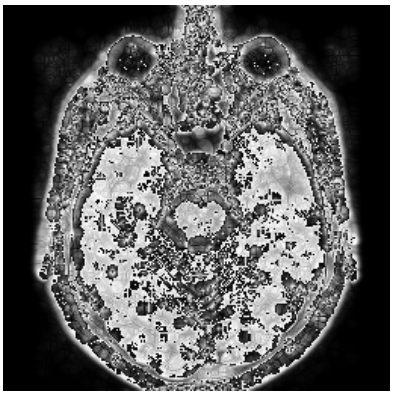
(a) Dilation by diffusion, for $dt = 0.05$ and $nbIter = 10$.



(b) Dilation by diffusion, for $dt = 0.05$ and $nbIter = 50$.



(c) Erosion by diffusion, for $dt = 0.05$ and $nbIter = 50$.



Correction of the unstability

Another scheme, more numerically stable, can be preferred to the previous one. Results are presented in Fig. 9.5.

```python
def degenerateDiffusion3(image, nbIter, dt):
    Zdilation = image.copy()
    Zerosion = image.copy()
    for i in range(nbIter):
        gW = np.roll(Zdilation, 1, axis=1) - Zdilation
        gE = np.roll(Zdilation, -1, axis=1) - Zdilation
        gN = np.roll(Zdilation, 1, axis=0) - Zdilation
        gS = np.roll(Zdilation, -1, axis=0) - Zdilation

        jW = np.roll(Zerosion, 1, axis=1) - Zerosion
        jE = np.roll(Zerosion, -1, axis=1) - Zerosion
        jN = np.roll(Zerosion, 1, axis=0) - Zerosion
        jS = np.roll(Zerosion, -1, axis=0) - Zerosion

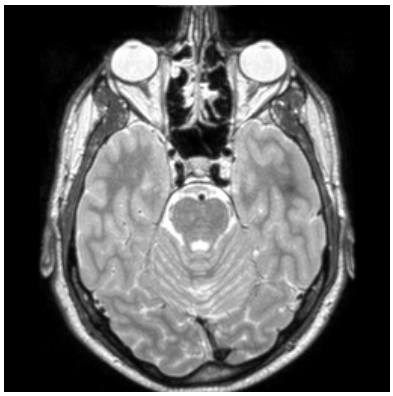
        g = np.sqrt(np.minimum(0, -gW)**2 + np.maximum(0, gE)**
                    2 + np.minimum(0, -gN)**2 + np.maximum(0, gS)**2)
        j = np.sqrt(np.maximum(0, -jW)**2 + np.minimum(0, jE)**
                    2 + np.maximum(0, -jN)**2 + np.minimum(0, jS)**2)

        Zdilation = Zdilation + dt * g
        Zerosion = Zerosion - dt * j

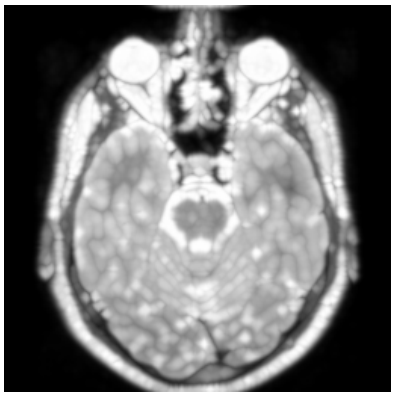
    return Zdilation, Zerosion
```

Figure 9.5: Mathematical morphology operations by diffusion, sophisticated version.

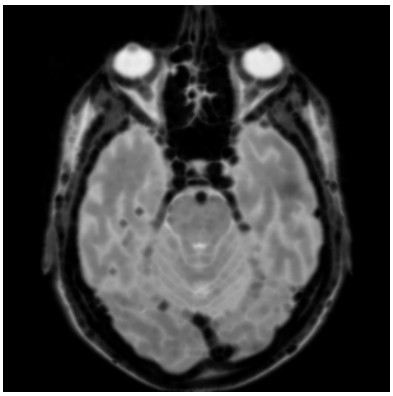
(a) Dilation by diffusion, for $dt = 0.05$ and nbIter=10.



(b) Dilation by diffusion, for $dt = 0.05$ and nbIter=50.



(c) Erosion by diffusion, for $dt = 0.05$ and nbIter=50.



10 Image Restoration: deconvolution

This tutorial aims to study and test different image restoration methods for blurred and noisy images. Altough the different software tools contain functions for each filter the objective is to exhaustively code these methods to fully understand their principles. The reader can refer to [37] for more informations on the algorithms.
Some of the images are based on observations made with the NASA/ESA Hubble Space Telescope, and obtained from the Hubble Legacy Archive, which is a collaboration between the Space Telescope Science Institute (STScI/NASA), the Space Telescope European Coordinating Facility (ST-ECF/ESA) and the Canadian Astronomy Data Centre (CADC/NRC/CSA). You may access to these resources at https://hla.stsci.edu/

The different processes will be applied on the following images.

10.1. Damage modeling

A damage process can be modeled by both a damage function D and an additive noise n, acting on an input image f for producing the damaged image g.

$$g = D(f) + n \tag{10.1}$$

Knowing g, the objective of restoration is to get an estimate $\hat{f}$ (the restored image) of the original image f. If D is a linear process, spatially invariant, then the equation can be simplified as:

$$g = h * f + n \tag{10.2}$$

where h is the spatial representation of the damage function (called the Point Spread Function - PSF), and $*$ denotes the convolution operation. In general, the more knowledge you have about the function H and the noise n, the closer $\hat{f}$ is to f.

This equation can be written in the frequency domain:

$$G = H.F + N \tag{10.3}$$

where the letters in uppercase are the Fourier transforms of the corresponding terms in the eq. 10.2.

?

- Generate the 'chessboard' image.
- Generate a PSF corresponding to a blur (motion or isotropic).
- Create a damaged image (a circular option is used, because the FFT implies that functions are periodical) and add a Gaussian noise.
- Visualize the different images.

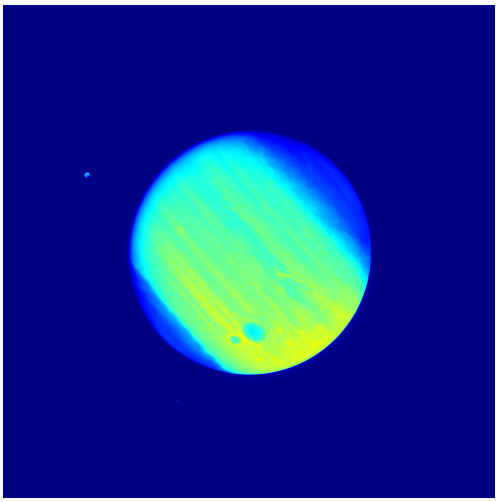
The following function generates in python a checkerboard.

```python
def checkerboard2(s=8):
    return np.kron([[1, 0] * 4, [0, 1] * 4] * 4, np.ones((s, s)))
```

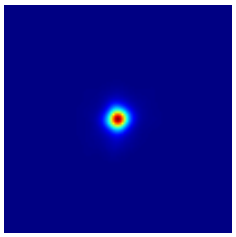
The recommanded convolution function is scipy.signal.convolve2d.

Figure 10.1: Images that can be used for testing the different restoration algorithms. Intensity levels are represented in false colors.

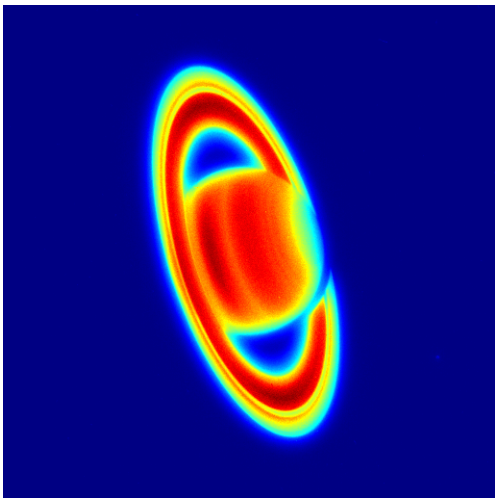
(a) Jupiter, from Hubble Space Telescope, ads/Sa.HST#ic3g01qlq.



(b) PSF for Jupiter image



(c) Saturn, HST.



(d) PSF for Saturn image

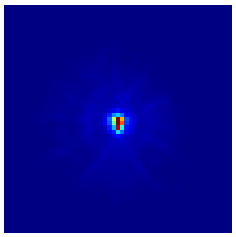
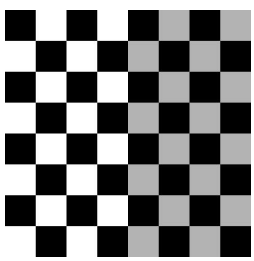


Figure 10.2: Chessboard image.



A numpy file psf_motion.npy stores the PSF of a motion blur.

10.2. Simple case: no noise

The objective is now to restore the damaged image.

For the first steps, no noise will be added. The degradation functions is $g = h * f$. Knowing H (Fourier Transform of the psf h) and ignoring the noise in the damage model, a simple estimate of the initial image can be done by the inverse filtering:

$$f = FT^{-1}\left(\frac{G}{H}\right) \tag{10.4}$$

where FT^{-1} denotes the inverse Fourier transform. The Fourier transform of the PSF is also called the OTF (Optical Transfer Function). For numerical considerations, H should be the same size as G, and thus you have to pad it with zeros.

A function psf2otf is provided to avoid uninteresting developments. The code is given, that works for our applications.



```
def psf2otf(psf, s):
    """
    Get OTF (Optical Transfer Function) from PSF (Point Spread Function)
    OTF is basically the Fourier Transform of the PSF, centered
    psf: PSF
    s: shape of the result, zero-padding is used to center is Fourier Transform
    """

    # computes padding values
    sh = np.array(psf.shape)
    s = np.array(s)
    pad = s - sh

    # centers psf
    h_centered = np.pad(psf, ((pad[0]//2+1, pad[0]-pad[0]//2-1), (pad[1]//2+1, pad[1]-
        ↪ pad[1]//2-1)), mode='constant')

    plt.imshow(h_centered)
    plt.title("psf centered")
    plt.show()
    # Fourier transform (aka OTF) of the psf
    h_centered = np.fft.fftshift(h_centered)
    H = np.fft.fft2(h_centered, s)

    # Keep only real values (simple approximation)
    H = np.real(H)
    return H
```



?

- Try the inverse filter in the Fourier space. Why is this filter not working?
- Try to prevent division by 0, with $f = FT^{-1}\left(\frac{G}{H+\alpha}\right)$, α being a small constant value (e.g. $\alpha = 0.1$).

10.3. Wiener filter

The Wiener filter is an optimal filter that minimizes the following error:

$$e^2 = E\{(f - \hat{f})^2\}$$

where E denotes the expected (mean) value, f is the original image and $\hat{f}$ is the restored image.

The solution to this expression, within the frequency domain, is given by:

$$\hat{F} = \underbrace{\left(\frac{1}{H}\frac{|H|^2}{|H|^2 + R}\right)}_{H_w} G$$

The value of R can be arbitrarily fixed. Another way is to define $R = S_n/S_f$, where S_n and S_f denote the power spectrum of the noise n and the original image f, i.e. $|N|^2$ and $|F|^2$ (matrices), respectively. This ratio can be defined globally (as the constant) or locally (i.e. it is computed for each pixel). If the ratio S_n/S_f (function) is non null, it can be passed to the Wiener filtering process.

10.3.1. Simple case: no noise

In the noiseless case, the Wiener filter reduces to the inverse filter:

$$H_w = \begin{cases} \dfrac{1}{H(u,v)} & \text{if } |H(u,v)| \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

?

Try a Wiener filter for the noiseless case.

10.3.2. Noisy images

Generally, the ratio S_n/S_f used in the Wiener filter is replaced by a constant value equal to the ratio of the mean power spectrum:

$$R = \frac{\frac{1}{PQ} \cdot \sum_u \sum_v S_n(u,v)}{\frac{1}{PQ} \cdot \sum_u \sum_v S_f(u,v)} \tag{10.5}$$

where PQ denotes the matrix size (the number of elements).

?

1. Calculate the constant ratio R and code the Wiener filter .
2. Test the different algorithms on the image 'brain' damaged by a blur and an additive noise.

10.4. Iterative filters for noisy images

10.4.1. Van Cittert iterative filter

The VanCittert iterative filter is based on the following iterative relation :

$$f_{k+1} = f_k + \beta(g - h * f_k)$$

with $f_0 = g$, the damaged image.



Code and test on different cases a function that performs the VanCittert restoration filter.

It should have the following prototype:



```
function [ fr ] = vca( g, psf, n_iter )
% Van Cittert iteration algorithm for deconvolution
%
% g: degraded image
% psf: point spread function
% n_iter: number of iterations
```



```
def vca(g, psf, n_iter, beta=.01):
    """
    Van Cittert iterative filter
    g: noisy image
    psf: point spread function
    n_iter: number of iterations
    beta: Jansson parameter
    """
```

10.4.2. Lucy-Richardson filtering

We are now going to use the nonlinear restoration filtering of Lucy-Richardson that is based on a maximum likelihood formulation in which the image is modeled by Poisson statistics. This optimization leads to the solution if the following iterative equation converges:

$$f_{k+1}(x,y) = f_k(x,y) \cdot \left(h(-x,-y) * \frac{g(x,y)}{h(x,y) * f_k(x,y)} \right) \tag{10.6}$$



Code and test your own function for the Lucy-Richardson filtering process. It should have the same prototype as the VanCittert function. Remember that $\cdot$ is the classical multiplication (point to point) and that $*$ is the convolution operation. You then have the choice to make a direct call to the convolution function, or to perform this operation in the Fourier space.

10.5. Blind deconvolution: restoration with no a priori

If the PSF is unknown, one of the main difficulties in image restoration is to get an accurate estimate of this PSF in order to use it in the deconvolution algorithms. These methods are called blind deconvolution methods. Generally, the PSF is estimated in an iterative way from an initial PSF.

1. Generate a damaged image of 'chessboard' with a Gaussian blur and an additive Gaussian noise.
2. Restore from a blind deconvolution the damaged image and compare the results with the previous ones.

Informations

Use the function skimage. restoration .unsupervised_wiener for example.

10.6. Astronomy images

A classical file format used in astronomy is the FITS format.

With python, fits images can be manipulated with the astropy module. You can use the following code to display an image.

```
hdu_list = fits.open('ic3g01qlq_drz.fits');
image = hdu_list[1].data;
image = np.nan_to_num(image);
plt.imshow(image, cmap='gray')
plt.show()
```

10.7. Python correction

```
from astropy.io import fits
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal
import progressbar
```

10.7.1. Damage modeling

There is no built-in function to generate the checkerboard image. A simple code can be used.

```
def checkerboard(s=8):
    return np.kron([[1, 0] * 4, [0, 1] * 4] * 4, np.ones((s, s)))
```

The psf can be generated as a Gaussian psf, or a motion psf. In this tutorial, the motion psf is loaded from a file that comes from the MATLAB® function fspecial.

```
C = checkerboard();
psf = np.load('psf_motion.npy');
```

In order to add Gaussian noise, the following function is used:

```
def addGaussianNoise(I, sigma=1000):
    """Adds Gaussian noise to an image
    I: original image
    sigma: signal/noise ratio
    """
    I2 = I.copy();
    m=np.min(I);
    M=np.max(I);
    N = (M-m)/sigma*np.random.randn(I.shape[0], I.shape[1]);
    I2 = I2 + N;
    I2[I2>M] = M;
    I2[I2<m]=m;
    return I2, I-I2;
```

The image is blurred and noise is then added.

```
Cb = signal.convolve2d(C, psf, boundary='wrap', mode='same');
Cbn, noise = addGaussianNoise(Cb);
plt.imshow(Cbn, cmap='gray')
plt.title('Motion blur + Gaussian noise on checkerboard')
plt.show();
```

Optical Transfer Function: OTF

The OTF is the centered Fourier Transform of the PSF (Point Spread Function). The following function is used to get the OTF from the PSF:

```
def psf2otf(psf, s):
    """
    Get OTF (Optical Transfer Function) from PSF (Point Spread Function)
    OTF is basically the Fourier Transform of the PSF, centered
    psf: PSF
    s: shape of the result, zero-padding is used to center is Fourier Transform
    """
    sh = psf.shape;
    sh = np.array(sh);
    s = np.array(s);
    pad = s - sh;
    h = np.pad(psf, ((0, pad[0]), (0, pad[1])), mode='constant');
    shift = (int(pad[0]/2+1), int(pad[1]/2+1));
    h_centered = np.roll(h, tuple(shift), axis =(0,1));
    h_centered = np.fft.fftshift(h_centered);
    H = np.fft.fft2(h_centered, s);
    H = np.real(H);
    return H;
```

10.7.2. Simple case: no noise

For the direct reconstruction, just ensure that there is no division by 0. The different results are illustrated in Fig.10.3.

```
H = psf2otf(psf, C.shape);
G = np.fft.fft2(Cb);
alpha = 0.001;
F = G / (H+alpha);
fr = np.real(np.fft.ifft2(F));
plt.imshow(fr);
plt.title('Noiseless (only motion blur) direct reconstruction')
plt.imsave("cb_reconstruction.png", fr, cmap='gray');
plt.show()
```

In the presence of noise, the reconstructed image is not perfect.

```
H = psf2otf(psf, C.shape);
G = np.fft.fft2(Cbn);
alpha = 0.001;
F = G / (H+alpha);
fr = np.real(np.fft.ifft2(F));
```

The automatic evaluation of the noise parameter is done in the Wiener filter by:



```
SpecPuissNoise=np.abs(np.fft.fft2(noise))**2;
PuissMoyNoise=np.mean(SpecPuissNoise);
SpecPuissImageOrig=np.abs(np.fft.fft2(C))**2;
PuissMoyImageOrig=np.mean(SpecPuissImageOrig);
ratio=PuissMoyNoise/PuissMoyImageOrig;
Hw = np.conjugate(H)/(np.abs(H)**2+ratio);
fr = np.fft.ifft2( Hw * np.fft.fft2(Cbn));
```

10.7.3. Iterative filters

Van-Cittert iterative filter

The parameter (Jansson parameter) controls the precision of convergence, but a small value requires a high number of iterations (and thus a high computation time).



```
def vca(g, psf, n_iter, beta=.01):
    """
    Van Cittert iterative filter
    g: noisy image
    psf: point spread function
    n_iter: number of iterations
    beta: Jansson parameter
    """
    H = psf2otf(psf, g.shape);

    # init iterations
    fr = g.copy();
    bar = progressbar.ProgressBar();
    for iter in bar(range(n_iter)):
        estimation = np.real(np.fft.ifft2(H * np.fft.fft2(fr)));
        fr= fr + beta* ( g - estimation);

    return fr;
```

Richardson-Lucy iterative filter

This algorithm is probably the most famous one in the domain. Remember that the symmetry in the space domain becomes the complex conjugate in the spectral domaine (after Fourier transform).



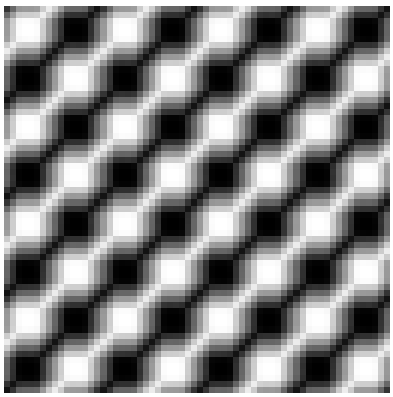
```
def rla(g, psf, n_iter):
    """
    Richardson Lucy algorithm
    g: noisy image
    psf: point spread function
    n_iter: number of iterations
    """
    H = psf2otf(psf, g.shape);
    # init iterations
    fr = g.copy();

    bar = progressbar.ProgressBar()
    for iter in bar(range(n_iter)):
        estimation = np.fft.ifft2(H * np.fft.fft2(fr));
        M = np.real(np.fft.ifft2(np.conjugate(H) * np.fft.fft2(g / estimation)));
        fr = fr * M;
        fr[fr<0] = 0;

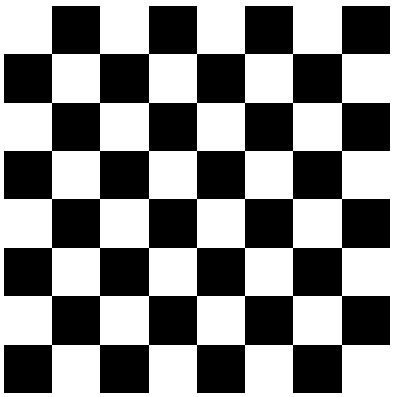
    return fr;
```

Figure 10.3: Different deconvolution methods applied on the synthetic image 'checkerboard'.

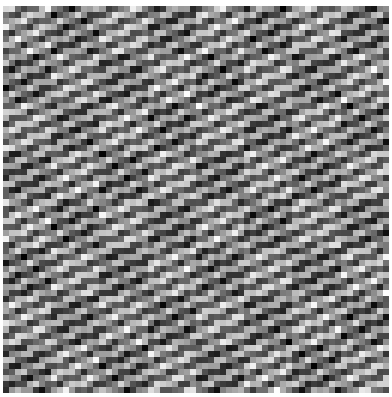
(a) Damaged checkerboard image, with motion blur and Gaussian noise.



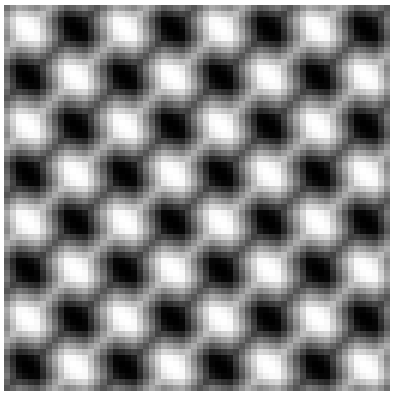
(b) Reconstruction of the checkerboard without noise.



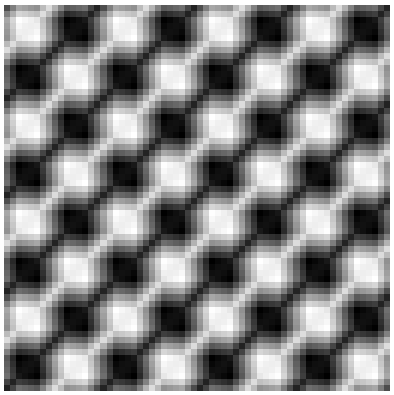
(c) Reconstruction of the checkerboard with noise.



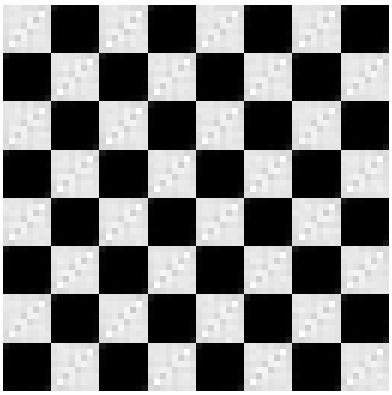
(d) Wiener deconvolution.



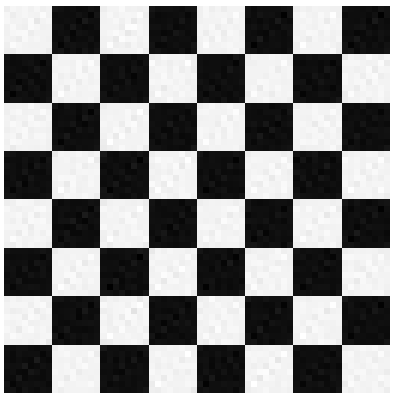
(e) Van-Cittert iterative deconvolution with 100 iterations, and Jansson parameter at 0.01.



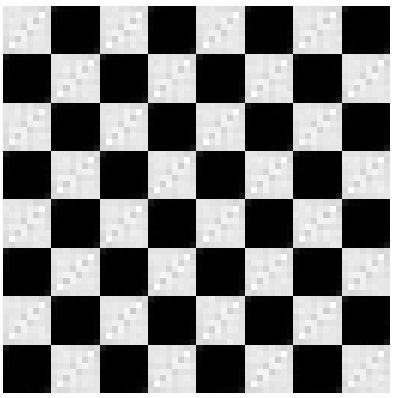
(f) Richardon-Lucy iterative deconvolution with 1000 iterations.



(g) Landweber iterative deconvolution with 1000 iterations and Jansson parameter at 1.



(h) Poisson Maximum A Posteriori iterative deconvolution with 1000 iterations.

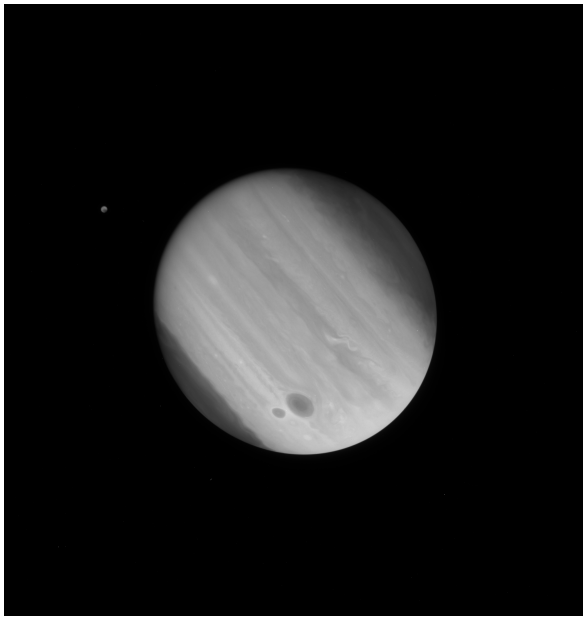


10.7.4. Astronomy images

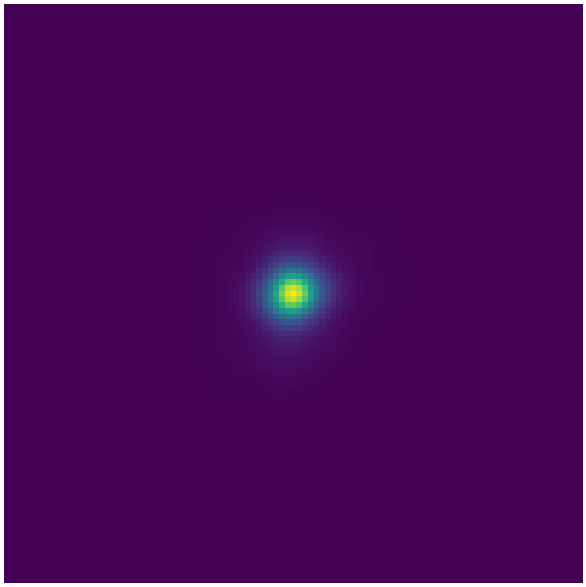
These real images come from the Hubble Space Telescope (HST, see credits), shown in Fig.10.4. The results for the Richardson-Lucy, Van-Cittert and Landweber algorithms are shown in Fig.10.5. The direct deconvolution does not give a correct result.

Figure 10.4: Original images and PSF, from the HST.

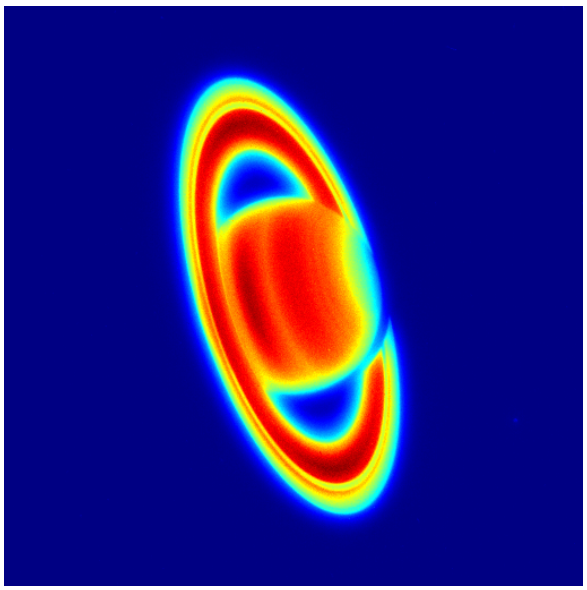
(a) Jupiter.



(b) PSF for the observation of Jupiter.



(c) Saturn.



(d) PSF for the observation of Saturn.

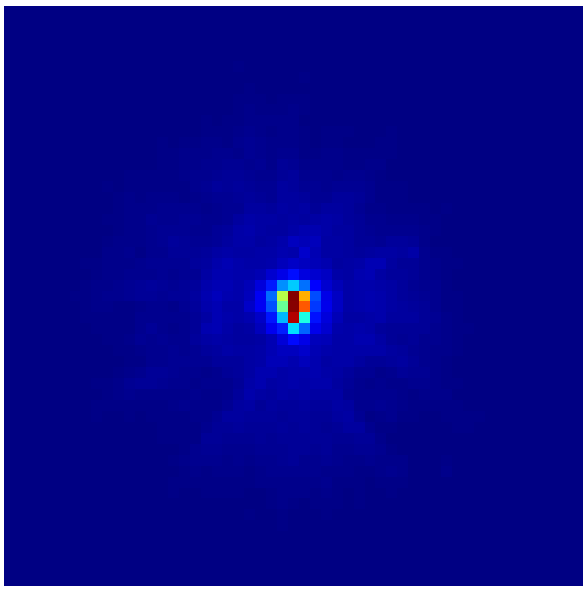
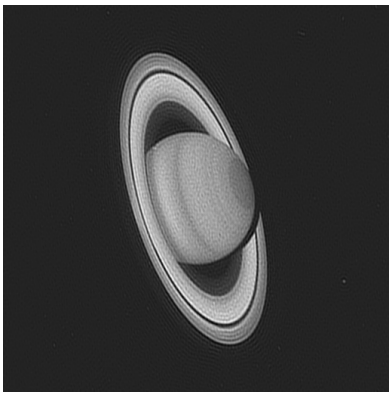


Figure 10.5: Different deconvolution methods applied for the astronomy images.

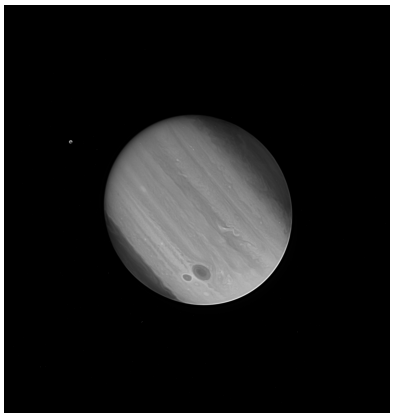
(a) Direct deconvolution.



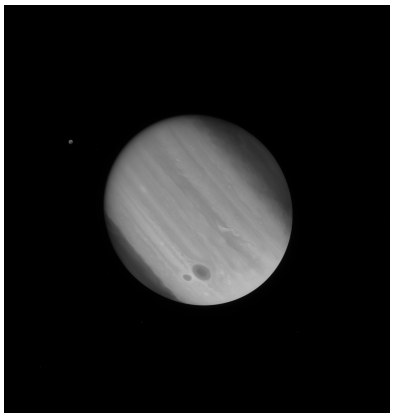
(b) Direct deconvolution.



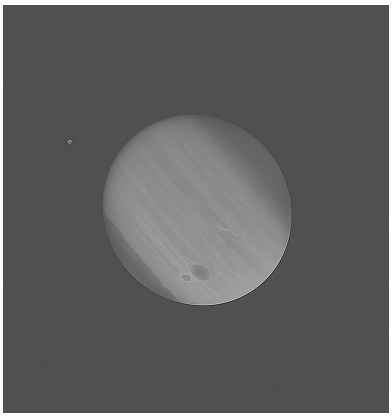
(c) Richardon-Lucy iterative deconvolution with 10 iterations.



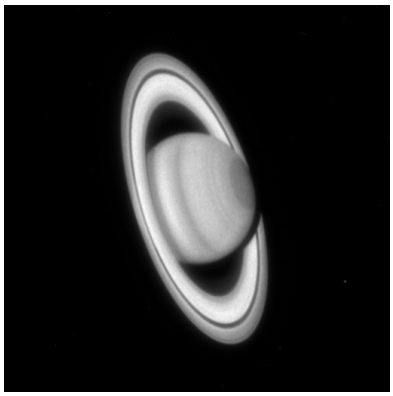
(d) Van-Cittert iterative deconvolution with 10 iterations.



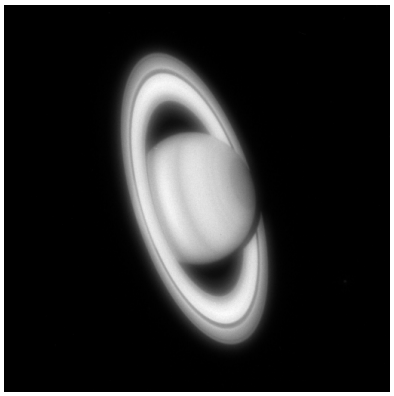
(e) Landweber iterative deconvolution with 200 iterations and Jansson parameter at 1.



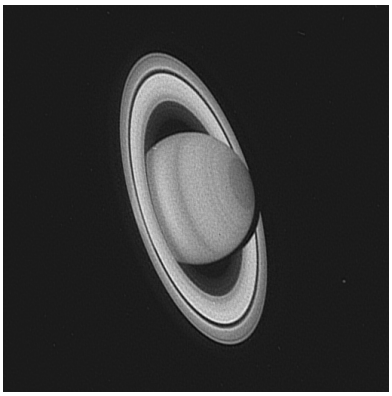
(f) Richardon-Lucy iterative deconvolution with 10 iterations.



(g) Van-Cittert iterative deconvolution with 10 iterations.



(h) Landweber iterative deconvolution with 200 iterations and Jansson parameter at 1.



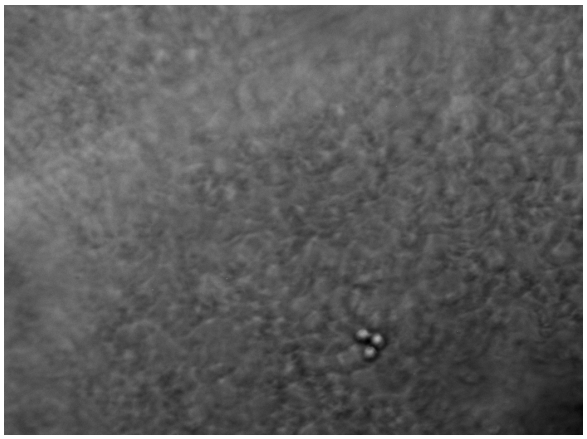
Part IV Image Formation and Reconstruction

⋆ 11 Shape From Focus

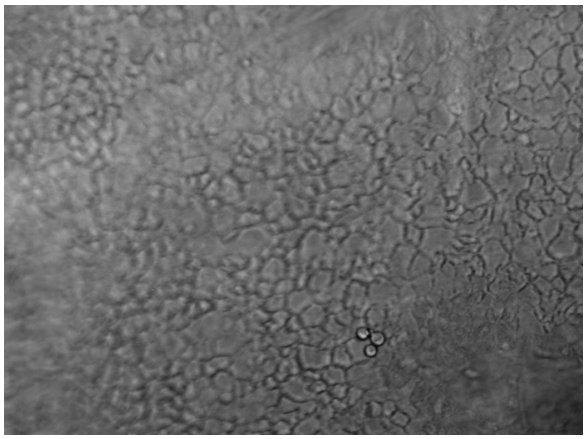
This tutorial introduces the basic shape from focus concepts. The objective is to reconstruct a focused image from a serie (generally a stack) of images with inhomogene focus (see Fig. 11.1).

Figure 11.1: Different images of the stack, from a corneal endothelium in optical microscopy (with 3D microscope).

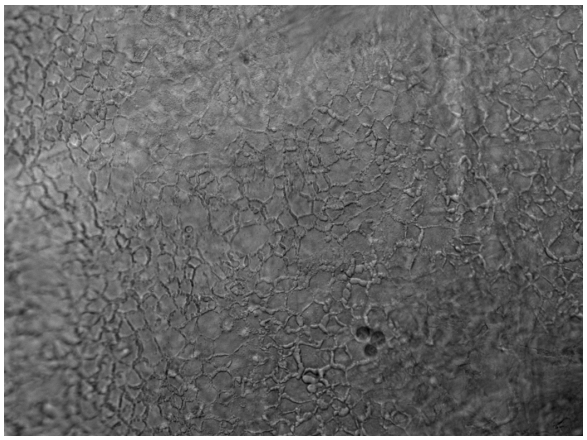
(a) Layer 1



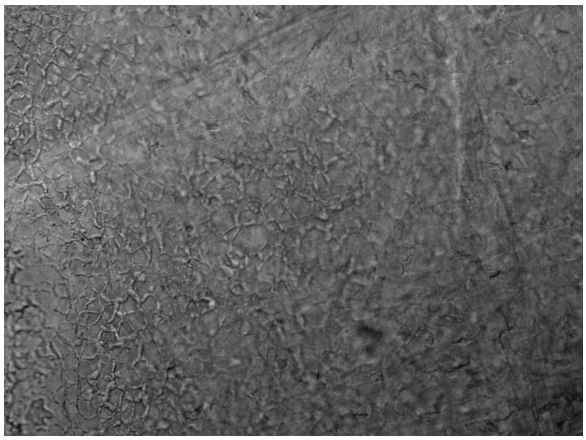
(b) Layer 9



(c) Layer 18



(d) Layer 24



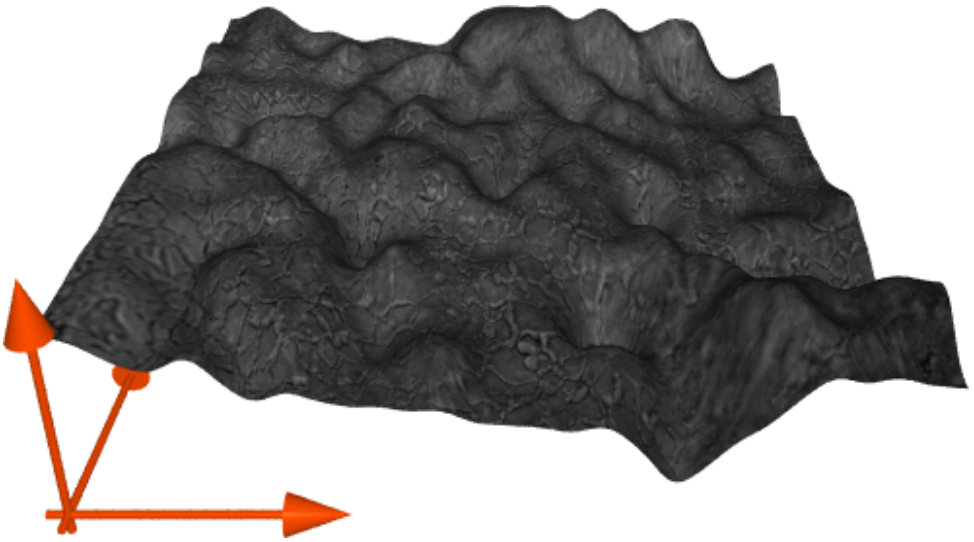
11.1. Introduction to classical methods

An optic system has a limited depth of field. When observing non plane surfaces with a microscope, some parts of the observation may be blurred as well as some others may be correctly focused.

To overcome this problem and reconstruct an all-focused image as well as a surface (see Fig. 11.2), an algorithm will look at every pixel of the images in the stack and select the most focused one, by the way of a focus measure. This tutorial proposes to test some classical focus measures. You can have a look at [7] and [8] to see real applications.

Practically, TIF files can handle stacks of images. Here is a way to open such a file:

Figure 11.2: Reconstruction of the surface and the texture.



```
from skimage import io

I = io.imread('volume.tif');
I = I.astype('float');
```

In the following, each of the proposed methods will compute a focus measure layer by layer, and will maximize this focus measure over the stack to find the most focused layer.

11.1.1. Variance

The measure of focus based on the variance is today the mainly used method [15, 42]. It is based on the computation of the variance in a window ω, with $N = \#\omega$ being the size of the window:

$$F_v = \sum_{\omega} \left(I - \underbrace{\frac{1}{\#\omega} \sum_{\omega} I}_{\text{Mean of I}} \right)^2$$

11.1.2. Sum of Modified Laplacian

One of the first methods had been proposed by [26]. It is based on the second derivatives, specifically the Laplacian operator:

$$\triangle^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

The problem with this operator is that the second derivatives in the x and y dimensions can have opposite signs. One way to overcome this problem is to introduce the modified Laplacian as follows:

$$\triangle^2_M I = \left| \frac{\partial^2 I}{\partial x^2} \right| + \left| \frac{\partial^2 I}{\partial y^2} \right|$$

The discrete approximation of the modified Laplacian is computed as (this can be coded as a convolution):

$$ML(x,y) = |2I(x,y) - I(x-1,y) - I(x+1,y)| + |2I(x,y) - I(x,y-1) - I(x,y+1)|$$

Then, the focus measure based on the modified Laplacian is:

$$F_{ML}(p) = \sum_{(x,y)\in\omega(p)} ML(x,y)$$

To clarify this formula, this means that for each pixel p, the focus measure is the sum of all coefficients ML in a small window ω centered on p. The size of this window is relatively small, a value of 7×7 pixels has been used in the correction.

11.1.3. Tenengrad

In [21], we can find the definition of the tenengrad [43]. Let $S(x, y)$ be the norm of the Sobel gradient of image I.

$$F_t(I) = \sum_{\omega} S^2$$

Notice that the original definition requires a threshold value that requires heuristic choices, which is out of the topic of this tutorial.

11.1.4. Variance of Tenengrad

We define the variance of Tenengrad measure of focus by:

$$F_{vt}(I) = F_v(S)$$

.

11.2. Texture and surface reconstruction

?

A set of images (Vickers indentation test [7], and a human corneal endothelium [8]) are proposed. For all the detailed methods:

- reconstruct the surface and the texture of the image (an image focused on its all field of view).

11.2.1. Open question

?

Several methods have been implemented in order to perform the 3D surface/texture reconstruction. Propose a numerical measure that could compare the results and measure the efficiency of these methods? What is the best method?



11.3. Python correction





```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from skimage import io
from scipy import ndimage
from matplotlib import cm
```

11.3.1. Main function

This code is used as the main function to perform shape-from-focus reconstruction.



```
volumes = ['cornee', 'vickers'];
for v in volumes:
    I = io.imread('volume_'+v+'.tif');
    I = I.astype('float');
    F = np.zeros(I.shape)
    N = 11;

    myfunctions = [variance, tenengrad, sml];
    for f in myfunctions:
        for i,im in enumerate(I):
            F[i] = f(im, N);

        # Evaluates altitudes and textures
        Z = np.argmax(F, axis=0);
        Z = ndimage.minimum_filter(Z, size=5);
        T = extractTexture(I, Z);
```

The extraction of the texture from the altitudes (indexes) in the stack of images is performed by the following function:



```
def extractTexture(I, Z):
    """
    Extract texture from stack of images I, where Z is the index (altitude)
    I: stack of images, of shape (n, X, Y)
    Z: index of SFF maximum (of shape (X,Y)), values are between 0 and n-1
    returns basically I[Z(i,j), i, j)] for all (i,j)
    """
    m,n = I.shape[1:]
    ii, jj = np.ogrid[:m,:n]
    T = I[Z, ii, jj];
    return T;
```

11.3.2. Sum of Modified Laplacian

Results are illustrated in Fig.11.3.

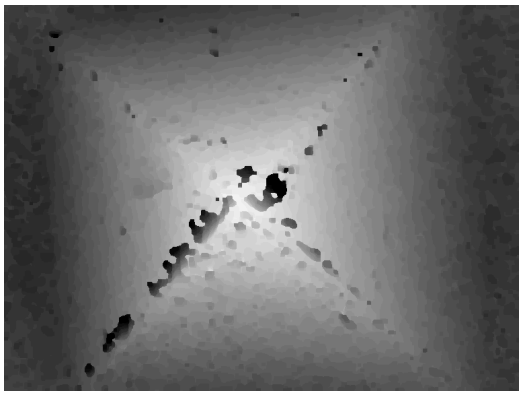
```python
def sml(I, N):
    """
    SFF measure, SUM of modified Laplacian
    I: image
    N: neiborhood size
    returns: SFF measure for each pixel
    """
    h=np.array([[-1, 2, -1]]);
    ML=np.abs(ndimage.convolve(I,h))+np.abs(ndimage.convolve(I,np.transpose(h)));
    S=ndimage.uniform_filter(ML, N);
    return S;
```

Figure 11.3: Texture and altitude reconstruction with the SML method.

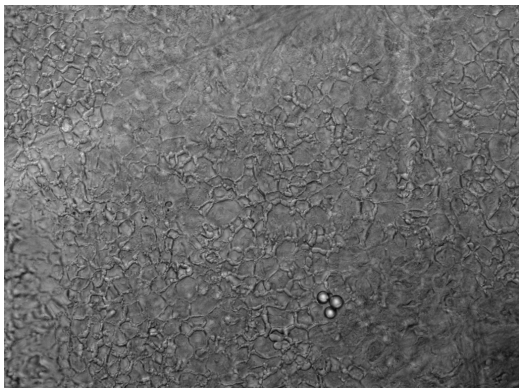
(a) Texture.



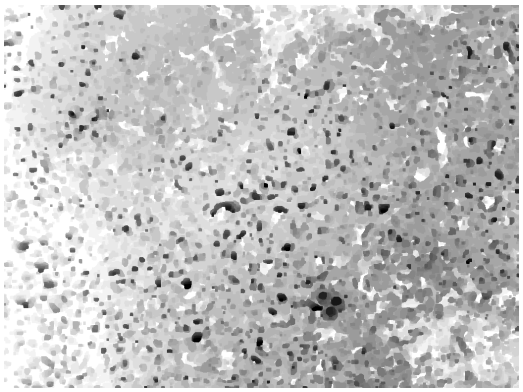
(b) Altitudes.



(c) Texture.



(d) Altitudes.



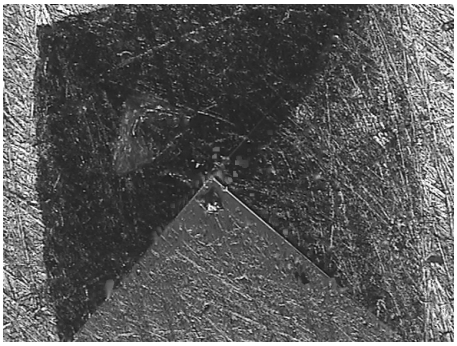
11.3.3. Variance

The focus measure based on the variance i a really simple method that works in most cases, see Fig.11.4.

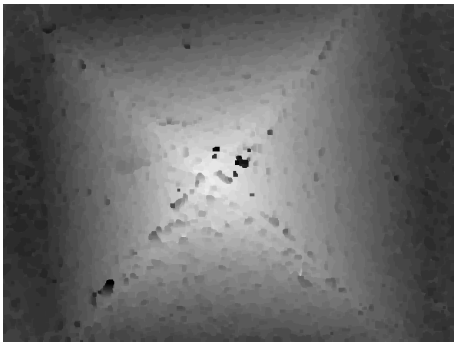
```python
def variance(I, N):
    """
    SFF measure
    I: image
    N: neighborhood size
    returns: SFF measure for each pixel, results is the same shape as I
    """
    M = ndimage.uniform_filter(I, N);
    D2 = (I - M) ** 2;
    V = ndimage.uniform_filter(D2, N);
    return V;
```

Figure 11.4: Texture and altitude reconstruction with the variance method.

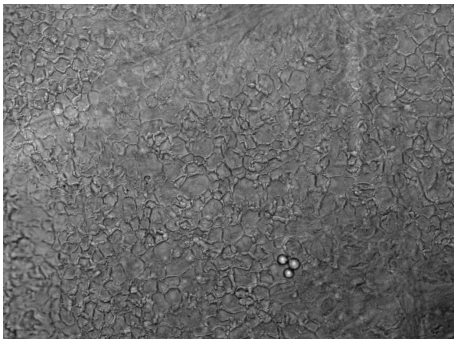
(a) Texture.



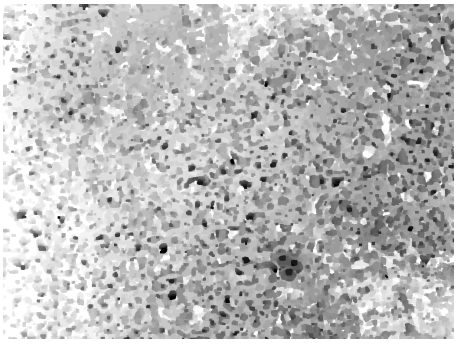
(b) Altitudes.



(c) Texture.



(d) Altitudes.



11.3.4. Tenengrad

The tenengrad method is base on a Sobel filter, see Fig.11.5.

```python
def tenengrad(I, N):
    """
    SFF measure, Tenengrad method
    I: image
    N: neighborhood size
    returns: SFF measure for each pixel, results is the same shape as I
    """
    Sx = ndimage.sobel(I, axis=0);
    Sy = ndimage.sobel(I, axis=1);
    S = np.hypot(Sx, Sy);
    T = ndimage.uniform_filter(S, N);
    return T;
```

11.3.5. Variance of Tenengrad

The variance of Tenengrad is an improvement of the Tenengrad method, see Fig.11.6.

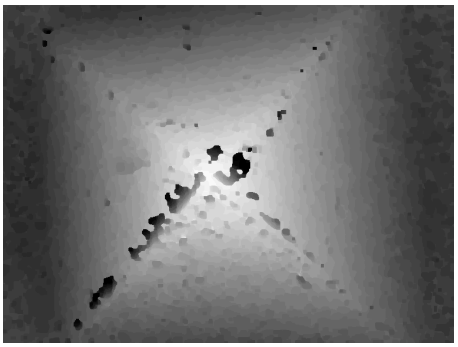
```python
def varianceTenengrad(I, N):
    """
    SFF measure, variance of Tenengrad
    I: image
    N: neiborhood size
    returns: SFF measure for each pixel
    """
    Sx = ndimage.sobel(I, axis=0);
    Sy = ndimage.sobel(I, axis=1);
    S = np.hypot(Sx, Sy);
    vt = variance(S, N);
    return vt;
```

Figure 11.5: Texture and altitude reconstruction with the SML method.

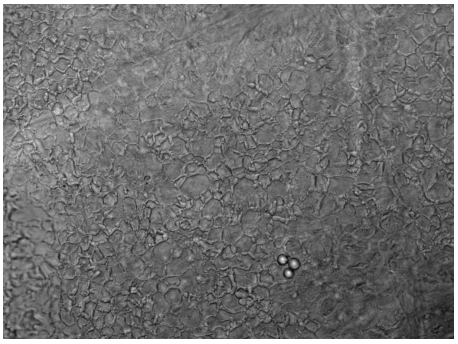
(a) Texture.



(b) Altitudes.



(c) Texture.



(d) Altitudes.

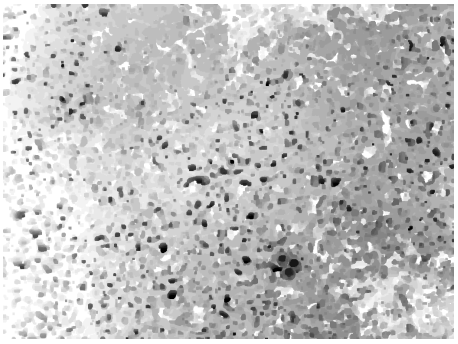
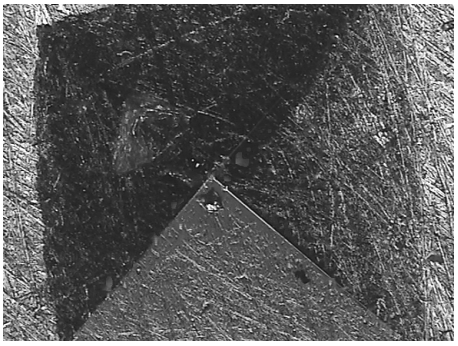
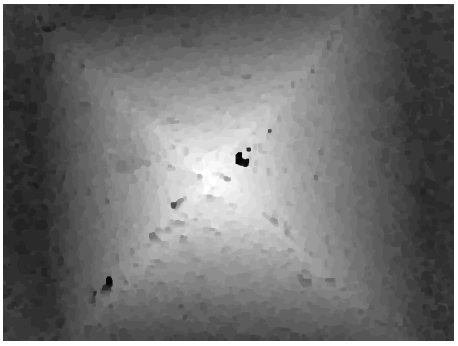


Figure 11.6: Texture and altitude reconstruction with the variance of Tenengrad method.

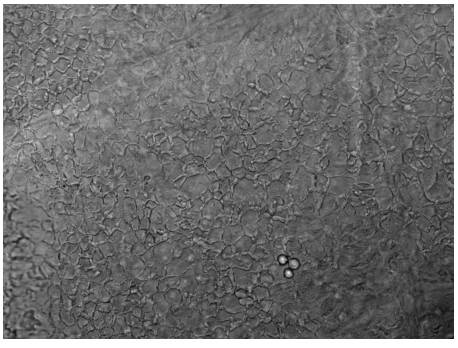
(a) Texture.



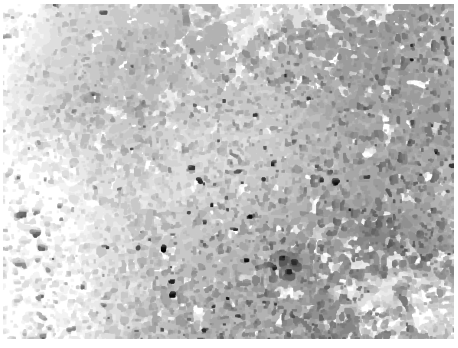
(b) Altitudes.



(c) Texture.



(d) Altitudes.



⋆ 12 Introduction to tomographic reconstruction

12.1. X ray tomography

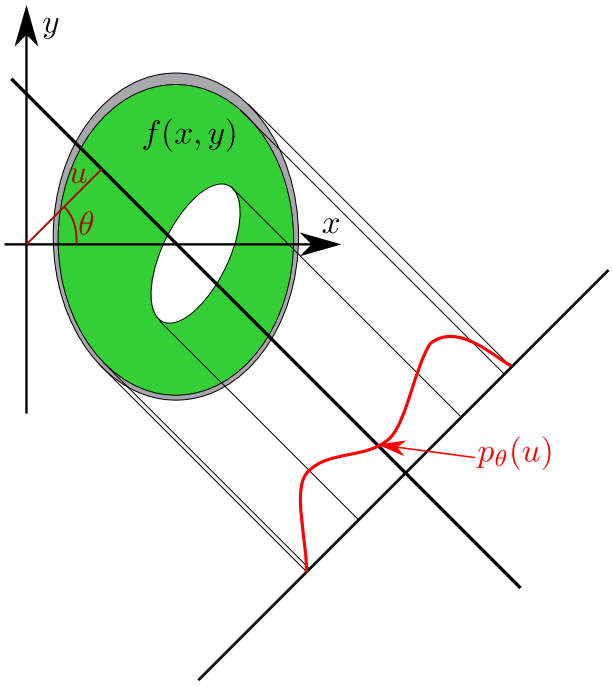


Figure 12.1: Radon transform notations.

An X-ray beam (considered as monochromatic) going through objects is attenuated, following a logarithmic law (f is the linear attenuation, v an elementary volume, I the intensity of the beam entering the volume dv and $I + dI$ the intensity exiting the volume):

$$\log \frac{I + dI}{I} = -f dv$$

By integrating this formula on a line D (direction of the beam), yields:

$$I = I_0 \exp\left(\int_D f(x, y) dv\right)$$

where $f(x, y)$ is the attenuation at point (x, y). This principle is used in scanners, scintigraphy, PET...

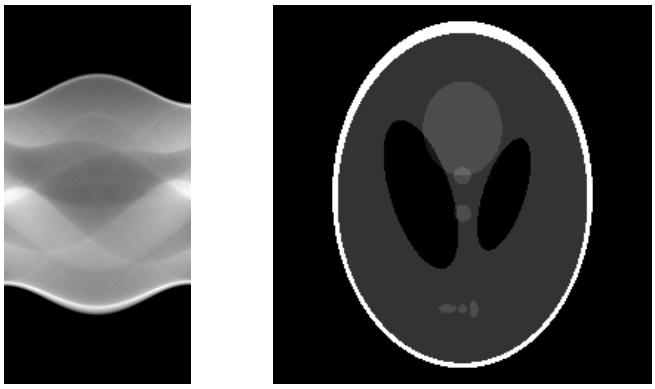
12.2. Acquisition simulation

This first exercise will allow us to simulate the projections. The mathematical operator behing this is the Radon transform.

?

- Generate a "phantom" image of size 256×256. Display it. Notice that any image may be good for testing the simulation and reconstruction via backprojection.

- To simulate the attenation process, considere the sum of all gray levels of all pixels. Angles of projections will be between 0 and 180 degres with a unit step. The result is presented as a sinogram $p_\theta(u)$ (see Fig. 12.2), with u the length (in ordinates), and θ the projection angle (in abscissa).



(a) Sinogram. (b) Phantom image.

Figure 12.2: Simulated sinogram of the phantom image.

12.3. Backprojection algorithm

A non exact reconstruction is the backprojection operator B (it is not the inverse Radon transform). It attributes the value $p_\theta(u)$ to each point of the projection line that gave this attenation, and sum up all contributions from all projections (at the different angles).

$$B[p](x,y) = \int_0^\pi p_\theta(x\cos\theta + y\sin\theta)d\theta$$

A simple (but slow) method is described to compute the backprojection. p_θ is the attenuation vector for a given angle θ.

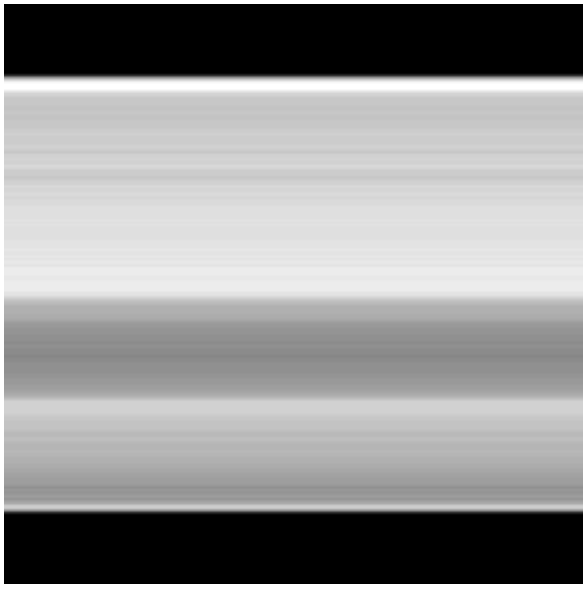
- Use a command to replicate a matrix to obtain the contribution of each line D, like on Fig. 12.3a.
- Each contribution line D should be turned of a certain angle to be added to the overall contribution. This backprojection is really fuzzy compared to the original phantom image (see Fig. 12.3b).

12.4. Filtered backprojection

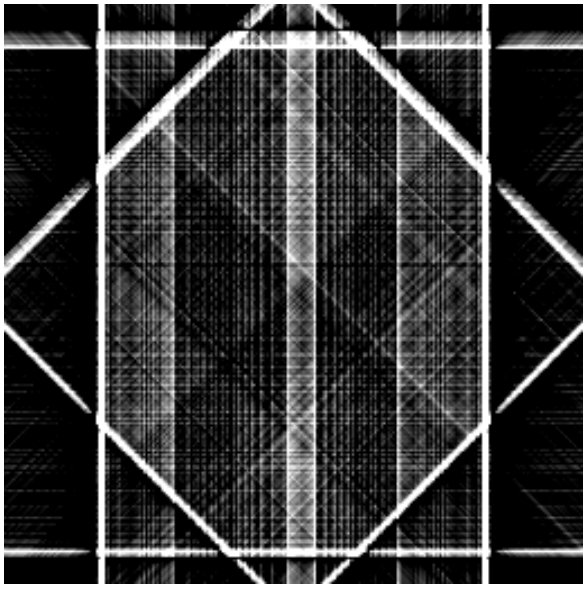
It can be shown that backprojection is in fact the convolution of f by a filter. A solution would be to make a deconvolution in the Fourier domain. The solution used here is the filtered backprojection.

Numerous filters can be used. Among them, the Ram-Lak filter RL is approximated by :

$$\begin{aligned} k \in [-B;B] \subset \mathbb{N}, RL(k) &= \frac{\pi}{4} \text{ if } k = 0 \\ &= 0 \text{ if } |k| \text{ is even} \\ &= \frac{-1}{\pi k^2} \text{ if } |k| \text{ is odd} \end{aligned}$$



(a) Contribution of a line D before rotation.



(b) Reconstruction after projection every 45 degrees (4 projections).

Figure 12.3: Backprojection

?

- Write a function that generates this filter (see prototype). The parameter $width$ is the number of points of the resulting vector.
- Modify the backprojection algorithm to filter (by convolution of the projection vector) each contribution line before the summation. Observe the improved result.



12.5. Python correction





```
import numpy as np
import matplotlib.pyplot as plt

from skimage.io import imread
from skimage.transform import radon, rescale, rotate
from scipy.signal import convolve
import skimage.io
```

12.5.1. Acquisition simulation

The phantom image is first loaded and displayed.



```
# Read image
image = imread("phantom.png", as_gray=True)
image = image [:,:,0];
plt.imshow(image, cmap='gray');
```

The simulation of the projection is simply an addition of all gray-levels of the pixels, after rotating the image in order to simulate the rotation of the object (or of the sensor). See Fig.12.3a



```
def simuProjection(I, theta):
    """
    simulation of the generation of a sinogram
    I : original image (phantom for example)
    theta: angles of projection
    """
    N = I.shape[1];
    M = len(theta);
    S = np.zeros((N,M));

    for i,ang in enumerate(theta):
        image1 = rotate(I, ang);
        S[:, i] = np.sum(image1, axis=1);

    return S;
```

12.5.2. Backprojection algorithm

The backprojection algorithm will sum-up all the contributions of each projection.



```
def backprojection(P, theta, filtre):
    """
    Backprojection of
    P: image
    theta: list of projection angles
     filtre : bool, True if filtered
    """

    N = P.shape[0];
    R = np.zeros((N,N));

    # in case of filtered back-projection
    if filtre :
        h = RamLak(31);

    # loops over all angles
    for i,ang in enumerate(theta):
        proj = P[:, i];

        # filtered back-projection
        if filtre :
            proj = convolve(proj, h, mode='same');

        proj2 = np.matlib.repmat(proj, N, 1);
        proj2 = rotate(proj2, ang);
        R = R + proj2;
    return R.transpose();
```

The results is better in the case of a filtered backprojection. The RamLak function is provided and illustrated in Fig.12.4.



```
def RamLak(width):
    """
    Ramlak filter of size width
    width must be odd
    """
    ramlak = np.zeros((2* width+1,));
    for indice, val in enumerate(np.arange(-width, width+1)):
      val = np.abs(val);

      if val==0: # center
          ramlak[indice]=np.pi/4;
      else:
        if val%2==1: # even indices
            ramlak[indice]=-1/(np.pi*val**2);
        else: # odd indices
            ramlak[indice]=0;

    return ramlak;
```

The reconstruction of the original image is obtained by the following code:

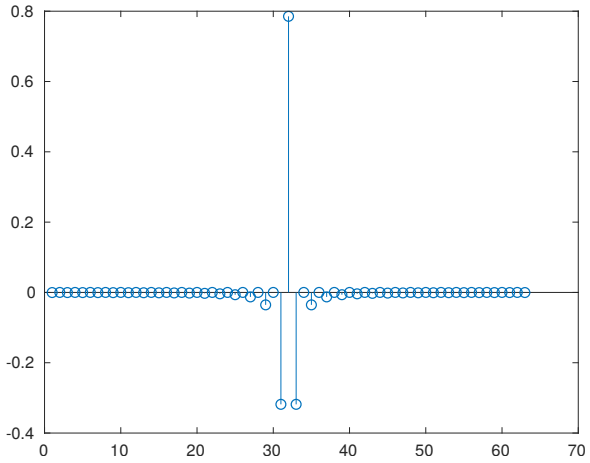


Figure 12.4: RamLak function.

```
# Performs unfiltered backprojection
ubp = backprojection(P, theta, False);
plt.figure();
plt.imshow(ubp, cmap='gray');
plt.show()

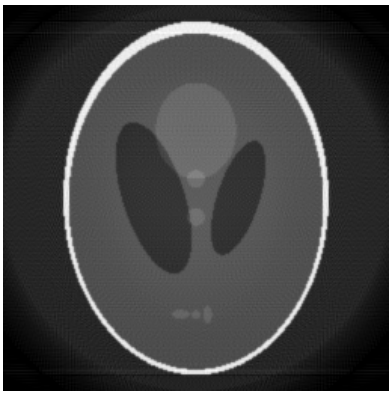
# Performs filtered backprojection
fbp = backprojection(P, theta, True);
plt.figure();
plt.imshow(fbp, cmap='gray');
plt.show()
```



(a) Original phantom image.



(b) Unfiltered backprojection.



(c) Filtered backprojection.

Figure 12.5: Reconstruction by backprojection.

Part V Alternative Algebraic Frameworks

13 Logarithmic Image Processing (LIP)

This tutorial aims to study the LIP model, which is a vector space of gray-level images, consistent with the physical laws of Weber and Fechner as well with the visual perception laws. More particularly, the LIP dynamic expansion, used for image enhancement, will be implemented as well as the Sobel LIP filter for edge detection.

The different processes will be applied on a mammographic image.



Figure 13.1: Breast

13.1. Introduction

The LIP model (Logarithmic Image processing) has been introduced in the mid 1980s. It defines a mathematical framework for image processing in a bounded interval. It is mathematically rigorous and physically justified. For more informations, refer to [29, 16, 30].

An image is represented by its gray-tone function f, defined on a spatial domain $D \subset \mathbb{R}^2$, and with values into $[0, M[$, where $M > 0$. For all gray-tone functions S defined on D, a vector space is defined by the operations of addition $\triangleq$ and multiplication $\triangleq$, and by extension with the operations of subtraction $\triangleq$ and negation:

$$\forall f, g \in S,\ f \triangleq g = f + g - \frac{fg}{M} \tag{13.1}$$

$$\forall f \in S, \forall \lambda \in \mathbb{R},\ \lambda \triangleq f = M - M\Big(1 - \frac{f}{M}\Big)^{\lambda} \tag{13.2}$$

$$\forall f \in S,\ \triangleq f = \frac{-Mf}{M - f} \tag{13.3}$$

$$\forall f, g \in S,\ f \triangleq g = M\frac{f - g}{M - g} \tag{13.4}$$

The vector space S of gray-tone functions is algebrically and topologically isomorph to the classical vector space, defined by the isomorphism φ:

$$\forall f \in S,\ \varphi(f) = -M\ln\Big(1 - \frac{f}{M}\Big) \tag{13.5}$$

The inverse isomorphism φ^{-1} is then defined by:

$$f = \varphi^{-1}(\varphi(f)) = M\Big(1 - \exp\Big(-\frac{\varphi(f)}{M}\Big)\Big) \tag{13.6}$$

This fundamental isomorphism allows the definition of the following operations on gray-tone functions:

- Dot product: $\forall f, g \in S, < f, g >_{\triangle} = \int \varphi(f)\varphi(g)$
- Euclidean norm: $\forall f \in S,\ \|f\|_{\triangle} = |\varphi(f)|_{\mathbb{R}}$, with $|\cdot|_{\mathbb{R}}$ is the classical absolute value.

13.2. Elementary LIP operations

Classically, gray-tones are coded with 8 bits, and thus one can choose $M = 256$. Be careful that there might exist sampling issues, and it might be required to considere images in the LIP space with a double precision.

?

1. Implement the elementary LIP operations $\oplus$, $\otimes$ and $\ominus$.
2. Test these operators on the image 'breast'.

13.3. LIP dynamic expansion

The LIP model enables to define an image transformation that expanses, in an optimal way, the overall dynamic range (gray-tones) while preserving a physical or visual sense. Let a graytone function denoted f be defined on the spatial support $D \subset \mathbb{R}^2$. Its upper and lower bounds are denoted $f_u = \sup_{x \in D} f(x)$ and $f_l = \inf_{x \in D} f(x)$, with $0 < f_l < f_u < M_0$.

The dynamic range of f on D is defined by $R(f) = f_u - f_l$. The LIP scalar multiplication of f by a real number $\lambda > 0$ yields to the following dynamic :

$$R(\lambda \otimes f) = \lambda \otimes f_u - \lambda \otimes f_l \tag{13.7}$$

It has been shown [17] that there exists an optimal value $\lambda_0(f)$ that maximizes the dynamic range, i.e.:

$$R(\lambda_0(f) \otimes f) = \max_{\lambda > 0} R(\lambda \otimes f). \tag{13.8}$$

?

