MATHEMATICAL MORPHOLOGY EXERCISES With MAMBA

(Release 2 - Solutions included)

Serge BEUCHER

with contributions by

Nicolas BEUCHER

Here is a copy of the license of MAMBA. This license is known as the X11 license (also named MIT license).

TABLE OF CONTENTS

INTRODUCTION

This document is the second release of a Mathematical Morphology (abbreviated MM) exercises book using the MAMBA Image library. It is the adaptation to this library of a former handbook written fifteen years ago which came with the Micromorph software. Some exercises have been removed, some new ones have been added, these latter ones being taken from the examples given in the MAMBA web page and the user manual.

Compared to the Micromorph release, the philosophy of these new exercises is somewhat different as the reader is not encouraged to design the various morphological operators contained in the library. This task would certainly be too boring and not really very useful to understand how these operators work and what is their purpose. Therefore, we simply give some hints and suggestions regarding the choice, among all the available operators, of those which could be relevant to solve the proposed problems.

Note also that this handbook is not a Mathematical Morphology course. It is divided into different chapters, each one devoted to a given class of operators. Although some notions and definitions are given at the beginning of each chapter, their purpose is only to provide a quick reminder. Moreover, they are defined in a very simplistic way. The reader is invited to refer to the bibliography given at the end of each chapter to go deeper into the concepts and tools introduced in the exercises.

The latest MAMBA release (version 2.0) is used for the solutions of the exercises. Compared to the previous versions, some differences exist. However, it should not be difficult to adapt these solutions to the previous versions (1.1) if necessary. The reader is invited to get and to read the various documentation coming with the MAMBA library in order to benefit from these exercises.

No doubt that many flaws and errors still appear in this document. My apologies for this.

I would like also to thank Nicolas Beucher for his major involvement in the design of the MAMBA software (versions 1 and 2), the writing of the documentation, the examples, the MAMBA web page, etc.

Fontainebleau, June 30, 2016

BEFORE YOU START

1 Installing and using the MAMBA library

All the exercises of this handbook use the MAMBA Image library. Therefore, you must install this library before starting. It can be found at the MAMBA web page:

http://www.mamba-image.org

Reading the MAMBA Image Library User Manual, which can also be found there, is strongly recommended. You will find also other interesting documents, in particular the MAMBA Image Library Python Quick Reference, which lists all the operators of the library and the MAMBA Image Library Python Reference if you need a more detailed information about these operators.

Remember also that MAMBA is not a programming language but simply a Mathematical Morphology library for Python. Therefore, you will need also to have a basic knowledge of this language to use efficiently this handbook.

Many exercises require only that you launch MAMBA in a Python shell. If you are under Windows, you can simply launch the MambaShell (IDLE shell) installed with MAMBA. It automatically defines some images and display them, which allows you to start immediately. If you are under Linux, this shell can be used too but you need to launch it manualy. Then the exercises can be solved by entering successive commands in the shell. Some exercises however require the definition of more sophisticated operators which are not already present in the MAMBA library. When it is the case, these operators have been gathered in a Python module named Mamba_solutions.py which can be found here:

http://cmm.ensmp.fr/~beucher/stockage/Mamba_solutions.py

Finally, some exercises use other Python libraries to achieve specific tasks: curve drawings with matplotlib, image calculations with scipy/numpy, 3D displays with VTK, etc. The necessary instructions for using them will be indicated everytime they are required.

2 Organisation of the exercises

As said before, each chapter is devoted to a specific class of operators. A short reminder is followed by the exercises. These exercises are generally briefly presented in order to let you think about it and try to find a solution by your own. The difficulty level of each exercise is given by the following quotation (mamba snake!):


means that the exercise is quite easy and immediate.


means that the exercise is at an intermediary level of difficulty. You will have to combine several tools and operators to get the solution.

 means that the exercise is quite difficult (at least, I think so!). Such an exercise can be considered as a real image analysis application.

We have limited the number of difficulty levels to three, but some exercises in this category would deserve to have a quotation with more than three snakes...

In order to prevent you to jump directly to the solutions, these solutions are postponed to the end of each chapter. Please, try to find a solution by yourself before looking at the proposed one, because, as it is often the case in image analysis, this solution is seldom unique and many tracks are available to reach it. Therefore, I do not pretend to give you the best one...

3 The image database

During these exercises, you can use the images which are provided in the Mamba_Images database (you can obviously use your own images). This database is available here (zip file):

http://cmm.ensmp.fr/~beucher/stockage/Mamba_Images.zip

Download and unzip it at the location of your choice. Five sub-directories are available: ***bin*** contains binary images, ***grey*** contains greyscale images (8-bit), ***color*** (self explanatory), ***32bit*** contains 32-bit images and ***3D*** contains few 3D images and sequences (which are considered as 3D images by MAMBA). Once copied on your disk, you are advised to define the Python working directory to avoid long and boring path entries while manipulating these images. For instance, on Windows, assuming that the database has been unzipped at the root of disk C:/, you have to enter the following commands:

```
import os
os.chdir("c:/Mamba_Images")
```

Adapt these commands if you are working with Linux.

Note also that the binary images used in the exercises are inverted in the illustrations. A white pixel in the image is displayed in black in the manual, for aesthetic purposes.

4 List of references

You will find at the end of each chapter a list of references. As mentioned before, the operators and concepts used in the exercises are, most of the time, introduced very rapidly. Therefore, this list will allow you to gain insight into these tools. These references have been chosen because they are freely and easily available in the Web. Obviously, you can also dig into the abundant bibliography available here and there.

Chapter 1

BASIC NOTIONS

In this chapter are reviewed the various notations used throughout this book, together with some elementary definitions and image operators. This should allow you to get familiar with many helpful operators for image handling and display.

1 Notations

Sets represent binary images, functions (from $\mathbb{R}^2$ into $\mathbb{R}$) represent 2D greytone images.

1.1 Sets

Sets are generally denoted with capital letters, X, Y, Z. X_i is the i-th connected component of the set X.

1.2 Structuring elements

The capital letters B, H, L, M, T, etc... are used for the structuring elements. T_1 , T_2 are the two components of a two-phase structuring element $T = (T_1 , T_2)$.

1.3 Functions

Functions are generally denoted with small letters, f, g, h, etc. .

1.4 Set and function transforms

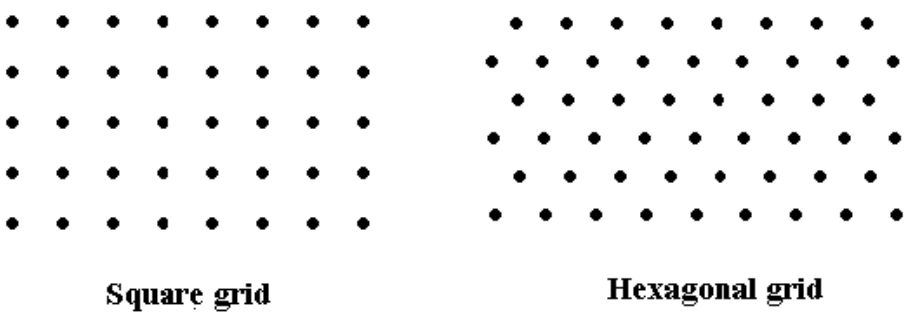
The greek capital letters Ψ, Φ etc., denote transforms. Φ(X) (Φ(f) respectively) is the result of the transformation Φ applied to the set X (to the function f respectively).

1.5 Sub-graph

The sub-graph or umbra of a function f from $\mathbb{R}^2$ into $\mathbb{R}$ is denoted by U(f) and represents the set of the points (x,y) of $\mathbb{R}^2 x \mathbb{R}$ such that $y \leq f(x)$.

2 Definitions

2.1 Raster

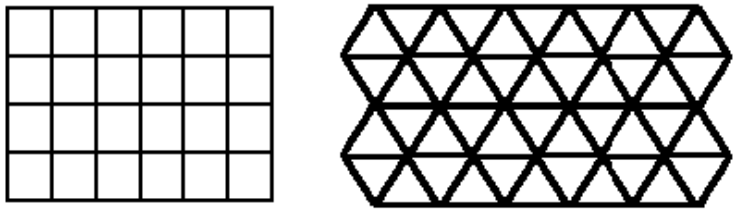


Square grid Hexagonal grid

$\mathbb{N}$ generally denotes the integer set (i.e. positive, negative numbers or zeros), $\mathbb{N}^2$ the set of ordered pairs of two integers. $\mathbb{N}^2$ is called a raster, and its elements are the pixels of a 2D image.

2.2 Grid, neighborhood graph

A graph is defined by a set of points which are called the vertices of the graph, and a set of pairs of points taken among the vertices and called the edges of the graph.
A 2D grid is a graph:
- whose vertices belong to $\mathbb{N}^2$
- which is invariant under translation in $\mathbb{N}^2$
- and where the segments joining every edge extremity cannot cross each other.



Square graph **Hexagonal graph**

Apart from a few exceptions, the exercises dealing with 2D images use the hexagonal grid (this corresponds to the default setup in MAMBA).

2.3 Complementation

2.3.1 Binary images

X^c is the complementary set of X. Any point x which is not included in X belongs to X^c.

2.3.2 Greytone images

Image complementation is defined by:

$$f^c = MAX - f$$

where f is the original image and MAX is the maximum grey level that can be used according to the MAMBA image depth (255 for 8-bit images, $2^{32} - 1$ for 32-bit images).

2.4 Union, intersection

2.4.3 Binary images

$X \cup Y$ represents the union of the two sets X and Y, that is the set of all the points that belongs to X or to Y. $X \cap Y$ represents the intersection of the two sets X and Y, that is the set of all the points that both belong to X and Y.

2.4.4 Greytone images, Sup, Inf

Sup(f,g), also denoted $f \vee g$, designates the sup of two functions f and g: for every point x, $(f \vee g)(x)$ is the highest of the two values f(x) and g(x). Inf(f,g), also denoted $f \wedge g$, designates the inf of two functions f and g: for every point x, $(f \wedge g)(x)$ is the smallest of the two values f(x) and g(x).

3 Some relations

3.1 De Morgan's formulae

These formulae express the duality of union and intersection:

$$(X \cup Y)^c = X^c \cap Y^c \text{ and } (X \cap Y)^c = X^c \cup Y^c$$

Similarly:

$$(f \vee g)^c = f^c \wedge g^c \text{ and } (f \wedge g)^c = f^c \vee g^c$$

3.2 Difference, symmetrical difference

X\Y denotes the set difference of X and Y. It is the set of those points belonging to X and not to Y.

$$X \backslash Y = X \cap Y^c$$

X*Y denotes the symmetrical set difference. It is the set of the points that belong to one and only one of the two sets X and Y.

$$X * Y = (X \cap Y^c) \cup (Y \cap X^c) = (X \cup Y) \backslash (X \cap Y)$$

3.3 Commutativity, associativity, distributivity

Union, symmetrical difference and intersection are commutative and associative operations. The same properties apply to sup and inf.

Commutativity:
$$X \cup Y = Y \cup X$$
$$f \vee g = g \vee f$$

Associativity:
$$(X \cup Y) \cup Z = X \cup (Y \cup Z) = X \cup Y \cup Z$$
$$(f \vee g) \vee h = f \vee (g \vee h)$$

Union and intersection are mutually distributive:
$$(X \cup Y) \cap Z = (X \cap Z) \cup (Y \cap Z)$$
$$(X \cap Y) \cup Z = (X \cup Z) \cap (Y \cup Z)$$

and so are sup and inf:
$$(f \vee g) \wedge h = (f \wedge h) \vee (g \wedge h)$$
$$(f \wedge g) \vee h = (f \vee h) \wedge (g \vee h)$$

4 Other definitions

You will find below a quick reminder of some definitions which will be useful all along the exercises.

4.1 Increasing transformations

A transformation Ψ is said to be increasing if it satisfies:

$$X \subset Y \Rightarrow \Psi(X) \subset \Psi(Y)$$
$$f \leq g \Rightarrow \Psi(f) \leq \Psi(g)$$

4.2 Extensivity/Anti-extensivity

Ψ is extensive if:

$$X \subset \Psi(X) \text{ or } f \leq \Psi(f)$$

Conversely, Ψ is anti-extensive if:

$$\Psi(X) \subset X \text{ or } \Psi(f) \leq f$$

4.3 Idempotence

Ψ is idempotent if:

$$\Psi[\Psi(X)] = \Psi(X) \text{ or } \Psi[\Psi(f)] = \Psi(f)$$

4.4 Duality by complementation

Φ and Ψ are both dual transformations if:

$$\Phi(X) = [\Psi(X^c)]^c \text{ or } \Phi(f) = [\Psi(f)^c]^c$$

EXERCISES

Exercise n° 1

This exercise aims at getting familiar with the MAMBA library. Perform the following operations:
- Load images into memory (use the various ways available in MAMBA to do this).
- Display images, change the display palettes, use the superposer, etc.
- Save images.
- Get image information (size, depth, etc.).
- Use the interactive thresholder tool.

To achieve this, use the following MAMBA operators: **load**, **show** and its various attributes, **hide**, **superpose**. Use also the **imageMb** constructor and its corresponding methods: **getSize**, **getDepth**.

Exercise n° 2

Prove that the function operators inf and sup can be expressed as set operations on sub-graphs. (Prove in particular that the intersection of the sub-graphs f and g is the sub-graph of the inf of f and of g).
Verify this with MAMBA (take two grey scale images and use the **threshold** and logic operators).

Exercise n° 3

1) Prove and verify the De Morgan's formulae.
2) A transformation Ψ is defined as follows:

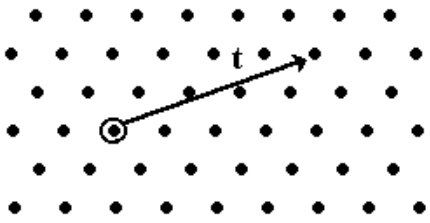
$$\Psi(X) = X \cup Y, \text{ Y fixed set}$$

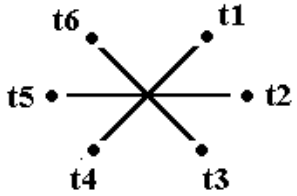
- Is this transformation increasing, extensive, idempotent?
- Does there exist a dual transformation by complementation, and if so, indicate it.

3) Practice the corresponding MAMBA operators (**logic**, **diff, negate**).

Exercise n° 4

On the hexagonal grid each point has six neighbors. Translations are easily defined on this grid.





Each translation is composed of elementary ones, e.g.:

$$t = t_1 \circ t_1 \circ t_2 \circ t_2 \circ t_2$$

The set of translations equipped with the composition law constitutes the group of displacements.
Practice the different shift operators available in MAMBA (**shift**, **shiftVector**). Do these operators satisfy this group structure? Verify and explain your answer. (This phenomenon is the first occurrence of the edge effects resulting from the fact that we work on a finite field of analysis).

As a reminder, edge effects are due to the fact that, by default, MAMBA assumes that pixels liable to fall outside the field are lost. When re-entering these pixels (by means of a translation in another direction), they are given an arbitrary value. This behavior will have important consequences for the definition of geodesic transforms, but also for classical euclidean transforms. This will clearly appear in the following exercises.

SOLUTIONS

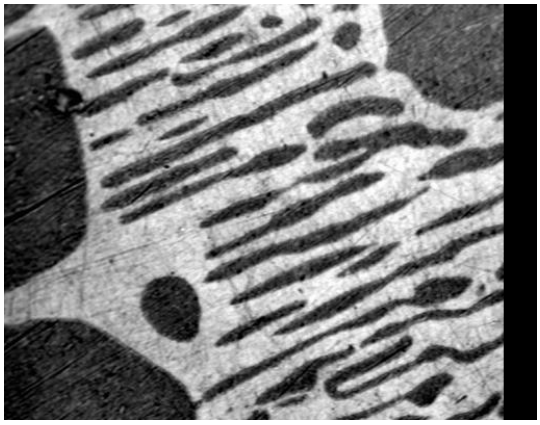
Exercise n° 1

Assuming that the image database has been properly downloaded and installed and that the working directory has been set up as described in the previous chapter, there are mainly three ways for loading images into MAMBA.
Firstly, you can create a MAMBA image and load an image file with a single operator, the **imageMb** constructor. For instance:

```
>>> imA = imageMb("grey/alliage.bmp")
```

defines a MAMBA image imA and loads the image file *alliage.bmp* into it.
You can display imA with the method **show**:

```
>>> imA.show()
```



Display of the image alliage. The image is padded with 0 (right stripe) to adjust its size (480x400) with the MAMBA image size (512x400).

