TOOLBOX IPR

Yann GAVET
Johan DEBAYLE
Sept. 2020

MINES
Saint-Étienne

Table of Contents

1 Binary Mathematical Morphology 5
2 Granulometry 15
3 Topological Description 21
4 Integral Geometry 27
5 Stochastic Geometry / Spatial Processes 33
6 Shape Diagrams 45
7 Boolean Models 53
8 Stereology and Bertrand's paradox 59
9 Hough transform and line detection 73
10 Freeman Chain Code 79
11 Voronoï Diagrams and Delaunay Triangulation 89
12 Harris corner detector 97
13 Local Binary Patterns 101
14 Morphological skeletonization 107
15 Convex Hull 113
16 Alpha Shapes 119

Credits
Cover image: Pepper & Carrot, David Revoy, www.davidrevoy.com.


Informations

For MATLAB® product information, please contact:
The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA, 01760-2098 USA
Tel: 508-647-7000
Fax: 508-647-7001
E-mail: info@mathworks.com
Web: https://www.mathworks.com
How to buy: https://www.mathworks.com/store
Find your local office: https://www.mathworks.com/company/worldwide

Contributors:

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

1 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. 1.1:

Figure 1.1: Images to use for this tutorial.

(a) I_1. (b) M. (c) I_2. (d) Cells.

1.1 Introduction to mathematical morphology

Mathematical morphology started in the 1960s with Serra and Matheron [?]. 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 [?].

The erosion of a binary set A by the structuring element B is defined by:

$$\varepsilon_B(A) = A \ominus B = \{z \in A | B_z \subseteq A\} \tag{1.1}$$

where, $B_z = \{b + z | b \in B\}$.

The dilation can by obtained by:

$$\delta_B(A) = A \oplus B = \{z \in A | (B^s)_z \cap A \neq \varnothing\} \tag{1.2}$$

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: $A \bullet B = (A \oplus B) \ominus B$.

1.2 Elementary operators

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 MATLAB® functions are imdilate, imerode, imopen and imclose. The function strel creates a structuring element.

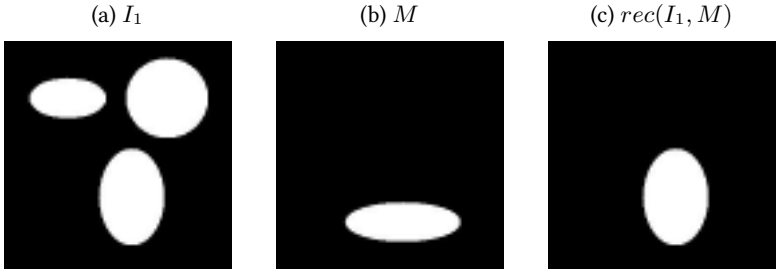
The python functions come from the python module scipy.ndimage.morphology. Useful functions are binary_dilation, binary_erosion, binary_opening and binary_closing. The function ndimage.generate_binary_structure creates a structuring element.

1.3 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. 1.2.

To do this, we iteratively dilate the marker M while being included in I_1 (see Eq.1.3). 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 1.2: 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:

$$\rho_I(M) = \lim_{n \to \infty} \underbrace{\delta_I \circ \ldots \circ \delta_I(M)}_{n \text{ times}} \tag{1.3}$$

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 1: The algorithm of this morphological reconstruction

1. Implement the algorithm.
2. Test this operator with the images I_1 and M.

The function bwarea evaluates the area of a 2D binary object.

Evaluate the area of a set may be done by counting the number of its pixels.

1.4 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 imreconstruct.

The morphological reconstruction function to use is binary_propagation in the module ndimage.morphology.

1.5 Cleaning of the image of cells

1. Threshold the image of cells (Fig. 1.1d).
2. Process the resulting binary image with the 3 cleaning processes of the previous question.

1.6. Python correction

```python
from scipy import ndimage, misc
import numpy as np

import matplotlib.pyplot as plt
```

1.6.1 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.

```python
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.

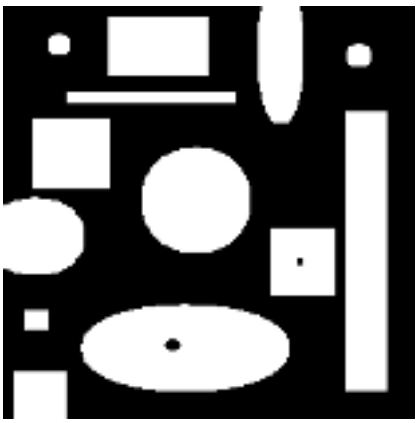
```
# Opening
Bsquare_open = ndimage.morphology.binary_opening(B, structure=square);
plt . subplot (233) ;
plt . imshow(Bsquare_open);plt. title ("opening")
imageio.imwrite('open.png', Bsquare_open);
# Closing
Bsquare_close = ndimage.morphology.binary_closing(B, structure =square);
plt . subplot (234) ;
plt . imshow(Bsquare_close); plt . title (" closing ")
imageio.imwrite('close .png', Bsquare_close);
```

The results are presented in Fig. 1.3.

Figure 1.3: Basic mathematical morphology operations.

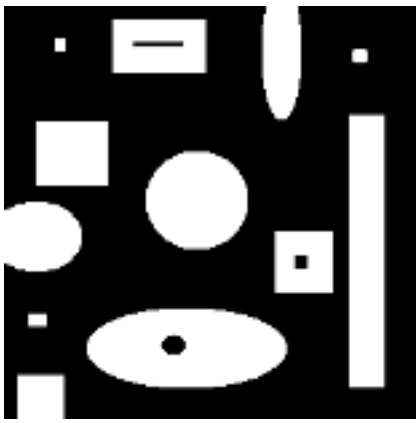
(a) Original image.

(b) Dilation.



(c) Erosion.

(d) Opening.



(e) Closing.

1.6.2 Morphological reconstruction

The algorithm of morphological reconstruction is coded like this in python:

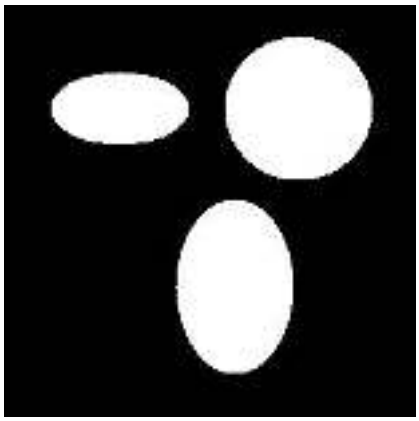
```python
def reconstruct(image, mask):
    # should be binary images
    M = np.minimum(mask, image);

    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. 1.4 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();
```

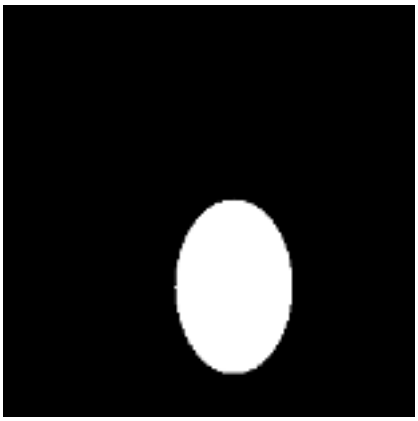
Figure 1.4: Morphological reconstruction.

(a) Image A.



(b) Image M.

(c) AM=reconstruct(A, M);.



1.6.3 Operators by reconstruction

Remove objects touching the borders

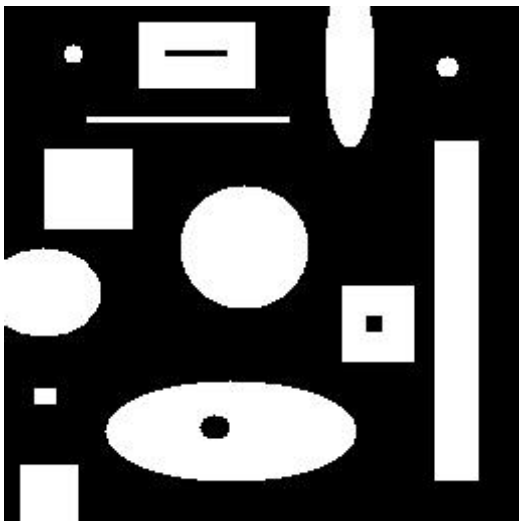
The Fig. 1.5 illustrates the suppression of objects touching the borders of the image. If $\mathcal{B}$ represents the border of the image (create an array of the same size as the image, with zeros everywhere and ones at the sides), then this operation is defined by:

$$\text{killBorders}(I) = I \setminus \rho_I(\mathcal{B})$$

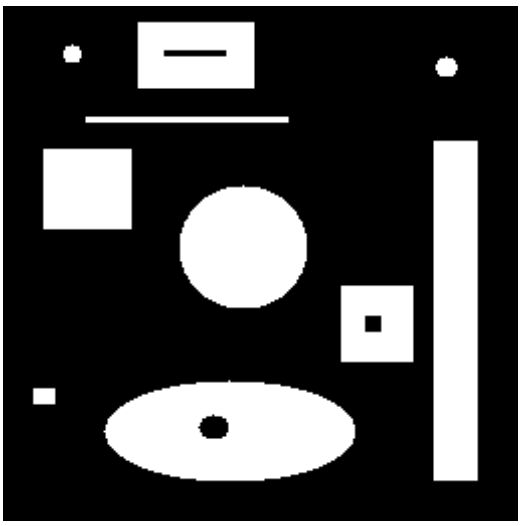
```
def killBorders(A):
    # remove cells touching the borders of the image
    m, n = A.shape
    M = np.zeros((m,n));
    M[0,:] = 1;
    M[m-1,:] = 1;
    M[:,0] = 1;
    M[:,n-1] = 1;
    M = reconstruct(A, M);
    return A-M
```

Figure 1.5: Suppress border objects.

(a) Original image.



(b) Objects removed.



Remove small objects

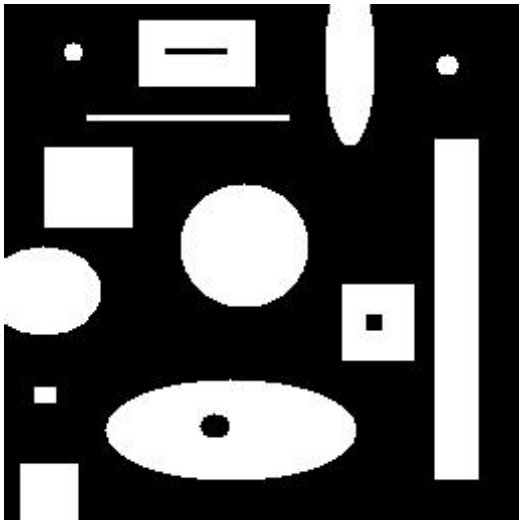
This is illustrated in Fig. 1.6. It consists in an erosion followed by a reconstruction. The structuring element used in the erosion defines the objects considered as "small".

$$\text{killSmall}(I) = \rho_I(\varepsilon(I))$$

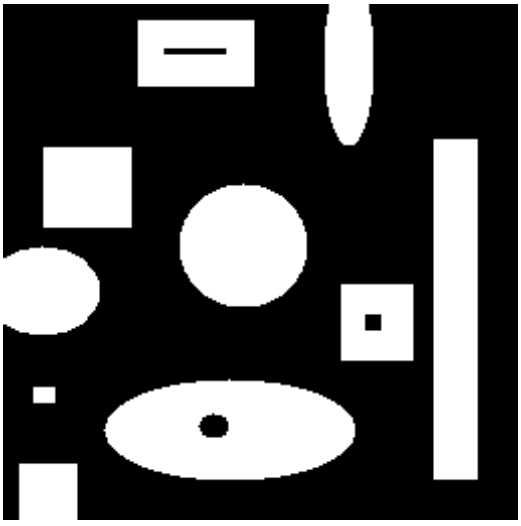
```
def killSmall(A, n):
    # destroy small objects (size smaller than n)
    se = np.ones((n, n));
    M = ndimage.morphology.binary_erosion(A, structure=se);
    return reconstruct(A, M);
```

Figure 1.6: Small objects removal.

(a) Original image.



(b) Small objects are removed.



Close holes in objects

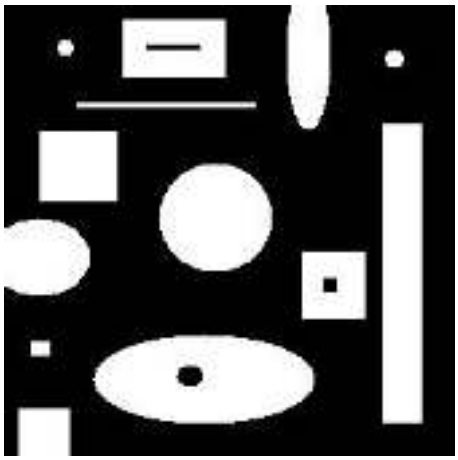
This is illustrated in Fig. 1.7. The operation is given by the following equation, with $\mathcal{B}$ the border of image I, and X^C denoting the complementary of set X:

$$\text{removeHoles(I)} = \{\rho_{I^C}(\mathcal{B})\}^C$$

```
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 1.7: Hole filling.

(a) Original image.



(b) Holes in objects are closed.

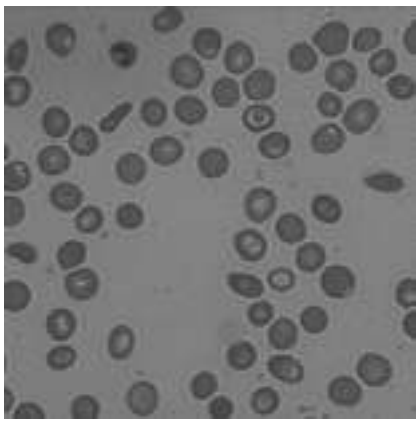
1.6.4 Application

The application shown in Fig. 1.8 presents the segmentation of blood cells after removing small cells, closing holes and removing the cells touching the borders of the image.

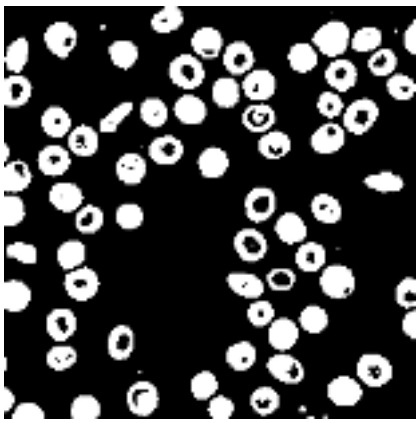
```
cells = imageio.imread('cells.jpg')<98;
imageio.imwrite('cellsbw.png', cells);
B = closeHoles(cells);
B = killBorders(B);
B = killSmall(B, 5);
plt.subplot(1,2,1);
plt.imshow(cells);
plt.subplot(1,2,2);
plt.imshow(B);
plt.title('clean image')
plt.show()
imageio.imwrite('clean.png', B);
```

Figure 1.8: Segmentation of the cells.

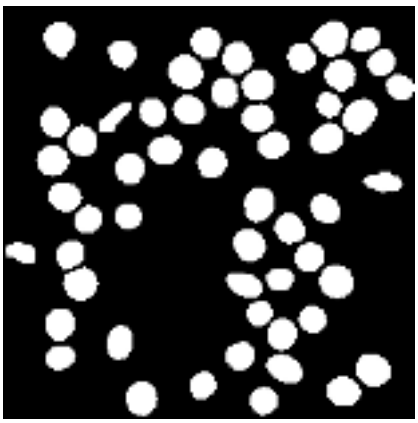
(a) Original image.



(b) Binarization.



(c) Final segmentation of the cells.



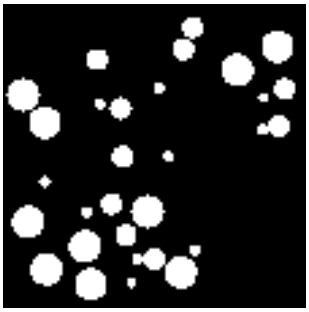
2 Granulometry

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.

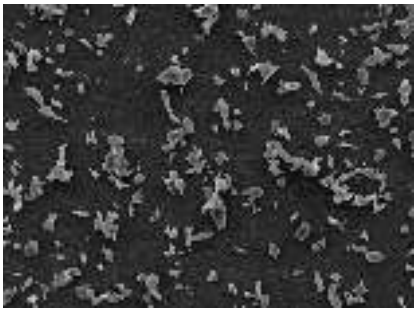
The image measurements will be realized on a simulated image of disks and an image of silicon carbide powder acquired by Scanning Electron Microscopy (SEM, Fig. 2.1).

Figure 2.1: These images are used for computing a granulometry of objects by mathematical morphology.

(a) Simulated image.



(b) Real image of powder, by SEM.



2.1 Morphological granulometry

2.1.1 Introduction to mathematical morphology

The tutorial 1 introduces the basic operations of the mathematical morphology, as well as the morphological reconstruction. More informations can be found in [?].

With Matlab, the useful functions are imopen for the morphological opening, imreconstruct for the geodesic reconstruction. For counting the number of objects, one can use bweuler is the case of objects with no hole, or bwlabel for a labelling and counting algorithm.

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.

2.1.2 Granulometry

- Load and visualize the image 'simulation' Fig.2.1.
- 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.