1. Determine the explicit (analytic) expression of the parameter $\lambda_0(f)$.
2. Calculate the value of the parameter $\lambda_0(f)$ for the image 'breast'.
3. Enhance the image and compare the result with the histogram equalization.



In python, one can use the scikit-image module, and the `skimage.exposure.equalize_hist` function.

13.4. LIP edge detection

?

1. Implement the Sobel filter within the LIP framework.
2. Compare the results with the classical Sobel filter.



13.5. Python correction



13.5.1. Elementary operations

The most important function to code is the graytone transformation function. It considers M as the maximum value (the absolute white). This code means that the absolute white cannot be reached, and thus $f \in [0; M[$. After discretizing the gray values, $F \in [0; M-1]$.



```
"""
 file LIP.py
LIP simple module
USAGE: import LIP

"""
import numpy as np

def graytone(F, M):
    # graytone function transform
    # M: maximal value
    # F: image function
    f = M-np.finfo( float ).eps-F;
    return f;
```



```
def phi(f, M):
    # LIP isomorphism
    # f: graytone function
    # M: maximal value
    l = -M * np.log(-f/M+1);
    return l
```



```
def invphi(l, M):
    # inverse isomorphism
    f = M*(1-np.exp(-l/M));
    return f
```



```
def plusLIP(a, b, M):
    # LIP addition
    z = a+b-(a*b)/M;
    return z;
```



```
def timesLIP(alpha, x, M):
    # LIP multiplication by a real
    z = M-M*(np.ones(x.shape)-x/M)**alpha;
    return z;
```

13.5.2. LIP dynamic expansion

The optimal value for dynamic expansion is given by λ_0:

$$\lambda_0(f) = \arg\max_{\lambda} \{\max(\lambda \vartriangleleft f) - \min(\lambda \vartriangleleft f)\}$$

Let $A(\lambda) = \max(\lambda \vartriangleleft f) - \min(\lambda \vartriangleleft f) = \lambda \vartriangleleft \max(f) - \lambda \vartriangleleft \min(f)$ and $B = \ln(1 - \min(f)/M)$ et $C = \ln(1 - \max(f)/M)$.

$$\begin{aligned}
A'(\lambda) = 0 &\Leftarrow [(M - M\exp(\lambda C)) - (M - M\exp(\lambda B))]' = 0 \\
&\Leftarrow [\exp(\lambda B) - \exp(\lambda C)]' = 0 \\
&\Leftarrow B\exp(\lambda B) - C\exp(\lambda C) = 0 \\
&\Leftarrow \ln(B) + \lambda B = \ln(C) - \lambda C \\
&\Leftarrow \lambda = \frac{\ln(C) - \ln(B)}{B - C} \\
&\Leftarrow \lambda = \frac{\ln(C/B)}{B - C}
\end{aligned}$$

Thus, yielding to:

$$\lambda_0(f) = \frac{\ln\left(\frac{\ln(1 - \max(f)/M)}{\ln(1 - \min(f)/M)}\right)}{\ln\left(\frac{M - \min(f)}{M - \max(f)}\right)}$$

This is coded in python by:



```
def computeLambda(f, M):
    # compute optimal value for dynamic expansion
    B = np.log(1-f.min()/M);
    C = np.log(1-f.max()/M);
    l = np.log(C/B)/(B-C);
    return l;
```

13.5.3. Complete code

This code uses a transform called histogram equalization. This version proposes our own code for the histogram equalization (see also 7.



```
import LIP
import numpy as np
from scipy import misc
import matplotlib.pyplot as plt
import skimage
""" Histogram equalization
"""
def histeq(im,nbr_bins=256):
    #get image histogram
    imhist,bins = np.histogram(im.flatten(),nbr_bins,normed=True)
    cdf = imhist.cumsum() #cumulative distribution function
    cdf = 255 * cdf / cdf[-1] #normalize
    #use linear interpolation of cdf to find new pixel values
    im2 = np.interp(im.flatten(),bins[:-1],cdf)

    return im2.reshape(im.shape), cdf
```

It is compared to the LIP dynamic enhancement. The results are illustrated in Fig. 13.2.



```
M = 256.

# reads original image
B = imageio.imread("breast.jpg");

# conversion to gray-tones (see LIP definition)
tone = LIP.graytone(B, M);
D = LIP.graytone(LIP.timesLIP(.5, tone, M), M);

# compute optimal enhancement value
l = LIP.computeLambda(tone, M);
print "lambda: %f" % l
# apply enhancement and get back into classical space
E = LIP.graytone(LIP.timesLIP(l, tone, M), M);

# histo equalization, for comparison purposes
heq,cdf = histeq(B);

# display results
plt.figure();
plt.subplot(1,3,1); plt.imshow(E/M, cmap=plt.cm.gray, vmin=0, vmax=1); plt.title('dynamic expansion');
plt.subplot(1,3,2); plt.imshow(B/M, cmap=plt.cm.gray, vmin=0, vmax=1); plt.title('original image');
plt.subplot(1,3,3); plt.imshow(heq/M, cmap=plt.cm.gray, vmin=0, vmax=1); plt.title('after histo equalization');
```

13.5.4. Edge detection

When applying an operator in the LIP framework, it is better to apply first the isomorphisme, then the operator, and then get back into the classical space. This is applied for example for an edge detection operator.

The method applied here is the Sobel gradient. The results are presented in Fig. 13.3.

Figure 13.2: Comparison of the LIP enhancement with the classical histogram equalization method.

(a) Original image



(b) LIP enhancement.



(c) Histogram equalization.





```
# contours detection
# Sobel detection
def Sobel(I):
    sobx=cv2.Sobel(I, -1, 1, 0);
    soby=cv2.Sobel(I, -1, 0, 1);
    sob = np.sqrt(sobx**2 + soby**2);
    return sob;

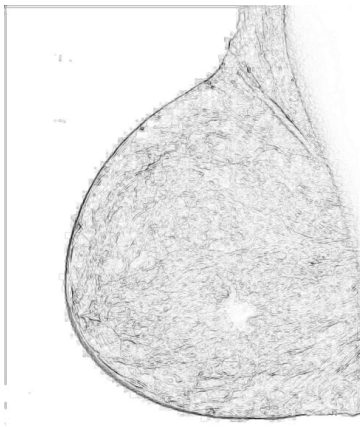
# go into LIP space
tonelip = LIP.phi(tone, M);

# apply Sobel filter
sobellip = LIP.graytone(LIP.invphi(Sobel(tonelip), M), M);
imageio.imwrite("sobellip.png", sobellip);
plt.figure();
plt.subplot(1,2,1); plt.imshow(sobellip, cmap=plt.cm.gray);
plt.title('LIP Sobel edge detection')

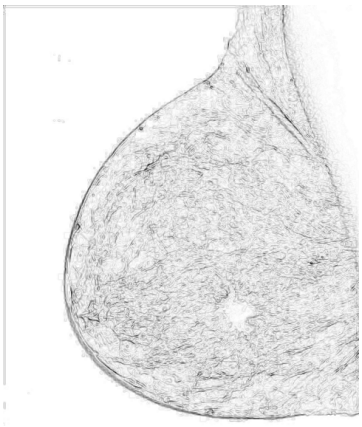
# apply Sobel filter in the classic space
sobel = 255-Sobel(B);
plt.subplot(1,2,2); plt.imshow(sobel, cmap=plt.cm.gray);
plt.title('Sobel edge detection')
imageio.imwrite("sobel.png", sobel);
```

Figure 13.3: Sobel edge detection

(a) LIP Sobel edge detection.



(b) Classic Sobel edge detection.

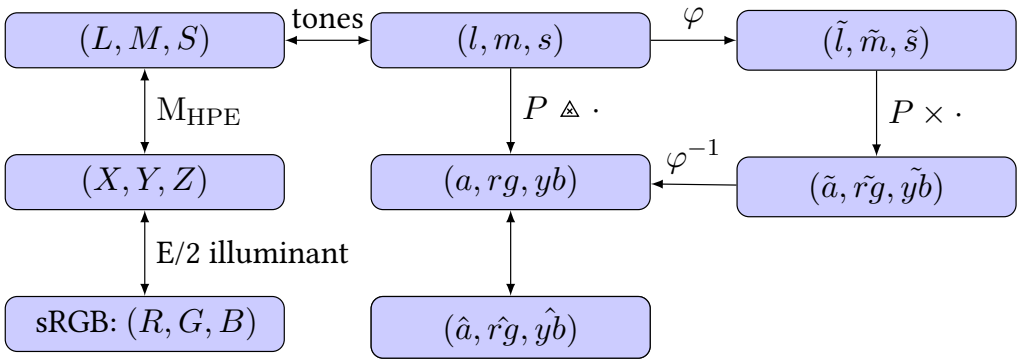


14 Color Logarithmic Image Processing (CoLIP)

The objective of this tutorial is to have an overview of the Color Logarithmic Image Processing (CoLIP) framework [12, 13, 9, 10]. The CoLIP is a mathematical framework for the representation and processing of color images. It is psychophysically well justified since it is consistent with several human visual perception laws and characteristics. It allows to consider color images as vectors in an abstract linear space, contrary to the classical color spaces (e.g., RGB and $L^*a^*b^*$). The purpose of this tutorial is to present the mathematical fundamentals of the CoLIP by manipulating the color matching functions and the chromaticity diagram.

14.1. Definitions

This diagram summarizes the transitions between the classical RGB space and the CoLIP space. The transfer matrices are presented in the following.



14.1.1. RGB to XYZ

Here, we choose to keep the conventions of the CIE 1931, i.e. a 2° 1931 observer, with the equal energy illuminant E and the sRGB space. Depending on the illuminant, there exist many conversion matrices.

Use the module skimage.color.

14.1.2. XYZ to LMS

The matrix M_{HPE} (Hunt, Pointer, Estevez) is defined by:

$$\begin{pmatrix} L \\ M \\ S \end{pmatrix} = \mathrm{M_{HPE}} \times \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}, \tag{14.1}$$

with

$$\mathrm{M_{HPE}} = \begin{pmatrix} 0.38971 & 0.68898 & -0.07868 \\ -0.22981 & 1.18340 & 0.04641 \\ 0.00000 & 0.00000 & 1.00000 \end{pmatrix}. \tag{14.2}$$

14.1.3. LMS to lms

$$\forall c \in \{l,m,s\}, C \in \{L,M,S\}, c = M_0\left(1-\frac{C}{C_0}\right), \tag{14.3}$$

with C_0 is the maximal transmitted intensity value. M_0 is arbitrarily chosen at normalized value 100. Notice that $C \in]0;C_0]$ and $c \in [0;M_0[$.

14.1.4. CoLIP homeomorphisme

$$\forall f \in S,\ \varphi(f) = -M_0 \ln\left(1-\frac{f}{M_0}\right).$$

The logarithmic response of the cones, as in the LIP theory, is modeled through the isomorphism φ:

$$\text{for } c \in \{l,m,s\},\ \tilde{c} = \varphi(c) = -M_0 \ln\left(1-\frac{c}{M_0}\right), \tag{14.4}$$

where $(\tilde{l},\tilde{m},\tilde{s})$ are called the logarithmic chromatic tones.

14.1.5. Opponent process

$$\begin{pmatrix} \tilde{a} \\ \tilde{rg} \\ \tilde{yb} \end{pmatrix} = P \times \begin{pmatrix} \tilde{l} \\ \tilde{m} \\ \tilde{s} \end{pmatrix}, \tag{14.5}$$

with

$$P = \begin{pmatrix} 40/61 & 20/61 & 1/61 \\ 1 & -12/11 & 1/11 \\ 1/9 & 1/9 & -2/9 \end{pmatrix}. \tag{14.6}$$

14.1.6. Bounded vector space

Three channels $\hat{f} = (\hat{a},\hat{rg},\hat{yb})$ are defined by:

$$\hat{a} = a \tag{14.7}$$

$$\hat{rg} = \begin{cases} rg & \text{if} \quad rg \geq 0 \\ -\triangleq rg & \text{if} \quad rg < 0 \end{cases} \tag{14.8}$$

$$\hat{yb} = \begin{cases} yb & \text{if} \quad yb \geq 0 \\ -\triangleq yb & \text{if} \quad yb < 0 \end{cases} \tag{14.9}$$

14.2. Applications

14.2.1. CMF

?

Represent the classical chromaticity by using the color matching functions.

The color matching functions in the different spaces, as well as the wavelength and the purple line, are provided in the matrix 'cmf.npz'. Load it with numpy.load.

14.2.2. CoLIP chromaticity diagram

?

Represent the chromaticity diagram in the plane $(\hat{rg}, \hat{yb})$ as well as the Maxwell triangle of the RGB colors, as in Fig.14.1.

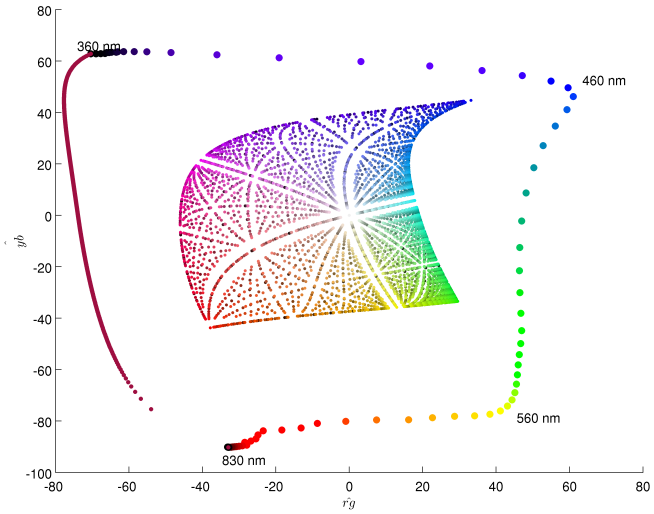


Figure 14.1: Chromaticity diagram in the plane $(\hat{rg}, \hat{yb})$.

14.3. Python correction

```
import numpy as np
import matplotlib.pyplot as plt
import skimage.color as color
```

First of all, the maximal value M_0 is arbitrarily fixed at 100.

```
def getColipM0():
    # return M0 value
    return 100;
```

14.3.1. LMS tones

This is the difficult part of LIP and CoLIP. Be careful with the use of the function eps that returns the precision at a given double value.

```
def lmstone(LMS):
    M0= getColipM0();
    return (M0-np.finfo(float).eps)*(1-LMS/M0);
```

14.3.2. Isomorphism

The isomorphism is the conversion into/back from the logarithmic space.

```
def phi(f, M):
    # LIP isomorphism
    # f: graytone function
    # M: maximal value
    l = -M * np.log(-f/M+1);
    return l
```

```
def invphi(l, M):
    # inverse isomorphism
    f = M*(1-np.exp(-l/M));
    return f
```

14.3.3. XYZ to LMS

This conversion uses the HPE matrix (Hunt, Pointer, Estevez).



```
def XYZ2LMS(XYZ):
    """
    conversion function
    XYZ: data in XYZ space
    """
    U=np.array ([[0.38971,  0.68898,  - 0.07869],
      [- 0.22981,1.18340,    0.04641],
      [0,          0,          1        ]]) ;
    m, n = XYZ [:,:,0]. shape;
    print (m,n)
    XYZ = XYZ.reshape((m*n, 3)) . transpose () ;
    print (XYZ.shape)
    LMS = np.matmul(U, XYZ);

    return LMS.transpose() .reshape ((m, n, 3)) ;
```

14.3.4. Conversions into Colip space

The conversion into $(\hat{a}, \hat{rg}, \hat{yb})$ goes first into $(\tilde{a}, \tilde{rg}, \tilde{yb})$. It consists on a conversion matrix followed by the absolute value and the operation to get symmetric channels in the colip space.



```
def LMS2ARGYBhat(LMS):
    ARGYBtilde = LMS2ARGYBtilde(LMS);
    return ARGYBtilde2ARGYBhat(ARGYBtilde);
```



```
def LMS2ARGYBtilde(LMS):
    """
    conversion function
    LMS: data in LMS space
    """
    M0 = getColipM0();
    m, n, p = LMS.shape;
    LMStone = lmstone(LMS);
    LMStilde= phi(LMStone, M0);
    P =  [[40/61,20/61,1/61],   [1, - 12/11,1/11],   [1/9,1/9, -2/9  ]];

    LMStilde = LMStilde.reshape((m*n, 3)) . transpose () ;
    ARGYBtilde = np.matmul(P, LMStilde) . transpose () ;
    return ARGYBtilde.reshape((m, n, 3)) ;
```

```
def ARGYBtilde2ARGYBhat(ARGYBtilde):
    """
    conversion into ARGYB tilde space
    ARGYBtilde: data in ARGYB tilde space

    """
    M0 = getColipM0();

    ARGYBhat = np.zeros(ARGYBtilde.shape);
    ARGYBhat[:,:,0] = invphi(ARGYBtilde [:,:,0], M0);

    for c in (1,2):
        tmp = np.abs(ARGYBtilde [:,:, c]);

        ARGYBhat[:,:,c] = np.sign(ARGYBtilde [:,:, c]) * invphi(tmp, M0);

    return ARGYBhat;
```

14.3.5. CMF

The color matching functions are provided for convenience. There exist many resources on the internet where they can be found. The classical diagram in the xy space (the horseshoe) is shown in Fig.14.2.

```
with np.load('cmf.npz') as data:
    cmap = data['cmap'];
    pourpresLMS = data['pourpresLMS'];
    SpecXYZ = data['SpecXYZ'];
    SpecLMS = data['SpecLMS'];
    SpecRGB = data['SpecRGB'];
    l = data['l'];

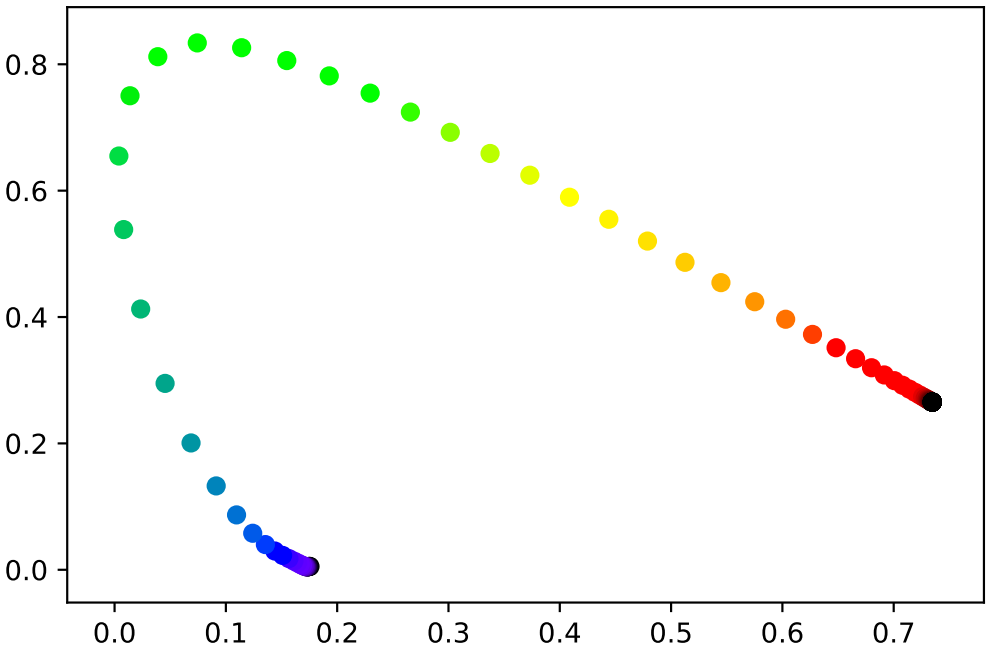
# display classical CMF in xy
xn = SpecXYZ [:,:,0]/ np.sum(SpecXYZ, axis=2);
yn = SpecXYZ [:,:,1]/ np.sum(SpecXYZ, axis=2);
zn = 1-xn-yn;
plt.scatter (xn, yn, c=cmap);
```

To display the CMF and the cube of all RGB colors in the $(\hat{rg}, \hat{yb})$ space, the following code is used: The result is shown in Fig.14.3.

```
#Color matching functions in colip space
ARGYB_hat = LMS2ARGYBhat(SpecLMS);
plt.figure ();
plt.scatter (ARGYB_hat[:,0,1], ARGYB_hat[:,0,2], c = cmap);
# purple line
purple_ARGYB_hat = LMS2ARGYBhat(pourpresLMS);
plt.scatter (purple_ARGYB_hat[:,0,1], purple_ARGYB_hat[:,0,2], c='black');
```

The RGB cube is first generated and then converted into the colip space.

Figure 14.2: Color matching functions in the xy space.



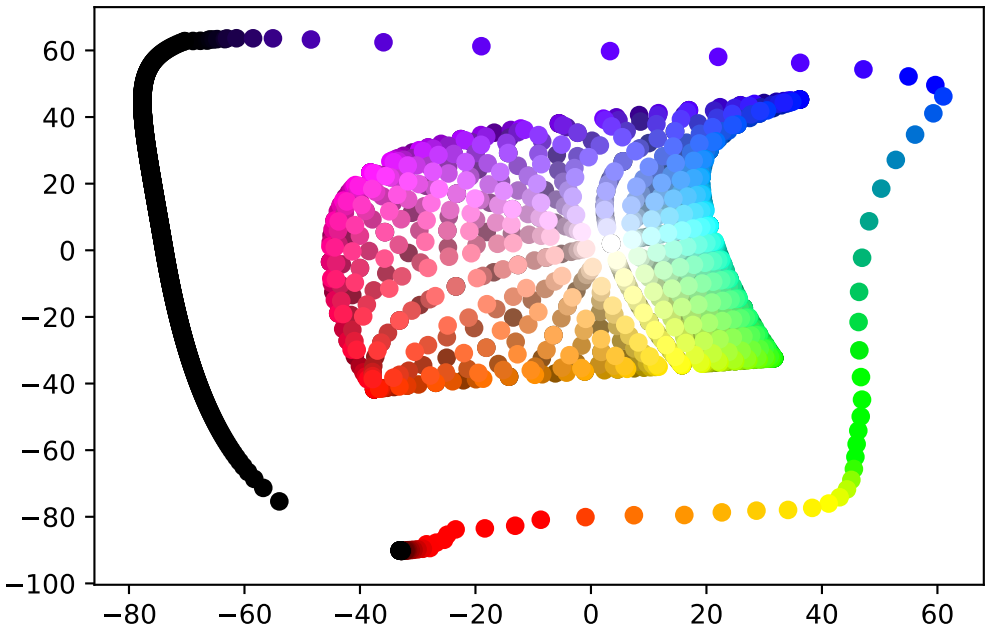

```
# number of color in each direction
N=10;
cols = np.linspace(0, 255, num=N);
R, G, B = np.meshgrid(cols, cols, cols);

# reshape the cube for manipulation
R = R.reshape((R.size, 1));
G = G.reshape((G.size, 1));
B = B.reshape((B.size, 1));
NN = R.size;
colRGB = np.concatenate((R, G, B), axis=1);
cubeRGB = colRGB.reshape((NN, 1, 3));
colRGB = colRGB / 255;

# conversions
cubeXYZ = color.rgb2xyz(cubeRGB);
cubeLMS = XYZ2LMS(cubeXYZ);
cubeARGYBhat = LMS2ARGYBhat(cubeLMS);
plt.scatter(cubeARGYBhat[:,0,1], cubeARGYBhat[:,0,2], c=colRGB);
```

The color matching functions in the colip space are represented in Fig.14.3. A lot could be said about this diagram. For examples, one can notice the approximations in the high (red) wavelengths: this comes from the numerical approximations and the lack of data of the original CMFs. Moreover, the purple line (represented here in black) is not a line in this diagram. This raises the question of the choice of this line in (xy), which was made only by simplicity.

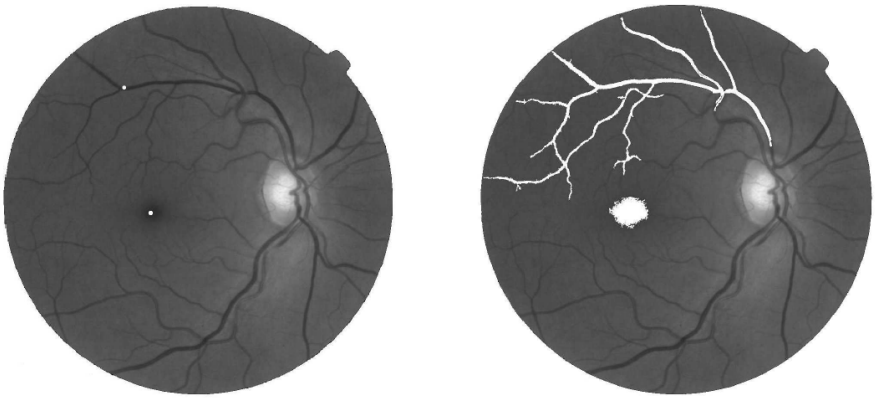
Figure 14.3: Color matching functions in the $(\hat{rg}, \hat{yb})$ space.



15 GANIP

This tutorial aims to test the elementary operators of the GANIP framework.
The General Adaptive Neighborhood Image Processing (GANIP) is a mathematical framework for adaptive processing and analysis of gray-tone and color images. An intensity image is represented with a set of local neighborhoods defined for each pixel of the image to be studied. Those so-called General Adaptive Neighborhoods (GANs) are simultaneously adaptive with the spatial structures, the analyzing scales and the physical settings of the image to be addressed and/or the human visual system.

The following figure illustrates the GANs of two points on an image of retinal vessels.



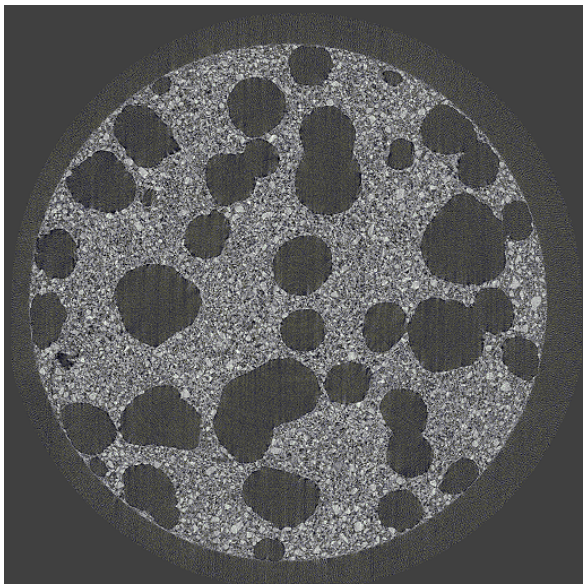
The GANs are then used as adaptive operational windows for local image transformations (morphological filters, rank/order filters...) and for local image analysis (local descriptors, distance maps...).
The different processes will be applied on the following gray-tone images:

Figure 15.1: Two example images for testing the GAN framework.

(a) lena



(b) cement paste (2-D section / X-ray tomography)



15.1. GAN

Let f be a gray-tone image. The GAN of a point x using the luminance criterion and the homogeneity tolerance m within the CLIP framework is defined as:

$$V_m^f(x) = C_{\{y;|f(y)-f(x)|\leq m\}}(x) \tag{15.1}$$

where $C_X(x)$ denotes the connected component of X holding x.

?

1. Load the image 'lena' and compute the GAN of a selected point in the image.
2. Look at the influence of the homogeneity tolerance.
3. Compute the GAN of different points and comment.

15.2. GAN Choquet filtering

The Choquet filters generalize the rank-order filters. In this exercise, we are going to implement some GAN Choquet filters such as the GAN mean operator. For each point x of the image, the mean value of all the intensities of the points inside the GAN of x is computed. So, a first algorithm consists in making a loop on the image points for computing the different GAN and the mean intensity values, but it is time consuming. Nevertheless, by using some properties of the GAN (particularly the one giving that iso-valued points can have exactly the same GAN), it is possible to create a second algorithm by making a loop on the gray-tone range:

Data: original (8-bit) image f, homogeneity tolerance m
Result: GAN mean filtered image g
for $s = 0$ **to** 255 **do**
  $seed =$ points x with intensity $f(x) = s$;
  $thresh =$ points y with intensity satisfying $s - m \leq f(y) \leq s + m$;
  $threshGAN =$ connected components of $thresh$ holding $seed$;
  foreach *label of* $threshGAN$ **do**
    $currentLabel =$ current connected component of $threshGAN$;
    $meanValue =$ intensity mean value of the image points inside $currentLabel$;
    set $g(x) = meanValue$ to the points x of $seed$ belonging to $currentLabel$;
  end
end

?

1. Implement the proposed algorithm.
2. Test this operator on the 'lena' image with different homogeneity tolerances.
3. Compare the result with a classical mean filtering.



See scipy.ndimage.filters.

15.3. GAN morphological filtering

As previously explained, it is also possible to compute the GAN dilation and GAN erosion by making a loop on the gray-tone range. The algorithm for the GAN dilation is:

Data: original (8-bit) image f, homogeneity tolerance m
Result: GAN dilated image g
set $g(x) = 0$ for all points x;
for $s = 0$ **to** 255 **do**
  $seed =$ points x with intensity $f(x) = s$;
  $thresh =$ points y with intensity satisfying $s - m \leq f(y) \leq s + m$;
  $threshGAN =$ connected components of $thresh$ holding $seed$;
  foreach *label of* $threshGAN$ **do**
    $currentLabel =$ current connected component of $threshGAN$;
    $maxValue =$ intensity maximum value of the image points inside $currentLabel$;
    set $g(x) = max(g(x), maxValue)$ to the points x belonging to $currentLabel$;
  end
end

Algorithm 2: GAN dilation

The algorithm for the GAN erosion is similar.

?

1. Implement the GAN dilation and GAN erosion.
2. Test this operator on the 'lena' image with different homogeneity tolerances.
3. Compare the results with the classical morphological dilation and erosion..
4. Compute, test, compare and check the properties (idempotence, extensivity/anti-extensivity) of the GAN opening and GAN closing.



Compare with the functions dilation and erosion from skimage.morphology.

15.4. Python correction

```
import numpy as np
import skimage.morphology
import skimage.io
import skimage.measure
import matplotlib.pyplot as plt
import scipy.ndimage.filters
```

15.4.1. GAN

In order to show the General Adaptive Neighborhood (GAN) of one image point, the following function is implemented. Note that the function is given within the Classical Linear Image Processing (CLIP) framework, i.e. with the usual addition, subtraction and scalar multiplication. But the function could be generalized to other models such as the Logarithmic Image Processing (LIP).

```
def GAN(A, p, m):
    """
    GAN construction
    A: grayscale image
    p: point
    m: tolerance
    """
    RES = np.zeros(A.shape);
    RES[p[0], p[1]] = 1;
    s = A[p[0], p[1]];
    thresh = np.logical_and(A>= s-m, A<=s+m);
    SE = np.ones((3,3));
    RES = skimage.morphology.reconstruction(RES, thresh, selem=SE);
    return RES;
```

In this way, we can visualize the GAN of any point of the 'Lena' image:

```
I = skimage.io.imread('lena.png');
p=[100,200];
for m in 5,50,75:
    RES = GAN(I, p, m);
    plt.imshow(RES, cmap='gray');
    plt.show()
```

15.4.2. GAN Choquet filtering

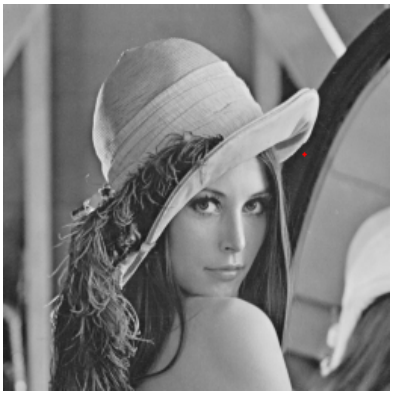
In order to compute the GAN mean filtering, the basic idea is to make a loop on the image points, compute the GAN of the current point and then calculate the mean intensity of the points within the GAN. But this is time computing in the sense that two points with the same intensity can have exactly the same GAN (when one point is included in the GAN of the second point with same intensity). Therefore, a more effective way is to make a loop on the gray levels of the image. The GAN mean filtering is then implemented as:

Figure 15.2: GAN of a specific point of the 'Lena' image using different homogeneity tolerances m within the CLIP framework.

(a) Original image.



(b) Point location.



(c) GAN (m=5).



(d) GAN (m=50).



(e) GAN (m=75).



```
def GANmean(A, m):
    """
    GAN mean filter
    Apply mean value of the adaptive neighborhood to result
    A: original image
    m: homogeneity tolerance
    return: GAN mean filter
    """
    RES = np.zeros(A.shape).astype('float');
    SE = np.ones((3,3));
    for s in np.arange(256):
        thresh = np.logical_and(A>=s-m, A<=s+m);
        seed = A==s;
        thresh = skimage.morphology.reconstruction(seed, thresh, selem=SE);
        L,n = skimage.measure.label(thresh, connectivity=2, return_num=True);
        for l in np.arange(1, n+1):
            currentLabel = L==l;
            values = A[currentLabel];
            meanValues = np.mean(values);
            result = meanValues * np.logical_and(seed, currentLabel).astype('float');
            RES = RES + result;
    return RES;
```

Looking at this function, for the first loop on the gray levels s, we detect the GANs of the pixels with gray level s. So 'thresh' contains all these connected components (GANs), where some points with identical gray levels s can have the same GAN. 'label' enables the different GANs to be labeled. For the second loop on these GANs, we

extract for each GAN its gray levels which are stored in 'values'. Afterwards, we take the mean value 'meanVaue' that is stored as the resulting gray level for pixels inside the GAN with the same gray level s.

We can compare this GAN filtering with the classical one using a fixed operational window.

```
B=scipy.ndimage. filters . uniform_filter (I, size =5);
m=30;
C = GANmean(I, m);
```

Figure 15.3: Classical ($r = 5$) vs. GAN ($m = 30$) mean filtering of the 'Lena' image.

(a) Original image.



(b) Classical filtering ($r = 5$).



(c) GAN filtering ($m = 30$).



You can see the blurring effect caused by the classical filtering, contrary to the adaptive GAN filtering where the transitions are much more preserved while smoothing the image.

GAN morphological filtering

The following codes enable the GAN dilation and erosion to be computed. As previously mentioned, it is based on a loop on the gray levels and not on the image points. Note that the GANs ae computed on the criterion image (it is important for satisfying the good properties of opening and closing, such as idempotence). Please notice that this function as well as the following will have the same parameters:

```
"""
A: original image
criterion : criterion image
m: tolerance
"""
```

```
def GANdilation(A, criterion , m):
    RES = np.zeros(A.shape).astype('float');
    SE = np.ones ((3,3) );
    for s in np.arange(256):
        thresh = np.logical_and( criterion >=s-m, criterion <=s+m);
        seed = criterion ==s;
        thresh = skimage.morphology.reconstruction(seed, thresh, selem=SE);
        L,n = skimage.measure.label(thresh, connectivity =2, return_num=True);
        for l in np.arange(1, n+1):
            currentLabel = L==l;
            values = A[currentLabel];
            values.sort();
            result = currentLabel.astype('float') * values[-1];
            RES = np.maximum(RES, result);
    return RES;
```

```
def GANerosion(A, criterion , m):
    RES = 255 * np.ones(A.shape).astype('float');
    SE = np.ones ((3,3) );
    for s in np.arange(256):
        thresh = np.logical_and( criterion >=s-m, criterion <=s+m);
        seed = criterion ==s;
        thresh = skimage.morphology.reconstruction(seed, thresh, selem=SE);
        L,n = skimage.measure.label(thresh, connectivity =2, return_num=True);
        for l in np.arange(1, n+1):
            currentLabel = L==l;
            values = A[currentLabel];
            values.sort();
            result = currentLabel.astype('float') * values[0] + 255*np.logical_not(currentLabel.astype('float')
                ↪ );
            RES = np.minimum(RES, result);
    return RES;
```

The following script shows the difference between classical and adaptive morphology.

```
Cdil = GANdilation(I, m);
Cero = GANerosion(I, m);
se =skimage.morphology.disk(2);
Bdil = skimage.morphology.dilation(I, selem=se);
Bero = skimage.morphology.erosion(I, selem=se);
```

As previously mentioned with the Choquet filtering, you can see the blurring effect caused by the classical filtering, contrary to the adaptive GAN filtering where the transitions are much more preserved while smoothing the image.

Now, we can compute the GAN opening and closing as combined operators of dilation and erosion.

Figure 15.4: Classical ($r = 2$) vs. GAN ($m = 30$) morphological filtering of the 'Lena' image.

(a) Original image.



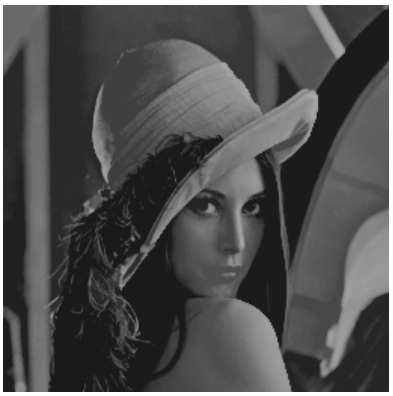
(b) Classical erosion ($r = 2$).



(c) Classical dilation ($r = 2$).



(d) GAN erosion ($m = 30$).



(e) GAN dilation ($m = 30$).



```
def GANopening(A, criterion, m):
    """
    """
    temp = GANerosion(A, criterion , m);
    RES  = GANdilation(temp, criterion , m);
    return RES;
```

```
def GANclosing(A, criterion , m):
    """
    """
    temp = GANdilation(A, criterion , m);
    RES  = GANerosion(temp, criterion , m);
    return RES;
```

It is very important to use the same criterion h when combining these two operators of GAN dilation and erosion. It means that we compute the GANs at the beginning and therefater we use these same GANs for computing dilation and erosion. It enables the idempotence, extensivity/anti-extensivity of the GAN opening and GAN closing to be satisfied.

```
Copen=GANopening(I, I, m);
Cclose=GANclosing(I, I, m);

# classical open close
Bopen = skimage.morphology.opening(I, selem=se);
Bclose= skimage.morphology.closing(I, selem=se);
```

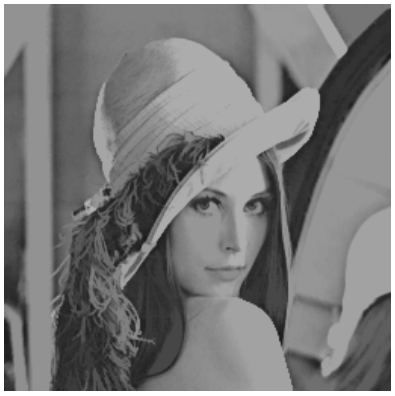
The result of such morphological filters is presented in Fig.15.5.

Figure 15.5: GAN ($m = 30$) opening and closing of the 'Lena' image.

(a) Original image.



(b) GAN opening ($m = 30$).



(c) GAN closing ($m = 30$).



We can check some properties of these morphological filters such as extensivity/anti-extensivity and idempotence:

```
Copen2 = GANopening(Copen, I, m);
Cclose2= GANclosing(Cclose, I, m);
if np.max(np.abs(Copen2-Copen))==0:
    print("OK: idempotence in opening");
else:
    print("error in opening");

if np.max(np.abs(Cclose2-Cclose))==0:
    print("OK: idempotence in closing");
else:
    print("error in opening");
```

```
#%% check extensivity and anti-extensivity
if np.all(Copen<=I):
    print("OK: extensivity");
else:
    print("error in extensivity");

if np.all(Cclose>=I):
    print("OK: anti-extensivity");
else:
    print("error in anti-extensivity");
```

Result verifies these properties.



```
OK: idempotence in opening
OK: idempotence in closing
OK: extensivity
OK: anti-extensivity
```

Part VI Mathematical Morphology

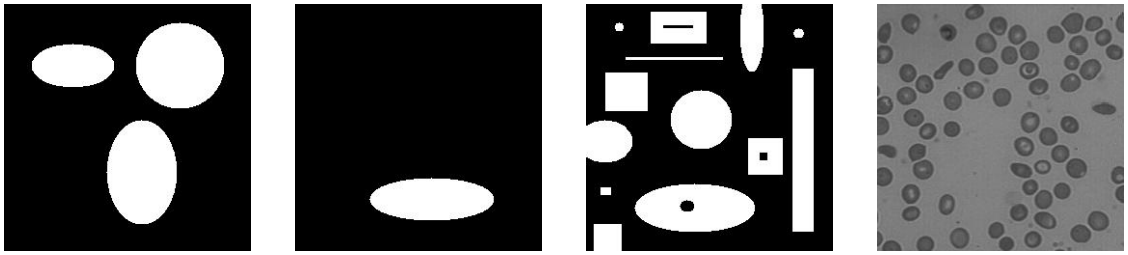
⋆ 16 Binary Mathematical Morphology

The objective of this tutorial is to process binary images with the elementary operators of mathematical morphology. More particularly, different image transformations, based on the morphological reconstruction, will be studied (closing holes, removing small objects…).

The different transformations will be applied on the following images Fig. 16.1:

Figure 16.1: Images to use for this tutorial.

(a) I_1. (b) M. (c) I_2. (d) Cells.



16.1. Introduction to mathematical morphology

Mathematical morphology started in the 1960s with Serra and Matheron [39]. It is based on Minkowski addition of sets. The main operators are erosion and dilation, and by composition, opening and closing. More informations can be found in [41].

The erosion of a binary set A by the structuring element B is defined by:

$$\varepsilon_B(A) = A \ominus B = \{z \in A | B_z \subseteq A\} \tag{16.1}$$

where, $B_z = \{b + z | b \in B\}$.

The dilation can by obtained by:

$$\delta_B(A) = A \oplus B = \{z \in A | (B^s)_z \cap A \neq \varnothing\} \tag{16.2}$$

where $B^s = \{x \in E | -x \in B\}$ is the symmetric of B.

By composition, the opening is defined as: $A \circ B = (A \ominus B) \oplus B$. The closing is defined as: $A \bullet B = (A \oplus B) \ominus B$.

16.2. Elementary operators

?

- Create a square structuring element by simply generating a square matrix of ones.
- Create a circular structuring element.
- Test the functions of dilation, erosion, opening and closing on the image I_2 by varying:
  1. the shape of the structuring element,
  2. the size of the structuring element.



The python functions come from the python module scipy.ndimage.morphology. Useful functions are binary_dilation, binary_erosion, binary_opening and binary_closing.
There are multiple modules and functions for reading images; among them, you could use imagio.imread or skimage.io.imread.

For generating a circular structuring element, the following function makes use of the numpy.meshgrid functions.



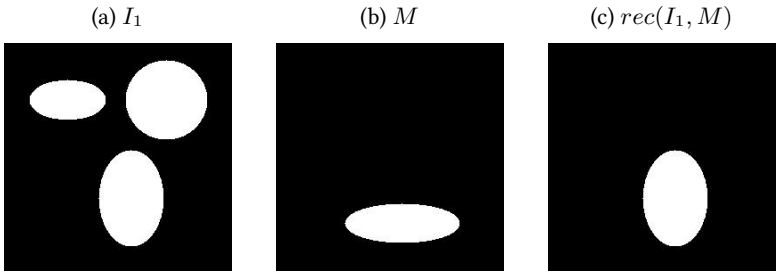
```
def disk(radius):
    # circular structuring element with a given radius
    x = np.arange(-radius, radius+1, 1)
    xx, yy = np.meshgrid(x, x)
    d = np.sqrt(xx**2 + yy**2)
    return d <= radius
```

16.3. Morphological reconstruction

The operator of morphological reconstruction ρ is very powerful and largely used for practical applications. The principle is very simple. We consider two binary images: I_1 (the studied binary image) and M (the marker image). The objective is to reconstruct the elements of I_1 marked by M as illustrated in the Fig. 16.2.

To do this, we iteratively dilate the marker M while being included in I_1 (see Eq.16.3). In order to guarantee this inclusion in A, we keep from each dilated set its intersection with I_1. The algorithm is stopped when the process dilation-intersection is equal to the identity transformation (convergence).

Figure 16.2: Illustration of morphological reconstruction of I_1 by M.



Let $\delta_I^c(M) = \delta_{B_1}(M) \cap I$ be the dilation of the marker set M constrained to the set I. Then, the morphological reconstruction is defined as:

$$\rho_I(M) = \lim_{n \to \infty} \underbrace{\delta_I \circ \ldots \circ \delta_I(M)}_{n \text{ times}} \tag{16.3}$$

Data: image I and marker M
Result: reconstructed image $rec(I, M)$
$r = area(M)$;
$s = 0$;
while $r \neq s$ **do**
  $s = r$;
  $M = I \cap (M \oplus B_1)$;
  $r = area(M)$;
end
$rec(I, M) = M$;

Algorithm 3: The algorithm of this morphological reconstruction

?

1. Implement the algorithm.
2. Test this operator with the images I_1 and M.

Evaluate the area of a set may be done by counting the number of its pixels.

16.4. Operators by reconstruction

?

Using the reconstruction operator, implement the 3 following transformations:

1. removing the border objects,
2. removing the small objects,
3. closing the object holes.

Test these operators on the image I_2.

The morphological reconstruction function to use is binary_propagation in the module ndimage.morphology.

16.5. Cleaning of the image of cells

?

1. Threshold the image of cells (Fig. 16.1d).
2. Process the resulting binary image with the 3 cleaning processes of the previous question.

16.6. Python correction

```
from scipy import ndimage, misc
import numpy as np

import matplotlib.pyplot as plt
```

16.6.1. Elementary operators

First of all, one have to create a structuring element. The following function can be used to create a circular structuring element. A square can also be used.

```
def disk(radius):
    # defines a circular structuring element with radius given by 'radius'
    x = np.arange(-radius, radius+1, 1);
    xx, yy = np.meshgrid(x, x);
    d = np.sqrt(xx**2 + yy**2);
    return d<=radius;
```

The following code presents basic mathematical morphology operations. First of all, declare the structuring element.

```
# read binary image (and ensure binarization)
B = imageio.imread("B.jpg");
B = B>100;
# Structuring element
square = np.ones((5,5));
```

Erosion and dilation are the two elementary function of mathematical morphology.

```
# Erosion
Bsquare_erode = ndimage.morphology.binary_erosion(B, structure=square);
plt.subplot(231);
plt.imshow(Bsquare_erode); plt.title("erosion")
imageio.imwrite('erosion.png', Bsquare_erode);
# Dilation
Bsquare_dilate = ndimage.morphology.binary_dilation(B, structure=square);
plt.subplot(232);
plt.imshow(Bsquare_dilate); plt.title("dilation")
imageio.imwrite('dilation.png', Bsquare_dilate);
```

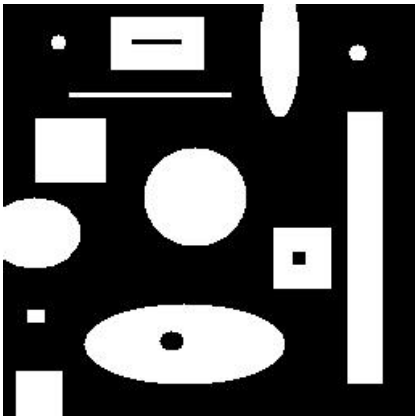
Opening and closing are a combination of the two previous functions.

```
# Opening
Bsquare_open = ndimage.morphology.binary_opening(B, structure=square);
plt.subplot(233);
plt.imshow(Bsquare_open);plt.title("opening")
imageio.imwrite('open.png', Bsquare_open);
# Closing
Bsquare_close = ndimage.morphology.binary_closing(B, structure=square);
plt.subplot(234);
plt.imshow(Bsquare_close);plt.title("closing")
imageio.imwrite('close.png', Bsquare_close);
```

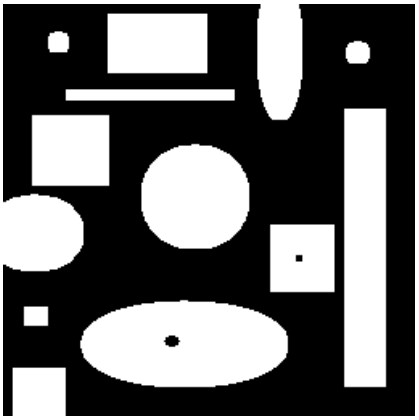
The results are presented in Fig. 16.3.

Figure 16.3: Basic mathematical morphology operations.

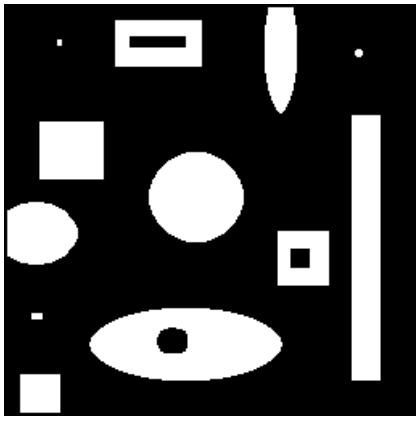
(a) Original image.



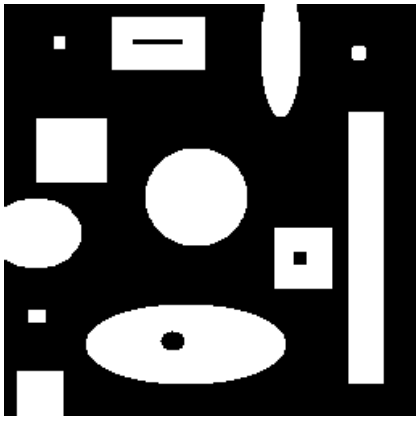
(b) Dilation.



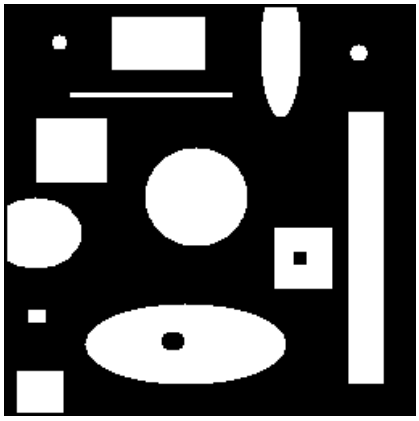
(c) Erosion.



(d) Opening.



(e) Closing.



16.6.2. Morphological reconstruction

The algorithm of morphological reconstruction is coded like this in python:

```
def reconstruct(image, mask):
    # should be binary images
    M = np.minimum(mask, image);

    area = ndimage.measurements.sum(M);
    s=0

    se = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]]);
    while (area != s):
        s = area;
        M = np.minimum(image, ndimage.morphology.binary_dilation(M, structure=se));
        area = ndimage.measurements.sum(M);

    return M
```

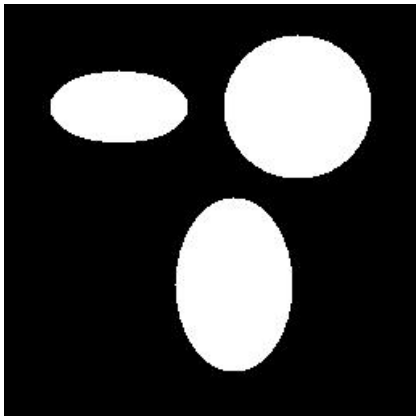
The Fig. 16.4 illustrates the morphological recontruction.

```
A=imageio.imread('A.jpg');
A = A > 100;
M=imageio.imread('M.jpg');
M = M > 100;
# reconstruction de A par M
AM=reconstruct(A, M);

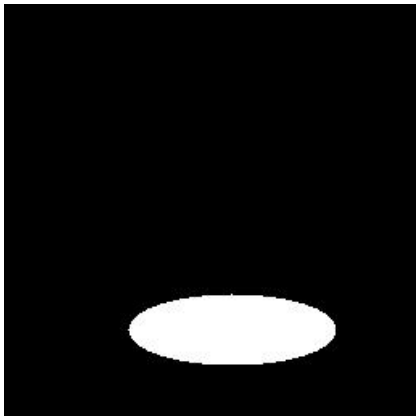
# display results
plt.subplot(1, 3, 1);
plt.imshow(A);
plt.subplot(1, 3, 2);
plt.imshow(M);
plt.subplot(1, 3, 3);
plt.imshow(AM);
plt.show();
```

Figure 16.4: Morphological reconstruction.

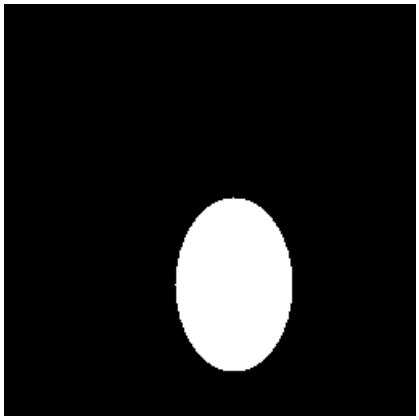
(a) Image A.



(b) Image M.



(c) AM=reconstruct(A, M);.



16.6.3. Operators by reconstruction

Remove objects touching the borders

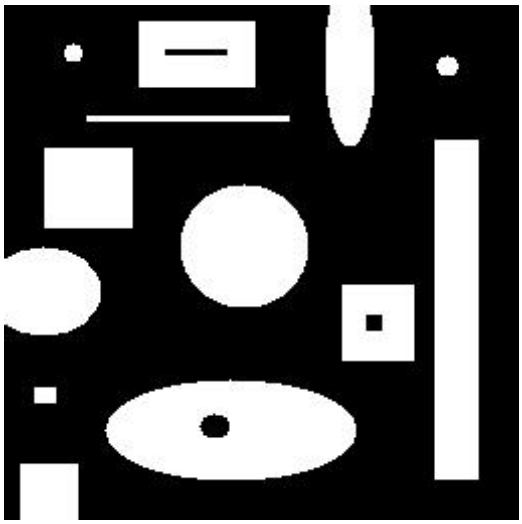
The Fig. 16.5 illustrates the suppression of objects touching the borders of the image. If $\mathcal{B}$ represents the border of the image (create an array of the same size as the image, with zeros everywhere and ones at the sides), then

this operation is defined by:

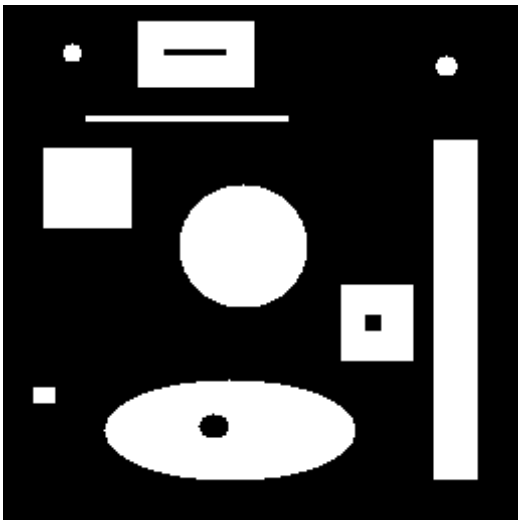
$$\text{killBorders}(I) = I \setminus \rho_I(\mathcal{B})$$

Figure 16.5: Suppress border objects.

(a) Original image.



(b) Objects removed.



```
def killBorders(A):
    # remove cells touching the borders of the image
    m, n = A.shape
    M = np.zeros((m,n));
    M[0,:] = 1;
    M[m-1,:] = 1;
    M[:,0] = 1;
    M[:,n-1] = 1;
    M = reconstruct(A, M);
    return A-M
```

Remove small objects

This is illustrated in Fig. 16.6. It consists in an erosion followed by a reconstruction. The structuring element used in the erosion defines the objects considered as "small".

$$\text{killSmall}(I) = \rho_I(\varepsilon(I))$$

```
def killSmall(A, n):
    # destroy small objects (size smaller than n)
    se = np.ones((n, n));
    M = ndimage.morphology.binary_erosion(A, structure=se);
    return reconstruct(A, M);
```

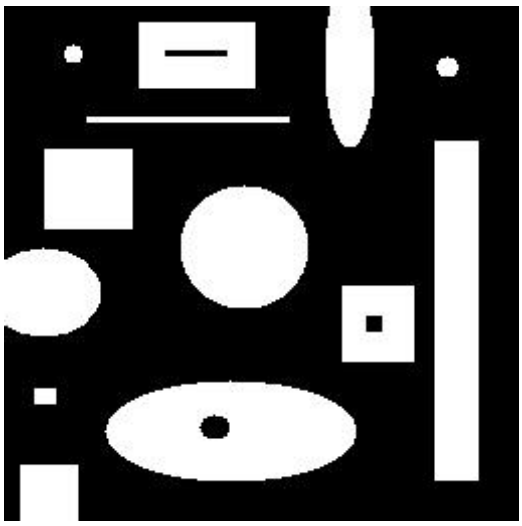
Close holes in objects

This is illustrated in Fig. 16.7. The operation is given by the following equation, with $\mathcal{B}$ the border of image I, and X^C denoting the complementary of set X:

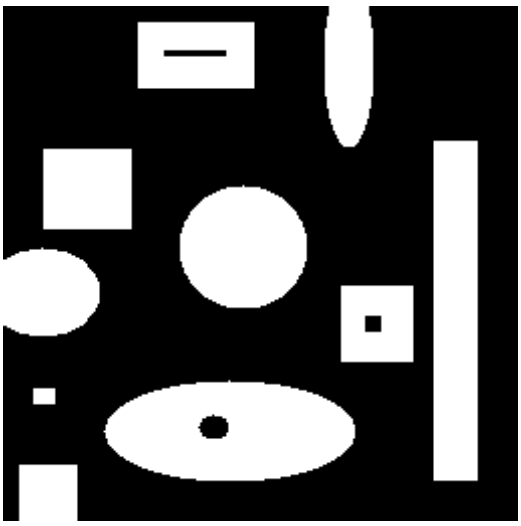
$$\text{removeHoles(I)} = \{\rho_{I^C}(\mathcal{B})\}^C$$

Figure 16.6: Small objects removal.

(a) Original image.



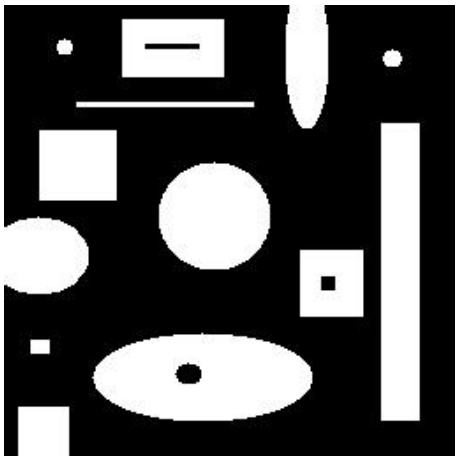
(b) Small objects are removed.



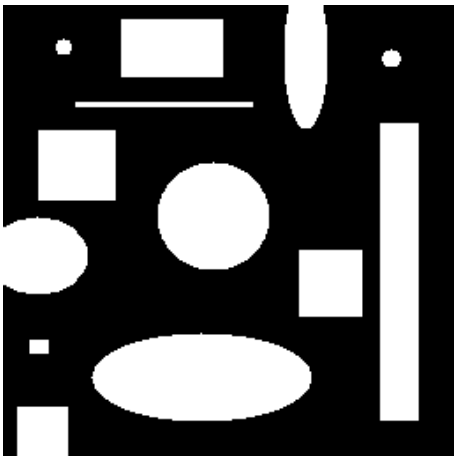
```
def closeHoles(A):
    # close holes in objects
    Ac = ~A;
    m,n = A.shape;
    M = np.zeros((m,n));
    M[0,:] = 1;
    M[m-1,:] = 1;
    M[:,0] = 1;
    M[:,n-1] = 1;
    M = reconstruct(Ac, M);
    return ~M
```

Figure 16.7: Hole filling.

(a) Original image.



(b) Holes in objects are closed.



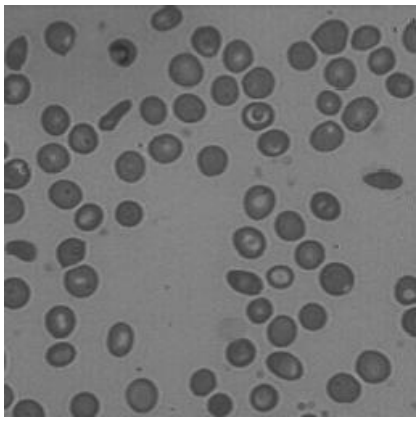
16.6.4. Application

The application shown in Fig. 16.8 presents the segmentation of blood cells after removing small cells, closing holes and removing the cells touching the borders of the image.

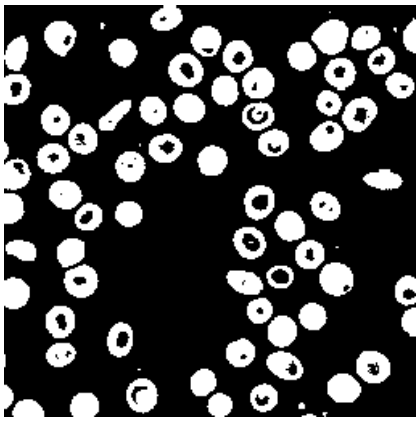
```
cells = imageio.imread('cells.jpg')<98;
imageio.imwrite('cellsbw.png', cells);
B = closeHoles(cells);
B = killBorders(B);
B = killSmall(B, 5);
plt.subplot(1,2,1);
plt.imshow(cells);
plt.subplot(1,2,2);
plt.imshow(B);
plt.title('clean image')
plt.show()
imageio.imwrite('clean.png', B);
```

Figure 16.8: Segmentation of the cells.

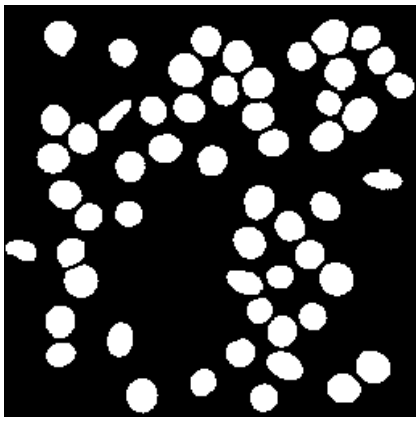
(a) Original image.



(b) Binarization.



(c) Final segmentation of the cells.



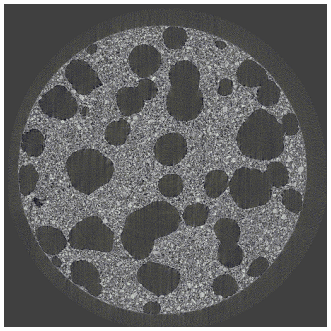
★★★ 17 Morphological Geodesic Filtering

This tutorial aims to test different morphological filters and particularly the geodesic ones (using reconstruction) for gray-level images.

The different processes will be applied on the following images:



(a) Cameraman



(b) 2-D section of a cement paste (X-ray tomography)

17.1. Morphological centre

The morphological centre is an auto-dual filter using a family of operators $\{\psi_i\}_i$:

$$C(f) = (f \vee (\wedge\{\psi_i(f)\})) \wedge (\vee\{\psi_i(f)\}) \tag{17.1}$$

?

- Implement this transformation with the family $\{\gamma\phi\gamma, \phi\gamma\phi\}$ where γ denotes the morphological opening and ϕ the morphological closing.
- Test this operator by varying the size of the structuring element.
- Add a 'salt and pepper' noise to the 'cameraman' image and compare the morphological center with the median filtering.
- Try to reduce the noisy image 'cement paste'.

17.2. Alternate sequential filters (ASF)

ASF can be defined from a family of openings and closings:

$$N_i(f) = \gamma_i\phi_i \circ \gamma_{i-1}\phi_{i-1} \ldots \gamma_2\phi_2 \circ \gamma_1\phi_1(f) \tag{17.2}$$
$$M_i(f) = \phi_i\gamma_i \circ \phi_{i-1}\gamma_{i-1} \ldots \phi_2\gamma_2 \circ \phi_1\gamma_1(f) \tag{17.3}$$

where γ_k (resp. ϕ_k) denotes the opening (resp. closing) with a structuring element of size k.

?

- Implement these two operators.
- Test these transformations on the noisy image by varying the parameter i of the filter.

17.3. Reconstruction filters

The geodesic dilation of size 1 and n are respectively defined as:

$$\delta_f(g) = \wedge(\delta_{B_1}(g), f) \quad (17.4)$$
$$\delta_f^n(g) = \delta_f(\delta_f \ldots (\delta_f(g))) \quad (17.5)$$

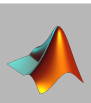
where δ_{B_1} denotes the morphological dilation with a disk of radius 1 as structuring element. The opening ($\gamma_k^{rec}(f)$) and closing ($\phi_k^{rec}(f)$) by reconstruction are then defined as:

$$\gamma_k^{rec}(f) = \vee\{\delta_f^n(\epsilon_{B_k}(f)), n > 0\} \quad (17.6)$$
$$\phi_k^{rec}(f) = M - \gamma_k^{rec}(M - f) \quad (17.7)$$

where ϵ_{B_k} denotes the morphological erosion with a disk B of radius k as structuring element.

?

- Implement these two operators $\gamma_k^{rec}(f)$ and $\phi_k^{rec}(f)$.
- Test these transformations by varying the parameter k of the filter.
- Implement and test (on the noisy image) the filters of morphological center and ASF using the geodesic operators. Compare with the classical ones.



17.4. Python correction





```
from skimage.util import random_noise, img_as_ubyte
import matplotlib.pyplot as plt
from skimage import morphology as m
from skimage import filters
import numpy as np
from skimage.io import imread, imsave
```

17.4.1. Morphological center

The noisy image is obtained with the function random_noise from skimage.util.



```
L = imread('camera.png')
# the noisy image is converted back to 8 bits, as the duality
# used by 'closerec' requires 8 bits images
A = img_as_ubyte(random_noise(L, mode='s&p', amount=.04))
```

Following the definition, the morphological center is obtained with the following code, and illustrated in Fig.17.1:



```
def morphoCenter(I, c, o, footprint=m.disk(1)):
    """
    c, o: closing and opening operators
    (classical or geodesic ones)
    """
    coc = c(o(c(I, footprint), footprint), footprint)
    oco = o(c(o(I, footprint), footprint), footprint)
    cMin = np.minimum(oco, coc)
    cMax = np.maximum(oco, coc)
    F = np.minimum(np.maximum(I, cMin), cMax)
    return F

B = morphoCenter(A, m.closing, m.opening)
Bmed = filters.median(A, footprint=np.ones((3, 3)))
```

Figure 17.1: Morphological center compared to the classical median filter.

(a) Noisy image.



(b) Median filter of size 3×3.



(c) Morphological center.



17.4.2. Alternate sequential filters

The order of these filters are often chosen empirically. The results are illustrated in Fig.17.2.

```
def asf_n(I, order=3):
    F = I.copy()
    for r in np.arange(1, order+1):
        se = m.disk(r)
        F = m.opening(m.closing(F, se), se)
    return F
```

```
def asf_m(I, order=3):
    F = I.copy()
    for r in np.arange(1, order+1):
        se = m.disk(r)
        F = m.closing(m.opening(F, se), se)
    return F
```

Figure 17.2: Alternate Sequential Filters compared to original image.

(a) Original image.



(b) ASF of order 3, starting with a closing operation (denoted N).



(c) ASF of order 3, starting with an opening operation (denoted M).



17.4.3. Geodesic reconstruction filters

These two functions are simply implemented using erosion and reconstruction operators. Notice the duality property, that is used to code closerec. In this example, 8 bits images (unsigned) are considered, and the results are illustrated in Fig.17.3.

```
def openrec(I, footprint=m.disk(1)):
    B = m.erosion(I, footprint)
    F = m.reconstruction(B, I)
    return F.astype(I.dtype)
```

```
def closerec(I, footprint=m.disk(1)):
    F = 255-openrec(255-I, footprint)
    return F
```

Figure 17.3: Opening and closing by reconstruction.

(a) Original image.



(b) Opening by reconstruction.



(c) Closing by reconstruction.



17.4.4. ASF by reconstruction

This is an example of a 3rd order alternate sequential filter, illustrated in Fig.17.4.

```
def asfrec(I, order=3):
    A = I.copy()
    for r in np.arange(1, order+1):
        se = m.disk(r)
        A = closerec(openrec(A, se), se)
    return A
```

17.4.5. Morphological center by reconstruction

Morphological center by reconstruction replaces the opening and closing operations by their equivalent by reconstruction (see Fig.17.5).

Figure 17.4: Alternate Sequential Filtering by reconstruction.

(a) Original image.



(b) ASF of order 3 by reconstruction.