The settings of the image imA are automatically defined to fit the characteristics of the image file:

```
>>> imA.getSize()
(512, 400)
>>> imA.getDepth()
8
```

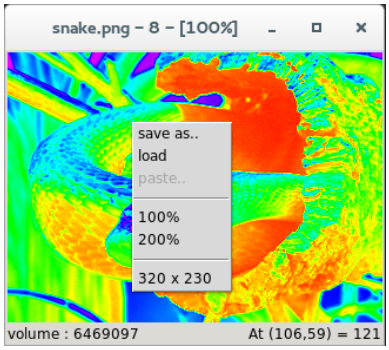
Note that the images defined by the **imageMb** operator have always a size multiple of (64x2). Therefore, if the size of the loaded image does not correspond exactly to the MAMBA format, the size of the MAMBA image is determined so that the loaded image is entirely contained in it (the image is padded with 0, as it can be shown with the *alliage* image whose actual size is 480x400).
You can also create a MAMBA image with a given size and depth, then load an image file into it. The image will be padded or cropped to adapt itself to the MAMBA image size:

```
>>> imB = imageMb(448, 320, 8)
>>> imB.load("c:/grey/alliage.bmp")
```

Loading the alliage image into a 448x320 MAMBA image. The original image is cropped to fit with the size of the MAMBA image.

The third way of loading an image consists in opening an image display. Then an image can be loaded by means of the "load" command in the display menu which appears when clicking in the display window. You can change the palette used for the display. For instance, to display image im1 with the **rainbow** palette, type:

```
>>> im1.show(palette="rainbow")
```



MAMBA display window with its popup menu.

Saving a MAMBA image can be achieved with the method **save**:

```
>>> im1.save("image1.png")
```

saves the content of image im1 into the png file image1.png in the working directory.

Saving an image can also be performed with the save command in the display window menu.
You can also display and save a combination of images: a greyscale one and a binary one or two binary images with the **superpose** operator. You must import this operator from the **mambaDisplay.extra** module:

```
>>> from mambaDisplay.extra import *
```

Load the *tools* image into im1, threshold it between 100 and 255 into imbin1 (these two images have already been defined if you use the MAMBA Python shell):

```
>>> threshold(im1, imbin1, 100, 255)
```

You can then superpose these two images:

```
>>> superpose(im1, imbin1)
```

The MAMBA image superposer.

Close the display window to exit the superposer.
Remember also that you can **hide**, **freeze** or **unfreeze** the display:

```
>>> im1.hide()
```

The im1 image is hidden. Restoring it can be achieved by simply showing it again.

Exercise n° 2

The solution is immediate:
Let us denote U(f), the sub-graph (sometimes called umbra) of f:
$U(f) = \{(x,y) : y \leq f(x)\}$
and U(g), the sub-graph of g. We have:

$$\begin{aligned}\forall (x,y), (x,y) \in U(f) \cap U(g) &\Leftrightarrow (x,y) \in U(f) \text{ and } (x,y) \in U(g) \\ &\Leftrightarrow y \leq f(x) \text{ and } y \leq g(x) \\ &\Leftrightarrow y \leq \inf(f(x), g(x)) \\ &\Leftrightarrow y \leq \inf(f,g)(x) \\ &\Leftrightarrow (x,y) \in U(\inf(f,g))\end{aligned}$$

Let $X_i(f)$ be a section at level I of the sub-graph of f:

$$X_i(f) = \{x : f(x) \geq i\}$$

$X_i(f)$ is the threshold of f between i and MAX of f. We have:

$$X_i(f \vee g) = X_i(f) \cup X_i(g)$$

$$X_i(f \wedge g) = X_i(f) \cap X_i(g)$$

Ley us verify this with an example. Load image *tools* in im1 and image *circuit* in im2 (both are greyscale images, MAX=255). Let us threshold these two images at level 150:

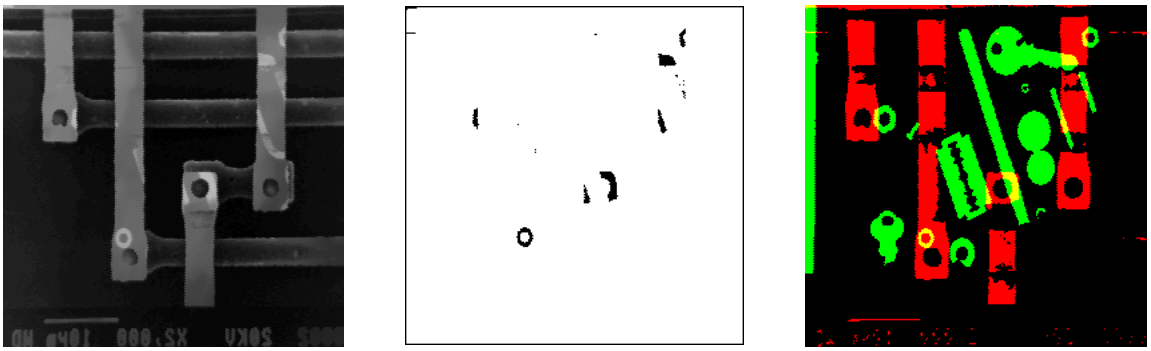
```
>>> threshold(im1, imbin1, 150, 255)
>>> threshold(im2, imbin2, 150, 255)
```

Perform the intersection of the two binary images with the **logic** operator:

```
>>> logic(imbin1, imbin2, imbin3, "inf")
```

Then perform the inf between the two greyscale images (with the same **logic** operator):

```
>>> logic(im1, im2, im3, "inf")
```



Inf of the two images tools and circuit (left), threshold at level 150 of the inf (middle) and display of the thresholds at level 150 of the two initial images (right). Their intersection, in yellow, is identical to the middle image.

We can threshold im3 at level 150 and put the result in imbin4:

```
>>> threshold(im3, imbin4, 150, 255)
```

The comparison between imbin3 and imbin4 shows that the two images are identical:

```
>>> compare(imbin3, imbin4, imbin4)
(-1, -1)
```

Exercise n° 3

1) De Morgan formulae
Let us prove that :

$$(X \cap Y)^c = X^c \cup Y^c$$

Let x be a point that belongs to the complementary set of $X \cap Y$, x does not belong to $X \cap Y$. But, if x does not belong to both X and Y, it means that it does not belong to one of them at least: $x \notin X$ or $x \notin Y$. Q.E.D.
The same property holds for functions with the sup, inf and negate operators.
2) Let $\Psi(X) = X \cup Y$, Y is fixed.
Ψ is increasing:

$$\forall X_1 \subset X_2,\ \Psi(X_1) = X_1 \cup Y \subset X_2 \cup Y = \Psi(X_2)$$

Ψ is extensive (obvious).

Ψ is idempotent:

$$\Psi[\Psi(X)] = (X \cup Y) \cup Y = X \cup Y = \Psi(X)$$

If there exists a dual transformation Φ of Ψ, it satisfies:

$$[\Psi(X^c)]^c = \Phi(X), \text{ that is:}$$

$$\Phi(X) = (X^c \cup Y)^c = X \cap Y^c$$

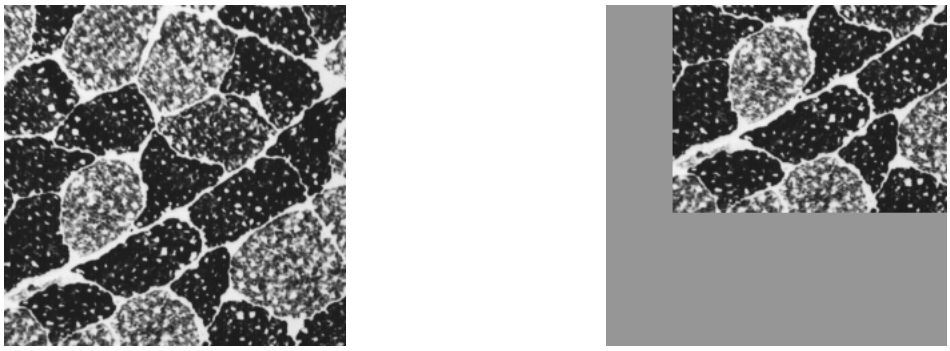
3) Beside the **logic** operator, **negate** allows to complement a binary image or to invert a greyscale one. The operation $X \cap Y^c$ can be perform with **diff**. Note that **diff** can also be used with greytone images but with a slightly different behavior however (see the MAMBA reference manuals).

Exercise n° 4

Translation in a 2D or 3D space is obviously an operation satisfying group properties. In paricular, translating a point in any direction and translating it again at the same distance in the opposite direction does not change anything. Unfortunately, it is not the case when working in a finite space as a point falling outside the image window is definitely lost. This is a very annoying edge effect which is constantly present when designing morphological operators and which must be taken into account to avoid biases and errors in the implementation of operators.
Two operators exist to shift images in MAMBA: **shift** (**shift3D**) and **shiftVector** (which has no corresponding 3D operator yet).
shift performs image shiftings of any size in a given direction of the grid in use. For instance:

```
>>> shift(im1, im2, 1, 100, 150, HEXAGONAL)
```

shifts (size 100) image im1 in direction 1 in the hexagonal grid and puts the result in image im2. The third parameter (fill) indicates that the outside “virtual” pixels which are entering the image window have the value 150.

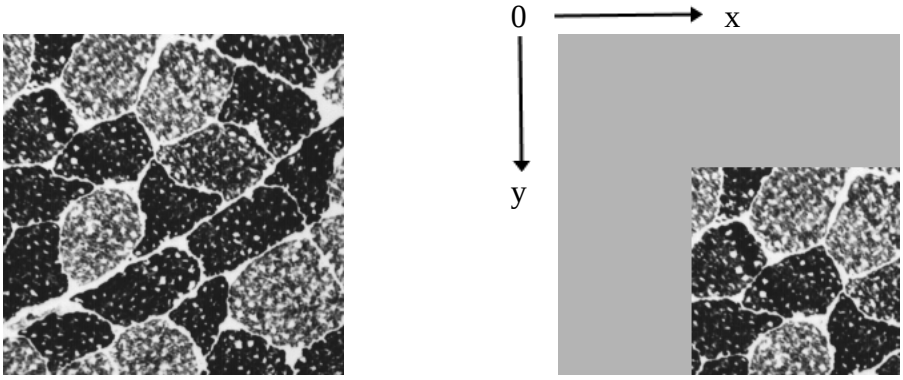


Shifting (right) of size 100 of the left image. The pixels which are propagated inside the image have a grey value equal to 150.

shiftVector is a vector-based shifting operator. In this case, the image is simply considered as an array of pixels and is shifted by a vector defined by its coordinates contained in a tuple. We have:

```
>>> shiftVector(im1, im2, (100, 100), 180)
```

This command performs a shift of image im1 to image im2 by a vector (100, 100). As for **shift**, the value which is propagated inside the shifted image is defined by the parameter 180 (fill).



Left image shifted by a vector (x, y) of size (100, 100). The shifted image is filled with pixels valued at 180.

REFERENCES

[1] N.Beucher, S.Beucher: Mamba Image Library User Manual
(**http://mamba-image.org/docs/2.0/mamba-um.pdf**)
This document contains all you need to get started, install, compile and understand how to use MAMBA in the best possible way.

[2] N.Beucher: Mamba Image Library Python Reference
(**http://mamba-image.org/docs/2.0/mamba-pyref.pdf**)
It is a complete description of all the functions of the MAMBA library. It will help you understand how/when to use the operators that already exist in the library (so you don't have to recode them by yourself).

[3] N.Beucher: Mamba Image Library Python Quick Reference
(**http://mamba-image.org/docs/2.0/mamba-pyquickref.pdf**)
It is a fast and handy list of all the functions of the library plus some quick reminders that are always needed when coding with MAMBA. This document and the previous one should always be on hand while using this exercise book.

Chapter 2

EROSIONS & DILATIONS

1 Erosions, dilations, reminder

1.1 Binary case

The dilation of a set X by a set B of center O called a structuring element is a set Y thus defined:

$$Y = X \oplus \check{B} = \{x \ : B_x \cap X \neq \emptyset \ \}$$

where $B_x = \left\{x + \overrightarrow{Ob}, b \in B\right\}$ is the translation of the structuring element B at point x.

$(X \oplus \check{B})$can also be written:

$$X \oplus \check{B} = \bigcup_{b \in \check{B}} X_{\overrightarrow{Ob}}$$

$$X \oplus \check{B} = \{z \ : z = x + y \ , \ x \in X \ , \ y \in \check{B} \ \}$$

$\check{B}$ is the transposed set of B (i. e. the symmetrical set of B with respect to O). The dilated set Y is then the union of the translates of X.

Similarly, the eroded set is defined by:

$$Z = X \ominus \check{B} = \{x \ : B_x \subset X\}$$

which is also written:

$$X \ominus \check{B} = \bigcap_{b \in \check{B}} X_{\overrightarrow{Ob}}$$

These two transformations have the following properties:

$$\begin{aligned} (X \cup Y) \oplus \check{B} &= (X \oplus \check{B}) \cup (Y \oplus \check{B}) \\ (X \cap Y) \ominus \check{B} &= (X \ominus \check{B}) \cap (Y \ominus \check{B}) \\ (X \oplus \check{B}_1) \oplus \check{B}_2 &= X \oplus (\check{B}_1 \oplus \check{B}_2) \\ (X \ominus \check{B}_1) \ominus \check{B}_2 &= X \ominus (\check{B}_1 \oplus \check{B}_2) \end{aligned} \qquad (1)$$

which hold, whichever are the centers of B, B_1 et B_2 .

1.2 Greytone case

The sub-graph U(f) of a function f can be eroded and dilated by a three-dimensional structuring element B and it can be shown, under certain conditions, that the dilation (resp. the erosion) of a sub-graph is still the sub-graph of a function, called the dilation (resp. the erosion) of f by B and denoted $f \oplus \check{B}$ (resp. $f \ominus \check{B}$).

If the structuring elements are planar, the dilation of a function f can be written:

$$(f \oplus \check{B})(x) = \sup_{b \in \check{B}} \left(f\left(x + \overrightarrow{Ob}\right)\right)$$

or else:

$$f \oplus \check{B} = \sup_{b \in \check{B}} f_{\overrightarrow{Ob}}$$

where $f_{\overrightarrow{Ob}}$ is the translated function of f of vector $\overrightarrow{Ob}$.

Similarly, the erosion is defined by:

$$(f \ominus \check{B})(x) = \inf_{b \in \check{B}} \left(f\left(x + \overrightarrow{Ob}\right)\right)$$

or else:

$$f \ominus \check{B} = \inf_{b \in \check{B}} f_{\overrightarrow{Ob}}$$

The relations (1) are immediately transposable to functions.

$$\begin{aligned} (f \vee g) \oplus B &= (f \oplus B) \vee (g \oplus B) \\ (f \wedge g) \ominus B &= (f \ominus B) \wedge (g \ominus B) \\ (f \oplus B_1) \oplus B_2 &= f \oplus (B_1 \oplus B_2) \\ (f \ominus B_1) \ominus B_2 &= f \ominus (B_1 \oplus B_2) \end{aligned} \qquad (1')$$

EXERCISES

Exercise n° 1

1) Prove relations (1) and (1').
2) Prove or invalidate the following statements:
- Erosion and dilation are increasing,
- extensive, anti-extensive,
- idempotent,
- dual by complementation (or inversion for the functions).