Homothetic structuring element can be defined as `se = strel('disk', i, 0)` for disks of size i.

Use `generate_binary_structure` and `iterate_structure` from python module `scipy.ndimage` to define homothetic structuring elements.

2.2 Real application

- Load and visualize the image 'powder' of Fig. 2.1b.
- 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?

2.3. Python correction

2.3.1 Simulation

The construction of an homothetic structuring element is necessary for computing a granulometry (through the function iterate_structure .

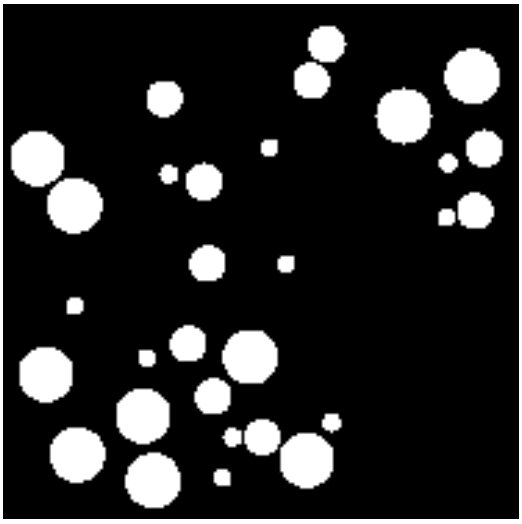
```
def granulometry(BW, T=35):
    # total original area
    A = ndimage.measurements.sum(BW);

    # 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. 2.2, generated by the following code:

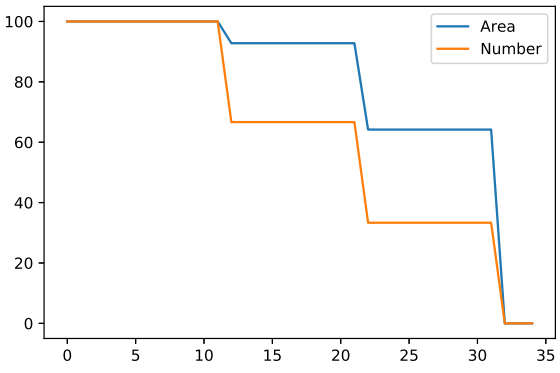
```
## Granulometry of synthetic image
# read binary simulated image, normalize it
I = imageio.imread("simulation .png") /255;
I = I [:,:,2]>.5;
plt . figure () ;
plt .imshow(I);
plt .show();

granulometry(I, 35) ;
```

Figure 2.2: Granulometry on simulated image.

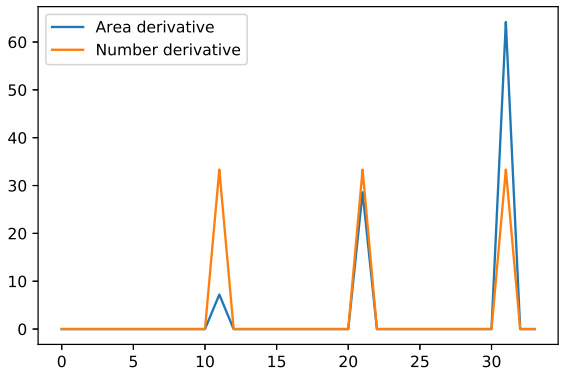
(a) Simulated image of disks.



(b) Granulometry in number and area.



(c) Derivatives.



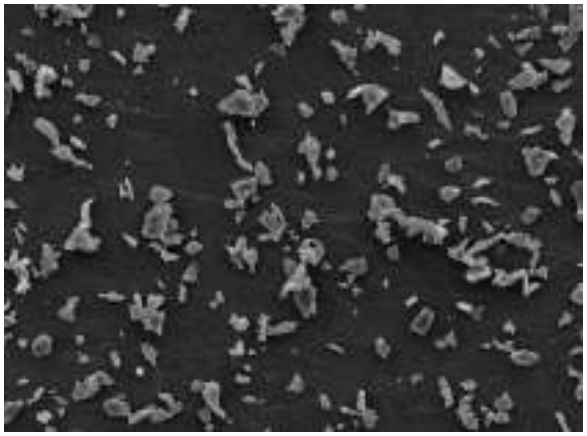
2.3.2 Powder image and segmentation

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. 2.3c.

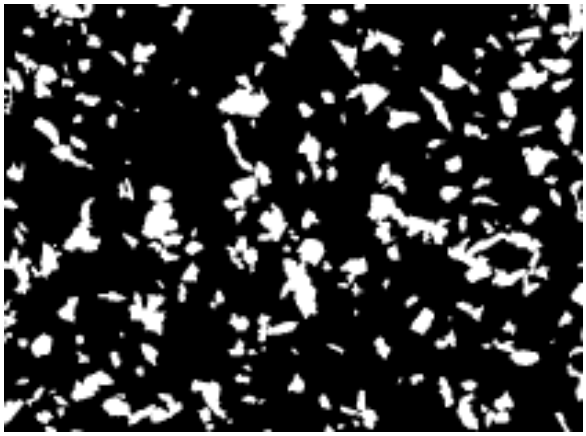
```
## Granulometry of real image
I = imageio.imread("poudre.bmp");

# 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 2.3: Results of granulometry analysis of original image from (a).

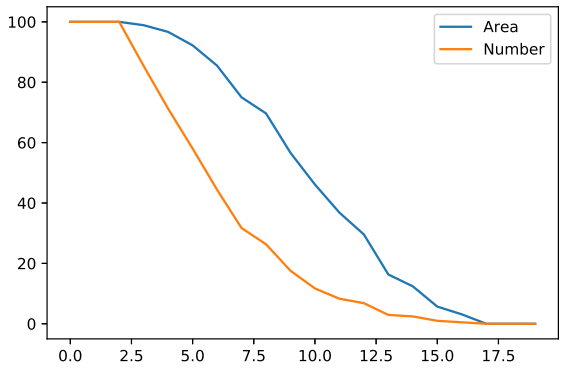
(a) Original image of powder.



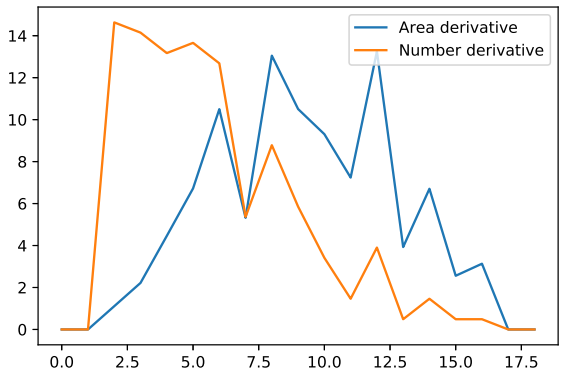
(b) Segmentation of image of (a).



(c) Granulometry.



(d) Derivative.

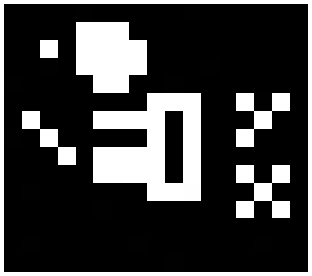


3 Topological Description

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 [?]for more details.

The different processes will be realized on the following binary image.

Figure 3.1: Test

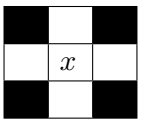


Preliminary definitions:

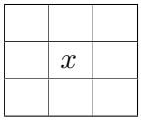
- y is 4-adjacent to x if $|y_1 - x_1| + |y_2 - x_2| \leq 1$.
- y is 8-adjacent to x if $\max(|y_1 - x_1|, |y_2 - x_2|) \leq 1$.
- $V_4(x) = \{y : y \text{ is 4-adjacent to } x\}$; $V_4^*(x) = V_4(x) \backslash \{x\}$.
- $V_8(x) = \{y : y \text{ is 8-adjacent to } x\}$; $V_8^*(x) = V_8(x) \backslash \{x\}$.

Figure 3.2: Different neighborhoods. By convention, pixels in white are of value 1, in black of value 0.

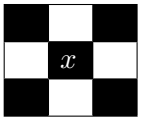
(a) $V_4(x)$



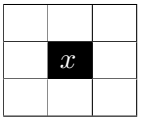
(b) $V_8(x)$



(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.

3.1 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.3.3), $T_8 = 2$, $\bar{T}_8 = 2$ and $TT_8 = 3$.

Figure 3.3: The pixel in the bottom right corner is not C-adjacent-4 to x.

(a) X (b) $V_8^*(x) \cap X$

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 the functions bwlabel and bwlabeln.

Use scipy.ndimage.measurements.

3.2 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. 3.4 shows the classification of 4 points.

Figure 3.4: Points configurations.

(a) end (b) isolated (c) border (d) interior

With the help of this table, classify the points of the image 'test'.

3.3. Python correction

3.3.1 Toplogical 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 = mes.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,selem=V4);
    L, T8c = mes.label(X, structure=V4);

    return T8, T8c, TT8
```

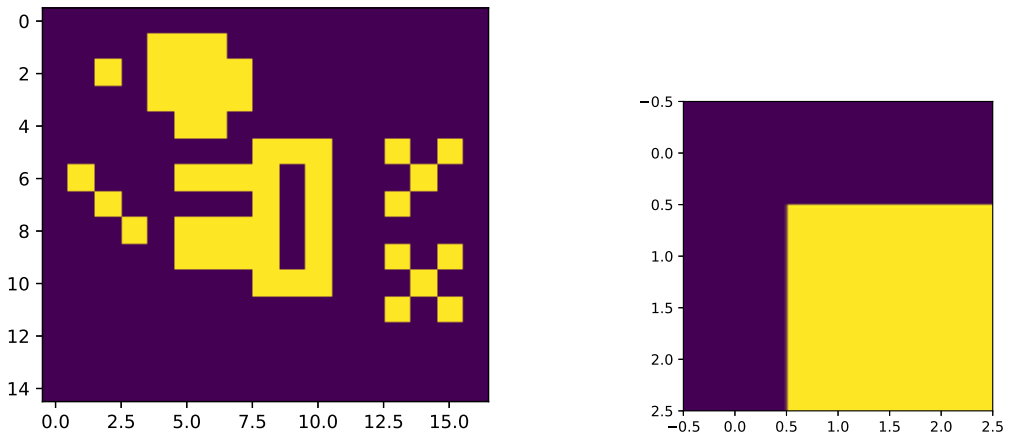


Figure 3.5: Extraction of the window centered in $x = (1, 4)$, given as (row,col).

3.3.2 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.3.6

```
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
```

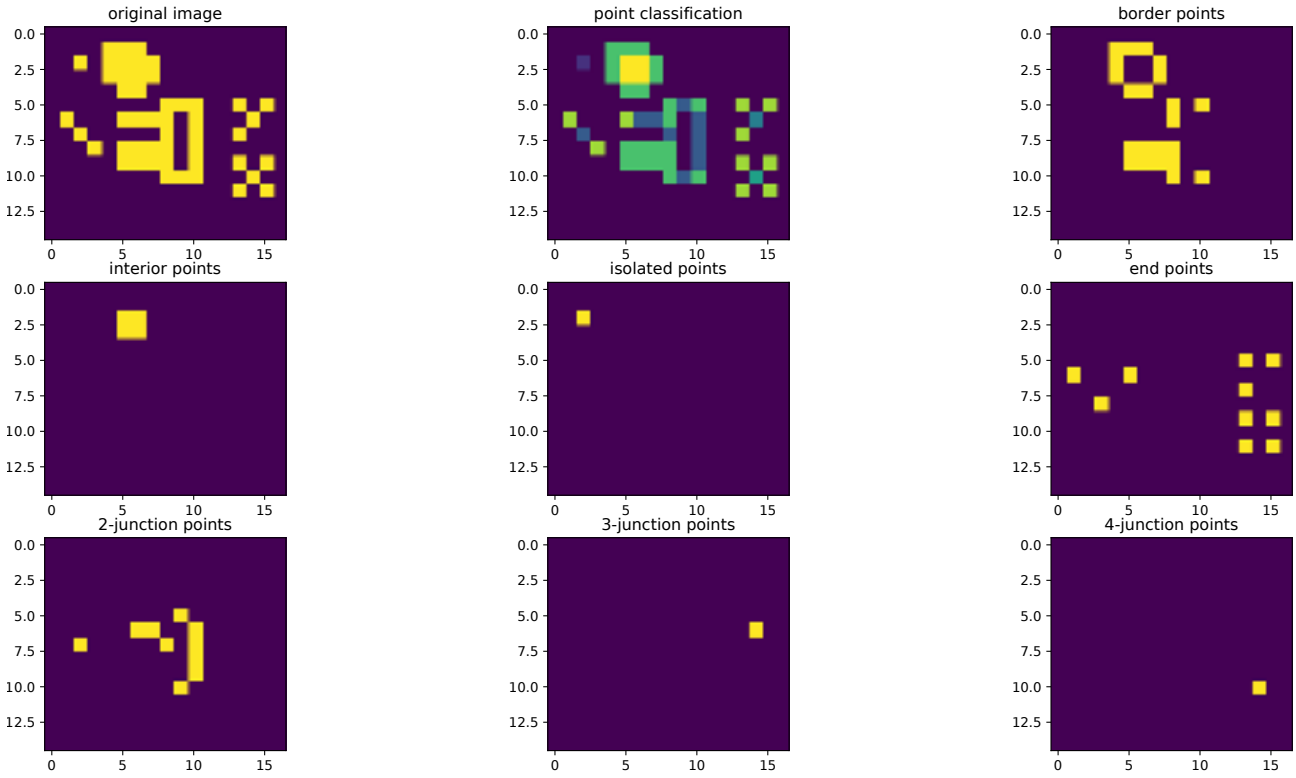


Figure 3.6: Classification of all the pixels of the the original image.

4 Integral Geometry

This tutorial aims to characterize objects by measurements from integral geometry.

The different processes will be applied on the following synthetic image:

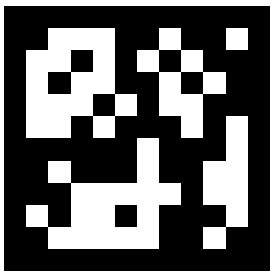


Figure 4.1: X

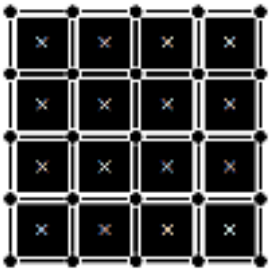
4.1 Cell configuration

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 4.2: 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:

$$A = f^{intra} = v^{inter} \tag{4.1}$$

$$P = -4f^{intra} + 2e^{intra} \tag{4.2}$$

$$\chi_8 = v^{intra} - e^{intra} + f^{intra} \tag{4.3}$$

$$\chi_4 = v^{inter} - e^{inter} + f^{inter} \tag{4.4}$$

1. Count manually the number of faces, edges and vertices of the image X for the two configurations (Intra- and Inter-pixel) of Fig. 4.1.
2. Deduce the measurements from integral geometry (Eq. 4.1-4.4).

4.2 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. 4.1.

Table 4.1: 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. 4.2). 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:

$$v = \sum_{\alpha=0}^{15} v_\alpha h(\alpha) \tag{4.5}$$

$$e = \sum_{\alpha=0}^{15} e_\alpha h(\alpha) \tag{4.6}$$

$$f = \sum_{\alpha=0}^{15} f_\alpha h(\alpha) \tag{4.7}$$

The following table gives the values of v_α, e_α and f_α for each cell configuration.

1. Compute the distribution of the neighborhood configurations from image X.
2. Deduce the number of vertices, edges and faces for each cell representation and compare these values with the previous (manually computed) results.

Table 4.2: 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

4.3 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:

$$P_4 = \frac{\pi}{2}\left(i_0 + i_{\pi/2}\right) \tag{4.8}$$

$$P_8 = \frac{\pi}{4}\left(i_0 + \frac{i_{\pi/4}}{\sqrt{2}} + i_{\pi/2} + \frac{i_{3\pi/4}}{\sqrt{2}}\right) \tag{4.9}$$

These perimeter measurements can be computed from the neighborhood configurations of the original image:

$$P_4 = \sum_{\alpha=0}^{15} P_\alpha^4 h(\alpha) \tag{4.10}$$

$$P_8 = \sum_{\alpha=0}^{15} P_\alpha^8 h(\alpha) \tag{4.11}$$

with the following weights P_α^4 and P_α^8 of these linear combinations (Tab. 4.3):

Table 4.3: 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.4.1

4.4. Python correction

4.4.1 Cell configurations

The following values are reported:

$$f^{intra} = 50$$
$$e^{intra} = 158$$
$$v^{intra} = 107$$

$$f^{inter} = 4$$
$$e^{inter} = 42$$
$$v^{inter} = 50$$

Then, it is easy to compute the following values:

$$\begin{aligned} A &= f^{intra} = 50 \\ P &= -4f^{intra} + 2e^{intra} = 116 \\ \chi_8 &= v^{intra} - e^{intra} + f^{intra} = -1 \\ \chi_4 &= v^{inter} - e^{inter} + f^{inter} = 12 \end{aligned}$$

4.4.2 Neighborhood configuration

The configuration is computed using the convolution function scipy.signal.convolve2d. The histogram of the different configurations is presented in Fig.4.3.

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([[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);
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:

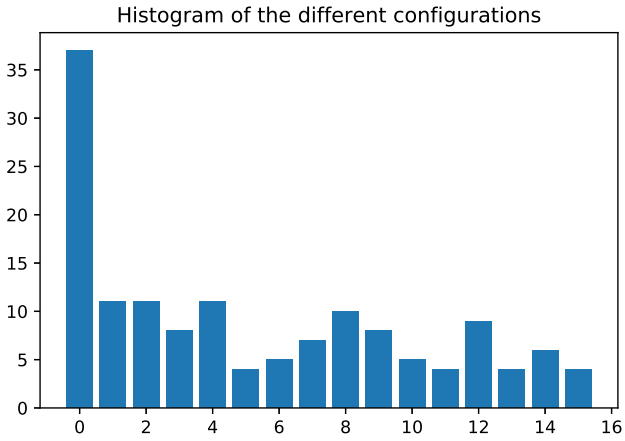


Figure 4.3: Distribution of the different neighborhood configurations.