(c) Noisy image.



```
def centerrec(I, footprint=m.disk(1)):
    """
    """
    B = morphoCenter(I, closerec, openrec, footprint)
    return B
```

Figure 17.5: Morphological center by reconstruction.

(a) Original image.



(b) Center by reconstruction.



(c) Noisy image.



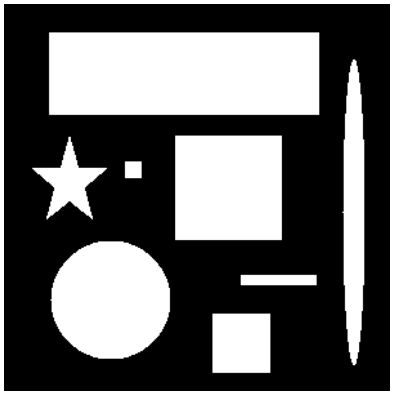
18 Morphological Attribute Filtering

This tutorial aims to test the attribute morphological filters for gray-level images. The objective of such filters is to remove the connected components of the level sets which do not satisfy a specific criterion (area, elongation, convexity...).

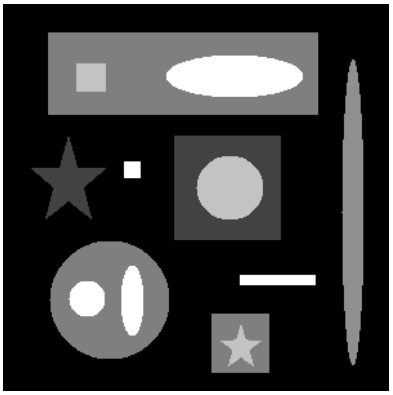
Fig.18.1 are presented some results of attribute filtering applied on the grayscale image.

Figure 18.1: Attribute filtering examples.

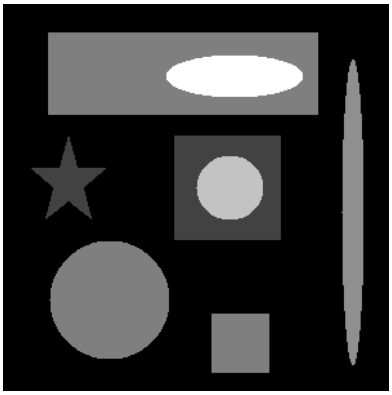
(a) Binary image.



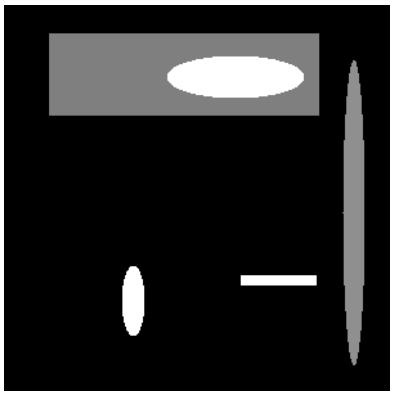
(b) Grayscale image.



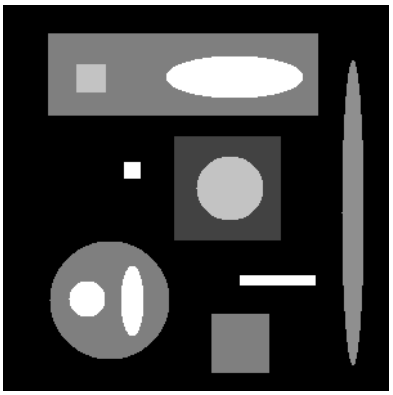
(c) Filtering by area.



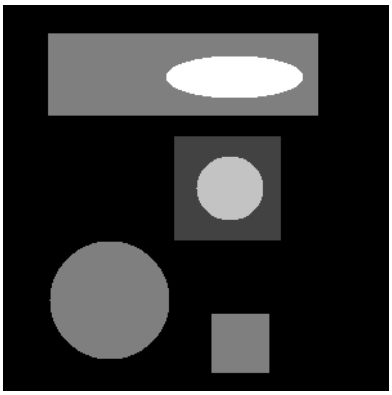
(d) Filtering by elongation.



(e) Filtering by convexity.



(f) Filtering objects larger than a square.



A *binary connected opening* Γ_x transforms a binary image f given a pixel x, by keeping the connected component that contains x and removing all the others.
Binary trivial opening Γ_T operates on a given connected component X according to an increasing criterion T applied to the connected component. If the criterion is satisfied, the connected component is preserved, otherwise it is removed according to:

$$\Gamma_T(X) = \begin{cases} X & \text{if } T(X) = \text{true} \\ \emptyset & \text{if } T(X) = \text{false} \end{cases} \tag{18.1}$$

In general, one or more features of the connected component, on which the filter is applied, are compared to a given threshold defined by the rule.
Binary attribute opening Γ^T, given an increasing criterion T, is defined as a binary trivial opening applied on all the connected components of F. This can be formally represented as:

$$\Gamma^T(F) = \bigcup_{x \in F} \Gamma_T(\Gamma_x(F)) \tag{18.2}$$

If the criterion T evaluated in the transformation is not increasing, e.g., when the computed attribute is not dependent itself on the scale of the regions (e.g. shape factor, orientation, etc.), the transformation also

becomes not increasing. Even if the increasingness property is not fulfilled, the filter remains idempotent and anti-extensive. For this reason, the transformation based on a non-increasing criterion is not an opening, but a thinning. In this case, one speak about a *binary attribute thinning*.

Attribute openings and thinnings introduced for the binary case can be extended to grayscale images employing the threshold decomposition principle. A grayscale image can be represented by thresholding the original image at each of its graylevels. Then, the binary attribute opening can be applied to each binary image and the *grayscale attribute opening* is given by the maximum of the results of the filtering for each pixel and can be mathematically represented as:

$$\gamma^T(f)(x) = \max\{k; x \in \Gamma^T(Th_k(f))\} \tag{18.3}$$

where $Th_k(f)$ is the binary image obtained by thresholding f at graylevel k.

The extension to grayscale of the binary attribute thinning is not straightforward and not unique. For example, a possible definition of a *grayscale attribute thinning* can be analogously defined as Equation (3). However, other definitions are possible, according to the filtering rule considered in the analysis (max, min, subtractive, direct...). Again, if the criterion T is increasing (i.e. the transformation is an opening), all the strategies lead to the same result.

Attribute openings are a family of operators that also includes openings by reconstruction. If we consider a binary image with a connected component X and the increasing criterion "the size of the largest square enclosed by X must be greater than λ", the result of the attribute opening is the same as applying an opening by reconstruction with a squared structuring element of size λ.

18.1. Binary attribute opening/thinning

We are going to implement and test some binary attribute filters according to different criteria (area, elongation, convexity).

?

- Regarding the area criterion, implement the associated binary attribute opening. Test on the binary image.
- Regarding the elongation criterion, implement the associated binary attribute thinning (non-increasing criterion). Test on the binary image.



Code your own function bwFilter based on regionprops from skimage.measure.

18.2. Grayscale attribute opening/thinning

We are going to extend the binary filters to grayscale images.

?

- Implement the decomposition operator into binary sections.
- Conversely, implement the reconstruction operator from binary sections.
- Implement and test the grayscale attribute filters according to the following criteria: area, elongation, convexity.



18.3. Python correction





```
from skimage import morphology as m
from scipy import misc
import matplotlib.pyplot as plt
from skimage.measure import regionprops
from skimage.measure import label
import numpy as np
```

18.3.1. Binary attribute filtering

If bw is a binary image, the different attribute are evaluated with regionprops and filtered according to the given upper and lower thresholds.



```
def bwFilter(bw, attribute, thresholds):
    """
    binary filtering according to attribute
    bw: binary image
    attribute: string representing attribute, defined by regionprops
    thresholds: threshold values, inside which objects are removed
    returns binary image
    """
    F = bw.copy();
    L = label(bw);
    for region in regionprops(L):
        a = getattr(region, attribute);
        if a < thresholds[1] and a>= thresholds[0]:
            F[L==region.label] = False;
    return F;
```

18.3.2. Grayscale filtering

The grayscale filtering is the previous binary filtering process applied to all level-sets of the original image.

```python
def grayFilter(I, attribute, thresholds):
    """
    grayscale image filtering by attribute
    for 8 bits unsigned images
    I: original grayscale image (N, M) ndarray
    attribute: string representing attribute, defined by regionprops
    thresholds: threshold values, inside which objects are removed
    returns grayscale filtered image
    """
    N, M = I.shape
    F = np.zeros((N, M, 256));
    for s in range(256):
        F[:,:,s] = s * bwFilter(I>=s, attribute=attribute, thresholds=thresholds);

    # reconstruction
    R = np.amax(F, axis=2);
    return R;
```

Then, for each level, the binary set is filtered by some attribute, and the resulting image is reconstructed by taking the maximum value on all levels.

The shape filtering process do not take the results from the regionprops function. The shape is filtered by using the morphological function of reconstruction.

```python
def shapeFilter(I, selem=m.square(25)):
    """
    image filtering when attribute is a shape of a given size, defined by selem
    I: grayscale image
    selem: structuring element
    returns: grayscale filtered image
    """
    N, M = I.shape
    F = np.zeros((N, M, 256));
    for s in range(256):
        F[:,:,s] = s * m.reconstruction(m.opening(I>=s, selem=selem), I>=s);

    # reconstruction
    R = np.amax(F, axis=2);
    return R;
```

The examples are generated via the following code:

```python
F = bwFilter(I>50, 'area', (0,5000));

F = grayFilter(I, 'area', (0,1000));

F = grayFilter(I, 'eccentricity', (0, 0.75));

F = grayFilter(I, 'solidity', (0, 0.75));

F = shapeFilter(I);
```

Results are illustrated in Fig.18.1.

19 Morphological skeletonization

This tutorial aims to skeletonize objects with specific tools from mathematical morphology (thinning, maximum ball…).

The different processes will be applied on the following image:



Informations

In this tutorial, you will use the modules scipy.ndimage for the operations of mathematical morphology, and skimage for dealing with images.

19.1. Hit-or-miss transform

The hit-or-miss transformation enables specific pixel configurations to be detected. Based on a pair of disjoint structuring elements $T = (T^1, T^2)$, this transformation is defined as:

$$\eta_T(X) = \{x, T^1_x \subseteq X, T^2_x \subseteq X^c\} \quad (19.1)$$
$$= \epsilon_{T^1}(X) \cap \epsilon_{T^2}(X^c) \quad (19.2)$$

where $\epsilon_B(X)$ denotes the erosion of X using the structuring element B.

?

1. Implement the hit-or-miss transform.
2. Test this operator with the following pair of disjoint structuring elements:

$$\begin{array}{|c|c|c|}\hline +1 & +1 & +1 \\ \hline 0 & +1 & 0 \\ \hline -1 & -1 & -1 \\ \hline \end{array} \quad \left(T^1 = \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}, \quad T^2 = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \right)$$

where the points with $+1$ (resp. -1) belong to T^1 (resp. T^2).

19.2. Thinning and thickening

Using the hit-or-miss transform, it is possible to make a thinning or thickening of a binary object in the following way:

$$\theta_T(X) = X \backslash \eta_T(X) \quad (19.3)$$
$$\chi_T(X) = X \cup \eta_T(X^c) \quad (19.4)$$

These two operators are dual in the sense that $\theta_T(X) = (\chi_T(X^c))^c$.

?

1. Implement these two transformations.
2. Test these operators with the previous pair of structuring elements.

19.3. Topological skeleton

By using the two following pairs of structuring elements with their rotations (90°) in an iterative way (8 configurations are thus defined), the thinning process converges to a resulting object which is homothetic (topologically equivalent) to the initial object.

+1	+1	+1
0	+1	0
−1	−1	−1

0	+1	0
+1	+1	−1
0	−1	−1

?

1. Implement this transformation (the convergence has to be satisfied).
2. Test this operator and comment.

19.4. Morphological skeleton

A ball $B_n(x)$ with center x and radius n is maximum with respect to the set X if there exists neither indice k nor centre y such that:

$$B_n(x) \subseteq B_k(y) \subseteq X$$

In this way, the morphological skeleton of a set X is constituted by all the centers of maximum balls. Mathematically, it is defined as:

$$S(X) = \bigcup_r \epsilon_{B_r(0)}(X) \backslash \gamma_{B_1(0)}(\epsilon_{B_r(0)}(X)) \tag{19.5}$$

?

1. Implement this transformation.
2. Test this operator and compare it with the topological skeleton.



19.5. Python correction





```
from scipy import ndimage
import numpy as np

import imageio
import matplotlib.pyplot as plt
```

19.5.1. Hit or miss transform

The hit-or-miss transform is illustrated in Fig.19.1.



```
def hitormiss(X, T):
    """
    hit or miss transform
    X: binary image
    T: structuring element (values -1, 0 and 1)

    return: result of hit or miss transform (binary image)
    """
    T1 = (T==1);
    T2 = (T==-1);
    E1 = ndimage.morphology.binary_erosion(X, T1);
    E2 = ndimage.morphology.binary_erosion(np.logical_not(X), T2);
    B = np.minimum(E1, E2);
    return B;
```

Figure 19.1: Elementary functions

(a) Hit or miss (intensities are inverted).



(b) Thinning.



(c) Thickening.



19.5.2. Thinning and thickening

The code is split into 2 elementary operations of thinning and thickening, so that the thinning consists in iterating this operation. The thickening operation is obtained by thinning the complementary set.



```
def elementary_thinning(X, T):
    """
    thinning function
    X: binary image
    T: structuring element (values -1, 0 and 1)

    return: result of thinning
    """
    B = np.minimum(X, np.logical_not(hitormiss(X, T)));
    return B;
```



```
def elementary_thickening(X, T):
    """
    thickening function
    X: binary image
    T: structuring element (values -1, 0 and 1)

    return: result of thickening
    """
    B = not(elementary_thinning(not(X), T));
    return B;
```



```
def thinning(X, TT):
    """
    morphological thinning
    TT is a configuration of 8 pairs of structuring elements
    """
    for T in TT:
        X = elementary_thinning(X, T);

    return X;
```

19.5.3. Skeletons

The topological skeleton is the iteration of the operation of thinning, for structuring elements defined like:



```
TT = [];
TT.append(np.array([[-1,-1,-1],[0,1,0],[1,1,1]]));
TT.append(np.array([[0,-1,-1],[1,1,-1],[0,1,0]]));
TT.append(np.array([[1,0,-1],[1,1,-1],[1,0,-1]]));
TT.append(np.array([[0,1,0],[1,1,-1],[0,-1,-1]]));
TT.append(np.array([[1,1,1],[0,1,0],[-1,-1,-1]]));
TT.append(np.array([[0,1,0],[-1,1,1],[-1,-1,0]]));
TT.append(np.array([[-1,0,1],[-1,1,1],[-1,0,1]]));
TT.append(np.array([[-1,-1,0],[-1,1,1],[0,1,0]]));
```



```
def topological_skeleton (X, TT):
    """
    Topological skeleton: preserves topology
    X: binary image to be transformed
    TT: set of pairs of structuring elements
    return skeleton
    """
    B = np.logical_not(np.copy(X));
    while not(np.all(X == B)):
        B = X;
        X = thinning(X, TT);
    return B;
```

The topological skeleton is the iteration of the thinning with structuring elements in all 8 directions. It has the property of preserving the topology of the discrete structures, contrary to the morphological skeleton (see Fig.19.2). The morphological skeleton does not preserve the connexity of the branches, but it can be used to reconstruct the original image. Pay attention to the construction of structuring elements, which should be homothetic ($B_r = \underbrace{B_1 \oplus \cdots \oplus B_1}_{r \text{ times}}$).



```
def morphological_skeleton(X):
    """
    morphological skeleton
    X: binary image
    using disk structuring elements

    in order to perform the reconstruction from the skeleton, one has to store
    the value of S for each size of structuring element

    return: S, morphological skeleton, grayscale image
    """
     strel_size = -1;
    pred = True;
    S = np.zeros(X.shape);
    se = ndimage.generate_binary_structure(2, 1);
    E = np.copy(X);
    while pred:
        strel_size +=1;
        E = ndimage.morphology.binary_erosion(X, se);
        if np.all(E==0):
            pred = False;
        D = ndimage.morphology.binary_dilation(E, se);
        S = np.maximum(S, (strel_size +1)*np.minimum(X, np.logical_not(D)));
        X = E;
    return S;
```

```python
def reconstruction_skeleton (S):
    """
    Reconstruction of the original image from the morphological skeleton
    S: Skeleton, as constructed by morphological_skeleton

    return: original image I
    """
    X = np.zeros(S.shape).astype("bool");
    n = np.max(S);
    se = ndimage.generate_binary_structure (2, 1);

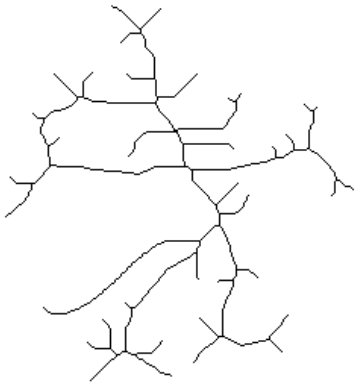
    for strel_size in range(int(n)):
        Sn = S == strel_size +1;

        # this is for preserving homothetic structuring elements
        for k in range( strel_size ):
            Sn = ndimage.morphology.binary_dilation(Sn, se);

        X = np.maximum(X, Sn);
    return X;
```

Figure 19.2: Skeletons. The topology is not preserved in the morphological skeleton, but it can be used to reconstruct the original image. The intensities are inverted in order to facilitate the display.

(a) Topological skeleton.



(b) Morphological skeleton.



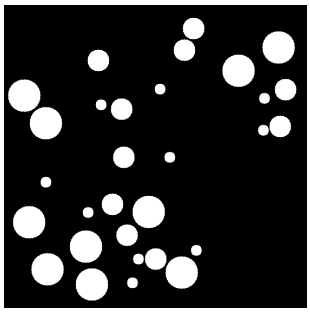
⋆ 20 Granulometry

This tutorial aims to compute specific image measurements on digital images. It consists in determining the size distribution of the 'particles' included in the image to be analyzed.

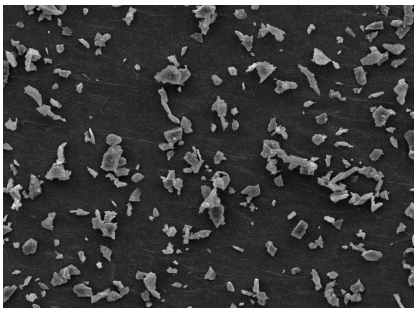
The image measurements will be realized on a simulated image of disks and an image of silicon carbide powder acquired by Scanning Electron Microscopy (SEM, Fig. 20.1).

Figure 20.1: These images are used for computing a granulometry of objects by mathematical morphology.

(a) Simulated image.



(b) Real image of powder, by SEM.



20.1. Morphological granulometry

20.1.1. Introduction to mathematical morphology

The tutorial 16 introduces the basic operations of the mathematical morphology, as well as the morphological reconstruction. More informations can be found in [41].

With Python, the useful functions are binary_opening of the module scipy.ndimage for the morphological opening, binary_propagation for the geodesic reconstruction. For counting the number of objects, one can use measurements.label for a labelling and counting algorithm.

20.1.2. Granulometry

?

- Load and visualize the image 'simulation' Fig.20.1.
- Generate n images by applying morphological openings (with reconstruction) to the binary image with the use of homothetic structuring elements of increasing size: $1 < \cdots < n$.
- Compute the morphological granulometry expressed in terms of surface area density. It consists in calculating the specific surface area of the grains in relation with the size n of the structuring element.
- Compute the morphological granulometry expressed in terms of number density. It consists in calculating the specific number of grains in relation with the size n of the structuring element.
- Compare and discuss.



Use generate_binary_structure and iterate_structure from python module scipy.ndimage to define homothetic structuring elements.

20.2. Real application

?

- Load and visualize the image 'powder' of Fig. 20.1b.
- Realize the image segmentation step. I can simply consist in a binarization by a global threshold, followed by hole filling. Noise can also be removed my mathematical morphology opening.
- Compute the morphological granulometry (surface area, number).
- What can you conclude?



20.3. Python correction



20.3.1. Simulation

The construction of an homothetic structuring element is necessary for computing a granulometry (through the function iterate_structure .



```
def granulometry(BW, T=35):
    # total original area
    A = ndimage.measurements.sum(BW);

    # number of objects
    label , N = ndimage.measurements.label(BW);

    area=np.zeros ((T,) , dtype=np. float ) ;
    number=np.zeros((T,) , dtype=np. float ) ;

    """
    Warning: the structuring elements must verify B(n) = B(n-1) o B(1) .
    """
    se = ndimage.generate_binary_structure (2, 1);
    for i in np.arange(T):
        SE = ndimage. iterate_structure (se, i-1);
        m = ndimage.morphology.binary_erosion(BW, structure=SE);
        G = ndimage.morphology.binary_propagation(m, mask=BW);
        area[i]=100*ndimage.measurements.sum(G)/A
        label , n = ndimage.measurements.label(G);
        number[i] = 100*n/N; # beware of integer division

    plt . figure ()
    plt . plot (area , label ='Area')
    plt . plot (number, label ='Number')
    plt .legend ()
    #plt . savefig ("granulo_poudre1.pdf") ;
    plt .show()

    plt . figure () ;
    plt . plot (-np. diff (area) , label ='Area derivative ') ;
    plt . plot (-np. diff (number), label ='Number derivative') ;
    plt .legend ()
    #plt . savefig ("granulo_poudre2.pdf") ;
    plt .show()
```

The results are shown in Fig. 20.2, generated by the following code:

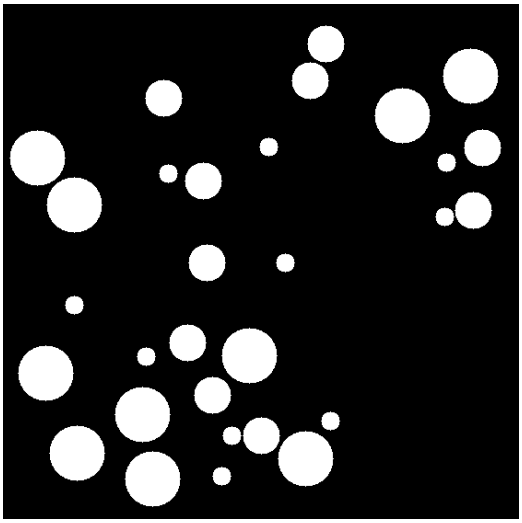


```
## Granulometry of synthetic image
# read binary simulated image, normalize it
I = imageio.imread("simulation .png") /255;
I = I [:,:,2]>.5;
plt . figure () ;
plt .imshow(I);
plt .show();

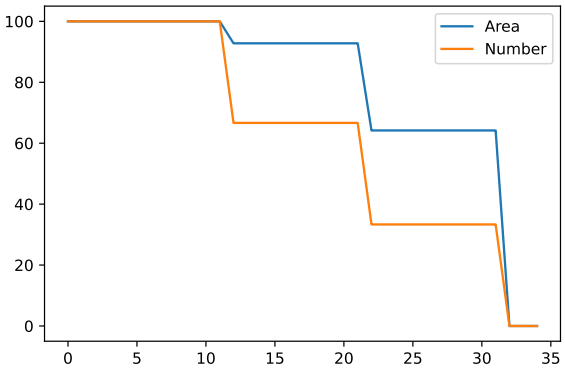
granulometry(I, 35) ;
```

Figure 20.2: Granulometry on simulated image.

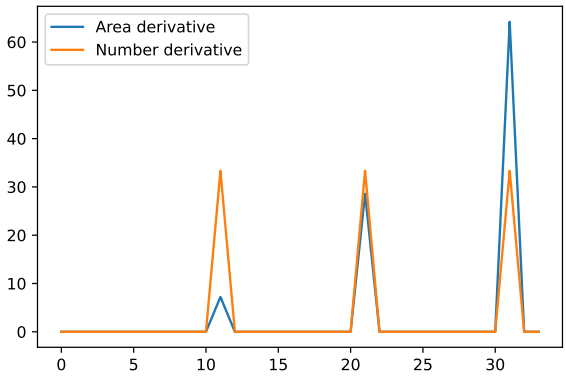
(a) Simulated image of disks.



(b) Granulometry in number and area.



(c) Derivatives.



20.3.2. Powder image and segmentation

First of all, the image must be binarized, i.e. segmented. The following code is a proposition of segmentation, leading to the result presented in Fig. 20.3c.

```python
## Granulometry of real image
I = imageio.imread("poudre.bmp");

# segmentation
BW = I>74;
BW = ndimage.morphology.binary_fill_holes(BW);

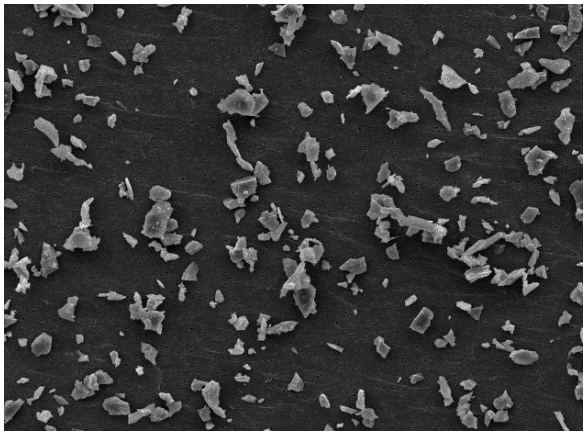
# suppress small objects
se = ndimage.generate_binary_structure (2, 1);
m = ndimage.morphology.binary_opening(BW);
# opening by reconstruction
BW=ndimage.morphology.binary_propagation(m, mask=BW);

imageio.imwrite("segmentation.png", BW);
plt.imshow(BW)
plt.show()

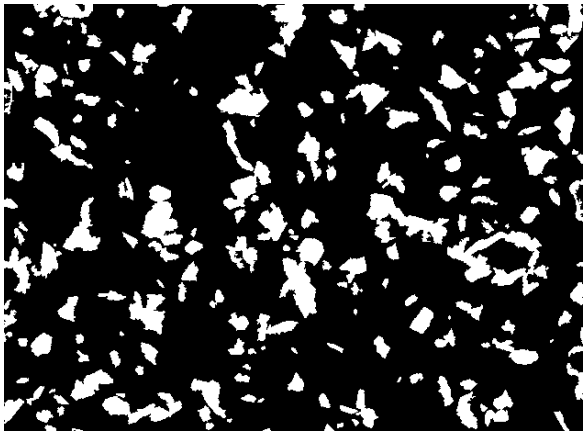
granulometry(BW, 20);
```

Figure 20.3: Results of granulometry analysis of original image from (a).

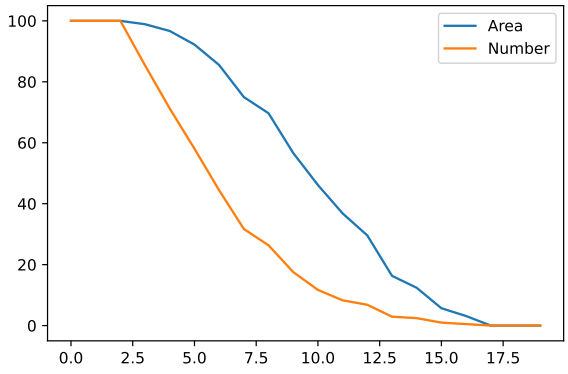
(a) Original image of powder.



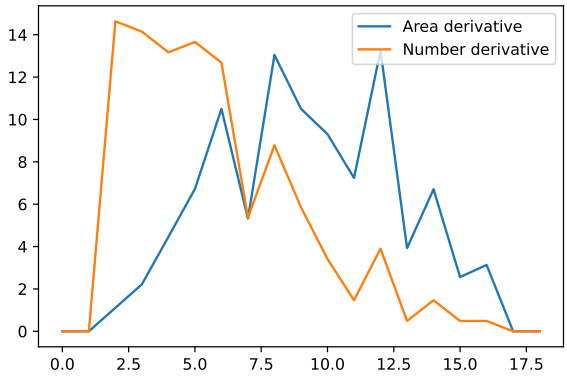
(b) Segmentation of image of (a).



(c) Granulometry.



(d) Derivative.

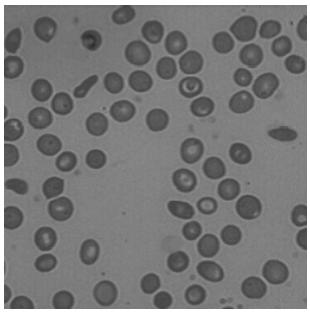


Part VII Segmentation

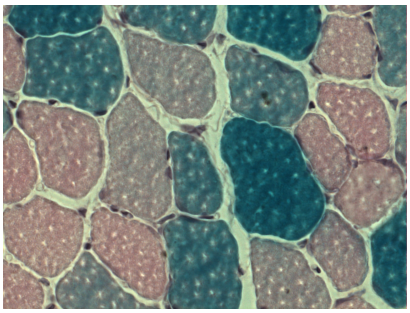
⋆ 21 Histogram-based image segmentation

This tutorial aims to implement some image segmentation methods based on histograms (thresholding and "k-means" clustering).

The different processes will be applied on the following images:



(a) Cells.



(b) Muscle cells, author: Damien Freyssenet, University Jean Monnet, Saint-Etienne, France.

21.1. Manual thresholding

The most simple segmentation method is thresholding.

?

- Visualize the histogram of the grayscale image 'cells'.
- Make the segmentation with a threshold value determined from the image histogram.

21.2. k-means clustering

Let $X = \{x_i\}_{i\in[1;n],n\in\mathbb{N}}$ be a set of observations (the points) in $\mathbb{R}^d$. The k-means clustering consists in partitioning X in k $(k < n)$ disjoint subsets $\tilde{S}$ such that:

$$\tilde{S} = \arg\min_{S=\{S_i\}_{i\leq k}} \sum_{i=1}^{k} \sum_{x_j\in S_i} \|x_j - \mu_i\|^2 \tag{21.1}$$

where μ_i is the mean value of the elements in S_i. The k-means algorithm is iterative. From a set of k initial elements $\{m_i^{(1)}\}_{i\in[1;k]}$ (randomly selected), the algorithm iterates the following (t) steps:

- Each element of X is associated to an element m_i according to a distance criterion (computation of a Voronoi partition):

$$S_i^{(t)} = \left\{\mathbf{x}_j : \|\mathbf{x}_j - \mathbf{m}_i^{(t)}\| \leq \|\mathbf{x}_j - \mathbf{m}_{i^*}^{(t)}\|, \forall i^* \in [1;k]\right\} \tag{21.2}$$

- Computation of the new mean values for each class:

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\mathbf{x}_j\in S_i^{(t)}} \mathbf{x}_j \tag{21.3}$$

where $|S_i^{(t)}|$ is the number of elements of $S_i^{(t)}$.

Although interesting, the goal of this tutorial is not to code the k-means algorithm in a general case. We will code it for grayscale images, and use builtin functions for other cases with higher dimensions.

21.3. Grayscale image, $k = 2$ in one dimension

The objective is to binarize image image 'cells', which is a grayscale image. The set X is defined by $X = \{I(p)\}$, for p being the pixels of the image I.

?

- Implement the algorithm proposed below (Alg. 4).
- Test this operator on the image 'cells'.
- Test another method of automatic thresholding (defined by Otsu in [27]).
- Compare the values of the thresholds (manual and automatic).

Data: Original image A
Data: Stop condition ε
Result: thresholded image
Initialize T_0, for example at $\frac{1}{2}(\max(A) + \min(A))$;
$done \leftarrow False$;
while *NOT done* **do**
  Segment the image A with the threshold value T;
  it generates two classes G_1 (intensities $\geq T$) and G_2 (intensities $< T$);
  Compute the mean values, denoted μ_1, μ_2, of the two classes G_1, G_2, respectively;
  Compute the new threshold value $T_i = \frac{1}{2}(\mu_1 + \mu_2)$;
  if $|T_i - T_{i-1}| < \varepsilon$ **then**
    $done \leftarrow True$
  end
end
Segment the image with the estimated threshold value.

Algorithm 4: K-means algorithm for automatic threshold computation of grayscale images.

For use with python, use the module skimage. filter and the function threshold_otsu.

21.4. Simulation example, $k = 3$ in two dimensions

The objective is to generate a set of 2-D random points (within $k = 3$ distinct classes) and to apply the k-means clustering for separating the points and evaluating the method (the classes are known!).

?

Write a function for generating a set of n random points (x_i, y_i) around a point with coordinates (x, y).

This function is described with the following equation:

$$Points = \begin{bmatrix} x_1 & \dots & x_n \\ y_1 & \dots & y_n \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

We will use the python function randn from the module numpy.random. The following function generates such a point cloud:

```
def generation(n, x, y):
    Y = np.random.randn(n, 2) + np.array([[x, y]])
    return Y
```

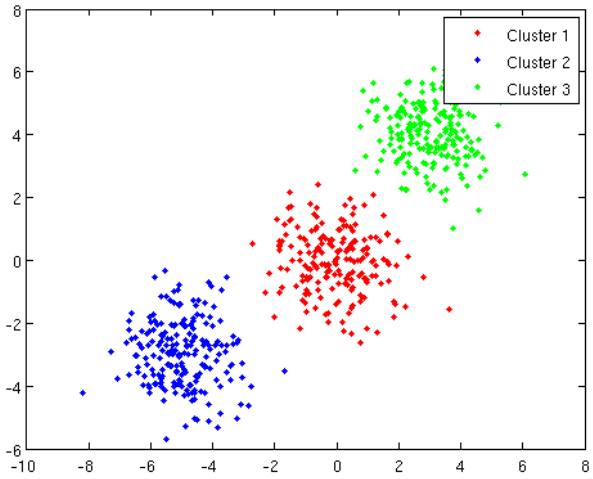
The concatenation of numpy arrays can be done by using `np.concatenate((A1, A2, A3))`.

?

- Use this function to generate 3 set of points (in a unique matrix) around the points $(0, 0)$, $(3, 4)$ and $(-5, -3)$.
- Use the builtin kmeans function for separating the points. The result is presented in Fig. 21.1.

Use the python function `sklearn.cluster.KMeans`. Verify the utility of the option `n_init=10`.

Figure 21.1: Resulting clustering of random points.



21.5. Color image segmentation using K-means: $k = 3$ in 3D

The k-means clustering is now used for segmenting the color image representing the muscle cells 'Tv16.png'.

?

Which points have to be separated? Transform the original image into a vector of size $N \times 3$ (where N is the number of pixels) which represent the 3 components R, G and B of each image pixel.

Informations

An image is an array of shape $(nLines, nCols, nChannels)$. The `sklearn.cluster.KMeans` functions requires data of shape $(nPoints, nDimensions)$. Thus, the following lines will *numpy.reshape* the arrays within the right shape:



```
# image is the color image
[nLines, nCols, nChannels] = image.shape
data = np.reshape(image, (nLines*nCols, nChannels))
data = data.astype(np.float32) # convert in float32 data type
```

After applying the KMeans function on data, the result should be converted into the image space, i.e. a 2D array of shape $(nLines, nCols)$:



```
# k_means_labels are the results of the Kmeans algorithm, see doc
result = np.reshape(k_means_labels, (nLines, nCols))
```



- Visualize the 3-D map (histogram) of all these color intensities.
- Make the clustering of this 3-D map by using the K-means method.
- Visualize the corresponding segmented image.



21.6. Python correction



21.6.1. Manual thresholding

Visual analysis of histogram

When manually choosing a threshold value, one has to analysis the histogram (Fig. 21.2).



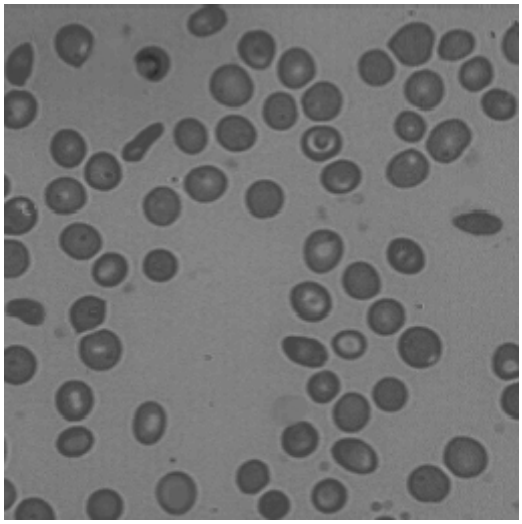
```
import numpy as np
import imageio
import matplotlib.pyplot as plt # plots
from skimage import filter # otsu thresholding

# read image
cells =imageio.imread('cells.png');

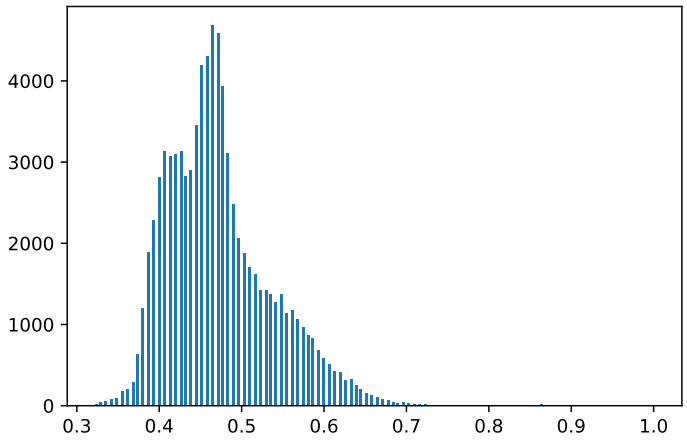
# display histogram
fig =plt.figure();
plt.hist(cells.flatten(), 256)
fig.show();
fig.savefig("histo.pdf");
```

Figure 21.2: Original image and its histogram.

(a) Original image.



(b) Histogram.



Segmentation



```
fig =plt.figure();
plt.subplot(1,2,1)
plt.imshow(cells, plt.cm.gray); plt.title('Original image');
plt.subplot(1,2,2)
plt.imshow(cells>80, plt.cm.gray); plt.title('Manual segmentation');
fig.savefig("manual.pdf");
```

21.6.2. Automatic thresholding

```
def autothresh(image):
    """ Automatic threshold method
    @param image: image to segment
    @return : threshold value
    """
    s = 0.5*( np.min(image) + np.max(image));
    done = False;
    while ~done:
        B = image>=s;
        sNext = .5*( np.mean(image[B]) + np.mean(image[~B]));
        done = abs(s-sNext)<.5;
        s = sNext;
    return s
```

The results are displayed using the following code (Fig. 21.3):

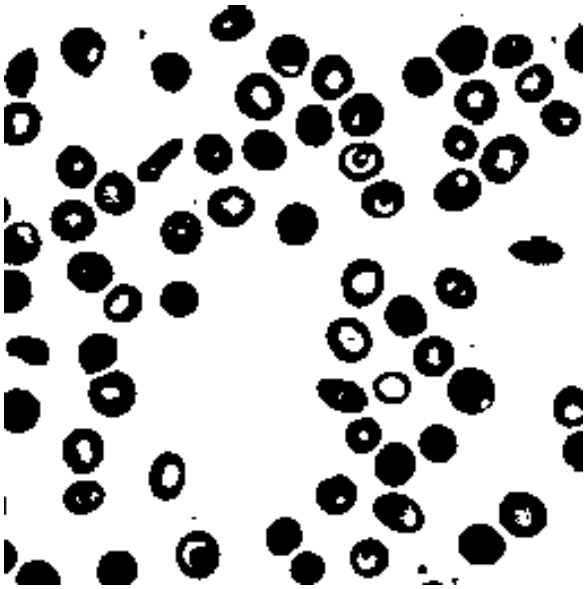
```
# Automatic threshold
s_auto= autothresh( cells );

# Otsu thresholding
s_otsu = filter.threshold_otsu( cells );

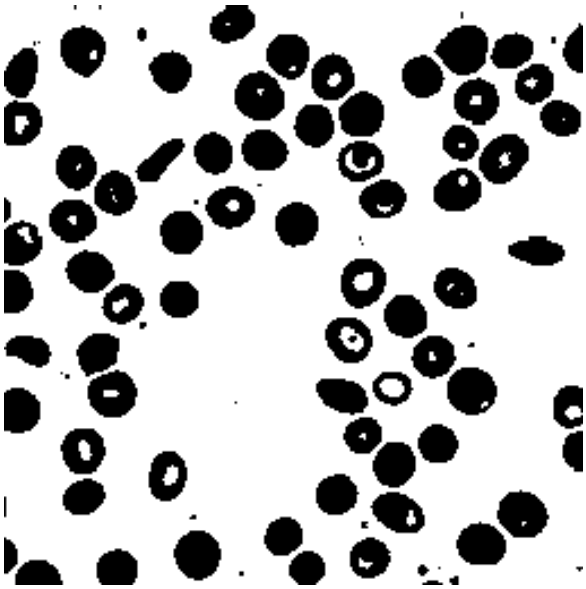
plt.figure();
plt.subplot(1,2,1)
plt.imshow(cells>s_auto, plt.cm.gray); plt.title('Automatic thresholding')
plt.subplot(1,2,2);
plt.imshow(cells>s_otsu, plt.cm.gray); plt.title('Otsu thresholding');
```

Figure 21.3: Automatic thresholding and thresholding by Otsu. Results are almost identical because threshold values are 105.3 and 105, respectively.

(a) Automatic threshold.



(b) Otsu threshold.



21.6.3. k-means clustering

Different techniques can be found in the scikit documentation. A point cloud will first be generated, from 3 clustered cloud points. The objective is then to segment all the points into their original cluster.

Imports



```
import numpy as np
import matplotlib.pyplot as plt
import time

from sklearn.cluster import KMeans
```

Generation of point clouds



```
def generation(n, x, y):
    Y = np.random.randn(n, 2) + np.array([[x, y]]);
    return Y

points1=generation(100, 0, 0);
points2=generation(100, 3,4);
points3=generation(100, -5, -3);

pts=np.concatenate((points1, points2, points3));
plt.plot(pts[:,0], pts[:,1], 'ro');

plt.show();
```

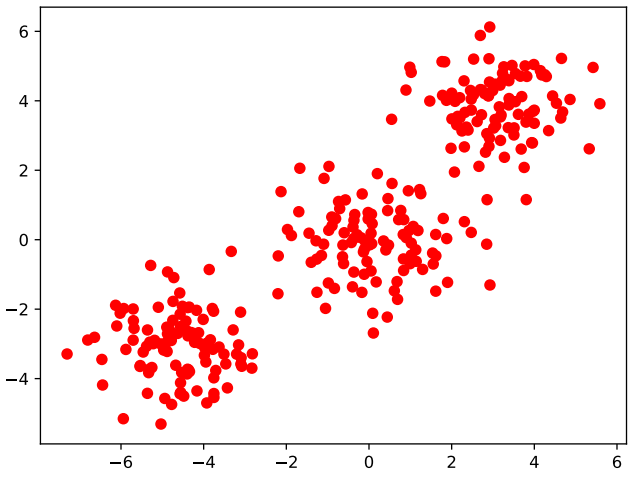


Figure 21.4: Point cloud.

k-means clustering

```
n=3; # number of clusters

# k-means initialization
k_means = KMeans(init='k-means++', n_clusters=n, n_init=10)
t0 = time.time(); # computation time
k_means.fit(pts); # kmeans segmentation

t_batch = time.time() - t0;

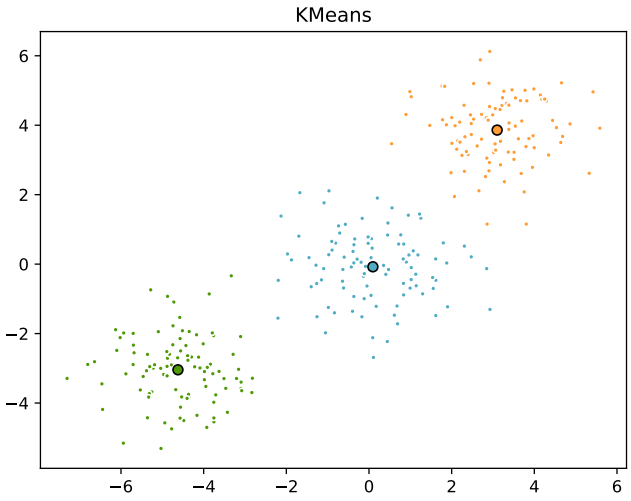
# retrieve results
k_means_labels = k_means.labels_;
k_means_cluster_centers = k_means.cluster_centers_;

# plot
fig = plt.figure()
colors = ['#4EACC5', '#FF9C34', '#4E9A06']

# k-means
# zip agregates values two by two
for k, col in zip(range(n), colors):
    my_members = k_means_labels == k
    cluster_center = k_means_cluster_centers[k]

    # display points
    plt.plot(pts[my_members, 0], pts[my_members, 1], 'w',
            markerfacecolor=col, marker='.')

    # display centroid
    plt.plot(cluster_center[0], cluster_center[1], 'o',
             markerfacecolor=col, markeredgecolor='k',
             markersize=6)
plt.title('KMeans')
plt.show()
fig.savefig("kmeans.pdf");
```



21.6.4. Color image segmentation

Three different colors can be observed in the image. The objective is to separate the 3 colors with the help of the K-means algorithm. Thus, the segmentation is performed in the RGB color space, and each pixel is represented by a point in this 3D space. Initialization steps are identical to previous code. The data is converted from a color image (of size $(n, m, 3)$) to a vector (of size $(n \times m, 3)$), done by the reshape function of numpy.

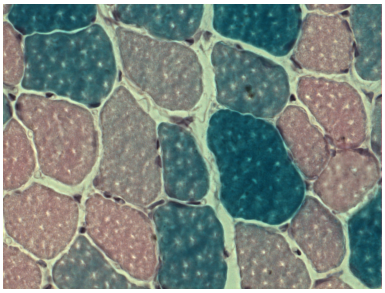


```
# load color image
cells =imageio.imread('Tv16.png');
[nLines,nCols,channels] = cells.shape
# reshape data
data = np.reshape( cells , (nLines*nCols, channels));
k_means.fit(data);
# convert result to an image
# as we got labels , we expand the dynamic (multiply by 70)
segmentation = 70*np.reshape(k_means.labels_, (nLines, nCols));

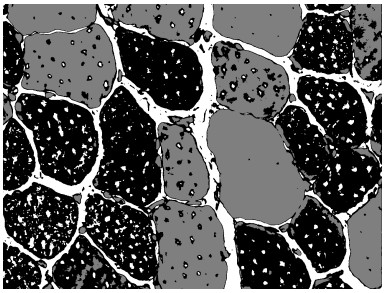
fig=plt.figure();
plt.imshow(segmentation, cmap=plt.cm.gray);
imageio.imwrite("segmentation_kmeans.png", segmentation);
```

Figure 21.5: Segmentation result.

(a) Original image.



(b) Segmented.



3D scatter plot

This is a method to display colors in the RGB cube. This method is really slow, depending on your GPU.



```
from mpl_toolkits.mplot3d import Axes3D # 3D scatter plot
# plot
colors = ['#4EACC5', '#FF9C34', '#4E9A06']

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot scatter points
for k, col in zip(range(n), colors):
    my_members = k_means_labels == k
    cluster_center = k_means_cluster_centers[k]
    ax.scatter(data[my_members, 0], data[my_members, 1],
               data[my_members, 2], c=col)
    ax.scatter( cluster_center [0], cluster_center [1],
                cluster_center [2], s=30, c=col)
```

★ 22 Segmentation by region growing

This tutorial proposes to program the region growing algorithm.

22.1. Introduction

The region growing segmentation method starts from a seed. The initial region first contains this seed and then grows according to

- a growth mechanism (in this tutorial, the N_8 will be considered)
- an homogeneity rule (predicate function)

The algorithm is simple and barely only needs a predicate function:

Data: I: image
Data: seed: starting point
Data: queue: queue of points to considere
Result: visited: boolean matrix, same size as I
begin
  queue.enqueue(seed);
  while *queue is not empty* **do**
    p = queue.dequeue();
    foreach *neighbor of* p **do**
      if *not visited(p)* ***and*** *neighbor verifies predicate* **then**
        queue.enqueue(neighbor);
        visited(neighbor) = true;
      end
    end
  end
  return visited
end

Algorithm 5:

To get the coordinate of the mouse click, use the matplotlib connect utilities. Define a function def onpick.

```
# start by displaying a figure,
# ask for mouse input (click)
fig = plt.figure();
ax = fig.add_subplot(211);
ax.set_title('Click on a point')
# load image
img = misc.ascent();
ax.imshow(img, picker = True, cmap = plt.gray());
# connect click on image to onpick function
fig.canvas.mpl_connect('button_press_event', onpick);
plt.show();
def onpick(event):
    """ connector """
```

22.2. Region growing implementation

?

The seed pixel is denoted s.

- Code the predicate function: for an image f and a pixel p, p is in the same segment as s implies $|f(s) - f(p)| \leq T$.
- Code a function that performs region growing, from a starting pixel (seed).
- Try others predicate functions like:
  - pixel p intensity is close to the region $\mathcal{R}$ mean value, i.e.:

$$|I(p) - m_{\mathcal{R}}| \leq T$$

  - Threshold value T varies depending on the region $\mathcal{R}$ and the intensity of the pixel $I(p)$. It can be chosen this way, with σ and m representing the standard deviation and the mean, respectively:

$$T = \left(1 - \frac{\sigma_{\mathcal{R}}}{m_{\mathcal{R}}}\right) \cdot T_0$$



22.3. Python correction



22.3.1. imports



```
import queue
from scipy import misc
import matplotlib.pyplot as plt
import numpy as np
```

22.3.2. Predicate

This function defines the agregation condition.



```
def predicate(image, i, j, seed) :
    f=image[i,j];
    g=image[seed[0], seed[1]];
    return abs(f-g)<20
```

The following code is used to start the region growing from a pixel manually clicked on an image.



```
# start of code
fig = plt.figure();
ax = fig.add_subplot(211);
ax.set_title('Click on a point')

# load image
img = misc.ascent();
ax.imshow(img, picker=True,cmap=plt.gray());

fig.canvas.mpl_connect('button_press_event', onpick)
plt.show();
```

And here comes the main function for region growing. The result is illustrated in Fig.22.1.



```
def onpick (event):
    """
    this functions gets the event 'click', and starts the region growing algorithm
    Notice that img is a global variable
    """
    print("x, y: ", event.xdata, event.ydata);
    # original pixel
    seed = np.array ([int (event.ydata), int (event.xdata)]);
    q = queue.Queue ();

    myfunctions = [predicate, predicate2, predicate3 ];

    for index,f in enumerate(myfunctions):

        # initializes the queue
        q.put (seed);

        #Visited matrix : result of segmentation
        #this matrix will contain 1 if in the region, -1 if visited but not in the region, 0 if not visited
        visited = np.zeros (img.shape)
        #---------------
        #Start of algorithm
        visited [seed [0], seed [1]] = 1;

        while not q.empty ():
            p = q.get ();

            for i in range (max (0, p[0] - 1), min (img.shape [0], p[0] + 2)):
                for j in range (max (0, p[1] - 1), min (img.shape [1], p[1] + 2)):
                    if not visited [i, j]:
                        if f(img, i, j, seed, visited ):
                            visited [i, j] = 1;
                            q.put (np.array ([i, j]));
                        else:
                            visited [i, j] = -1;

        #visited matrix contains the segmentation result
        #display results
        ax = fig.add_subplot (142+index);
        ax.imshow (visited == 1);

        imageio.imsave(f.__name__ +'_seg.python.png', (visited ==1).astype('int'))

    fig.canvas.draw ();
    plt.show();
    return;
```

Notice that values -1 of the visited matrix avoid testing multiple times the same pixel. In the predicate function, the visited matrix is used in case of adapting the predicate to the current region. In the next case, the candidate pixel's graylevel is compared to the mean gray value of the region. The results are illustrated Fig.22.1.



```
def predicate2(image, i, j, seed, visited):
    f = int(image[i, j]);
    m = np.mean(image[visited==1]);
    return abs (f - m) < 20
```

Another predicate function would be:



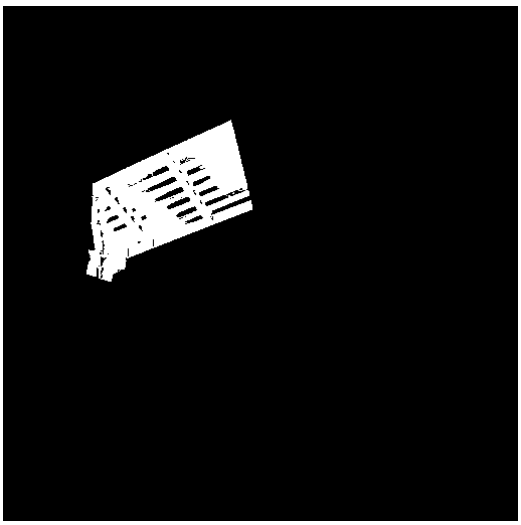
```
def predicate3(image, i, j, seed, visited):
    f = int(image[i, j]);
    m = np.mean(image[visited==1]);
    s = np.std(image[visited==1]);
    return abs(f - m) < 20 * (1-s/m)
```

Figure 22.1: Region growing illustration for pixel (189,136). The segmentation result highly depend on the order used to populate the queue, on the predicate function and on the seed pixel.

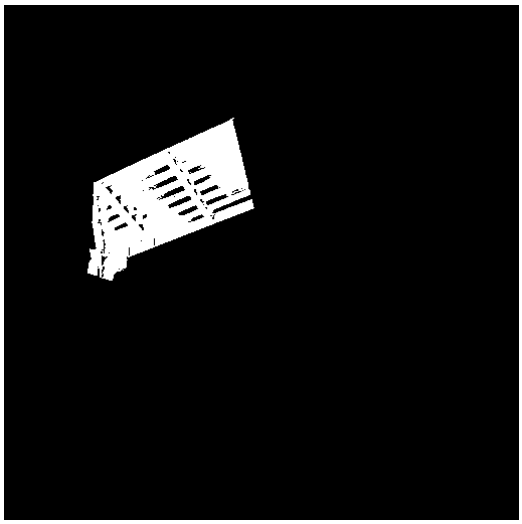
(a) Original image.



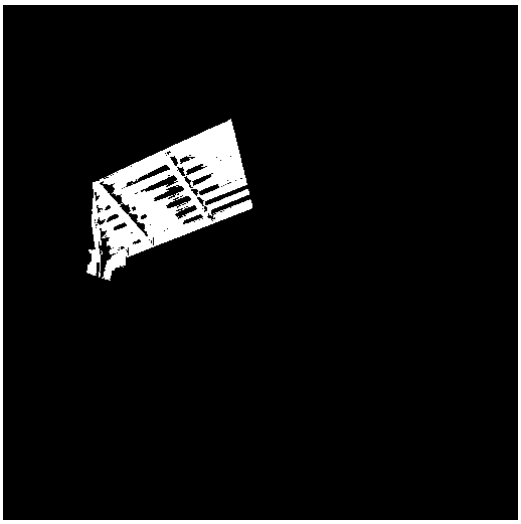
(b) Segmented region.



(c) Segmented region.



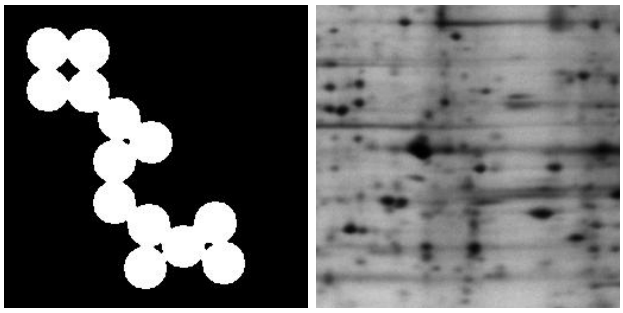
(d) Segmented region.



★★★ 23 Watershed

This tutorial aims to study the watershed transform for image segmentation. In image processing, an image can be considered as a topographic surface. If we flood this surface from its minima and if we prevent the merging of the water coming from different sources, we partition the image into two different sets: the catchment basins separated by the watershed lines.

The different processes will be applied on the following images:



(a) circles (b) gel

23.1. Watershed and distance maps

The objective is to individualize the disks by disconnecting them with the distance map.

?

- Calculate the distance map of the image 'circles'.
- Take the complementary of this distance map and visualize its minima.
- Calculate the watershed transform of the inverted distance map.
- Subtract the watershed lines to the original image.

Look at the documentation of scipy.ndimage.morphology for Euclidean or Chamfer distance transform. Reconstruction operator can be found in skimage.morphology.

23.2. Watershed and image gradients

?

- Calculate the Sobel gradient of the image 'gel'.
- Visualize the minima of the image gradient.
- Apply the watershed transform on the gradient image.

The watershed transform, applied in a direct way, leads to an over-segmentation. To overcome this limitation, the watershed operator can be applied on a filtered image.

?

- Smooth the original image with a low pass filter. A Gaussian filter is also a good solution. To stay in the mathematical morphology field, you can use an alternate morphological filter (opening followed by closing).
- Calculate the gradient operator on the filtered image.
- Calculate the corresponding watershed.



Look at the module skimage.morphology to find the useful functions.

23.3. Constrained watershed by markers

In order to individualize the image spots, we have to determine the internal and external markers for the constrained watershed.

?

- Calculate the gradient (Sobel) of the filtered image.
- Calculate the internal markers (minima of the filtered image) and external (watershed of the filtered image) as minima of the gradient image.
- Calculate the corresponding watershed.
- Superimpose the watershed lines of the resulting segmentation to the original image.



skimage.filters.sobel find the edges in an image using the Sobel filter. The Prewitt gradient is also a good solution.

23.4. Python correction

```
import numpy as np
from scipy import misc
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from scipy.ndimage.morphology import distance_transform_edt
from scipy.ndimage.morphology import distance_transform_cdt
from skimage import morphology
```

23.4.1. Watershed and distance map

Regional maxima

The following code is an implementation of the regional maxima (from mathematical morphology definition). It deals with the case of float images as input.

```
def rmax(I):
    """
    Own version of regional maximum
    This avoids plateaus problems of peak_local_max
    I: original image, int values
    returns: binary array, with True for the maxima
    """
    I = I.astype('float');
    I = I / np.max(I) * 2**31;
    I = I.astype('int32');
    h = 1;
    rec = morphology.reconstruction(I, I+h);
    maxima = I + h - rec;
    return maxima>0
```

Distance map followed by a watershed

This method is a classical way of performing the separation of some objects by proximity or influence zones. It is illustrated in Fig.23.1.

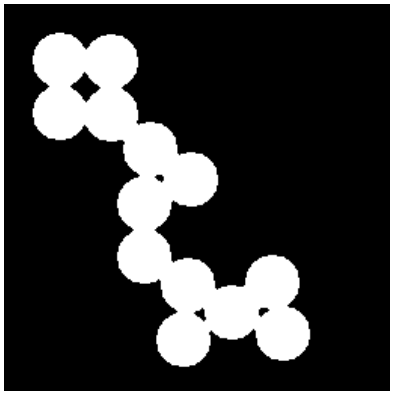
```
A = imageio.imread('circles.tif');
# chamfer distance gives integer distances
dm = distance_transform_edt(A);
# regional maxima
local_maxi = rmax(dm);
```

The local maxima will be used as a marker for the watershed operation, in order to perform the separation of the grains, see Fig.23.1.

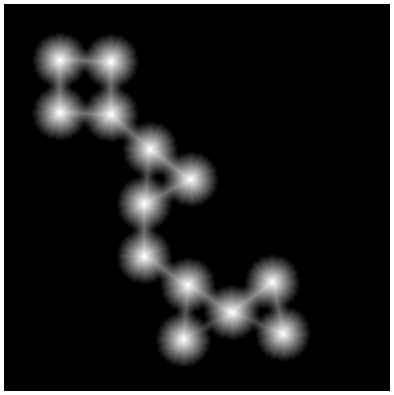
```
# watershed segmentation for separating the circles
markers = ndi.label(local_maxi, np.ones((3, 3)))[0]
W = morphology.watershed(-dm, markers, watershed_line=True)
# separation of the grains
B = A & W==0;
separation = ndi.label(B, np.ones((3,3)))[0];
```

Figure 23.1: Steps of the separation of the grains.

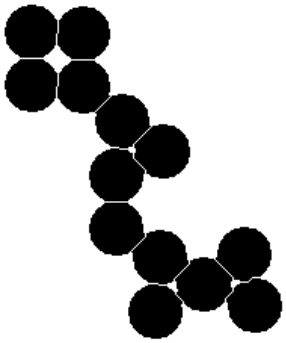
(a) Original image.



(b) Distance map.



(c) Separation of the grains.



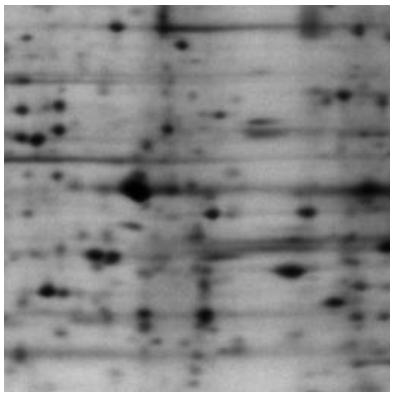
23.4.2. Watershed and image gradients

The gradient image amplifies the noise. Thus, the watershed operator directly applied to the gradient of the image produces an over-segmented image (see Fig.23.2).

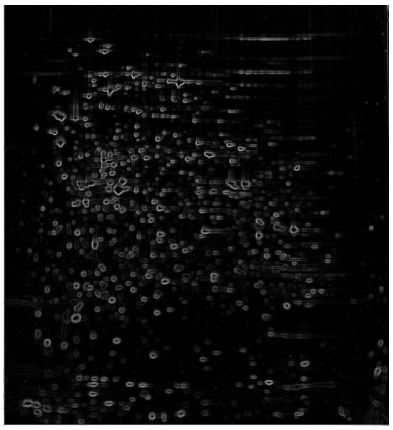
```
def sobel_mag(im):
    """
    Returns Sobel gradient magnitude
    im: image array of type float
    returns: magnitude of gradient (L2 norm)
    """
    dx = ndi.sobel(im, axis=1)    # horizontal derivative
    dy = ndi.sobel(im, axis=0)    # vertical derivative
    mag = np.hypot(dx, dy)  # magnitude
    return mag;
```

Figure 23.2: Performing the watershed on the gradient image is usually not a good idea.

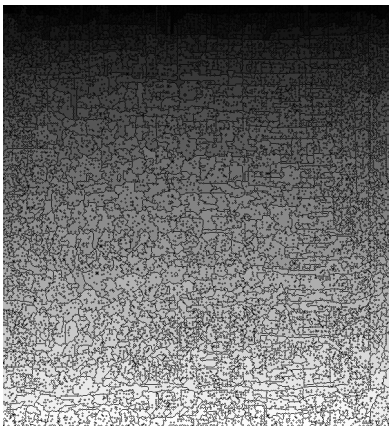
(a) Original image.



(b) Amplitude of the gradient (Sobel).

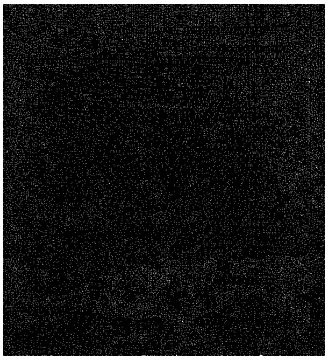


(c) Watershed segmentation.



In fact, this method produces as many segments as there are minima in the gradient image Fig.23.3.

Figure 23.3: Local minima of the gradient image.



Solution: filtering the image

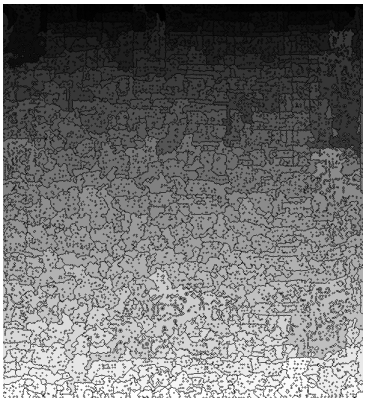
Before evaluating the gradient, the image is filtered. The number of minima is lower and this leads to a less over-segmented image (Fig.23.4).

```
SE = morphology.disk(2);
O = morphology.opening(gel, selem=SE);
F = morphology.closing(O,    selem=SE).astype('float');
g = sobel_mag(F).astype('float');
```

23.4.3. Watershed constrained by markers

The watershed can be constrained by markers: the markers can provide the correct number of regions. This method imposes both the background (external markers) and the objects (internal markers). The results are illustrated in Fig.23.5. The ultimate erosion of the internal markers is used to deconnect these markers from the external markers.

Figure 23.4: Even if the gradient is performed on the filtered image, there is still a high over-segmentation.



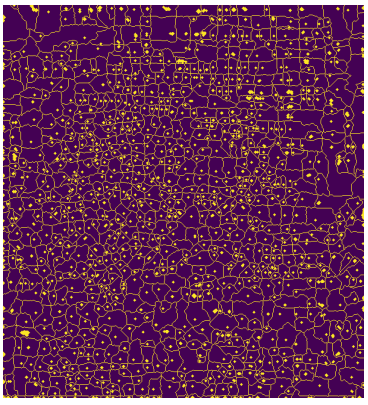
```
local_maxi = rmax(255-F);
markers = ndi.label(local_maxi, np.ones((3, 3)))[0]
W = morphology.watershed(F, markers, watershed_line=True)

markers2 = local_maxi | (W==0);
M = ndi.label(markers2, np.ones((3, 3)))[0]
segmentation = morphology.watershed(g, M, watershed_line=True);

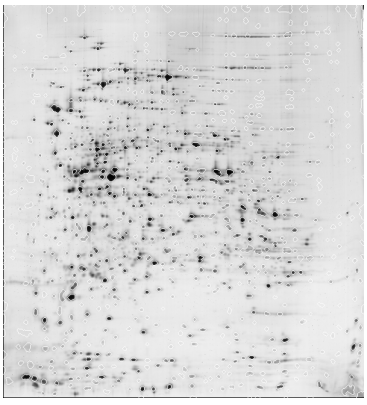
gel[segmentation==0] = 255;
plt.imshow(gel);
plt.show();
imageio.imwrite("segmentation.python.png", gel);
```

Figure 23.5: Watershed segmentation by markers.

(a) Markers of the background and of the objects.



(b) Final segmentation.



★★★ 24 Active contours

This tutorials aims at introducing the active contours (a.k.a. snakes) method as originally presented in [18]. It is a segmentation method based on the optimization of a contour that will converge to a specific object.

24.1. Definition

A snake is a parametric curve $v(s)$ with $s \in [0;1[$. The energy functional is represented by:

$$E_{snake} = \int_0^1 E_{\text{int}}(v(s))\mathrm{d}s + \int_0^1 E_{\text{ext}}(v(s))\mathrm{d}s.$$

The internal energy is detailed in Eq.24.1. The first order derivation constrains the length of the curve, the second order derivation constrains the curvature. The parameters α and β a priori depend on s, but for simplicity, constant values will be taken.

$$E_{\text{int}}(v(s)) = \frac{1}{2}\left(\underbrace{\alpha(s)|v'(s)|^2}_{length} + \underbrace{\beta(s)|v''(s)|^2}_{curvature}\right) \tag{24.1}$$

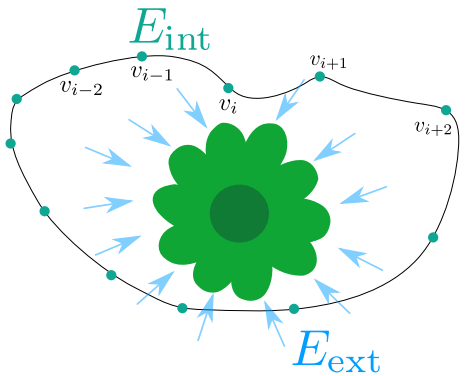
The snake will be attracted by some edges points from the external energy (the image to be segmented). The, the energy functional will be in a local minimum: it is shown that the Euler-Lagrange equation is satisfied:

$$-\alpha v^{(2)} + \beta v^{(4)} + \nabla E_{\text{ext}} = 0$$

with $v^{(2)}$ and $v^{(4)}$ denoting the 2nd and 4th order derivatives. To solve this equation, the gradient descent method is employed: the snake is now transformed into a function of the position s and the time t, with $F_{\text{ext}} = -\nabla E_{\text{ext}}$

$$\frac{\partial s}{\partial t} = \alpha v^{(2)} - \beta v^{(4)} + F_{\text{ext}}$$

Figure 24.1: Illustration of the active contours segmentation method. Two energies are at stake: internal energies depend only on the snake shape and control points, external energies are related to the image properties.