Exercise n° 2

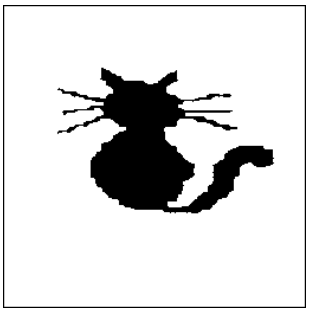
1) Use the MAMBA operators **linearErode**, **linearDilate**, **doublePointErode** and **doublePointDilate** to perform the following transformations:
- erosion and dilation by a segment L_n of size n (consisting of n+1 consecutive points) in both binary and greytone cases.

$$X \ominus \check{L}_n,\ X \oplus \check{L}_n,\ f \ominus \check{L}_n,\ f \oplus \check{L}_n$$

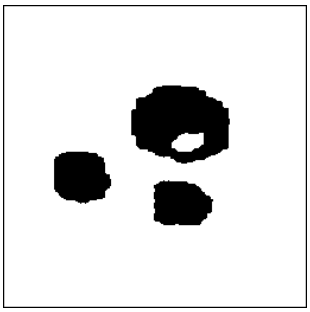
- erosion and dilation by a pair of points K_n at a distance n:

$$X \ominus \check{K}_n,\ X \oplus \check{K}_n,\ f \ominus \check{K}_n,\ f \oplus \check{K}_n$$

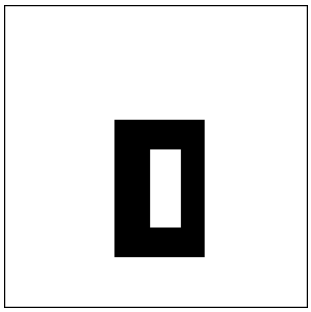
(the origin of the two structuring elements is arbitrarily chosen at one extremity).



cat



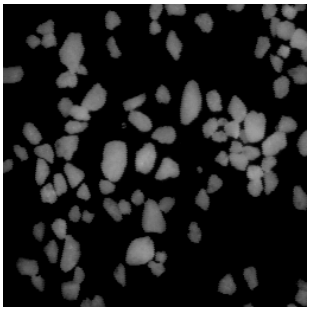
objects



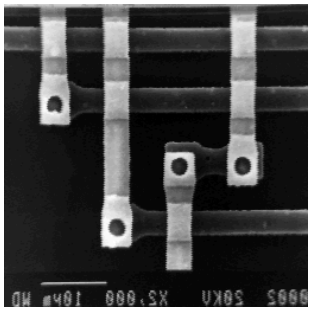
shape

2) Prove that:

$$X \ominus \check{L}_n \subset X \ominus \check{K}_n$$



salt



circuit

Exercise n° 3

Prove that an elementary hexagon may be generated by three successive dilations of a point by three judiciously chosen segments. Then, deduce an algorithm that allows to obtain the erosion and the dilation by an hexagon. Program these transformations (both for the hexagonal and square grids), and verify your algorithms on binary and greytone images.

Exercise n° 4

1) Perform erosions and dilations on binary and greyscale images (use the **erode** and **dilate** operators).
2) Which structuring element is used?
3) Modify the structuring element. Use a square one.
4) Verify the duality of the erosion and the dilation. Is the result satisfactory Can you explain the behavior of these operators at the edge of the images? Change the edge parameter and see what happens.

Exercise n° 5

1) Perform the erosion by an elementary triangle, pointing upwards (take the point O for origin).

2) Perform also the erosion by a 2x2 square.
3) What happens when the transformation is iterated? (Perform $(X \ominus \check{B}) \ominus \check{B}$). Represent the structuring element B' such that:

$$(X \ominus \check{B}) \ominus \check{B} = X \ominus \check{B}'$$

A simple way to know how it looks is to dilate a point by B, twice. Indeed:

$$\left[\{.\} \oplus \check{B}\right] \oplus \check{B} = \check{B} \oplus \check{B} = \overset{\vee}{\overline{B \oplus B}}$$

$$(X \ominus \check{B}) \ominus \check{B} = X \ominus (\check{B} \oplus \check{B}) \Rightarrow B' = B \oplus B$$

4) Program the dilation by a triangle pointing downwards.
5) Perform the dilations by the following structuring elements:

If H is the elementary hexagon, find the structuring element which is equivalent to:

$$B_1 \oplus H \oplus B_2$$

In this exercise, the class **structuringElement** can be useful to define some of the proposed structuring elements. Note also that some of them already exist in MAMBA: **TRIANGLE**, **SQUARE2x2**, **TRIPOD**, etc.

Exercise n° 6

1) Use the dilations and erosions by dodecagons and octogons (**dodecagonalErode**, **dodecagonalDilate**, **octogonalErode**, **octogonalDilate**). Can you explain how these operations are performed (see previous exercise)?
2) Use large structuring elements (size > 100) and compare the speed of the previous operators with the speed of the **largeDodecagonalDilate** and **largeOctogonalDilate** transforms. Which kind of difficulty must be overcome when programming these large structuring elements?

Exercise n° 7: Distance function

Let d be a distance defined on the points of the euclidean grid as the length of the shortest path drawn on the grid between two points. At any point x of X, the value of the distance function is:

$$dist(x) = d(x, X^c) = \min_{y \in X^c} d(x, y) \ .$$

Prove that, if d is the distance defined on the hexagonal grid as the length of the shortest path drawn on the grid between two points, the distance function can be obtained by means of the successive erosions of X by a hexagon.
Verify this by using the **computeDistance** operator and by comparing successive thresholds of the obtained distance function with the corresponding erosions of increasing sizes.
Use also the **computeDistance** operator on the square grid.

SOLUTIONS

Exercise n° 1

1) Let us prove that:

$$(X \cup Y) \oplus \check{B} = (X \oplus \check{B}) \cup (Y \oplus \check{B})$$

$$(X \cup Y) \oplus \check{B} = \bigcup_{b \in \check{B}} (X \cup Y)_b = \bigcup_{b \in \check{B}} (X_b \cup Y_b) = \left(\bigcup_{b \in \check{B}} X_b \right) \cup \left(\bigcup_{b \in \check{B}} Y_b \right)$$

that is $(X \oplus \check{B}) \cup (Y \oplus \check{B})$.
The second relation can be proved in the same way. Let us prove the third relation :

$$(X \oplus \check{B}_1) \oplus \check{B}_2 = \bigcup_{b_2 \in \check{B}_2} (X \oplus \check{B}_1)_{b_2} = \bigcup_{b_2 \in \check{B}_2} \left(\bigcup_{b_1 \in \check{B}_1} X_{b_1} \right)_{b_2} = \bigcup_{b_1 \in \check{B}_1, b_2 \in \check{B}_2} (X_{b_1 + b_2})$$

$$= \bigcup_{b \in \check{B}_1 \oplus \check{B}_2} (X_b) = X \oplus (\check{B}_1 \oplus \check{B}_2)$$

2) Let us verify the statements:

- Erosion and dilation are increasing transformations: true (since set union and set intersection, sup and inf are increasing).
- Extensivity or anti-extensivity?
Consider the case of erosion.

$$X \ominus \check{B} = \bigcap_{b \in \check{B}} X_b$$

Since the eroded set is the intersection of the translates of X, it does not contain X, unless B is reduced to the point of origin.
Conversely, stating that $X \ominus \check{B} \subset X$, requires to be able to write:

$$X \ominus \check{B} = X \cap \left(\bigcap_{b \in \check{B} - \{\emptyset\}} X_b \right)$$

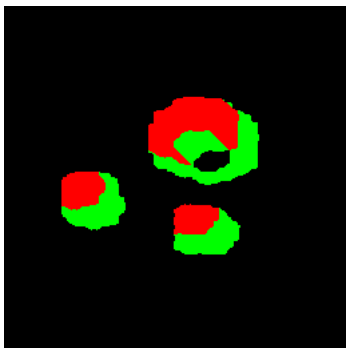
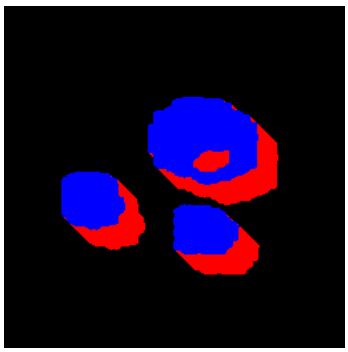
which is right only when the structuring element B contains its origin. As a general rule, erosion is then neither extensive nor anti-extensive. It is the same for dilation.
- Erosion and dilation are idempotent : false (obvious).
- Erosion and dilation are dual transforms:

$$(X \oplus \check{B})^c = \left(\bigcup_{b \in \check{B}} X_b \right)^c = \bigcap_{b \in \check{B}} (X_b)^c = \bigcap_{b \in \check{B}} (X)_b^c = (X^c \ominus \check{B}) \quad \text{Q.E.D.}$$

Exercise n° 2

1) The linear (the structuring element is a segment) erosion (binary one, $X \ominus \check{L}_n$ or greytone one, $f \ominus \check{L}_n$) is performed by means of the **linearErode** operator which works for binary and greytone images:

```
>>> linearErode(imbin1, imbin2, 4, n=15, grid=SQUARE)
>>> linearErode(im1, im2, 1, n=15, grid=HEXAGONAL)
```

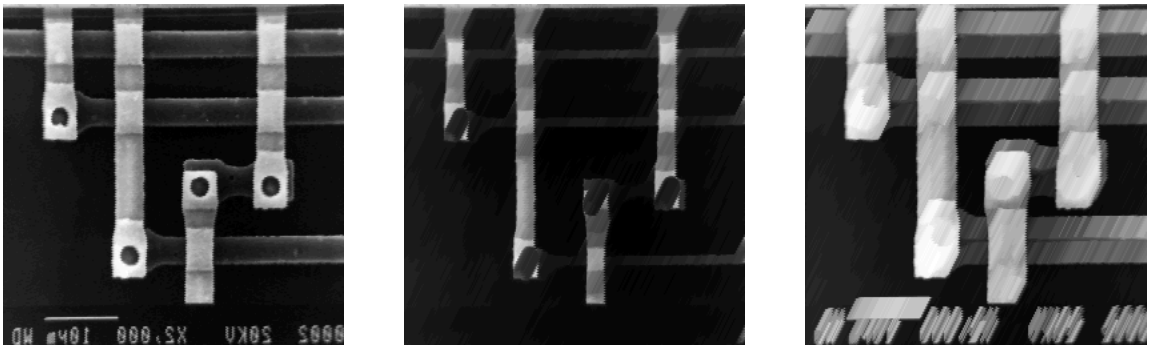



Linear erosion of the objects image (left) of size 15 in direction 4 of the square grid (the eroded set in in red). Linear dilation (right) of size 15 in direction 8 of the square grid (the initial set is in blue, the dilation adds the red points).

The linear dilation ($X \oplus \check{L}_n$ and $f \oplus \check{L}_n$) is realised with the **linearDilate** operator:

```
>>> linearDilate(imbin1, imbin2, 8, n=15, grid=SQUARE)
>>> linearDilate(im1, im2, 4, n=15, grid=HEXAGONAL)
```

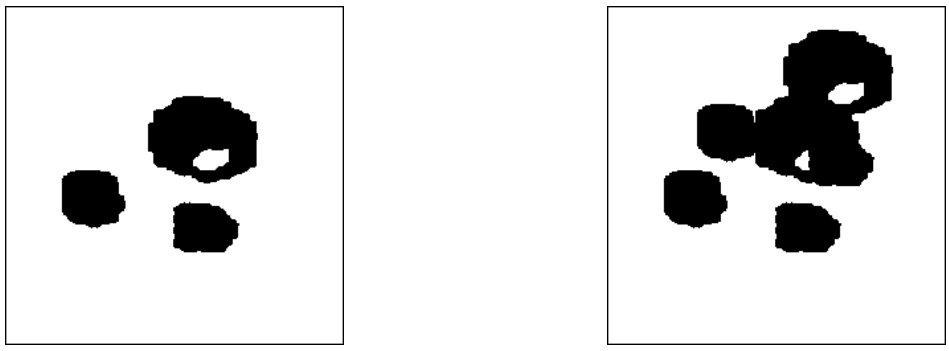
You must indicate the direction of the segment, its size n (if different from 1) and the grid used for the transformation. In the above example, the directions used are successively 4 on the square grid, 1 on the hexagonal grid, 8 on the square grid and 4 on the hexagonal grid.
Note also that the origin of the structuring element is always an extremity of the segment.



Linear erosion (middle image) of size 15 in direction 1 of the hexagonal grid of the initial image (left). Linear dilation (right image) of size 15 in direction 4 of the hexagonal grid.

The erosion by a pair of points ($X \ominus \check{K}_n$ and $f \ominus \check{K}_n$) is performed by the MAMBA operator **doublePointErode** and the dilation ($X \oplus \check{K}_n$ and $f \oplus \check{K}_n$) by **doublePointDilate**. The parameters are the same as those used by **linearErode** and **linearDilate**. The origin of the doublet of points always corresponds to one of these two points:

```
>>> doublePointDilate(imbin1, imbin2, 4, 50)
```



Dilation by a pair of points (right) at distance 50 in direction 4 (hexagonal grid) of the objects image (left).

2) Prove that $X \ominus \check{L}_n \subset X \ominus \check{K}_n$. Let x be a point of $X \ominus \check{L}_n$. The structuring element L_n translated in x is then included in X, and so are its two extremities consequently. Then x belongs to the eroded set $X \ominus \check{K}_n$ (Q.E.D.)

Exercise n° 3

The figure below shows how to obtain a hexagon from three segments. We have:

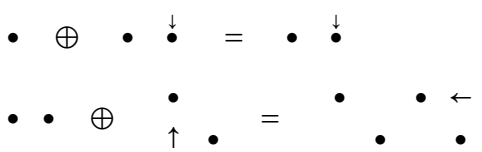
$$H = L_1 \oplus L_2 \oplus L_3$$

Then, we write:

$$X \oplus H = ((X \oplus L_1) \oplus L_2) \oplus L_3$$

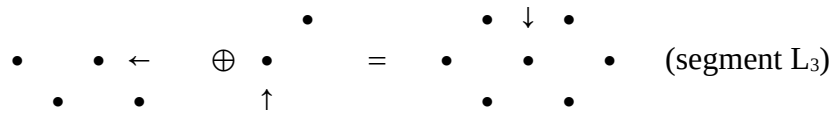
and also:

$$X \ominus H = X \ominus (L_1 \oplus L_2 \oplus L_3) = ((X \ominus L_1) \ominus L_2) \ominus L_3$$



(segment L_1)

(segment L_2)

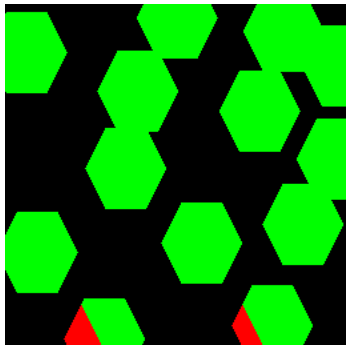


The same procedure can be used, with a fourth direction, for defining a square erosion.

(a) initial picture, (b) linear dilation at 60°, the point exits the field and is lost.

We could use the same formula to define the hexagonal dilation. However, in practice, this approach does not work because it does not take into account border effects. Indeed, a point which is on the field border will not always be correctly dilated, because some linear dilation may propagate a white point outside the field. This white point is then irremediably lost. Note that this problem does not occur with the square grid. Try to apply the following three successive linear dilations on image *points* to be aware of the problem:

```
>>> linearDilate(imbin1, imbin2, 1, 30)
>>> linearDilate(imbin2, imbin2, 3, 30)
>>> linearDilate(imbin2, imbin2, 5, 30)
```



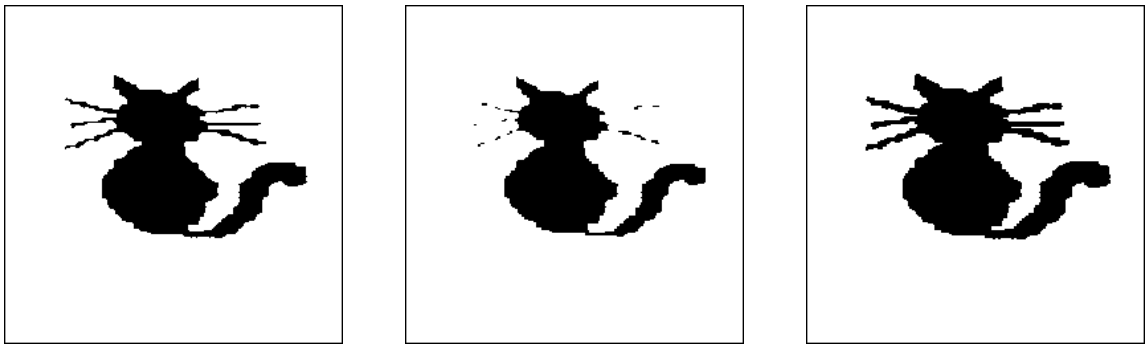
Result of three successive linear dilations of size 30 in directions 1, 3 and 5 applied to image points (in green). In red, the points which have been lost because of edge effects.