```
# 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
```

4.4.3 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
```

5 Stochastic Geometry / Spatial Processes

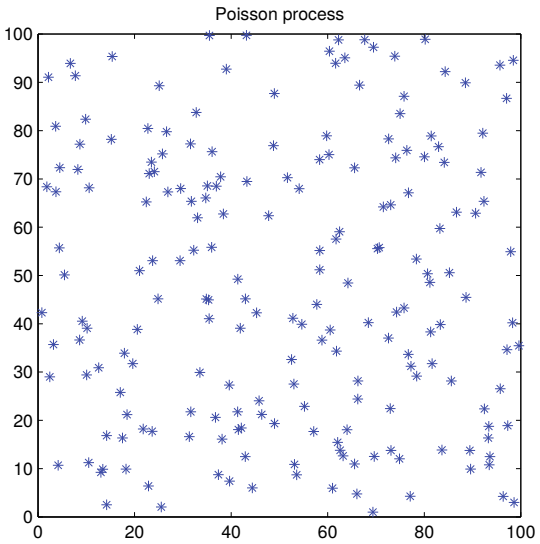
This tutorial aims to simulate different spatial point processes.

5.1 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 5.1: 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. 5.1).
2. Simulate a Gaussian distribution of points around a given center.

The function poissrnd can be used to generate and random variable following a Poisson law. Use rand for generating uniform distribution random variables.

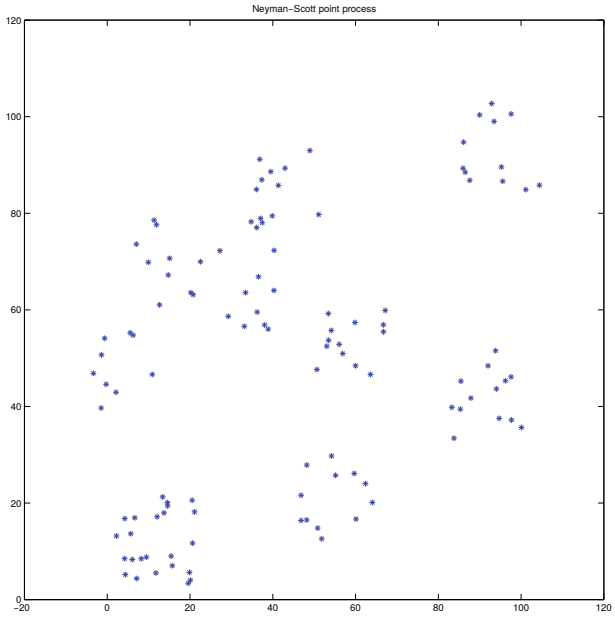
Import scipy.stats to use the function poisson and np.random for more classical stochastic functions.

5.2 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. 5.2).

Figure 5.2: 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.

5.3 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. 5.3). 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

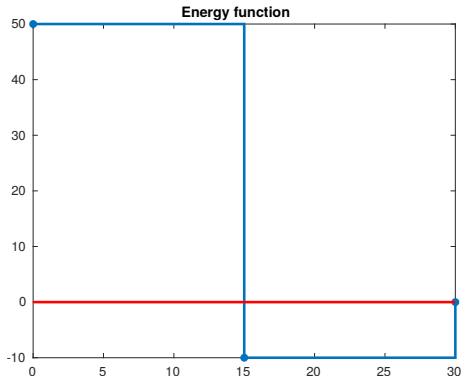
$$f(d) = \begin{cases} 50 & \text{if } 0 < d \leq 15 \\ -10 & \text{if } 15 < d \leq 30 \\ 0 & otherwise \end{cases}$$

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:

$$e = \sum_{(i,j),i \neq j} f(dist(x_i, x_j)) \tag{5.1}$$

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

Figure 5.3: Energy function f.



either preserved or replaced (by one of the four points) according to the configuration of minimal energy (see Fig.5.4).

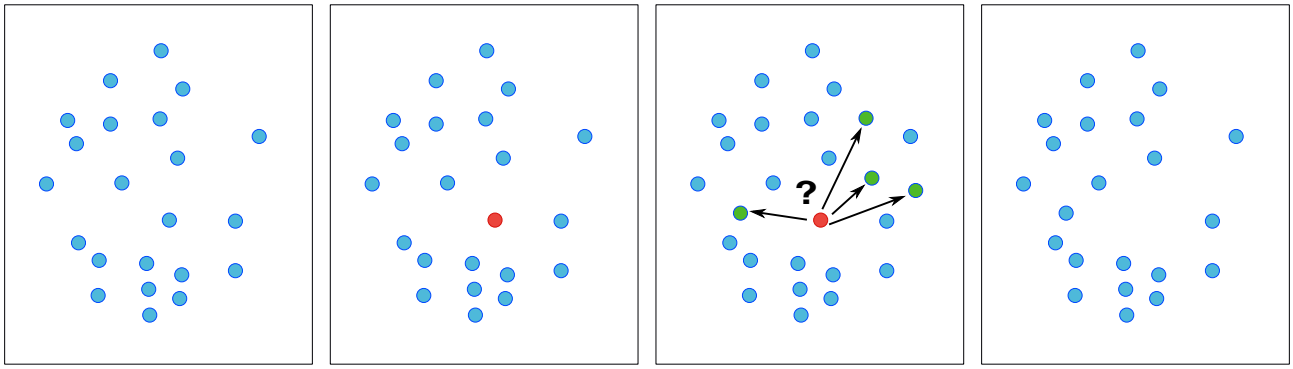
Figure 5.4: Illustration of iterative construction of the Gibbs point process.

(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.

```
function e = energy(energyFunction, x, y, xk, yk)
```

```
1 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. 5.5).

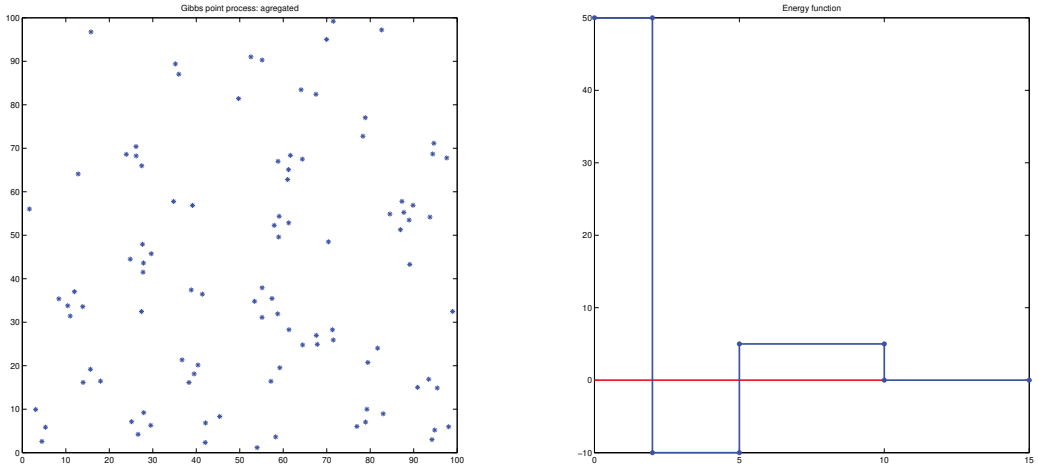
Informations

Use the function pdist from scipy.spatial.distance, which computes pairwise distances of all points.

Informations

Use the function pdist, which computes pairwise distances of all points. You may also consider the pdist2 function.

Figure 5.5: Gibbs agregated point process and its energy function.



5.4 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 $K(r)$:

$$K(r) = \frac{1}{\lambda}\frac{1}{N}\sum_{i=1}^{N}\sum_{i\neq j} k_{ij} \tag{5.2}$$

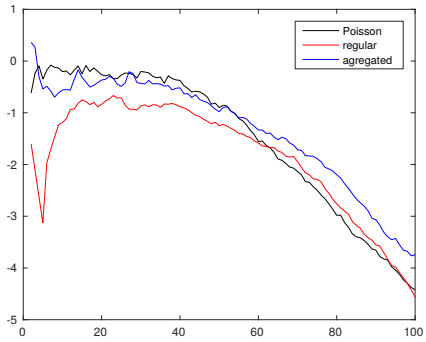
where N is the number of points in the studied domain D, $\lambda = N/D$ is the estimator of the density of the process and k_{ij} takes the value 1 if the distance between the point i and the point j is lower than r, and 0 in the other case.

We denote:

$$L(r) = \sqrt{K(r)/\pi} \tag{5.3}$$

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 5.6.

Figure 5.6: Ripley's function $L(r) - r$.



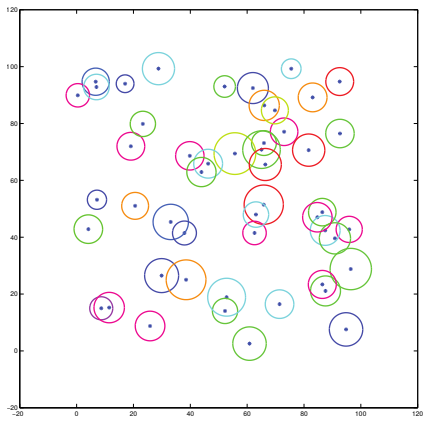
5.5 Marked point processes

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. 5.7

Figure 5.7: 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.



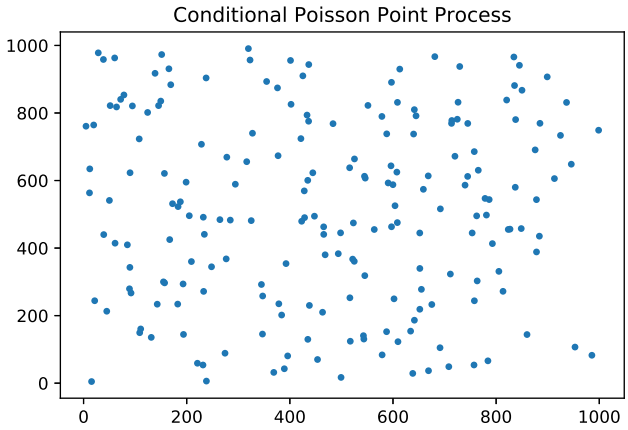
5.6. Python correction

```
import numpy as np;
import matplotlib.pyplot as plt;
from scipy.spatial.distance import pdist;
from scipy.stats import poisson
```

5.6.1 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.5.8.

Figure 5.8: Conditional Poisson point process, with N=100 points.



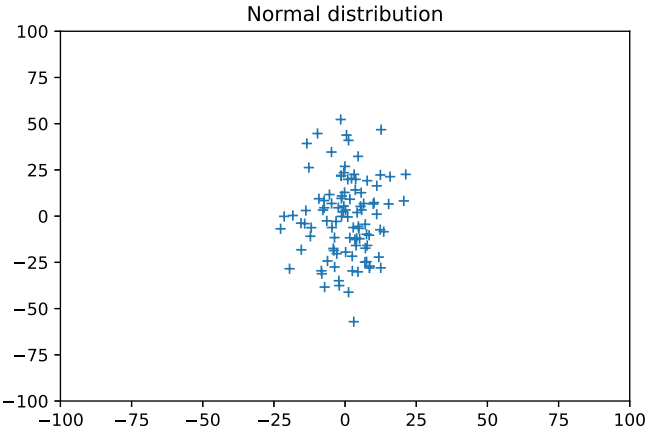
```
def cond_Poisson(nb_points, xmin, xmax, ymin, ymax):
    # Conditional Poisson Point Process
    # uniform distribution
    # nb_points: number of points
    # xmin, xmax, ymin, ymax: defined the domain (window)
    x = xmin + (xmax-xmin)*np.random.rand(nb_points)
    y = ymin + (ymax-ymin)*np.random.rand(nb_points);
    return x,y

def test_ppp():
    # testing function
    x,y = cond_Poisson(100, 0, 100, 0, 100);
    plt.plot(x,y,'+');
```

5.6.2 Normal distribution

The normal distribution is illustrated in Fig.5.9. Each marginal distribution (distribution on each axis) follows the normal distribution.

```python
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 5.9: Normal distribution of points around $(0, 0)$, with $\sigma = (10, 20)$.

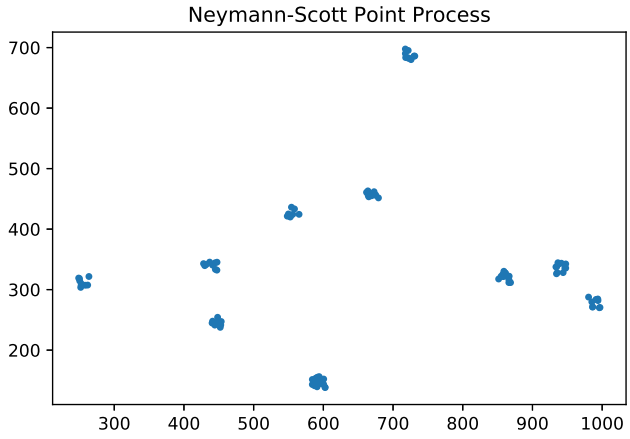


5.6.3 Neyman-Scott Process

Neymann-Scott point process is an agregated Poisson point process. It is illustrated in Fig.5.10. It consists on a "sub" point processes generated at locations corresponding to a point process.

```python
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
    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 5.10: Neyman-Scott point process, with lambdaS=10 and rSon=10.



5.6.4 Gibbs Point Process

Gibbs point process allows attraction and repulsion at different distances. It is illustrated in Fig.5.11. The attraction/repulsion law is given by the following code for regular or agregated point process.

```
def exampleEnergyFunction(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.

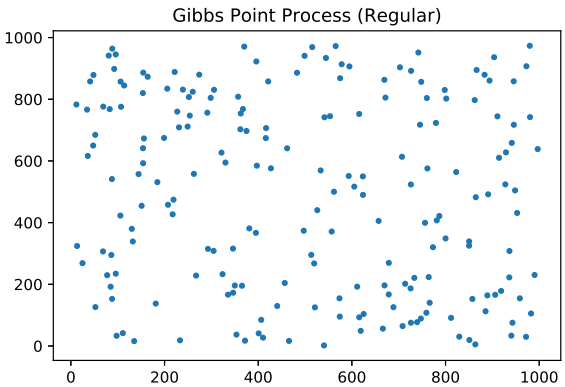
```
def energy(P, eFunction=exampleEnergyFunction):
    """
    This computes the energy in the set of points P, with the energy function
    given as a parameter.
    return a float value
    """
    P = np.transpose(P);
    d = pdist(P);
    e = exampleEnergyFunction(d);
    return np.sum(e);

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.

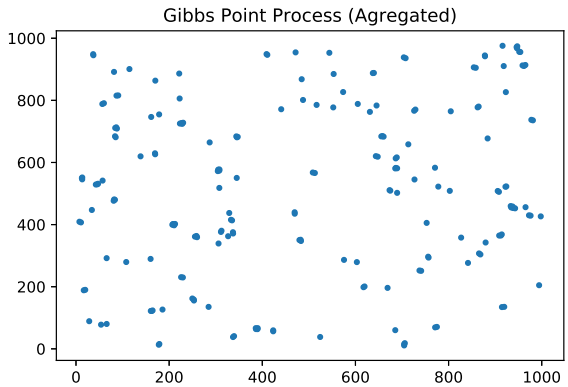
```python
def gibbs(nb_points, xmin, xmax, ymin, ymax, nbiter , eFunction=exampleEnergyFunction):
    """
    Gibbs point process
    xmin, xmax, ymin, ymax represents the spatial window
    nb_points: number of generated points
    nbiter : number of iterations
    returns (x,y) coordinates of the points
    """

    # start with a Poisson Point Process
    x,y = spat_pp.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 = spat_pp.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 5.11: Gibbs point processes (with the same number of points).

(a) Regular Gibbs point process.



(b) Agregated Gibbs point process.



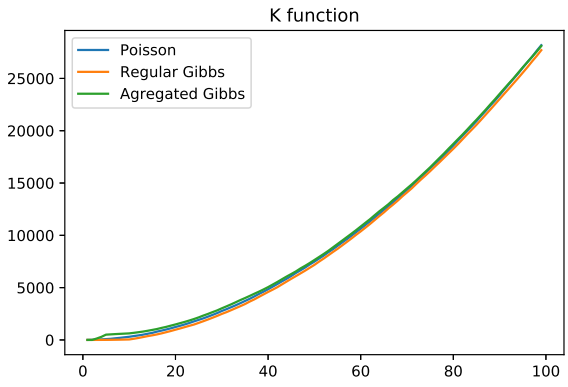
5.6.5 Ripley functions

The Ripley functions are useful to characterize a point process. Agregation and repulsion can be observed with regard to the distance (see Fig.5.12). 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 .