24.2. Numerical resolution

The spatial derivatives are approximated with the finite difference method, and the snake is now composed of n points, $i \in [1;n]$:

$$\begin{aligned} x^{(2)} &= x_{i-1} - 2x_i + x_{i+1} \\ x^{(4)} &= x_{i-2} - 4x_{i-1} + 6x_i - 4x_{i+1} + x_{i+2} \end{aligned}$$

The gradient descent method can be written as, with γ being the time step and controls the convergence speed:

$$\begin{aligned} \frac{x_t - x_{t-1}}{\gamma} &= A \cdot x_t + f_x(x_{t-1}, y_{t-1}) \\ \frac{y_t - y_{t-1}}{\gamma} &= A \cdot y_t + f_y(x_{t-1}, y_{t-1}) \end{aligned}$$

with

$$A = \begin{bmatrix} -2\alpha{-}6\beta & \alpha{+}4\beta & -\beta & 0 & \cdots & 0 & -\beta & \alpha{+}4\beta \\ \alpha{+}4\beta & -2\alpha{-}6\beta & \alpha{+}4\beta & -\beta & 0 & \cdots & 0 & -\beta \\ -\beta & \alpha{+}4\beta & -2\alpha{-}6\beta & \alpha{+}4\beta & -\beta & 0 & \cdots & 0 \\ 0 & -\beta & \alpha{+}4\beta & -2\alpha{-}6\beta & \alpha{+}4\beta & -\beta & 0 & \cdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & & \vdots \\ 0 & \cdots & -\beta & \alpha{+}4\beta & -2\alpha{-}6\beta & \alpha{+}4\beta & -\beta & 0 \\ 0 & \cdots & 0 & -\beta & \alpha{+}4\beta & -2\alpha{-}6\beta & \alpha{+}4\beta & -\beta \\ -\beta & 0 & \cdots & 0 & -\beta & \alpha{+}4\beta & -2\alpha{-}6\beta & \alpha{+}4\beta \\ \alpha{+}4\beta & -\beta & 0 & \cdots & 0 & -\beta & \alpha{+}4\beta & -2\alpha{-}6\beta \end{bmatrix}$$

The forces f_x and f_y are the components of F_{ext}. For example for an image I, if $*$ denotes the convolution and G_σ a gaussian kernel of standard deviation σ:

$$F_{\text{ext}} = -\nabla(\|\nabla(G_\sigma * I)\|)$$

24.2.1. Binary disk

Generate a binary image (of size $n \times n$ pixels, with values 0 and 1) containing a disk (of a radius R).

Informations

Use the np.meshgrid function to generate a discretization of the space, given by X and Y, and notice that a binary disk can be defined by:

$$X^2 + Y^2 \leq R^2$$

24.2.2. Ellipse discretization

Generate the initial contour as an ellipse, with the same center as the disk.

The equation of the ellipse is given by:

$$\begin{aligned} x &= x_0 + a\cos\theta \\ y &= y_0 + b\sin\theta \end{aligned}$$

24.2.3. Construction of the pentadiagonal matrix

?

Generate the pentadiagonal matrix A. The different parameters can be $\alpha = 10^{-5}$ and $\beta = 0.05$.

`scipy.sparse.diags` will be usefull to generate this matrix.

24.2.4. Last step : iterations

?

Program the iterations and visualize the results. For example, $\gamma = 200$ and 1000 iterations may give an idea of the parameters to use.



Informations

You can use the methods `filters.gaussian_gradient_magnitude` and `prewitt` from the module `scipy.ndimage` to compute the external forces.

24.3. Python correction

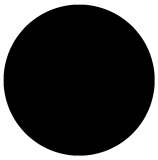
```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import scipy.ndimage.filters
from scipy import interpolate
import progressbar
```

24.3.1. Binary image generation

A disk (Fig.24.2) is generated via the meshgrid function.

```
# disk: number of points
n = 1024;
# disk: radius
R = 300;
# construct a binary image of a disk
X, Y = np.meshgrid(np.arange(-n/2,n/2,1), np.arange(-n/2,n/2,1));
I = X**2+Y**2<=R**2;
I = I.astype('float')
```

Figure 24.2: Discrete circle generated via meshgrid.



24.3.2. Initial contour

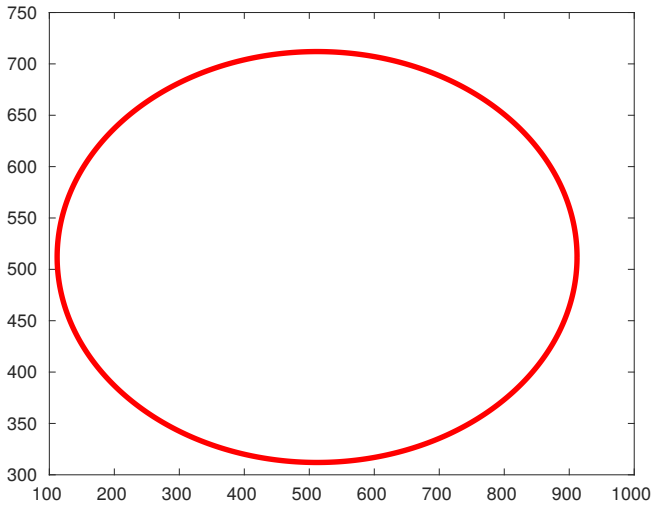
The choice of the initial contour (Fig.24.3) is crucial in this method. The parameters used in this example ensure the convergence of the snake.

```
step=.01;
x = n/2 + 400 * np.cos(np.arange(0,2*np.pi+step,step));
y = n/2 + 200 * np.sin(np.arange(0,2*np.pi+step,step));
```

The different parameters are defined by:

```
k=.1;
alpha = .0001;
beta = 10;
gamma= 100;
 iterations = 1000;
```

Figure 24.3: Discrete Ellipse.



24.3.3. Matrix construction

This is maybe the hardest part of this code, with the use of the scipy.sparse.diags function.

```
N = x.size;
X = np.array([-beta, alpha+4*beta, -2*alpha-6*beta, alpha+4*beta, -beta, -beta, alpha+4*beta, -beta, alpha+4*
    ↪ beta])
A = scipy.sparse.diags(X, np.array([-2, -1, 0, 1, 2, N-2, N-1, -N+2, -N+1]), shape=(N,N)).toarray();
AA = np.identity(N)-gamma*A;
invAA = np.linalg.inv(AA);
```

24.3.4. External forces

The external forces are computed with the following code. Notice that axis 0 correspond to the vertical axis (y) and and the axis 1 corresponds to the horizontal axis (x).

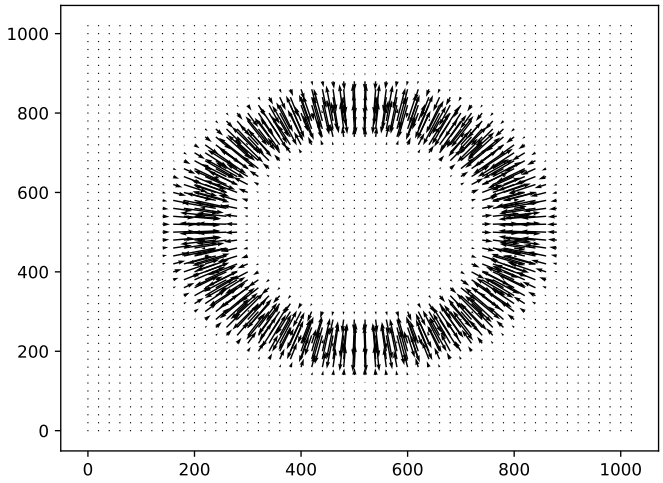
```
# external forces computation
G = scipy.ndimage.filters.gaussian_gradient_magnitude(I, 30);
# Notice that horizontal axis x is 1, and vertical axis is 0
Fy = scipy.ndimage.prewitt(G, axis=0);
Fx = scipy.ndimage.prewitt(G, axis=1);
```

24.3.5. Display results

To enhance the role of the external forces, the arrows showing the force are displayed (quiver function, see Fig.24.4).

```
imshow(I,[])
hold on
plot([x;x(1)], [y; y(1)], 'g', 'linewidth', 3);
# display arrows for external forces
step=20;
subx = 1:step:size(I,1);
suby = 1:step:size(I,2);
[Xa, Ya] = meshgrid(subx, suby);
quiver(Xa, Ya, Fx(subx, suby), Fy(subx,suby));
```

Figure 24.4: External forces that will be applied to the snake.

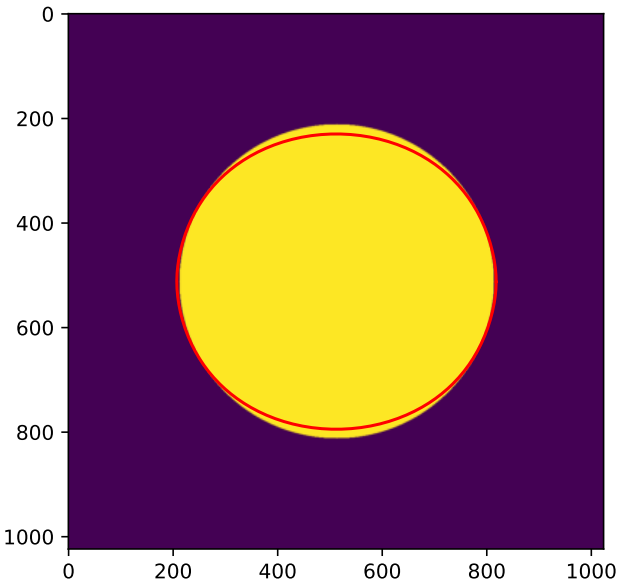


24.3.6. Convergence algorithm

```
# interpolation methods to get values of the external forces at the
# coordinates of the snake
sx, sy = I.shape;
ix = interpolate.interp2d(np.arange(n), np.arange(n), Fx);
iy = interpolate.interp2d(np.arange(n), np.arange(n), Fy);
# loop for convergence of the snake
bar = progressbar.ProgressBar();
for index in bar(range( iterations )):
    fex = np.array([float(ix(XX,YY)) for XX,YY in zip(x,y)]);
    fey = np.array([float(iy(XX,YY)) for XX,YY in zip(x,y)]);
    #print(np.max(fex), np.min(fex))
    x = np.matmul(invAA, x+gamma*fex);
    y = np.matmul(invAA, y+gamma*fey);
```

The results are displayed in Fig.24.5.

Figure 24.5: Result of the snake converging toward the disk, after 1000 iterations with the proposed parameters.



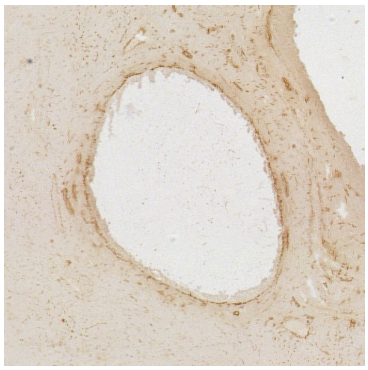
★ 25 Segmentation of follicles

This practical work aims to investigate image segmentation with a direct application to ovarian follicles. The overall objective is to extract and quantify the granulosa cells and the vascularization of each follicle included in an ewe's ovary.

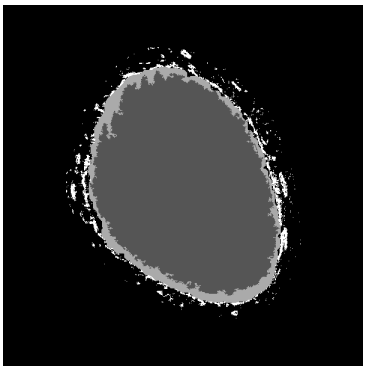
The image to be processed is a 2D histological image of an ewe's ovary acquired by optical microscopy in Fig.25.1. The presented image contains one entire follicle (the white region and its neighborhood) and a part of a second one (right-upper corner). The follicle is composed of different parts shown in: antrum, granulosa cells and vascularization. The theca is the ring region around the antrum where the follicle is vascularized.

Figure 25.1: Different parts of the follicles, to be segmented.

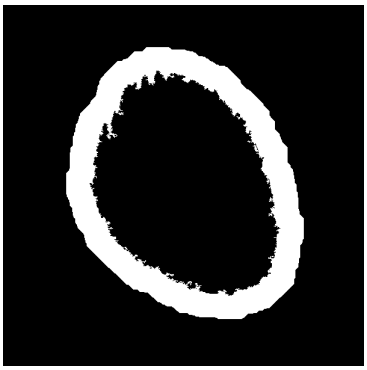
(a) Original image with one entire follicle (white region and its neigborhood).



(b) Antrum (dark gray), granulosa cells (light gray) and vascularization (white) of the follicle.



(c) Theca.



25.1. Vascularization

?

- Load and visualize the image.
- Extract the antrum of the follicle.
- Extract the vascularization (inside a ring around the antrum).

25.2. Granulosa cells

?

Which kind of processing could be suitable for extracting the granulosa cells?

25.3. Quantification

?

Provide some geometrical measurements of the different entities of the follicle (antrum, vascularization, granulosa cells).

25.4. Python correction

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import misc,ndimage
from skimage import morphology
```

25.4.1. Vascularization

Antrum segmentation

The first step consists in the segmentation of the antrum by thresholding the blue component. Some post-processes are used to remove artifacts such as holes. This is illustrated in Fig. 25.2.

```
A = imageio.imread('follicle.png');
plt.imshow(A);
plt.show();
## Antrum
# segmentation by mathematical morphology
# manual selection of the antrum
B = A[:,:,2];
antrum = B > 220;
L = morphology.label(antrum, connectivity=2);
antrum = L == L[300,300];
antrum = ndimage.morphology.binary_fill_holes(antrum);
plt.imshow(antrum);
plt.title('Antrum')
plt.show();
```

Theca segmentation

The second step provides the segmentation of the theca which is extracted as a spatial region (corona) adjacent to and outside the antrum. The width of the corona is selected by the user (expert).

```
se40 = morphology.disk(40);
theca = morphology.binary_dilation(antrum, selem=se40);
theca = theca - antrum;
plt.imshow(theca); plt.title('Theca')
plt.show()
```

Vascularization segmentation

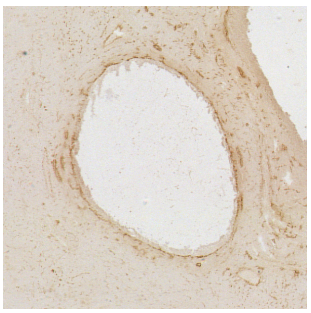
The final step extracts the vascularization of the considered follicle. Pixels belonging to the vascularization are considered to have a low blue component and they should also be included in the antrum of the follicle.

```
vascularization = B < 140;
vascularization = vascularization * theca;
plt.imshow(vascularization);
plt.title('vascularization')
plt.show();
```

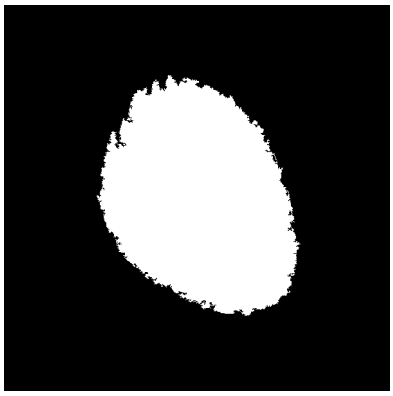
Results

Figure 25.2: Extraction of the different components of the follicle.

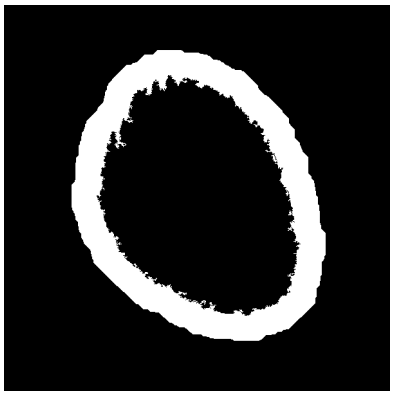
(a) Original image.



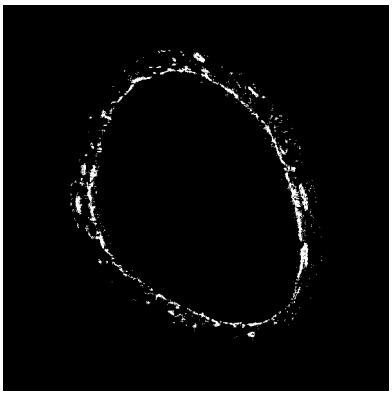
(b) Antrum.



(c) Theca.



(d) Vascularization.



25.4.2. Granulosa cells

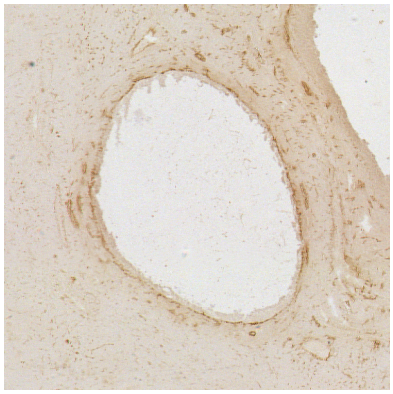
First solution

The granulosa cells have a low contrast, so it is difficlut to use thresholding techniques. But we know they are localized between the antrum and the vascularization. Nevertheless the vascularization is not a closed region outside the antrum. Therefore, the proposed solution consits in firt trying to close the vascularization region and taking the corona between this region and the antrum. The result is shown in Fig. 25.3.

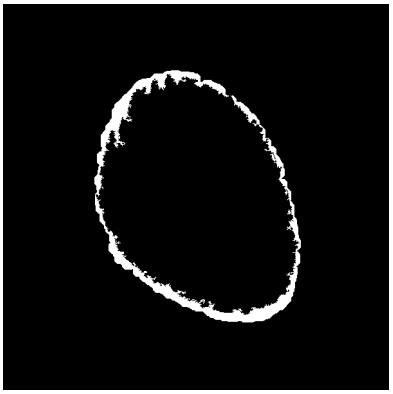
```
se10 = morphology.disk(10);
dil = 1-morphology.binary_closing( vascularization , se10);
L = morphology.label( dil , connectivity =1);
dil = L == L [300,300];
granulosa = dil - antrum;
plt .imshow(granulosa);
plt . title ('granulosa')
plt .show()
```

Figure 25.3: Extraction of the granulosa cells of the follicle.

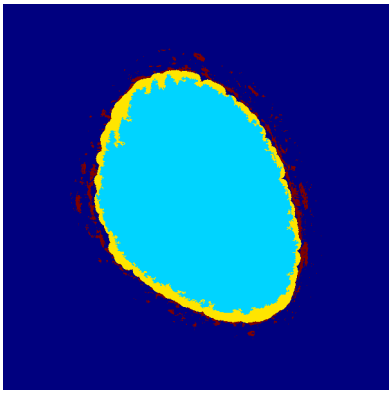
(a) Original image.



(b) Granulosa cells.



(c) Segmentation of the different parts.



Second solution

This first solution is not really robust. The closing of the vascularization region is not really accurate. A more robust solution consists in using deformable models to get the corona between the vascularization and the antrum. But this kind of method is out of the scope of this tutorial.

25.4.3. Quantification

The quantification is easy to process. In addition, we can represent the different extracted components of the follicle in false colors.

```
result = antrum + 2*granulosa + 3* vascularization ;
plt .imshow(result, cmap='jet');
plt .show();
plt .imsave(" result .png", result , cmap='jet');

follicle = antrum + theca;
q_vascularization = np.sum( vascularization ) / np.sum( follicle );
print (' vascularization : ', q_vascularization )
q_granulosa = np.sum(granulosa)/np.sum( follicle );
print ('Granulosa: ', q_granulosa);
```

This quantification gives:

```
vascularization :  0.0428495233516
Granulosa:  0.0913707490403
```

Part VIII Feature Detection and Description

26 Hough transform and line detection

This tutorial introduces the Hough transform. Line detection operators are implemented.

26.1. Introduction

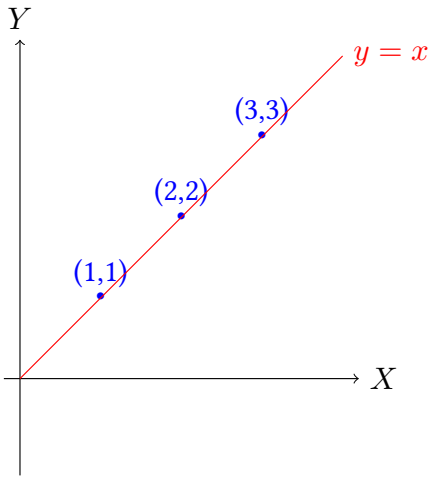
This tutorial deals with line detection in an image. For a given point of coordinates (x, y) in $\mathbb{R}^2$, there exists an infinite number of lines going by this point, with different angles θ. These lines are represented by the following equation:

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta).$$

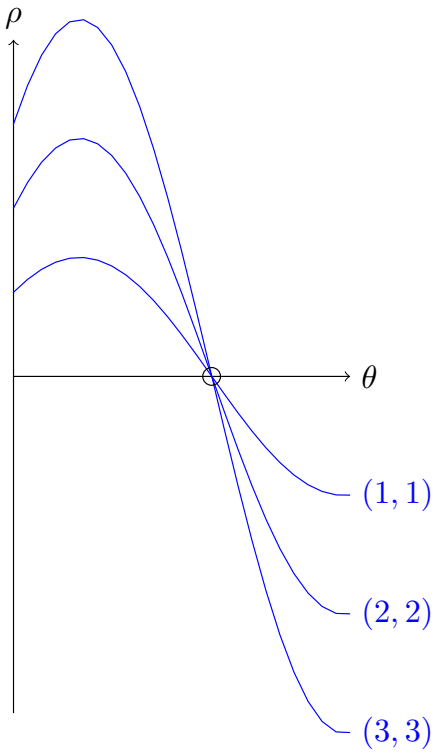
Thus, for each point (x, y) (Fig. 26.1a) corresponds a curve parametered by $[\theta, \rho]$, where $\theta \in [0;2\pi]$ (Fig. 26.1b). The intersection of these curves represents a line (in this case, $y = x$).

Figure 26.1: Representation of the Hough transform.

(a) Different points in $\mathbb{R}^2$.



(b) Hough transform of the 3 points.



26.2. Algorithm

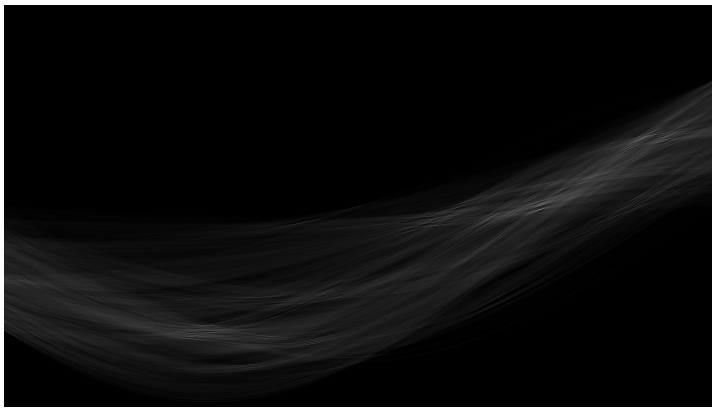
The (general and simple) method for line detection is then:

1. Compute contours detections (get a binary image BW).
2. Apply the Hough transform on the contours BW.
3. Detect the maxima of the Hough transform.
4. Get back in the Euclidean space and draw the lines on the image.

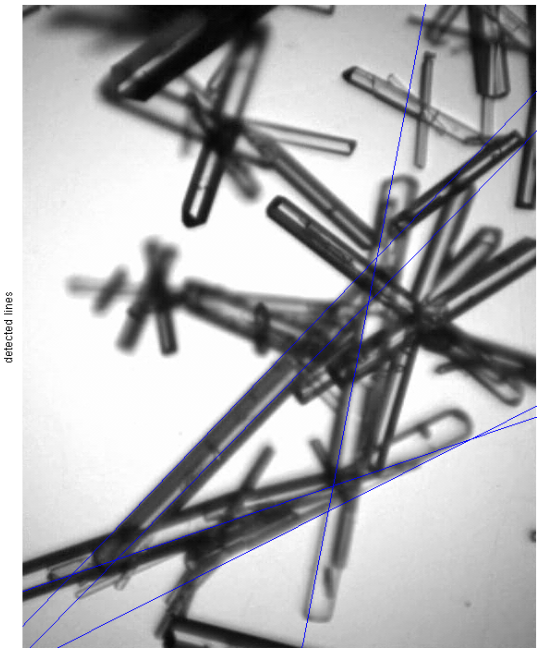
Results should look like in Fig. 26.2.

Figure 26.2: Lines detection via Hough transform.

(a) Hough transform and maxima detection. Angles θ are represented in abscissa, pixels ρ are represented in ordinates. The detection of the absolute maxima of this images will lead to the lines.



(b) Line detection.



26.3. Hough transform

?

Code a function that will transform each point of a binary image into a curve in the Hough space. For each curve, increment each pixel by one in the Hough space.

26.4. Maxima detection

?

Use or code a function to detect maxima (regional maxima). For each maximum, keep only one point.

26.5. Display lines

?

For each maximum, display the corresponding line above the original image.

26.6. Python correction

This correction makes use of the python modules numpy, opencv, skimage and matplotlib.

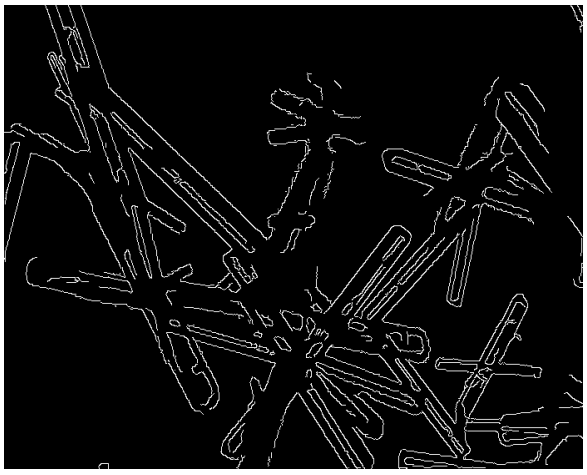
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.feature import peak_local_max
```

26.6.1. Contours detection

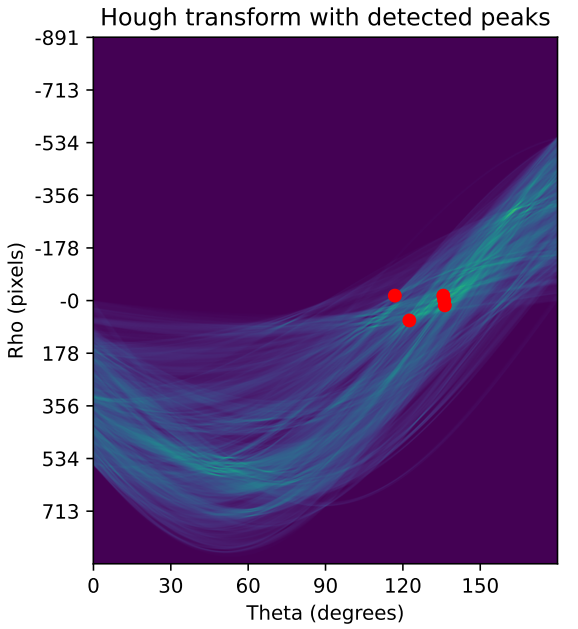
The contours are detected using the Canny edge detection method. In this code, the method from OpenCV is employed, see Fig.26.3a. Other methods can be used, for example gradient or laplacian filters.

```
img = cv2.imread('TestPR46.png');
plt.figure()
plt.imshow(img)

# perform contours detection
edges = cv2.Canny(img,100,200);
plt.figure()
plt.imshow(edges)
```



(a) Canny edge detection.



(b) Representation of the sinogram and detection of the maxima in the Hough space.

Figure 26.3: The algorithm of the Hough line detection consists in detecting the edges, then representing each pixel in the Hough space and finally detecting the maximal value in the sinogram.

26.6.2. Hough transform

Notice that OpenCV contains Hough fonctions: HoughLines and HoughLinesP. The result (sinogram) of the image is presented Fig.26.3b. The difficulty lays is the discretization (and the approximation) of the ρ coordinates.

First step consists in getting the size of the image and creating the accumulator array H.



```
# size of image
X = img.shape[0];
Y = img.shape[1];

angular_sampling = 0.002; # angles in radians
```

The accumulator is initialized by the following code.



```
# initialization of matrix H
rho_max = np.hypot(X,Y);
rho = np.arange(-rho_max, rho_max, 1);
theta = np.arange(0, np.pi, angular_sampling);
cosTheta = np.cos(theta);
sinTheta = np.sin(theta);
H = np.zeros([rho.size, theta.size]);
```

Then, the Hough transform consists in looping over all pixels on edges, and transforming each pixel into a sinus wave. This one is discretized (round and casted to an integer). Notice the line `H[R, range(theta.size)] += 1` which is really tricky and allows us to update the accumulator for all values of the sinusoid in one instruction.



```
# loop on all contour pixels
for i in range(X):
    for j in range(Y):
        if (edges[i,j] != 0):
            R = i*cosTheta + j*sinTheta
            R = np.round(R + rho.size/2).astype(int)
            H[R,range(theta.size)] += 1

plt.imshow(H);
```

26.6.3. Maxima detection

The function peak_local_max from skimage is used to detect local maxima in the Hough transform. Matrix H is first smoothed with a Gaussian filter and represented in Fig.26.3b.

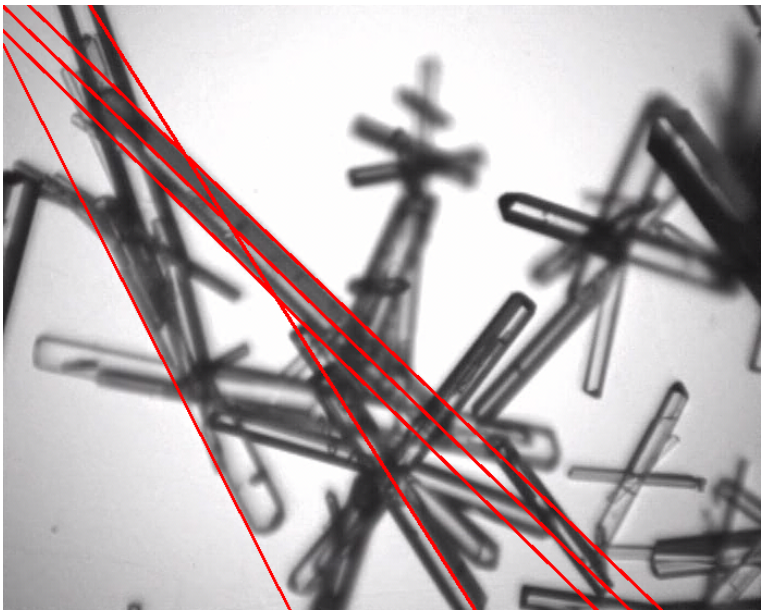


Figure 26.4: Lines detected with the Hough transform.

```
# Maxima detection
G = cv2.GaussianBlur(H, (5,5), 5);
maxima = peak_local_max(H, 5, threshold_abs=150, num_peaks=5);
plt.figure();
plt.imshow(G);

# display maxima on Hough transform image G
plt.scatter(maxima[:,1], maxima[:,0], c='r');
plt.show();
```

26.6.4. Resulting lines

The result is shown in Fig.26.4.

```
# display the results as lines in image
for i_rho, i_theta in maxima:
    print rho[i_rho], theta[i_theta]
    a = np.cos(theta[i_theta])
    b = np.sin(theta[i_theta])
    y0 = a*rho[i_rho]
    x0 = b*rho[i_rho]
    y1 = int(y0 + 1000*(-b))
    x1 = int(x0 + 1000*(a))
    y2 = int(y0 - 1000*(-b))
    x2 = int(x0 - 1000*(a))

    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)

# display in window
cv2.imshow('hough transform', img);
# write resulting image
cv2.imwrite('cv_hough.png', img);
```

26.6.5. OpenCV builtin function



```
import cv2
import numpy as np

# read image and convert it to gray
img = cv2.imread('TestPR46.png')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 100, 200, apertureSize = 3)

# threshold value for lines selection :
# lower value means more lines
threshold = 150;

# perform lines detection
lines = cv2.HoughLines(edges, 1, np.pi /180, threshold)

# display lines
for rho,theta in lines [0]:
    print rho, theta
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    print x1, y1, x2, y2
    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)

cv2.imshow('hough transform', img);
cv2.imwrite('cv_hough.png', img);
```

⋆ 27 Harris corner detector

The aim of this tutorial is to develop a simple Harris corner detector. This is the first step in pattern matching, generally followed by a feature descriptor construction, and a matching process.

27.1. Corner detector and cornerness measure



Use the functions sobel_h, sobel_v and gaussian from the skimage.filters module.

27.1.1. Gradient evaluation

The Harris corner detector is based on the gradients of the image, I_x and I_y in x and y directions, respectively.

?

Apply a Sobel gradient in both directions in order to compute I_x and I_y.

27.1.2. Structure tensor

The structure tensor is defined (for each point of coordinates (u, v)) by the following matrix.

$$M(u,v) = \begin{bmatrix} I_x(u,v)^2 & I_x(u,v)I_y(u,v) \\ I_x(u,v)I_y(u,v) & I_y(u,v)^2 \end{bmatrix}$$

This defines 4 arrays M_1, M_2, M_3, M_4 (that have the same size as the image I), written in a condensed way:

$$M(u,v) = \begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix}$$

?

- Using the Sobel gradients in horizontal and vertical direction, compute M_1 to M_4.
- Use a gaussian filter (with parameter σ) in order to smooth the arrays M_1 to M_4.

27.1.3. Cornerness measure

The cornerness measure C, as proposed by Harris and Stephens, is defined as follows for every pixel of coordinates (u, v):

$$C(u,v) = \det(M(u,v)) - K\text{trace}(M(u,v))^2$$

with K between 0.04 and 0.15.

?

By using the capacity of numpy to make operations on all values of arrays, compute C for all pixels and display it for several scales σ (parameter used in the gaussian filter).

27.2. Corners detection

A so-called Harris corner is the result of keeping only local maxima above a certain threshold value. You can use the checkerboard image for testing, or load the sweden road sign image Fig.27.1.

Use the following function to generate a checkerboard pattern.

```
def checkerboard(nb_x=2, nb_y=2, s=10):
    """
    checkerboard generation
    a grid of size 2*nb_x by 2*nb_y is generated
    each square has s pixels.
    """
    C = 255*np.kron([[1, 0] * nb_x, [0, 1] * nb_x] * nb_y, np.ones((s, s)))
    return C
```

Figure 27.1: Sweden road sign to be used for corner detection, and the expected result.

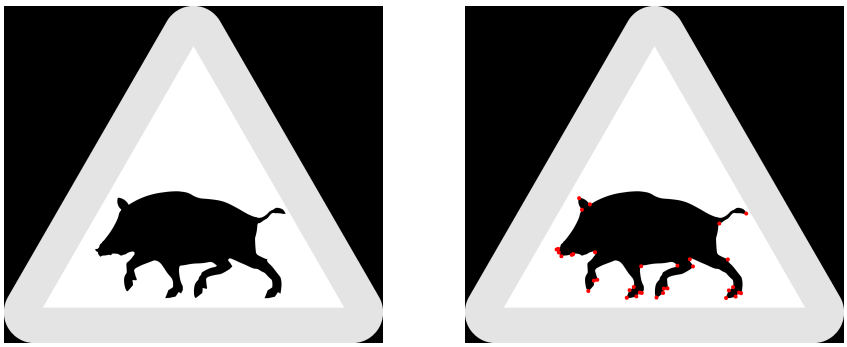
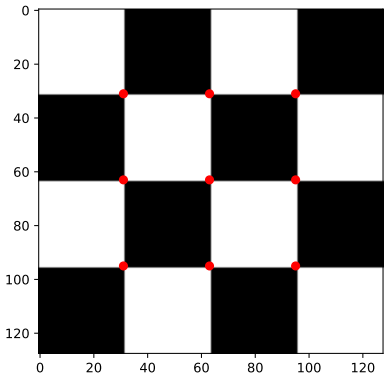


Figure 27.2: Checkerboard, and its expected corner detection.



?

- Evaluate the extended maxima of the image.
- Only the strongest values of the cornerness measure should be kept. Two strategies can be employed in conjonction:
  - Use a threshold value t on C: the choice of this value is not trivial, and it strongly depends on the considered image. An adaptive method would be preferred.
  - Keep only the n strongest values.

- The previous operations are affected by the borders of the image. Thus, eliminate the corner points near the borders.
- The detected corners may contain several pixels. Keep only the centroid of each cluster.



Informations

There are numerous methods for computing local maxima. One can choose between skimage.morphology.local_maxima, skimage.morphology.h_maxima, skimage.feature.peak_local_max.



27.3. Python correction



27.3.1. Cornerness measure

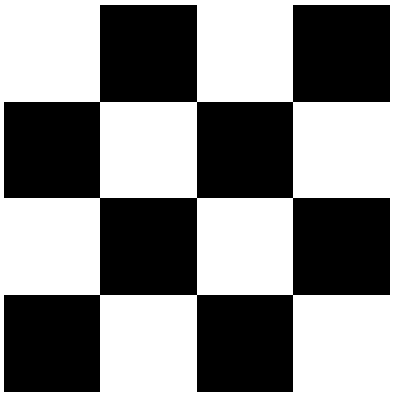
First, load the functions from skimage:



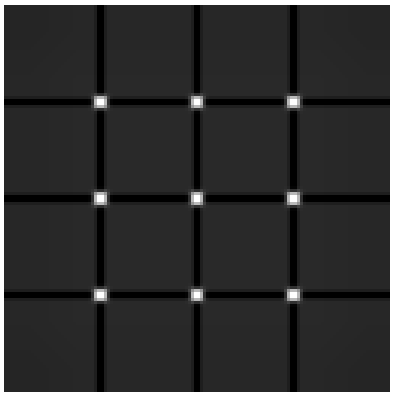
```
from skimage. filters  import gaussian, sobel_h, sobel_v
```

Figure 27.3: Cornerness measure for the checkerboard image. The next step is to locate the corners by thresholding the measure, extracting the local maxima, eliminating points near the edges...

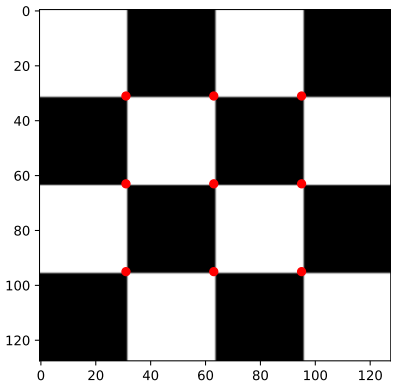
(a) Checkerboard.



(b) Cornerness measure.



(c) Harris corner points.



The first step is to compute the gradient in both x and y directions.



```
Ix = sobel_h(I)
Iy = sobel_v(I)
```

Then, the coefficients of the matrix are computed.



```
M1 = Ix * Ix
M2 = Iy * Ix
M4 = Iy * Iy
```

In case of using a scale parameter, these coefficients should be filtered (for example via a gaussian filter).



```
M1 = gaussian(M1, sigma=sigma)
M2 = gaussian(M2, sigma=sigma)
M4 = gaussian(M4, sigma=sigma)
```

Finally, the cornerness measure is evaluated.



```
K = 0.04
C = (np.multiply(M1, M4) - np.multiply(M2, M2)) - K * np.multiply(M1+M4, M1+M4);
```

The cornerness measure is displayed in Fig.27.3.

27.3.2. Corners detection

In order to keep only the strongest corner points, a threshold value t is applied on C. This value is really depending on the considered image, thus such a global threshold is not generally a good idea. One would probably prefer a h-maximum or equivalent operator. For the purpose of this tutorial, we will keep this strategy.



```
C[C<t] = 0
```

The local maxima are then extracted.



```
local_maxi = peak_local_max(C, min_distance=2)
corners = np.zeros_like(C, dtype=bool)
# mark the corners
corners[tuple(local_maxi.T)] = True
```

The result is a binary image, where some clusters of points are the corner points. To keep only one point per cluster, the centroid of each is detected. The final result is presented in Fig.27.3c.



```
L = measure.label(corners)
props = measure.regionprops(L)
centers =[]
for prop in props:
    centers.append(prop.centroid)
```

27.3.3. Road sign image application

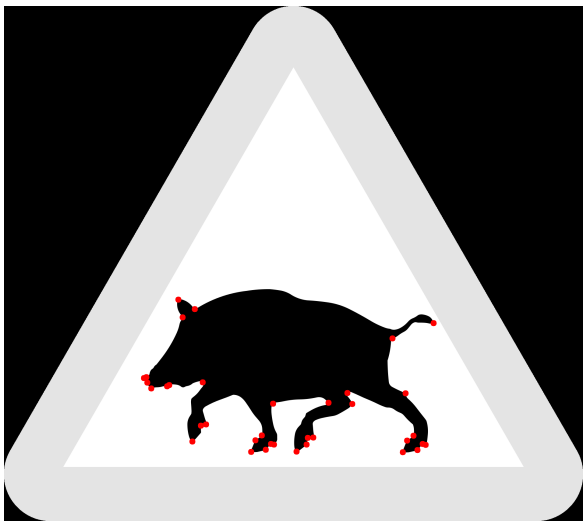
In this case, the values $t = 10^7$ and $\sigma = 3$ are used. The result is illustrated in Fig.27.4.

Figure 27.4: Harris corner detection with $K = 0.04$, scale $\sigma = 3$ and threshold value $t = 10^7$.

(a) Cornerness measure.



(b) Corner points.



28 Elliptic Fourier Descriptors

This tutorial introduces the elliptic Fourier descriptors (EFD), which decompose a closed contour into a sum of ellipses. The coefficients of these ellipses are a compact description of the shape: they are used here to reconstruct a contour with a controlled level of detail, then to recognize shapes automatically.

28.1. Definition

The parametric equation of an ellipse of centre (x_C, y_C), of semi-axes a and b and of orientation θ is, for $t \in \mathbb{R}$:

$$\begin{cases} x(t) = x_C + a\cos\theta\cos t - b\sin\theta\sin t \\ y(t) = y_C + a\sin\theta\cos t + b\cos\theta\sin t \end{cases}$$

Let now a *closed* curve be parameterized by t, through the points of coordinates $(x(t), y(t))$. Its approximation by N ellipses is written:

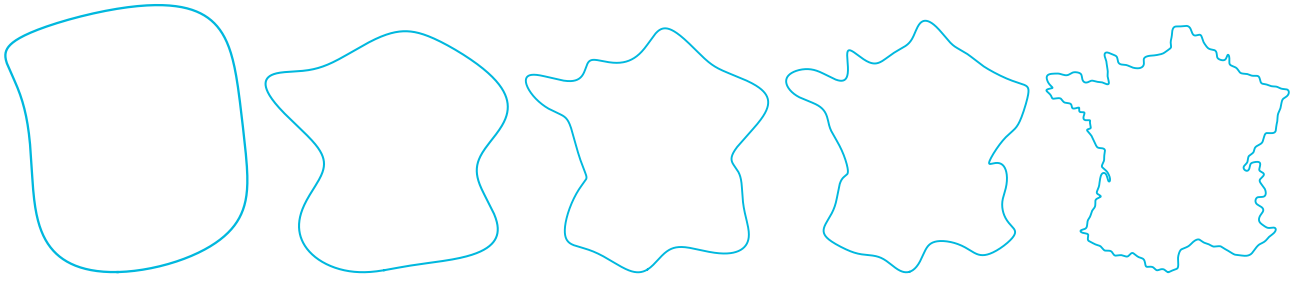
$$x(t) = x_C + \sum_{n=1}^{N} a_n \cos\left(\frac{2\pi nt}{T}\right) + b_n \sin\left(\frac{2\pi nt}{T}\right) \quad (28.1)$$

$$y(t) = y_C + \sum_{n=1}^{N} c_n \cos\left(\frac{2\pi nt}{T}\right) + d_n \sin\left(\frac{2\pi nt}{T}\right) \quad (28.2)$$

where n is the rank of the harmonic (of the ellipse), N the total number of harmonics, T the period that describes the contour, and t the parameter of a point along the contour, $0 \leq t \leq T$.

Each rank n therefore contributes four coefficients (a_n, b_n, c_n, d_n), which together describe one ellipse. The constant term (x_C, y_C) is the centre of gravity of the contour; it carries the position of the shape, and nothing about its form.

Figure 28.1: Approximation of the contour of France with, from left to right, 2, 5, 10 and 20 harmonics. The last figure is the original contour.



The module pyefd implements these descriptors. The useful functions are elliptic_fourier_descriptors, calculate_dc_coefficients, reconstruct_contour and plot_efd. It is installed with pip install pyefd.

28.2. Contour of a shape

The descriptors are computed on a contour, that is to say on an ordered list of points, and not on an image.

?

1. Load the binary image france.png and extract its contour.
2. Display the contour as a curve, and check that the order of the points follows the boundary of the shape.
3. Compute the centre of gravity of the contour.

Informations

The function skimage.measure.find_contours returns a list of contours, each of them being an array of coordinates (row, column). Beware of the difference of convention between an image (the first axis points downwards) and a plot (the second axis points upwards): the image can be flipped with np.flipud, and the two columns swapped with np.fliplr.

28.3. Decomposition and reconstruction

?

1. Compute the elliptic Fourier descriptors of the contour, for a large number of harmonics (100 for example). What is the size of the result?
2. Reconstruct the contour with 2, 5, 10 and 20 harmonics, and superimpose each reconstruction on the original contour (Fig. 28.1). How many harmonics are needed for the shape to become recognizable?

The reconstruction is centred on the origin if the constant term is forgotten. The function reconstruct_contour takes it as the argument locus.

28.4. Sum of the ellipses

In order to really understand Eq. 28.1 and 28.2, the reconstruction is now computed by hand, ellipse after ellipse.

?

1. Sample the parameter t with n points, $t \in [0, 1]$, and compute the coordinates of the points of *each* ellipse, for every rank.
2. Display a single ellipse, for example the one of rank 3. Compare the sizes of the ellipses of the first ranks.
3. Obtain the approximations by a *cumulative sum* of the ellipses, and display a few of them.

numpy.cumsum computes the cumulative sum along one axis: the column k of the result is the sum of the columns 0 to k of the input, that is to say exactly the approximation with $k + 1$ harmonics.



28.5. Python correction



This correction uses the module pyefd, skimage for the images and the contours, and sklearn for the classification.



```
import glob
import numpy as np
import matplotlib.pyplot as plt

from skimage import measure, io

from pyefd import (elliptic_fourier_descriptors, plot_efd,
                   reconstruct_contour, calculate_dc_coefficients)

from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sn
import pandas as pd
```

28.5.1. Contour of a shape

The two conventions (image and plot) are reconciled once and for all inside a small function: the image is flipped upside down, and the coordinates $(row, column)$ returned by find_contours are swapped to become (x, y). The contour can then be plotted directly, and the descriptors are computed in a direct cartesian frame.



```
def get_contour(filename):
    """
    Contour of a binary shape, as an array of (x, y) points.

    filename: name of the binary image
    returns: array of size (nb_points, 2)
    """
    I = np.flipud(io.imread(filename) > 128)
    contours = measure.find_contours(I, 0.8)
    contour = max(contours, key=len)        # the longest one
    return np.fliplr(contour)
```



Only the *frontier* between the two phases matters here, not the convention used for the object: the contour of a white shape on a black background and the contour of the same shape in black on white are the same curve.

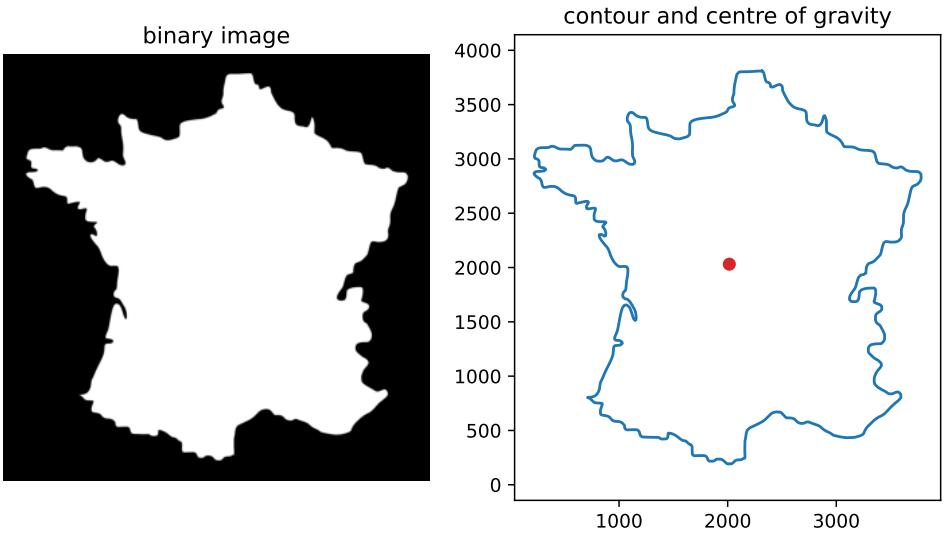
The constant term of the decomposition is the centre of gravity of the contour (Fig. 28.2).



```
contour = get_contour("france.png")
centre = calculate_dc_coefficients(contour)
print("number of points of the contour:", contour.shape[0])
print("centre of gravity: ({0:.0f}, {1:.0f})".format(*centre))
```

```
number of points of the contour: 20447
centre of gravity: (2015, 2031)
```

Figure 28.2: The binary image, and the contour extracted from it with its centre of gravity.



28.5.2. Decomposition and reconstruction

The descriptors are returned as an array of size (order, 4): one line per harmonic, and the four coefficients (a_n, b_n, c_n, d_n) in the columns.

```python
order = 100
coeffs = elliptic_fourier_descriptors (contour, order=order)
print(" coefficients :", coeffs.shape)
print(coeffs [:3, :])
```

```
coefficients : (100, 4)
[[ 310.40515651 -1498.95757972  1491.86232309   509.26454183]
 [ -66.71420852   -99.19857072   -62.26677536  -133.89993073]
 [ 188.22120524   -89.95055218    79.84239257   -69.83314785]]
```

The function plot_efd draws the successive truncations of the series (Fig. 28.3), and reconstruct_contour rebuilds the contour from a limited number of harmonics (Fig. 28.4). The argument locus places the reconstruction back on the original shape.

```
fig = plt.figure(figsize=(14, 5))
plot_efd(coeffs[:10, :])

for N in [2, 5, 10, 20]:
    C = reconstruct_contour(coeffs[:N], locus=centre, num_points=1000)
    plt.plot(C[:, 0], C[:, 1])
```

Figure 28.3: Successive truncations of the series, from 1 to 10 harmonics.

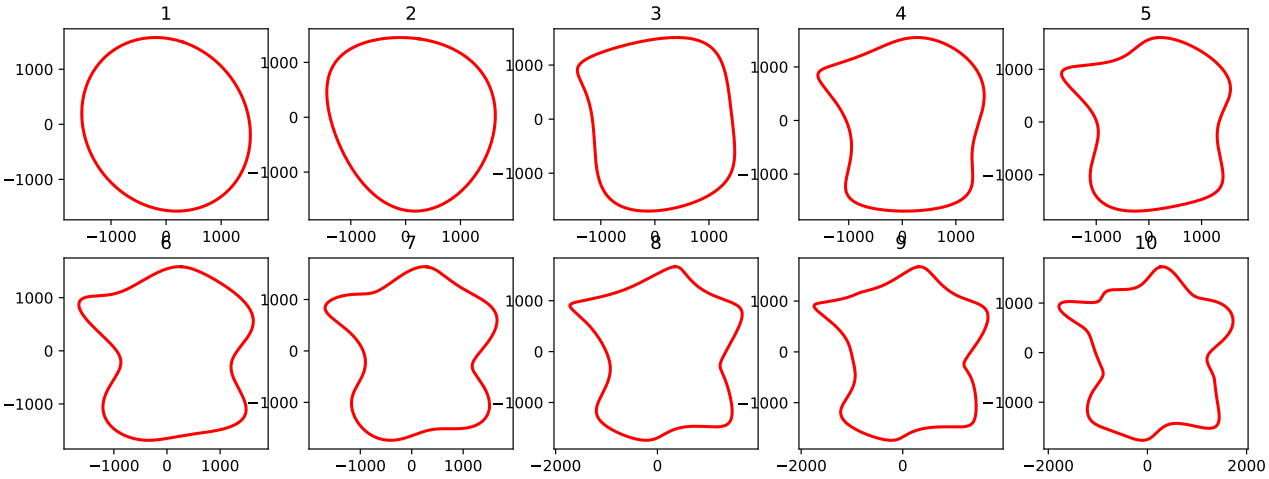
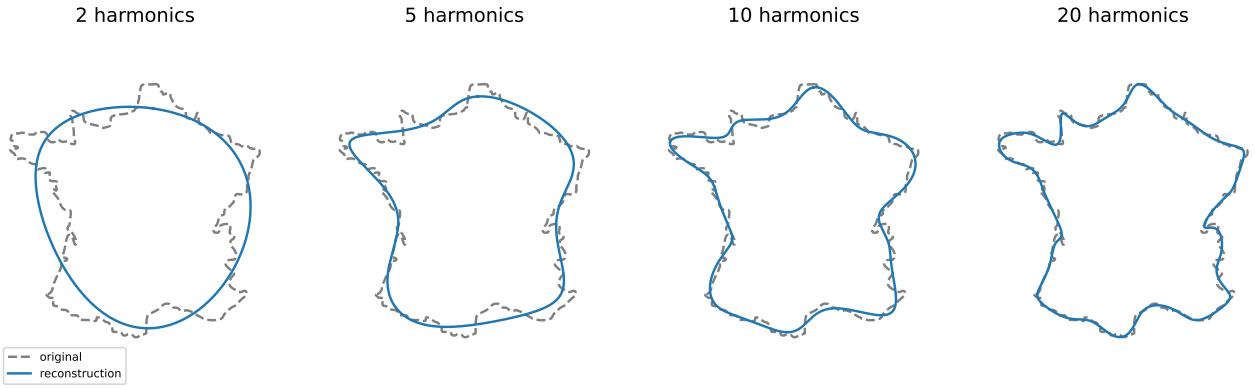


Figure 28.4: Reconstruction of the contour with an increasing number of harmonics. Five ellipses are enough to recognize the shape, twenty restore the main capes and bays.



28.5.3. Sum of the ellipses

The reconstruction is now written explicitly, in order to see the ellipses one by one. Each column of X_ellipse contains the abscissas of one ellipse, sampled with n points.

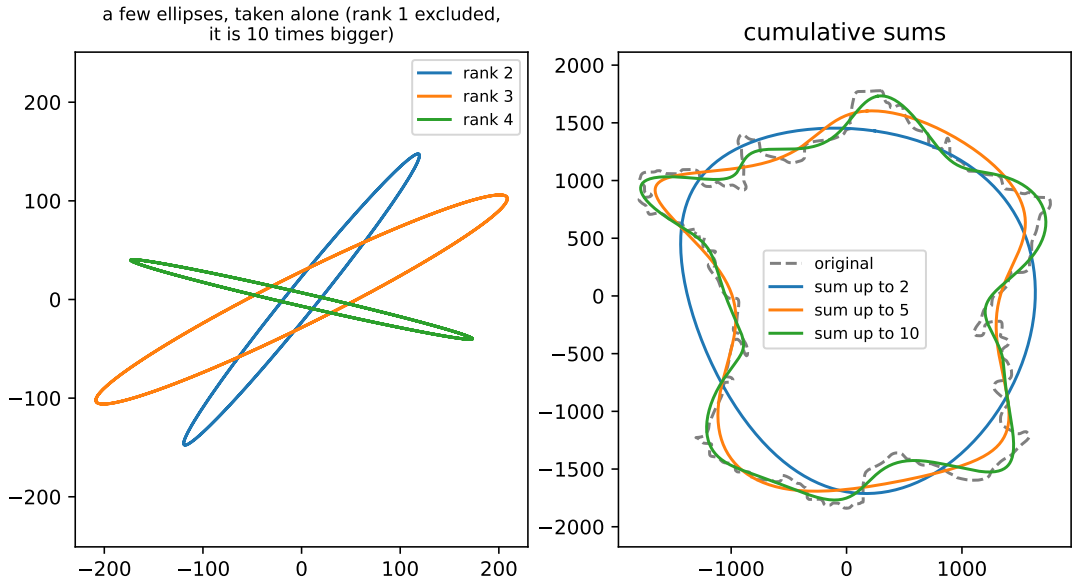
```python
n = 3000                          # number of points of each ellipse
T = np.linspace(0, 1.0, n)

X_ellipse = np.zeros((n, order))
Y_ellipse = np.zeros((n, order))
for k in range(order):
    X_ellipse[:, k] = (coeffs[k, 0] * np.cos(2*(k+1)*np.pi*T)
                       + coeffs[k, 1] * np.sin(2*(k+1)*np.pi*T))
    Y_ellipse[:, k] = (coeffs[k, 2] * np.cos(2*(k+1)*np.pi*T)
                       + coeffs[k, 3] * np.sin(2*(k+1)*np.pi*T))

# an approximation is a cumulative sum of the ellipses
X = np.cumsum(X_ellipse, axis=1)
Y = np.cumsum(Y_ellipse, axis=1)
```

The first ellipse is by far the largest: it carries the global shape, and the following ones only add details of decreasing amplitude (Fig. 28.5). The column k of the cumulative sum is exactly the approximation with $k+1$ harmonics.

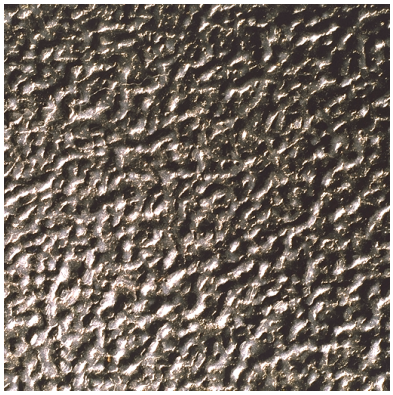
Figure 28.5: Left: three ellipses taken alone. Right: the cumulative sums, superimposed on the original contour centred on the origin.



29 Local Binary Patterns

This tutorial aims to study a texture descriptor named 'Local Binary Patterns'. The first objective is to implement this descriptor. Therefater, digital images of textures will be classified using this descriptor and the k-means algorithm.

The different processes will be applied on this kind of texture images:



(a) metal image



(b) sand image



(c) ground image

29.1. Local Binary Patterns

The Local Binary Patterns (LBP) descriptor is a simple yet very efficient texture operator which labels the pixels of an image by thresholding the neighborhood of each pixel and considers the result as a binary number. Due to its discriminative power and computational simplicity, LBP texture operator has become a popular approach in various applications. It can be seen as a unifying approach to the traditionally divergent statistical and structural models of texture analysis. Perhaps the most important property of the LBP operator in real-world applications is its robustness to monotonic gray-scale changes caused, for example, by illumination variations. Another important property is its computational simplicity, which makes it possible to analyze images in challenging real-time settings.
The LBP feature vector, in its simplest form, is created in the following manner:

- For each pixel, compare the pixel to each of its 8 neighbors (on its left-top, left-middle, left-bottom, right-top, etc.). Follow the pixels along a circle, i.e. clockwise or counter-clockwise.
- Where the center pixel's value is greater than the neighbor's value, write "1". Otherwise, write "0". This gives an 8-digit binary number (which is usually converted to decimal for convenience).
- Compute the histogram of the frequency of each "number" occurring (i.e., each combination of which pixels are smaller and which are greater than the center).
- Normalize the histogram.

?

- Code a function for computing the Local Binary Patterns.
- Test this operator on a texture image from the given database.

Consider the function numpy.histogram for histogram computation.

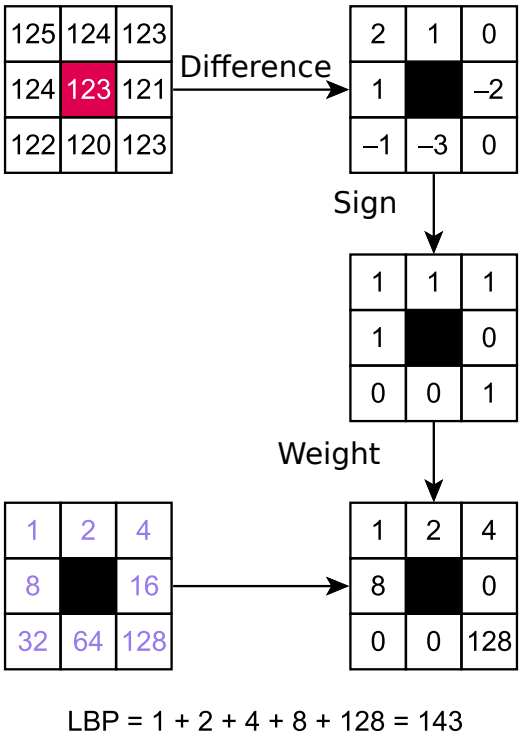


Figure 29.1: Local binary pattern. From wikipedia, author Xiawi, CC-By-SA.

29.2. Classification of texture images

The objective is to classify the texture images from the given databse by using the LBP descriptor.

?

1. Calculate the LBP descriptor for each image of the database.
2. Compare the descriptors for each class of images.
3. Compute the distance between each pair of images in order to get a dissimilarity matrix. Comment the result.
4. Use the k-means algorithm to classify the images of the database into three classes ($k = 3$).

See KMeans from sklearn . cluster .



29.3. Python correction



The following imports are used.



```
import numpy as np
import matplotlib.pyplot as plt
import glob
import seaborn as sn
import pandas as pd
import os
from sklearn.cluster import KMeans
```

29.3.1. LBP computation

Each pixel is given a specific 8 bits value according to a code as follows.



```
def LBP(I):
    B = np.zeros_like(I)
    code = np.array    ([[1,2,4],[8,0,16],[32,64,128]])

    # loop over all pixels except border pixels
    for i in np.arange(1,I.shape[0]-2):
        for j in np.arange(1, I.shape[1]-2):
            w = I[i-1:i+2, j-1:j+2]
            w = w >= I[i,j]
            w = w * code
            B[i,j] = np.sum(w)
```

Then, all values (except for border values) are summarized in the histogram. The density parameter is set to 1 in order to normalize the values (this is a density of probability).

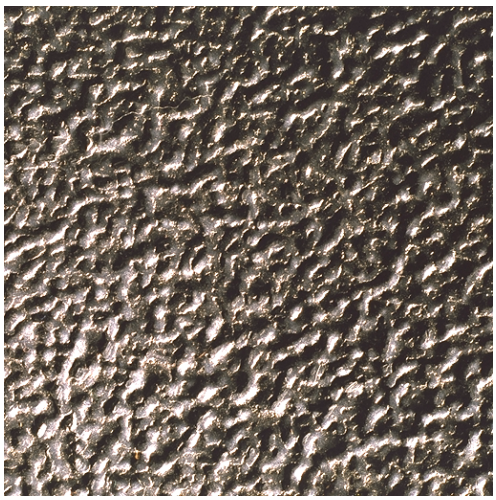


```
h,edges = np.histogram(B[1:-1, 1:-1], density=True, bins=256);
```

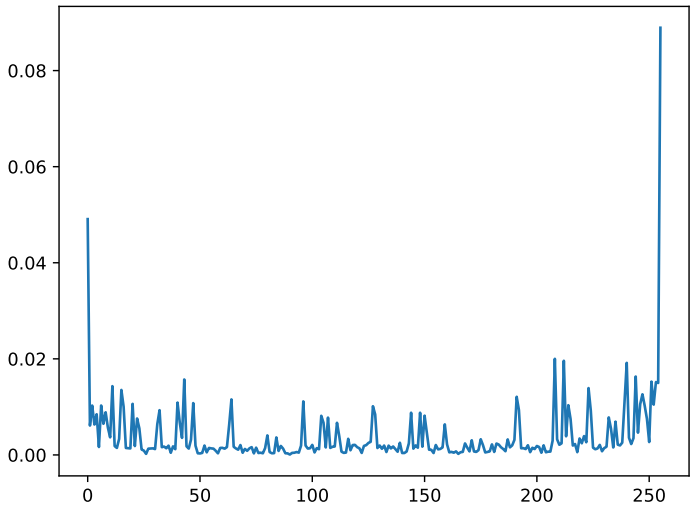
For the first image of sand, the histogram is shown in Fig.29.2.

Figure 29.2: Illustration of the Local Binary Pattern computed on an entire image.

(a) Texture image.



(b) LBP of texture.



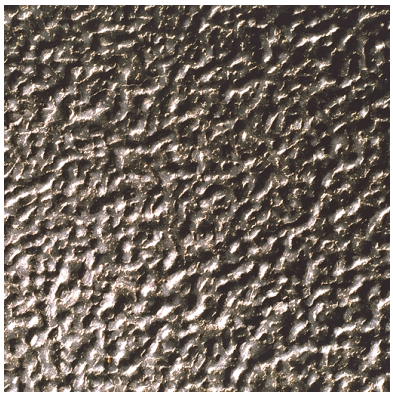
29.3.2. Classification

For all images of the same family, the LBP are computed and represented in the same graph. The histograms really look similar (see Fig.29.3). The following code is used for the “sand” family.

```
classes = ['Terrain', 'Metal', 'Sand']
names = []
hh = []
for c in classes:
    print(c)
    fig=plt.figure()
    for file in sorted(glob.glob('../matlab/images/'+c+'*.bmp')):
        names.append(os.path.basename(file))
        I = imageio.imread(file)
        I = I[:,:,1]
        h, edges = LBP(I)
        plt.plot(h)
        plt.show()
        hh.append(h)
```

Figure 29.3: Illustration of the LBP of 4 images of each family. The histogram are almost equivalent, which shows that this descriptor can be employed to discriminate between the different families.

(a) Metal image example.



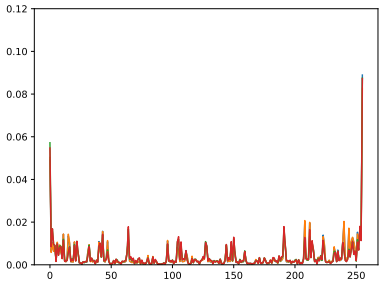
(b) Sand image example.



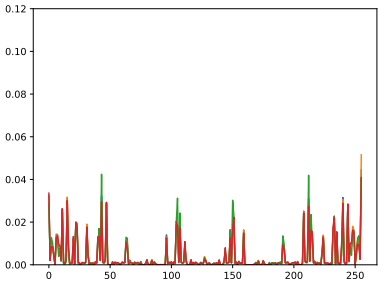
(c) Terrain image example.



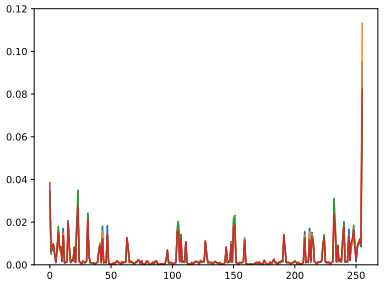
(d) Four metal images.



(e) Four sand images.



(f) Four terrain images.



A distance criterion is used to compare the different histograms: the classical SAD (Sum of Absolute Differences) gives a numerical values. All pairs of distances are concatenated in a matrix, displayed as an image in Fig.29.4.