The solution consists in performing translations in the six directions of the hexagonal grid. In the hexagonal case, the erosion and the dilation are not correct when the transformations are made using only three directions. Therefore, it is necessary, in order to obtain unbiased transformations at the edges, to perform the hexagonal erosion and dilation by means of translations in the six main directions of the grid.
Edge effects are particularly annoying when large structuring elements are used. Specific and complex algorithms must be designed to cope with this problem. These algorithms are described in [2].

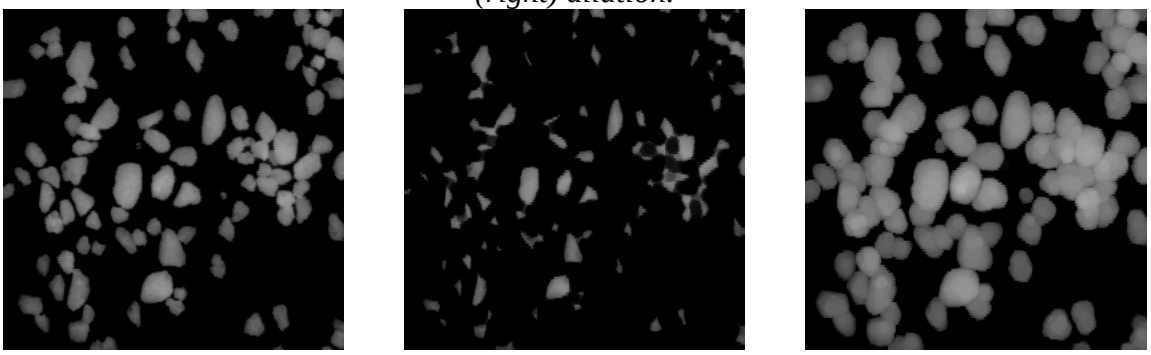
Exercise n° 4

1) **erode** and **dilate** are the two MAMBA operators which perform erosions and dilations with structuring elements defined on the neighborhood of a point (hexagonal or square).

```
>>> erode(imbin1, imbin2)
>>> dilate(imbin1, imbin2)
>>> erode(im1, im2, 3)
>>> dilate(im1, im2, 3)
```



Hexagonal binary erosion and dilation of size 1, (left) initial picture, (middle) erosion, (right) dilation.



Hexagonal greytone erosion and dilation of size 3, (left) initial picture, (middle) erosion, (right) dilation.

2) By default, the structuring element is an hexagon. You can know the default structuring element and change it by entering the following commands:

```
>>> DEFAULT_SE
structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)
>>> DEFAULT_SE.setAs(SQUARE3X3)
>>> DEFAULT_SE
structuringElement([0, 1, 2, 3, 4, 5, 6, 7, 8], mamba.SQUARE)
```

3) When changing the default structuring element to a square one (see above), the erosion and dilation operators use these new structuring element without any further specification:

```
>>> dilate(imbin1, imbin2, 25)
```

You can also, without changing the default structuring element, force the **dilate** or **erode** operators to use a specific one. For instance, assuming that the default structuring element is an hexagon, entering the command:

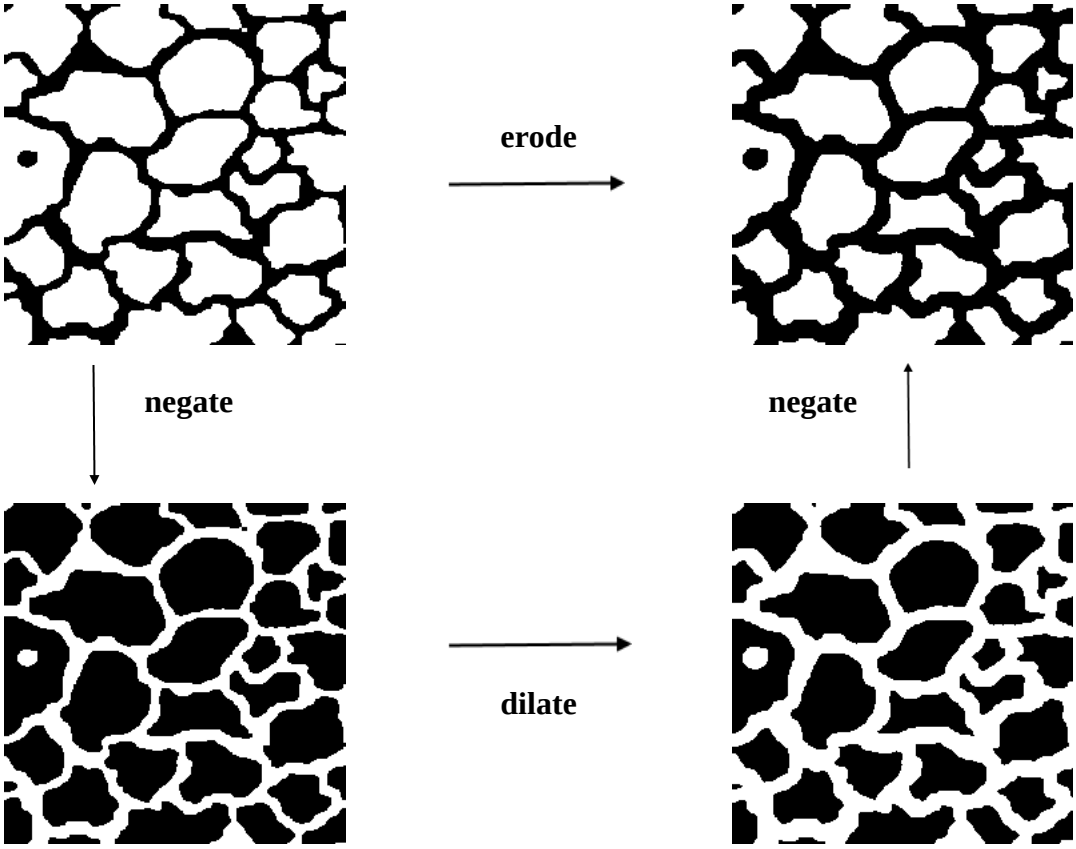
```
>>> dilate(imbin1, imbin2, 25, SQUARE3X3)
```

will perform a dilation of the image imbin1 into imbin2 by a square of size 25.

Dilation of size 25 of the shape image with an hexagon (left), dilation of size 25 with a square (right).

4) Let us use the *alumine* binary image loaded in imbin1. If we perform the following operations:

```
>>> erode(imbin1, imbin2, 2)
>>> negate(imbin1, imbin3)
>>> dilate(imbin3, imbin3, 2)
>>> negate(imbin3, imbin3)
```

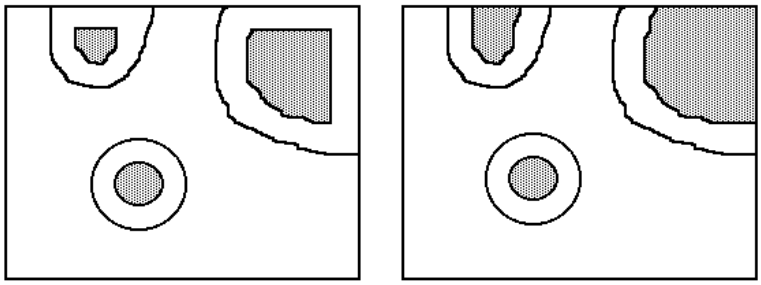


Duality of the erosion and of the dilation by complementation (the images have not been inverted in this figure).

The images imbin3 and imbin2 are identical as it can be asserted by comparing them:

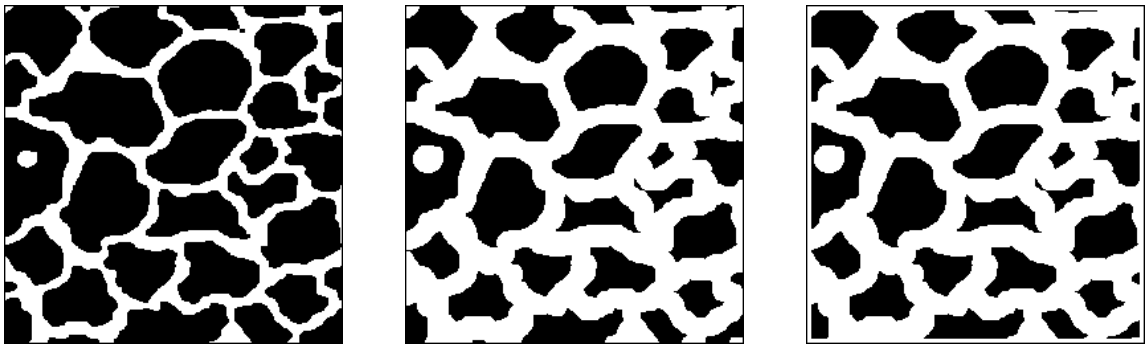
```
>>> compare(imbin2, imbin3, imbin4)
(-1, -1)
```

This awaited result seems strange however. Indeed, if we look at the particles of the initial image which cut the edges, they should also have been reduced along these edges (see illustration below). Fortunately, it is not the case, otherwise the duality would not be verified.



Erosion when the edge parameter is set to EMPTY (left) and to FILLED (right).

To achieve this, the parameter 'edge' can be set in **erode** and **dilate**. If 'edge' is set to **EMPTY**, the outside of the binary image is considered as being empty. The objects under study are entirely included in the field of analysis. The outside of a greytone image is equal to 0. When performing an erosion, a black border appears which increases and deeply reduces the image field when the size of the transformation increases. Conversely, when setting the edge to **FILLED**, the outside of the image is considered as being filled. So, in the case of the erosion, the structuring element B, when crossing the boundary of the image field D, does not take into account its points which are outside D (they are not considered in the inf). This is equivalent to perform the transformation with the structuring element $B \cap D$. This mode is, in fact, a geodesic mode, see chapter on the geodesic transforms.
By default, 'edge' is set to **FILLED** in the erosion and to **EMPTY** in the dilation. This allows to discard the outside when computing the transformations: **FILLED** points are not taken into account in the erosion and **EMPTY** points are not considered in the dilation. Changing the 'edge' value is possible but you must be aware of the consequence of this setting in the edge management.



Hexagonal erosion of size 4 of the initial image (left) when the parameter 'edge' is set to FILLED (middle) and when set to EMPTY (right).

```
>>> erode(imbin1, imbin2, 4)
>>> erode(imbin1, imbin2, 4, edge=EMPTY)
```

Exercise n° 5

This exercise gives you the opportunity to use the various structuring elements pre-defined in MAMBA.
1) Erosion by a triangle (point upwards): this structuring element is already defined in MAMBA (**TRIANGLE**):

```
>>> erode(im1, im2, se=TRIANGLE)
```

2) The erosion by an elementary square is performed with the **SQUARE2X2** structuring element:

```
>>> erode(im1, im2, se=SQUARE2X2)
```

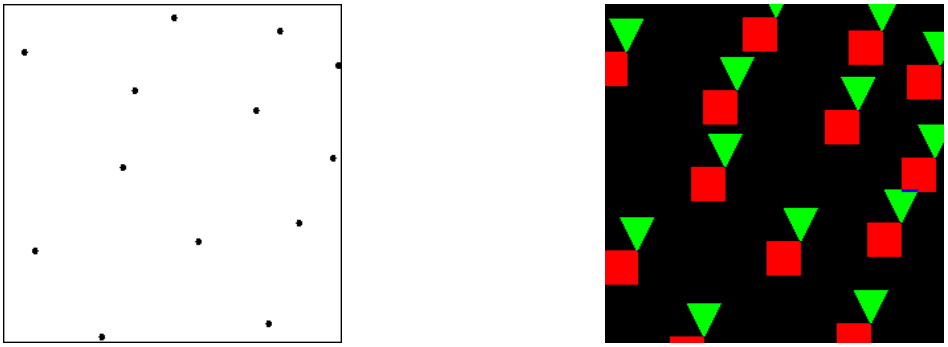
Note the position of the origin of the structuring element (lower left point).

3) Repeating two triangular transformations (erosion or dilation) results in a triangular transformation of size 2. More generally, n iterations are equivalent to a single transformation by a triangle of size n. Note the orientation of the triangle in the transformation when dilating single points (use the *points* image loaded in imbin1):

```
>>> dilate(imbin1, imbin2, 2, se=TRIANGLE)
```

Similarly, iterating a 2x2 square transformation n times is equivalent to use a (n+1) square. Note however the position of the origin which corresponds always to the lower left point and the result of the dilation when applied to the image *points*.

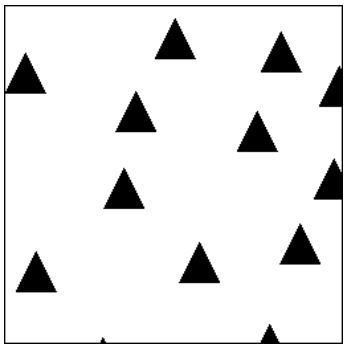
```
>>> dilate(imbin1, imbin2, 25, se=SQUARE2X2)
```



Right image: dilation of the points image (left) by a TRIANGLE structuring element of size 25 (in green) and by a SQUARE2X2 structuring element of same size (in red).

4) The dilation by a triangle pointing downwards is realised by using the method **transpose** applied to the **TRIANGLE** structuring element.This method computes the symmetry around the origin of the structuring element:

```
>>> dilate(imbin1, imbin2, 30, se=TRIANGLE.transpose())
```

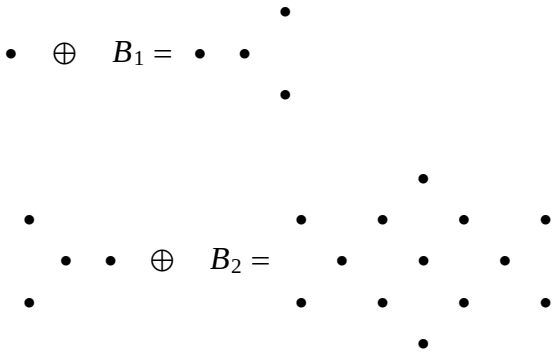


Dilation of the image points by a triangle of size 30 pointing downwards (the result shows triangles pointing upwards).

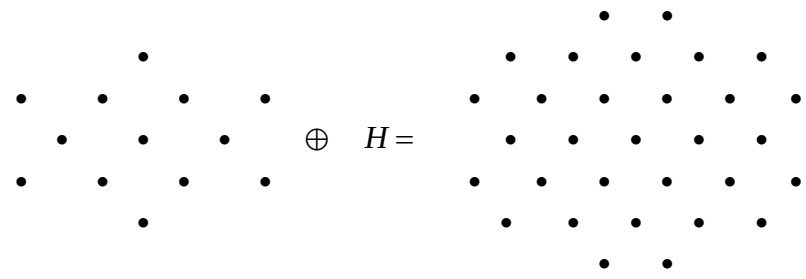
5) The structuring element B_1 is already defined in MAMBA and is named **TRIPOD**. This structuring element is defined on the hexagonal grid. B_2 is the transposition of B_1. Therefore, elementary dilations (size 1) using structuring elements B_1 and B_2 can be performed by:

```
>>> dilate(im1, im2, se=TRIPOD)
>>> dilate(im1, im2, se=TRIPOD.transpose())
```

Three successive dilations by B_1, B_2 and H are equivalent to a dodecagonal dilation:



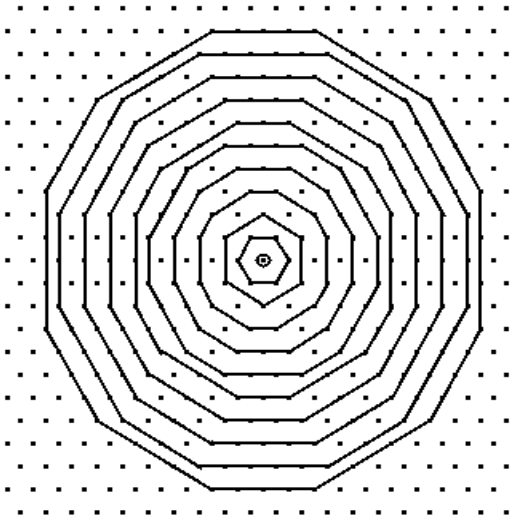
Note that these two successive dilations by B1 and B2 produce a dilation by the conjugate hexagon (hexagon turned by a 30° angle).