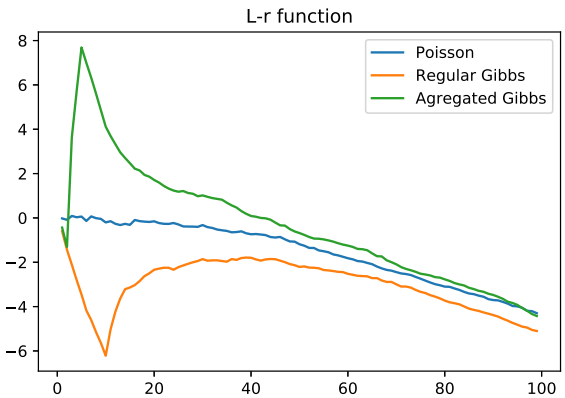
```python
def ripley(x, y, xmin, xmax, ymin, ymax, edges):
    # Ripley K and L functions, vals is values of radius
    # this function has border effects !
    # x, y: coordinates of points
    # xmin, xmax, ymin, ymax: window
    # edges: values of bins for histogram evaluation

    # 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 5.12: Ripley functions.

(a) Ripley K function.



(b) Ripley L function.

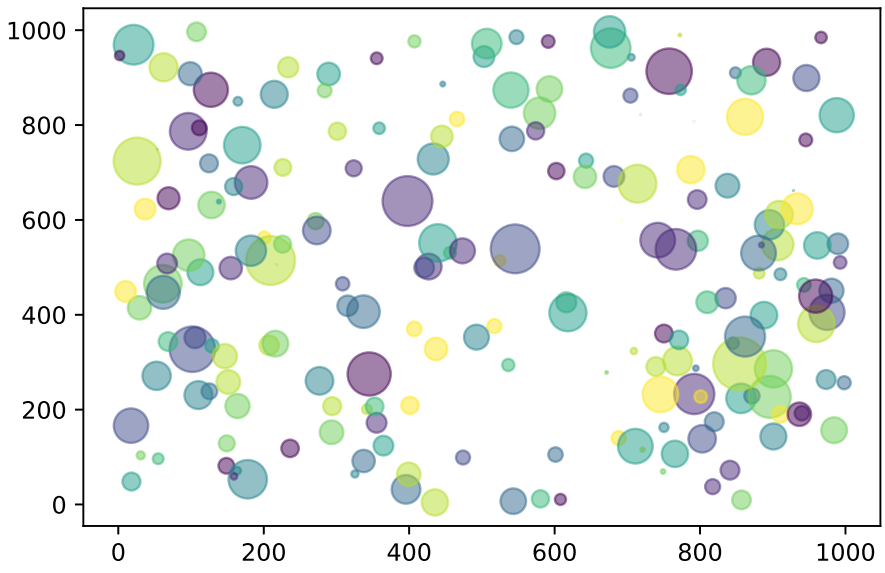


5.6.6 Marked Point Process

An illustration is presented in Fig.5.13. The algorithm simply consists in adding two new random variables in order to generate the radius and the color of each point.

```
def marked(nb_points, xmin, xmax, ymin, ymax):
    """
    marked point process
    """

    # points
    x,y = spat_pp.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 5.13: Marked point process.



6 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.

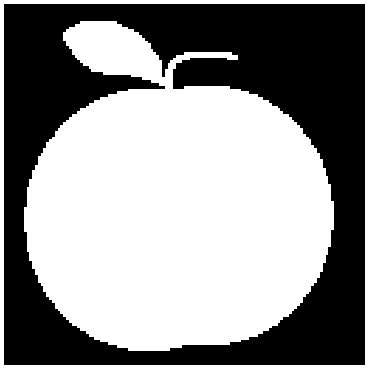
Shape diagrams are representations of single shapes (connected compact sets, see [?, ?, ?]) as points in the 2-D unit square plane. They are based on inequalities between 6 geometrical measurements: area A, perimeter P, radius of the inscribed circle r, radius of the circumscribed circle R, minimum Feret 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	$2r \leq \omega$	$2r/\omega$

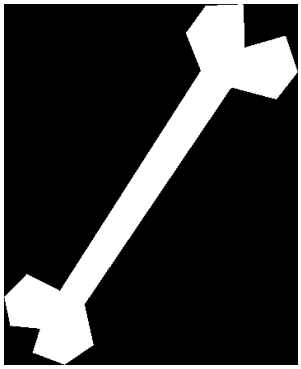
Table 6.1: Morphometrical functionals.

Figure 6.1: The different processes will be applied on images from the Kimia database [?, ?].

(a) Apple.



(b) Bone.



(c) Camel.

6.1 Geometrical functionals

Code functions in order to evaluate the different parameters:

- the area, Crofton perimeter and Feret diameters have been already presented in tutorial 4;
- the radius of the inscribed circle can be defined from the ultimate erosion of a set.

The function bwdist computes the distance map of a binary image.

The function scipy.ndimage.morphology.distance_transform_cdt computes the distance map of a binary image (chamfer distance transform).

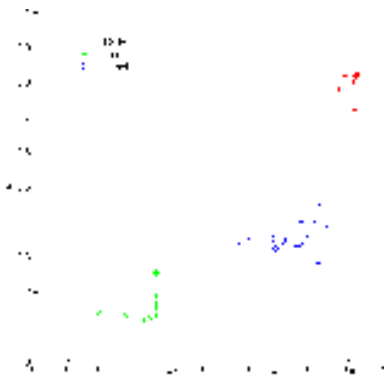
6.2 Morphometrical functionals

Code and evaluate some of the morphometrical functionals listed in the table 6.1. 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).

6.3 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.6.2 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 6.2: Example of a shape diagram.



6.4 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.

6.5. Python correction

```python
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
```

6.5.1 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).

```python
def crofton_perimeter(I):
    """ Computation of crofton perimeter
    """
    inter = [];
    h = np.array([[1, -1]]);
    for i in range(4):
        II = np.copy(I);
        I2 = scipy.misc.imrotate(II, 45*i, interp='nearest');
        I3 = scipy.ndimage.convolve(I2, h);

        inter.append(np.sum(I3>100));


    crofton = np.pi/4. * (inter[0]+inter[2] + (inter[1]+inter[3])/np.sqrt(2));
    return crofton
```

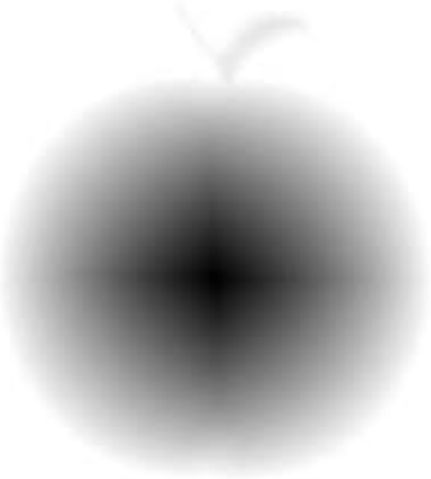
```
def feret_diameter(I):
    """
    Computation of the Feret diameter
    minimum: d (meso-diameter)
    maximum: D (exo-diameter)

    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.6.3a.

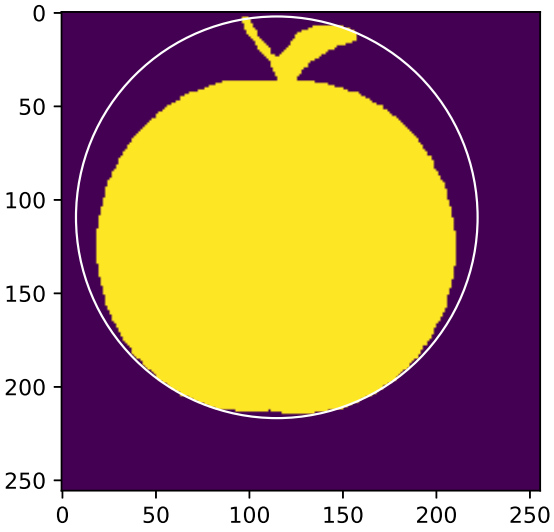
```
def inscribedRadius(I):
    """
    computes the radius of the inscribed circle
    """
    dm = scipy.ndimage.morphology.distance_transform_cdt(I>100);
    radius = np.max(dm);
    return radius;
```

Figure 6.3: 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.6.3b.

```
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;
```

6.5.2 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);
```

6.5.3 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.6.4 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]));
    accuracy +=np.sum(k_means_labels[20:40] == scipy.stats.mode(k_means_labels[20:40]));
    accuracy +=np.sum(k_means_labels[40:60] == scipy.stats.mode(k_means_labels[40:60]));
    accuracy = accuracy / 60 * 100;
    print('Accuracy: {0:.2f}%'.format(accuracy));
```

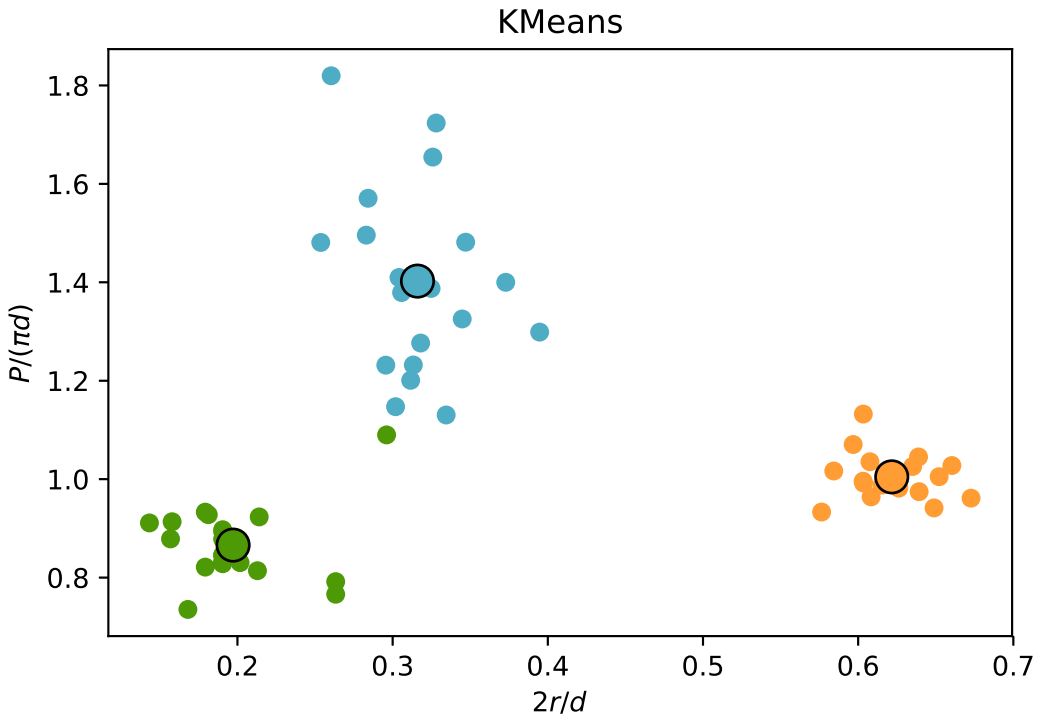


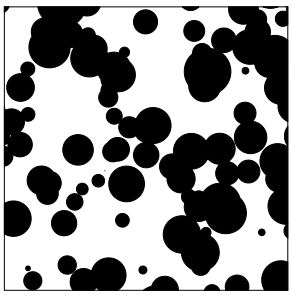
Figure 6.4: Illustration of the accuracy of the classification from a k-means method. The k-means method is not necessarily the adapted to these data. The measured accurary if of 98.3%.

7 Boolean Models

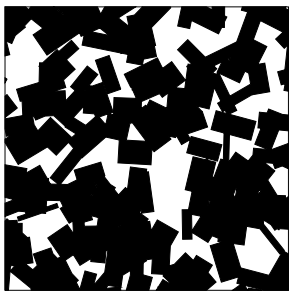
This tutorial aims to study a classical model coming from stochastic geometry: the Boolean model. The first objective is to simulate some realizations of a Boolean model of 2-D disks representing a population of overlapped particles. Thereafter, the geometrical characteristics of the individual disks (from a statistical point of view) will be analyzed.

Figure 7.1: Illustration of 2-D Boolean models observed in a squared window W.

(a) Boolean model of disks



(b) Boolean model of rectangles



7.1 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 $\gamma > 0$. Let $Z_0, Z_1, Z_2, \ldots$ 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 $x_1, x_2, \ldots$ are the germs and the random sets $Z_1, Z_2, \ldots$ are the grains of the Boolean model. The random set Z_0 is called the typical grain. The union of the translated grains:

$$Z = \bigcup_{i=1}^{\infty} (Z_i + x_i) \tag{7.1}$$

is a random closed set, which is called the stationary Boolean model with intensity λ and grain distribution $\mathbb{Q}$. The random collection $X = \{Z_1 + x_1, Z_2 + x_2, \ldots\}$ of the shifted grains is the particle process underlying the Boolean model.

Figure 7.1 shows a realization of two different Boolean models Z observed in a compact and convex observation window W.

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 W 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 MATLAB® function poissrnd and randi.
The meshgrid function can be used to generate the disks.

Look at the numpy functions random.poisson and random.randint.
Use skimage.draw. circle to generate the disks in an array.

7.2 Geometrical characterization of a 2-D Boolean model

Assume we observe Z in a compact, convex observation window W with positive volume (as shown in Figure 7.1). Our aim is to extract distributional information from the geometric properties of the sample $Z \cap W$. Thus, we assume we can measure geometric functionals like the perimeter of the sample $Z \cap W$ 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_ν that are related in the 2-D space to the measures of area (A), perimeter (P) and Euler number (χ):

$$W_0 = A \tag{7.2}$$
$$W_1 = P/2 \tag{7.3}$$
$$W_2 = \pi\chi \tag{7.4}$$

The boundary of the observation window W 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:

$$\overline{\phi}(Z) = \lim_{r\to\infty} \frac{\mathbb{E}[\phi(Z \cap rW)]}{W_0(rW)} \tag{7.5}$$

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:

$$\overline{\phi}(X) = \gamma\mathbb{E}[\phi(Z_0)] \tag{7.6}$$

Miles formulas express the observable Minkowski densities $\overline{W}_\nu(Z)$ in terms of the Minkowski densities $\overline{W}_\nu(X)$ and can be inverted.
For example in 2-D:

$$\overline{W}_0(Z) = 1 - e^{-\overline{W}_0(X)} \tag{7.7}$$
$$\overline{W}_1(Z) = e^{-\overline{W}_0(X)}\overline{W}_1(X) \tag{7.8}$$
$$\overline{W}_2(Z) = e^{-\overline{W}_0(X)}\left(\overline{W}_2(X) - \overline{W}_1(X)^2\right) \tag{7.9}$$

- 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.

7.3. Python correction

```
import numpy as np
from skimage import draw
from scipy import misc, signal
import matplotlib.pyplot as plt
import progressbar
```

7.3.1 Simulation of a 2-D Boolean model

The process for simulating the proposed Boolean model of 2-D disks consists in four steps:

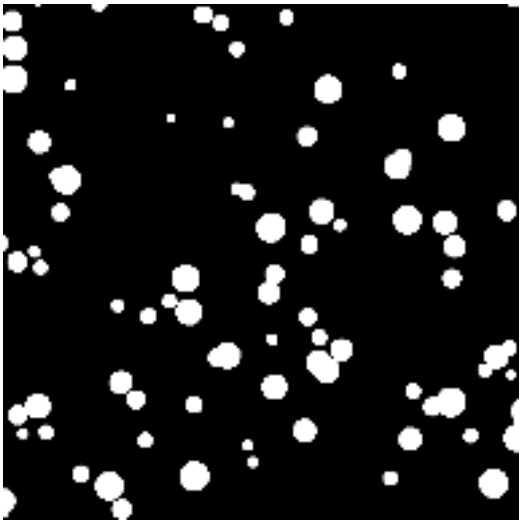
1. Generate the random number of points (by using the intensity parameter of the Poisson distribution). In order to avoid edge effects, one consider a larger window than the observation window to generate the disks. Indeed, a germ outside the observation window can generate a disk that intersects the 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 7.2: 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.7.2.