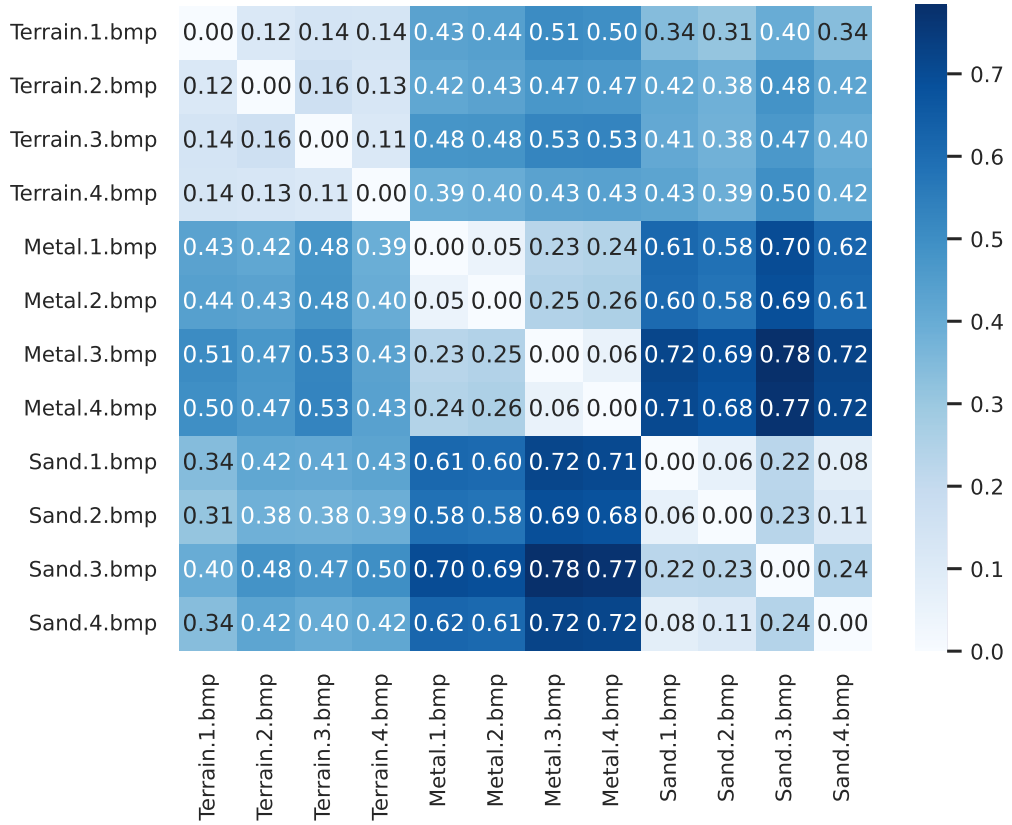


```
# compute distance between LBPs
n = len(hh)
dists = np.zeros((n, n))
for i in np.arange(n):
    for j in np.arange(n):
        dists[i,j] = np.sum(np.abs(hh[i]-hh[j]))
fig = plot_dists(dists, names)
```

In order to display this matrix, the module seaborn is used.

```python
def plot_dists(dists, classes, cmap=plt.cm.Blues):
    """
    Plot matrix of distances
    dists: all computed distances
    classes: labels to be used
    cmap: colormap
    returns: figure that can be used for pdf export
    """
    df_cm = pd.DataFrame(dists, index = classes, columns = classes)
    fig = plt.figure()
    sn.set(font_scale=.8)
    sn.heatmap(df_cm, annot=True, cmap = cmap, fmt='.2f')
    return fig
```

Figure 29.4: Sum of Absolute Differences between the different LBP histograms of each image. 3 families of 4 textures are represented here, terrain images are in the first part, metal images in the second and sand images in the last. Black represents 0 distance and white is 1 (highest distance, the values are normalized).



The kmeans algorithm uses such a distance (L_2 norm, euclidean distance by default), and we can verify that the clustering process works as expected.

```python
# kmeans clustering
n=3;
k_means = KMeans(init='k-means++', n_clusters=n, n_init=10)
k_means.fit(hh)
print(k_means.labels_)
```

The result show that the kmeans algorithm perfectly performs the classification. The indices are not sorted, but the reading of the dataset is done material by material, thus we recognize groups of 4 images of the same material.

```
[1 1 1 1 0 0 0 0 2 2 2 2]
```

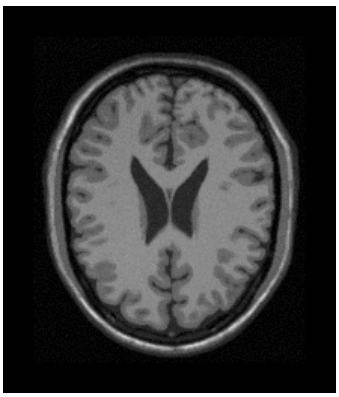
30 Image Registration

This tutorial aims to implement the Iterative Closest Point (ICP) method for image registration. More specifically, we are going to estimate a rigid transformation (translation + rotation without scaling) between two images.

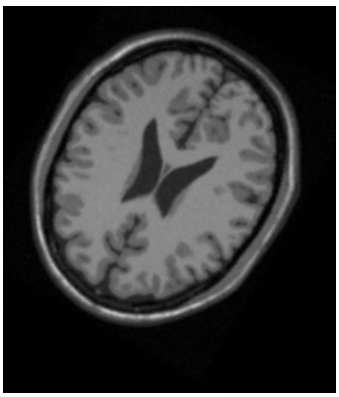
The different processes will be applied on T1-MR images of the brain in Fig.30.1.

Figure 30.1: Original images.

(a) brain1



(b) brain2



30.1. Transformation estimation

A classical method in image registration first consists in identifying and matching some characteristic points by pairs. Thereafter, the transformation is estimated from this list of pairs (displacement vectors).

30.1.1. Preliminaries

Pairs of points are first manually selected.

- Read and visualize the two MR images 'brain1' and 'brain2' (moving and source images).
- Manually select a list of corresponding points.

There is no built-in function for manual selection of pairs of points. You can use the following points or code your own function, see tutorial 22

```python
# define control points
A_points = np.array([[136, 100], [127, 153], [96, 156], [87, 99]]);
B_points = np.array([[144, 99] , [109, 140], [79, 128], [100, 74]]);
```

30.1.2. Rigid transformation

With this list of pairs $(p_i, q_i)_i$, this tutorial proposes to estimate a rigid transformation between these points. It is composed of a rotation and a translation (and not scaling). For doing that, we make a Least Squares (LS) optimization, which is defined as follows. The parameters of the rotation R and the translation t minimize the following criterion:

$$C(R,t) = \sum_i ||q_i - R.p_i - t||^2$$

Calculation of the translation :
The optimal translation is characterized by a null derivative of the criterion:

$$\frac{\partial C}{\partial t} = -2\sum_i (q_i - R.p_i - t)^{\mathrm{T}} = 0 \Leftrightarrow \sum_i q_i - R.\left(\sum_i p_i\right) = N.t$$

where N denotes the number of matching pairs. By denoting $\bar{p} = \frac{1}{N}\sum_i p_i$ and $\bar{q} = \frac{1}{N}\sum_i q_i$ the barycenters of the point sets and by changing the geometrical referential: $p'_i = p_i - \bar{p}$ and $q'_i = q_i - \bar{q}$, the criterion can be written as:

$$C'(R) = \sum_i ||q'_i - R.p'_i||^2$$

The estimated rotation $\hat{R}$ will provide the expected translation:

$$\hat{t} = \bar{q} - \hat{R}.\bar{p} \tag{30.1}$$

calculation of the rotation by the SVD method:
We will use the following theorem: Let $U.D.V^{\mathrm{T}} = K$ a singular decomposition of the correlation matrix $K = {q'}^{\mathrm{T}}.p'$, for which the singular values are sorted in the increasing order. The minimum of the criterion: $C(R) = \sum_i ||q'_i - R.p'_i||^2$ is reached by the matrix :

$$\hat{R} = U.S.V^{\mathrm{T}} \tag{30.2}$$

with $S = \begin{pmatrix} 1 & 0 \\ 0 & \det(U).\det(V) \end{pmatrix}$.

30.1.3. Summary of rigid transformation evaluation

To summarize this algorithm:

1. Compute the center of each set of points $\bar{p}$ and $\bar{q}$
2. Subtract them to their associated points $p'_i = p_i - \bar{p}$ and $q'_i = q_i - \bar{q}$
3. Compute the (2×2) matrix $K = {q'}^{\mathrm{T}}.p'$
4. Use the SVD decomposition to get U, D and V
5. Evaluate the rotation $\hat{R}$
6. Evaluate the translation $\hat{t} = \bar{q} - \hat{R}.\bar{p}$

30.1.4. Questions

?

- Code a function that takes as parameters the pairs of points, and returns the rigid transformation elements of rotation $\hat{R}$ and translation $\hat{t}$ as defined in Eqs.30.1 and 30.2.
- Apply the transformation to the moving image.

- Visualize the resulting registered image.



- See svd function from scipy.linalg or numpy.linalg.
- See cv2.warpAffine and cv2.transform for transformation application. Import the module OpenCV with import cv2.

30.2. ICP-based registration

When the points are not correctly paired, it is first necessary to reorder them before estimating the transformation. In this way, the ICP (iterative Closest Points) consists in an iterative process of three steps: finding the correspondence between points, estimating the transformation and applying it. The process should converge to the well registered image.



To simulate the mixing of the points, randomly shuffle them selected on the first image and perform the registration with the previous method.



See np.random.permutation.



1. From the list of the characteristic points p of the image 'brain1', find the nearest neighbors q in the image 'brain2'. Be careful to the order of the input arguments.
2. Estimate the transformation T by using the LS minimization coded previously.
3. Apply this transformation to the points p, find again the correspondence between these resulting points $T(p)$ and q and estimate a new transformation. Repeat the process until convergence.
4. Visualize the resulting registered image.



Look at scipy.spatial.cKDTree for nearest neighbor search.

30.3. Automatic control points detection

The manual selection of points may be fastidious. Two simple automatic methods for detecting control points are the so-called Harris corners detection or Shi-Tomasi corner detector.



The function goodFeaturesToTrack returns the corners from the Shi-Tomasi method.



Replace the manual selection in the two previous parts of this tutorial.

Notice that the automatic points detection does not ensure to give the same order in the points of the two images. More generally, other salient points detectors do not give the same number of points, and thus the algorithms have to remove outliers (non matching points). This case is not taken into account in this tutorial.

30.4. Python correction

```
from scipy import misc
import matplotlib.pyplot as plt
import numpy as np
import cv2
from scipy.spatial import cKDTree
```

30.4.1. Transformation estimation based on corresponding points

Image visualization

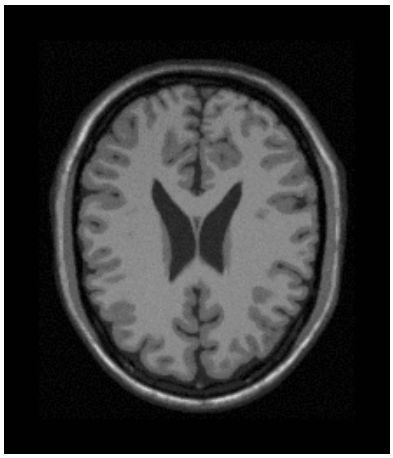
The following code is used to display the images (see Fig.30.2).

```
# Read images and display
A=imageio.imread("brain1.bmp")
B=imageio.imread("brain2.bmp")
plt.imshow(A,cmap='gray');
plt.show();
plt.imshow(B,cmap='gray');
plt.show();

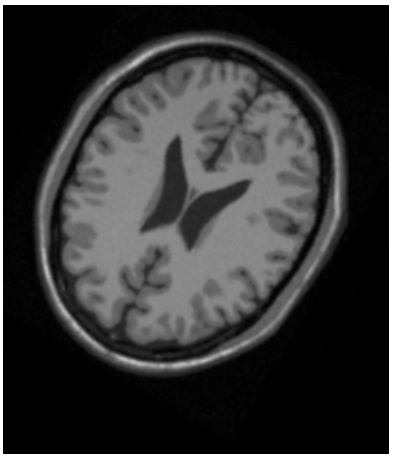
# define control points
A_points = np.array([[136, 100], [127, 153], [96, 156], [87, 99]]);
B_points = np.array([[144, 99], [109, 140], [79, 128], [100, 74]]);
```

Figure 30.2: Initial images.

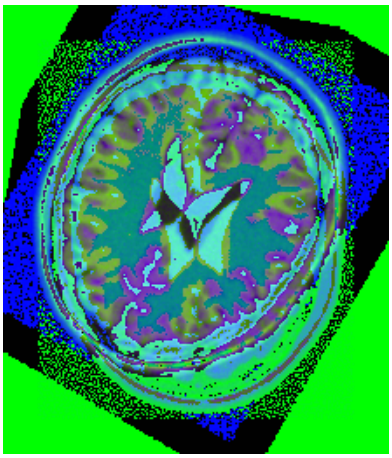
(a) Moving image.



(b) Source image.



(c) Superimposition.



If you want to display and save the fusion of both images, you can use this function:



```
def superimpose(G1, G2, filename=None):
    """
    superimpose 2 images, supposing they are grayscale images and
    of same shape.
    For display purposes.
    """
    r,c=G1.shape;
    S = np.zeros((r,c,3));
    S [:,:,0]  = np.maximum(G1-G2, 0)+G1;
    S [:,:,1]  = np.maximum(G2-G1, 0)+G2;
    S [:,:,2]  = (G1+G2) / 2;
    S = 255 * S / np.max(S);
    S = S.astype('uint8');
    plt.imshow(S);
    plt.show()
    if filename!=None:
        cv2.imwrite(filename, S);
    return S
```

Manual selection of corresponding points

With openCV, there is not built-in function to make a manual selection of pairs of control points. The following code uses a global variable I in order to manage the display of the points, which are finally stored into the pts variable. First, the callback function on_mouse is defined to handle mouse event.



```
pts = [];
def on_mouse(event, x, y, flags, param):
    """
    callback method for detecting click on image
    It draws a circle on the global variable image I
    """
    global pts, I;
    if event == cv2.EVENT_LBUTTONUP:
        pts.append((x, y));
        cv2.circle(I,(x,y), 2, (0,0,255), -1)
```

The function cpselect allows the selection of multiple points.



```
def cpselect():
    """
    method for manually selecting the control points
    It waits until 'q' key is pressed.
    """
    cv2.namedWindow("image")
    cv2.setMouseCallback("image", on_mouse)
    print("press 'q' when finished")
    # keep looping until the 'q' key is pressed
    while True:
      # display the image and wait for a keypress
      cv2.imshow("image", I)
      key = cv2.waitKey(1) & 0xFF

      # if the 'c' key is pressed, break from the loop
      if key == ord("q"):
          break

    # close all open windows
    cv2.destroyAllWindows()
    return pts;
```

Transformation estimation

The rigid transformation is estimated from the corresponding points by the following function:



```
def rigid_registration(data1, data2):
    """
    Rigid transformation estimation between n pairs of points
    This function returns a rotation R and a translation t
    data1 : array of size nx2
    data2 : array of size nx2
    returns transformation matrix T of size 2x3
    """
    data1 = np.array(data1);
    data2 = np.array(data2);

    # computes barycenters, and recenters the points
    m1 = np.mean(data1,0);
    m2 = np.mean(data2,0);
     data1_inv_shifted = data1-m1;
     data2_inv_shifted = data2-m2;

    # Evaluates SVD
    K = np.matmul(np.transpose(data2_inv_shifted), data1_inv_shifted);
    U,S,V = np.linalg.svd(K);

    # Computes Rotation
    S = np.eye(2);
    S[1,1] = np.linalg.det(U)*np.linalg.det(V);
    R = np.matmul(U,S);
    R = np.matmul(R, np.transpose(V));

    # Computes Translation
    t = m2-np.matmul(R, m1);

    T = np.zeros((2,3));
    T[0:2,0:2] = R;
    T[0:2,2] = t;
    return T;
```

Then, you can apply this function to the manually selected points. The transformation is applied with the following code, using OpenCV.



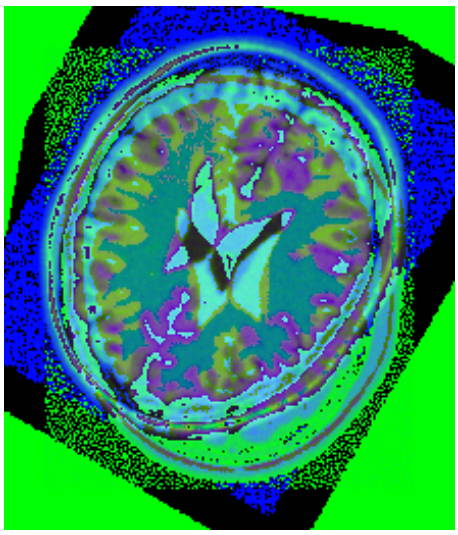
```
import cv2
def applyTransform(points, T):
    """
    Apply transform to a list of points
    points: list of points
    T: rigid transformation matrix (shape 2x3)
    """
    dataA = np.array(points)
    src = np.array([dataA])
    data_dest = cv2.transform(src, T)
    a, b, c = data_dest.shape
    data_dest = np.reshape(data_dest, (b, c))
    return data_dest
```

```
# 1st case, rigid registration, with pairs of points in the correct order
T = rigid_registration(A_points, B_points);

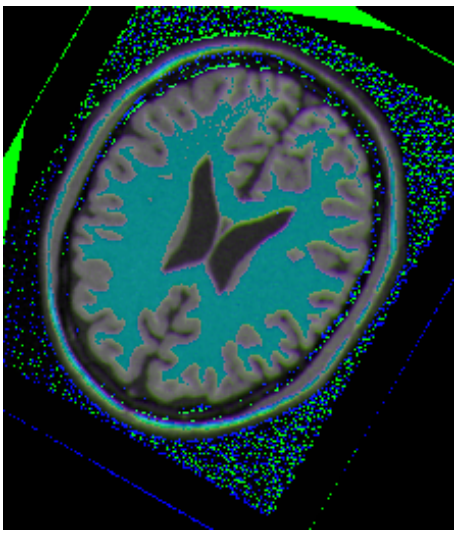
# Apply transformation on control points and display the results
data_dest = applyTransform(A_points, T);
I = B.copy();
for pb,pa in zip(data_dest, B_points):
    cv2.circle(I, totuple(pa), 1, (255,0,0), -1)
    cv2.circle(I, totuple(pb), 1, (0,0,255), -1)
plt.imshow(I);
plt.show();
# Apply transformation on image
rows, cols= B.shape;
dst = cv2.warpAffine(A,T, (cols, rows));
plt.imshow(dst);
superimpose(dst, B, "rigid_manual.png");
```

Figure 30.3: Result of the registration for the manually selected control points.

(a) Without registration.



(b) With registration.



The result is good, because the manual selection of the points is good (the points are given in the correct order for both images).

30.4.2. ICP registration

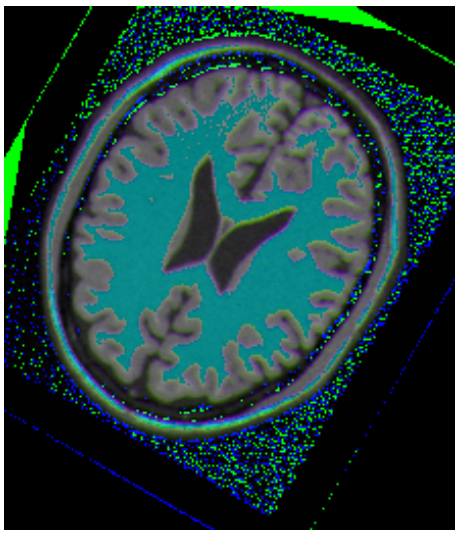
Random permutation of points

The following code randomly shuffles the points of the first vector. The result is of course a wrongly registered image, see Fig.30.4.

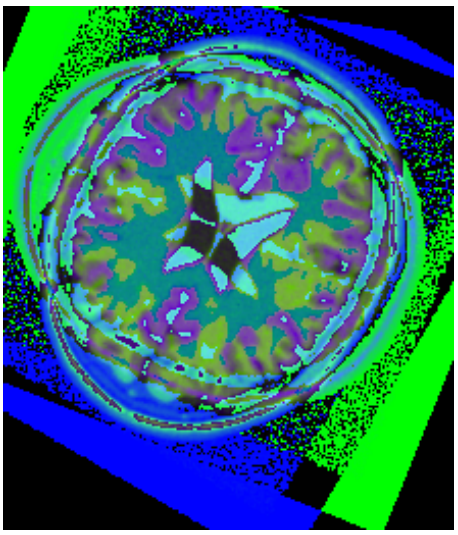
```
# random permutation of the points
p = np.random.permutation(np.arange(4));
A_points = A_points[p];
T = rigid_registration(A_points, B_points);
```

Figure 30.4: Result of the registration for when the control points are not found in the same order.

(a) Matching pairs of points.



(b) Permulation of the points.



ICP

The previous operations simulates a general non-manual selection of the control points: there is not reason to finding the points by matching pairs. Thus, a reordering is necessary. This propositions implies that the number of points is exactly the same in order to perform the registration process, and that these are matching points (every point has a matching point in the other image). The ICP method (see code for icp_transform) reorders the points via a nearest neighbor rule.

```
def icp_transform(dataA, dataB):
    """
    Find a transform between A and B points
    with an ICP ( Iterative Closest Point) method.
    dataA and dataB are of size nx2, with the same number of points n
    returns the transformation matrix of shape 2x3
    """
    data2A = dataA.copy();
    data2B = np.zeros(data2A.shape);
    T = np.zeros ((2,3) );
    T [0:2,0:2] = np.eye(2);
    T [0:2,2] = 0;

    nb_loops=5;
    tree = cKDTree( dataB );

    for loop in range(nb_loops):
        # search for closest points and reorganise array of points accordingly
        d, inds = tree.query( data2A);

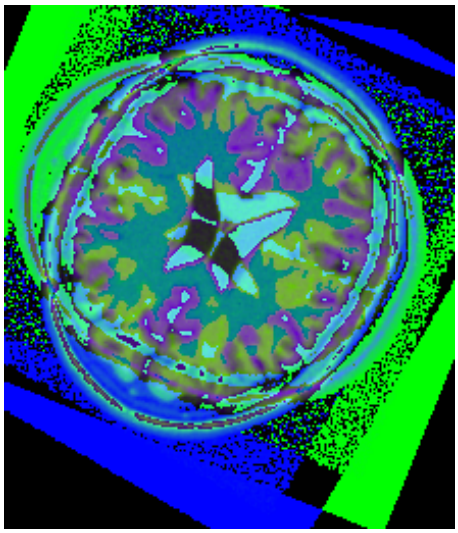
        data2B = dataB[inds ,:];
        # find rigid registration with reordered points
        t_loop = rigid_registration (data2A, data2B);

        T = composeTransform(t_loop, T);
        # evaluates transform on control points, to make the iteration
        data2A = applyTransform(dataA, T);

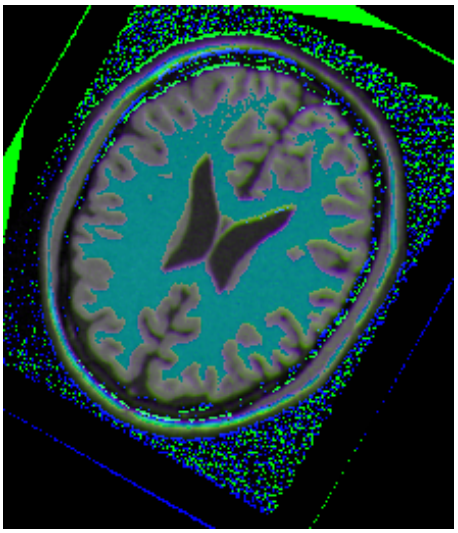
    return T;
```

Figure 30.5: Result of the registration for when the control points are not found in the same order. The ICP algorithm reorders the points and gives a good result.

(a) Random shuffle of points and direct rigid transformation estimation.



(b) ICP registration on the same points.



Automatic extraction of corner points

Generally, the points are automatically detected, and thus, there is no warranty that they are found in the same order, nor that each pair of point correspond to matching points (some points –called outliers– need to be eliminated to compute the correct transformation). In this tutorial, we do not address the problem of outliers. Please notice that with these parameters and images, by chance, the same points are detected in the correct order.

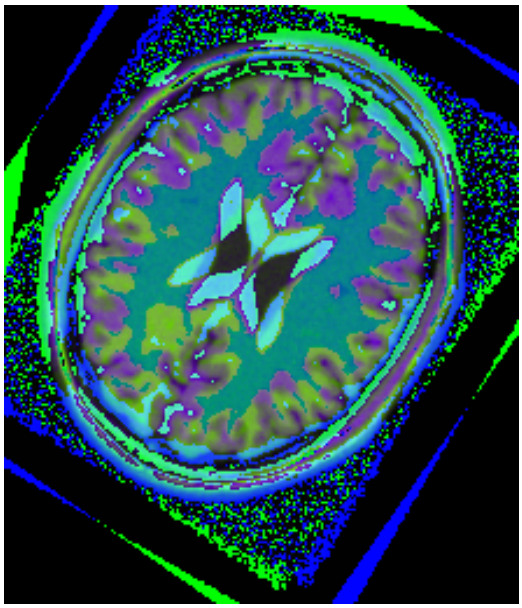
```python
# Automatic extraction of corner points
# Harris corners detection or Shi and Tomasi
# this method do not ensure the correct order in the points (A_points and
# B_points)
nb_points = 4; # number of points to detect
corners = cv2.goodFeaturesToTrack(A,nb_points ,0.01,10, blockSize =3, useHarrisDetector=False)
corners = np.int0(corners)
a, b, c = corners.shape;
A_points = np.reshape(corners, (a,c));

corners = cv2.goodFeaturesToTrack(B,nb_points ,0.01,10, blockSize =3, useHarrisDetector=False)
corners = np.int0(corners)
a, b, c = corners.shape;
B_points = np.reshape(corners, (a,c));
```

The Fig.30.6 displays the results with the automatic detection of corners (in this case, the method from Shi and Tomasi). Notice that by chance, the points in both images match. In other cases, one would have to remove outliers.

Figure 30.6: Shi and Tomasi corners detection. By chance, the points correspond and the ICP method gives a correct result.

(a)



(b)



★★★ 31 Phase Correlation

After a Fourier transform, a translation becomes a phase shift, which is very easy to compute. This tutorial exploits this property to estimate the transformation between two images: first a translation, then a rotation and a scaling, by bringing them back to a translation through polar and log-polar coordinates.

31.1. Principle

Let $F(u, v)$ (resp. $G(u, v)$) be the Fourier transform of the image $f(x, y)$ (resp. $g(x, y)$). The coordinates (x, y) are pixel indices, the coordinates (u, v) are frequencies. The **phase correlation** $C(u, v)$ is defined by:

$$C = \frac{F^* \cdot G}{\|F^* \cdot G\|} \tag{31.1}$$

where F^* is the complex conjugate of F. Only the phase is kept: the modulus, that is to say the contrast of the images, is removed by the normalization.

31.1.1. Translation

If the image is translated by $(-\delta x, -\delta y)$, its Fourier transform becomes:

$$g(x, y) = f(x - \delta x, y - \delta y) \Leftrightarrow G = F(u, v)e^{2i\pi(u\delta x + v\delta y)} \tag{31.2}$$

In that case $C = e^{2i\pi(u\delta x + v\delta y)}$, and the *maximum* of the inverse Fourier transform of C, denoted R, gives the parameters of the translation:

$$(\delta x, \delta y) = \arg\max_{(x,y)}\{R\} \tag{31.3}$$

31.1.2. Rotation and scaling

A rotation of angle α around the origin becomes a translation in *polar* coordinates:

$$g(\rho, \theta) = f(\rho, \theta - \alpha) \Leftrightarrow G(w, \phi) = F(w, \phi - \alpha)$$

An isotropic scaling becomes a translation in *logarithmic* coordinates:

$$g(x, y) = f(ax, by) \Leftrightarrow G(\ln(u), \ln(v)) = F(\ln(u) - \ln(a), \ln(v) - \ln(b))$$

The two can therefore be estimated together, in **log-polar** coordinates, where a rotation of angle θ combined with an isotropic scaling of factor s is nothing but a translation:

$$g(\rho, \alpha) = f(s\rho, \alpha - \theta) \Leftrightarrow G(\ln(w), \phi) = F(\ln(w) - \ln(s), \phi - \theta) \tag{31.4}$$

31.2. Elementary transformations

The test images are built by transforming a reference image.

The Fourier transform considers the data as periodic: the translation must be periodic as well, so that the parts of the image that leave on one side come back on the other. This is exactly what numpy.roll does, axis by axis. The following function is given; it is written so that the translation is directly recovered by the phase correlation afterwards.

```python
def translation(I, dx, dy):
    """
    Periodic translation of an image.

    I: image
    dx, dy: translation, in pixels
    returns: translated image
    """
    It = np.roll(I, I.shape[0]-dx, axis=0)
    It = np.roll(It, I.shape[1]-dy, axis=1)
    return It
```

Apply to a reference image a translation, a rotation and a scaling, and display the three results. What is lost, in each case?

For the two other transformations, use skimage.transform.rotate and skimage.transform.rescale.

31.3. Phase shift in the Fourier domain

Before estimating a translation, let us verify the property 31.2 by applying it backwards: a phase shift is introduced in the Fourier domain, and the image is rebuilt from the modified transform. If the parameters of the translation are denoted (dx, dy):

$$G(u, v) = F(u, v) \cdot e^{-2i\pi(u \cdot dx + v \cdot dy)}$$

where u and v are *normalized* coordinates, between -0.5 and 0.5.

1. Compute the Fourier transform of the image, and centre the low frequencies (fftshift).
2. Build the two arrays of normalized coordinates (u, v), then multiply the transform by the phase shift.
3. Come back to the spatial domain and display the image: the translation must be visible.

Informations

numpy.linspace and numpy.meshgrid build the coordinates; do not forget numpy.fft.ifftshift before the inverse transform, and keep only the real part of the result.

31.4. Estimating a translation

1. Program the phase correlation, by strictly following the formulas 31.1 and 31.3. The prototype of the function is:

```
def phase_correlation(I, Itranslate):
    """
    Phase correlation between two images.

    I : array_like
        reference image.
    Itranslate : array_like
        translated image.

    Returns
    -------
    delta : array_like
        translation vector (two elements).
    """
```

2. Apply a translation of about one hundred pixels to an image, and check that the function finds it back.

3. Display the map R. What does it look like, and why?

Informations

Useful functions: `np.fft.fft2`, `np.fft.ifft2`, `np.conjugate`, `np.abs`, and `skimage.feature.peak_local_max` with the argument `num_peaks=1` to locate the maximum, and the argument `exclude_border=False` to include maximal values on the sides of the correlation map.

31.5. Estimating a rotation

1. Apply a rotation of a known angle to the image, *without* resizing it.

2. Transform the two images into polar coordinates, then estimate the rotation with the *same* function as before.

3. Compare the two polar images: where is the rotation visible?

`skimage.transform.warp_polar` performs the change of coordinates. With its default parameters, the output has 360 lines: one per degree, so the angle can be read directly.

31.6. Estimating a rotation and a scaling

?

1. Apply to the image a rotation *and* an isotropic scaling.
2. Transform the two images into log-polar coordinates, and estimate the two parameters at once.
3. Convert the radial shift into a scaling factor, and compare with the applied value. Test several scaling factors, and find the limits of the method.

Informations

The option `scaling='log'` of `warp_polar` gives the log-polar coordinates. The *same* radius, in pixels, must be used for the two images. If N is the number of radial samples and r the radius, the shift k along this axis corresponds to a scaling factor $s = e^{-k/\kappa}$ with $\kappa = N/\ln(r)$.

The method presented here estimates a rotation and a scaling around the *centre* of the images, and a translation separately. To estimate the three at the same time, the log-polar transform is applied to the *modulus of the Fourier transform* of the images, which does not depend on the translation: this is the Fourier-Mellin transform.

31.7. Python correction

This correction only uses numpy for the Fourier transforms, and skimage for the images and the changes of coordinates.

```
import numpy as np
import matplotlib.pyplot as plt
import skimage
from skimage.feature import peak_local_max
from skimage.transform import rotate, rescale, warp_polar
```

31.7.1. Elementary transformations

The Fourier transform considers the data as periodic. The translation must therefore be periodic as well, which numpy.roll does directly; the function is written so that the phase correlation gives the translation back with the right sign.

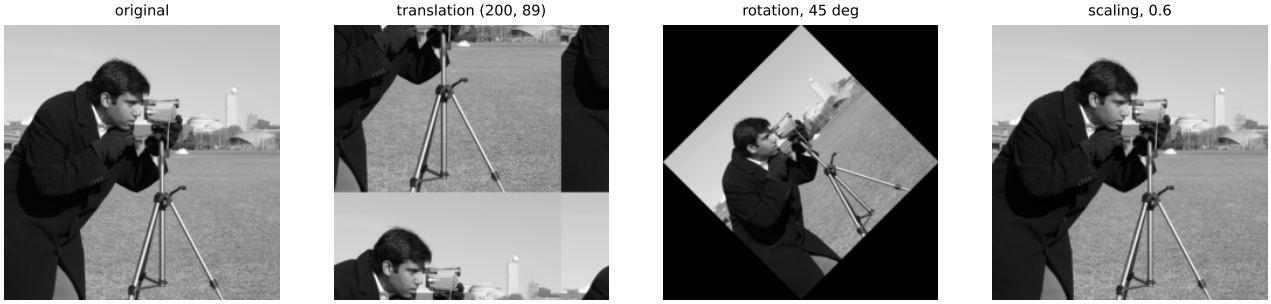
```
def translation(I, dx, dy):
    """
    Periodic translation of an image: the data are considered as periodic,
    exactly as the Fourier transform does.

    I: image
    dx, dy: translation, in pixels
    returns: translated image
    """
    It = np.roll(I, I.shape[0]-dx, axis=0)
    It = np.roll(It, I.shape[1]-dy, axis=1)
    return It


I = skimage.data.camera()
It = translation(I, 200, 89)
Ir = rotate(I, angle=45, resize=True, mode='constant', cval=0)
Is = rescale(I, scale=0.6, mode='constant', cval=0)
```

The periodic translation wraps the image around (Fig. 31.1): the parts that leave on one side come back on the other. The rotation and the scaling, on the contrary, lose information in the corners or change the size of the image.

Figure 31.1: The three elementary transformations applied to the test image.



31.7.2. Phase shift in the Fourier domain

The property is first verified backwards: a phase shift is applied to the transform, and the image is rebuilt. The coordinates (u, v) must be normalized between -0.5 and 0.5, and centred like the transform itself — hence the fftshift before, and the ifftshift after.



```
F = np.fft.fft2(I)
F = np.fft.fftshift(F)

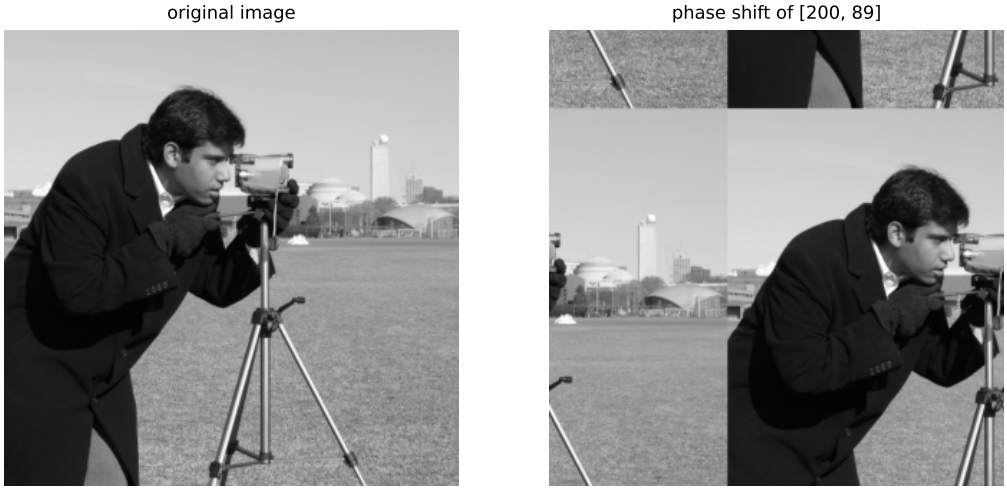
# normalized coordinates, between -0.5 and 0.5
x = np.linspace(-.5, .5, I.shape[0], endpoint=False)
u, v = np.meshgrid(x, x)

dx, dy = 100, 150
G = F * np.exp(-1j * 2 * np.pi * (dx * u + dy * v))
G = np.fft.ifftshift(G)

I2 = np.real(np.fft.ifft2(G))
```

The result is the image translated by $(100, 150)$, in a periodic way (Fig. 31.2). The inverse transform is complex because of the numerical errors: only its real part is kept.

Figure 31.2: A phase shift in the Fourier domain is a translation in the spatial domain.



31.7.3. Estimating a translation

The function follows the formulas strictly: product of one transform by the conjugate of the other, normalization by the modulus, inverse transform, and finally the position of the maximum.

```python
def phase_correlation(I, Itranslate):
    """
    Phase correlation between two images.

    I : array_like
        reference image.
    Itranslate : array_like
        translated image.

    Returns
    -------
    delta : array_like
        translation vector (two elements).
    """
    GI = np.fft.fft2(I)
    GT = np.fft.fft2(Itranslate)

    P = GI*np.conjugate(GT)

    C = P / np.abs(P)

    R = np.real(np.fft.ifft2(C))

    delta = peak_local_max(R, num_peaks=1, exclude_border=False)

    return delta
```

```python
It = translation(I, 200, 89)
delta = phase_correlation(I, It)
print(delta)
```

```
[[200  89]]
```

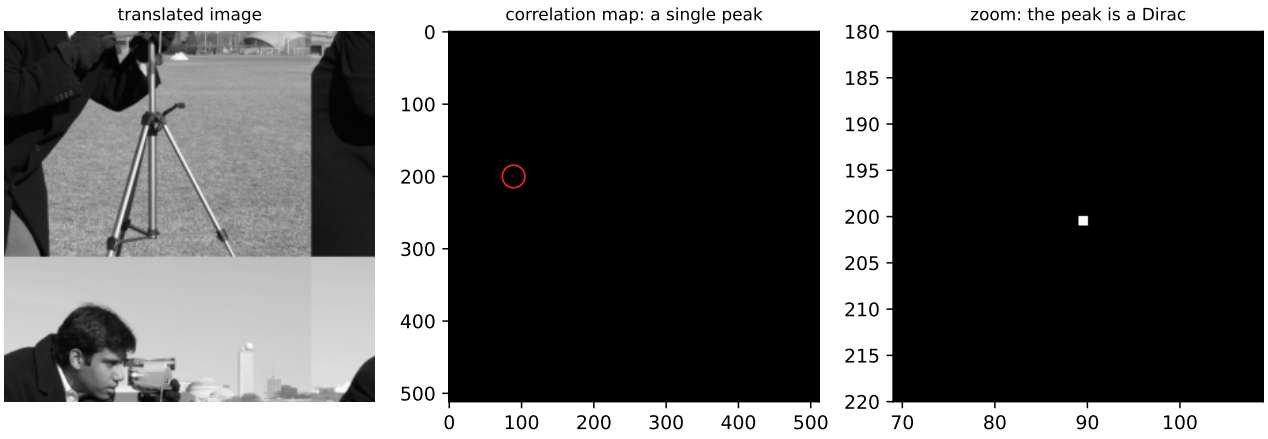
The translation is found exactly, and the map R explains why (Fig. 31.3): since only the phase is kept, C is a pure complex exponential, and its inverse transform is a **Dirac** located at $(\delta x, \delta y)$. The peak occupies a single pixel, which makes the detection extremely robust — much more than a classical correlation, whose maximum is spread out.

The precision is here of one pixel, since the maximum is searched on the sampling grid. Sub-pixel versions exist: they fit a model on the neighbourhood of the peak, or use an upsampled transform.

31.7.4. Estimating a rotation

In polar coordinates, a rotation of the image becomes a translation along the axis of the angles: the *same* function can therefore be used, without any modification.

Figure 31.3: Translated image, correlation map, and zoom on the peak: the maximum is a single pixel.




```
theta = 33
rotated = rotate(I, theta, resize=False)

img_polar = warp_polar(I)
template_polar = warp_polar(rotated)

delta = phase_correlation(img_polar, template_polar)
print("angle applied: {0} deg, found: {1} deg".format(theta, delta[0, 0]))
```

```
angle applied: 33 deg, found: 33 deg
```

With its default parameters, warp_polar returns 360 lines, one per degree: the first coordinate of the result is directly the angle in degrees (Fig. 31.4).

31.7.5. Estimating a rotation and a scaling

In log-polar coordinates, the scaling becomes a translation along the radial axis. The two images must be sampled with the **same radius in pixels**, otherwise the scaling is applied twice and the shift disappears.

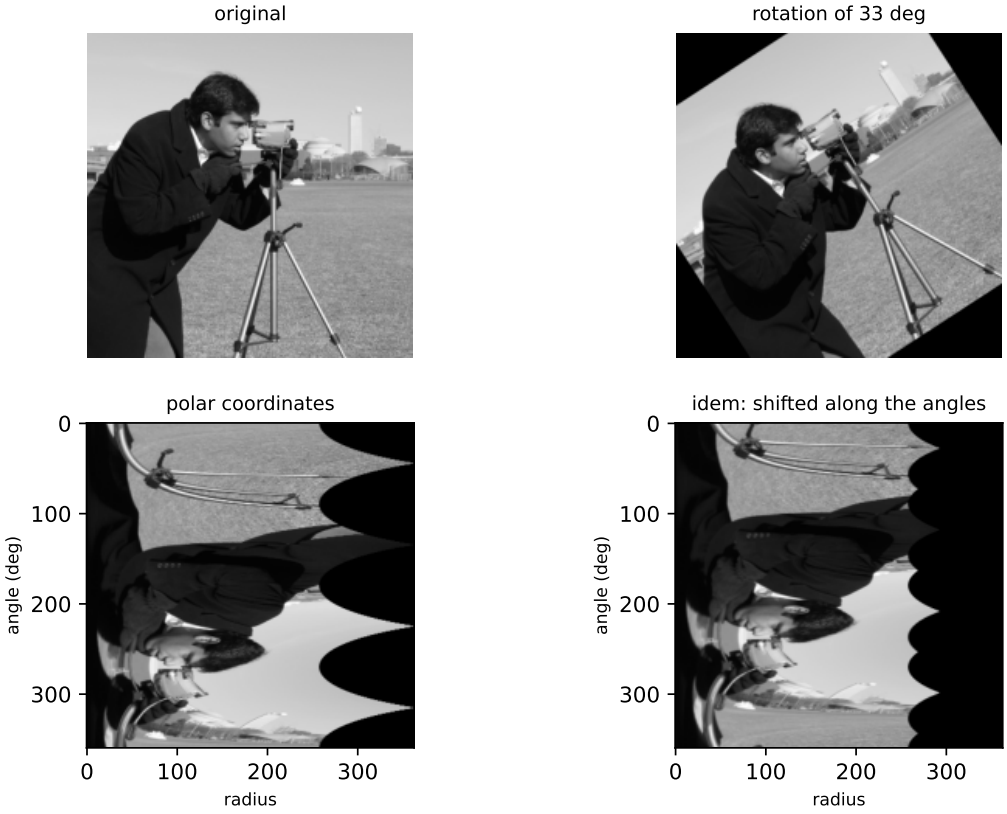
```
scale = 1.4
transformed = rescale(rotate(I, theta, resize=False), scale)

# the two transforms must use the same radius, in pixels
radius = I.shape[0]//2
shape = (360, 256)
img_logpolar = warp_polar(I, scaling='log', radius=radius,
                          output_shape=shape)
template_logpolar = warp_polar(transformed, scaling='log', radius=radius,
                               output_shape=shape)

delta = phase_correlation(img_logpolar, template_logpolar)
```

The radial shift is read modulo the width of the log-polar image: a shift larger than half of the image is in fact a negative shift. The scaling factor is then obtained by the inverse of the logarithmic sampling.

Figure 31.4: The two images and their polar representation. The rotation has become a vertical translation, modulo 360°.




```
shift = delta[0, 1]
if shift > shape[1]//2:
    shift -= shape[1]
klog = shape[1] / np.log(radius)

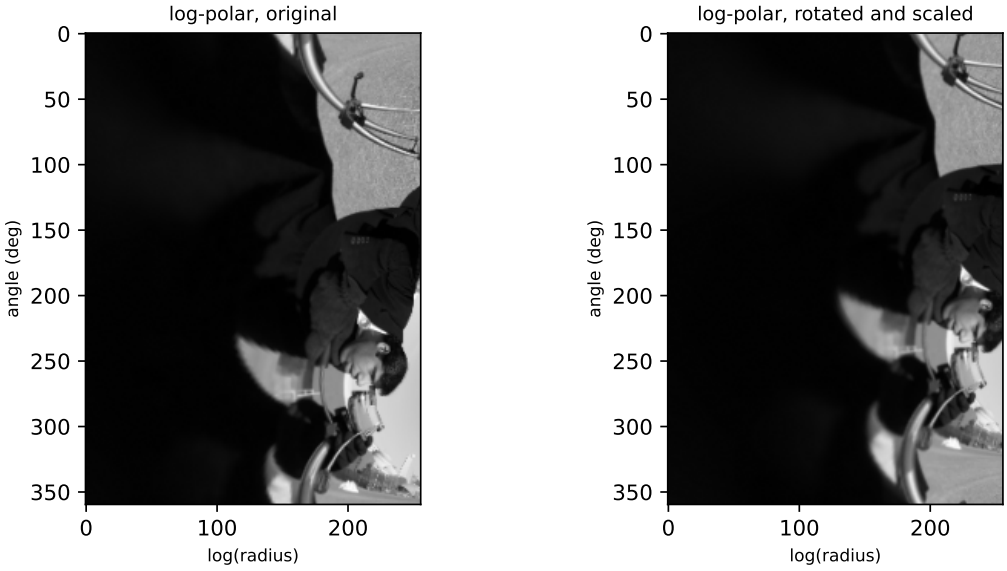
print("angle applied: {0} deg, found: {1} deg".format(theta, delta[0, 0]))
print("scale applied: {0}, found: {1:.3f}".format(
    scale, np.exp(-shift/klog)))
```

```
angle applied: 33 deg, found: 33 deg
scale applied: 1.4, found: 1.414
```

The method has two limits, which are easy to observe by varying the scaling factor:

- the radial axis is sampled logarithmically, so the precision on s degrades when s grows: with the values used here, one pixel of shift already represents $2\,\%$;
- for a large factor ($s \geq 2$), a good part of the content of the image leaves the disk of radius r, the two log-polar images no longer describe the same scene, and the estimation collapses.

Figure 31.5: Log-polar representation of the two images: the rotation is a vertical shift, the scaling a horizontal one.



32 Machine Learning

The objective of this tutorial is to classify images by using machine learning techniques. Some images, as illustrated in Figure 32.1 of the Kimia database will be used [1, 40].

Figure 32.1: The different processes will be applied on images from the Kimia database. Here are some examples.

(a) bird



(b) camel



(c) ray



(d) turtle



32.1. Feature extraction

The image database used in this tutorial is composed of 18 classes. Each class contains 12 images. All these 216 images come from the Kimia database. In order to classify these images, we first have to extract some features of each binary image.

?

1. For each image in the database, extract a number of different features, denoted *nbFeatures*.
2. Organize these features in an array of *nbFeatures* columns and 216 lines (the shape is thus $(216, nbFeatures)$). In this way, each line i represents the different features of the image i.

You can use the Python function skimage.measure.regionprops to extract the same geometrical features. In order to load all image, from the directory, you can use this for loop and use the glob module in order to load all files.

```
import glob
rep = 'images_Kimia216/'
classes = ['bird', 'bone', 'brick', 'camel', 'car', 'children',
           'classic', 'elephant', 'face', 'fork', 'fountain',
           'glass', 'hammer', 'heart', 'key', 'misk', 'ray', 'turtle']
nbClasses = len(classes)
nbImages = 12

# The features are manually computed
properties = np.zeros((nbClasses*nbImages, 9))
target = np.zeros(nbClasses * nbImages)
index = 0
for ind_c, c in enumerate(classes):
    filelist = glob.glob(rep+c+'*')
    for filename in filelist :
        print(filename)
```

32.2. Image classification

In order to classify the images, we are going to use neural networks. Pattern recognition networks are feedforward networks that can be trained to classify inputs according to target classes. The inputs are the features of each image. The target data are the labels, indicating the class of each image.

32.2.1. Construction of the array of properties

- Build the target data, representing the class of each image. The target data for pattern recognition networks should consist of one vector containing the index of the target class. In our example, the target data will be a vector of 216 elements.
- The database will be divided into a training set (75%) and a test set (25%).

Use from sklearn.model_selection import train_test_split to perform this partition into training/test sets.

32.2.2. Training and classification

- Run the training task and classify the test images.
- Show the classification confusion matrix as well as the overall performance.

Look at the Python module sklearn.neural_network.MLPClassifier to make the classification. You may also use SVM classifier. Notice that your data may be normalized (use from sklearn.preprocessing import StandardScaler) before performing the classification.

?

- Try to run the same process again (starting from the traint/test split). What can you conclude?
- Try to change the parameters of the network to improve the classification performance.



32.3. Python correction



The following imports may be considered.



```
# image manipulation and features construction
import glob
from skimage import measure, io
import numpy as np
import matplotlib.pyplot as plt
# preprocessing data and normalization
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing.data import QuantileTransformer
# learning methods
from sklearn import svm
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report ,confusion_matrix
# plot confusion matrix
import seaborn as sn
import pandas as pd
```

32.3.1. Feature extraction

We make a loop on the whole database to extract some features of each image. The 9 features used here are: area, convex area, eccentricity, equivalent diameter, extent, major axis length, minor axis length, perimeter and solidity.

Each image in this database is named with the following pattern: *name–number.bmp*. One method for reading all images is to define the list of names, and ready all the files by this name, associated to a class of object.



```
# Definitions of the database, classes and images
rep = 'images_Kimia216/';
classes = ['bird','bone','brick','camel','car','children',
    'classic','elephant','face','fork','fountain',
    'glass','hammer','heart','key','misk','ray','turtle'];
```

The following arrays are constructed in order to store the properties and the classes.



```
nbClasses = len(classes);
nbImages = 12;
# The features are manually computed
properties = np.zeros((nbClasses*nbImages,9));
target = np.zeros(nbClasses * nbImages);
```

Each class contains 12 images. The glob python module is used to easily read all images with the given pattern. The variable index handles the index of the class of the object.

```
index=0;
for ind_c, c in enumerate(classes):
    filelist = glob.glob(rep+c+'*');
    for filename in filelist :
        I = io.imread(filename);
        prop = measure.regionprops(I);
        properties[index, 0] = prop[0].area;
        properties[index, 1] = prop[0].convex_area;
        properties[index, 2] = prop[0]. eccentricity ;
        properties[index, 3] = prop[0].equivalent_diameter;
        properties[index, 4] = prop[0].extent;
        properties[index, 5] = prop[0].major_axis_length;
        properties[index, 6] = prop[0].minor_axis_length;
        properties[index, 7] = prop[0].perimeter;
        properties[index, 8] = prop[0]. solidity ;
        target[index] = ind_c;

        index = index+1;
```

Note that in the same time, the target array (required in the following) is built within this loop. It represents the true classes of the objects.

32.3.2. Classification

We used a training set of 75% of the database and 25% for the test set. The scaler is used in order to rescale the data having different ranges and dimensions. Other scalers are proposed in sklearn.preprocessing.

```
# percentage of the data used for splitting into train/test
percentTest = .25;
# MLP Classifier (Multi-Layer Perceptron)
# the data are first scaled
propertiesMLP = StandardScaler().fit_transform(properties);
prop_train, prop_test, target_train, target_test = train_test_split(propertiesMLP, target, test_size=percentTest
    ↪ , random_state=0);
```

The network is created with 1 hidden layers of 10 neurons. One could also use SVM instead of MLP (classifier = svm.SVC(kernel='linear');).

```
# feedforward neural network
# max_iter should be extended max_iter=100000 for adam or sgd solvers
mlp = MLPClassifier(hidden_layer_sizes=(10,), solver='lbfgs');
target_pred = mlp.fit(prop_train, target_train).predict(prop_test)
print("Training set score: %f" % mlp.score(prop_train, target_train))
```

The training score should be around 1, according to the training dataset and the different parameters employed.

```
Training set score: 1.000000
```

32.3.3. Performance

The confusion matrix gives an idea of the overall performance. The following function uses the modules seaborn and pandas to generate a heatmap that displays the confusion matrix.



```
def plot_cm(cm, classes, normalize=False, cmap=plt.cm.Blues):
    """
    Plot confusion matrix
    cm: confusion matrix, as ouput by sklearn.metrics.confusion_matrix
    classes: labels to be used
    normalize: display number (False by default) or fraction (True)
    cmap: colormap
    returns: figure that can be used for pdf export
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    fmt = '.1f' if normalize else 'd'
    df_cm = pd.DataFrame(cm, index = classes, columns = classes);

    fig = plt.figure();
    sn.set(font_scale=.8)
    sn.heatmap(df_cm, annot=True, cmap = cmap, fmt=fmt);
    plt.xlabel('Target label')
    plt.ylabel('True label')

    return fig;
```

The previous function is then used, and illustrated in Fig.32.2.



```
cnf_matrix = confusion_matrix(target_test, target_pred);
fig=plot_cm(cnf_matrix, classes, normalize=True);
```

sklearn.metrics.classification_report can be used to display the performance.

	precision	recall	f1-score	support
bird	0.75	1.00	0.86	3
bone	1.00	1.00	1.00	5
brick	1.00	1.00	1.00	1
camel	1.00	1.00	1.00	3
car	1.00	1.00	1.00	1
children	1.00	1.00	1.00	2
classic	1.00	1.00	1.00	5
elephant	1.00	1.00	1.00	3
face	1.00	1.00	1.00	4
fork	1.00	1.00	1.00	2
fountain	1.00	1.00	1.00	3
glass	1.00	1.00	1.00	5
hammer	1.00	0.75	0.86	4
heart	1.00	1.00	1.00	2
key	1.00	1.00	1.00	3
misk	1.00	1.00	1.00	3
ray	1.00	1.00	1.00	4
turtle	1.00	1.00	1.00	1
avg / total	0.99	0.98	0.98	54

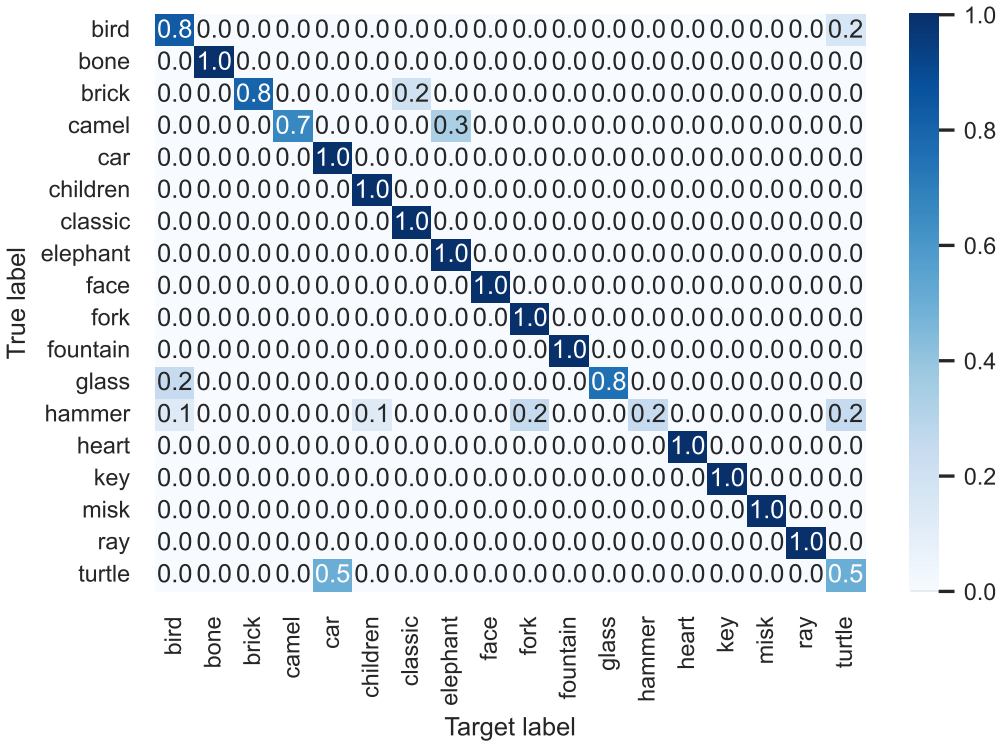


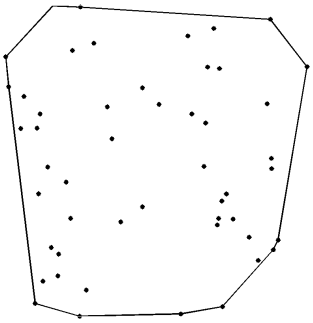
Figure 32.2: Normalized confusion matrix of the classification result.

Part IX Geometry, Shape Description and Measurement

33 Convex Hull

This tutorial aims to determine the convex hull of a set of 2D points with a simple and classical algorithm. This tool is largely used in computational geometry and image modeling. This tutorial is widely inspired of the Wikipedia page https://en.wikipedia.org/wiki/Graham_scan.

Figure 33.1: Convex Hull example.



33.1. Graham scan

The Graham scan is a method of computing the convex hull of a finite set of points in the plane with time complexity $O(n \log n)$, n is the number of points. It is an evolution of the Gift wrapping algorithm ($O(nh)$, h is the number of points in the hull) in the sense that it avoids evaluating all pairs of angles by first sorting the points.

33.1.1. Lowest y-coordinate point

The first step, as in the gift wrapping algorithm, is to find the point with the lowest y-coordinate. If two points exist in the set, choose the one with the lowest x-coordinate: it is denoted P. This step obviously takes $O(n)$.



Use the numpy.lexsort function.

33.1.2. Sort by angle

Next, the set of points must be sorted in increasing order of the angle they and the point P make with the x-axis.



Use the numpy.argsort function.

This is the limiting step, it takes $O(n \log n)$. Notice that the cosine of the angle is a decreasing function between 0 and 180 degrees, and will thus avoid to evaluate the angle itself. The sorted set of points is denoted $\mathcal{S}$ (it does not contain P).

33.1.3. Check angles: left or right turn?

From the star-like shape issued from the sorting algorithm, construct a list $\mathcal{L}$ of points as: $\mathcal{L} = \{P, \mathcal{S}, P\}$.

Then, for each triplet of consecutive points (P_i, P_{i+1}, P_{i+2}) of $\mathcal{L}$, check if the angle $P_i\widehat{P_{i+1}P_{i+2}}$ is a right turn or a left turn. In case of a right turn, remove P_{i+1} from the list $\mathcal{L}$. Process the entire list this way.

33.1.4. Left or right turn?

Again, determining whether three points constitute a "left turn" or a "right turn" does not require computing the actual angle between the two line segments, and can actually be achieved with simple arithmetic only. Consider the cross product of the vectors $\overrightarrow{P_iP_{i+1}}$ and $\overrightarrow{P_iP_{i+2}}$. $p_1.x$ and $p_1.y$ denote the two coordinates of point p_1.

Procedure ccw(p_1, p_2, p_3)
  return $(p_2.x - p_1.x) * (p_3.y - p_1.y) - (p_2.y - p_1.y) * (p_3.x - p_1.x)$;

Three points are a counter-clockwise turn if $ccw > 0$, clockwise if $ccw < 0$, and collinear if $ccw = 0$ because ccw is a determinant that gives the signed area of the triangle formed by p_1, p_2 and p_3.

33.1.5. Convex hull algorithm

This pseudo-code shows a different version of the algorithm, where points in the hull are pushed into a new list instead of removed from $\mathcal{L}$.

Data: n: number of points
Data: $\mathcal{L}$: sorted list of $n + 1$ elements
Data: First and last elements are the starting point P.
Data: All other points are sorted by polar angle with P.
`stack will denote a stack structure, with push and pop functions.`
Data: stack.push($\mathcal{L}(1)$)
Data: stack.push($\mathcal{L}(2)$)
for $i = 3$ *to* $n + 1$ **do**
  while *stack.size*≥ 2 *AND* ccw(*stack.secondlast, stack.last,* $\mathcal{L}(i) < 0$) **do**
    stack.pop();
  end
  stack.push($\mathcal{L}(i)$);
end

In this pseudo-code, stack.secondlast is the point just before the last one in the stack. When coding this algorithm, you might encounter problems with floating points operations (collinearity or equality check might be a problem).

?

1. Generate a set of random points.
2. Implement and apply the algorithm, and visualize the result.



33.2. Python correction





```
import numpy as np
import matplotlib.pyplot as plt
```

33.2.1. Graham scan algorithm

For convex hull and other computational geometry algorithms, robustness must be handled with special care. Floating points operations may be really tricky and the following code is not ensured to work for all cases.



```
# very naive precision handling
points = np.round(points, decimals=4);
```

The first step is to get the starting point.



```
# sort first by y, then x. get first point
ind=np.lexsort(points.transpose());
P = points[ind[0],:];

# all points but first one
points = points[ind]
pp =points[1:,:]
```

Then, the points are sorted by the cosinus of the angle. This step has complexity $O(n \log n)$.



```
# sort all points by angle
hypothenuse=np.sqrt( (pp[:,0]-P[0])**2 + (pp[:,1]-P[1])**2 );
adj_side = pp[:,0] - P[0];
# as cos is decreasing, we use minus
cosinus= -adj_side/hypothenuse;
ind=cosinus.argsort();

# construct ordered list of points
list_points = [];
list_points.append(P.tolist());
list_points = list_points + pp[ind,:].tolist();
list_points.append(P.tolist());
```

Finally, the sorted points allow to construct the convex hull by testing the orientation of the turn of the hull (see Fig.33.2).

```
first = list_points.pop(0);
second= list_points.pop(0);
hull = []; # convex hull
hull.append(first);
hull.append(second);

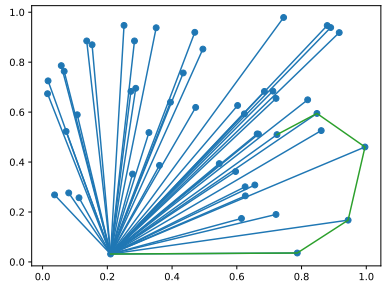
for i, p in enumerate(list_points):
    while len(hull)>=2 and crossProduct(hull, p)<0:
        hull.pop();

    hull.append(p);

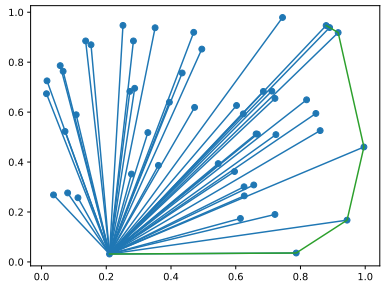
    # display result every 10 points
    if i%10 == 0:
        displayPointsAndHull(points, P, hull, 'chull_'+str(i)+'.python.pdf');
```

Figure 33.2: Graham scan illustration while constructing the convex hull.

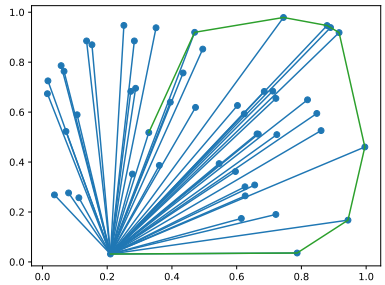
(a) After 10 tested points.



(b) After 20 tested points.



(c) After 30 tested points.



33.2.2. Useful functions

The cross-product function first extract the last two points of the hull, and check if there is a left-turn or a right-turn to go to the point p_3.

```
def crossProduct(hull, p3):
    """
    Cross product
    hull : list that should contain at least 2 points
    p3   : point
    """
    p1 = hull[-2];
    p2 = hull[-1];

    c= (p2[0] - p1[0])*(p3[1] - p1[1]) - (p3[0] - p1[0])*(p2[1] - p1[1]);
    return c;
```

In order to display the results, one function is proposed.



```
def displayPointsAndHull(points, P, hull, filename=None):
    """
    Fonction for display points and hull
    optionally save figure into pdf file
    """
    fig = plt.figure();
    if P is not(None):
        for i in np.arange(points.shape[0]):
            plt.plot([P[0], points[i,0]], [P[1], points[i,1]], 'C0');
    plt.scatter(points[:,0], points[:,1]);

    hull = np.array(hull);
    plt.plot(hull[:,0], hull[:,1], 'C2');
    plt.show()
    if filename:
        fig.savefig(filename, bbox_inches='tight');
```

33.2.3. Simple tests

For 5 points:



```
Points=np.array([[ 1,  2],
                 [ 1, -4],
                 [ 2, -1],
                 [ 3, -4],
                 [ 4,  1],
                 [ 3,  0]])

H = conv_hull(Points);
displayPointsAndHull(Points, H, 'sample_hull.python.pdf');
```

For a few random points:

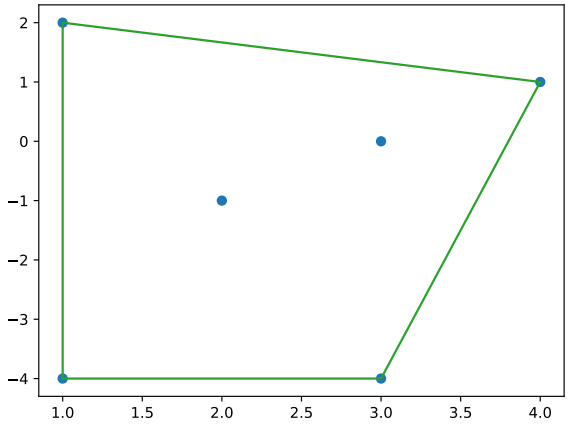


```
nb=50;
Points = np.random.rand(nb, 2);
H = conv_hull(Points);
displayPointsAndHull(Points, H, 'random_hull.python.pdf');
```

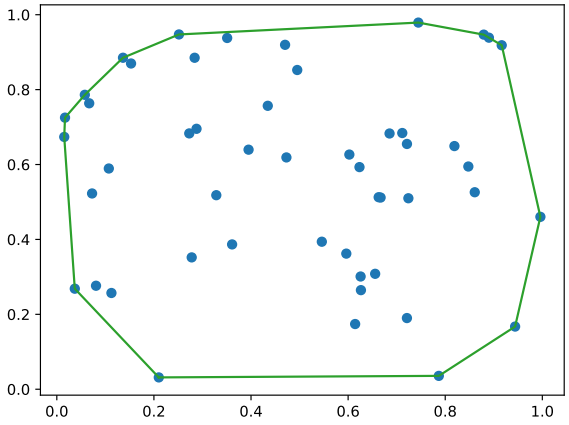
The results are illustrated in Fig.33.3.

Figure 33.3: Illustration of the convex hull computation.

(a) Convex points of 5 points.



(b) Convex hull of 50 random points.

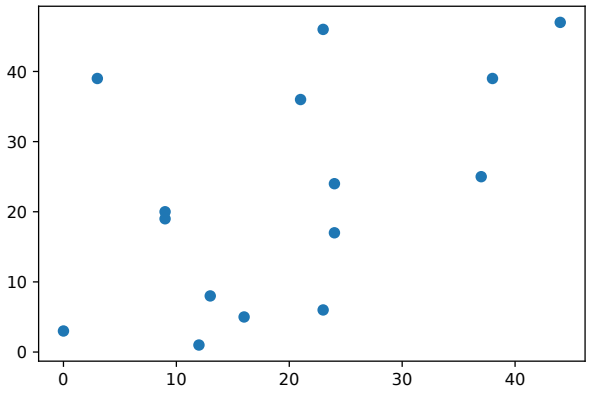


34 Voronoï Diagrams and Delaunay Triangulation

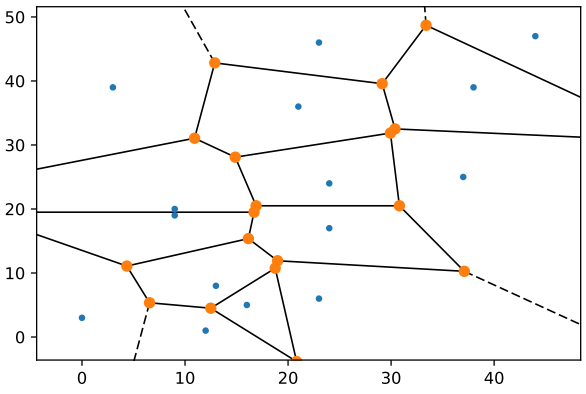
This tutorial aims to spatially characterize a spatial point pattern by using some tools of computational geometry: the Voronoi diagram, the Delaunay triangulation and the Minimum Spanning Tree (MST), illustrated in Fig.34.1.
For biomedical issues, this point pattern analysis can help the biologists to classify different populations of cells.

Figure 34.1: Random point pattern and some geometrical structures used to characterize it.