Note also that Minkowski sums are used in the above operations instead of dilations (the structuring elements are not transposed in the formulae). This has obvious no consequence as, in the one hand, B_1 is the transposed set of B_2 and vice versa (H is isotropic) and, in the second hand, dilations are commutative.

Exercise n° 6

1) Dilations and erosions by dodecagons are available in MAMBA by means of the **dodecagonalErode** and **dodecagonalDilate** operators. These transformations are built by successive erosions or dilations by hexagons and conjugate hexagons, as described in the previous exercise. The sizes of the two operations are computed so that the resulting dodecagon be as isotropic as possible and that each dodecagon of size n be included in the hexagon of same size.



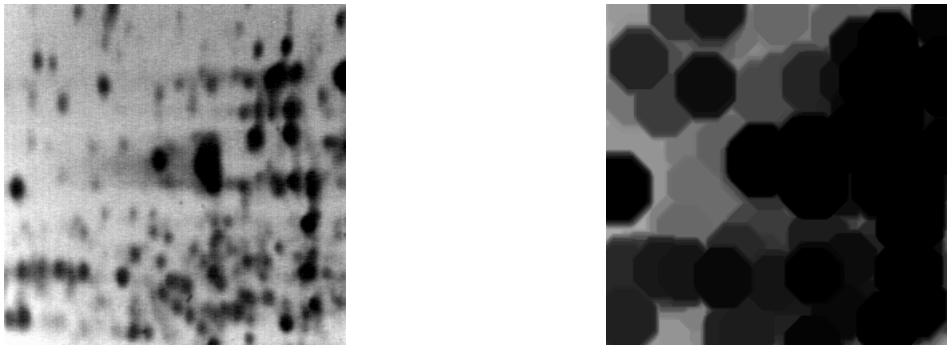
Successive dodecagons of increasing sizes used by the dodecagonal operators. Each dodecagon is included in the hexagon of same size.

```
>>> dodecagonalDilate(imbin1, imbin2, 20)
```

Dodecagonal dilation of size 20 of the objects image (left). This transformation can be compared to the hexagonal dilation of same size (right).

Morphological operations using octogons exist also in MAMBA: **octogonalErode** and **octogonalDilate**. These transformations are built in the same way as the dodecagonal ones by combining operations using the **SQUARE3X3** and the **DIAMOND** structuring elements.
The algorithmic implementation of all these transformations is thoroughly explained in [1].

```
>>> octogonalErode(im1, im2, 20)
```



Octogonal erosion of size 20 (right) of the electrop image (left).

2) Fast implementations of erosions and dilations exist in MAMBA when working with large structuring elements: squares, hexagons, dodecagons or octogons. The larger the size of the structuring element, the higher the speed increase of the transformation. You can verify this with the following experiment. Define two 1024x1024 binary images imA and imB, display imA:

```
>>> imA = imageMb(1024, 1024, 1)
>>> imB = imageMb(imA)
>>> imA.show()
```

Load the *bigcat* image in imA (do not display imB!). Then, import the Python module named **time**:

```
>>> from time import *
```

This module will allow to calculate approximately the computation time of a transformation. Then compare the computation time of **dodecagonalDilate** and **largeDodecagonalDilate** by entering the following command (all the instructions must be typed on a single line):

```
>>> t0 = time(); dodecagonalDilate(imA, imB, 200); t1 = time(); print(t1 - t0)
```

The value returned at the end of the operation is the approximate time in milliseconds of the transformation (in this case a dodecagonal dilation of size 200). Then, do the same with the **largeDodecagonalDilate** operator:

```
>>> t0 = time(); largeDodecagonalDilate(imA, imB, 200); t1 = time(); print(t1 - t0)
```

The speed of the operation can be multiplied by 5 on a performing computer. Similar speed increases can be observed with the other operators (**largeHexagonaldilate**, **largeSquareDilate**, etc.).

The main difficulty when implementing these operators comes from the important edge effects which may happen if they are not controlled and avoided (see previous exercises). Therefore, this implementation is quite tricky. More details regarding the design of these algorithms can be found in [2].

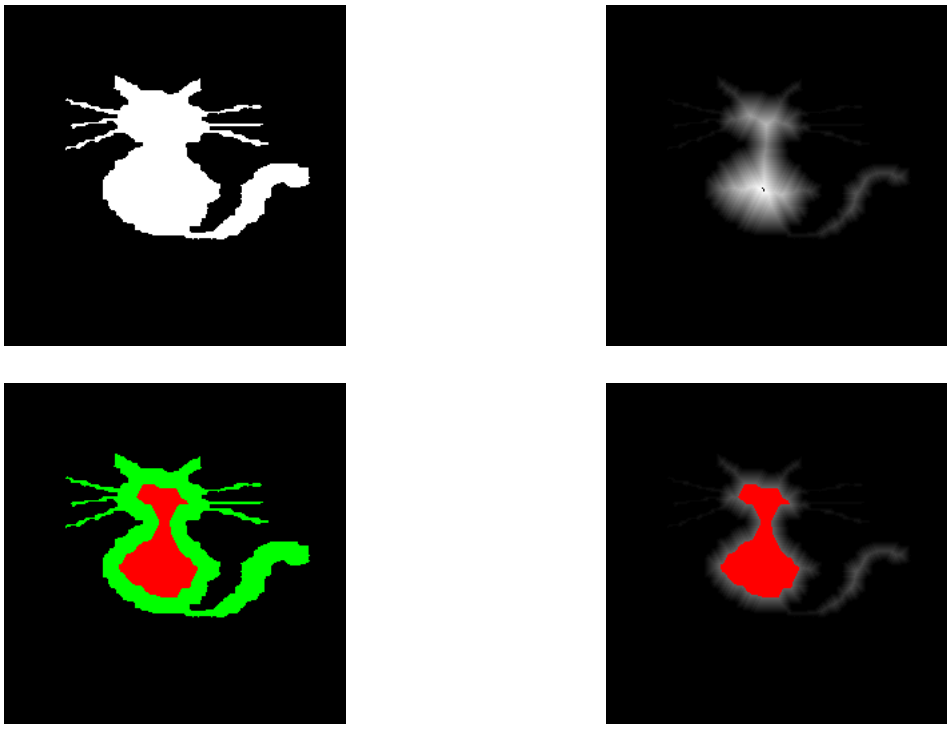
Exercise n° 7: Distance function

For this distance, the set of points located at a distance smaller than or equal to n from a point x is the hexagon of size n centered in x.
Let x be a point of X located at a distance n from X^c (the distance function at x is n). The hexagon of size n centered in x contains a point of X^c (by hypothesis) and the hexagon of size n-1 centered in x does not contain any point of X^c (otherwise the distance would be smaller than n). Then x belongs to the set X eroded by a hexagon of size n-1 and does not belong to the set X eroded by a hexagon of size n.
Each section at level i of the distance function of a set X therefore corresponds to the erosion of size i-1 of X. Computing this distance function could then be achieved by adding the successive erosions of the set X. However, this implementation would be quite slow, the computation time being proportional to the size of the connected components of X. The MAMBA implementation, **computeDistance**, does not depend on the size of each connected component. It is based on propagations. However, each section at level i of the result is equal to the eroded set of size i-1.

```
>>> computeDistance(imbin1, im32_1)
```

Remind that the result of **computeDistance** must be stored in a 32-bit image. To verify the equality with the erosion, do for example:

```
>>> threshold(im32_1, imbin2, 13, computeMaxRange(im32_1)[1])
>>> erode(imbin1, imbin3, 12)
>>> compare(imbin2, imbin3, imbin4)
(-1, -1)
```



Initial set (upper left), distance function computed on the hexagonal grid (upper right), erosion of size 12 of the initial set (in red, lower left) and section at level 13 of the distance function (lower right).

Note also that **computeDistance** works on the square grid (the parameter 'grid' is set to HEXAGONAL by default).

REFERENCES

[1] S.Beucher: Algorithmic description of erosions and dilations in Mamba
(**http://cmm.ensmp.fr/~beucher/publi/Algorithmic description of erosions and dilations in Mamba.pdf**)
This document is a short description of the basic morphological operators using hexagons, squares, dodecagons and octogons.

[2] S.Beucher: Fast implementation of large erosions and dilations in Mamba
(**http://cmm.ensmp.fr/~beucher/publi/Mamba-LargeSE_implementation.pdf**)
A complete description of a fast implementation of large erosions and dilations with hexagons, squares, dodecagons and octogons. This implementation is the fastest one ever of these basic transformations with a full management of the edge effects. In Mamba 2.0, these algorithms have been applied to fast 3D linear dilations and erosions.

Chapter 3

MEASURES

1 Reminder

Any morphological transformation is to lead eventually to a measure on the transformed set. The main purpose of a transformation (or sequence of transformations) is to detect the objects to be measured: the openings revealing size distributions are an example.
Measures that satisfy good compatibility properties with translations and homotheties are not so numerous. Namely, in 2D:
- area
- diameter variations
- perimeter
- connectivity number

2 Measures and morphological transformations

Any measurement consists of two steps: transformation + counting of the points of the transformed image. Measuring an area is very simple as the transformation is identity, it then amounts to counting the points of the set. Besides, this measure is obtained when applying the MAMBA **computeVolume** operator on any binary image.
In digitized images, the intercept numbers correspond to diameter variations. On the hexagonal grid, we have three directions:

$$I_1 = N(0\ 1)\ ;\ I_2 = N\begin{pmatrix} & 1 \\ 0 & \end{pmatrix}\ ;\ I_3 = N\begin{pmatrix} 1 & \\ & 0 \end{pmatrix}$$

On the square grid, a fourth direction is used:

$$I_1 = N\begin{pmatrix} 1 \\ 0 \end{pmatrix}\ ;\ I_2 = N\begin{pmatrix} & 1 \\ 0 & \end{pmatrix}\ ;\ I_3 = N\begin{pmatrix} 0 & 1 \end{pmatrix}\ ;\ I_4 = N\begin{pmatrix} 0 & \\ & 1 \end{pmatrix}$$

The transformation associated with this measure then consists in extracting the intercepts.
Measuring the connectivity number is a little more complex, since it requires multiple transformations and countings. On the hexaconal grid, we can prove that:

$$v = N\begin{pmatrix} 0 & 0 \\ & 1 \end{pmatrix} - N\begin{pmatrix} & 0 & \\ 1 & & 1 \end{pmatrix}$$

On the square grid, three transformations and countings are needed:

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

Remind that in 2D, the connectivity number is equal to the number of connected components minus the number of holes of the set.
Programming measures is not enough, you have to learn how to interpret the results. Some of the following exercises are intended to familiarize you with this important step of morphological treatments.

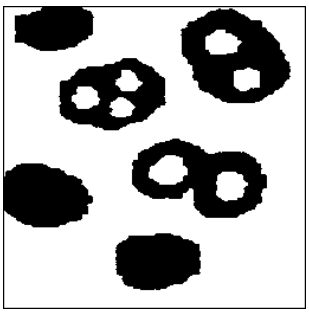
3 Measures in MAMBA, precisions and suggestions

The MAMBA library contains some basic measurement operators. They are named **computeArea**, **computeDiameter**, **computePerimeter** and **computeConnectivityNumber** and are acting only on binary images. More deteails about these operators can be found in [1].
As said before, all these operators use the **computeVolume** function which computes the number of pixels of a binary image and the integral (volume of the sub-graph) of a greytone image. **computeVolume** returns an integer value.
The other operators contain a parameter (a tuple), the scale factor, allowing to return calibrated measures corresponding to the real dimensions of the image. The returned measures are real values, except for the connectivity number which is always an integer. In the following exercises, this scale factor is set by default to (1.0, 1.0).
In this chapter and in the next ones, some exercises require the computation of curves and graphical data. To achieve this, we recommend the use of the **matplotlib** library. Install it, together with the **scipy** and **numpy** libraries which are very useful Python libraries when performing scientific calculations.

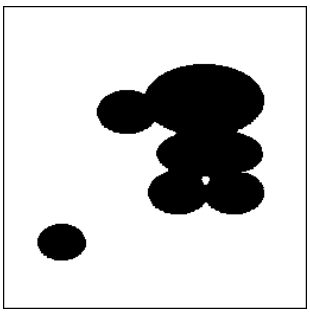
EXERCISES

Exercise n° 1

1) Use the aforementioned operators to measure areas and diameters of various binary images (*holes*, *cells*, *grains1*, *grains2*) on hexagonal and square grids.
2) Measure the connectivity number of the *holes*, *cells* and *noise* images on the hexagonal and square grids. Can you explain the differences of values obtained on the hexagonal and square grids?
3) Measure the perimeter of *holes* and *cells* on the two grids. Can you propose alternative ways of obtaining this measure? Compare their respective accuracy and discuss their biases.



holes



cells

noise

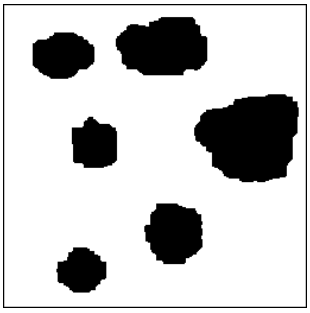
Exercise n° 2: Transitive and stationary hypotheses

Observing the *grains1* image suggests that the set under study is entirely known and included in the field of measurement. In that case, it is quite legitimate to speak of the area of the set and of its connectivity number. Similarly, it is possible to define the area of the eroded (or dilated) set provided that it is entirely contained in the field of measurement. When this

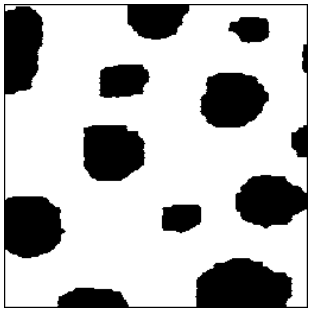
assumption of exhaustive knowledge is enforced, the measurement working mode is said to be transitive.
Conversely, on the *grains2* image, such a hypothesis is hardly acceptable. Obviously, only a part of a more extended set appears in the field of analysis. In that case, the only meaningful measures are those related to the unit area: ratio, specific connectivity number, etc. . We then speak of a stationary working mode. Measurements are performed by constructing non biased estimates. Thus, the ratio of a set X is estimated by:

$$t = \frac{mes(X \cap D)}{mes(D)}$$

D is the field of measurement, and $X \cap D$ corresponds to the part of set X that is known.
1) Can you compute a non biased estimate of the ratio of the eroded set X? (As the eroded set X is biased in the field D, you have to find a field D' in which the eroded set is exact, and use it to obtain the ratio estimate).
2) Same question for the dilated set.



grains1

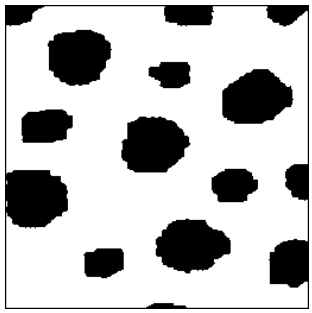


grains2

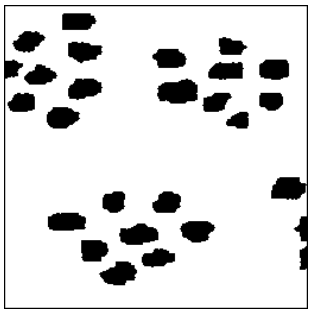
Exercise n° 3

C(l) denotes the covariance of size l, the measure of the area (transitive case) or of the ratio (stationary case) of the set X eroded by a pair of points at a distance l. In fact, the transformation itself is often called covariance.
In the stationary case, the covariance is an estimate of the probability for a pair of points to be included into the set X assumed to extend to the whole space.
1) Program the covariance (stationary case only). To do so, refer to the suggestions made above (especially, concerning the plotting of the curve with **matplotlib**).
2) Application to *particle1*, *particle2* and *eutectic* images.
3) Interpret the main features of the curve C(l), more particularly C(0), C(∞), the tangent at the origin, and the overall outlook of the curve.

eutectic



particle1



particle2

Exercise n° 4: Individual analysis of particles

The measurement operators introduced and used in the previous exercises work on the whole binary image. These measures are not able to distinguish the various connected components contained in the image. However, it is often necessary to measure separately each particle in the image.
The purpose of this exercise is to present some MAMBA operators which can be very helpful to extract easily each connected component of a set and to measure it individually. Two operators, in particular, are interesting: **label** and **getHistogram**.
label generates a 32-bit image where each connected component of the initial image is assigned a single grey value. This operator also returns the number of connected components of the set.
getHistogram works on greyscale (8-bit) images and produces a list of 256 integer values. The i-th value correponds to the number of pixels in the image with a grey value equal to i.
1) Practice the **label** and the **getHistogram** operators. Use the **matplotlib** library to plot the histograms.
2) Find a way to easily extract each connected component of a set.
3) Can you design a procedure for measuring rapidly the area of each particle of a set?