```
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:

```
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;
```

7.3.2 Geometrical characterization of a 2-D Boolean model

We can use the following function to compute the Minkowski functionals of the Boolean model (see the tutorial on Integral Geometry). The regionprops function from skimage.measure is not used because it computes the properties of each object.

```
def minkowskiFunctionals(X):
    """
    Evaluation of the Minkowski functionals
    X: boolean 2D array
    returns area, perimeter, euler number N8, euler number n4

    """
    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];
    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];
    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);

    return Area, Perimeter, EulerNb8, EulerNb4;
```

So we can estimate the Minkowski densites on different realizations of the Boolean model:

```
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, chi8, chi4 = minkowskiFunctionals(Z);
        W[i,:] = np.array([a, p/2, chi8*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):

```
W = realizations (Wsize, gamma, radius, 1000);
W = np.mean(W, axis=0);
# 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;
```

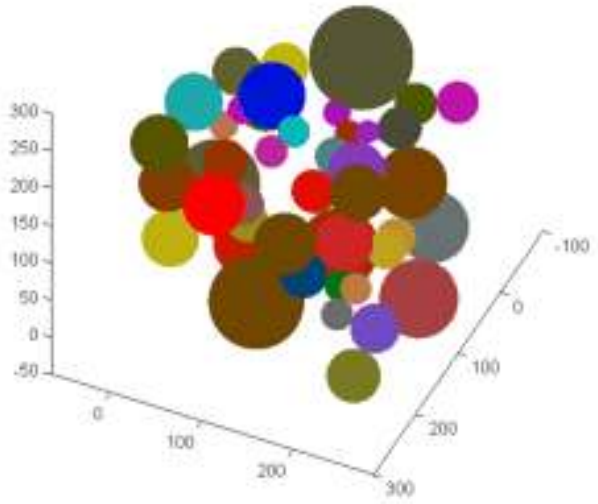
Here are the results for 100 specific realizations:

```
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.
Note that the Miles formulas can be inverted to estimate the Minkowski functionals of the typocal grain from the Minkowski mean densities of the Boolean model.

8 Stereology and Bertrand's paradox

This tutorial introduces the problems of stereological measurements, based on simple probes (points, lines...). In a second part, the Bertrand's paradox is explored in the case of the analysis of the distribution of chord lengths of disks and spheres. This tutorial uses the notation that can be found in [?] (among others).



8.1 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

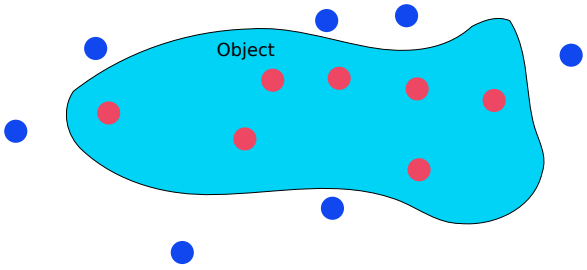
8.1.1 Probes

The measures are performed with the following definitions:

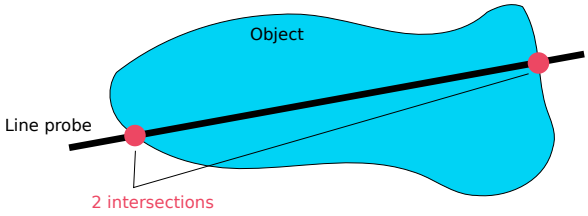
- $\langle P_P \rangle$: the count of the points that lie in the phase, normalized by the total number of points.
- $\langle P_L \rangle$: the count of the number of intersection of lines and the surface of the phase, normalized by the total length of the lines (in m^{-1}, see Fig.8.1). L_A is the ratio 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 8.1: 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.

8.2 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.

8.2.1 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.

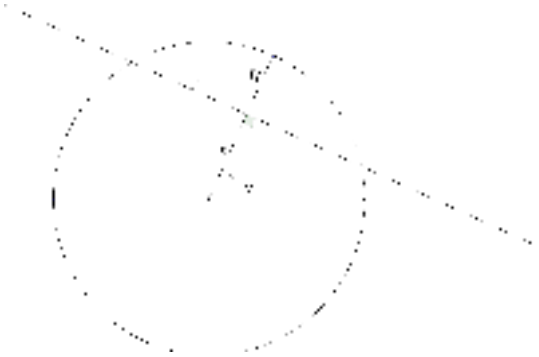
8.2.2 Random radius

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. 8.1 and Fig. 8.2).

$$r = \sqrt{R^2 - x^2} \tag{8.1}$$

[1] https://en.wikipedia.org/wiki/Bertrand_paradox_(probability)

Figure 8.2: Relation between the radius of the chord and the distance to the center of the disk.



As presented in Fig. 8.2, θ is randomly chosen in $[0; 2\pi[$ (uniform law), and x is randomly chosen in $[0; R]$ (uniform law).

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.

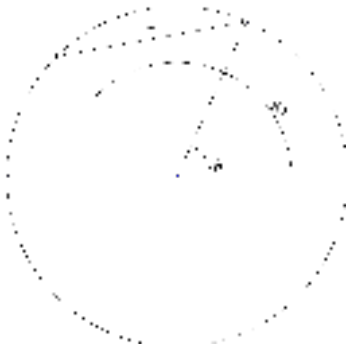
8.2.3 Random endpoints

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 8.3).

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 8.3: Second method for simulating a random chord of the disk.



8.2.4 Analytical values

The probability to obtain a chord of half length x between a and b is given by Eq. 8.2 (see [?]):

$$\mathbb{P}(x \in [a; b]) = \int_a^b \frac{\rho}{R\sqrt{R^2 - \rho^2}} \mathrm{d}\rho \tag{8.2}$$

By discretizing the interval $[0; R]$, compute the analytical value of the probability density function of the distribution of the radii with the Eq. 8.2.

8.3 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.

8.3.1 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. 8.4).

Code the following method:

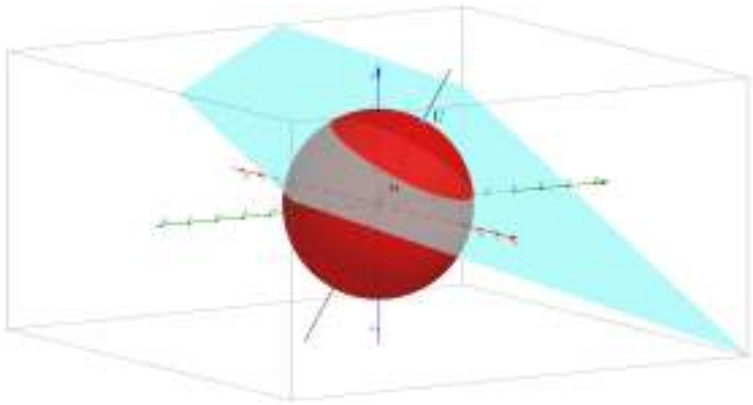
- A random point on the sphere is chosen with the following method (see [**?**]): let x, y and z be 3 Gaussian random variables, the point on the sphere is defined by the normed vector $\vec{u}$, such that:

$$\vec{u} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{8.3}$$

- 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. 8.4).

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 8.4: 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.

8.3.2 Second simulation: 3 random endpoints

The random plane $\mathcal{P}$ is defined by 3 random points laying on the sphere (see Eq. 8.3).

- 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.

8.3.3 Third simulation: 2 random endpoints

Choose 2 random points laying on the sphere (see Eq. 8.3) 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.

8.3.4 Comparison

Compare the 3 results and comment.

8.4. Python correction

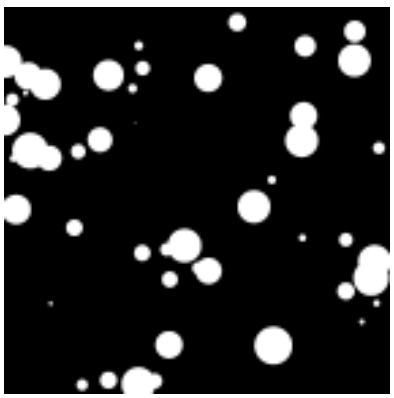
```python
import numpy as np
import matplotlib.pyplot as plt
import skimage.measure
import scipy
```

8.4.1 Classical measurements of stereology

To generate a binary image with overlapping random disks (see Fig.8.5), the following function is used:

```python
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 8.5: 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.

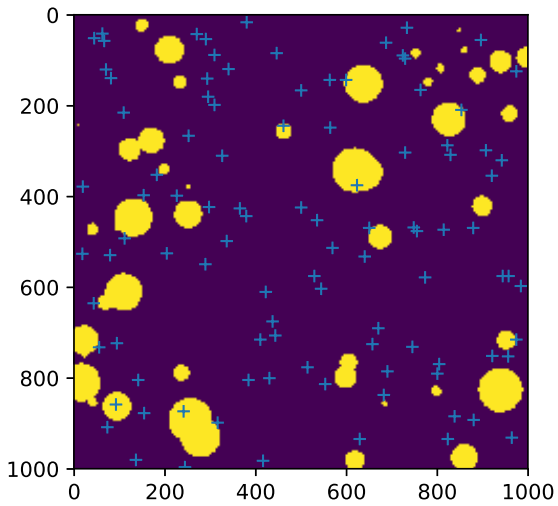
```python
def areaFraction(nb_probes, I):
    """
    Evaluates area fraction via point probes
    @param nb_probes: number of probes
    @param I: binary image, square
    """
    P = np.random.randint(I.shape[0], size = (nb_probes,2));

    # 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));
```

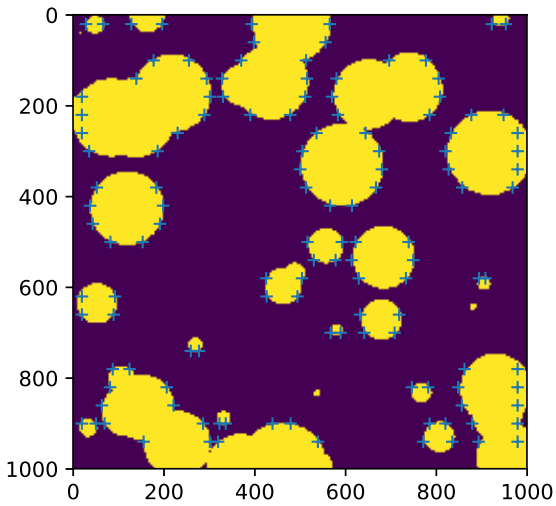
There is a good agreement between AA and PP. The method is illustrated in Fig.8.6a.

Figure 8.6: 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.8.6b.

```
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.

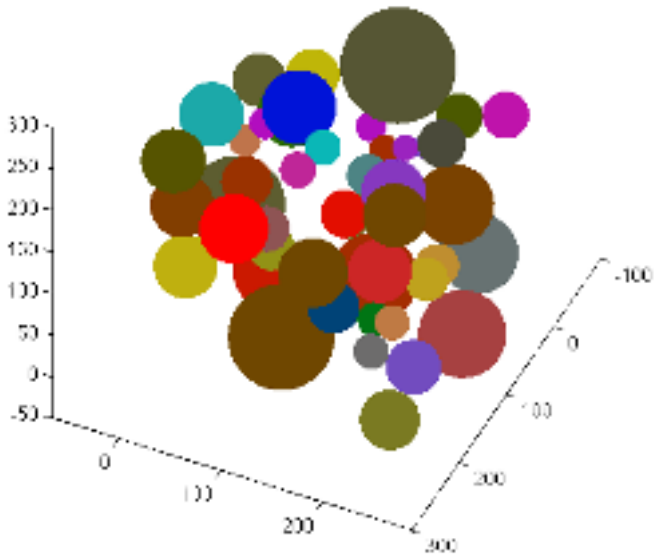
```
LA (true count): 18.02%
pi/2*PL (evaluated fraction): 15.20%
```

Volume fraction

This example loads a MATLAB® matrix.

```python
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 8.7: Random population of 3D overlapping spheres. This function uses povray and the vapory python API.



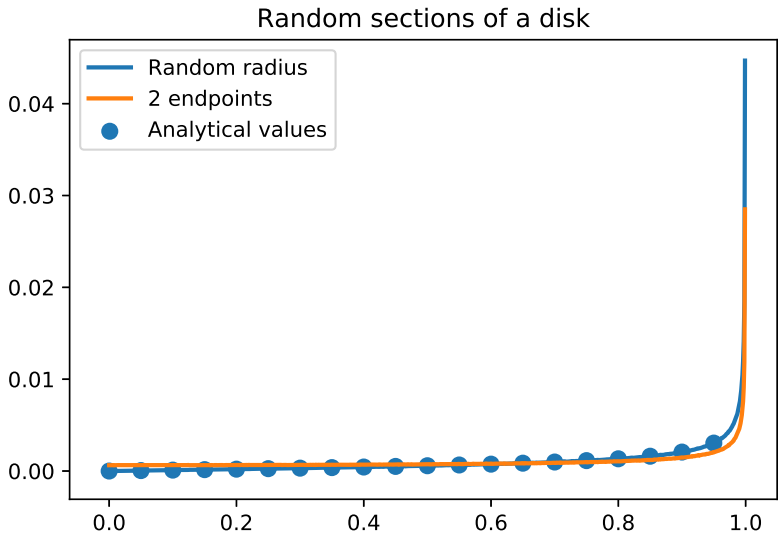
As for the area fraction, the volume fraction is precise. The results on this example are:

```
VV (true count): 5.02%
AA (evaluated fraction): 5.28%
```

8.4.2 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. 8.8.

Figure 8.8: 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);
probaSimu = np.histogram( radii , bins=nBins);
plt.plot(probaSimu[1][:-1], probaSimu[0]/N, linewidth=2);
```

The second method consists in choosing 2 random points on the circle, given by two random angles.

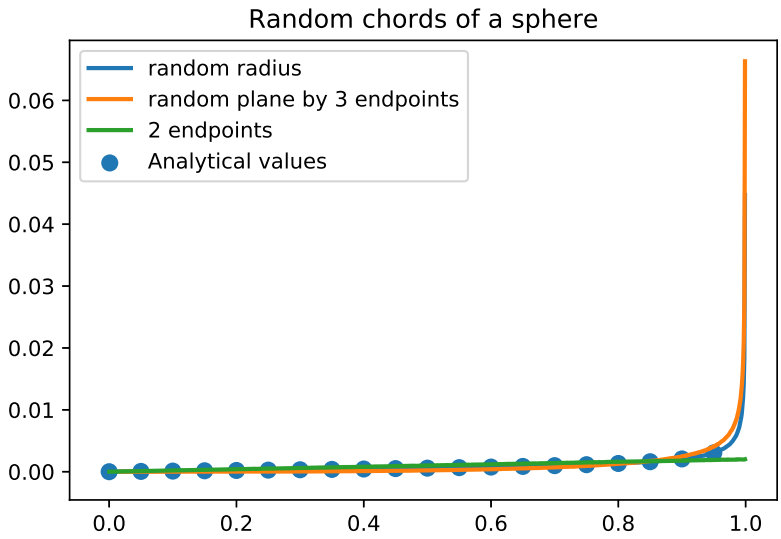
```
#2nd method: random points on the circle
# from 2 random angles
theta = np.pi*2*np.random.rand(int(N),2);
dX = np.diff(R * np.cos(theta))
dY = np.diff(R * np.sin(theta));
radii = 1./2 * np.sqrt(dX**2 + dY**2);
probaSimu2 = np.histogram( radii , bins=nBins);
plt.plot(probaSimu2[1][:-1], probaSimu2[0]/N, linewidth=2);
```

Finally, the analytical results are computed for comparison.

```
# 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')
```

8.4.3 Random planar sections of a sphere

Figure 8.9: Simulations of random chords of a sphere.




```python
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.

```python
# 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. 8.9.