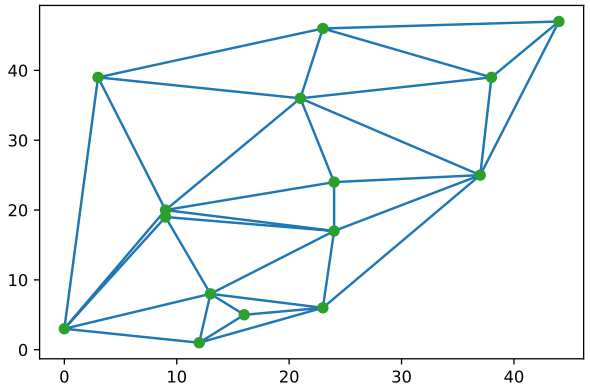
(a) Initial points



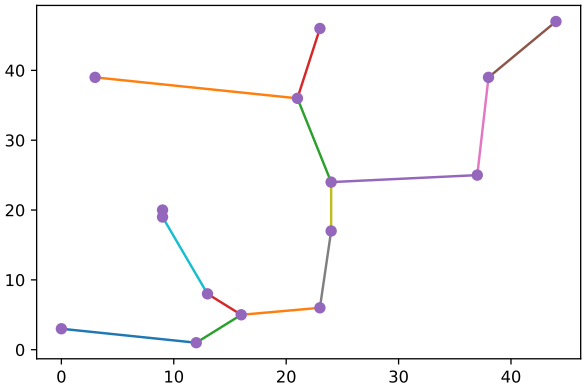
(b) Voronoi diagram



(c) Delaunay triangulation



(d) Minimum spanning tree of the Delaunay triangulation



34.1. Voronoi and Delaunay

A voronoi diagram, in 2D, is defined as a partition of the plane into cells R_k according to a distance function d and a set of seeds (germs) P_k.

$$R_k = \{x \in X \mid d(x, P_k) \leq d(x, P_j) \text{ for all } j \neq k\}$$

The Delaunay graph is the dual graph that links the germs of the neighborhing Voronoi cells.

34.1.1. Random tesselation

Follow these instructions to generate a random tesselation:

1. Generate a simple random point process, using a uniform distribution. The result is an array of size N by 2.
2. Compute the Delaunay triangulation.

Use the python functions Delaunay and Voronoi from scipy.spatial.

Notice that these structures will be used to loop over vertices, edges or regions. They contain attributes that make this process easy.

34.1.2. Characterization of the Voronoi diagram

This basic approach characterizes the set of the cells. With the help of the Voronoi diagram, it is possible to make the two following measurements, Area Disorder (AD) and Round Factor Homogeneity (RFH), defined by:

$$AD = 1 - \frac{1}{1 + \frac{\sigma(A)}{\mu(A)}} \tag{34.1}$$

$$RFH = 1 - \frac{\sigma(RF)}{\mu(RF)} \tag{34.2}$$

where A and RF are calculated on the regions R_k of the Voronoi diagram. $\mu(A)$ and $\sigma(A)$ are the mean and standard deviation of the areas of the Voronoi cells.

The circularity (RF) of a polygon can be defined as the ratio between its area and the area of the disk of an equivalent perimeter. $\mu(RF)$ and $\sigma(RF)$ are thus the mean and standard deviation of RF.

- Code these measurements with the following prototypes:

```
function ad = AD(V, R)
% computes AD (area disorder) parameters
% V: Vertices of the Voronoi diagram
% R: Regions of the Voronoi diagram
```

```
def AD(vor):
# takes a voronoi diagram to compute area disorder
```

  In order to evaluate the area of each Voronoi cell, transform each cell to a polygon.

- RFH can be computed with (almost) the same algorithm.
- Represent the couple (ad, rfh) in a graph, which gives a characterization of the Voronoi diagram.

See shapely.geometry.Polygon for evaluating the area of a polygon.

34.1.3. Characterization of the Delaunay graph

If L denotes the set of the edge lengths of the Delaunay triangulation, the mean and the standard deviation of L can also give informations on the graph.



- Compute and display in a graph the point of coordinates $(\mu(L), \sigma(L))$, with μ representing the mean and σ the standard deviation.

It is advised to use import the following module:



```
import networkx as nx
```

Then, create a function that transforms the simplices of the Delaunay triangulation into a networkx graph, with the following prototype:



```
def triToNx(tri):
    """
    Convert a triangulation into a NX graph.
    tri : Delaunay triangulation from scipy.spatial

    Returns
    G : networkx graph
    """
```

This function will loop over all simplices of the triangulation, and add an edge in the graph with the computed distance between vertices.

34.2. Minimum spanning tree

Definition from Wikipedia: a minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

One of the methods to compute the MST is the Kruskal algorithm.



The networkx module has a function in order to compute the minimum spanning tree. You will then be able to get all edges weights of this MST.



- Compute the MST.
- Compute $(\mu(L^*), \sigma(L^*))$ where L^* denotes the set of the edge lengths of the MST.

34.3. Characterization of various point patterns

?

1. Generate n conditional Poisson point processes of 100 points each. For each realization, calculate the parameters (AD, RFH), $(\mu(L), \sigma(L))$ and $(\mu(L^*), \sigma(L^*))$. Display these n points in a 2D diagram in order to analyze the robustness of the quantification.
2. Generate 3 different point processes with regular, uniform and Gaussian dispersion. Display the different diagrams. Which one is the most discriminant?



34.4. Python correction





```
from scipy.spatial import Voronoi, voronoi_plot_2d, Delaunay, distance
import numpy as np

import matplotlib.pyplot as plt
from shapely import geometry # for polygons, area and perimeter
import networkx as nx # graphs and minimum spanning tree
```

34.4.1. Random tessellations

Random tessellations are generated following normal standard and uniform distribution. A regular pattern is also employed. They are illustrated in Fig.34.2.



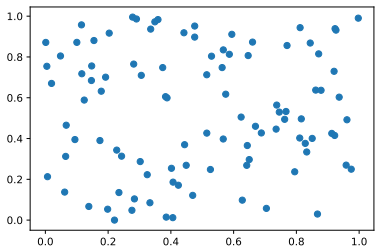
```
def dist_poisson(N=100):
    points = np.random.rand(N, 2)
    return points

def dist_gaussienne(N=100):
    points = np.random.randn(N, 2)
    return points

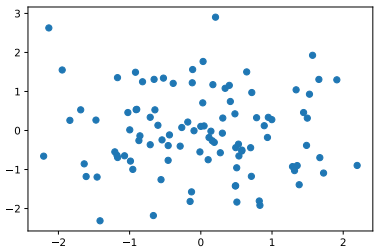
def dist_regular(N=100):
    c = np.floor(np.sqrt(N));
    x2, y2 = np.meshgrid(range(int(c)), range(int(c)));
    points = np.vstack([x2.ravel(), y2.ravel()])
    return points.transpose();
```

Figure 34.2: Different point patterns.

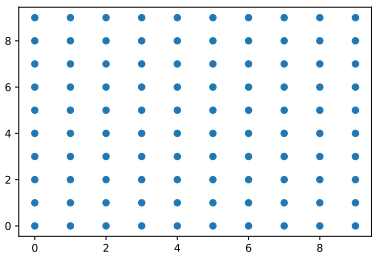
(a) Uniform distribution.



(b) Gaussian distribution.



(c) Regular distribution.



34.4.2. Voronoi diagram and analysis

The Voronoi diagram is simply generated via the following command, with points being generated with the previous functions:



```
vor = Voronoi(points);
```

The two characterization functions RFH and AD are defined on the Voronoi cells.



```
def RFH(vor):
    """
    Evaluates Round Factor Homogeneity from voronoi diagram
    """
    rfs =[];
    for cell in vor.regions:
        if cell and -1 not in cell:
            poly = geometry.Polygon([(vor.vertices[p-1]) for p in cell]);
            rfs.append(4*np.pi*poly.area/(poly.length**2));
    res = 1 - np.std(rfs) / np.mean(rfs);
    return res;
```



```
def AD(vor):
    """
    Evaluates Area Disorder from voronoi diagram
    """
    areas =[];
    for cell in vor.regions:
        if cell and -1 not in cell:
            poly = geometry.Polygon([(vor.vertices[p-1]) for p in cell]);
            areas.append(poly.area);
    res = 1 - 1/(1+np.std(areas) / np.mean(areas));
    return res;
```

34.4.3. Delaunay triangulation and minimum spanning tree

The Delaunay triangulation is computed with



```
tri = delaunay(points);
```

Then, the conversion to networkx graphs is done by the following function. Notice that the pdist function is used to compute the distance between all pairs of vertices, which is definitely not efficient, but probably simplifies the notations.



```
def triToNx(tri):

    G = nx.Graph()
    d = distance.pdist(tri.points)
    distances = distance.squareform(d)
    for s in tri.simplices:
        G.add_edge(s[0], s[1], weight=distances[s[0], s[1]])
        G.add_edge(s[1], s[2], weight=distances[s[1], s[2]])
        G.add_edge(s[0], s[2], weight=distances[s[0], s[2]])
    return G
```

Then, the characterization of the triangulation is done by measuring the distances of the edges.

```python
def characterization(tri):
    """
    Characterization of the Delaunay triangulation (mean and std dev of edges)
    """
    G = triToNx(tri)
    L = [w['weight'] for _, _, w in G.edges(data=True)]
    return np.mean(L), np.std(L)
```

The characterization of the minimum spanning tree is done with the same kind of function as for the Delaunay graph, and the MST is computed with:

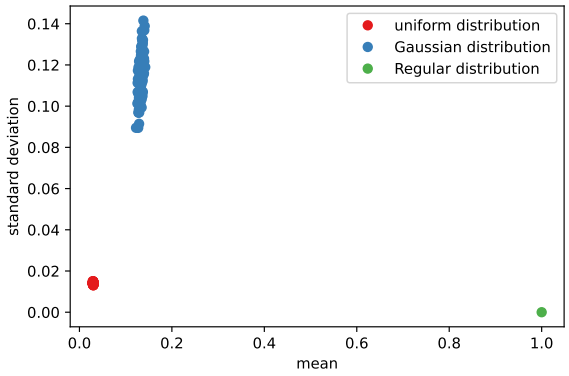
```python
mst = nx.minimum_spanning_tree(G)
```

34.4.4. Characterization of different realizations

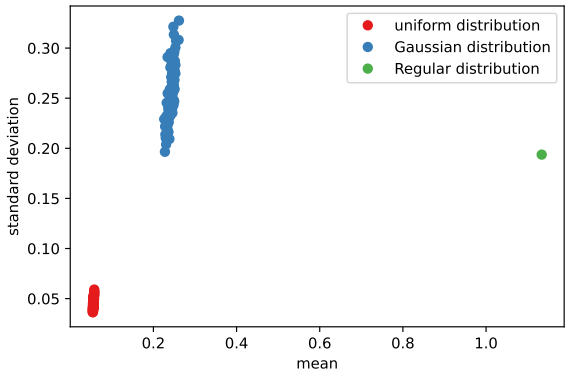
n realizations of the two distributions (uniform and Gaussian) are simulated. Then, the Voronoi diagram and the Delaunay triangulation are computed and characterized. The results are presented in Fig.34.3.

Figure 34.3: Characterization of several point processes. Each color represent a different process, and each point represent one realization. These characterizations can enhance a difference between the spatial distributions of the points.

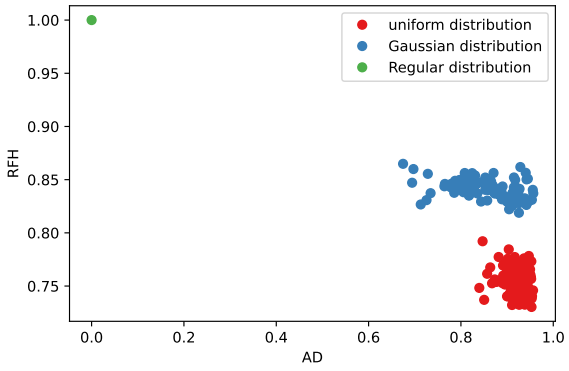
(a) Characterization by the mean and standard deviation of the lengths of the Delaunay triangulation.



(b) Characterization of the minimum spanning tree of the Delaunay triangulation by the mean and standard deviation of the lengths of the MST.



(c) AD and RFH on the Voronoi diagram.



⋆ 35 Alpha Shapes

The objective of this tutorial is to compute the alpha-shape of a set of points. Some tests will be done to reconstruct a shape from its random discretization as a point pattern (see Figure 1).

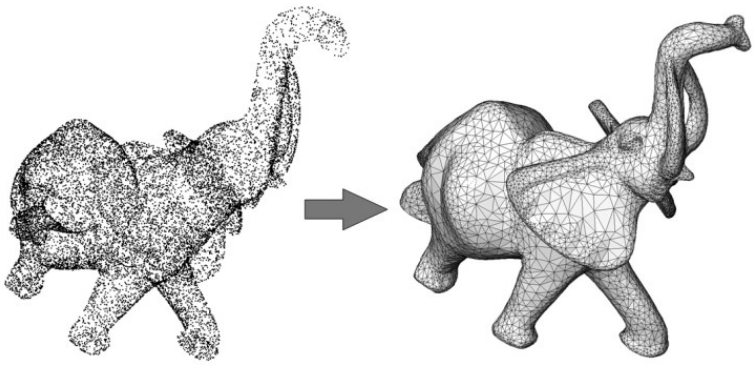


Figure 35.1: Shape reconstruction from a set of points.

35.1. Point pattern

From a binary image representing a shape, we need to extract a random set of points (included in the shape). It will define our input data.



(a) shape (b) point pattern representing the shape

Figure 35.2: Shape and random discretization as a point pattern.

?

1. Load the image 'camel.png'.
2. Discretize the set by using a random (uniform) point process to obtain the initial data. The density of the point process should be a user parameter.

A simple way to do this is to select all discrete points that are part of the image with `np.where` and randomly select a small quantity of these points.

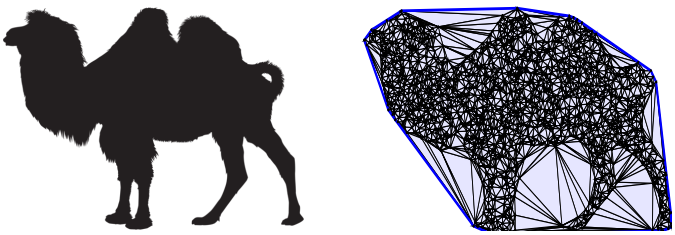
```
A = imread('camel.png')
pts = np.where(A)
pts = np.array(pts).transpose()

rng = np.random.default_rng()
pts = rng.permutation(pts)

# Pay attention to reference: points and image
# have not the same coordinates
pts = np.fliplr(pts)
points = pts[:num_points]
```

35.2. Delaunay triangulation

In order to build the alpha shape of a set of points, it is firstly required to compute its Delaunay triangulation.



(a) initial point pattern (b) Delaunay triangulation

Figure 35.3: Initial point pattern and its Delaunay triangulation.

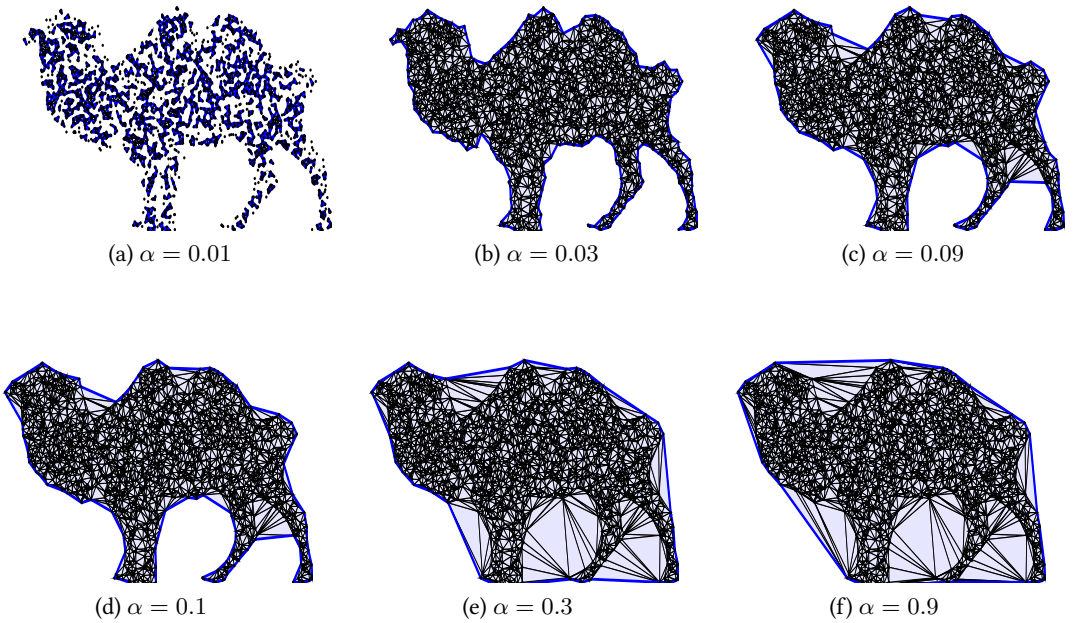
1. By using the initial point pattern, build its Delaunay triangulation.
2. Look at the resulting object to understand the structure of the triangulation.

You can import the module Delaunay with `from scipy.spatial import Delaunay`.
The display of the triangulation can be done by:

```
tri = Delaunay(points)
plt.triplot(points[:,0], points[:,1], tri.simplices, lw=.5)
```

35.3. Alpha-shape

The alpha-shape corresponds to the union of Delaunay triangles T_{ijk} such that the circumradius C_{ijk} is lower than $1/\alpha$.



(a) $\alpha = 0.01$ (b) $\alpha = 0.03$ (c) $\alpha = 0.09$

(d) $\alpha = 0.1$ (e) $\alpha = 0.3$ (f) $\alpha = 0.9$

Figure 35.4: Alpha-shapes for different values of α.



1. Implement the algorithm : loop over all simplices of the Delaunay triangulation, and check if the radius is lower than α. In this case, do not display this triangle.
2. Test this operator with the input data using different values of α (floating point parameter).

The radius of the circumcircle is not difficult to compute, however, you may use the module sympy in order to compute it easily.



```
from sympy import Point, Triangle
p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
t = Triangle(p1, p2, p3)
print(t.circumradius)
```



35.4. Python correction





```
from skimage.io import imread # read input image
import numpy as np
import matplotlib.pyplot as plt

from scipy.spatial import Delaunay # Delaunay triangulation
```

35.4.1. Point pattern

The image is first loaded.



```
A = imread('camel.png')
m,n = A.shape
```

All coordinates of pixels constituting the shape are extracted. The following code mainly consist of array manipulation.



```
pts = np.where(A)
pts = np.array(pts).transpose()

indices = np.arange(len(pts))
np.random.shuffle(indices)

# Pay attention to reference: points and image have not the same coordinates
pts = pts[indices]
pts = np.fliplr(pts)
pts[:,1] = m - pts[:,1]
```

Then, given a certain density, points are randomly chosen in the shape. They are displayed in Fig.35.5.

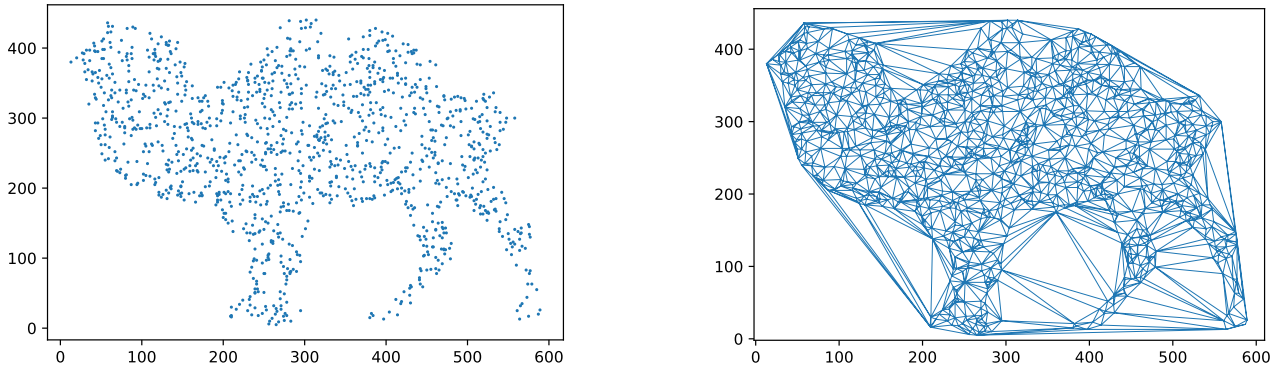


```
# Generate points
density =.01
nbPoints = int(len(pts)*density)

points = pts[:nbPoints]
plt.scatter(*zip(*points), s=1)
plt.savefig("points.pdf", bbox_inches='tight')
plt.show()
```

35.4.2. Delaunay triangulation

The Delaunay triangulation is simply obtained by the following code. The result is presented in Fig.35.5.



(a) Set of points, uniformely chosen in the shape.

(b) Delaunay triangulation.

Figure 35.5: Set of points and its Delaunay triangulation.

```
tri = Delaunay(points)

# Display result
plt.triplot(points[:,0], points[:,1], tri.simplices, lw=.5)
plt.show()
```

35.4.3. Alpha-solid

In order to build the alpha-solid, the circum-radii of all triangles should be computed. A rather simple way to do this is to use the class Triangle of sympy.geometry. The use of progressbar displays a progress bar, as the computation might take a long time. The sympy module is a symbolic computation module, and does not an optimal algorithm for this task.

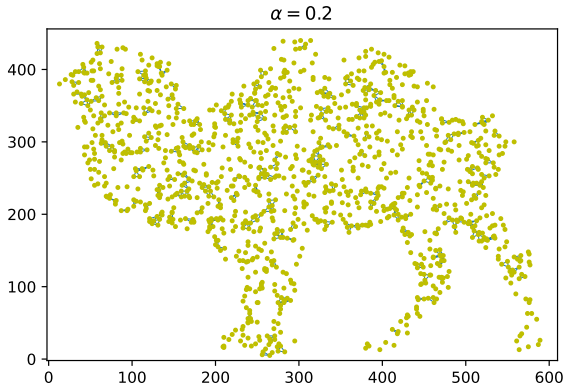
```
from sympy.geometry import Triangle
radius=[]
import progressbar
count=0

# takes a long time because of symbolic computation
with progressbar.ProgressBar(max_value=len(tri.simplices)) as bar:
    for t in tri.simplices:
        count+=1
        tt = Triangle(points[t[0], :], points[t[1], :], points[t[2], :] )
        radius.append(tt.circumradius)
        bar.update(count)
```

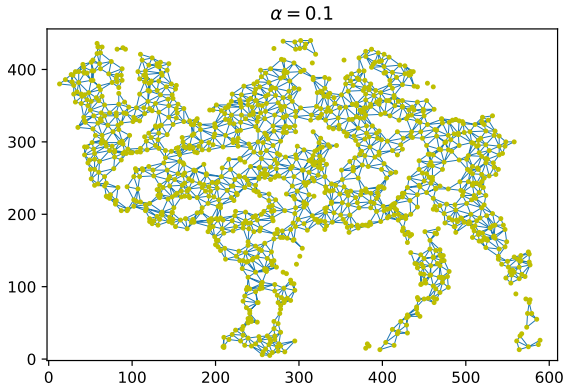
Then, given a radius, one can filter the triangles. The results are presented in Fig.35.6.

```
for R in progressbar.progressbar([5,10, 50, 100, 100000]):

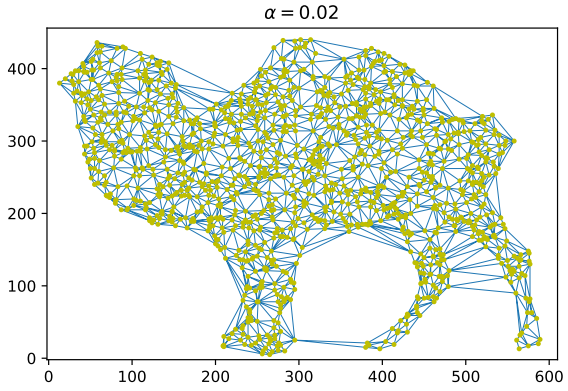
    r = np.array(radius)<R
    fig = plt.figure()
    plt.triplot(points[:,0], points[:,1], tri.simplices[r], lw=.5)
    plt.scatter(points[:,0], points[:,1], c='y', s=10)
    plt.show()
```



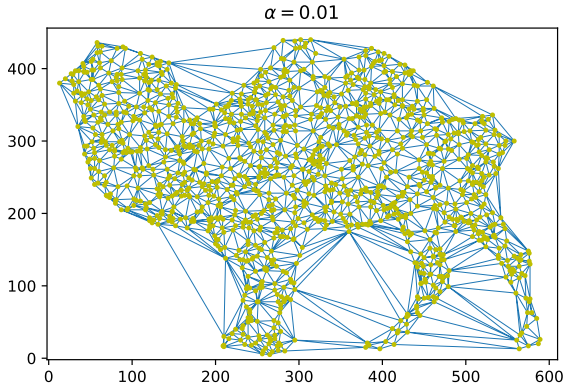
(a) $\alpha = 1/5$.



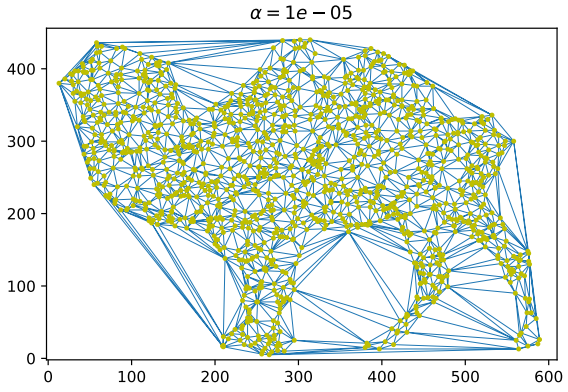
(b) $\alpha = 1/10$.



(c) $\alpha = 1/50$.



(d) $\alpha = 1/100$.



(e) $\alpha = 1/100000$.

Figure 35.6: Different alpha-solids.

35.4.4. To go further

There exist a python module dedicated to alpha-shapes. Here is a solution that uses it:



```
import alphashape
import matplotlib.pyplot as plt
from descartes import PolygonPatch
```



```
for R in progressbar.progressbar([5, 10, 50, 100, 100000]):
    # Generate the alpha shape
    alpha_shape = alphashape.alphashape(points, 1/R)

    # Initialize plot
    fig, ax = plt.subplots()

    # Plot input points
    ax.scatter(*zip(*points), s=1)

    # Plot alpha shape
    ax.add_patch(PolygonPatch(alpha_shape, alpha=.2))
    plt.title(fr"$\alpha={R}$")
    plt.show()
```

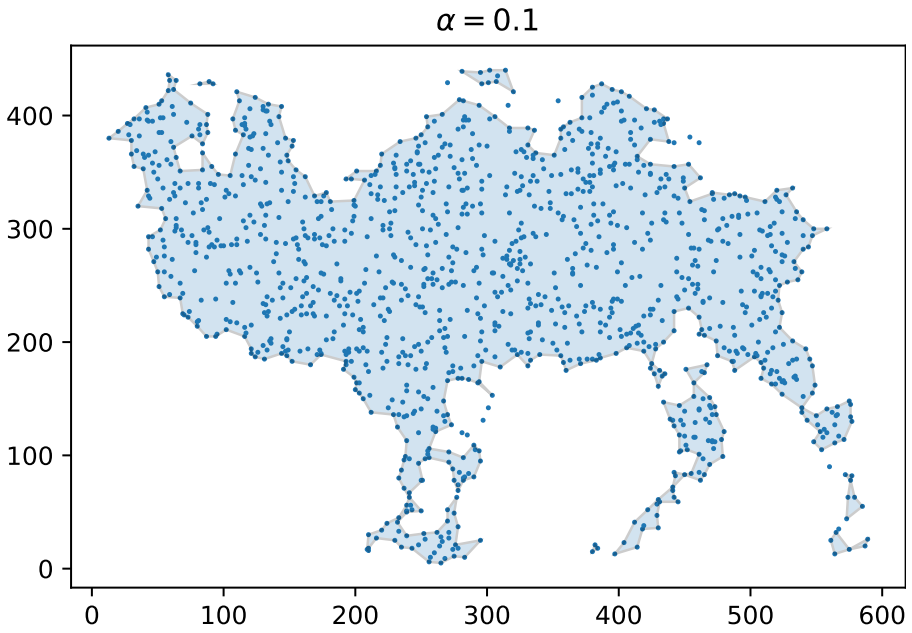


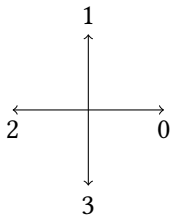
Figure 35.7: Illustration with the alphashape module.

36 Freeman Chain Code

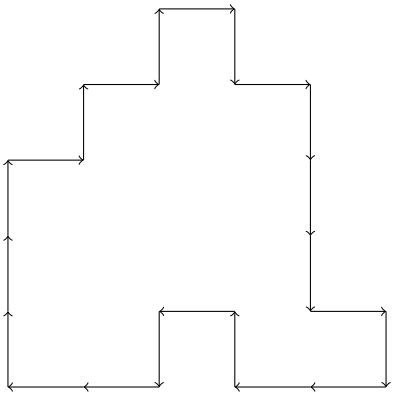
This tutorial is focused on shape representation by the Freeman chain code. This code is an ordered sequence of connected segments (of specific sizes and directions) representing the contour of the shape to be analyzed. The direction of each segment is encoded by a number depending on the selected connectivity (Fig. 36.1). In this example, the contour of the shape and its Freeman code are given in 4-connectivity.

Figure 36.1: Freeman chain code examples.

(a) 4-connected neighborhood.

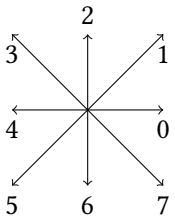


(b) Example Freeman code in a 4-neighborhood.



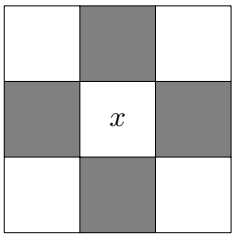
11101010303330322123 22

(c) 8-connected neighborhood.

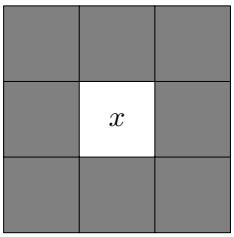


Notations:

- y is 4-adjacent to x if $|y_1 - x_1| + |y_2 - x_2| \leq 1$.
- y is 8-adjacent to x if $\max(|y_1 - x_1|, |y_2 - x_2|) \leq 1$.
- $N_4(x) = \{y : y \text{ is 4-adjacent to } x\}$; $N_4^*(x) = N_4(x) \backslash \{x\}$.
- $N_8(x) = \{y : y \text{ is 8-adjacent to } x\}$; $N_8^*(x) = N_8(x) \backslash \{x\}$.



(a) $V_4(x)$



(b) $V_8(x)$

36.1. Shape contours

Before determining the Freeman chain code of the shape, it is necessary to extract its contour.

?

1. Generate or load a simple shape as a binary image A.
2. Extract its contour $C_4(A)$ ou $C_8(A)$ according to the 4-connectivity or the 8-connectivity, respectively:

$$x \in C_4(A) \quad \Leftrightarrow \quad \exists y \in N_8(x) \quad y \in {}^cA$$
$$x \in C_8(A) \quad \Leftrightarrow \quad \exists y \in N_4(x) \quad y \in {}^cA$$

Informations

You can erode the object and subtract this erosion to it, with a structuring element that corresponds to N_4 or N_8 if you want to have 8 or 4 connectivity, respectively. In the module skimage.morphology, you will probably need to use the functions binary_erosion and disk.
The following function will compute the inner contour of a binary shape.

```
def bwperim(I, connectivity=8):
    """
    Morphological inner contour, in 4 or 8 connectivity
    I: binary image
    return: binary image representing the contour
    """
    if connectivity == 8:
        SE = disk(1)
    else:
        SE = rectangle(3, 3)

    E = binary_erosion(I, footprint=SE)

    return I ^ E
```

36.2. Freeman chain code

A simple shape can be generated by :

```
A = np.zeros((20, 20)).astype('bool')
A[4:14, 9:17] = True
A[1:18, 11:16] = True
```

From the contours, the Freeman chain code can be calculated.

Informations

You can use np.argwhere to locate the first point.



1. From the binary array of pixels, extract the first point belonging to the shape (from left to right, top to bottom).
2. From this initial point, determine the Freeman chain code c (counterclockwise direction) using the N_4 or N_8 connectivity.

36.3. Normalization

The Freeman chain code is depending on the initial point and is not invariant to shape rotation. It is then necessary to normalize this code.

1. The first step consists in defining a differential code d from the code c :

$$d_k = c_k - c_{k-1} (\text{mod } 4 \text{ or } 8)$$

   In this example, $d = 30031313313001330311 30$;

2. The second step consists in normalizing the code d. We have to extract the lowest number p from all the cyclic translations of d.
   In this example, $p = 00133031130300313133 13$.
   This Freeman chain code p is then independant from the initial point and invariant by rotations of angles $k * \pi/2$ radians ($k \in \mathbb{Z}$).



- Code above steps 1 and 2. Prototypes of functions are given.
- Validate your code by rotating the shape of $3\pi/2$ radians.



```
def codediff(fcc, connectivity=8):
# computes differential code
```



Use numpy.roll for circularly shifting the freeman code array. Code a small function that tests all elements of array, one by one, in order to get the minimum of two arrays.

36.4. Geometrical characterization

36.4.1. Perimeter

From the Freeman chain code, it is possible to make different geometrical measurement of the shape.



Calculate the perimeter of the shape (take care of the diagonals).

36.4.2. Area

The area is evaluated by the following algorithm:

- The area is initialized to 0 and the parameter B is initialized to 0.
- For each iteration on the Freeman chain code c, the area and the parameter value B are incremented with the following rules:

8-code	area	B
0	-B	0
1	-B-0.5	1
2	0	1
3	B+0.5	1
4	B	0
5	B-0.5	-1
6	0	-1
7	-B+0.5	-1

?

Calculate the area of the shape in 8-connectivity. Note that the area does not correspond to the total number of pixels belonging to the shape.



36.5. Python correction





```
import matplotlib.pyplot as plt
import numpy as np
from skimage.morphology import binary_erosion, disk, rectangle
from skimage.measure import perimeter
```

36.5.1. Shape contours

To generate a simple object, here is an example:



```
A = np.zeros ((20,20) ).astype('bool');
A[4:14, 9:17] = True;
A[1:18,11:16]= True;
```

The contours are computed in 4- or 8-connectivity, see Fig.36.2. This function uses the mathematical morphology in order to get the contour.



```
def bwperim(I, connectivity=8):
    """
    Morphological inner contour, in 4 or 8 connectivity
    I: binary image
    return: binary image representing the contour
    """
    if connectivity==8:
        SE = disk(1);
    else:
        SE = rectangle (3,3) ;

    E = binary_erosion(I, selem=SE);

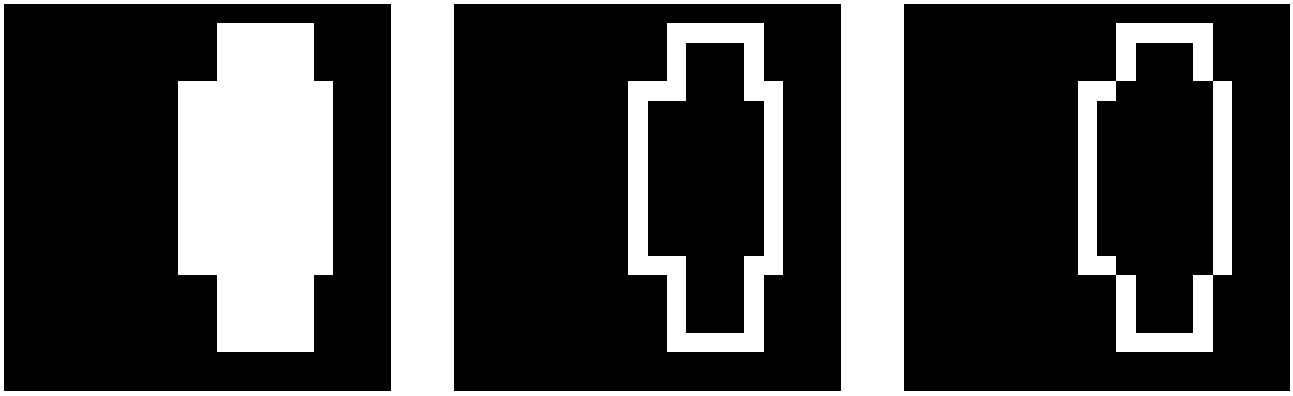
    return I^E;

# compute the contours
contours8 = bwperim(A, 4);
contours4 = bwperim(A, 8);
```

36.5.2. Freeman chain code

First point of the shape

The important thing is to find one point in the contour. The Freeman chain code is sensitive to this choice, but several methods can transform this code so that this first choice does not have any importance.



(c) Sample object. (d) Contour in 4-connectivity. (e) Contour in 8-connectivity.

Figure 36.2: Simple object and its contours in 4- or 8-connectivity.

```
def firstPoint(C):
    """
    find first point of contour
    returns point (as array)
    """
    p = np.argwhere(C);
    return p[0];
```

```
>>[r0,c0]= firstPoint(A)

r0 =
     5
c0 =
    10
```

Freeman chain code

The principle is to follow the contour, delete each pixel at each step, and find the direction of the next pixel.

```python
def freeman(C, p, connectivity=8):
    def getIndex(contour, point, connectivity):
        """ subfunction for getting the local direction
        """
        if connectivity==8:
            lut= np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]]);
        else:
            lut= np.array([[0, 2, 0], [8, 0, 4], [0, 6, 0]]);

        window = contour[point[0]-1:point[0]+2, point[1]-1:point[1]+2];
        window = window * lut;
        index  = np.max(window);
        return index-1;


    # Be careful that these LUTs consider coordinates from left to right, top to bottom
    lutx = np.array([-1, -1, -1, 0, 1, 1, 1, 0]);
    luty = np.array([-1, 0, 1, 1, 1, 0, -1, -1]);
    lutcode = np.array([3, 2, 1, 0, 7, 6, 5, 4]);

    nbrpoints = np.sum(C);
    code=[];
    point = p.copy();
    C2 = C.copy();

    for i in np.arange(nbrpoints):
        C2[point[0], point[1]] = 0;

        index = getIndex(C2, point, connectivity);

        if (index==0):
            C2[p[0], p[1]] = 1;
            index = getIndex(C2, point, connectivity);

        # new point
        point[0] = point[0] + lutx[index];
        point[1] = point[1] + luty[index];

        # add code
        code.append(lutcode[index]);

    return code;
```

```python
code = freeman(C8, p);
```

```
code = array([6, 6, 5, 4, 6, 6, 6, 6, 6, 6, 6, 6, 6, 0, 7, 6, 6, 6, 0, 0, 0, 0, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2,
    ↪ 2, 2, 2, 3, 2, 2, 4, 4, 4, 4])
```

36.5.3. Normalization

Differential code

This is the first step towards independence from the first point.

```
def codediff(fcc, connectivity=8):
    sr = np.roll(fcc, 1);
    d = fcc - sr;
    return d%connectivity;
```

Normalization

The differential code is then normalized, in order to get a rotation invariant code. All the circular shifts are evaluated, and a criterion (the minimum value) is established to be able to always find the same result, for every position of the first point.

```
def minmag(code):
    # high value for min computing
    codemin = np.max(code)* np.ones(code.shape);
    nb = len(code);
    for i in np.arange(len(code)):
        C = np.roll(code, i);

        for j in np.arange(nb):
            if C[j] > codemin[j]:
                break;
            elif C[j] < codemin[j]:
                codemin = C;
                break;
            if j == nb:
                codemin = C;

    return codemin;
```

The differential code is evaluated in d8, the normalization gives shapenumber8:

```
c = codediff(code, 8);
shapenumber8 = minmag(c);
```

```
c= array([2, 0, 7, 7, 2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 7, 7, 0, 0, 2, 0, 0, 0, 2, 0, 0, 7, 1, 0, 0, 0, 0, 0, 0, 0,
    ↪ 0, 1, 7, 0, 2, 0, 0, 0])

shapenumber8= array([0, 0, 0, 0, 0, 0, 0, 0, 1, 7, 0, 2, 0, 0, 0, 2, 0, 7, 7, 2, 0, 0, 0, 0, 0, 0, 0, 2, 7,
    ↪ 7, 0, 0, 2, 0, 0, 0, 2, 0, 0, 7, 1])
```

Validation

This validation shows the effect on a different starting point.

```
p = np.array([4, 9]);
```

Another test is to verify the result after a rotation. To prevent discretization problems, we use 90 degrees and take the transpose of the matrix.



```
% check for rotation by 90 deg
contours8rot=contours8';
```

The same code should be found in both cases.

36.5.4. Geometrical characterization

Perimeter for 8-connectivity

We first need to extract the codes in the diagonal directions and apply a $\sqrt{2}$ factor, then add the number of codes in vertical and horizontal directions.



```
def Perimeter(fcode):
    """
    fcode: Freeman code
    """
    nb_diag = np.sum(np.array(fcode)%2);
    perim = nb_diag*np.sqrt(2) + len(fcode)-nb_diag;
    return perim;
```

The perimeter is evaluated in the same way in skimage.



```
Perimeter:  43.65685424949238
skimage.measure.perimeter:  43.65685424949238
```

Area for 8-connectivity



```
def Area(fcode):
    """
    """
    area = 0;
    B = 0;
    lutB = np.array([0, 1, 1, 1, 0, -1, -1, -1]);
    for i in np.arange(len(fcode)):
        lutArea = np.array([-B, -(B+0.5), 0, (B+0.5), B, (B-0.5), 0, -(B-0.5)]);
        area = area + lutArea[fcode[i]];
        B = B + lutB[fcode[i]];
    return area;
```

Notice that the area evaluated by this way is different from the number of pixels.



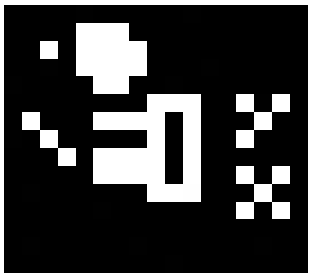
```
Area:  93.0
Number of pixels (area):  115
```

37 Topological Description

The objective of this tutorial is to classify all foreground points of a binary image according to their topological signification: interior, isolated, border.... The reader can refer to [6]for more details.

The different processes will be realized on the following binary image. By convention, white pixels have value 1 (or True), and black pixels have value 0 (or False).

Figure 37.1: Test

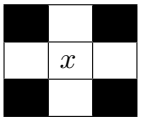


Preliminary definitions:

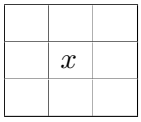
- y is 4-adjacent to x if $|y_1 - x_1| + |y_2 - x_2| \leq 1$.
- y is 8-adjacent to x if $\max(|y_1 - x_1|, |y_2 - x_2|) \leq 1$.
- $V_4(x) = \{y : y \text{ is 4-adjacent to } x\}$; $V_4^*(x) = V_4(x) \backslash \{x\}$.
- $V_8(x) = \{y : y \text{ is 8-adjacent to } x\}$; $V_8^*(x) = V_8(x) \backslash \{x\}$.

Figure 37.2: Different neighborhoods. By convention, pixels in white are of value 1, in black of value 0.

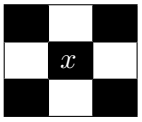
(a) $V_4(x)$



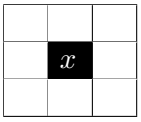
(b) $V_8(x)$



(c) $V_4^*(x)$



(d) $V_8^*(x)$



- a n-path is a point sequence $(x_0, \ldots, x_k)$ with x_j n-adjacent to x_{j-1} for $j = 1, \ldots, k$.
- two points $x, y \in X$ are n-connected in X if there exists a n-path $(x = x_0, \ldots, x_k = y)$ such that $x_j \in X$. It defines an equivalence relation.
- the equivalence classes of the previous binary relation are the n-components of X.

37.1. Connectivity numbers

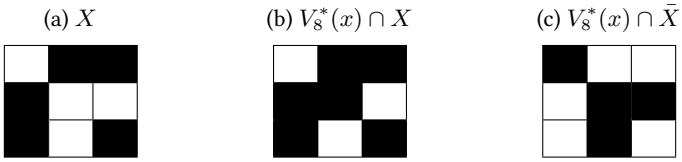
Let $Comp_n(X)$ be the number of n-components ($n = 4$ or $n = 8$ within the selected topology V_4 or V_8) of the set X of foreground points (object). We define the following set:

$$CAdj_n(x, X) = \{C \in Comp_n(X) : C \text{ is n-adjacent to } x\}$$

We select the 8-connectivity for the set X of foreground points (object) and the 4-connectivity for the complementary $\overline{X}$.

Warning: the definition of $CAdj_n$ introduces the n-adjacency to the central pixel x. In the case of the following configuration (Fig.37.3), $T_8 = 2$, $\bar{T}_8 = 2$ and $TT_8 = 3$.

Figure 37.3: The pixel in the bottom right corner is not C-adjacent-4 to x.



?

1. Create a function for determining the connectivity number: $T_8(x, X) = \#CAdj_8(x, V_8^*(x) \cap X)$,
2. Create a function for determining the connectivity number: $\overline{T}_8(x, X) = \#CAdj_4(x, V_8^*(x) \cap \overline{X})$,
3. Create a function for determining the number: $TT_8(x, X) = \#(V_8^*(x) \cap X)$.

Test these functions on some foreground points of the image 'test'.

Use scipy.ndimage and more specifically the function label.

37.2. Topological classification of binary points

From the connectivity numbers $T_8(x, X)$, $\overline{T}_8(x, X)$ and $TT_8(x, X)$, it is possible to classify a foreground point x within the binary image X according to its topological signification:

$T_8(x, X)$	$\overline{T}_8(x, X)$	$TT_8(x, X)$	Type
0	1		isolated point
1	0		interior point
1	1	> 1	border point
1	1	1	end point
2	2		2-junction point
3	3		3-junction point
4	4		4-junction point

The following Fig. 37.4 shows the classification of 4 points.

Figure 37.4: Points configurations.



?

With the help of this table, classify the points of the image 'test'.



37.3. Python correction





```
import numpy as np
from scipy.ndimage import label
import skimage.morphology as morpho
from skimage.io import imread
import matplotlib.pyplot as plt
```

37.3.1. Topological description

The operations are not difficult, except that the $CAdj_4$ should be coded carefully.



```
def nc(A):
    # A : block 3x3, binary
    # complementary set of A
    invA=1-A;
    # neighborhoods
    V8=np.ones((3,3)).astype(int);
    V8_star=np.copy(V8);
    V8_star[1,1] = 0;
    V4=np.array([[0, 1, 0],[1, 1, 1],[0, 1, 0]]).astype(int);
    # intersection is done by the min operation
    X1=np.minimum(V8_star,A);
    TT8=np.sum(X1);
    L, T8 = label(X1, structure=V8);
    # The C-ajd-4 might introduce some problems if a pixel
    # is not 4-connected to the central pixel
    X2=np.minimum(V8,invA);
    Y=np.minimum(X2,V4);
    X=morpho.reconstruction(Y, X2, footprint=V4);
    L, T8c = label(X, structure=V4);

    return T8, T8c, TT8
```

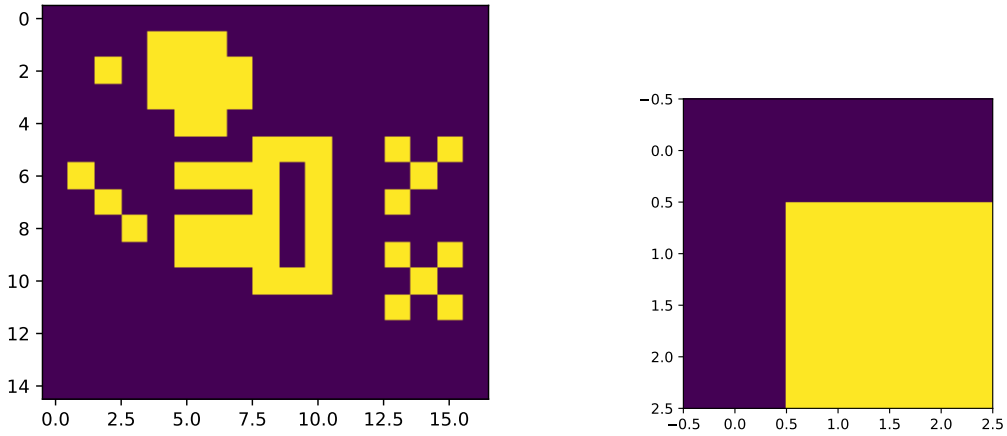


Figure 37.5: Extraction of the window centered in $x = (1, 4)$, given as (row,col).

37.3.2. Topological classification

The different types are given by the following code.



```
def nc_type(X):
    # evaluates the connectivity numbers
    a,b,c=nc(X);
    if (a==0):
        y=1; # isolated point
    if ((a==1) and (b==1) and (c>1)):
        y=5;  # border point
    if (b==0):
        y=7;  # interior point
    if ((a==1) and (b==1) and (c==1)):
        y=6;  # end point
    if (a==2):
        y=2;  # 2-junction point
    if (a==3):
        y=3;  # 3-junction point
    if (a==4):
        y=4;  # 4-junction point:
    return y;
```

In order to perform the classification of all pixels of an image, one has to loop over all the pixels, except the ones at the sides. The results are presented in Fig.37.6



```
def classification (A):
    # for the whole image
    m, n = A.shape
    B=np.zeros((m,n));
    for i in range(1, m-1):
        for j in range(1, n-1):
            if A[i,j]> 0:
                X=A[i-1:i+2,j-1:j+2];
                B[i,j]=nc_type(X);
    return B
```

Here is a code that can be used to get human-readable connectivity type.

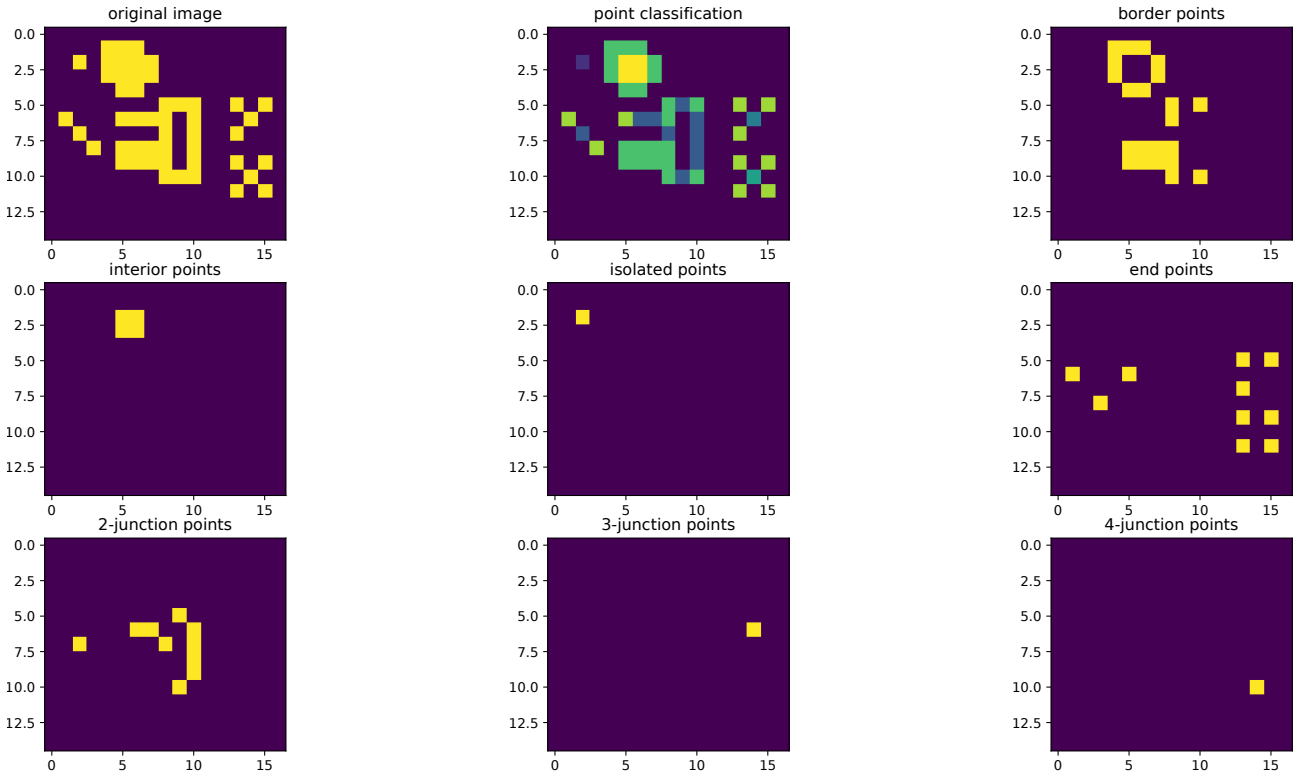


Figure 37.6: Classification of all the pixels of the the original image.

```python
def translate_type(type):
    """
    Translates the type of a pixel to a human-readable string.

    Args:
        type (int): The type of the pixel.

    Returns:
        str: The human-readable string.
    """
    if type == 1:
        return "isolated point"
    if type == 2:
        return "2-junction point"
    if type == 3:
        return "3-junction point"
    if type == 4:
        return "4-junction point"
    if type == 5:
        return "border point"
    if type == 6:
        return "end point"
    if type == 7:
        return "interior point"
    return "unknown"
```

⋆ 38 Integral Geometry

This tutorial aims to characterize objects by measurements from integral geometry.

The different processes will be applied on the following synthetic image:

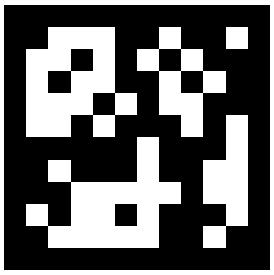


Figure 38.1: X

38.1. Cell configuration

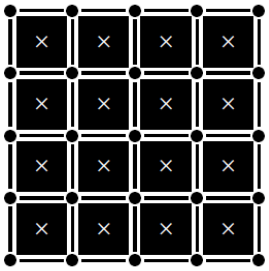
The spatial support of an image can be covered by cells associated with pixels. A cell (square of interpixel distance size) is composed of 1 face, 4 edges and 4 vertices. Either a cell is centered in a pixel (intrapixel cell) or a cell is constructed by connecting pixels (interpixel cell). The following figure shows the two possible representations of an image with 16 pixels:

Figure 38.2: Pixels representation.

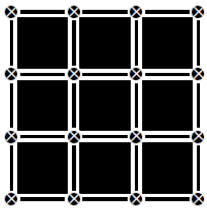
(a) Image.



(b) Intrapixel cells.



(c) Interpixel cells.



We respectively denote f^{intra} (resp. f^{inter}), e^{intra} (resp. e^{inter}) and v^{intra} (resp. v^{inter}) the number of faces, edges and vertices for the intrapixel (resp. interpixel) cell configuration Using these two configurations, the measurements from integral geometry (area A, perimeter P, Euler number χ_8 or χ_4) can be computed as:

$$A = f^{intra} = v^{inter} \tag{38.1}$$

$$P = -4f^{intra} + 2e^{intra} \tag{38.2}$$

$$\chi_8 = v^{intra} - e^{intra} + f^{intra} \tag{38.3}$$

$$\chi_4 = v^{inter} - e^{inter} + f^{inter} \tag{38.4}$$

?

1. Count manually the number of faces, edges and vertices of the image X for the two configurations (Intra- and Inter-pixel) of Fig. 38.1.
2. Deduce the measurements from integral geometry (Eq. 38.1-38.4).

38.2. Neighborhood configuration

In order to efficiently calculate the number of vertices, edges and faces of the object, the various neighborhood configurations (of size 2x2 pixels) of the original binary image X are firstly determined. Each pixel corresponds to a neighborhood configuration α. Thus, sixteen configurations are possible, presented in Tab. 38.1.

Table 38.1: Neighborhood configurations.

	$\begin{matrix}0&0\\0&0\end{matrix}$	$\begin{matrix}0&0\\0&1\end{matrix}$	$\begin{matrix}0&1\\0&0\end{matrix}$	$\begin{matrix}0&1\\0&1\end{matrix}$	$\begin{matrix}0&0\\1&0\end{matrix}$	$\begin{matrix}0&0\\1&1\end{matrix}$	$\begin{matrix}0&1\\1&0\end{matrix}$	$\begin{matrix}0&1\\1&1\end{matrix}$
α	0	1	2	3	4	5	6	7
	$\begin{matrix}1&0\\0&0\end{matrix}$	$\begin{matrix}1&0\\0&1\end{matrix}$	$\begin{matrix}1&1\\0&0\end{matrix}$	$\begin{matrix}1&1\\0&1\end{matrix}$	$\begin{matrix}1&0\\1&0\end{matrix}$	$\begin{matrix}1&0\\1&1\end{matrix}$	$\begin{matrix}1&1\\1&0\end{matrix}$	$\begin{matrix}1&1\\1&1\end{matrix}$
α	8	9	10	11	12	13	14	15

Thereafter, each configuration contributes to a known number of vertices, edges and faces (Tab. 38.2). To determine the neighborhood configurations of all the pixels, an efficient algorithm effective consists in convolving the image X by a mask F, whose values are powers of two, and whose origin is the top-left pixel:

$$F = \begin{pmatrix} 1 & 4 \\ 2 & 8 \end{pmatrix}$$

The resulting image is $X * F$. Notice that this image X has no pixel touching the borders, your code should ensure this (for example by padding the array with zeros). In this way, the histogram h of $X * F$ gives the distribution of the neighborhood configurations from image X. And each configuration contributes to a known number of vertices, edges and faces:

$$v = \sum_{\alpha=0}^{15} v_\alpha h(\alpha) \tag{38.5}$$

$$e = \sum_{\alpha=0}^{15} e_\alpha h(\alpha) \tag{38.6}$$

$$f = \sum_{\alpha=0}^{15} f_\alpha h(\alpha) \tag{38.7}$$

The following table gives the values of v_α, e_α and f_α for each cell configuration.

?

1. Perform the computation of the convolution of X by F.
2. Compute the distribution (histogram) of the neighborhood configurations from image X.
3. Deduce the number of vertices, edges and faces for each cell representation and compare these values with the previous (manually computed) results.

Table 38.2: Contributions of the neighborhood configurations to the computation of v, e and f

intrapixel cells																
α	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f_α	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
e_α	0	2	1	2	1	2	2	2	0	2	1	2	1	2	2	2
v_α	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
interpixel cells																
α	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f_α	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
e_α	0	0	0	1	0	1	0	2	0	0	0	1	0	1	0	2
v_α	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

38.3. Crofton perimeter

The Crofton perimeter could be computed form the number of intercepts in different random directions. In discrete case, only the directions $0, \pi/4, \pi/2$ and $3\pi/4$ are considered; they are selected according to the desired connexity.

The number of intercepts are denoted $i_0, i_{\pi/4}, i_{\pi/2}$ and $i_{3\pi/4}$ for the orientation angles $0, \pi/4, \pi/2$ and $3\pi/4$ respectively.

In this way, the Crofton perimeter (in 4 and 8 connexity) is defined in discrete case as:

$$P_4 = \frac{\pi}{2}\left(i_0 + i_{\pi/2}\right) \tag{38.8}$$

$$P_8 = \frac{\pi}{4}\left(i_0 + \frac{i_{\pi/4}}{\sqrt{2}} + i_{\pi/2} + \frac{i_{3\pi/4}}{\sqrt{2}}\right) \tag{38.9}$$

These perimeter measurements can be computed from the neighborhood configurations of the original image:

$$P_4 = \sum_{\alpha=0}^{15} P_\alpha^4 h(\alpha) \tag{38.10}$$

$$P_8 = \sum_{\alpha=0}^{15} P_\alpha^8 h(\alpha) \tag{38.11}$$

with the following weights P_α^4 and P_α^8 of these linear combinations (Tab. 38.3):

Table 38.3: Weights for the computation of the Crofton perimeter

α	0	1	2	3	4	5	6	7
P_α^4	0	$\frac{\pi}{2}$	0	0	0	$\frac{\pi}{2}$	0	0
P_α^8	0	$\frac{\pi}{4}\left(1+\frac{1}{\sqrt{2}}\right)$	$\frac{\pi}{4\sqrt{2}}$	$\frac{\pi}{2\sqrt{2}}$	0	$\frac{\pi}{4}\left(1+\frac{1}{\sqrt{2}}\right)$	0	$\frac{\pi}{4\sqrt{2}}$
α	8	9	10	11	12	13	14	15
P_α^4	$\frac{\pi}{2}$	π	0	0	$\frac{\pi}{2}$	π	0	0
P_α^8	$\frac{\pi}{4}$	$\frac{\pi}{2}$	$\frac{\pi}{4\sqrt{2}}$	$\frac{\pi}{4\sqrt{2}}$	$\frac{\pi}{4}$	$\frac{\pi}{2}$	0	0

?

Evaluate the perimeters P_4 and P_8 with the previous formula on Fig.38.1



38.4. Python correction



38.4.1. Cell configurations

The following values are reported:

$$\begin{aligned} f^{intra} &= 50 \\ e^{intra} &= 158 \\ v^{intra} &= 107 \end{aligned}$$

$$\begin{aligned} f^{inter} &= 4 \\ e^{inter} &= 42 \\ v^{inter} &= 50 \end{aligned}$$

Then, it is easy to compute the following values:

$$\begin{aligned} A &= f^{intra} = 50 \\ P &= -4f^{intra} + 2e^{intra} = 116 \\ \chi_8 &= v^{intra} - e^{intra} + f^{intra} = -1 \\ \chi_4 &= v^{inter} - e^{inter} + f^{inter} = 12 \end{aligned}$$

38.4.2. Neighborhood configuration

The configuration is computed using the convolution function scipy.signal.convolve2d. The histogram of the different configurations is presented in Fig.38.3.

Be aware that this algorithms works if there is no object pixel touching the borders of the image. The example image is in this case, but you can ensure this property by padding 0 values around the image:



```
X = np.pad(I, ((1,1),), mode='constant');
```



```
# Neighborhood configuration
F = np.array([[1, 4], [2, 8]])
XF = signal.convolve2d(X,F,mode='same');
edges = np.arange(0, 17 ,1);
h,edges = np.histogram(XF[:], bins=edges);
plt.figure()
plt.bar(edges[0:-1], h);
plt.title("Histogram of the different configurations")
plt.show()
```

The Minkowski functionals are computed using the cells contributions:

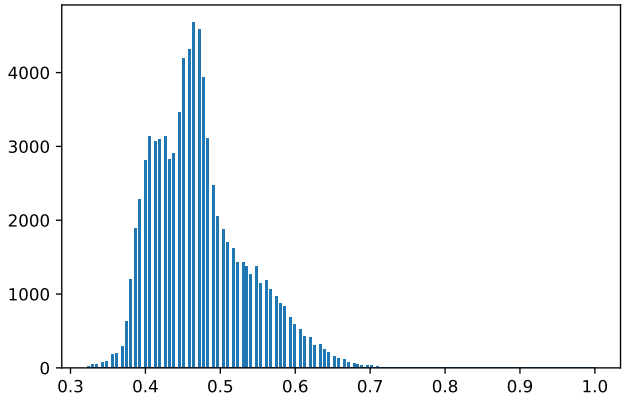


Figure 38.3: Distribution of the different neighborhood configurations.

```
# Computation of the functionals
f_intra = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1];
e_intra = [0,2,1,2,1,2,2,2,0,2,1,2,1,2,2,2];
v_intra = [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1];
EulerNb8 = np.sum(h*v_intra - h*e_intra + h*f_intra)
f_inter = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1];
e_inter = [0,0,0,1,0,1,0,2,0,0,0,1,0,1,0,2];
v_inter = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1];
```

Then, the values are easily verified.

```
EulerNb4 = np.sum(h*v_inter - h*e_inter + h*f_inter)
Area = sum(h*f_intra)
Perimeter = sum(-4*h*f_intra + 2*h*e_intra)
print("E_4 :{0}, A :{1}, P :{2} ".format(EulerNb4, Area, Perimeter));
```

```
E_4 :12, A :50, P :116
```

38.4.3. Crofton perimeter

The Crofton perimeter is a good approximation of a perimeter. One should notice that there is no definitive solution to perimeter evaluation. The Crofton perimeter is approximated in 2 or 4 directions, denoted P_4 and P_8 with a reference to the connectivity.

```
# Crofton perimeter
P4 = [0,np.pi /2,0,0,0, np.pi /2,0,0, np.pi /2,np.pi ,0,0, np.pi /2,np.pi ,0,0];
Perimeter4 = sum(h*P4)
P8 = [0,np.pi /4*(1+1/(np.sqrt(2))),np.pi /(4*np.sqrt(2)),np.pi /(2*np.sqrt(2)),0,np.pi /4*(1+1/(np.sqrt(2))),0,np.
    ↪ pi /(4*np.sqrt(2)),np.pi /4,np.pi /2,np.pi /(4*np.sqrt(2)),np.pi /(4*np.sqrt(2)),np.pi /4,np.pi /2,0,0];
Perimeter8 = sum(h*P8)
```

```
Perimeter4： 91.10618695410399，Perimeter8： 77.76399477870015
```

Remark that the Crofton perimeter is computed this same way in scikit-image.

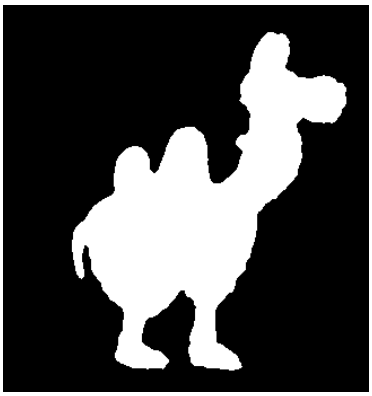
39 Image Characterization

This tutorial aims to characterize objects by geometrical and morphometrical measurements.

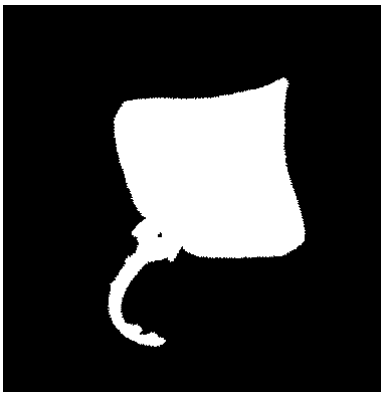
The different processes will be applied on synthetic images as well as images from the Kimia database [1, 40]:



(a) Bat.



(b) Camel.



(c) Ray.

39.1. Perimeters

We are going to calculate the perimeter using the Crofton formula. This formula consists in integrating the intercept number of the object with lines of various orientation and positions. Its expression in the 2-D planar case is given by:

$$P(X) = \pi \int \chi(X \cap L) dL$$

where the Euler-poincaré characteristic χ is equal to the number of connected components of the intersection of X with a line L.
In the discrete case, the Crofton formula can be estimated by considering the intercept numbers for the horizontal i_0, vertical $i_{\pi/2}$ and diagonal orientations $i_{\pi/4}$ and $i_{3\pi/4}$ as:

$$P(X) = \pi \times \frac{1}{4}\left(i_0 + \frac{1}{\sqrt{2}} i_{\pi/4} + i_{\pi/2} + \frac{1}{\sqrt{2}} i_{3\pi/4}\right)$$

?

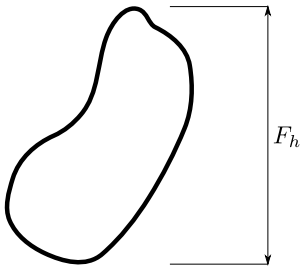
1. Calculate the intercept number of a binary object from the Kimia database with lines oriented in the following four directions: $0, \pi/4, \pi/2, 3\pi/4$.
2. Deduce the value of the Crofton perimeter.
3. Compare the result with the classical perimeter functions.

See perimeter from skimage.measure.

39.2. Feret Diameter

The Feret diameter (a.k.a. the caliper diameter) is the length of the projection of an object in one specified direction Fig.39.1.

Figure 39.1: Feret diameter of the object in horizontal direction.



?

1. Calculate the projections in different directions of a binary object from the Kimia database.
2. Deduce the minimum, maximum and mean value of the Feret diameters.

39.3. Circularity

We want to know if the object X is similar to a disk. For that, we define the following measurement (circularity criterion):

$$circ(X) = \frac{4\pi A(X)}{P(X)^2}$$

where $A(X)$ and $P(X)$ denote the area and perimeter of the object X.

?

1. Show that the circularity of a disc is equal to 1.
2. Generate an array representing an object as a discrete disc.
3. Calculate its circularity and comment the results.

Use numpy.meshgrid.

39.4. Convexity

We want to know if the object X is convex. For that, we define the following measurement:

$$conv(X) = \frac{A(X)}{A(CH(X))}$$

where $CH(X)$ denotes the filled convex hull of the object X.