SOLUTIONS

Exercise n° 1

1) Measuring areas is achieved with the **computeArea** operator and diameters (intercept numbers) with **computeDiameter**. Load the binary image *noise* in imbin1 and type:

```
>>> computeArea(imbin1)
21129.0
```

The area is a real number. As the scale factor is set by default to (1.0, 1.0), this value is equal to the value returned by computeVolume (number of pixels for a binary image). This latter value, however, is a long integer:

```
>>> computeVolume(imbin1)
21129L
```

To test the measurement of the diameters, load the *cat* image in imbin1, then type:

```
>>> computeDiameter(imbin1, 2)
244.0
```

This command returns a real value equal to 244.0 corresponding to the vertical diameter (the programmed direction is 2, that is the horizontal one on the hexagonal grid). In direction 1, we have:

```
>>> computeDiameter(imbin1, 1)
461.5244305559566
```

The non integer result in this direction comes from the fact that, the vertical scale factor being equal to 1, the hexagonal grid is not isotropic. Thus, parallel lines in directions other then 2 are not at a distance one apart.

Horizontal intercepts (right) of the left image. With a default scale factor, the number of intercept points in the horizontal direction is equal to the diameter.

The vertical diameter is proportional (equal if the scale factor is equal to 1) to the number of intercepts in the horizontal direction (configurations (1 0)). These configurations can be extracted with the **diffNeighbor** operator. For details on this operator and on the settings of its parameters, see the MAMBA user and reference manuals. You can verify this by loading the *cat* image in imbin1 and entering the following commands:

```
 >>> diffNeighbor(imbin1, imbin1, 4)
>>> computeVolume(imbin1)
244L
```

The integer returned value (number of intercepts points) is equal to the diameter obtained with the operator **computeDiameter**.
Note also that, in all the measurement operators, the edge of the binary image is always set to **EMPTY**. As a consequence, diameter measurements in transposed directions (1 and 4 on the hexagonal grid for instance) give identical results. Indeed, any entry in a given direction into a connected component must be followed by an exit: any intercept in this direction is followed by a corresponding intercept in the opposite direction.
You can also perform these measurements on the square grid:

```
>>> computeDiameter(imbin1, 3, grid=SQUARE)
244.0
```

The value of the diameter in direction 3 of the square grid is obviously equal to the diameter in direction 2 of the hexagonal grid.

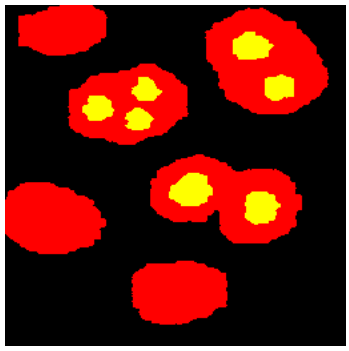
2) The connectivity number of a binary image is given by the **computeConnectivityNumber** operator. This operator extracts the various configurations used in the connectivity number formulae and returns the measure. In the design of this operator, we must take care of the origin of the structuring elements to avoid biases due to edge effects. With the *holes* image loaded in imbin1, we have:

```
>>> computeConnectivityNumber(imbin1)
-1L
```

Remember that, in 2D, the connectivity number is equal to the number of connected components of a set minus its number of holes. Note that another operator, named computeComponentsNumber, is available in MAMBA. It returns the number of connected components:

```
>>> computeComponentsNumber(imbin1)
6
```

Therefore, the holes images contains 7 holes (6 - 7 = -1).



The 6 connected components of the holes image (in red) and the 7 holes (in yellow).

The connectivity number can also be measured on the square grid:

```
 >>> computeConnectivityNumber(imbin1, grid=SQUARE)
-1L
```

The result is the same on both grids. However, most of the time, it is not the case, as it can be verified with the *noise* image:

```
>>> computeConnectivityNumber(imbin1)
1156L
>>> computeConnectivityNumber(imbin1, grid=SQUARE)
2097L
```

On the square grid (left), the two points are neighbors (one component). On the hexagonal grid (right), the same configuration is cut into two components (non neighbor points).

This difference is due to the fact that the neighborhood relationships between points are not the same on the hexagonal and square grids. On the hexagonal grid, every point, belonging either to the set or to the background, has six neighbors. On the square grid, points belonging to the set have eight neighbors, whereas points belonging to the background have only four neighbors. Note that the number of connected components is also different:

```
>>> computeComponentsNumber(imbin1)
2901
```

```
>>> computeComponentsNumber(imbin1, grid=SQUARE)
2549
```

3) The only unbiased measure of the perimeter is given by applying the digital version of the Cauchy formula which relates the perimeter to the mean diameter variation D in all the α directions:

$$P = \pi \underset{\alpha}{E}(D_\alpha) = \int_0^\pi D_\alpha d\alpha$$

The perimeter is equal to π multiplied by the mean diameter of the set. The digital version of this formula is implemented in the **computePerimeter** operator. The averaging is performed in three or four directions depending on the grid in use:

$$P = \frac{\pi}{k} \sum_{i=1}^{k} D_i$$

k is the number of directions used. For the images *cells* and *holes* respectively, we get:

```
>>> computePerimeter(imbin1)
788.8308609606588
```

for the *cells* image and:

```
>>> computePerimeter(imbin1)
2166.8588057782454
```

for the *holes* one.
There are other means for measuring a so-called perimeter, however, they all present more or less important biases. It is the case, in particular, for the internal contour measurement (number of points) of a set. The result is particularly inaccurate when the measured set is thin. The points of the boundary should be counted twice.
To reduce this bias, the external contour can be measured and the average of the two measurements can be taken. But, on the hexagonal grid, it can be proved that the difference between the external and internal contours is equal to 6ν, where ν is the connectivity number. Then we have:

(external_contour + internal_contour)/2 = internal_contour + 3ν

Since this average depends on the number of connected components of the picture, it is biased. Moreover, it is not exact for the connected components touching the edges. Load the *two_three_holes* image in imbin1 and enter the following commands:

```
>>> erode(imbin1, imbin2)
>>> diff(imbin1, imbin2, imbin2) # internal contour
>>> computeVolume(imbin2)
1606L
```

The internal contour contains 1606 points. Then enter:

```
>>> dilate(imbin1, imbin3)
>>> diff(imbin3, imbin1, imbin3)
>>> computeVolume(imbin3)
1576L
```

The external contour contains 1576 points. The difference is equal to -30. The computation of the connectivity number gives:

```
>>> computeConnectivityNumber(imbin1)
-5L
```

which verifies the above formula.

The two_three_holes image (left) and the internal (yellow) and external (red) contours.

Exercise n° 2: Transitive and stationary hypotheses

1) X being known in the field D only, computing an unbiased estimate of the ratio of the eroded set $X \ominus B$ then consists in looking for a field D' in which the eroded set is not biased. One should have :

$$[(X \cap D) \ominus B] \cap D' = (X \ominus B) \cap D'$$

$$\text{i.e. } (X \ominus B) \cap (D \ominus B) \cap D' = (X \ominus B) \cap D'$$

This equality is satisfied for any X if :

$$(D \ominus B) \cap D' = D'$$

Assume that $D' \subset D \ominus B$. Then, we have:

$$(D \ominus B) \cap D' = D'$$

The above equality is satisfied.

On the contrary, assume that $D' \supset D \ominus B$. We have:

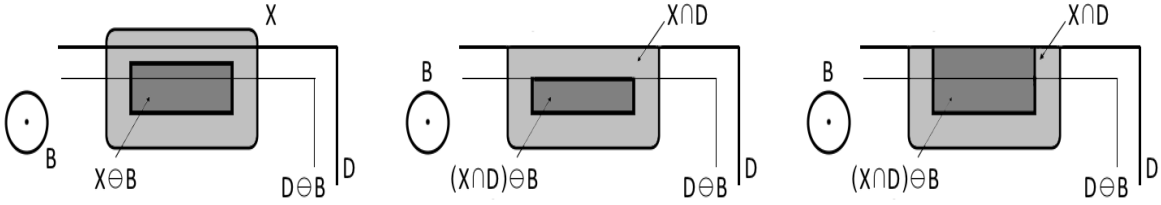
$$(D \ominus B) \cap D' = D \ominus B$$

which satisfies the equality if and only if $(D \ominus B) = D'$

Therefore, the largest mask D' in which the eroded set is unbiased is the mask D eroded by B.

Thus, an unbiased estimate of the eroded set ratio will be:

$$t_{X \ominus B} = \frac{mes[(X \cap D) \ominus B]}{mes(D \ominus B)}$$



Left image: set X and its erosion. Middle image: erosion when the edge is set to EMPTY. Right image: erosion when the edge is set to FILLED. In both cases, the erosion is unbiased only in the field D eroded by B.

Note that the correction (erosion of the mask) is the same whatever the setting of the edge (**EMPTY** or **FILLED**) in the erosion. Indeed, if we consider the restriction of a set X supposed to spread over the whole space, setting the edge to FILLED amounts to replace the outside part of X by a filled space which obviously is not identical to the set X.

2) If the ratio of the dilated set $X \oplus B$ is not biased, so will be the ratio of the eroded set $X^c \ominus B$ and vice versa. This means that the mask D' in which the ratio of the dilated set is measured is once again the mask $D \ominus B$.
Let us prove this a posteriori. The mask D' must satisfy:

$$[(X \cap D) \oplus B] \cap D' = (X \oplus B) \cap D'$$

Let us prove there is equality when $D' = D \ominus B$. The second expression can be written:

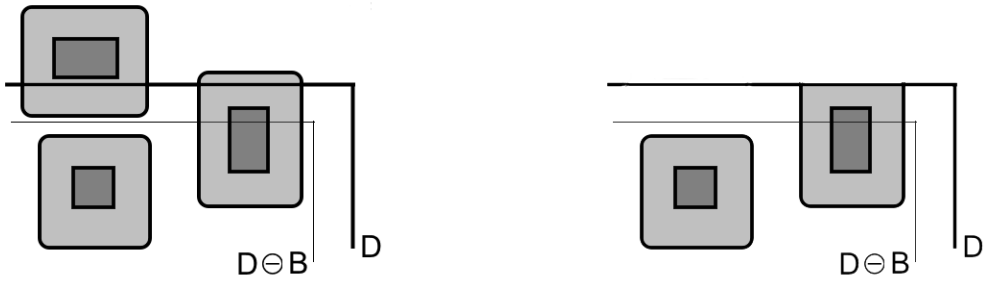
$$\begin{aligned}(X \oplus B) \cap D' &= (X \oplus B) \cap (D \ominus B) \\ &= ([(X \cap D) \cup (X \cap D^c)] \oplus B) \cap (D \ominus B) \\ &= [[(X \cap D) \oplus B] \cup [(X \cap D^c) \oplus B]] \cap (D \ominus B) \\ &= [[(X \cap D) \oplus B] \cap (D \ominus B)] \cup \underbrace{[[(X \cap D^c) \oplus B] \cap (D \ominus B)]}_{\text{(b)}}\end{aligned}$$

(b) is empty. Indeed:

$$(X \cap D^c) \oplus B \subset D^c \oplus B = (D \ominus B)^c$$

Then, $(X \cap D)^c \oplus B$ is included in the complementary set of $(D \ominus B)$. The intersection is empty. Q.E.D.
We can write:

$$t_{X \oplus B} = \frac{mes[[(X \cap D) \oplus B] \cap (D \ominus B)]}{mes(D \ominus B)}$$



Left image: the set X (dark grey) and its dilation (light grey). Right image: the restriction of X to the field D does not propagate some outside parts of X inside D. The resulting dilation is biased. Only its part inside the erosion of D is correct.

It may seem strange that, in the case of a dilation (especially when the operation is extensive), the correct measurement mask is obtained by an erosion. However, this result becomes clear when we consider the connected components of X outside D, which are likely to come inside it when dilated, contributing then to the measure. As they are not known, considering an eroded field is compulsory to avoid measurement biases.

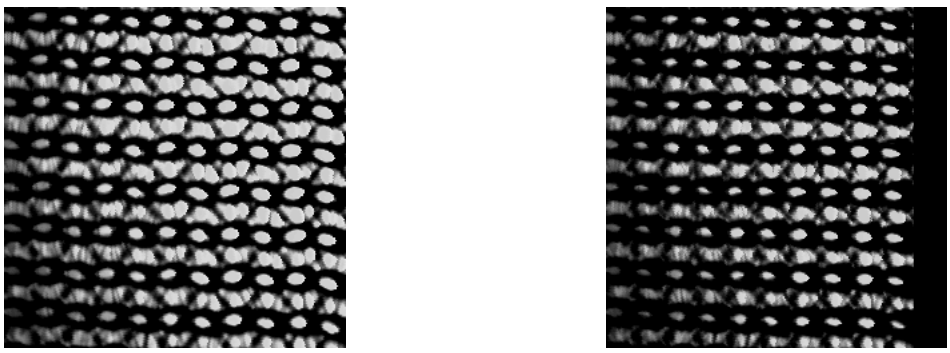
Exercise n° 3

1) The operator used to compute the covariance is already known: it is named **doublePointErode**. Let us define this covariance function. Its parameters are the input image, the direction of the covariance, the range of sizes and the grid in use (set to default grid). The operator returns a list of values.

```
from mamba import *

def covariance(imIn, dir, sizeRange, grid=DEFAULT_GRID):
  """
  This operator calculates the covariance of image 'imIn' in the direction
  'dir' of 'grid' for all tha sizes in 'size_range'.
  It returns a list of real values.
  """

  imField = imageMb(imIn, 1) # Binary measurement field
  imWrk = imageMb(imIn) # Working image
  covarList = [] # Initializing the covariance list
  for i in range(sizeRange):
    doublePointErode(imIn, imWrk, dir, i, grid=grid, edge=EMPTY)
    v1 = computeVolume(imWrk)
    # Performing the erosion by a doublet of points and measuring
    # its area. Note the setting of edge!
    imField.fill(1)
    doublePointErode(imField, imField, dir, i, grid=grid, edge=EMPTY)
    v2 = computeVolume(imField)
    # Generating the current unbiased measurement mask and measuring
    # its area.
    covarValue = float(v1) / v2
    covarList.append(covarValue)
  return covarList
```

The **covariance** measure is unbiased thanks to the correction which has been introduced in the previous exercise (erosion of the measurement mask).
Note that the edge in the erosion is set to **EMPTY**. It is compulsory if we want to obtain a black strip in the measuremnt mask image. If not, this image would never change whatever the size of the stucturing element. Consequently, the measure of the covariance would be false.
Note also that setting the edge to **EMPTY** for the image to be measured makes unnecessary the intersection of the eroded image with the eroded mask, as this setting generates in the measured image a black strip of same size as the one appearing in the eroded mask.
Finally, the covariance function can be used also on greytone images. Verify it on the *knitting* image loaded in im1, for instance:

```
>>> c = covariance(im1, 3, 30, grid=SQUARE)
```



The erosion of the left greyscale image by a doublet of points at distance 25 with the edge set to EMPTY produces a black strip, as shown in right image.