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.8.9).

```
# 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. 8.9.

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.

9 Hough transform and line detection

This tutorial introduces the Hough transform. Line detection operators are implemented.

9.1 Introduction

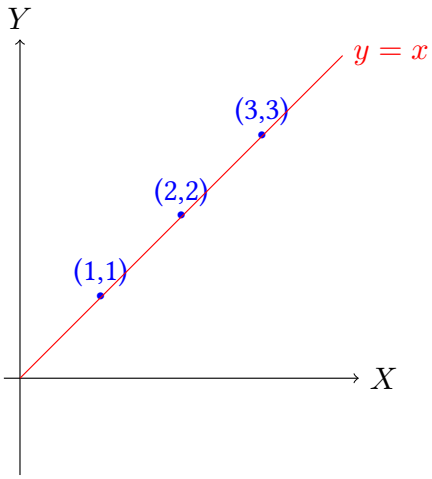
This tutorial deals with line detection in an image. For a given point of coordinates (x, y) in $\mathbb{R}^2$, there exists an infinite number of lines going by this point, with different angles θ. These lines are represented by the following equation:

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta).$$

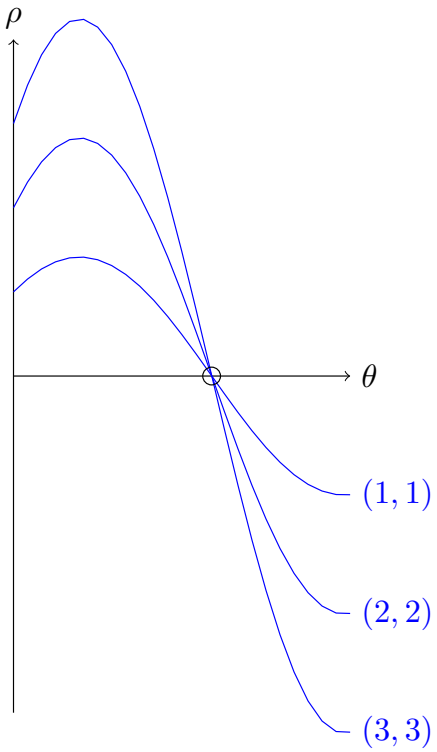
Thus, for each point (x, y) (Fig. 9.1a) corresponds a curve parametered by $[\theta, \rho]$, where $\theta \in [0;2\pi]$ (Fig. 9.1b). The intersection of these curves represents a line (in this case, $y = x$).

Figure 9.1: Representation of the Hough transform.

(a) Different points in $\mathbb{R}^2$.



(b) Hough transform of the 3 points.



9.2 Algorithm

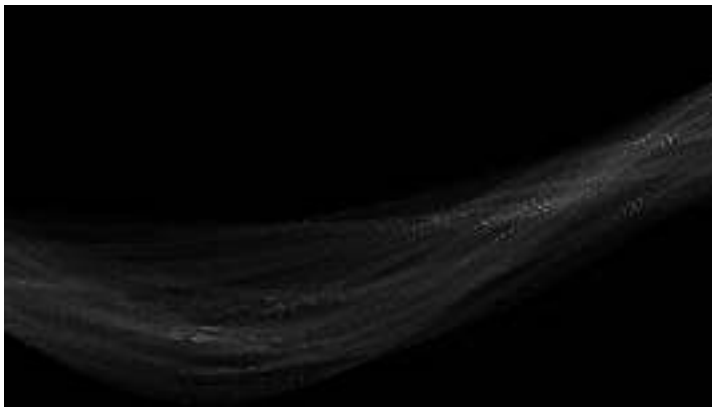
The (general and simple) method for line detection is then:

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.

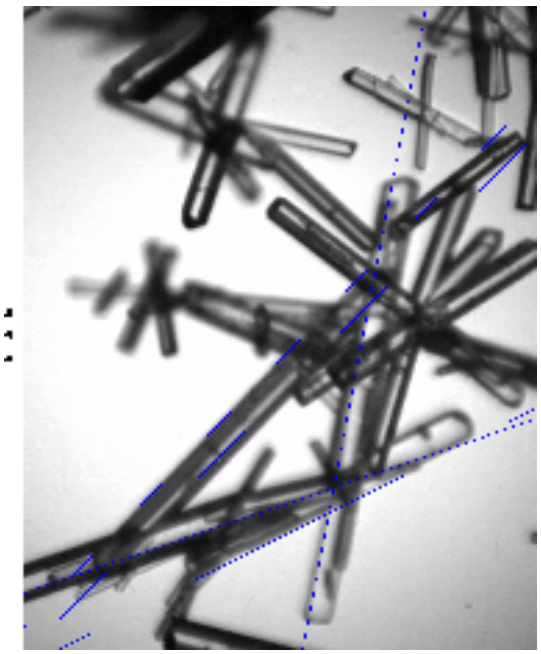
Results should look like in Fig. 9.2.

Figure 9.2: 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.



9.3 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.

9.4 Maxima detection

Use or code a function to detect maxima (regional maxima). For each maximum, keep only one point.

9.5 Display lines

For each maximum, display the corresponding line above the original image.

9.6. Python correction

This correction makes use of the python modules numpy, opencv, skimage and matplotlib.

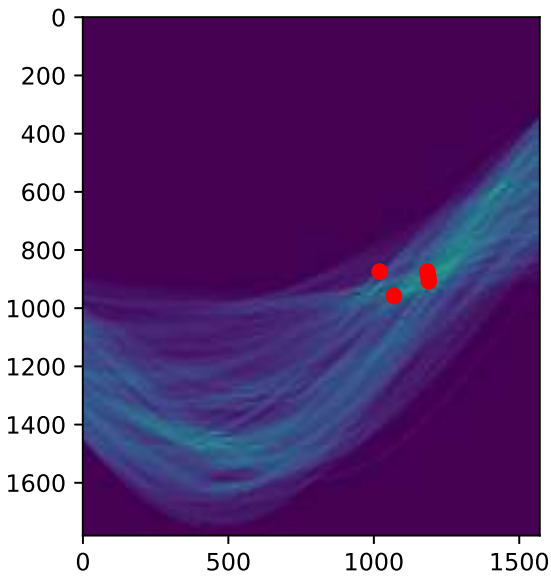
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.feature import peak_local_max
```

9.6.1 Contours detection

The contours are detected using the Canny edge detection method. In this code, the method from OpenCV is employed, see Fig.9.3a.

```
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 9.3: 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.

9.6.2 Hough transform

Notice that OpenCV contains Hough fonctions: HoughLines and HoughLinesP. The result (sinogram) of the image is presented Fig.9.3b.

```python
## Hough transform
# size of image
X = img.shape[0];
Y = img.shape[1];

angular_sampling = 0.01; # angles in radians

# 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]);

# Hough transform
# 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);
```

9.6.3 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.9.3b.

```python
# 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();
```

9.6.4 Resulting lines

The result is shown in Fig.9.4.

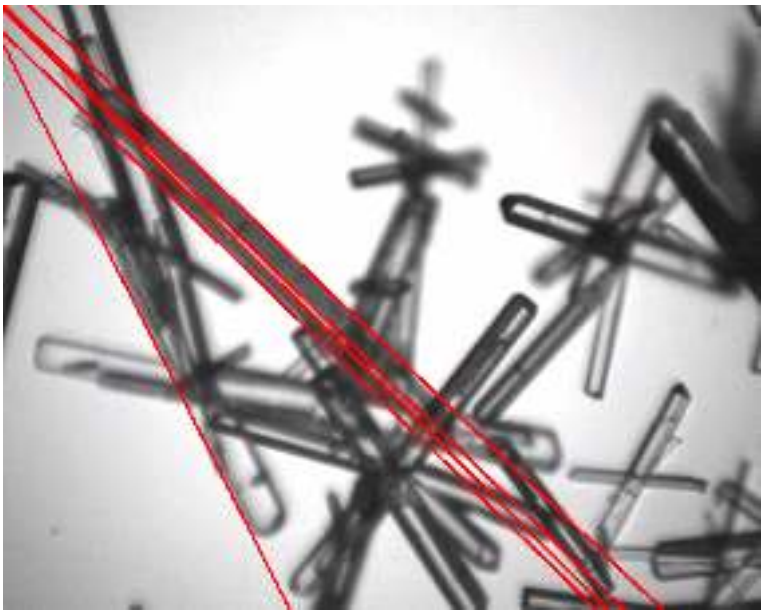


Figure 9.4: Lines detected with the Hough transform.

```
# 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);
```

9.6.5 OpenCV builtin function

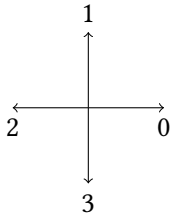
```
import cv2
import numpy as np

# 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);
```

10 Freeman Chain Code

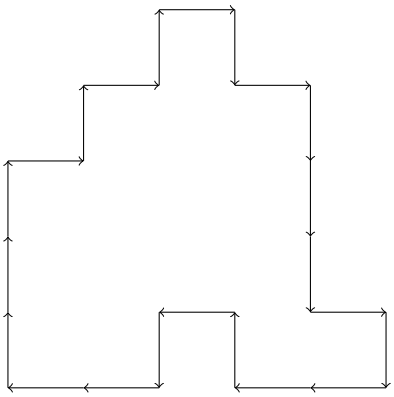
This tutorial is focused on shape representation by the Freeman chain code. This code is an ordered sequence of connected segments (of specific sizes and directions) representing the contour of the shape to be analyzed. The direction of each segment is encoded by a number depending on the selected connectivity (Fig. 10.1). In this example, the contour of the shape and its Freeman code are given in 4-connectivity.

Figure 10.1: Freeman chain code examples.

(a) 4-connected neighborhood.

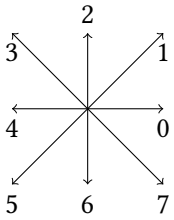


(b) Example Freeman code in a 4-neighborhood.



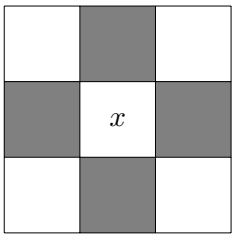
11101010303330322123 22

(c) 8-connected neighborhood.

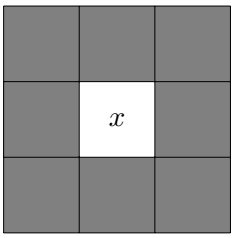


Notations:

- 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) $V_8(x)$

10.1 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:

$$\begin{aligned} x \in C_4(A) &\Leftrightarrow \exists y \in N_8(x) \quad y \in {}^cA \\ x \in C_8(A) &\Leftrightarrow \exists y \in N_4(x) \quad y \in {}^cA \end{aligned}$$

Informations

You can use the function bwperim with the appropriate connectivity number.

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.

10.2 Freeman chain code

From the shape contours, the Freeman chain code can be calculated.

1. From the binary array of pixels, extract the first point belonging to the shape (from left to right, top to bottom).

   Informations

   You can use np.argwhere to locate the first point.

   Informations

   You can use find to locate the first point.

2. From this initial point, determine the Freeman chain code c (counterclockwise direction) using the N_4 or N_8 connectivity.

10.3 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, $d = 3003131331300133031130$;

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.

```
function d=codediff(fcc,conn)
```

Use circshift for circularly shifting the freeman code array. Use polyval for transforming the array of values into a number, although rounding errors might occur, due to numerical approximation of floating numbers.

10.4 Geometrical characterization

10.4.1 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).

10.4.2 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.

10.5. Python correction

TB_IPR_TUT.IMG.freeman_pythonqrcode.png

```
import matplotlib.pyplot as plt
import numpy as np
from skimage.morphology import binary_erosion, disk, rectangle
from skimage.measure import perimeter
```

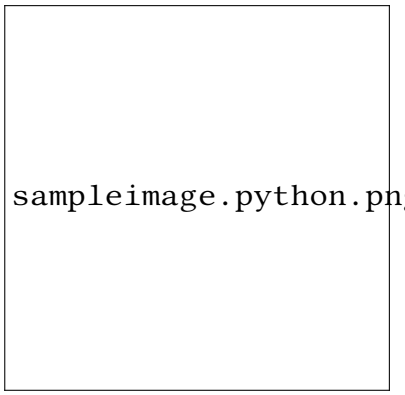
10.5.1 Shape contours

To generate a simple object, here is an example:

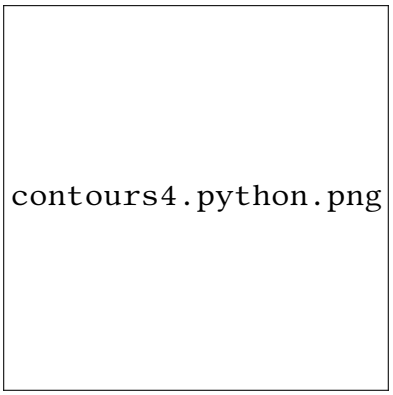
```
A = np.zeros ((20,20) ).astype('bool');
A[4:14, 9:17] = True;
A[1:18,11:16]= True;
```

The contours are computed in 4- or 8-connectivity, see Fig.10.2. This function uses the mathematical morphology in order to get the contour.

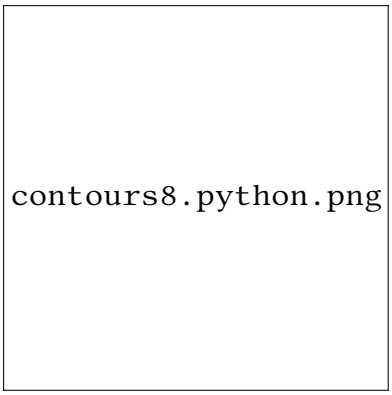
```python
def bwperim(I, connectivity=8):
    """
    Morphological inner contour, in 4 or 8 connectivity
    I: binary image
    return: binary image representing the contour
    """
    if connectivity==8:
        SE = disk(1);
    else:
        SE = rectangle(3,3);

    E = binary_erosion(I, selem=SE);

    return I^E;

# compute the contours
contours8 = bwperim(A, 4);
contours4 = bwperim(A, 8);
```



(c) Sample object.



(d) Contour in 4-connectivity.



(e) Contour in 8-connectivity.

Figure 10.2: Simple object and its contours in 4- or 8-connectivity.

10.5.2 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.

```python
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.

```
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;
```

```
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])
```

10.5.3 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, 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.

10.5.4 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;
```

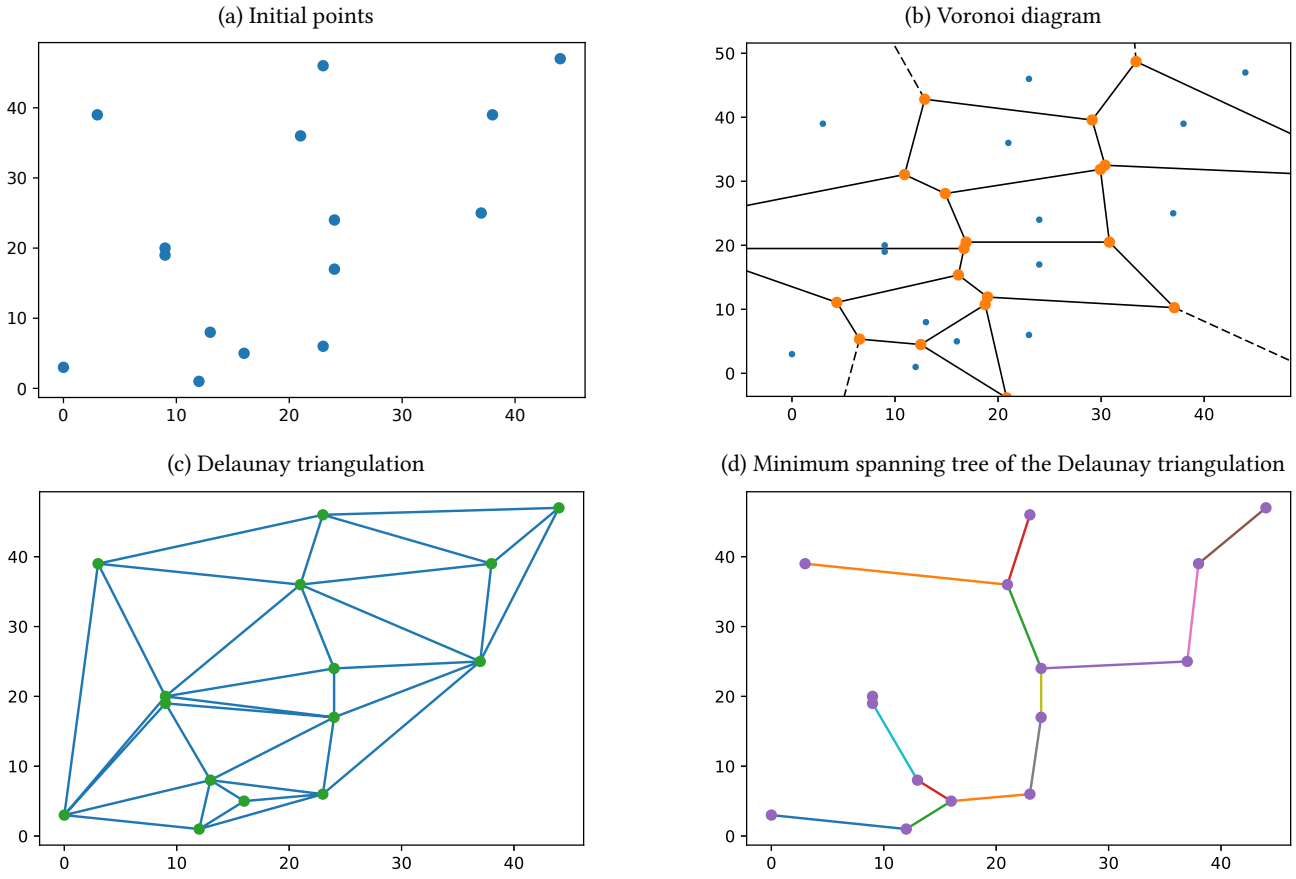
Notice that the area evaluated by this way is different from the number of pixels.

```
Area:  93.0
Number of pixels (area):  115
```

11 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.11.1.
For biomedical issues, this point pattern analysis can help the biologists to classify different populations of cells.

Figure 11.1: Random point pattern and some geometrical structures used to characterize it.



11.1 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.

11.1.1 Random tesselation

Follow these instructions to generate a random tesselation:

1. Generate a random point process.
2. Determine the Delaunay triangulation.
3. Determine the Voronoi diagram.

Use the MATLAB® functions delaunayTriangulation and voronoiDiagram.

Use the python functions Delaunay and Voronoi from scipy.spatial.

11.1.2 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:

$$AD = 1 - \frac{1}{1 + \frac{\sigma(A)}{\mu(A)}} \tag{11.1}$$

$$RFH = 1 - \frac{\sigma(RF)}{\mu(RF)} \tag{11.2}$$

where A and RF are calculated on the regions R_k of the Voronoi diagram. μ and σ 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.

- 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.

- Represent the couple (ad, rfh) in a graph, which gives a characterization of the Voronoi diagram.

See polyarea for evaluating the area of a polygon.

See shapely.geometry.Polygon for evaluating the area of a polygon.

11.1.3 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.

11.2 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.

It is implemented in Matlab via the function minspantree (introduced in MATLAB® 2015b) or graphminspantree.

- Compute the MST.
- Compute $(\mu(L^*), \sigma(L^*))$ where L^* denotes the set of the edge lengths of the MST.

11.3 Characterization of various point patterns

1. Generate n condition 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?

11.4. 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
from scipy.sparse import csgraph # minimum spanning tree
```

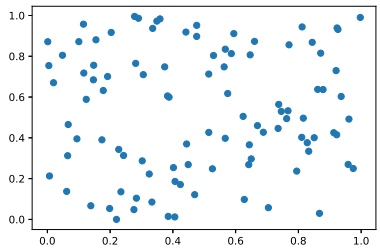
11.4.1 Random tessellations

Random tessellations are generated following normal standard and uniform distribution. A regular pattern is also employed. They are illustrated in Fig.11.2.