?

1. Compute the convex hull of a pattern from the Kimia database.
2. Evaluate the area of the filled convex hull.
3. Deduce the convexity of the pattern.

See ConvexHull from scipy . spatial .



39.5. Python correction





```
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage
from scipy import signal
from scipy import misc
from scipy import spatial
from skimage import measure
```

39.5.1. Perimeters

The convolution is used here as an easy way of getting the borders of the object in one direction, i.e. counting the number of intercepts. This method is efficient because, for one given direction, a high number of lines are considered. Keep in mind that only 4 orientations are used in the proposed code.



```
def countIntercepts(I, h):
    B = np.abs( signal.convolve2d(I, h, mode='same'));
    n = np.sum(B) / 2;
    return n;

def perimCrofton(I):
    """
    Approximate the Crofton perimeter with 4 directions
    I is the input binary image
    return the perimeter, float value
    """
    # defines an orientation
    h = np.array([[-1, 1]]);
    n1 = countIntercepts(I, h);

    n2 = countIntercepts(I, h.transpose());

    h = np.array([[1, 0], [0, -1]]);
    n3 = countIntercepts(I, h);

    h = np.array([[0, 1], [-1, 0]]);
    n4 = countIntercepts(I, h);

    perim_Crofton = np.pi/4 * (n1+n2+ 1/np.sqrt(2)*(n3+n4));
    return perim_Crofton;
```



```
Crofton perimeter: 1305.50015965
Classical perimeter in N4: 1374.19509294
Classical perimeter in N8: 1642.82842712
```

39.5.2. Feret diameter

The code consists in rotating the image and evaluating the projected diameter. The rotation function can interpolate the pixel, the nearest method is thus required. The function np.max directly performs the projection in one direction.



```
def feretDiameter(I):
    """
    I: input binary image
    Returns min, max and mean Feret diameter, which is the length of the
    projected object in one direction
    """
    diameter = [];
    for angle in range(180):
        I2 = ndimage.rotate(I, angle, mode='nearest');
        I3 = I2.max(axis=0);
        diameter.append(np.sum(I3 > 0));

    return np.min(diameter), np.max(diameter), np.mean(diameter);
```

For the camel image [1], the Feret diameters are:



```
min, Max, average Feret diameters:  182 325 266.05
```

39.5.3. Circularity

For a disk, the perimeter is $\pi \cdot D$ and the surface is $\pi \cdot \frac{D^2}{4}$, which shows that the circularity criterion has value 1 for a disk.

In order to generate a binary image containing a disk, one simple way is to use the formula: $(x - x_0)^2 + (y - y_0)^2 \leq R^2$. The efficient way to do this is to use np.meshgrid.



```
def disk(t, r):
    """
    Generates a binary array representing a disk, centered, of radius r
    an array of size [2t,2t] is generated
    """
    x = np.arange(-t, t, 1);
    X,Y = np.meshgrid(x, x);
    I = (X**2 + Y**2)<= r**2;
    return I;
```

The circularity uses the perimeters as evaluated earlier.

```python
def circularity(I):
    """
    Circularity criterion
    4*pi*A/P**2
    returns crofton and classic
    """
    P = perimCrofton(I)
    print("Perimeter by crofton: ", P)
    A = np.sum(I);
    C = np.pi*4*A/P**2

    p = measure.perimeter(I, neighbourhood=4)
    print("Usual perimeter: ", p)
    c = np.pi*4*A/p**2
    return C, c
```

This gives for the camel image:

```
Perimeter by crofton: 2514.43300853
Usual perimeter: 2649.36074863
circularity by crofton: 0.999019146137
circularity usual 0.89985338478
```

39.5.4. Convexity

The convex hull is computed with the scipy.spatial tools. The following function plots the result. Notice that the coordinates of an image and the coordinates of an array are different, and one has to flip the image array in order to display both data in the same figure.

```python
def convexity(I):
    """
    Evaluates the convexity criterion
    I is a binary image (np array)
    return convexity
    """
    # be careful that coordinates between images and arrays are inversed
    # thus, image is flipped before extracting points coordinates
    points = np.transpose(np.where(np.flip(I,0)));
    hull = spatial.ConvexHull(points);
    A = np.sum(I);
    Ah = hull.volume;

    plt.figure()
    for simplex in hull.simplices:
        plt.plot(points[simplex, 1], points[simplex, 0], 'k-')
    plt.axis('equal')
    plt.show()
    return A/Ah;
```

This criterion is between 0 and 1. This is illustrated in Fig.39.2.

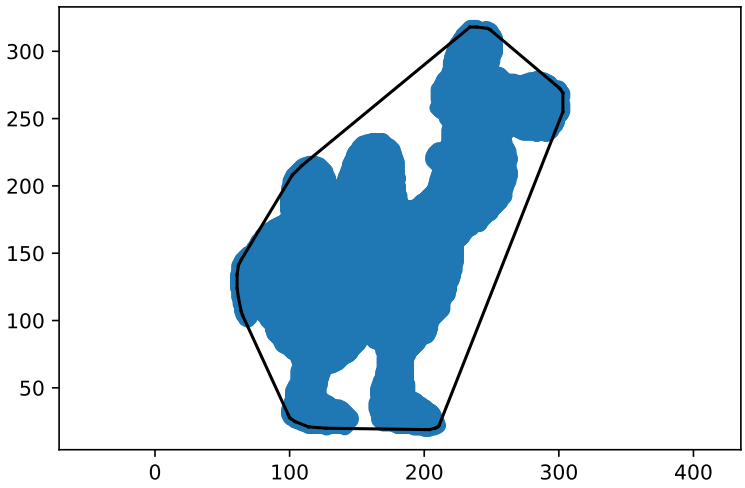


Figure 39.2: Convex hull of the image Camel.

```
convexity of Camel: 0.644014426566
```

40 Shape Diagrams

The objective is to study some shape diagrams and the possibility to define properties that may be useful in order to distinguish the different objects.

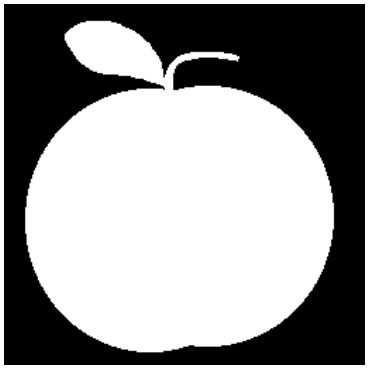
Shape diagrams are representations of single shapes (connected compact sets, see [31, 32, 33]) as points in the 2-D unit square plane. They are based on inequalities between 6 geometrical measurements: area A, perimeter P, radius of the inscribed circle r, radius of the circumscribed circle R, minimum Feret diameter ω and maximum Feret diameter d. In this way, the morphometrical functionals used in the different shape diagrams are normalized ratios of such geometrical functionals. The following table shows the morphometrical functionals for non-convex sets:

Geometrical functionals	Inequalities	Morphological functionals
r, R	$r \leq R$	r/R
ω, R	$\omega \leq 2R$	$\omega/2R$
A, R	$A \leq \pi R^2$	$A/\pi R^2$
d, R	$d \leq 2R$	$d/2R$
r, d	$2r \leq d$	$2r/d$
ω, d	$\omega \leq d$	ω/d
A, d	$4A \leq \pi d^2$	$4A/\pi d^2$
R, d	$\sqrt{3}R \leq d$	$\sqrt{3}R/d$
r, P	$2\pi r \leq P$	$2\pi r/P$
ω, P	$\pi\omega \leq P$	$\pi\omega/P$
A, P	$4\pi A \leq P^2$	$4\pi A/P^2$
d, P	$2d \leq P$	$2d/P$
R, P	$4R \leq P$	$4R/P$
r, A	$\pi r^2 \leq A$	$\pi r^2/A$
r, ω	$2r \leq \omega$	$2r/\omega$

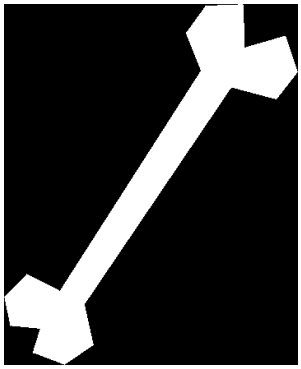
Table 40.1: Morphometrical functionals.

Figure 40.1: The different processes will be applied on images from the Kimia database [1, 40].

(a) Apple.



(b) Bone.



(c) Camel.



40.1. Geometrical functionals

?

Code functions in order to evaluate the different parameters:

- the area, Crofton perimeter and Feret diameters have been already presented in tutorial 38;
- the radius of the inscribed circle can be defined from the ultimate erosion of a set.

The function scipy.ndimage.morphology.distance_transform_cdt computes the distance map of a binary image (chamfer distance transform).

40.2. Morphometrical functionals

?

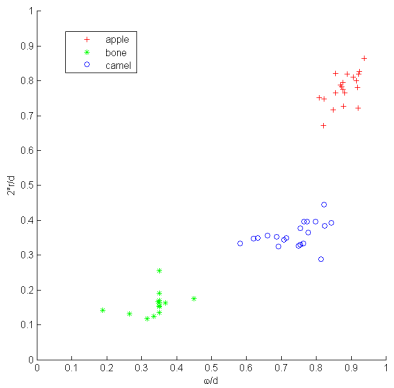
Code and evaluate some of the morphometrical functionals listed in the table 40.1. Note that each of them has a physical meaning, e.g. $\frac{4\pi A}{P^2}$ (circularity), $\frac{4A}{\pi d^2}$ (roundness), $2\omega/P$ (thinness).

40.3. Shape diagrams

?

- Visualize the different shape diagrams for all the images (from the Kimia database) within the three classes 'apple', 'bone' and 'camel'. The Fig.40.2 illustrates the result for the shape diagram $(x = 2r/d, y = P/\pi d)$.
- Which shape diagram is the most appropriate for the discrimination of such objects?

Figure 40.2: Example of a shape diagram.



40.4. Shape classification

?

- Use a K-means clustering method for automatic classification of such shapes.
- Propose a method to quantify the classification accuracy for each shape diagram.



40.5. Python correction





```
import numpy as np

import scipy.ndimage
import imageio # imread and imwrite
import matplotlib.pyplot as plt
import skimage.measure # some geometrical descriptors

# for reading files
import glob

from sklearn.cluster import KMeans
```

40.5.1. Geometrical functionals

The Crofton perimeter is defined by multiple projections, as well as the Feret diameter. Be careful while performing the rotation of the object (as it is a binary object, the interpolation method could introduce non integer values).



```
def crofton_perimeter(I):
    """ Computation of crofton perimeter
    """
    inter = []
    h = np.array([[1, -1]])
    for i in range(4):
        II = np.copy(I)
        I2 = rotate(II, 45*i)>.5
        I3 = scipy.ndimage.convolve(I2, h)

        inter.append(np.sum(I3))

    crofton = np.pi/4. * (inter[0]+inter[2] + (inter[1]+inter[3])/np.sqrt(2))
    return crofton
```



```
def feret_diameter(I):
    """
    Computation of the Feret diameter
    minimum: d (meso-diameter)
    maximum: D (exo-diameter)

    Input: I binary image
    """
    d = np.max(I.shape);
    D = 0;

    for a in np.arange(0, 180, 30):
        I2 = scipy.misc.imrotate(I, a, interp='nearest');
        F = np.max(I2, axis=0);
        measure = np.sum(F>100);

        if (measure<d):
            d = measure;
        if (measure>D):
            D = measure;
    return d,D;
```

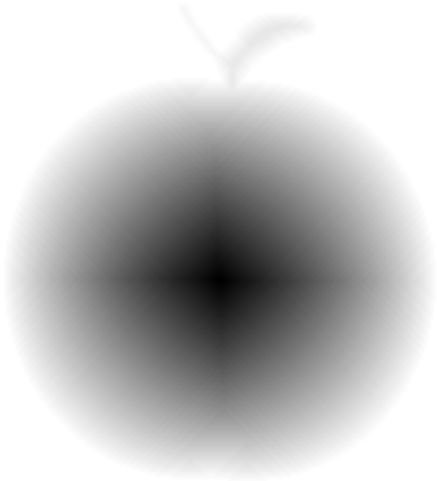
The inscribed circle is just the maximum of the distance transform inside the object. The distance map for an image apple is shown in Fig.40.3a.



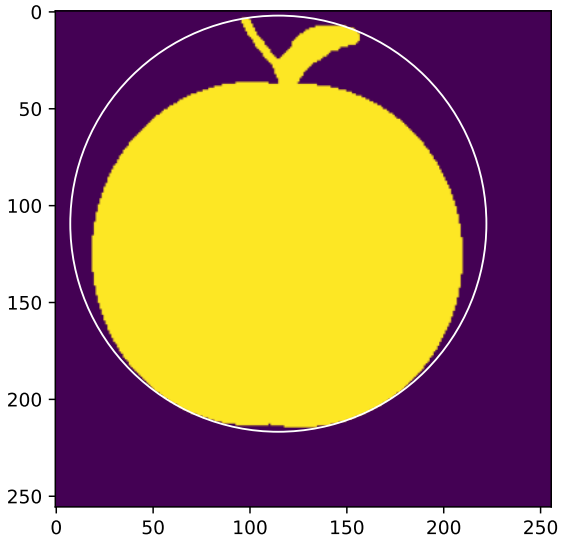
```
def inscribedRadius(I):
    """
    computes the radius of the inscribed circle
    """
    dm = scipy.ndimage.distance_transform_cdt(I > 100)
    radius = np.max(dm);
    return radius;
```

Figure 40.3: Illustration of the computation of two shape parameters.

(a) Distance map of an object apple. The inverse is actually displayed in order to see correctly the progression. The maximum of the distance map is the radius of the inscribed circle.



(b) Circumscribed circle.



The smallest enclosing circle is computed by using the code from the Project Nayuki[1] published under the GNU Lesser General Public License. The result is presented in Fig.40.3b.

```
def circumCircle(I):
    """
    this version uses a function provided by Project Nayuki
    under GNU Lesser General Public License
    """
    points = np.argwhere(I > 100);
    c = smallestenclosingcircle.make_circle(points);
    return c;
```

40.5.2. Shape diagrams

The shape diagrams are constructed by reading all the images and computing the shape descriptors.

[1] https://www.nayuki.io/page/smallest-enclosing-circle

```
def diagrams():
    name=['apple-*.bmp', 'Bone-*.bmp', 'camel-*.bmp'];
    elongation =[];
    thinness =[];
    roundness=[];
    z=[];
    for pattern in name:
        namesList = glob.glob(pattern);
        for fichier in namesList:
            I = imageio.imread( fichier );
            radius = inscribedRadius(I);
            d,D = feret_diameter(I);
            crofton = crofton_perimeter(I);

            elongation.append(d/D);
            thinness.append(2*radius / D);
            roundness.append(4*np.sum(I>100)/(np.pi * D**2));
            z.append(crofton / (np.pi * D));
```

The display of the different plots is just a use of the function plt.plot.

```
    plt.plot(elongation[0:20], thinness[0:20], "o", label='Apple')
    plt.plot(elongation[20:40], thinness[20:40], "+", label='Bone')
    plt.plot(elongation[40:60], thinness[40:60], ".", label='Camel')
    plt.legend(name)
    plt.show
    evaluateQuality(elongation, thinness);

    plt.figure();
    plt.plot(z[0:20], roundness[0:20], "o", label='Apple')
    plt.plot(z[20:40], roundness[20:40], "+", label='Bone')
    plt.plot(z[40:60], roundness[40:60], ".", label='Camel')
    plt.legend(name)
    plt.show
    evaluateQuality(z, roundness);

    plt.figure();
    plt.plot(thinness[0:20], z[0:20], "o", label='Apple')
    plt.plot(thinness[20:40], z[20:40], "+", label='Bone')
    plt.plot(thinness[40:60], z[40:60], ".", label='Camel')
    plt.legend(name)
    plt.show
    evaluateQuality(thinness, z);
```

40.5.3. Shape classification

The following code evaluates the quality by comparing the known class of the shape with the segmented (via the kmeans method) class. The result is illustrated in Fig.40.4 with an accurary of 98.3%.



```
def evaluateQuality(x, y):
    global i
    n = 3;
    k_means = KMeans(init='k-means++', n_clusters=n)
    X = np.asarray(x);
    Y = np.asarray(y);
    pts = np.stack((X, Y));
    pts = pts.T;
    #print(pts)
    k_means.fit(pts);
    k_means_labels = k_means.labels_;
    k_means_cluster_centers = k_means.cluster_centers_;
    # plot
    fig = plt.figure()
    colors = ['#4EACC5', '#FF9C34', '#4E9A06']
    # KMeans
    for k, col in zip(range(n), colors):
        my_members = k_means_labels == k
        cluster_center = k_means_cluster_centers[k]
        plt.plot(pts[my_members, 0], pts[my_members, 1], 'o',
                 markerfacecolor=col, markersize=6)
        plt.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,
                 markeredgecolor='k', markersize=12)
    plt.title('KMeans')
    plt.show()
    fig.savefig("kmeans"+str(i)+".pdf");
    i += 1;
    # Evaluation of the quality: count the number of shapes
    # correctly detected
    accuracy = np.sum(k_means_labels[0:20] ==
                      scipy.stats.mode(k_means_labels[0:20], keepdims=False).mode)
    accuracy += np.sum(k_means_labels[20:40] ==
                       scipy.stats.mode(k_means_labels[20:40], keepdims=False).mode)
    accuracy += np.sum(k_means_labels[40:60] ==
                       scipy.stats.mode(k_means_labels[40:60], keepdims=False).mode)
    accuracy = accuracy / 60 * 100
    print('Accuracy: {0:.2f}%'.format(accuracy));
```

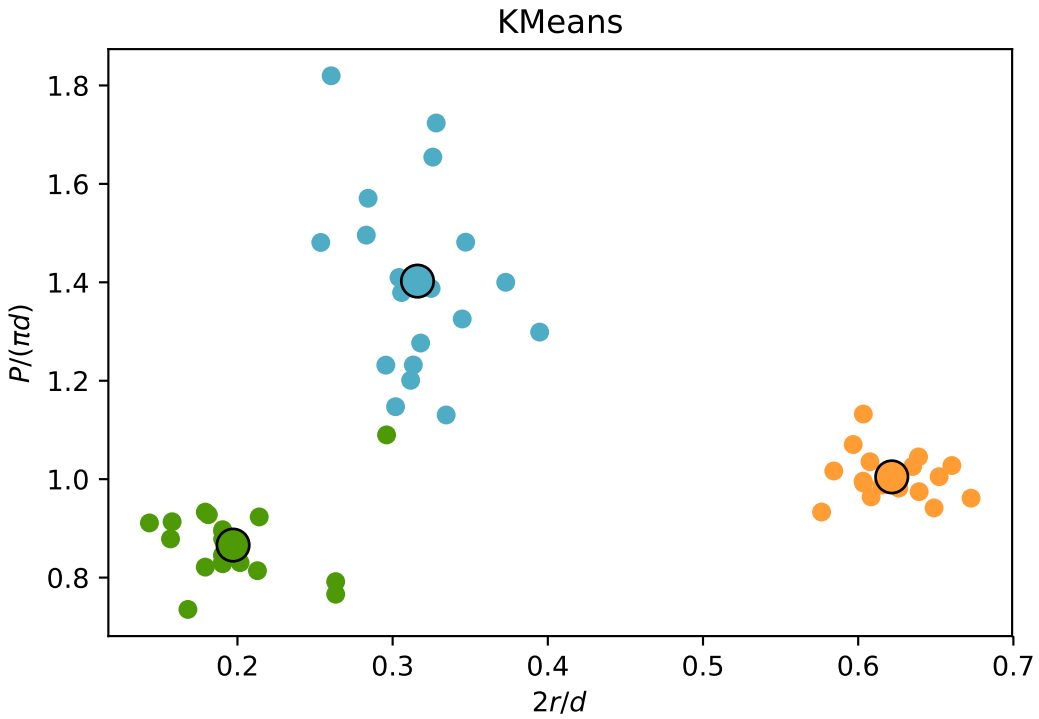


Figure 40.4: Illustration of the accuracy of the classification from a k-means method. The k-means method is not necessarily the adapted to these data. The measured accurary if of 98.3%.

Part X Stochastic Geometry and Random Fields

41 Stochastic Geometry / Spatial Processes

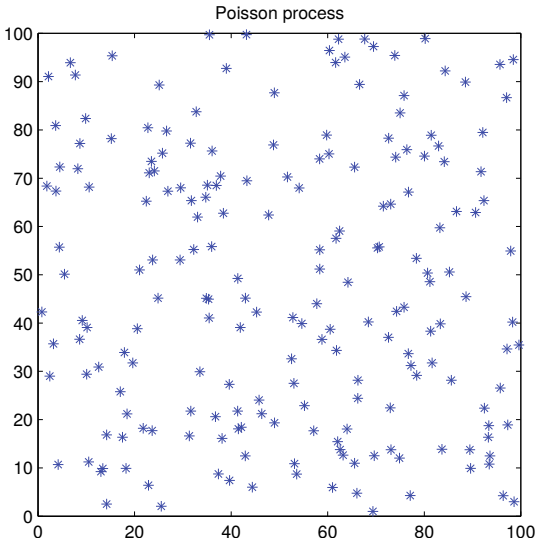
This tutorial aims to simulate different spatial point processes.

41.1. Poisson processes

This process simulates a conditional set of $point_{nb}$ points in a spatial domain D defined by the values $x_{min}, x_{max}, y_{min}, y_{max}$. In order to simulate a non conditional Poisson point process of intensity λ within a domain S, it is necessary to generate a random number of points according to a Poisson law with the parameter $\lambda S : point_{nb} = poisson(\lambda S)$ (i.e. the number of points is a random variable following a Poisson Law).

The coordinates of each point follow a uniform distribution.

Figure 41.1: Conditional Poisson point process, with 200 points.



?

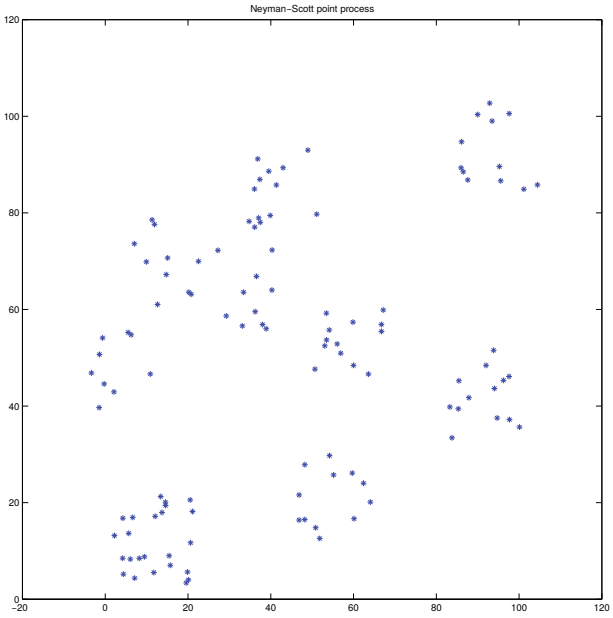
1. Simulate a conditional Poisson point process on a spatial domain D with a fixed number of points (see Fig. 41.1).
2. Simulate a Gaussian distribution of points around a given center.

Import `scipy.stats` to use the function `poisson` and `np.random` for more classical stochastic functions.

41.2. Neyman-Scott processes

This process simulates aggregated sets of points within a spatial domain D defined by the values $x_{min}, x_{max}, y_{min}, y_{max}$. For each aggregate ($n_{par}$), we first generate the random position of the 'parent' point. Then, the 'children' points are simulated in a neighborhood (within a square box of size r^2_{child}) of the 'parent' point. The number of points for each aggregate is either fixed or randomized according to a Poisson law of parameter n_{child} (see Fig. 41.2).

Figure 41.2: Neyman-Scott point process with $h_{child} = 10$ and $n_{par} = 10$



?

1. Simulate a process of 3 aggregates with 10 points.
2. Simulate a process of 10 aggregates with 5 points.

41.3. Gibbs processes

The idea of Gibbs processes is to spatially distribute the points according to some laws of interactions (attraction or repulsion) within a variable range.

Such a process can be defined by a cost function $f(d)$ that represents the cost associated to the presence of 2 separated points by a distance d (see Fig. 41.3). For a fixed value r, if $f(d)$ is negative, there is a high probability to find 2 points at a distance d (attraction). Conversely, if $f(d)$ is positive, there is a weak probability to find 2 points at a distance d (repulsion).

- Code such a function, with prototype `function e=f(d)` or `def energy(d):`, where

$$f(d) = \begin{cases} 50 & \text{if } 0 < d \leq 15 \\ -10 & \text{if } 15 < d \leq 30 \\ 0 & otherwise \end{cases}$$

The generation process reorganizes an initial Poisson point process within the spatial domain according to a specific cost piecewise constant function. The reorganization consists in a loop of nb_{iter} iterations. For each iteration, we calculate the total energy:

$$e = \sum_{(i,j),i \neq j} f(dist(x_i, x_j)) \tag{41.1}$$

The objective is to reduce this energy iteratively. At each iteration step, we try to replace a point by four (for example) other random points and we calculate the energy for each new configuration. The initial point is either preserved or replaced (by one of the four points) according to the configuration of minimal energy (see Fig.41.4).

Figure 41.3: Energy function f.

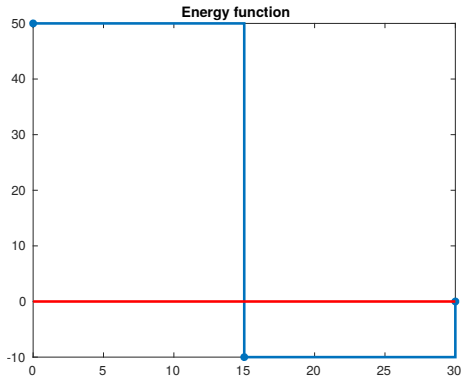


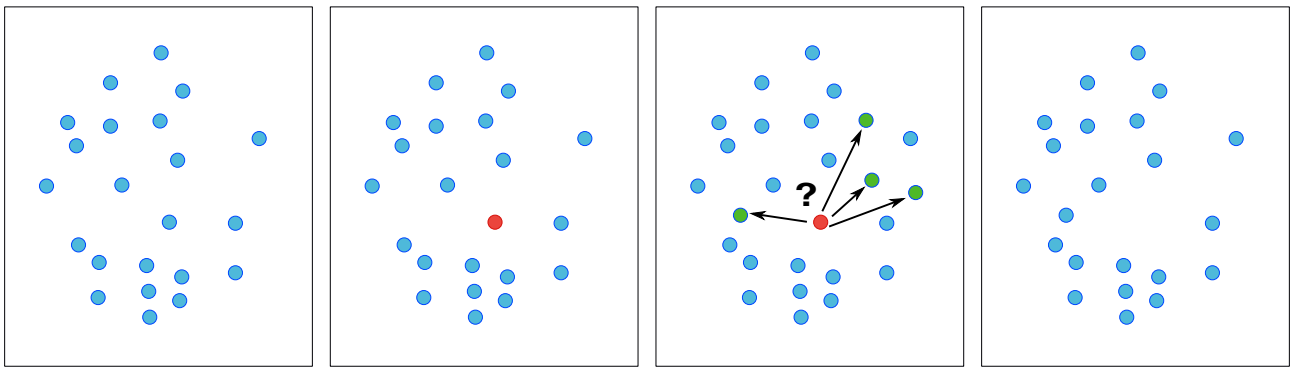
Figure 41.4: Illustration of iterative construction of the Gibbs point process.

(a) Start with an initial distribution of points. The energy is computed for all pairs of points.

(b) A random point is selected.

(c) Four candidate points are randomly generated, the energy of these four new configurations is evaluated.

(d) The configuration with the minimal energy is retained, other points are discarded.



?

1. Code a function with the following prototype (the function *energyFunction* is previously noticed f). It computes the energy between all the points present in an array of coordinates $[x, y]$ (the points that do not move) and point $[x_k, y_k]$ (the new point). The *energyFunction* converts a distance to an 'energy', reflecting attractivity or repulsivity.

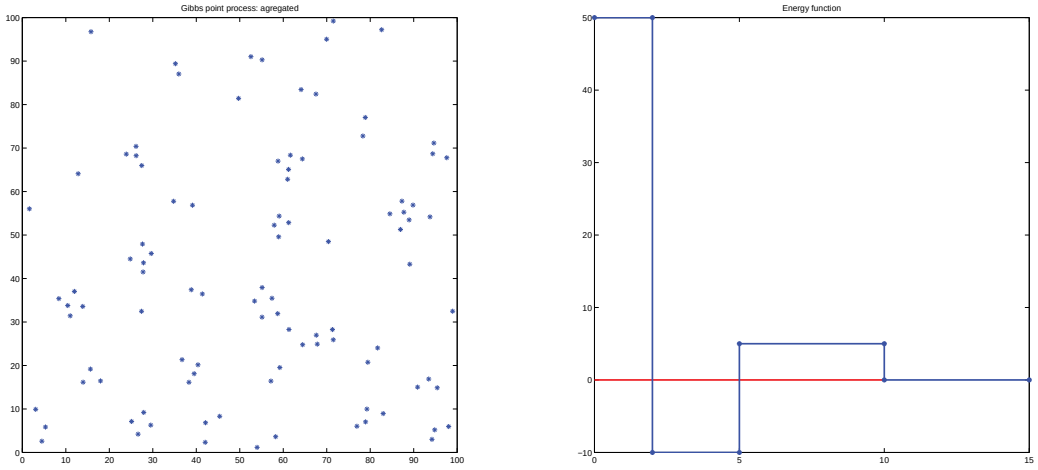
```
def energy(P, eFunction=exampleEnergyFunction):
```

2. Simulate a regular point process by choosing a specific energy function.
3. Change the cost function and simulate a few aggregated point processes (see Fig. 41.5).

Informations

Use the function `pdist` from `scipy.spatial.distance`, which computes pairwise distances of all points.

Figure 41.5: Gibbs agregated point process and its energy function.



41.4. Ripley function

The Ripley function $K(r)$ characterizes the spatial distribution of the points. For a Poisson process of density λ, $\lambda K(r)$ is equal to the mean value of the number of neighbors at a distance lower than r to any point. In the case where the process is not known (λ?), the Ripley function has to be estimated (approximated) with the unique known realization. It is the first estimator of $K(r)$:

$$K(r) = \frac{1}{\lambda}\frac{1}{N}\sum_{i=1}^{N}\sum_{i\neq j} k_{ij} \tag{41.2}$$

where N is the number of points in the studied domain D, $\lambda = N/D$ is the estimator of the density of the process and k_{ij} takes the value 1 if the distance between the point i and the point j is lower than r, and 0 in the other case.

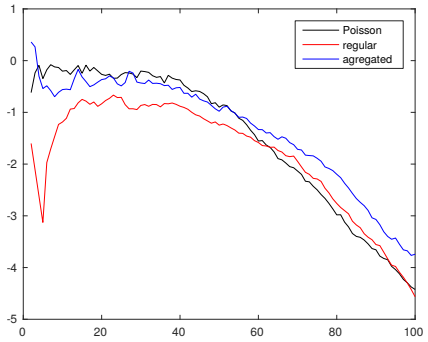
We denote:

$$L(r) = \sqrt{K(r)/\pi} \tag{41.3}$$

?

1. Code a function to compute K (`function K=ripley(points, box, r)`), with `points` being the considered points, `box` the simulation domain, and r the distance (or an array of all distances).
2. Calculate the Ripley function for an aggregated point process, a Poisson point process and a regular point process.
3. Display the value $L(r) - r$ as in Fig 41.6.

Figure 41.6: Ripley's function $L(r) - r$.



41.5. Marked point processes

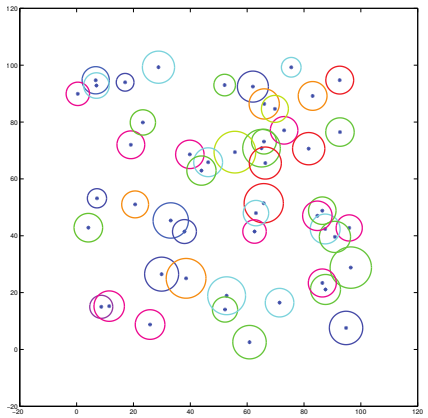
To simulate a complex point process, it can be useful to associate several random variables (marks) for each point.

?

1. Simulate a disk process, where the points (disk centers) are defined according to a Poisson process and the radii are selected with a Gaussian law.
2. Add a second mark for allocating a color to each disk (uniform law).

An example result is shown in Fig. 41.7

Figure 41.7: A Poisson point process is used to generates the center of the circles. The radii are chosen according a Gaussian law, and the color according a uniform law.



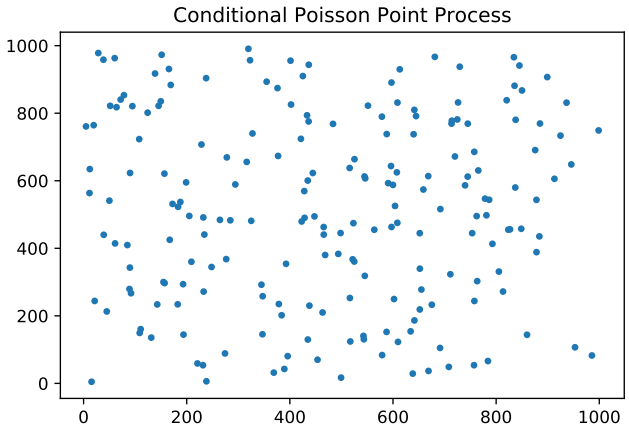
41.6. Python correction

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist
```

41.6.1. Conditional Poisson Process

The conditional Poisson Point Process uses a given number of point, contrary to the Poisson Process where the number of points follows a Poisson law. The result is illustrated in Fig.41.8.

Figure 41.8: Conditional Poisson point process, with N=100 points.




```
def cond_Poisson(nb_points, xmin, xmax, ymin, ymax):
    # Conditional Poisson Point Process
    # uniform distribution
    # nb_points: number of points
    # xmin, xmax, ymin, ymax: defined the domain (window)
    x = xmin + (xmax-xmin)*np.random.rand(nb_points)
    y = ymin + (ymax-ymin)*np.random.rand(nb_points);
    return x,y

def test_ppp():
    # testing function
    x,y = cond_Poisson(100, 0, 100, 0, 100);
    plt.plot(x,y,'+');
```

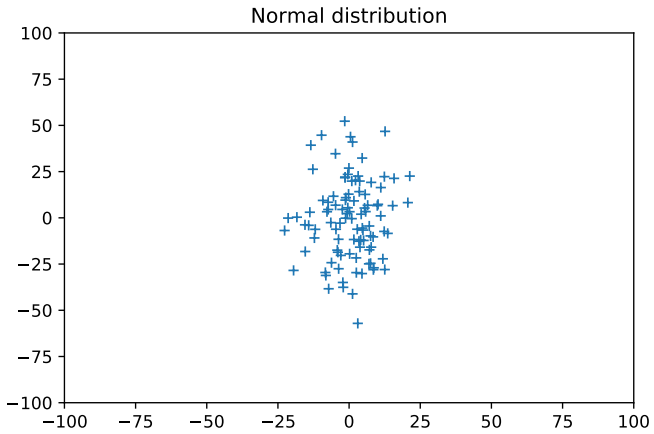
41.6.2. Normal distribution

The normal distribution is illustrated in Fig.41.9. Each marginal distribution (distribution on each axis) follows the normal distribution.



```
def normal_distribution(nb_points, mu, sigma):
    # Normal distribution centered around the point mu with stdev sigma
    x = mu[0] + sigma[0]*np.random.randn(nb_points);
    y = mu[1] + sigma[1]*np.random.randn(nb_points);
    return x,y;
```

Figure 41.9: Normal distribution of points around $(0, 0)$, with $\sigma = (10, 20)$.



41.6.3. Neyman-Scott Process

Neymann-Scott point process is an agregated Poisson point process. It is illustrated in Fig.41.10. It consists on a "sub" point processes generated at locations corresponding to a point process.

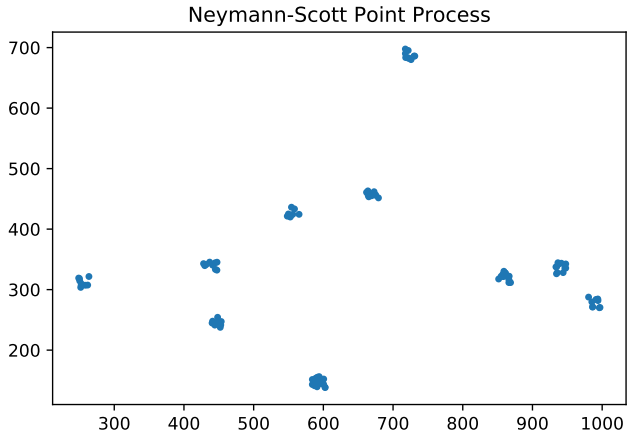


```
def neyman_scott(nRoot, xmin, xmax, ymin, ymax, lambdaS, rSon):
    # Neyman-scott process simulation
    # nRoot: number of agregates
    # xmin, xmax, ymin, ymax: domain
    # lambdaS: number of points. lambda is a density, S is the spatial domain
    # rSon: radius around agregate (points are distributed in a square)

    # number of sons, follows random law
    nSons = poisson.rvs(lambdaS, size=nRoot);
    # results
    x=[];
    y=[];
    # father points coordinates
    xf,yf = cond_Poisson(nRoot, xmin, xmax, ymin, ymax);
    for i in range(nRoot):
        # loop over all agregates
        xs,ys = cond_Poisson(nSons[i], xf[i]-rSon, xf[i]+rSon, yf[i]-rSon, yf[i]+rSon);
        x = np.concatenate((x,xs), axis=0);
        y = np.concatenate((y,ys), axis=0);

    return x,y
```

Figure 41.10: Neyman-Scott point process, with lambdaS=10 and rSon=10.



41.6.4. Gibbs Point Process

Gibbs point process allows attraction and repulsion at different distances. It is illustrated in Fig.41.11. The attraction/repulsion law is given by the following code for regular or agregated point process. Notice that this energy involves a distance, and thus is related to the size of the generation window (in the following examples, a rectangle of size 1000×1000 is used).

```
def regularEnergyFunction(distance):
    """ This function returns e with the same size as distance
    e takes the value given in the variable energy according to the steps
    """
    e = np.zeros(distance.shape)
    e[distance<10] = 10
    return e

def agregatedEnergyFunction(distance):
    """
    Agregated energy function
    """
    e = np.zeros(distance.shape)
    e[distance<2] = 50
    e[np.logical_and(distance>=2, distance<5)] = -10
    e[np.logical_and(distance>=5, distance<10)] = 5

    return e;
```

The evaluation of the energy computes all pairwise distances and sum up the energies associated, or it computes the distances between one single point to a set of points.



```
def energy(P, eFunction=exampleEnergyFunction):
    """
    This computes the energy in the set of points P, with the energy function
    given as a parameter.
    return a float value
    """
    P = np.transpose(P);
    d = pdist(P);
    e = eFunction(d);
    return np.sum(e);

def energyFromPoint(p, P, eFunction=exampleEnergyFunction):
    """
    Compute energy from point p to all points of P
    """
    dist = np.sqrt((p[0] - P[0,:])**2 + (p[1] - P[1,:])**2);
    ee = eFunction(dist);
    return np.sum(ee);
```

The principle of the algorithm is to iteratively add one point that minimizes the energy after several trials. In order to speed up the process, notice that only one point is moved, and it is thus sufficient to only compute the distances from this point to all others.

```python
def gibbs(nb_points, xmin, xmax, ymin, ymax, nbiter, eFunction=exampleEnergyFunction):
    """
    Gibbs point process
    xmin, xmax, ymin, ymax represents the spatial window
    nb_points: number of generated points
    nbiter: number of iterations
    returns (x,y) coordinates of the points
    """

    # start with a Poisson Point Process
    x,y = cond_Poisson(nb_points, xmin, xmax, ymin, ymax);
    nb_moves = 0;
    e_prev = energy(np.vstack((x,y)), eFunction);
    print("initial energy: {0:f}".format(e_prev));

    for i in range(nbiter):
        # choose a random point
        j = np.random.randint(0, nb_points);
        x2 = np.delete(x, j);
        y2 = np.delete(y, j);

        P = np.vstack((x2, y2));
        e1 = energyFromPoint([x[j], y[j]], P, eFunction);

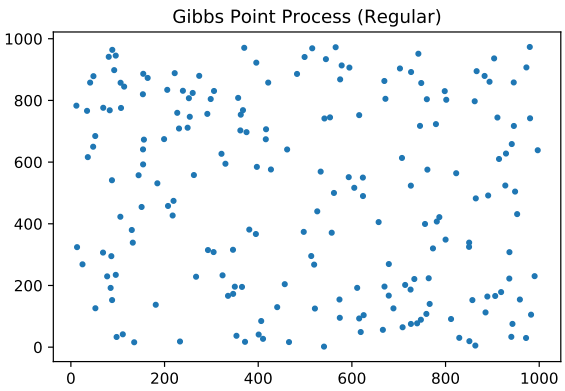
        for m in range(10):
            xm,ym = cond_Poisson(1, xmin, xmax, ymin, ymax);

            e2 = energyFromPoint([xm, ym], P, eFunction);
            if e2<e1:
                nb_moves+=1;
                x[j]=xm;
                y[j]=ym;
                e1=e2;

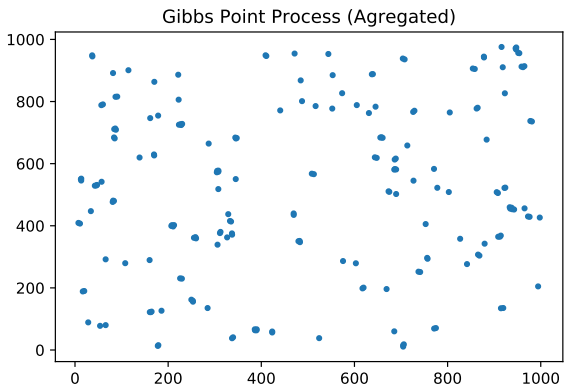
    print( "Number of moves: " + str(nb_moves));
    print("Final energy: " + str(e1));
    return x, y
```

Figure 41.11: Gibbs point processes (with the same number of points).

(a) Regular Gibbs point process.



(b) Agregated Gibbs point process.



41.6.5. Ripley functions

The Ripley functions are useful to characterize a point process. Agregation and repulsion can be observed with regard to the distance (see Fig.41.12). Notice that this function is biased because points in border of window are counted as points in the center. This could be corrected by the use of scipy . spatial . distance . cdist .

```
def ripley(x, y, xmin, xmax, ymin, ymax, edges):
    # Ripley K and L functions, vals is values of radius
    # this function has border effects !
    # x, y: coordinates of points
    # xmin, xmax, ymin, ymax: window
    # edges: values of bins for histogram evaluation

    # number of points
    nb_points = x.size;

    # compute pairwise distances
    P = np.transpose(np.vstack((x,y)));
    d = pdist(P);

    # compute cumulative histogram
    h,edges = np.histogram(d, edges);
    H = np.cumsum(h);

    # normalization of K
    K = 2*H/nb_points;
    area = (xmax-xmin)*(ymax-ymin);
    density = float(nb_points) / area;
    K = K / density;

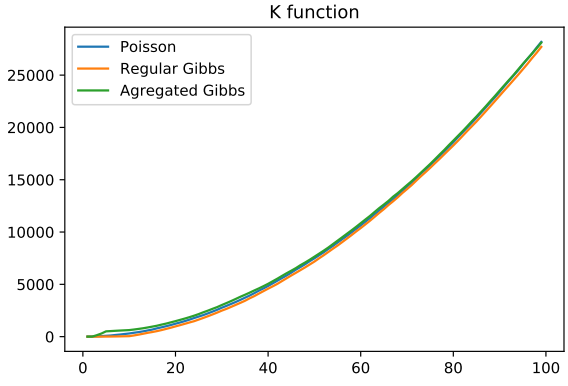
    # L
    L = np.sqrt(K/np.pi);

    # edge values
    vals = edges[:-1] + np.diff(edges);

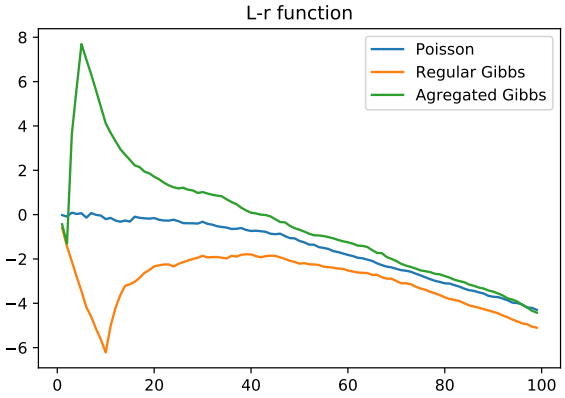
    return K, L, vals
```

Figure 41.12: Ripley functions.

(a) Ripley K function.



(b) Ripley L function.



41.6.6. Marked Point Process

An illustration is presented in Fig.41.13. The algorithm simply consists in adding two new random variables in order to generate the radius and the color of each point.

```python
def marked(nb_points, xmin, xmax, ymin, ymax):
    """
    marked point process
    """

    # points
    x,y = cond_Poisson(nb_points, xmin, xmax, ymin, ymax);

    # first mark: radii
    sigma = 5;
    mu = 10;
    r = sigma * np.random.randn(nb_points) + mu;
    r[r<0.1] = 0.1;

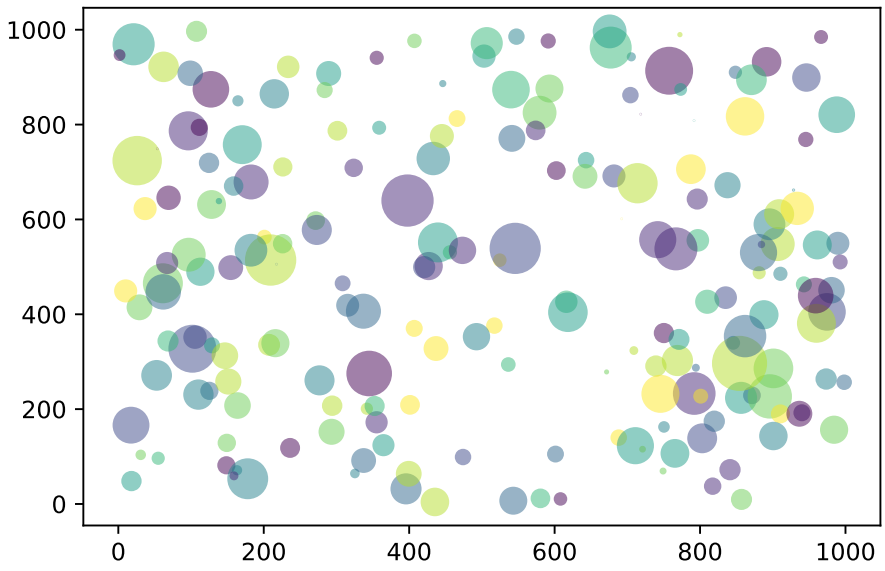
    # second mark: colors
    nb_colors = 10;
    c = np.random.randint(nb_colors, size = nb_points);

    # plot
    plt.scatter(x, y, r**2, c, alpha=.5);

    # save pdf figure
    plt.savefig("marked.pdf");

nb_points = 100;
N=100;
marked(nb_points, 0, N, 0, N);
```

Figure 41.13: Marked point process.

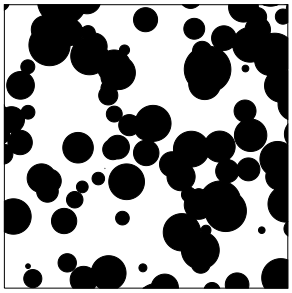


42 Boolean Models

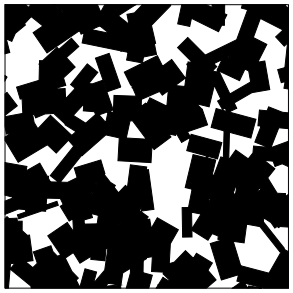
This tutorial aims to study a classical model coming from stochastic geometry: the Boolean model. The first objective is to simulate some realizations of a Boolean model of 2-D disks representing a population of overlapped particles. Thereafter, the geometrical characteristics of the individual disks (from a statistical point of view) will be analyzed.

Figure 42.1: Illustration of 2-D Boolean models observed in a squared window Ω.

(a) Boolean model of disks



(b) Boolean model of rectangles



42.1. Simulation of a 2-D Boolean model

Let $\{x_i; i \in \mathbb{N}\}$ be a random collection of points in $\mathbb{R}^2$ forming a stationary Poisson point process with intensity $\lambda > 0$. Let $Z_0, Z_1, Z_2, \dots$ be independent, identically distributed random 2-D convex bodies (nonempty, compact, convex sets) with distribution $\mathbb{Q}$, which are independent of the point process $\{x_i; i \in \mathbb{N}\}$. The random points $x_1, x_2, \dots$ are the germs and the random sets $Z_1, Z_2, \dots$ are the grains of the Boolean model. The random set Z_0 is called the typical grain. The union of the translated grains:

$$Z = \bigcup_{i=1}^{\infty} (Z_i + x_i) \tag{42.1}$$

is a random closed set, which is called the stationary Boolean model with intensity λ and grain distribution $\mathbb{Q}$. The random collection $X = \{Z_1 + x_1, Z_2 + x_2, \dots\}$ of the shifted grains is the particle process underlying the Boolean model.

Figure 42.1 shows a realization of two different Boolean models Z observed in a compact and convex observation window Ω.

We are going to simulate some realizations of a Boolean model of 2-D disks in a squared observation window. The final simulated model will be represented as a discrete binary image.

?

- Generate a 2-D discrete observation window Ω of size 500×500.
- Generate the random germs that follows a Poisson law with intensity $\lambda = 100/(500 * 500)$. Take care of the edge effects (a grain with a germ outside the observation window could intersect it!)
- Generate the random grains (as disks). The disk radius will follow a uniform distribution $\mathcal{U}(10, 50)$.
- Vizualize the realization.

Look at the numpy functions random.poisson and random.randint.
Use skimage.draw. circle to generate the disks in an array.

42.2. Geometrical characterization of a 2-D Boolean model

Assume we observe Z in a compact, convex observation window Ω with positive volume (as shown in Figure 42.1). Our aim is to extract distributional information from the geometric properties of the sample $Z \cap \Omega$. Thus, we assume we can measure geometric functionals like the perimeter of the sample $Z \cap \Omega$ which is a finite union of convex bodies (a polyconvex set). By a geometric functional ϕ we mean a real-valued functional defined on the space of polyconvex sets with specific additional properties. Important examples of geometric functionals are the Minkowski functionals W_n that are related in the 2-D space to the measures of area (A), perimeter (P) and Euler number (χ):

$$W_0 = A \tag{42.2}$$
$$W_1 = P/2 \tag{42.3}$$
$$W_2 = \pi\chi \tag{42.4}$$

The boundary of the observation window Ω has a disturbing effect. It is therefore of advantage to assume a sufficiently large observation window and to consider only limits as the window tends to infinity. This motivates the introduction of the density (or specific value) of the Boolean model for a geometric functional ϕ. The density of ϕ is the combined spatial and probabilistic mean value:

$$\overline{\phi}(Z) = \lim_{r\to\infty} \frac{\mathbb{E}[\phi(Z \cap r\Omega)]}{W_0(r\Omega)} \tag{42.5}$$

The crucial problem when studying a Boolean model is, that the particles overlap and can therefore not be observed individually.

For this purpose, the Miles formula gives relationships between the global Minkowski densities and the Minkowski densities of the particle. For isotropic Boolean models and by using the densities of the particle process X:

$$\overline{\phi}(X) = \lambda\mathbb{E}[\phi(Z_0)] \tag{42.6}$$

Miles formulas express the observable Minkowski densities $\overline{W}_n(Z)$ in terms of the Minkowski densities $\overline{W}_n(X)$ and can be inverted.
For example in 2-D:

$$\overline{W}_0(Z) = 1 - e^{-\overline{W}_0(X)} \tag{42.7}$$
$$\overline{W}_1(Z) = e^{-\overline{W}_0(X)}\overline{W}_1(X) \tag{42.8}$$
$$\overline{W}_2(Z) = e^{-\overline{W}_0(X)}\left(\overline{W}_2(X) - \overline{W}_1(X)^2\right) \tag{42.9}$$

Practically, in 2-D, A, P, χ are the measures (area, perimeter, Euler number) obtained in the entire observation, a, p, x are the measures of the typical grain, and λ is the density of the process. Ω_{size} is the area of the observation window Ω.

$$\frac{A}{\Omega_{size}} = 1 - e^{-\lambda a} \tag{42.10}$$
$$\frac{P}{\Omega_{size}} = e^{-\lambda a} \times \lambda p \tag{42.11}$$
$$\frac{\pi\chi}{\Omega_{size}} = e^{-\lambda a}\left(\pi\lambda x - \frac{1}{4}(\lambda p)^2\right) \tag{42.12}$$

?

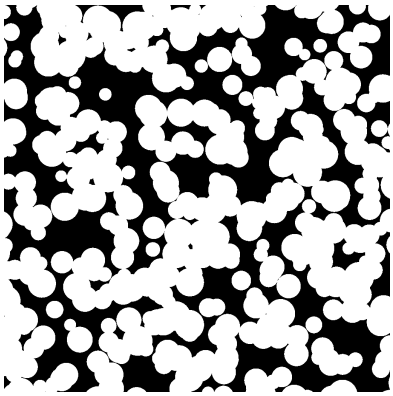
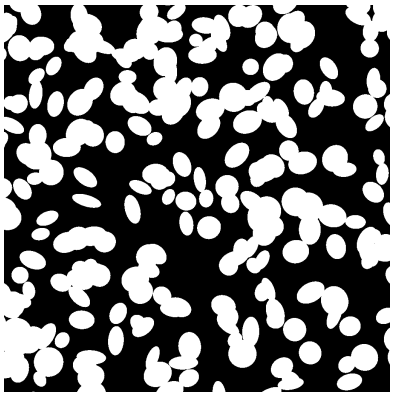
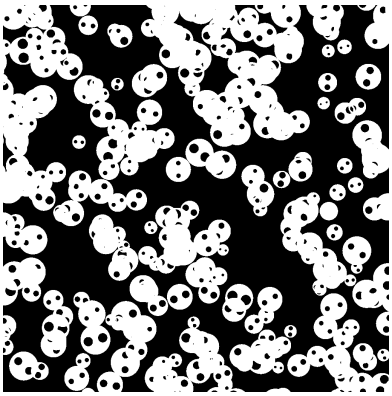
- Generate different realizations of the previous Boolean model and compute the Minkowski densities of Z (by using the functions done in the tutorial "Integral Geometry").
- Compute the theoretical Minkowski densities of Z by using the Miles formulas.
- Compare the computed and theoretical values.

Useful Python functions from skimage.measure: perimeter_crofton and euler_number. Use np.sum for computing the area.

42.3. Random model analysis

This part of the tutorial is an assignment and will get marked.

Let us consider a set of binary images of size 1024×1024 made of white particles on a black background. The particles are disks, ellipses, button-like or window-like shapes.

?

- For a disk or an ellipse, what is x (its Euler number)?
- Propose and implement a method to retrieve the characteristics of the mean particle for each set of images at your disposal: area, perimeter and mean radius for disks; mean equivalent radius for ellipses.
- Explain your methodology and any assumptions you may have to make.

42.3.1. Useful programming considerations

Useful Python functions: skimage.measure.area, skimage.measure.perimeter, skimage.measure.euler_number.

In order to load a set of images contained in a directory, you might use:



```
from glob import glob
import skimage.io

listOfFiles = glob("pix500/*.png")
W = np.zeros((len(listOfFiles), 3))
for i, filename in enumerate(listOfFiles):
    X = skimage.io.imread(filename)
```



42.4. Python correction



```
import numpy as np
from skimage import draw
from scipy import misc, signal
import matplotlib.pyplot as plt
import progressbar
```

42.4.1. Simulation of a 2-D Boolean model

The process for simulating the proposed Boolean model of 2-D disks consists in four steps:

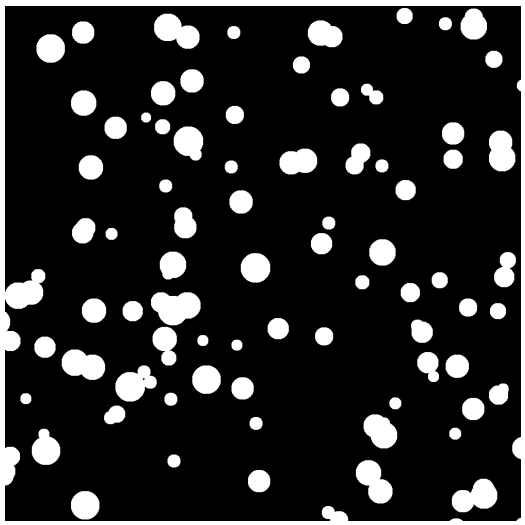
1. Generate the random number of points (by using the intensity parameter of the Poisson distribution). In order to avoid edge effects, one consider a larger window than the observation window to generate the disks. Indeed, a germ outside the observation window can generate a disk that intersects the observation window.

2. Generate the random locations of the germs (random coordinates from a uniform distribution).

3. Generate the random size of the disks (random radius from a probability distribution).

4. Generate the union of disks.

Figure 42.2: Boolean model of disks, with $Wsize = [1000, 1000]$ and $RadiusParam = [10, 30]$.



When executing this simulation with the following parameters, we get a realization of the Boolean model as a binary image in Fig.42.2.



```
Wsize=[1000, 1000];
gamma = 100 / (Wsize[0] * Wsize[1]);
radius = [10, 30];
Z = booleanModel(Wsize, gamma, radius);
plt.imshow(Z);
plt.show();
```

Here is the global function for generating such a Boolean model as a binary image:

```python
def booleanModel(Wsize, gamma, radius):
    """
    Generation of a 2D boolean model of disks, in a window of size Wsize
    Wsize: 2x1 array
    gamma: numerical value to control the Poisson process
    radius: min and max values of radii, 2x1 array
    returns: boolean array of size Wsize
    """
    edgeEffect =2 * np.max(radius) + 100;
    WsizeExtended = Wsize + 2* edgeEffect;
    # nb of points
    areaW = WsizeExtended[0] * WsizeExtended[1];
    nbPoints = np.random.poisson(lam = gamma * areaW);
    # positions of the germs
    x = np.random.randint(0, WsizeExtended[0], nbPoints);
    y = np.random.randint(0, WsizeExtended[1], nbPoints);
    # grains
    rGrains = np.random.randint(radius[0], radius[1], nbPoints);
    # union of grains
    Z = np.zeros((WsizeExtended[0], WsizeExtended[1]));
    for r, xx, yy in zip(rGrains, x, y):
        rr, cc = draw.circle(xx, yy, radius=r, shape=Z.shape)
        Z[rr, cc] = 1
    # restrain window for side effects
    Z = Z[edgeEffect:edgeEffect+Wsize[0], edgeEffect:edgeEffect+Wsize[1]]
    return Z
```

42.4.2. Geometrical characterization of a 2-D Boolean model

We can use the following function to compute the Minkowski functionals of the Boolean model . The regionprops function from skimage.measure is not used because it computes the properties of each object.

```python
import skimage
def minkowskiFunctionals(X):
    Area = np.sum(X>0)
    Perimeter = skimage.measure.perimeter(X, neighbourhood=4)
    EulerNb = skimage.measure.euler_number(X, connectivity=2)

    return Area, Perimeter, EulerNb
```

So we can estimate the Minkowski densites on different realizations of the Boolean model:

```python
def realizations (Wsize, gamma, radius, n=100):
    """
    This function iterates the different realizations
    Wsize: window size
    gamma: value of gamma, see booleanModel
    radius: min and max values of the radii of the generated disks
    """
    W = np.zeros((n, 3));
    areaWsize = Wsize[0] * Wsize[1];
    bar = progressbar.ProgressBar();
    for i in bar(range(n)):
        Z = booleanModel(Wsize, gamma, radius);
        a, p, chi = minkowskiFunctionals(Z);
        W[i,:] = np.array([a, p/2, chi*np.pi]) / areaWsize;

    return W;
```

Therefater, we can compare the estimated Minkowski mean densities of the Boolean model with the theoretical ones (by using the known parameters of the different probability distributions of this Boolean model):

```python
W = realizations (Wsize, gamma, radius, 1000);
W = np.mean(W, axis=0);
```

```python
# comparison with analytical values
rMean = np.mean(radius);
areaMean = np.pi*rMean**2;
perMean = 2*np.pi*rMean;

W_X = gamma * np.array([areaMean, perMean/2, np.pi]);
W_0 = 1-np.exp(-W_X[0]);
W_1 = np.exp(-W_X[0]) * W_X[1];
W_2 = np.exp(-W_X[0]) * (W_X[2] - W_X[1]**2);

error_0 = np.abs(W_0-W[0]) / W_0;
error_1 = np.abs(W_1-W[1]) / W_1;
error_2 = np.abs(W_2-W[2]) / W_2;
```

Here are the results for 100 specific realizations:

```
errorW0:  0.0190201754056
errorW1:  0.200478931771
errorW2:  0.0342984925035
```

The errors can be large due to the biais estimation of the Minkowski densities within an observation window (specifically for the perimeter and the Euler number). But you can use unbiased estimators which can be found in the literature.

★★★ 43 Geometry of Gaussian Random Fields

This tutorial aims at simulating Gaussian random fields and analyzing their geometry.

43.1. Introduction

Let ϕ be a real-valued stationary Gaussian Random Field (GRF) on $\mathbb{R}^2$. $\forall p \in \mathbb{R}^2$, $\phi(p)$ defines a random variable. The family $(\phi(p), p \in \mathbb{R}^2)$ consists of identically distributed random variables on a probability space $\Omega, \mathcal{F}, P$ that satisfies for any subset of $\{p_1, ..., p_n \in \mathbb{R}^2\}$ and $\alpha_k \in \mathbb{R}$ that the random variable $\sum_{k=1}^{n} \alpha_k \phi(p_k)$ is normally distributed. The reader could find more informations in [44, 2, 3].

Thus, ϕ is completely characterized by its mean and its covariance:

$$m = \mathbb{E}(\phi(p)) = \mathbb{E}(\phi(0)) \quad , p \in \mathbb{R}^2 \tag{43.1}$$

$$C(p) = \mathbb{E}(\phi(0)\phi(p)) - m^2 = \mathbb{E}(\phi(q)\phi(p+q)) - m^2, p, q \in \mathbb{R}^2 \tag{43.2}$$

In this tutorial, we will assume $m = 0$. The goal is to construct ϕ for a given covariance C.

43.2. Simulation

For more details on algorithms simulating GRFs in $\mathbb{R}^d$, see [22].

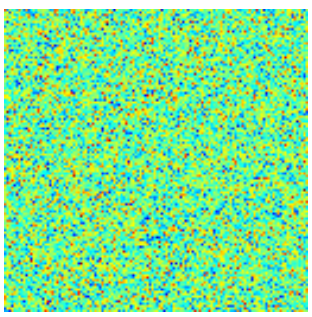
43.2.1. Gaussian white noise random field

?

- Generate a white noise on $\mathbb{R}^2$. Choose an $n \times n$ size. Display it on the screen (see for example Fig. 43.1).
- Modify the variance and observe the results.

Use randn from numpy.random. Use imshow from matplotlib.pyplot.

Figure 43.1: Gaussian White Noise, with $m = 128$ and $\sigma = 30$ for display purposes.



43.2.2. Gaussian Random Field

Let W be an independent centered white noise random fields. $\mathcal{F}$ is the Fourier Transform. The gaussian random field ϕ is defined by:

$$p \in \mathbb{R}^2, \phi(p) = \mathcal{F}^{-1}(\hat{\phi})(p)$$

and

$$\hat{\phi}(k) = \sqrt{\mathcal{F}(C)(k)\mathcal{F}(W)}$$

?

- Generate a 2D Gaussian covariance function , see Fig. 43.2a (we recall the definition, $u^2 = x^2 + y^2$):

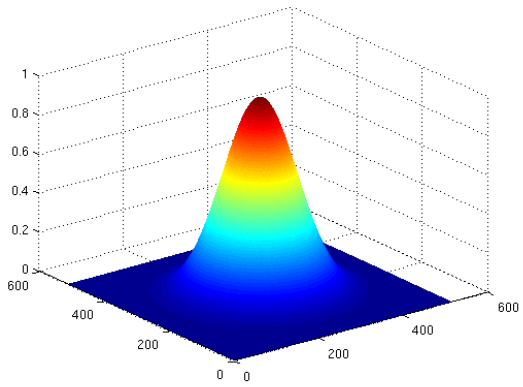
$$C(u) = e^{-\frac{u^2}{2\sigma^2}}$$

- Generate the Gaussian random field (see Fig. 43.2b). Test with different values of σ.

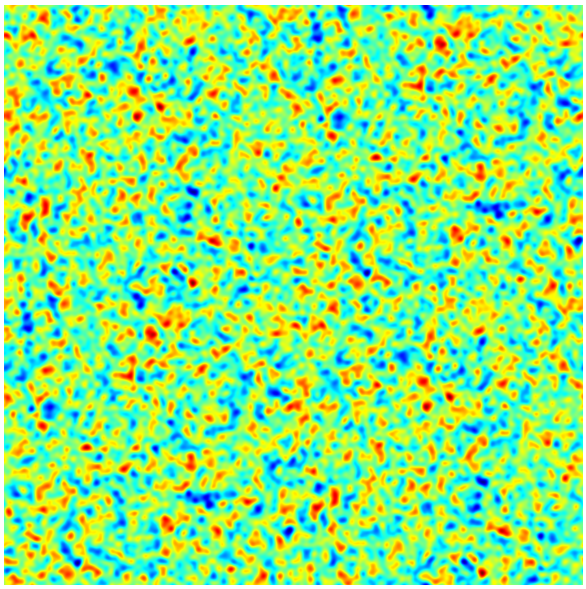
Use meshgrid from numpy.

Figure 43.2: Covariance and Gaussian random field examples.

(a) Generation of a 2D Gaussian function.



(b) Generation of a 2D Gaussian random field with Gaussian covariance.



43.3. Geometry

43.3.1. Excursion set

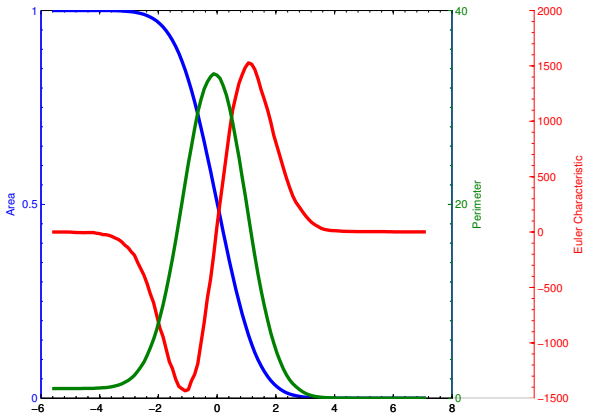
Let $Y(x)$, $x \in \mathbb{R}^2$, be a stationary real-valued random field. An excursion set, denoted by $E_h(Y, S)$, of Y inside a compact subset $D \subset \mathbb{R}^2$ above a level h, is defined as:

$$E_h(Y, D) = \{p \in D : Y(p) \geq h\}$$

In $D \subset \mathbb{R}^2$, an excursion set is a binary image.

Represent, for each level h, the area A, the perimeter P and the Euler number χ of $E_h(Y, D)$ (see Fig. 43.3). You can have a look at the tutorial about integral geometry for perimeter, area and Euler number computation.

Figure 43.3: Example of computation of Area, Perimeter and Euler number of level sets E_h.



43.3.2. Analytical values

We define three values, area A, perimeter C (Contour length) and Euler number χ, as functions of the level h, with:

$$A(h) = \mathcal{L}_2(D)\rho_0(h) \tag{43.3}$$

$$C(h) = 2\left(\mathcal{L}_1(D)\rho_0(h) + \frac{\pi}{2}\mathcal{L}_2(D)\rho_1(h)\right) \tag{43.4}$$

$$\chi(h) = \mathcal{L}_0(D)\rho_0(h) + \mathcal{L}_1(D)\rho_1(h) + \mathcal{L}_2(D)\rho_2(h) \tag{43.5}$$

with $\mathcal{L}_0(D) = \chi(D) = 1$, $\mathcal{L}_1(D)$ is half the boundary length of the rectangle D and $\mathcal{L}_2(D)$ is its area.

- First, compute the following values:

$$\rho_0(h) = \int_h^\infty \frac{1}{\sqrt{2\pi}} e^{-u^2/2} du \tag{43.6}$$

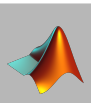
$$\rho_1(h) = \frac{\sqrt{\lambda}}{2\pi} e^{-h^2/2} \tag{43.7}$$

$$\rho_2(h) = \frac{\lambda}{(2\pi)^{\frac{3}{2}}} e^{-h^2/2} h \tag{43.8}$$

$$\lambda = \frac{1}{\sigma^2} \tag{43.9}$$

- Then, compute the analytical values of A, C and χ.
- Compare the analytical values to the empirical ones.

Use `erfc` from `scipy.special`.



43.4. Python correction



43.4.1. White noise

The white noise is generated with the following code.