2) The covariance measure can be applied to the *eutectic*, *particle1* and *particle2* images. Load *eutectic* in imbin1 and enter:

```
>>> c1 = covariance(imbin1, 2, 81)
```

The covariance measure in the range [0, 80] is stored in the list c1. Then, do the same with *particle1* stored in c2:

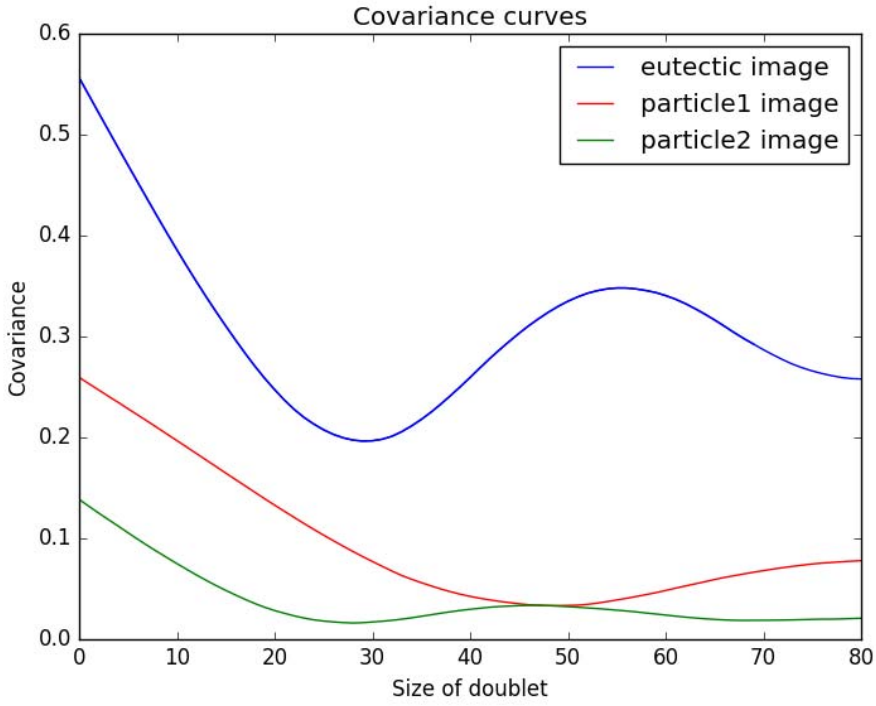
```
>>> c2 = covariance(imbin1, 2, 81)
```

And with *particle2*, in c3:

```
>>> c3 = covariance(imbin1, 2, 81)
```

The curves can then be plotted with **matplotlib** (assumed that you have installed it). Just enter the following commands:

```
>>> import matplotlib.pyplot as plt
>>> x = range(81)
>>> err = plt.xlabel("Size of doublet")
>>> err = plt.ylabel("Covariance")
>>> err = plt.title("Covariance curves")
>>> err = plt.plot(x, c1, label="eutectic image", color='blue')
>>> err = plt.plot(x, c2, label="particle1 image", color='red')
>>> err = plt.plot(x, c3, label="particle2 image", color='green')
>>> err = plt.legend(loc="upper right")
>>> err = plt.show()
```



Plotting of the covariance curves.

The 'err' variable is simply here to swallow an ugly string returned by the **matplotlib** functions.

2) As shown by the covariance definition, C(0) equals the ratio of set X. C(∞) is the probability for a couple of points at an infinite distance from each other to be both included in X. This probability is equal to the probability of inclusion for two independent points (non correlated occurrences). That is:

$$C(\infty) = C^2(0)$$

This can be verified with the covariance of *particle2*. The curve is relatively constant for the large size and we have C(0) = 0.1386, C^2 (0) = 0.0192 and C(80) = 0.0208.
The tangent of the curve at the origin is equal in absolute value to -C'(0). An estimate of this value is given by:

$$-C'(0) = C(0) - C(1)$$

$$-C'(0) = \frac{mes(X \cap D') - mes[(X \ominus \check{K}_1) \cap D']}{mes(D')}$$

with $D' = D \ominus K_1$ (K_1 is the doublet of points of size 1, that is the elementary digital segment).
The numerator can be written:

$$mes[(X \cap D')/[(X \ominus \check{K}_1) \cap D']] = mes[X \cap (X \ominus \check{K}_1)^c \cap D']$$

i.e.:

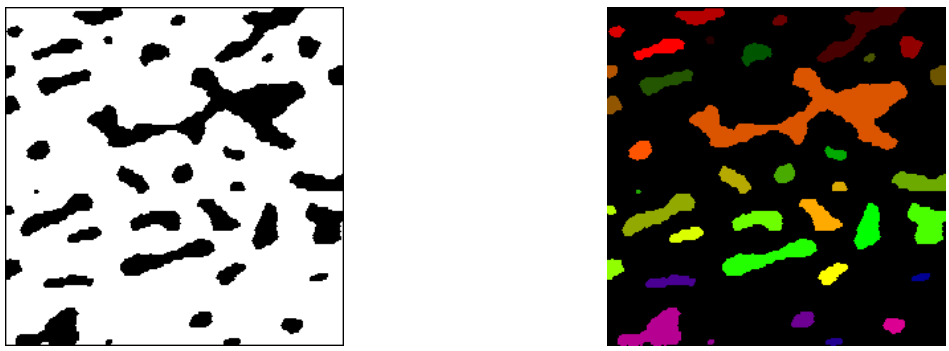
$$mes(X \cap X^c_{-1} \cap D')$$

But $X \cap X^c_{-1}$ is the set X eroded by the structuring element 01, used to compute the number of horizontal intercepts. -C'(0) is then an unbiased estimate of the vertical diameter of X.
We can see how the periodical structure of the *eutectic* image is reflected by the covariance curve. The minimum of the *particle2* curve indicates a repulsion phenomenon between the particles of the images: clusters of particles are separated by a minimal distance.

Exercise n° 4: Individual analysis of particles

1) The **label** operator, applied to a binary image, assigns to each connected component a single value. Apply it to the *metal1* image loaded in imbin1:

```
>>> label(imbin1, im32_1)
38
```



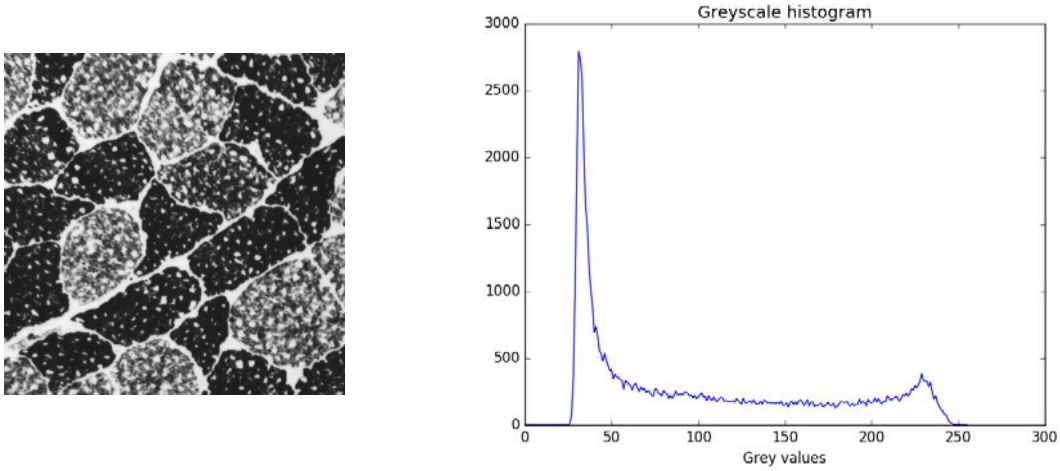
The metal1 image (left) is labelled: a single value is assigned to each connected component (right image displayed with the patchwork palette).

The labelled image is in the 32-bit image im32_1, as any image may contain more than 255 particles. The label function returns also the number of connected components in the image (38 in this case).
The histogram of a greyscale (8-bit) image is computed by the **getHistogram** function. This function returns a list of 256 integer values. The i-th component of the list indicates how many pixels with the grey value i are contained in the image. To get the histogram of image *muscle* loaded in im1, enter:

```
>>> histo =getHistogram
```

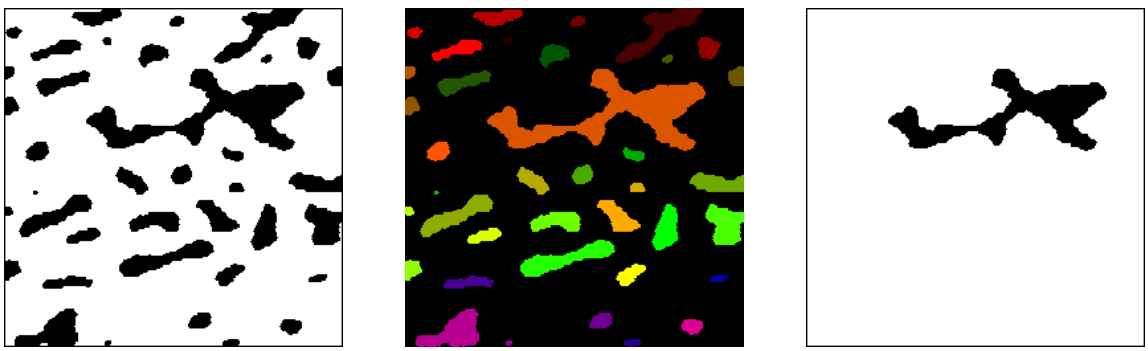
This histogram can be plotted with **matplotlib**:

```
>>> import matplotlib.pyplot as plt
>>> x = range(256)
>>> err = plt.xlabel("Grey values")
>>> err = plt.title("Greyscale histogram")
>>> err = plt.plot(x, histo)
>>> err = plt.show()
```



Grey level histogram of the muscle image.

2) As the label operator assigns a single value to each connected component of the labelled set, a simple threshold of the labelled image at level [i, i] extracts the i-th particle of the image.



Thresholding the labelled image (middle) at 12 extracts the twelfth particle (right) of the initial binary image (left).

```
>>> nb = label(imbin1, im32_1)
>>> threshold(im32_1, imbin2, 12, 12)
```

It is then possible to compute measures for all the particles of a binary set by extracting each of them in a loop. For instance, we can measure the horizontal and vertical sizes of the bounding boxes of the particles included in a binary image with the **particleBoundingBoxes** operatorn (which can be found in the Mamba_solutions.py module). .

```
from mamba import *

def particleBoundingBoxes(imIn):
    """
    This operator returns a list containing the horizontal and vertical dimensions
    (also called Feret diameters) of all the particles contained in the binary image
    'imIn'.
    """

    # Defining working images
    imWrk = imageMb(imIn)
    imWrk32 = imageMb(imIn, 32)
    # Initializing the list of results
    feretDiametersList = []
    # Labelling the binary image. nb contains the number of particles
    nb = label(imIn, imWrk32)
    # This loop calculates the size of the bounding box for each connected component
    for i in range(nb):
        threshold(imWrk32, imWrk, i+1, i+1)
        box = computeFeretDiameters(imWrk)
        feretDiametersList.append(box)
    # Returning the list of results
    return feretDiametersList
```

When applying this operator to the image *metal1* loaded in imbin1, we get:

```
>>> boxes = particleBoundingBoxes(imbin1)
>>> boxes
[(67.0, 41.0), (37.0, 12.0), (10.0, 7.0), (12.0, 8.0), (6.0, 5.0), (17.0, 16.0), (38.0, 15.0), (23.0,
17.0), (8.0, 6.0), (7.0, 9.0), (11.0, 14.0), (164.0, 61.0), (40.0, 19.0), (11.0, 13.0), (17.0, 14.0),
(16.0, 10.0), (15.0, 14.0), (24.0, 20.0), (40.0, 14.0), (11.0, 6.0), (3.0, 2.0), (55.0, 31.0), (32.0,
23.0), (20.0, 34.0), (28.0, 30.0), (6.0, 6.0), (38.0, 14.0), (25.0, 11.0), (71.0, 27.0), (12.0,
15.0), (22.0, 16.0), (13.0, 5.0), (37.0, 9.0), (47.0, 28.0), (17.0, 11.0), (16.0, 11.0), (12.0, 5.0),
(4.0, 3.0)]
```

This procedure performs an individual analysis of the particles of a binary image. You can obviously obtain any measure by using the same programming template and adapting it.

3) The particle areas can be computed by means of the procedure described previously. We just need to replace the Feret measurement by the area measurement (**computeArea**). However, the computation time is proportional to the number of particles in the image. A faster procedure can be designed with the help of the **getHistogram** operator. To describe this procedure, we suppose, in a first approach, that the number of particles in the image is less than 256. In this case, the area of all the particles can be obtained thanks to the following

steps (the binary image is loaded in imbin1). We start by labelling the imbin1 image (the labelled image is stored in im32_1):

```
>>> nb = label(imbin1, im32_1)
```

Then, after having verified that nb is less than 256), we extract the lower byte plane (its index is equal to 0) of im32_1 in the greyscale image im1 with the **copyBytePlane** function:

```
>>> copyBytePlane(im32_1, 0, im1)
```

Now, computing the histogram of the labelled image returns a list of 256 values, the i-th value (i different of 0) corresponding to the number of pixels of the i-th particle:

```
>>> histo = getHistogram(im1)
>>> histo
[52322, 906, 313, 71, 76, 34, 216, 312, 259, 42, 68, 155, 3501, 385, 124, 196, 116, 179,
265, 447, 71, 10, 648, 401, 463, 622, 42, 427, 197, 801, 150, 214, 67, 255, 780, 170, 151,
62, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The trailing zero values correspond to grey values which do not exist in the label image (no more particle available). Finally, if we keep only the values of the list in the range (1, nb+1), we obtain the list of all the particle areas (measured in number of pixels).