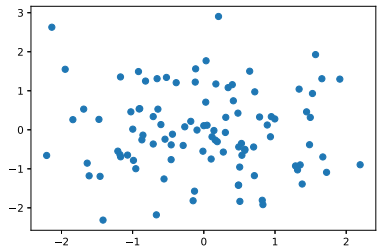
```
def dist_poisson(N=100):
    points = np.random.rand(N, 2)
    return points

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 11.2: Different point patterns.

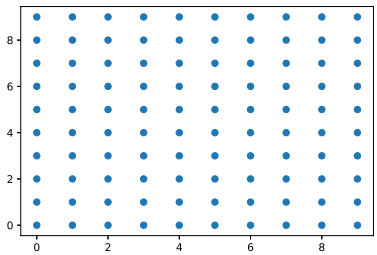
(a) Uniform distribution.



(b) Gaussian distribution.



(c) Regular distribution.



11.4.2 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;
```

11.4.3 Delaunay triangulation and minimum spanning tree

The Delaunay triangulation is computed with

```
tri = delaunay(points);
```

Then, the characterization of the triangulation is done by measuring the distances of the edges.

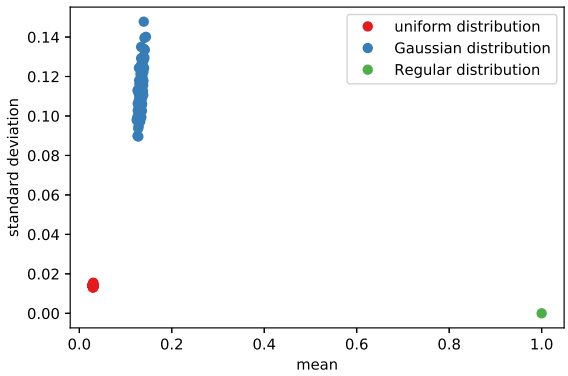
```
def characterization ( tri ):
    """
    Characterization of the Delaunay triangulation (mean and std dev of edges)
    """
    M = triToMat( tri );
    m = np.mean(M[M>0]);
    s = np.std(M[M>0]);
    return m, s;
def triToMat( tri , value =0.):
    """
    Transforms the triangulation into a matrix representation ,
    for simplicity
    """
    M = np.full ((tri.npoints, tri.npoints), value);
    d = distance.pdist(tri.points);
    distances = distance.squareform(d);
    for s in tri.simplices:
        M[s[0], s[1]] = distances[s[0], s[1]];
        M[s[1], s[2]] = distances[s[1], s[2]];
        M[s[2], s[0]] = distances[s[2], s[0]];
    return M;
```

11.4.4 Characterization of different realizations

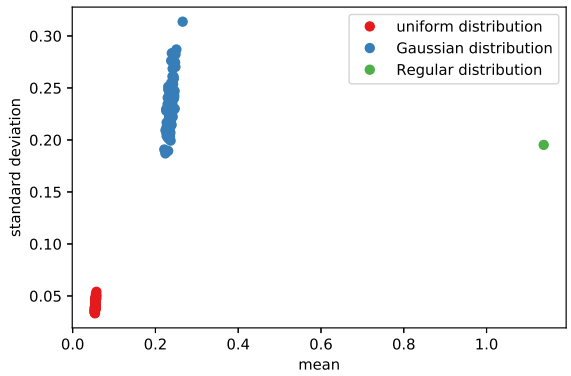
n realizations of the two distributions (uniform and Gaussian) are simulated. Then, the Voronoi diagram and the Delaunay triangulation are computed and characterized. The results are presented in Fig.11.3.

Figure 11.3: 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.

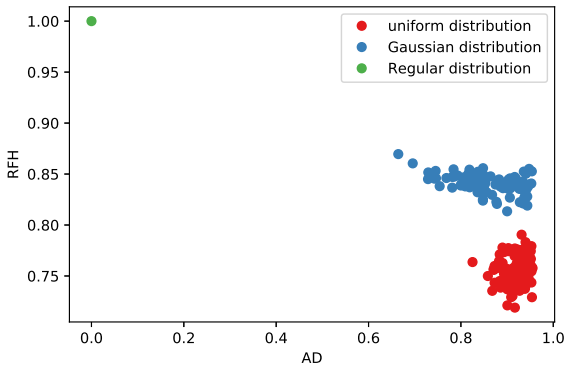
(a) Characterization by the mean and standard deviation of the lengths of the Delaunay triangulation.



(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.



12 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.

12.1 Corner detector and cornerness measure

Use imgradientxy and imgaussfilt with a scale parameter σ that will constrain the size of the window W.

Use the sobel and gaussian_filter from the scipy.ndimage module.

12.1.1 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.

12.1.2 Structure tensor

The structure tensor is defined by the following matrix. The coefficients ω follow a gaussian law, and each summation represents a gaussian filtering process. W is an operating window.

$$M = \begin{bmatrix} \sum\limits_{(u,v)\in W} \omega(u,v) I_x(u,v)^2 & \sum\limits_{(u,v)\in W} \omega(u,v) I_x(u,v) I_y(u,v) \\ \sum\limits_{(u,v)\in W} \omega(u,v) I_x(u,v) I_y(u,v) & \sum\limits_{(u,v)\in W} \omega(u,v) I_y(u,v)^2 \end{bmatrix} = \begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix}$$

- Evaluate M_1 to M_4 for each pixel of the image.

12.1.3 Cornerness measure

The cornerness measure C, as proposed by Harris and Stephens, is defined as follows for every pixel of coordinates (x, y):

$$C(x,y) = \det(M) - K\,\mathrm{trace}(M)^2$$

with K between 0.04 and 0.15.

Compute C for all pixels and display it for several scales σ.

12.2 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.12.1.

```
I = imread('sweden_road.png');
```

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 12.1: Sweden road sign to be used for corner detection.

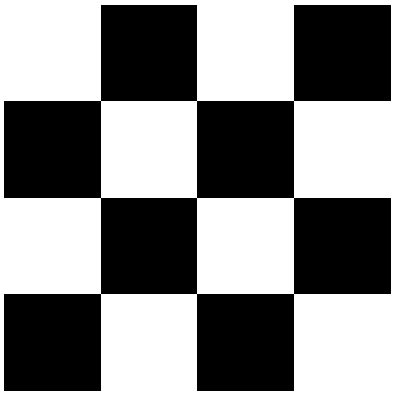
- Evaluate the extended maxima of the image.
- Only the strongest values of the cornerness measure should be kept. Two strategies can be employed in conjonction:
  - Use a threshold value t on C: the choice of this value is not trivial, and it strongly depends on the considered image. An adaptive method 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.

12.3. Python correction

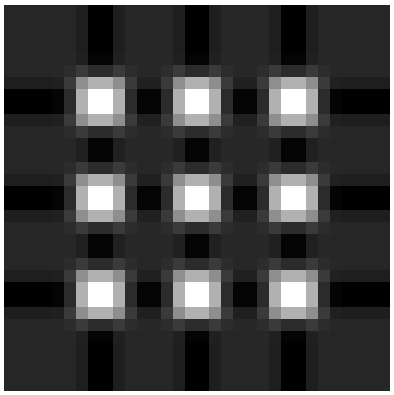
12.3.1 Cornerness measure

Figure 12.2: 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...

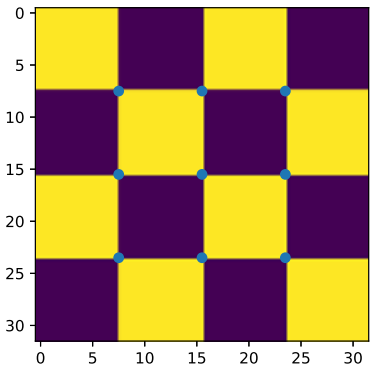
(a) Checkerboard.



(b) Cornerness measure.



(c) Harris corner points.



The first step is to compute the gradient in both x and y directions.

```
Ix = scipy.ndimage.sobel(I, axis=0);
Iy = scipy.ndimage.sobel(I, axis=1);
```

Then, the coefficients of the matrix are computed.

```
M1 = np.multiply(Ix, Ix);
M2 = np.multiply(Iy, Ix);
M4 = np.multiply(Iy, Iy);
```

In case of using a scale parameter, these coefficients should be filtered (for example via a gaussian filter).

```
M1 = scipy.ndimage.gaussian_filter(M1, sigma);
M2 = scipy.ndimage.gaussian_filter(M2, sigma);
M4 = scipy.ndimage.gaussian_filter(M4, sigma);
```

Finally, the cornerness measure is evaluated.

```
C = (np.multiply(M1, M4) - np.multiply(M2, M2)) - K * np.multiply(M1+M4, M1+M4);
```

The cornerness measure is displayed in Fig.12.2.

12.3.2 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.

```
corners = peak_local_max(C, indices=False, min_distance=2);
```

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.12.2c.

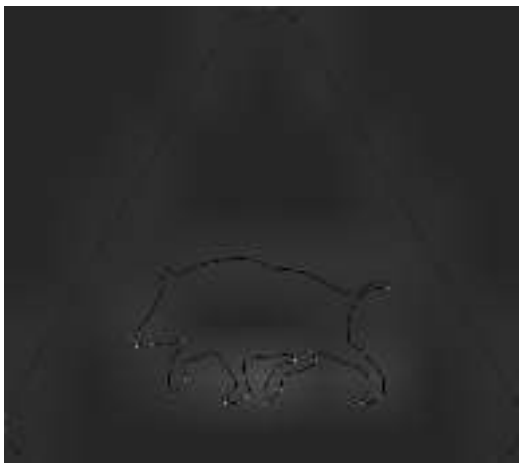
```
L = measure.label(corners);
props = measure.regionprops(L);
centers =[];
for prop in props:
    centers.append(prop.centroid);
```

12.3.3 Road sign image application

In this case, the values $t = 10^7$ and $\sigma = 3$ are used. The result is illustrated in Fig.12.3.

Figure 12.3: Harris corner detection with scale $\sigma = 3$ and threshold value $t = 10^7$.

(a) Cornerness measure.



(b) Corner points.

13 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

13.1 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.

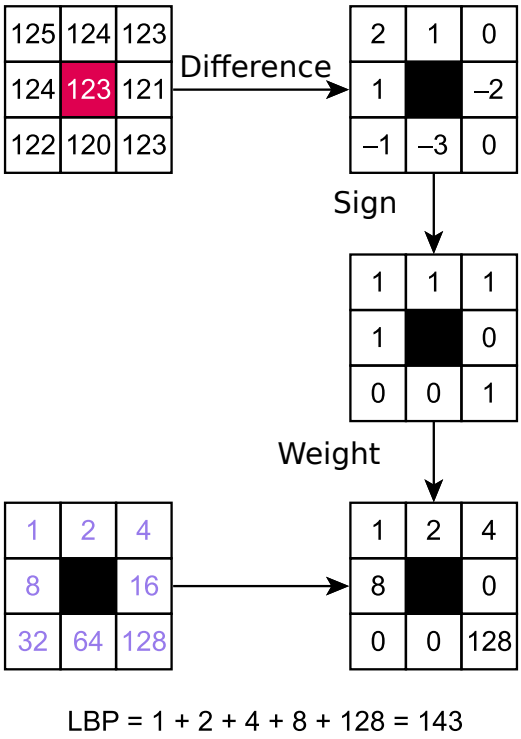


Figure 13.1: Local binary pattern. From wikipedia, author Xiawi, CC-By-SA.

Consider the function hiscounts for histogram computation.

Consider the function numpy.histogram for histogram computation.

13.2 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.

See KMeans from sklearn . cluster .

13.3. Python correction

The following imports are used.

```
import numpy as np
from scipy import misc
import matplotlib.pyplot as plt
import glob
import seaborn as sn
import pandas as pd
import os
from sklearn.cluster import KMeans
```

13.3.1 LBP computation

Each pixel is given a specific 8 bits value according to a code as follows.

```
def LBP(I):
    B = np.zeros(np.shape(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.

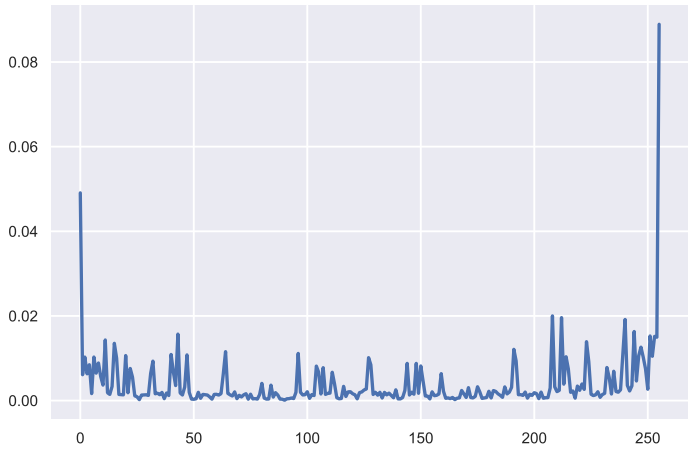
```
h,edges = np.histogram(B[1:-1, 1:-1], density=True, bins=256);
```

For the first image of sand, the histogram is shown in Fig.13.2.

Figure 13.2: Illustration of the Local Binary Pattern computed on an entire image.

(a) Texture image.

(b) LBP of texture.



13.3.2 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.13.3). 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);
        hh.append(h);
```

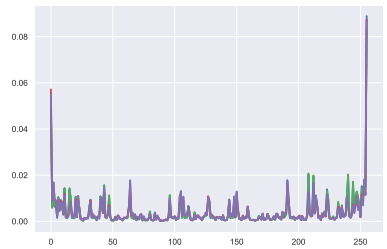
Figure 13.3: Illustration of the LBP of 4 images of each family. The histogram are almost equivalent, which shows that this descriptor can be employed to discriminate between the different families.

(a) Metal image example.

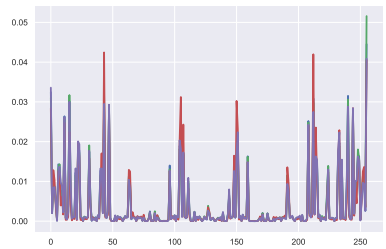
(b) Sand image example.

(c) Terrain image example.

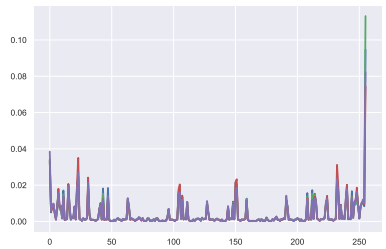
(d) Four metal images.



(e) Four sand images.



(f) Four terrain images.



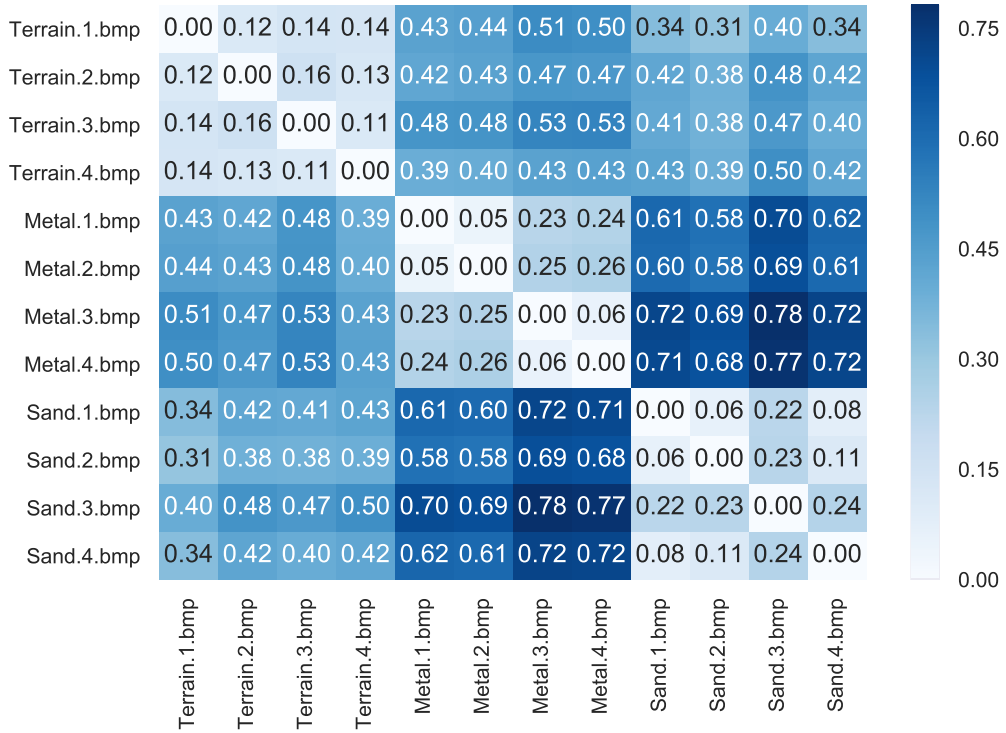
A distance criterion is used to compare the different histograms: the classical SAD (Sum of Absolute Differences) gives a numerical values. All pairs of distances are concatenated in a matrix, displayed as an image in Fig.13.4.

```
# 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.