```
%% white noise simulation
W = np.random.randn (N, N) ;
```

43.4.2. Gaussian Random Field

The Gaussian function that will serve as a covariance function is generated via the given code. Pay attention to the discretization grid: the FFT implies that functions are periodic, and in order to do that, the discretization must be set between $[-N/2 : N/2[$.

The Gaussian Random Field can be generated via the formula already presented (see Fig.43.4).



```
def grf2D(N, sigma):
    # Discrete space
    x = np.arange(-N/2 , N/2) ;
    [ X , Y ] = np.meshgrid( x , x ) ;
    # Covariance function
    C = np.exp(-1/2 * ( (X/ sigma /np.sqrt (2))**2+(Y/sigma/np.sqrt (2) )** 2 ) ) ;
    Cmat = np.fft . fftshift (C) ;
    # real positive part, then square root
    Cf = np.real(np.fft . fft2 ( Cmat ) ) ;
    Cf = np.sqrt ( np.maximum ( np.zeros ( Cf.shape ) , Cf ) ) ;
    # Complex white noise
    W = np.random.randn (N, N) ;

    A = Cf * np.fft . fft2 (W);
    G = np.real ( np.fft . ifft2 (A) ) ;
    return G;
```

Figure 43.4: Gaussian Random Field, with $\sigma = 10$ (pixels) and $N = 2^{10} = 1024$ pixels.

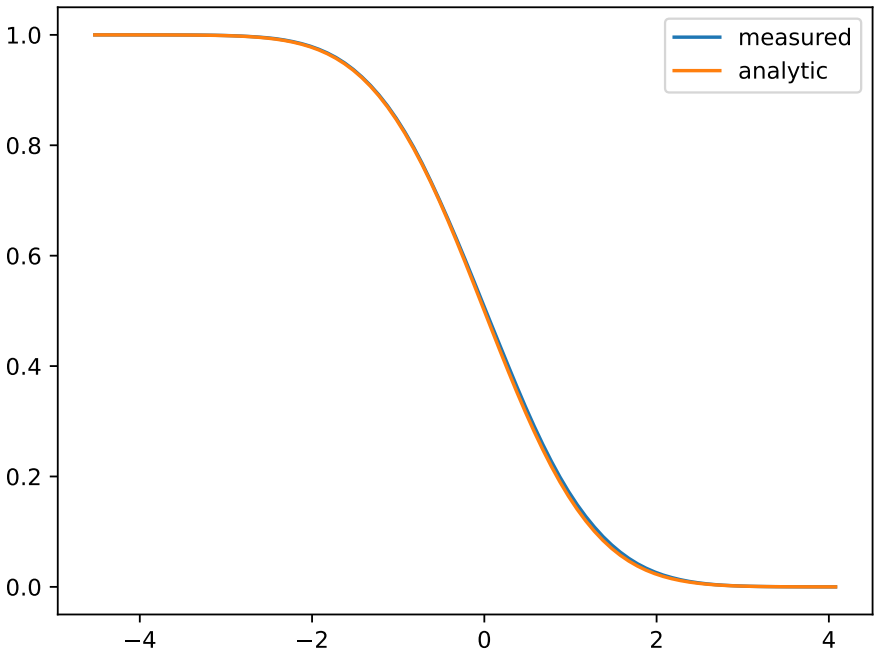


43.4.3. Minkowski functionals

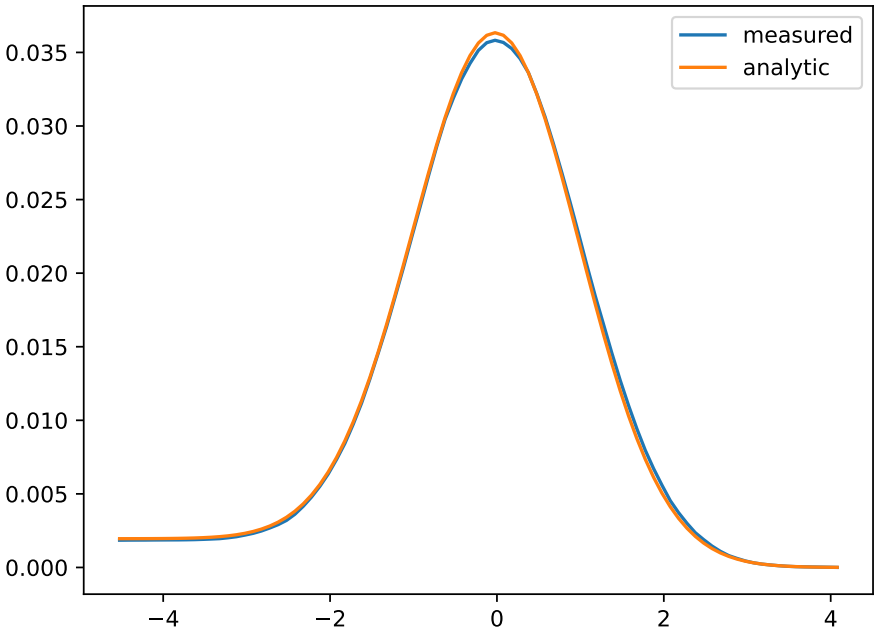
The Minkowski functionals are illustrated in Fig.43.5. The code follows.

Figure 43.5: Illustration of the simulated and analytical values of the Minkowski functionals of the level sets of the Gaussian Random Field, for $\sigma = 10$ and $N = 1024$.

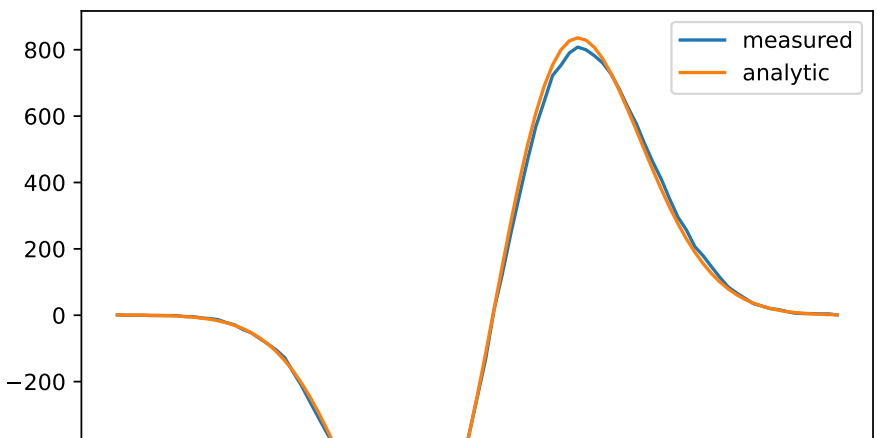
(a) Area.



(b) Perimeter.



(c) Euler number in 4 connectivity.



The measures can be performed with the code from the tutorial 38.



```
def bwminko(X):
    # zero padding of input
    X = np.pad(X, ((1,1), (1,1)), mode='constant');
    # Neighborhood configuration
    F = np.array([[0, 0, 0], [0, 1, 4], [0, 2, 8]]);
    XF = signal.convolve2d(X,F,mode='same');
    edges = np.arange(0, 17 ,1);
    h,edges = np.histogram(XF[:], bins=edges);

    f_intra = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1];
    e_intra = [0,2,1,2,1,2,2,2,0,2,1,2,1,2,2,2];
    v_intra = [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1];
    f_inter = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1];
    e_inter = [0,0,0,1,0,1,0,2,0,0,0,1,0,1,0,2];
    v_inter = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1];
    EulerNb4 = np.sum(h*v_inter - h*e_inter + h*f_inter)
    Area = np.sum(h*f_intra)
    Perimeter = np.sum(-4*h*f_intra + 2*h*e_intra)
    return Area, Perimeter, EulerNb4;
```

Then, the measurements consists in taking all the level-sets and evaluating the properties from these binary sets. Notice that the perimeter is allways a difficult task and is not really precise.



```
def minkoMeasured(G, sigma, hmin, hmax):
    H = np.arange(hmin, hmax, .1);
    A = [];
    P = [];
    E = [];
    bar = progressbar.ProgressBar();
    for h in bar(H):
        levelset = G >= h;
        a, p, e = bwminko(levelset);
        A.append(a);
        P.append(p);
        E.append(e);
    return A, P, E;
```

The analytical values are simply evaluated with the formulas.

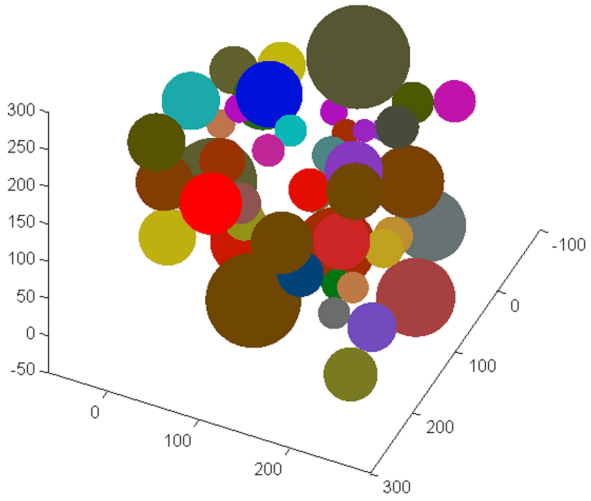


```
def minkoAnalytical(N, sigma, hmin, hmax):
    l = 1/(2*sigma**2);
    # analytical values
    H = np.arange(hmin, hmax, .1);
    rho_0 = 1/2 * erfc(H / np.sqrt(2));
    rho_1 = np.sqrt(l) * np.exp(- H**2 / 2) /(2*np.pi);
    rho_2 = l / (2*np.pi)**(3/2) * np.exp(- H**2 / 2) * H;

    Aa = N**2 * rho_0;
    Pa = 4*N*rho_0 + np.pi*N**2*rho_1;
    Ea = rho_0 + 2*N*rho_1 + N**2*rho_2;
    return Aa, Pa, Ea;
```

★★★ 44 Stereology

This tutorial introduces the problems of stereological measurements, based on simple probes (points, lines...). This tutorial uses the notation that can be found in [36] (among others). The distribution of chord lengths of disks and spheres (Bertrand's paradox) is studied in a separate tutorial.



44.1. Classical measurements of stereology

Let start by some definitions. The stereology is based on some measures, that can be seen as samples, called probes. A probe can be a point, a line, a curve, a plane, a surface... These probes allow us to estimate global geometrical properties through partial measures. Very simple and practical probes are now presented and used. The notation $\langle \cdot \rangle$ denotes an expected count for some normalized value.

Probe name	Notation	Definition
Point count	$\langle P_P \rangle$	Fraction of points in phase
Line intercept count	$\langle P_L \rangle$	Number of intersections per length
Area fraction	$\langle A_A \rangle$	Fraction of area in intersection

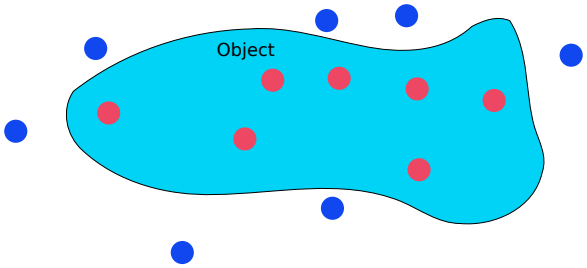
44.1.1. Probes

The measures are performed with the following definitions:

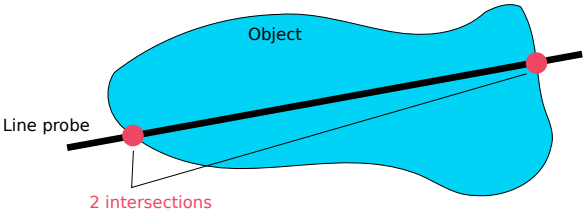
- $\langle P_P \rangle$: the count of the points that lie in the phase, normalized by the total number of points.
- $\langle P_L \rangle$: the count of the number of intersection of lines and the surface of the phase, normalized by the length of the lines **included** in the objects (in m^{-1}, see Fig.44.1). L_A is the ratio between the perimeter of the phase and the total area.
- $\langle A_A \rangle$: in a 3D object, the probes are planes and $\langle A_A \rangle$ is the area of the intersections of the planes and the phase, normalized by the total area of the planes.

Figure 44.1: Probes examples: points and lines.

(a) Evaluation of P_P: count the number of points that lie in the objects, normalized by the total number of points.



(b) Evaluation of P_L: count the number of intersection of some lines with the surface of the phase (object). Then, perform a ratio with the total length of the lines.



- Generate a 2-D binary image containing a population of disks.
- Estimate its area fraction by using points as probe population ($< P_P >= A_A$).
- Estimate its length per area by using lines as probe population ($< P_L >= \frac{2}{\pi} L_A$).
- Load the 3-D image and verify the following relation: $< A_A >= V_V$ with V_V being the volume fraction.

In order to load the volume of the spheres, you can use:

```
import scipy.io
sphere = scipy.io.loadmat("spheres.mat")
```



44.2. Python correction





```
import numpy as np
import matplotlib.pyplot as plt
import skimage.measure
import scipy
```

44.2.1. Classical measurements of stereology

To generate a binary image with overlapping random disks (see Fig.44.2), the following function is used:

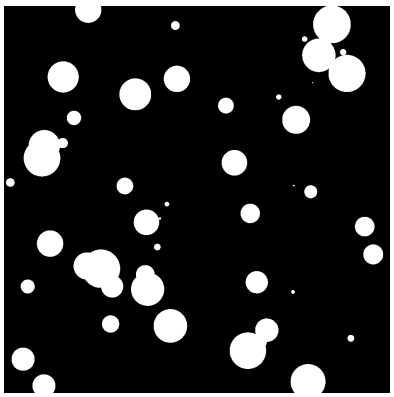


```
def popDisks(nb_disks, S, Rmax):
    """ Generates an image with random disks, with uniform distribution of the centers, and of the radii.
    @param nb_disks: number of disks
    @type S: int
    @param S: Size of spatial support
    @param rmax: maximum radius of disks
    @return: binary image of disks
    """
    centers = np.random.randint(S, size=(nb_disks,2));
    radii = Rmax * np.random.rand(nb_disks);

    N=1000;
    x = np.linspace(0, S, 1000);
    y = np.linspace(0, S, 1000);
    X, Y = np.meshgrid(x, y);
    I = np.zeros((N,N));
    for i in range(nb_disks):
        I2 = (X-centers[i,0])**2 + (Y-centers[i,1])** 2 <= radii[i]**2;
        I = np.logical_or(I, I2);

    return I;
```

Figure 44.2: Random population of disks.



Area fraction

The area fraction counts the number of probes that lay inside the objects. In order to verify this probe, the next function evaluates the real area covered by the disks.

```python
def areaFraction(nb_probes, I):
    """
    Evaluates area fraction via point probes
    @param nb_probes: number of probes
    @param I: binary image, square
    """
    P = np.random.randint(I.shape[0], size = (nb_probes,2));

    # count the number of probes in phase
    count = np.sum(I[P[:,0], P[:,1]]) ;
    return float(count) / nb_probes;
```

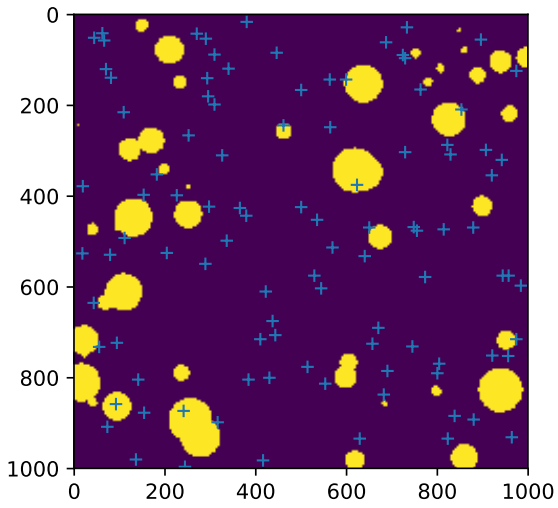
```python
def verifyAreaFraction():
    """
    Verify area fraction
    """
    I = popDisks(50, 1000, 20);
    plt.imshow(I);
    AA = float(np.sum(I)) / np.size(I);
    PP = areaFraction(3000, I);

    print("AA (true number of pixels): {:.2%} ".format(AA));
    print("PP (evaluated fraction) : {:.2%} ".format(PP));
```

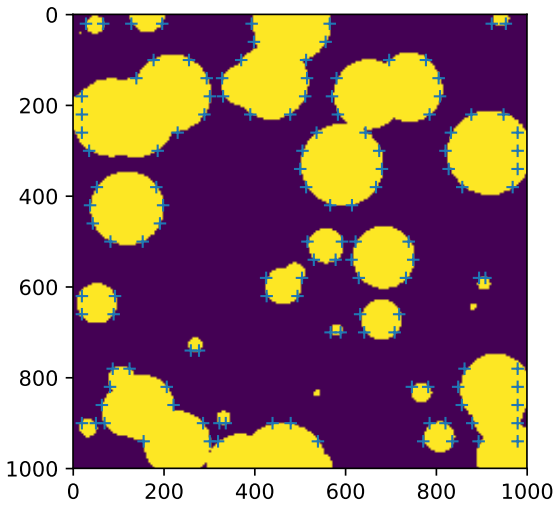
There is a good agreement between AA and PP. The method is illustrated in Fig.44.3a.

Figure 44.3: Illustration of the positions of the different probes.

(a) Area fraction evaluation.



(b) Length per area evaluation.




```
AA (true number of pixels): 2.05%
PP (evaluated fraction) : 1.87%
```

Length per area

A certain number of segments are used in order to perform the probing. The number of times a segment goes through the surface of the object is evaluated. This is illustrated in Fig.44.3b.



```
def lengthPerArea(I):
    """
    Evaluates the length per area
    """
    perim= skimage.measure.perimeter(I.astype(int), 8);
    LA = perim / np.sum(I);
    print("LA (true count): {:.2%} ".format(LA));

    # lines probes, every 10 pixels
    probe = np.zeros(I.shape);
    probe[20:-20:10, 20:-20] = 1;
    lines = I.astype(int) * probe;

    # count number of intercepts
    h = np.array([[1, -1, 0]]);
    points = scipy.signal.convolve2d(lines, h, mode='same');

    nb_lines = np.sum(lines);
    nb_points= np.sum(np.abs(points));
    PL = float(nb_points) / nb_lines;
    print("pi/2*PL (true count): {:.2%} ".format(np.pi/2*PL));
```



```
def verifyLengthPerArea():
    I = popDisks(50, 1000, 20);
    plt.imshow(I);
    plt.show();
    lengthPerArea(I);
```

The perimeter evaluation may vary a lot according to the implementations or to the connectivity chosen.

```
LA (true count): 18.02%
pi/2*PL (evaluated fraction): 15.20%
```

Volume fraction

This example loads a MATLAB® matrix.

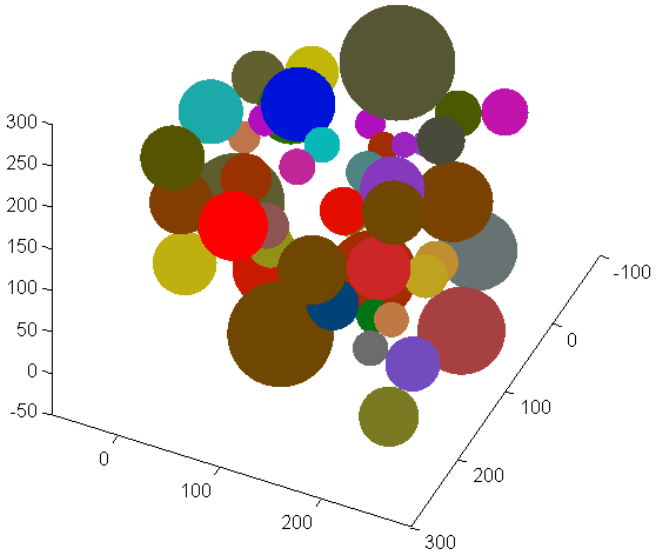
```
def volumeFraction(volume):
    """
    """
    VV = float(np.sum(volume)) / np.size(volume);
    probe= np.zeros(volume.shape);
    probe[10:-10:50, 10:, 10:] = 1;

    s = np.sum(probe);
    probe = probe * volume;
    AA = float(np.sum(probe)) / s;

    print("VV (true count): {:.2%}".format(VV));
    print("AA (true count): {:.2%}".format(AA));

def verifyVolumeFraction():
    sphere = scipy.io.loadmat("spheres.mat");
    volumeFraction(sphere['A']);
```

Figure 44.4: Random population of 3D overlapping spheres. This function uses povray and the vapory python API.



As for the area fraction, the volume fraction is precise. The results on this example are:

```
VV (true count): 5.02%
AA (evaluated fraction): 5.28%
```

★★★ 45 Bertrand's paradox: random chords and sections

The Bertrand's paradox is explored in the case of the analysis of the distribution of chord lengths of disks and spheres. The answer to "choose a chord at random" depends on how the drawing is done: the question is ill-posed until the random procedure is specified.

45.1. Random chords of a disk, and Bertrand's paradox

The goal of this exercise is to simulate the distribution of random chord lengths on a disk. In the field of process engineering, optical particle sizers provide chord length distributions of objects that are considered as spheres. These developments are not only theoretical, they have a practical use in laboratories or in industrial reactors; for example, Focused beam reflectance measurement (FBRM) evaluates chord length distribution of a population of crystals in a reactor.

45.1.1. Bertrand's paradox

From Wikipedia[1], the Bertrand paradox goes as follows: Consider an equilateral triangle inscribed in a circle. Suppose a chord of the circle is chosen at random. What is the probability that the chord is longer than a side of the triangle?

Bertrand provided three different methods in order to evaluate this probability. These gave three different values.

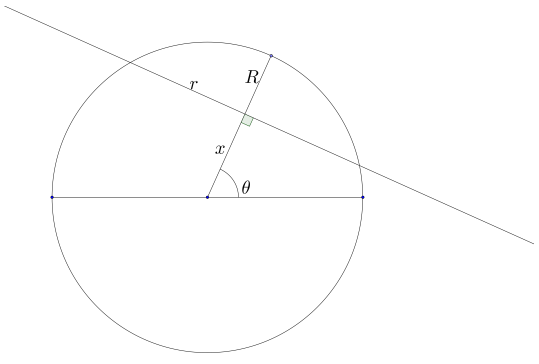
The objective is to evaluate the distribution of the chord lengths for two of these methods. The bertrand's paradox lays on the fact that the question is ill-posed, and the choice "at random" must be clarified. The reader will find more details in the different citations.

45.1.2. Random radius

The intersection of a random line with a disk of radius R is a segment whose half length r is linked to the distance x between the segment and the center of the disk (Eq. 45.1 and Fig. 45.1).

$$r = \sqrt{R^2 - x^2} \tag{45.1}$$

Figure 45.1: Relation between the radius of the chord and the distance to the center of the disk.



As presented in Fig. 45.1, θ is randomly chosen in $[0; 2\pi[$ (uniform law), and x is randomly chosen in $[0; R]$ (uniform law).

[1] https://en.wikipedia.org/wiki/Bertrand_paradox_(probability)

?

Repeat the simulation of r by the so-called random radius method a large number of times ($N = 1e7$), and compute the probability density function of the distribution.

This method is in agreement with the analytical results.

45.1.3. Random endpoints

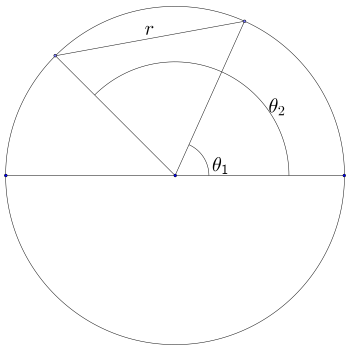
In this second case, two random angles θ_1 and θ_2 are randomly chosen (uniform law in $[0; 2\pi]$). This defines two points and thus a chord (see Fig 45.2).

?

Make the simulation for the "random endpoints" and compare it to the analytical values and to the first simulation. Notice that this simulation gives different values (search for the Bertrand paradox for more details).

This method is NOT in agreement with the analytical results.

Figure 45.2: Second method for simulating a random chord of the disk.



45.1.4. Analytical values

The probability to obtain a chord of half length x between a and b is given by Eq. 45.2 (see [25]):

$$\mathbb{P}(x \in [a; b]) = \int_a^b \frac{\rho}{R\sqrt{R^2 - \rho^2}} \mathrm{d}\rho \tag{45.2}$$

?

By discretizing the interval $[0; R]$, compute the analytical value of the probability density function of the distribution of the radii with the Eq. 45.2.

45.2. Random sections of a sphere and a plane

The two previous strategies can be applied on the sphere. What we are looking for are radii of the intersection of the sphere with a random plane.

45.2.1. First simulation: random radius

To find and random plane $\mathcal{P}$ intersecting a sphere $\mathcal{S}$ of radius R, one have to choose a direction $\vec{u}$ (i.e. a point on the sphere), find a point P between the latter and the center O of the sphere, and considere the plane $\mathcal{P}$ that is orthogonal to the direction $\vec{u}$ and passing at this point P (Fig. 45.3).

?

Code the following method:

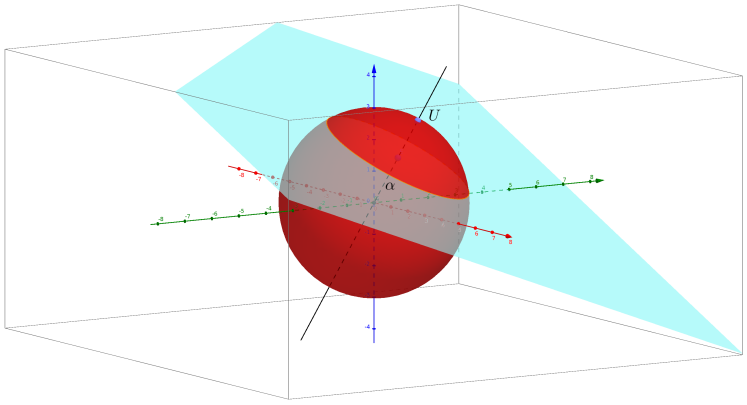
- A random point on the sphere is chosen with the following method (see [24]): let x, y and z be 3 Gaussian random variables, the point on the sphere is defined by the normed vector $\vec{u}$, such that:

$$\vec{u} = \frac{1}{\sqrt{x^2+y^2+z^2}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{45.3}$$

- The point P is chosen as $\vec{OP} = \alpha\vec{u}$, with O being the center of the sphere, and α being a uniform random variable in $[0; R]$. The plane $\mathcal{P}$ is orthogonal to $\vec{u}$, passing at P (see Fig. 45.3).

Notice that this simulation is equivalent to the first presented case (random radius) of the random chords of a disk, and that the choice of the random point on the sphere is useless.

Figure 45.3: First simulation method of a random intersection of a sphere and a plane.



The method that consists to choose a point by two angles does not provide a uniform distribution of the points on the sphere.

45.2.2. Second simulation: 3 random endpoints

The random plane $\mathcal{P}$ is defined by 3 random points laying on the sphere (see Eq. 45.3).

?

- Analytically, find the distance between the center of the sphere O and the plane $\mathcal{P}$.
- Simulate a high number of intersections and find numerically the distribution of the radii of these intersection disks.

45.2.3. Third simulation: 2 random endpoints

Choose 2 random points laying on the sphere (see Eq. 45.3) and evaluate their half distance.

?

Simulate a high number of couple of points and evaluate the distribution of their half distances.

The distribution of the length of the chords on a sphere is linear, and is different from the distribution of the radii resulting from the intersection of a random plane and the sphere.

45.2.4. Comparison

?

Compare the 3 results and comment.



45.3. Python correction



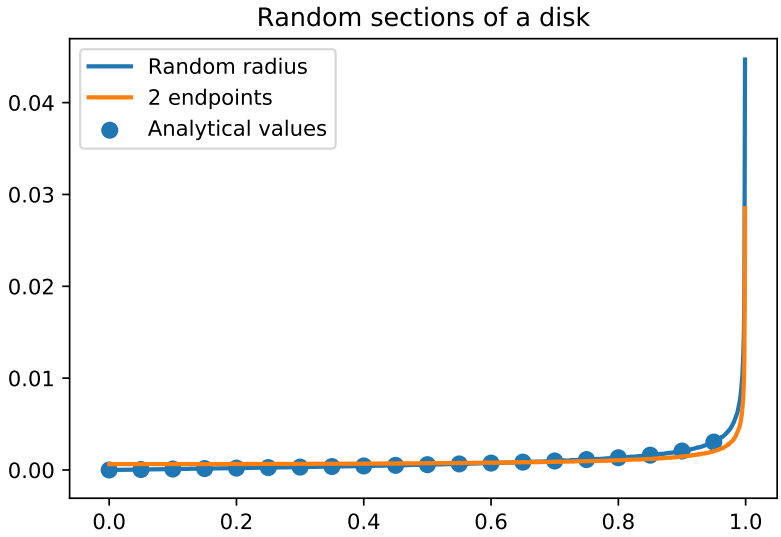


```
import numpy as np
import matplotlib.pyplot as plt
```

45.3.1. Random sections of a disk

The value N is the number of simulations performed. To find the probability, N values x are randomly chosen between 0 and R. The formula $r = \sqrt{R^2 - x^2}$ yields to the half length r of the chord. After discretizing the interval $[0; R]$ in $nBins$, the number of values x in each bin is counted (with python np.histogram function). The results are presented in Fig. 45.4.

Figure 45.4: Simulations of random chords of a disk.





```
import numpy as np
import matplotlib.pyplot as plt
N = 10000000;
nBins = 1000;
R = 1.;
# first simulation method: random radius
d = R * np.random.rand(N);
radii = np.sqrt(R**2 - d**2);
hi, edges = np.histogram(radii, bins=nBins);
plt.plot(edges[:-1], hi/N, linewidth=2);
```

The second method consists in choosing 2 random points on the circle, given by two random angles.

```
# 2nd method: random points on the circle
# from 2 random angles
theta = np.pi*2*np.random.rand(int(N),2);
dX = np.diff(R * np.cos(theta))
dY = np.diff(R * np.sin(theta));
radii = 1./2 * np.sqrt(dX**2 + dY**2);
hi, edges = np.histogram(radii, bins=nBins);
plt.plot(edges[:-1], hi/N, linewidth=2);
```

Finally, the analytical results are computed for comparison. In order to get the probability density function, you have to perform the approximation of the integral (here, by the so-called rectangle method):

$$\int_a^b f(t)\mathrm{d}t \approx (b-a)\cdot f(a)$$

with

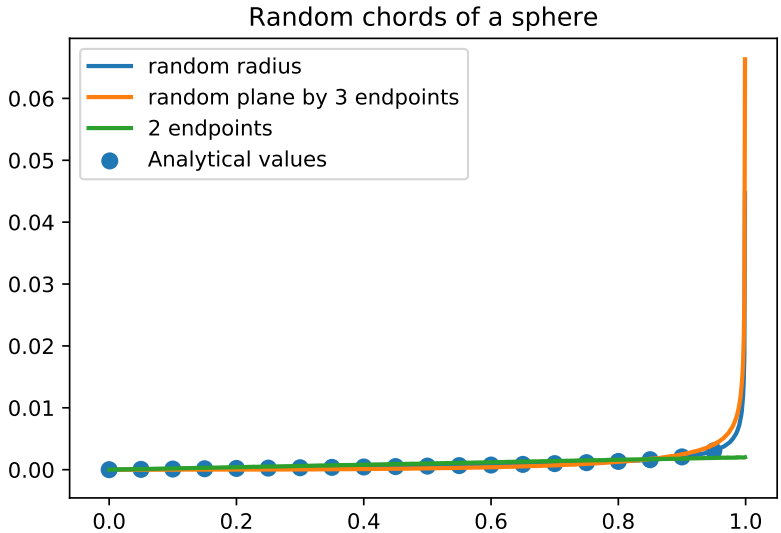
$$f(\rho) = \frac{\rho}{R\sqrt{R^2-\rho^2}}$$

In this case, $(b-a) = \frac{R}{nBins}$.

```
# analytical values
step = .05;
r2 = np.arange(0, R, step);
probaReal = 1./R * r2 / np.sqrt(R**2-r2**2);
probaReal = probaReal * R / nBins; # approximation of the integral
plt.scatter(r2, probaReal, 50);
# display results
plt.legend(["Unique random points", "2 random points", "Analytical values"])
plt.show()
plt.savefig('sections_disk.pdf')
```

45.3.2. Random planar sections of a sphere

Figure 45.5: Simulations of random chords of a sphere.





```
import numpy as np
import matplotlib.pyplot as plt

def generatePointsOnSphere(nb_points, R):
    """
    Generate points on a sphere
    @param nb_points: number of points
    @return n: array of size nb_points x 3
    """
    n = np.random.randn(nb_points, 3);
    mynorm = np.linalg.norm(n, axis=1);
    n = R * n / np.transpose(np.matlib.repmat(mynorm, 3, 1));

    return n

def dot(A, B, ax=1):
    """ dot product for arrays
    """
    return np.sum(A.conj()*B, axis=ax );
```

First simulation: random radius

This method is equivalent to the first case of the disk chord. The 3D property of the sphere is not used and thus the code is strictly equivalent to the 2D case.



```
# initial values
N = 1e7;        # number of samples, float number
R = 1;          # radius of the sphere
nBins = 1000; # number of bins for histogram computation
# first simulation method: random radius
d = R * np.random.rand(int(N));
radii = np.sqrt(R**2 - d**2);
probaSimu = np.histogram( radii , bins=nBins);
plt.plot(probaSimu[1][:-1], probaSimu[0]/N, linewidth=2);
```

Second simulation: 3 endpoints

Define a random plane from 3 points randomly chosen on the sphere. Let n_1, n_2 and n_3 be 3 points on the sphere. These points define the plane $\mathcal{P}$. The distance between the center of the sphere O and the plane $\mathcal{P}$ is given by the relation:

$$d(0, \mathcal{P}) = \frac{|\vec{n} \cdot \vec{u}|}{||\vec{n}||}$$

with $\vec{u} = \vec{n_2} - \vec{n_1}$, $\vec{v} = \vec{n_3} - \vec{n_1}$, and $\vec{n} = \vec{u} \wedge \vec{v}$ the normal vector to the plane. The results are presented in Fig. 45.5.

This is a case of Bertrand's paradox: the definition of randomness is not good in the present case.



```
# second simulation
# choose 3 points to define a plane,
# then, compute the distance from the origin to this plane
n1 = generatePointsOnSphere(N, R);
n2 = generatePointsOnSphere(N, R);
n3 = generatePointsOnSphere(N, R);
# u and v belong to the plane
u=n2-n1;
v=n3-n1;
# n: normal vector to the plane
n=np.cross(u,v);
x = dot(n, n1) / np.linalg.norm(n, axis=1);
# distance from the origin to the plane:
r = np.sqrt(R**2 - x**2);
probaSimu = np.histogram(r, bins=nBins);
plt.plot(probaSimu[1][:-1], probaSimu[0]/N, linewidth=2);
```

3rd simulation: 2 endpoints

This situation presents the random choice of two points on the sphere, and the computation of their distance. This produces a linear probability (see Fig.45.5).



```
# 3rd case:
# 2 points on the sphere and distance between them
n1 = generatePointsOnSphere(N);
n2 = generatePointsOnSphere(N);
r = 1./2 * np.linalg.norm(n1-n2, axis=1);
probaSimu = np.histogram(r, bins=nBins);
plt.plot(probaSimu[1][:-1], probaSimu[0]/N, linewidth=2);
```

Analytical values

This code evaluates analytical values.



```
# analytical values
step = .05;
r2 = np.arange(0, R, step);
probaReal = 1./R * r2 / np.sqrt(R**2-r2**2);
probaReal = probaReal * (R / nBins); # approximation of the integral
plt.scatter(r2, probaReal, 50);

# display
plt.legend(["random plane by 3 points on the sphere", "2 points on the sphere", "Analytical values"])
plt.show();
plt.savefig("section_sphere.pdf") # save as pdf file
```

The results are displayed in Fig. 45.5.

The Bertrand's paradox is illustrated by the fact that "at random" can provide several different interpretations. The objective here is to focus on the computational choices that can be made in order to provide random chords or random points on a sphere.

46 Fractal dimension

The fractal dimension measures how the border of a set fills the plane when it is observed at finer and finer scales. This tutorial computes it by the box-counting method, and checks the result on the Sierpinski triangle, whose theoretical dimension is known.

46.1. Fractal dimension

In fractal geometry, the Minkowski–Bouligand dimension, also known as Minkowski dimension or box-counting dimension, is a way of determining the fractal dimension of a set S in a Euclidean space $\mathbb{R}^n$.
Suppose that $N(\epsilon)$ is the number of boxes of side length ϵ required to cover the **border of the set**. Then the box-counting dimension is defined as:

$$\dim_{\text{box}}(S) := \lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log(1/\varepsilon)}.$$

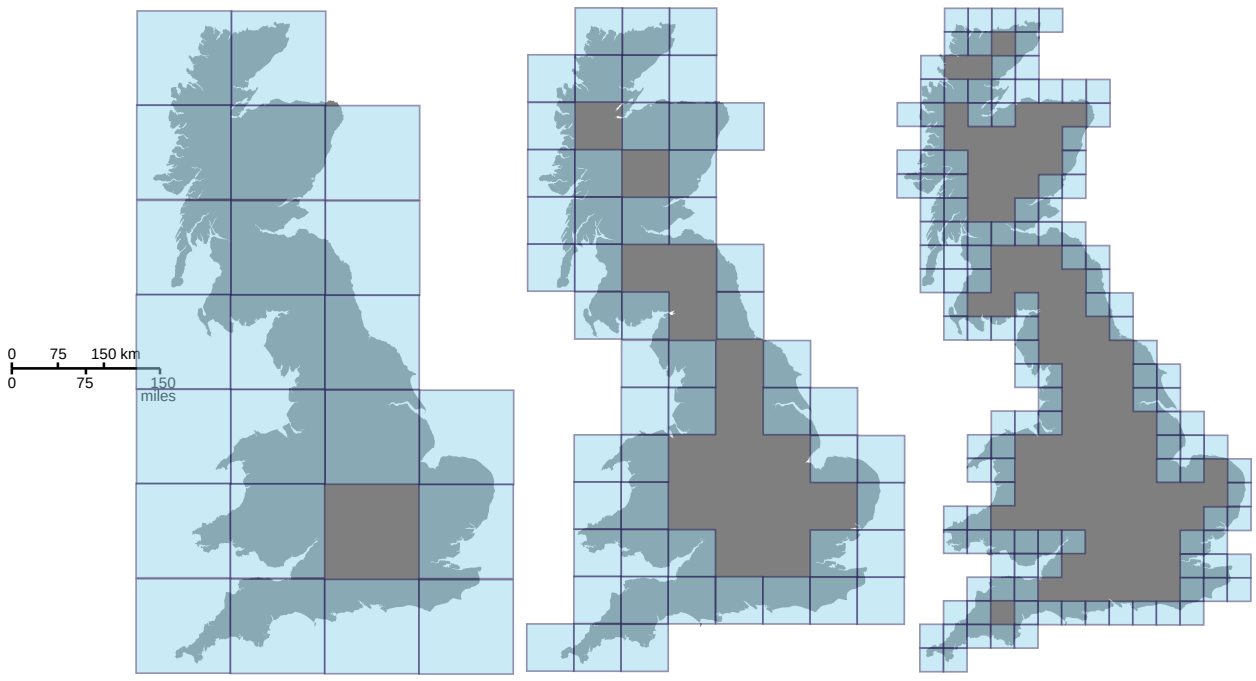


Figure 46.1: Grid decomposition for box counting.

46.1.1. Boxcount method

For a 2D binary array representing the set S, code a function boxcount that will return the sum of all values of the grid area (square).

```
def boxcount(A, epsilon):
    ### code here
    return B
```

For example, if A is an array ones 16×16 ones,

$$A = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{pmatrix}$$

$$\text{boxcount}(A, 6) = \begin{pmatrix} 36 & 36 & 24 \\ 36 & 36 & 24 \\ 24 & 24 & 16 \end{pmatrix}$$

$$\text{boxcount}(A, 5) = \begin{pmatrix} 25 & 25 & 25 & 5 \\ 25 & 25 & 25 & 5 \\ 25 & 25 & 25 & 5 \\ 5 & 5 & 5 & 1 \end{pmatrix}$$

46.1.2. Fractal dimension

Code a function fractal_dimension that takes a binary image as input and returns the fractal dimension. In order to do that:

- code a function $N(B, epsilon)$ that gives the number of grid areas that touch the border of B (see Fig.46.1).
- evaluate this number for different scales ϵ (these must be taken as powers of 2).
- represent a in plot the values $\log(N)$ as a function of the scales $\log(\epsilon)$ (see Fig.46.2).
- find an approximation of the fractal dimension.

46.1.3. Sierpinski triangle and numerical application

The Sierpinski triangle Fig.46.3 is a famous fractal figure.

- Give the theoretical value of its Haussdorf dimension (fractal dimension)
- Compute the fractal dimension with your code. As a hint, you should find a value around 1.6.

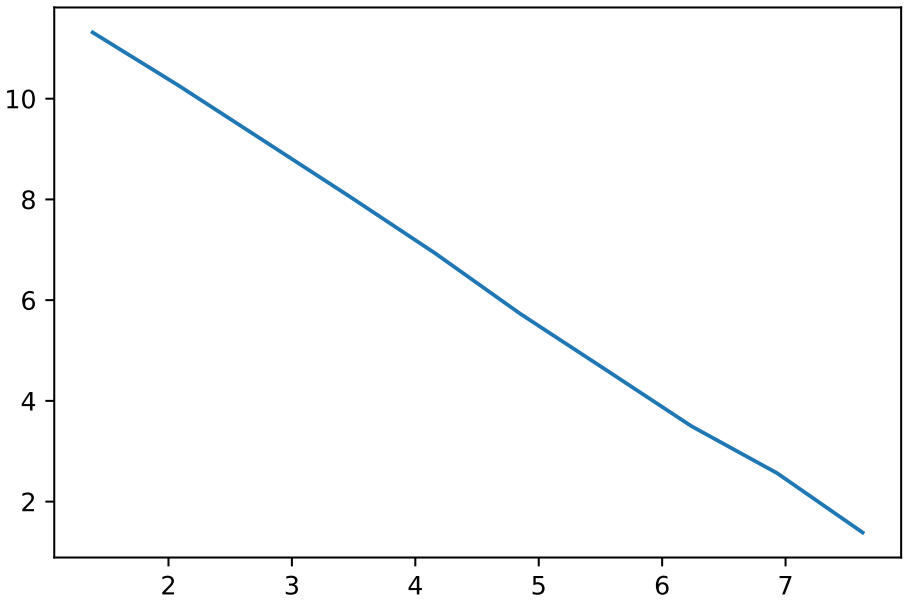


Figure 46.2: $\log(N)$ as a function of the scales $\log(\epsilon)$

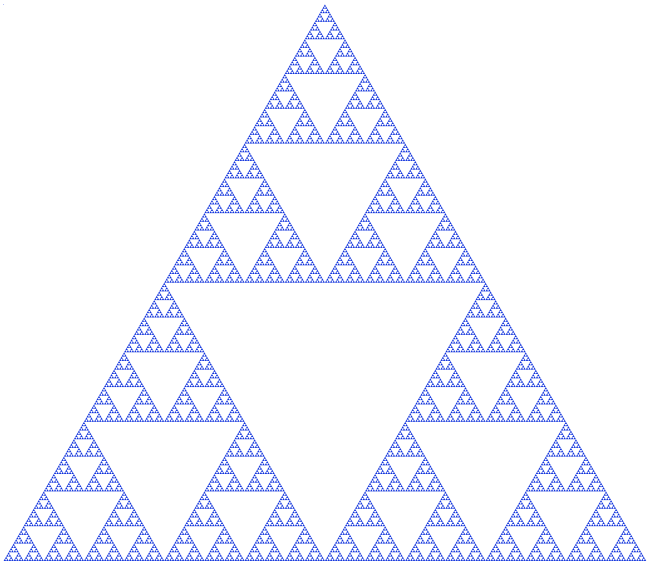


Figure 46.3: Sierpinski triangle.



46.2. Python correction





```
import numpy as np
import matplotlib.pyplot as plt
import skimage
from skimage import io
```

46.2.1. Boxcount method

The sum of the values over each box of side k is computed without any loop, by two successive calls to np.add.reduceat, along the rows then along the columns. A box touches the border of the set when it is neither empty (sum 0) nor full (sum k^2).

46.2.2. Fractal dimension

The box sizes are the powers of 2, from the greatest one that fits in the image down to 4. The fractal dimension is the opposite of the slope of $\log N(\epsilon)$ as a function of $\log \epsilon$, obtained by a linear fit (np. polyfit).



```
def fractal_dimension(Z, threshold=0.9):

    # Only for 2d image
    assert(len(Z.shape) == 2)

    # From https://github.com/rougier/numpy-100 (#87)
    def boxcount(Z, k):
        # plt.imshow(Z)
        # plt.title("Z")
        # plt.show()
        S = np.add.reduceat(
            np.add.reduceat(Z, np.arange(0, Z.shape[0], k), axis=0),
                                np.arange(0, Z.shape[1], k), axis=1)

        # plt.imshow(S)
        # plt.title(f"S {k}")
        # plt.show()
        # plt.imshow(np.logical_and(S>0, S < k*k))
        # plt.title(f"{k}")
        # plt.show()
        # We count non-empty (0) and non-full boxes (k*k)
        #print(len(np.where((S > 0) & (S < k*k))[0]), np.sum(S>0))
        #return np.sum(S>0) # len(np.where((S > 0) & (S < k*k))[0])
        return len(np.where((S > 0) & (S < k*k))[0])


    # Transform Z into a binary array
    Z = (Z < threshold)

    # Minimal dimension of image
    p = min(Z.shape)

    # Greatest power of 2 less than or equal to p
    n = 2**np.floor(np.log(p)/np.log(2))

    # Extract the exponent
    n = int(np.log(n)/np.log(2))

    # Build successive box sizes (from 2**n down to 2**1)
    sizes = 2**np.arange(n, 1, -1)
    print(sizes)

    # Actual box counting with decreasing size
    counts = []
    for size in sizes:
        counts.append(boxcount(Z, size))
    print(counts)
    # Fit the successive log(sizes) with log (counts)
    coeffs = np.polyfit(np.log(sizes), np.log(counts), 1)

    plt.plot(np.log(sizes), np.log(counts))
    return -coeffs[0]
```

46.2.3. Sierpinski triangle and numerical application

The Sierpinski triangle is made of 3 copies of itself, reduced by a factor 2: its Hausdorff dimension is $\log 3/\log 2 \approx 1.585$.

```
import matplotlib.pyplot as plt
import skimage
from skimage import io
I = io.imread(" sierpinski .png")

I = skimage.img_as_float(I [:,:,1]) >.5


plt.imshow(I)
plt.show()
print("Minkowski–Bouligand dimension (computed): ", fractal_dimension(I))
print("Haussdorf dimension ( theoretical ): ", (np.log(3)/np.log(2)))
```

```
[2048 1024 512 256 128 64 32 16 8 4]
[4, 13, 33, 101, 307, 1026, 3161, 9538, 28602, 82008]
Minkowski-Bouligand dimension (computed): 1.600939276174402
Haussdorf dimension ( theoretical ): 1.5849625007211563
```

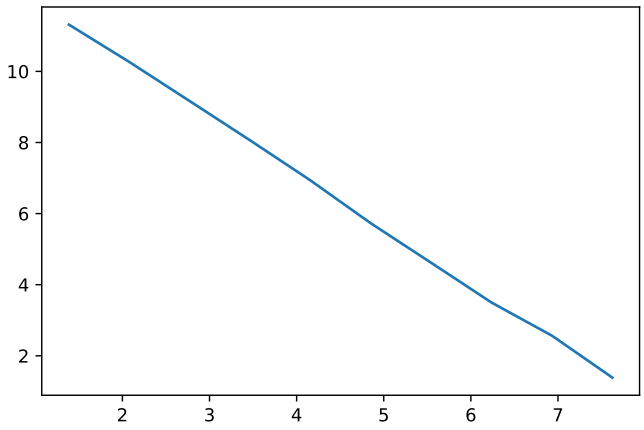


Figure 46.4: $\log(N)$ as a function of $\log(\epsilon)$ for the Sierpinski triangle (script boxcounting.py).

The script vonkoch.py draws the von Koch snowflake (with the turtle module), another classical fractal, of dimension $\log 4/\log 3 \approx 1.262$.

Part XI Exams

47 Practical exam 2016

All printed documents allowed. Send your code via the campus website (zip files if several files are to be sent).

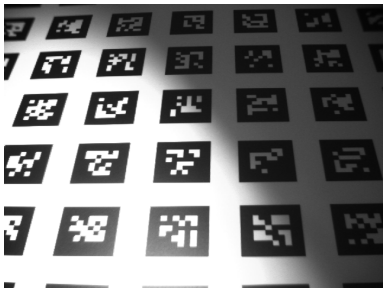
47.1. Segmentation (13 points)

?

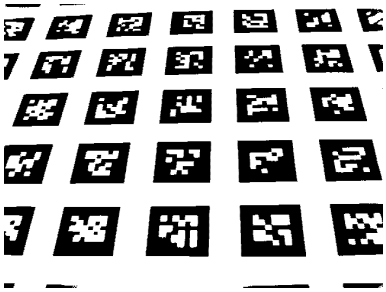
- Segment the image.

Figure 47.1: Test image

(a) Image to be segmented, from [4].



(b) Segmented image



47.2. Integral image (7 points)

If the original image is denoted f, the integral image I is defined by the following equation, for all pixels of coordinates (x, y):

$$I(x,y) = f(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1)$$

?

- Code a function that computes this integral image. The prototype of this function must be `function II=int_image(I)`.
- Code a function that computes the local average of the image f. Notice that:

$$\sum_{x=x_1}^{x_2}\sum_{y=y_1}^{y_2} f(x,y) = I(x_2,y_2) - I(x_2,y_1-1) - I(x_1-1,y_2) + I(x_1-1,y_1-1)$$

- Compare it to a mean filter.

48 Theoretical exam 2016

48.1. Basic notions

48.1.1. Image characterisation

?

- Cite the names of the major image file formats and their main differences. What is DICOM?
- Define sampling in numerical images. Define the image resolution.

48.1.2. Sensors

?

- What is a Bayer filter?
- Explain the principles of demosaicing.

48.1.3. Image processing

?

- Define the operation of histogram equalization.
- What is the difference with histogram stretching?

?

- From the mathematical definition of the derivative, explain the construction of the gradient operator, and then of the Laplacian.
- Cite some method for contours detection, and list their pros and cons.

48.2. Open question

?

The Fig. 48.1 shows a human retina (eye fundus). Propose a method for segmenting:

- the vessels,
- the optical nerve (bright disk).

Justify your choices. Indicate all elements that seem important and the way to deal with them. Look carefully at the shapes and intensities of objects you need to segment.

For your record, ophthalmologists need to measure the diameter and the circularity of the optical nerve, as well as the length and the number of vessels (branches, tortuosity...).

Figure 48.1: Human retina.

(a) Color image.



(b) Red channel.



(c) Green channel.



(d) Blue channel.



49 Theoretical exam 2017

49.1. Warm-up questions

49.1.1. Thresholding methods

?

Cite different methods in order to find a threshold value to binarize a grayscale image. Explain the Otsu's method. What are its limits?

49.1.2. Image filtering

?

- What is the difference between a rank filter and a linear filter?
- Cite the name of a filter of each type.
- In case of a salt and pepper noise present in the image, what type of filter would you require to restore it? Why?
- For the latter, precise its limits and propose a way to improve the result.

49.1.3. Retina images

?

As a project, you coded a method for retina vessels segmentation. Explain the principles of the algorithm.

49.2. Open question: QR code

A QR code is a binary code represented as a 2D matrix. A scanner is in charge of reading it (via a camera) and translating it into the actual code. The structure of a QR code is represented in Fig.49.1.

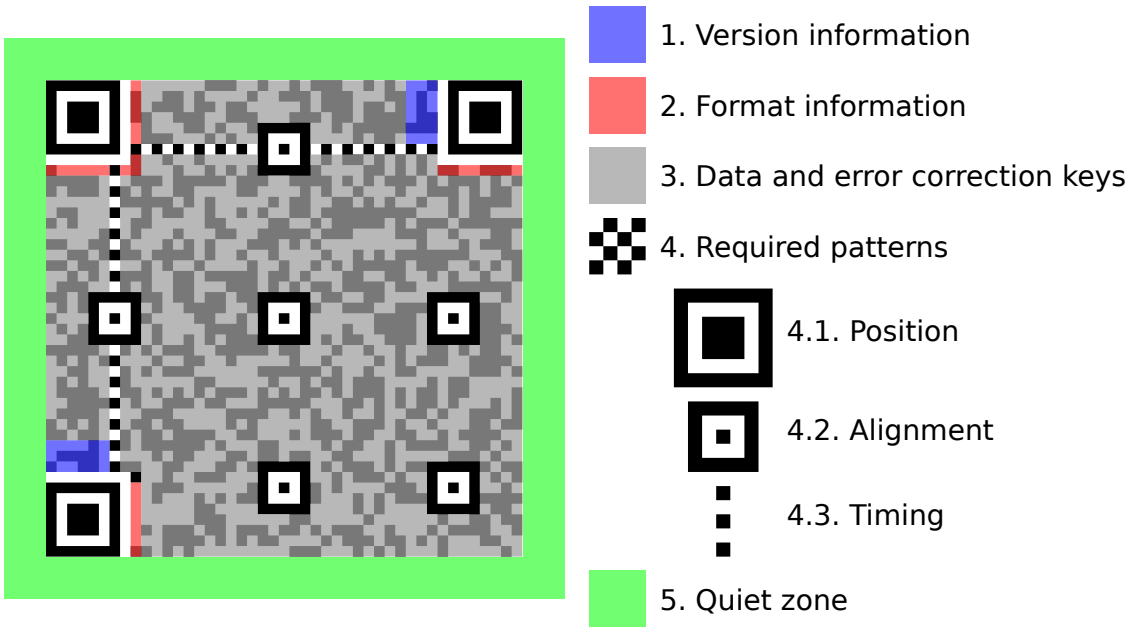
?

Describe the different steps required to perform the acquisition and geometric transformation of the QRcode into a binary 2D matrix representing the code (1 value for 1 square, see Fig.49.1).

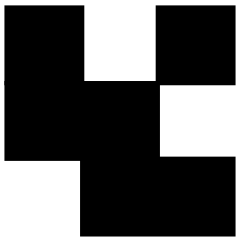
- List different situations that may complicate the image acquisition and analysis.
- Propose some (image processing) solutions in order to deal with these situations. Explain them, give the necessary details describing the methods.

Figure 49.1: Conversion of a binary pattern into a binary matrix.

(a) Structure of a QR code, Wikipedia, author: Bobmath, CC-By-SA



(b) Code.



(c) Matrix.

1	0	1
1	1	0
0	1	1

Figure 49.2: Different QR codes. The image acquisition and analysis must be tolerant to different observation conditions... Images from Collectif Raspouteam http://raspouteam.org/. Other images from unknown sources.



Index

Aliasing, 19, 31

Boxcount method, 405

Characterization, 295, 327, 349, 355
  Circularity, 350
  Convexity, 350
  Diameter, 350
  Elliptic Fourier Descriptors, 259
  Perimeter, 349
Classification, 266, 296
Color
  Chromaticity Diagram, 75
  Chromaticity diagram, 157
  CIELAB, 75
  Color Matching Functions, 76, 156
  Color Spaces, 75, 155
  HSV, 76
  MacAdam Ellipses, 77
Compression
  JPEG, 43
  Quantization, 44
Computational Geometry
  Convex Hull, 305, 350
  Delaunay Triangulation, 311
  Minimum Spanning Tree, 311
  Voronoi Diagram, 311

Discrete Cosine Transform, 43
  Definition, 43

Features, 295
  Detection, 275
  Harris Corner Detector, 253
  Local Binary Patterns, 265
Filtering
  Choquet, 164
  Convolution, 17, 27
  Degenerate Diffusion, 109
  High-pass, 17, 27
  Linear Diffusion, 107
  Low-pass, 17, 25
  Non Linear Diffusion, 108
  Partial Differential Equations, 107
Fourier Transform, 53, 115, 259
  2D FFT, 35
  Application, 36
  Filtering, 36
  Inverse, 35, 117
  Phase Correlation, 285
Fractal dimension, 405
Frameworks
  Color Logarithmic Image Processing, 155
  General Adaptive Neighborhood Image Processing, 163
  Logarithmic Image Processing, 147
Freeman Chain Code, 327

Geometry
  Crofton Perimeter, 345
  Integral Geometry, 343
Gradient
  Prewitt, 18, 27
  Sobel, 18, 27

Heat Equation, 107
Histogram, 16, 23
  Definition, 88
  Equalization, 88
  Matching, 88

Image
  Display, 15, 20
  Load, 15, 20
  Save, 15
Image Registration, 285

Laplacian, 17, 27

Mapping, 25
Mathematical Morphology, 175, 239
  Alternate Sequential Filters, 185
  Attribute Filters, 191
  Dilation, 175
  Erosion, 175
  General Adaptive Neighborhood, 165
  Geodesic Filtering, 185
  Granulometry, 201
  Hit-or-miss Transform, 195
  Reconstruction, 176, 186
  Skeleton, 195
  Topological Skeleton, 195
  Watershed, 225
Multiscale, 67

Filtering, 68
Kramer & Bruckner, 68
Pyramid, 67

Noise
Exponential, 95
Gaussian, 95
Salt & Pepper, 95
Uniform, 95

Point Spread Function, 115
Probability density function, 16

Quantization, 16, 22

Random Fields, 385
Registration, 273
Resizing, 21
Restoration
Adaptive median filter, 96
Blind deconvolution, 119
Deconvolution, 115
Lucy-Richardson filter, 119
Median filter, 96
Van Cittert filter, 118
Wiener filter, 118

Segmentation, 239
Active Contours, 231
Distance Map, 225
Histogram, 209
Hough Transform, 247
K-means, 209, 266, 356
Otsu thresholding, 210
Region Growing, 219
Threshold, 209
Watershed, 225
Watershed by Markers, 226
Shape From Focus, 131
SML, 132
Tenengrade, 133
Variance, 132
Variance of Tenengrad, 133
Spatial Processes, 365
Boolean Models, 377
Characterization, 368, 377
Gibbs Point Process, 366
Marked Process, 369
Miles Formula, 378
Neyman-Scott Point Process, 365
Poisson Point Process, 365
Ripley Functions, 368
Stereology, 391, 397
Random Chords, 397
Random Planes, 398

Tomography, 139
Backprojection, 140
Filtered Backprojection, 140
Topology
Classification, 338
Neighborhood, 327, 337, 343

Wavelets
Definition, 53

Bibliography

[1] http://vision.lems.brown.edu/content/available-software-and-databases. 295, 349, 352, 355

[2] R. J. Adler. *The Geometry of Random Fields*. Wiley, Chichester, UK, June 1981. 385

[3] O. S. Ahmad. *Stochastic representation and analysis of rough surface topography by random fields and integral geometry – Application to the UHMWPE cup involved in total hip arthroplasty*. PhD thesis, École Nationale Supérieure des Mines de Saint-Etienne, 2013. 385

[4] D. Bradley and G. Roth. Adaptive thresholding using the integral image. *Journal of graphics tools*, 12(2):13–21, 2007. 413

[5] F. Catté, P.-L. Lions, J.-M. Morel, and T. Coll. Image Selective Smoothing and Edge Detection by Nonlinear Diffusion. *SIAM Journal on Numerical Analysis*, 29(1):182–193, 1992. 108

[6] M. Couprie and G. Bertrand. Discrete topological transformations for image processing. In *Digital Geometry Algorithms*, pages 73–107. Springer, 2012. 337

[7] M. Fernandes, Y. Gavet, and J.-C. Pinoli. Improving focus measurements using logarithmic image processing. *Journal of Microscopy*, 242(3):228–241, 2010. 131, 133

[8] M. Fernandes, Y. Gavet, and J.-C. Pinoli. Robust 3-D reconstruction of surfaces from image focus by local cross-sectional multivariate statistical analyses: Application to human *ex vivo* corneal endotheliums. *Medical Image Analysis*, 16(6):1293–1306, 2012. 131, 133

[9] Y. Gavet, J. Debayle, and J.-C. Pinoli. The Color Logarithmic Image Processing (CoLIP) Antagonist Space. In *Color Image and Video Enhancement*, pages 155–182. Springer International Publishing, 2015. 155

[10] Y. Gavet, J. Debayle, and J.-C. Pinoli. The Color Logarithmic Image Processing (CoLIP) Antagonist Space and Chromaticity Diagram. In *International Workshop on Computational Color Imaging*, pages 131–138. Springer International Publishing, 2015. 155

[11] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Pearson, 2007. 96, 103

[12] H. Gouinaud. *Traitement logarithmique d'images couleur*. PhD thesis, Ecole Nationale Supérieure des Mines de Saint-Etienne, 2013. 155

[13] H. Gouinaud, Y. Gavet, J. Debayle, and J.-C. Pinoli. Color correction in the framework of color logarithmic image processing. In *Image and Signal Processing and Analysis (ISPA), 2011 7th International Symposium on*, pages 129–133. IEEE, 2011. 155

[14] E. Grisan, A. Paviotti, N. Laurenti, and A. Ruggeri. A lattice estimation approach for the automatic evaluation of corneal endothelium density. In *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 1700–1703, 2005. 41

[15] F. C. Groen, I. T. Young, and G. Ligthart. A comparison of different focus functions for use in autofocus algorithms. *Cytometry*, 6(2):81–91, 1985. 132

[16] M. Jourlin and J. C. Pinoli. Logarithmic image processing. *Acta Stereologica*, 6:651–656, 1987. 147

[17] M. Jourlin and J.-C. Pinoli. Image Dynamic Range Enhancement and Stabilization in the Context of the Logarithmic Image Processing Model. *Signal Process.*, 41(2):225–237, Jan. 1995. 148

[18] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International journal of computer vision*, 1(4):321–331, 1988. 231

[19] J. Koenderink. The structure of images. *Biological Cybernetics*, 50(5):363–370, 1984. 107

[20] H. P. Kramer and J. B. Bruckner. Iterations of a non-linear transformation for enhancement of digital images. *Pattern Recognition*, 7(1):53 – 58, 1975. 68

[21] E. Krotkov. Focusing. *International Journal of Computer Vision*, 1(3):223–237, 1988. 133

[22] A. Lang and J. Potthoff. Fast simulation of Gaussian random fields. *Monte Carlo Methods and Applications*, 17(3):195–214, 2011. 385

[23] P. Maragos and M. Akmal Butt. Partial differential equations in image analysis: continuous modeling, discrete processing. In *Image Processing, 1996. Proceedings., International Conference on*, volume 3, pages 61–64 vol.3, Sep 1996. 109

[24] G. Marsaglia. Choosing a Point from the Surface of a Sphere. *Ann. Math. Statist.*, 43(2):645–646, 04 1972. 399

[25] A. Mazzolo and B. Roesslinger. Monte-carlo simulation of the chord length distribution function across convex bodies, non-convex bodies and random media. *Monte Carlo Methods and Applications*, 10(3-4):443 – 454, 2004. 398

[26] S. K. Nayar and Y. Nakagawa. Shape from focus. *Pattern analysis and machine intelligence, IEEE Transactions on*, 16(8):824–831, 1994. 132

[27] N. Otsu. A Threshold Selection Method from Gray-Level Histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(1):62–66, Jan 1979. 210

[28] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(7):629–639, Jul 1990. 108

[29] J. C. Pinoli. *Contribution à la modélisation, au traitement et à l'analyse d'image*. PhD thesis, Département de Mathématiques, Université de Saint-Etienne, France, 1987. 147

[30] J. C. Pinoli. The logarithmic image processing model: Connections with human brightness perception and contrast estimators. *Journal of Mathematical Imaging and Vision*, 7(4):341–358, Oct. 1997. 147

[31] S. Rivollier, J. Debayle, and J.-C. Pinoli. Shape diagrams for 2d compact sets-part i: analytic convex sets. australian journal of. *The Australian Journal of Mathematical Analysis and applications*, 7(2), 2010. 355

[32] S. Rivollier, J. Debayle, and J.-C. Pinoli. Shape diagrams for 2d compact sets-part ii: analytic simply connected sets. *The Australian Journal of Mathematical Analysis and applications*, 7(2), 2010. 355

[33] S. Rivollier, J. Debayle, and J.-C. Pinoli. Shape diagrams for 2d compact sets-part iii: convexity discrimination for analytic and discretized simply connected sets. *The Australian Journal of Mathematical Analysis and applications*, 7(2), 2010. 355

[34] A. Ruggeri, E. Grisan, and J. Jaroszewski. A new system for the automatic estimation of endothelial cell density in donor corneas. *Br J Ophthalmol*, 89(3):306–311, 2005. 41

[35] A. Ruggeri, E. Grisan, and J. Schroeter. Evaluation of repeatability for the automatic estimation of endothelial cell density in donor corneas. *Br J Ophthalmol*, 0, 2007. 41

[36] J. C. Russ and R. T. Dehoff. *Practical Stereology*. Kluwer Academic, 2nd edition, 2000. 391

[37] D. Sage, L. Donati, F. Soulez, D. Fortun, G. Schmit, A. Seitz, R. Guiet, C. Vonesch, and M. Unser. Deconvolutionlab2: An open-source software for deconvolution microscopy. *Methods*, 115:28 – 41, 2017. Image Processing for Biologists. 115

[38] B. Selig, K. A. Vermeer, B. Rieger, T. Hillenaar, and C. L. Luengo Hendriks. Fully automatic evaluation of the corneal endothelium from in vivo confocal microscopy. *Biomed central*, 2015. 36, 41

[39] J. Serra. *Image Analysis and Mathematical Morphology.* London: academic press, 1982. 175

[40] D. Sharvit, J. Chan, H. Tek, and B. B. Kimia. Symmetry-based indexing of image databases. In *Content-Based Access of Image and Video Libraries, 1998. Proceedings. IEEE Workshop on*, pages 56–62. IEEE, 1998. 295, 349, 355

[41] P. Soille. *Morphological Image Analysis: Principles and Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 2003. 175, 201

[42] S. A. Sugimoto and Y. Ichioka. Digital composition of images with increased depth of focus considering depth information. *Applied optics*, 24(14):2076–2080, 1985. 132

[43] J. M. Tenenbaum. *Accommodation in computer vision.* PhD thesis, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1970. 133

[44] K. Worsley. The geometry of random fields. *Chance*, 9(1):27–40, 1997. 385

Image processing tutorials with python This book is a collection of tutorials and exercises given at MINES Saint-Etienne as part of the Master's Degree in Science and Executive Engineering ("Ingénieur Civil des Mines – ICM"). In recent years, project-based learning has been used to illustrate theoretical concepts with real and concrete applications.

Whether you are in the early years of your university studies, in preparatory classes for the French Grandes Ecoles or in an engineering school, or even as a teacher, this book is made for you. You will find a large number of tutorials, classified by field, to familiarize yourself with the theoretical concepts of image processing and analysis.

Go to http://iptutorials.science to download the complete codes in python.

Yann GAVET He graduated from Mines Saint-Etienne with a Master's Degree in Science and Executive Engineering ("Ingénieur Civil des Mines - ICM") in 2001, obtained a Master of Science in 2004 and defended his PhD thesis in 2008. He teaches signal-processing, image-processing and pattern-recognition as well as C programming at Master's level.

Johan DEBAYLE He received his master of science and PhD thesis in 2002 and 2005. He is the head of the master MISPA of MINES Saint-Étienne, and teaches signal-processing, image-processing and pattern-recognition at Master's level.