```
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 13.4: 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, and we can verify that the clustering process works as expected. The result is presented in the next box.

```
# 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.

```
[1 1 1 1 0 0 0 0 2 2 2 2]
```

14 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:

14.1 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:

$$\eta_T(X) = \{x, T^1_x \subseteq X, T^2_x \subseteq X^c\} \quad (14.1)$$
$$= \epsilon_{T^1}(X) \cap \epsilon_{T^2}(X^c) \quad (14.2)$$

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).

14.2 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:

$$\theta_T(X) = X \backslash \eta_T(X) \quad (14.3)$$
$$\chi_T(X) = X \cup \eta_T(X^c) \quad (14.4)$$

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.

14.3 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.

14.4 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:

$$S(X) = \bigcup_r \epsilon_{B_r(0)}(X) \backslash \gamma_{B_1(0)}(\epsilon_{B_r(0)}(X)) \tag{14.5}$$

1. Implement this transformation.
2. Test this operator and compare it with the topological skeleton.

14.5. Python correction

```
from scipy import ndimage
import numpy as np

import imageio
import matplotlib.pyplot as plt
```

14.5.1 Hit or miss transform

The hit-or-miss transform is illustrated in Fig.14.1.

```
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 14.1: Elementary functions

(a) Hit or miss (intensities are inverted).

(b) Thinning.

(c) Thickening.

14.5.2 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;
```

14.5.3 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;
```

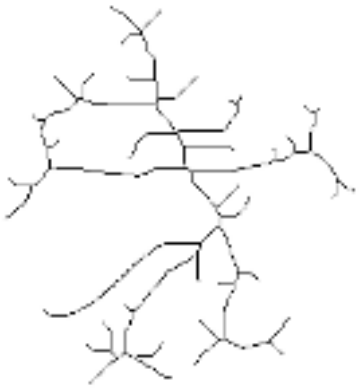
The topological skeleton is the iteration of the thinning with structuring elements in all 8 directions. It has the property of preserving the topology of the discrete structures, contrary to the morphological skeleton (see Fig.14.2). The morphological skeleton does not preserve the connexity of the 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 14.2: 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.

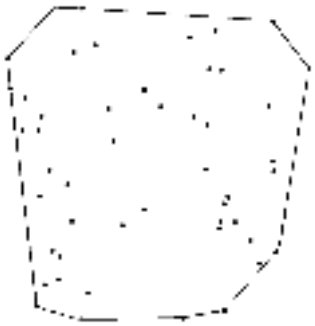


(b) Morphological skeleton.

15 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 15.1: Convex Hull example.



15.1 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.

15.1.1 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 min function.

Use the numpy.lexsort function.

15.1.2 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).

15.1.3 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.

15.1.4 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 $\overrightarrow{P_iP_{i+2}}$.

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.

15.1.5 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.

15.2. Python correction

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

15.2.1 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
pp =points[:-1,:]
```

Then, the points are sorted by the cosinus of the angle. This step has complexity $O(n \log n)$.

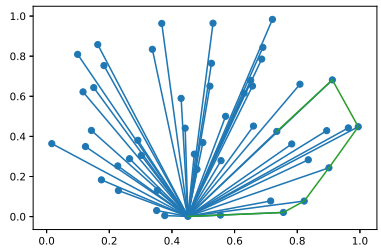
```
# sort all points by angle
hypothenuse=np.sqrt( (pp[:,0]-P[0])**2 + (pp[:,1]-P[1])**2 );
adj_side = pp[:,0] - P[0];
# as cos is decreasing, we use minus
cosinus= -adj_side/hypothenuse;
ind=cosinus.argsort();

# 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.15.2).

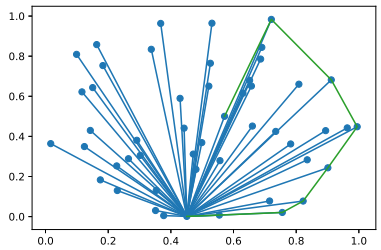
```
first = list_points.pop(0);
second= list_points.pop(0);
hull = []; # convex hull
hull.append(first);
hull.append(second);

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 15.2: Graham scan illustration while constructing the convex hull.

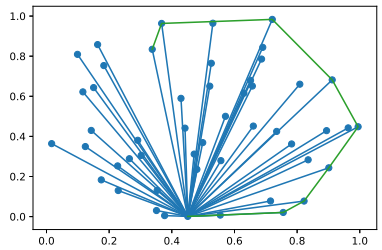
(a) After 10 tested points.



(b) After 20 tested points.



(c) After 30 tested points.



15.2.2 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.

```python
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');
```

15.2.3 Simple tests

For 5 points:

```python
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');
```

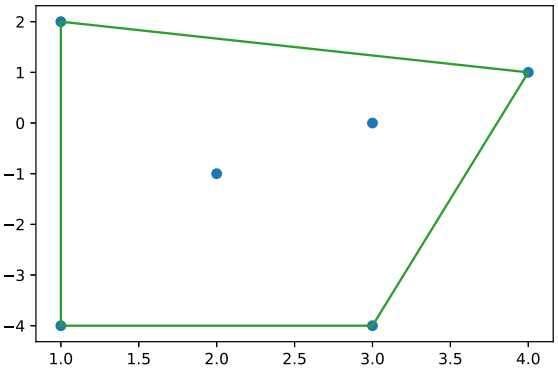
For a few random points:

```python
nb=50;
Points = np.random.rand(nb, 2);
H = conv_hull(Points);
displayPointsAndHull(Points, H, 'random_hull.python.pdf');
```

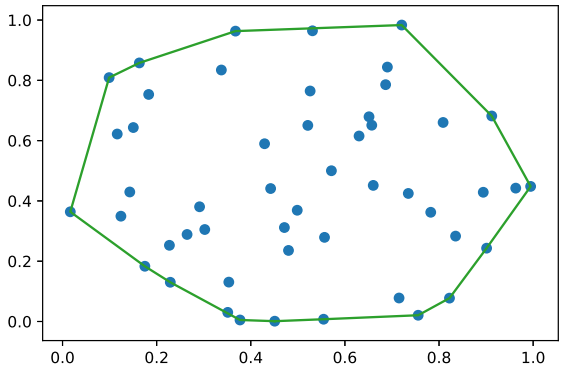
The results are illustrated in Fig.15.3.

Figure 15.3: Illustration of the convex hull computation.

(a) Convex points of 5 points.



(b) Convex hull of 50 random points.



16 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).

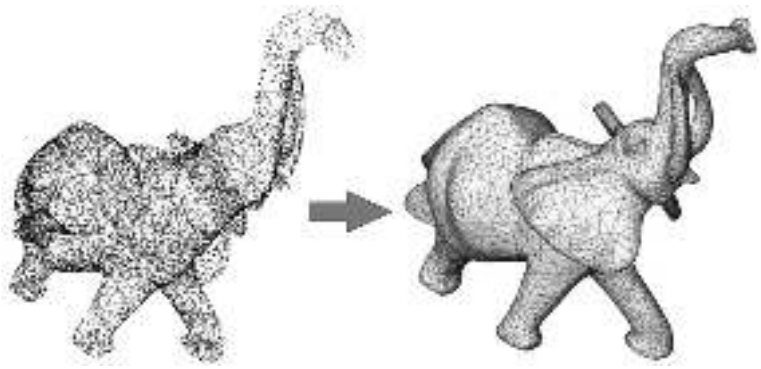


Figure 16.1: Shape reconstruction from a set of points.

16.1 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

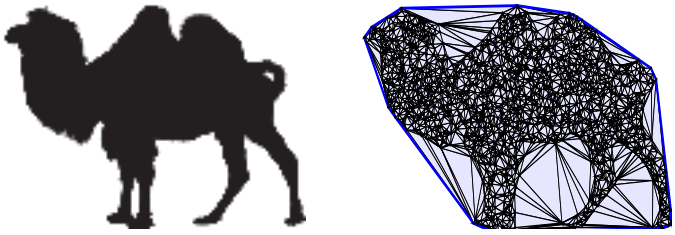
(b) point pattern representing the shape

Figure 16.2: 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.

16.2 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

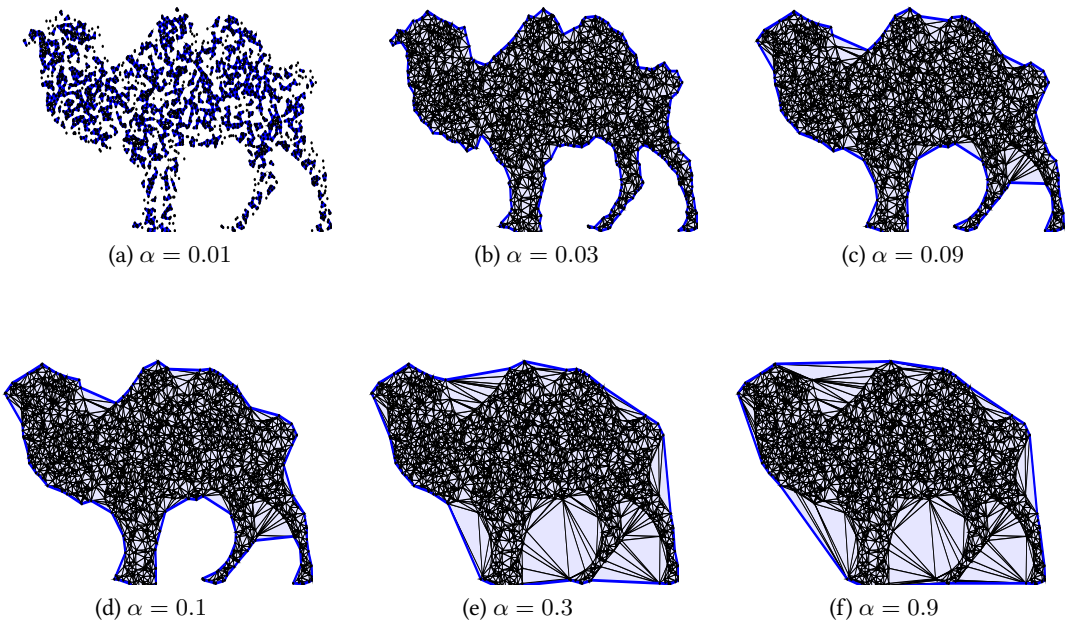
Figure 16.3: Initial point pattern and its Delaunay triangulation.

1. By using the initial point pattern, build its Delaunay triangulation.
2. Look at the resulting Matlab object to understand the structure of the triangulation.

You can use the matlab function delaunayTriangulation.

16.3 Alpha-shape

The alpha-shape corresponds to the union of Delaunay triangles T_{ijk} such that the circumradius C_{ijk} is lower than α.



(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 16.4: Alpha-shapes for different values of α.

1. Implement the algorithm.
2. Test this operator with the input data using different values of α.

16.4. 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
```

16.4.1 Point pattern

The image is first loaded.

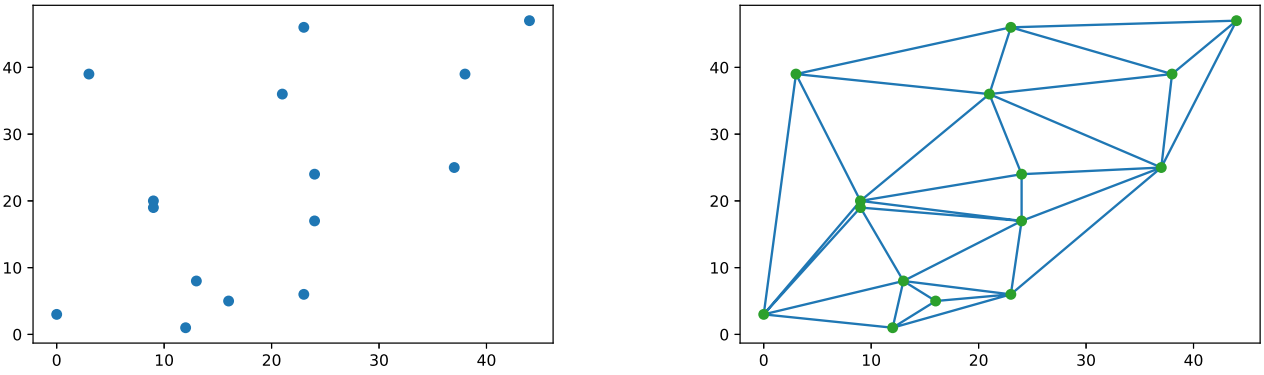
```
A = imread('camel.png')
m,n = A.shape
```

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.16.5.

```
# 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()
```



(a) Set of points, uniformely chosen in the shape.

(b) Delaunay triangulation.

Figure 16.5: Set of points and its Delaunay triangulation.

16.4.2 Delaunay triangulation

The Delaunay triangulation is simply obtained by the following code. The result is presented in Fig.16.5.

```
tri = Delaunay(points)

# Display result
plt.triplot(points[:,0], points[:,1], tri.simplices, lw=.5)
plt.show()
```

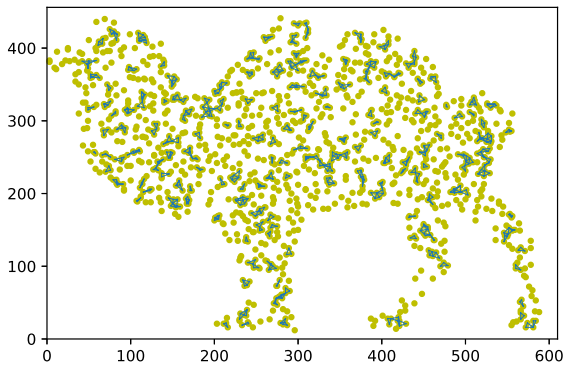
16.4.3 Alpha-solid

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.

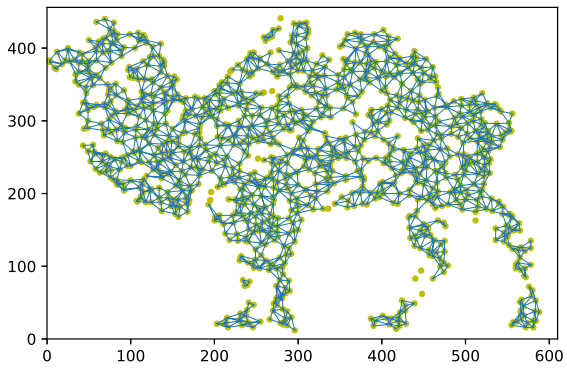
```
from sympy.geometry import Triangle
radius=[]
import progressbar
count=0

# 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.16.6.

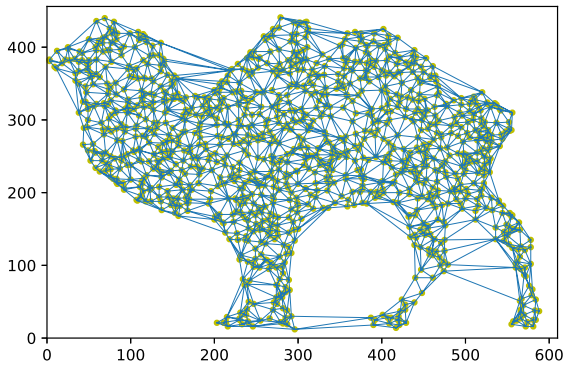
```
or R in progressbar.progressbar([5,10, 50, 100, 100000]):

    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()
```



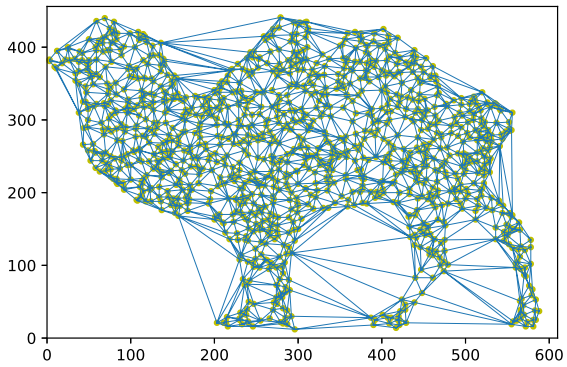
(a) $\alpha = 1/5$.



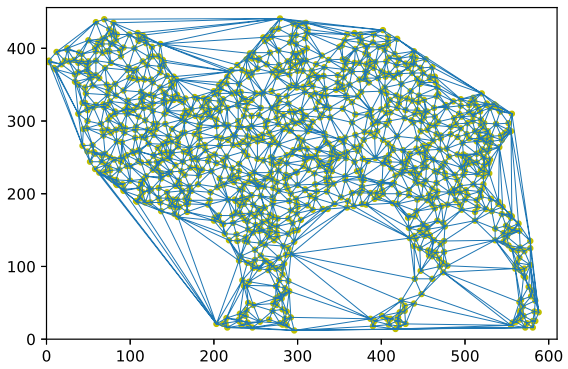
(b) $\alpha = 1/10$.



(c) $\alpha = 1/50$.



(d) $\alpha = 1/100$.



(e) $\alpha = 1/100000$.

Figure 16.6: Different alpha-solids.