```
>>> areaList = histo[1 : nb+1]
>>> areaList
[906, 313, 71, 76, 34, 216, 312, 259, 42, 68, 155, 3501, 385, 124, 196, 116, 179, 265, 447,
71, 10, 648, 401, 463, 622, 42, 427, 197, 801, 150, 214, 67, 255, 780, 170, 151, 62, 18]
```

This procedure is obviously much faster than the one extracting each component one by one. If the number of particles is higher than 255, this approach can still be used provided that the labelled image is processed in several steps. This more complex procedure, which remains however faster than the individual approach, is out of the scope of this exercise. A more detailled description of the complete procedure can be found in [2].

REFERENCES

[1] S.Beucher: Measures in Mamba
(**http://cmm.ensmp.fr/~beucher/publi/Measures in Mamba.pdf**)
A description of the main stereological measures available in MAMBA.

[2] S.Beucher: Labelling Operators
(**http://cmm.ensmp.fr/~beucher/publi/Labelling Operators.pdf**)
This paper describes some algorithmic implementations of labellings applied on sets or partitions. These implementations take advantage of specific charactreistics of some

MAMBA operators. The purpose of this document is also to show that operations which are often implemented with graphs can be efficiently and rapidly performed directly on images.

Chapter 4

OPENINGS, CLOSINGS

1 Openings, closings, definitions

1.1 Binary case

Erosion and dilation lead to the definition of two new transformations, the opening γ and the closing φ:

$$\gamma(X) = (X \ominus \check{B}) \oplus B$$
$$\varphi(X) = (X \oplus \check{B}) \ominus B$$

Note that the first operation is performed with the structuring element B and the second with the transposed structuring element $\check{B}$, otherwise the resulting set would be shifted and the opening would not be anti-extensive (extensive for the closing).

1.2 Greytone case

The opening and the closing are defined similarly:

$$\gamma(f) = (f \ominus \check{B}) \oplus B$$
$$\varphi(f) = (f \oplus \check{B}) \ominus B$$

1.3 Properties, size distribution

The opening is an increasing and anti-extensive operation. The closing is increasing and extensive. The two transformations are idempotent.

If λB denotes the homothetic set of a convex set B, the opening is a size distribution. In particular:

$$if\ \lambda > \mu,\ [(X)_{\lambda B}]_{\mu B} = \left[(X)_{\mu B}\right]_{\lambda B} = (X)_{\lambda B}$$
$$[(f)_{\lambda B}]_{\mu B} = \left[(f)_{\mu B}\right]_{\lambda B} = (f)_{\lambda B}$$

The opening and the closing are used for size distribution analysis on the one hand, and for filtering on the other hand. These two kinds of use are illustrated in the exercises.

EXERCISES

Exercise n° 1

1) In order to program the following transformations, use the ones (erosions, dilations) already generated in the MAMBA library (**erode**, **dilate**, **linearErode**, **linearDilate**):
- opening and closing by a hexagon of size n,
- opening and closing by a segment of size n.

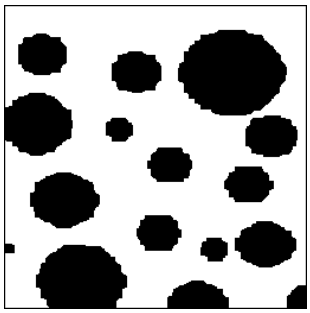
Compare your results of the two first transformations to the MAMBA operators (**opening**, **closing**, **linearOpen**, **linearClose**).

2) Define the opening and closing by a doublet of points at distance n with MAMBA.

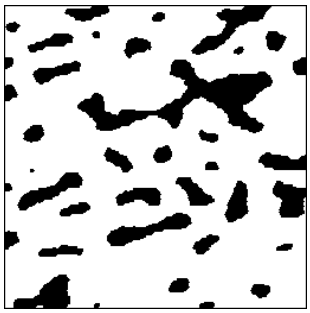
3) Verify the properties stated above. Prove in particular the duality by complementation of opening and closing. Verify also that the opening by a doublet of points is not a size distribution (openings of small size suppress more points than openings of greater size).

Exercise n° 2

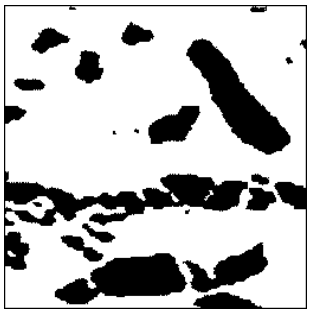
The purpose of this exercise is to get you used to the behavior of opening and closing. So, feel free to use the transformations introduced in the previous exercise and apply them to the provided images (*grains1*, *grains2*, *particle1*, *particle2*, *balls*, *metal1*, *metal2*, *salt*, *knitting*, *muscle*). To help you in your quest, here are some guidelines.



balls

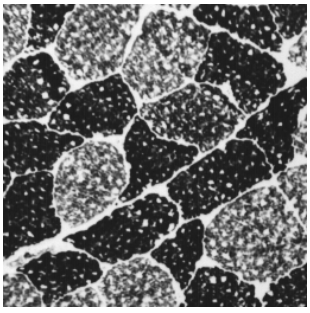


metal1

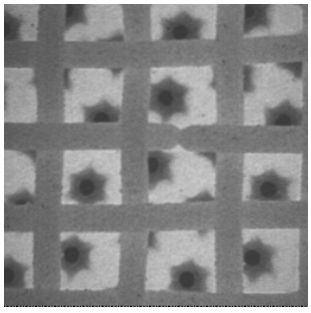


metal2

1) Verify the behavior of the opening with convex structuring elements as a size distribution, by applying transformations of increasing size to the images *grains2*, *balls*, *salt*, *knitting*, *muscle*.
2) The notion of size distribution (or granulometry) is not related to the notion of particle. Closings allow to perform (by duality) the size distribution of pores, and thus to reveal the spatial distribution of the connected components of a set (images *particle2*, *salt*, *knitting*, *muscle*).
3) Both opening and closing "sieve" the connected components of a set according to their size, but also according to their shape. You can observe this by performing hexagonal openings of increasing size on the image *balls*. Besides, opening "hexagonalizes" the remaining connected components. Then, define the opening by a dodecagon and compare it with the hexagonal opening of same size.
Note this selective effect on a greytone image by performing openings by segments of increasing size (images *circuit* and *burner*).

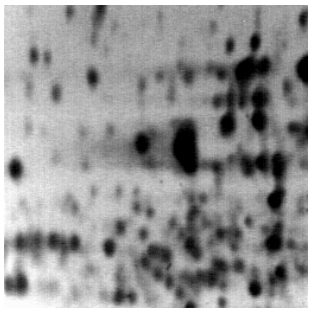


muscle



burner

noise



electrop

Exercise n° 3

Opening and closing are efficient tools for filtering noise in images. The *noise* image represents a set blurred by a scatter plot and the *electrop* image represents a blurred greytone image.
1) Open the image with an elementary hexagon. Do the same with closing.
2) Perform the two operations successively. Does the order matter?
3) Can you enhance the filtering? (use triangular or 2x2 square openings and closings).

Exercise n° 4: Generalization of the notion of opening

The notion of opening can be defined in a more general way. An algebraic opening is any transformation that is:
- increasing
- idempotent
- anti-extensive

The notion of algebraic closing is defined by duality: it is an increasing, idempotent and extensive transformation.
1) Give some examples of openings and closings.
2) Is the operation consisting in extracting particles with at least one hole an opening (binary case) ?
3) To obtain new openings or closings, consider a family (γ_i) of openings or (φ_i.) of closings. Prove that:
$\gamma = \sup_i \gamma_i$ resp. $\gamma = \bigcup_i \gamma_i$ is an opening.
$\varphi = \inf_i \varphi_i$ resp. $\varphi = \bigcap_i \varphi_i$ is a closing.
SupOpen is the MAMBA operator which performs the supremum of openings by segments in the directions of the grid and **infClose** is the corresponding closing. Try them.
4) is the inf of openings an algebraic opening?
5) Observe on the *circuit* and *tools* images that the new openings and closings have a different selective effect compared with those described up to now.
6) How to compute the area opening of a binary set?

Exercise n° 5

F(λ) denotes the ratio of the opening by a disk of size λ of a set X.
1) Verify that:

$$G(\lambda) = \frac{F(0) - F(\lambda)}{F(0)}$$

is always comprised between 0 and 1. Compute it using the area of the opened set and the area of the eroded field of measurement D' ($D' = D \ominus 2\lambda B$).
2) Program this measurement. Application to the size distribution of the *metal1* and *metal2* images.

SOLUTIONS

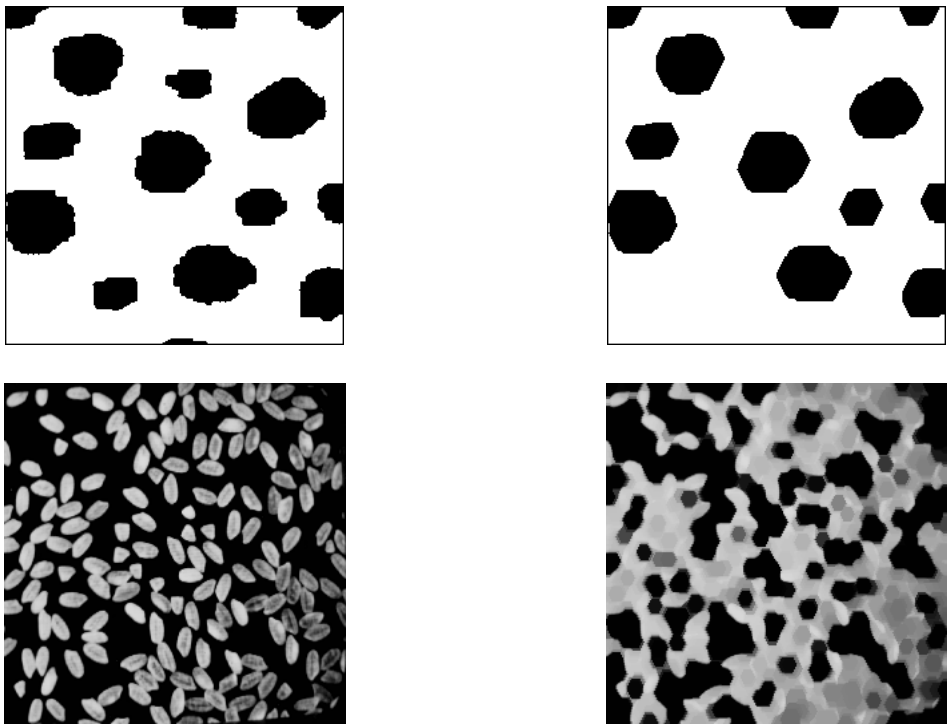
Exercise n° 1

1) You can simply obtain an opening (or a closing) by concatenating an erosion and a dilation of the same sizes. For instance:

```
>>> erode(imbin1, imbin2, 12)
>>> dilate(imbin2, imbin2, 12)
```

or (closing of the greyscale image *rice*):

```
>>> dilate(im1, im2, 5)
>>> erode(im2, im2, 5)
```



Top, opening of size 12 of the particle1 image. Bottom, closing of size 5 of the rice image.

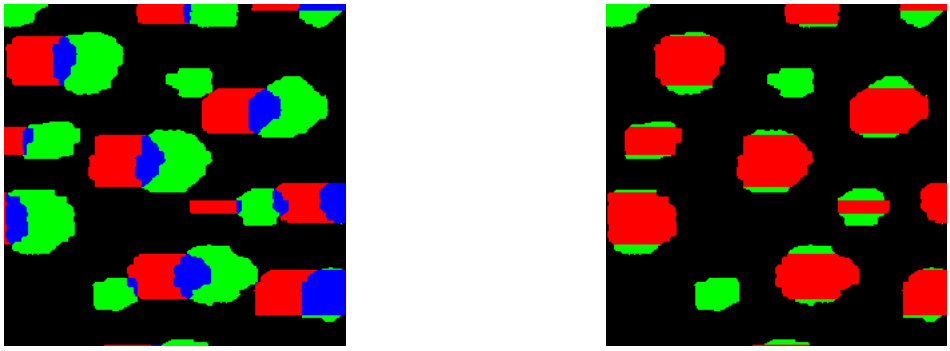
Do not forget, however, to transpose your structuring element if it is not isotropic. Otherwise, the transformation would not be anti-extensive or extensive. Let us illustrate this with an linear erosion of size 35 in direction 2 (hexagonal grid) followed by a linear dilation in the same direction applied to the *particle1* image:

```
>>> linearErode(imbin1, imbin2, 2, 35)
```

```
>>> linearDilate(imbin2, imbin2, 2, 35)
```

This operator is not anti-extensive. On the contrary, if we use the transposed direction for the dilation, the anti-extensivity is verified:

```
>>> linearErode(imbin1, imbin2, 2, 35)
>>> linearDilate(imbin2, imbin2, 5, 35)
```



If non transposed structuring elements are used, the resulting operation is not anti-extensive: the blue part is the intersection of the initial and final images. If transposed structuring elements are used (right), the transform (in red) is included in the initial image.

Two operators are available in MAMBA to transpose either a structuring element or a direction, **transpose** and **transposeDirection**:

```
>>> TRIANGLE
structuringElement([0, 3, 4], mamba.HEXAGONAL)
>>> TRIANGLE.transpose()
structuringElement([0, 1, 6], mamba.HEXAGONAL)
>>> transposeDirection(3, SQUARE)
7
```

General openings and closings already exist in MAMBA. The operators are named **opening** and **closing**. Linear (the structuring element is a digital segment) openings and closings also exist: **linearOpen** and **LinearClose**.
Another important point with openings and closings concerns the setting of the edge. In both operators, you can set the edge to **FILLED** (default) or **EMPTY**. In any case, this setting has no effect on the dilation which is always performed with the edge set to **EMPTY**. The setting of the edge only affects the erosion. An opening with a **FILLED** edge can be considered as a geodesic opening where the geodesic space corresponds to the field D of the image (see chapter 6 on geodesy for further explanations). When edge is set to **EMPTY**, be aware that the result is biased on the edge but it is still an opening (that is an increasing, anti-extensive and idempotent transform, see exercise 4). In the case of the closing, when edge is set to **EMPTY**, the result is not extensive and cannot be considered as an algebraic closing. In order to cope with this problem, the sup (or union in the binary case) of the result and the original image is performed. This corrected transform is an algebraic closing. Indeed, this transform is extensive by construction. It is also an increasing one, being composed of dilations, erosions and boolean operators whic are increasing. Let us prove that the corrected closing transform is idempotent.
X is supposed to be included in the mask D ($X \cap D = X$). The operation is performed in three steps:

- Dilation of X by B and intersection with D (the outside part is lost):

$$(X \oplus \check{B}) \cap D$$

- Erosion of the result by the transposed structuring element:

$$[(X \oplus \check{B}) \cap D] \ominus B$$

- The initial set is added (union) to the previous result:

$$[[(X \oplus \check{B}) \cap D] \ominus B] \cup X = Y$$

Let us apply the same operation on the resulting set Y:

$$(Y \oplus \check{B}) = [[[(X \oplus \check{B}) \cap D] \ominus B] \cup X] \oplus \check{B}$$
$$= \left[[[(X \oplus \check{B}) \cap D] \ominus B] \oplus \check{B}\right] \cup (X \oplus \check{B})$$
$$= [(X \oplus \check{B}) \cap D]_{\check{B}} \cup (X \oplus \check{B})$$

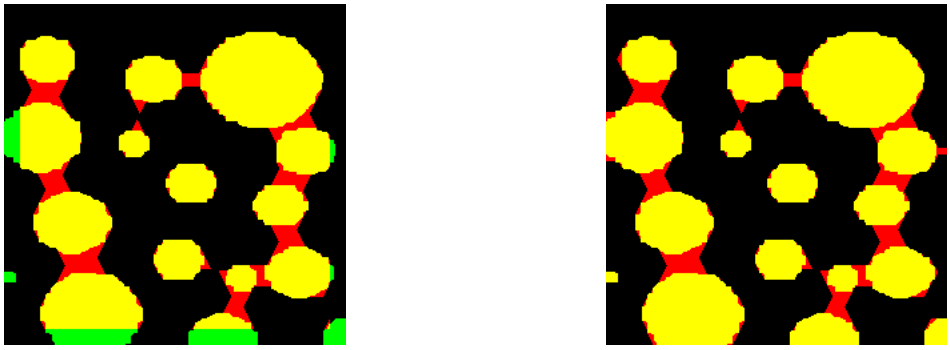
Then we intersect with D:

$$(Y \oplus \check{B}) \cap D = \left[[(X \oplus \check{B}) \cap D]_{\check{B}} \cap D\right] \cup [(X \oplus \check{B}) \cap D]$$

The first term is equal to $[(X \oplus \check{B}) \cap D]_{\check{B}}$, as $[(X \oplus \check{B}) \cap D]$ is included in D and its opening by $\check{B}$ also. We have:

$$(Y \oplus \check{B}) \cap D = [(X \oplus \check{B}) \cap D]$$
$$[(Y \oplus \check{B}) \cap D] \ominus B = [(X \oplus \check{B}) \cap D] \ominus B$$
$$[[(Y \oplus \check{B}) \cap D] \ominus B] \cup Y = [[(X \oplus \check{B}) \cap D] \ominus B] \cup [[[(X \oplus \check{B}) \cap D] \ominus B] \cup X]$$
$$[[(Y \oplus \check{B}) \cap D] \ominus B] \cup Y = [[(X \oplus \check{B}) \cap D] \ominus B] \cup X = Y$$

This operation is idempotent.



Closing without correction, when the edge is set to EMPTY (left). The operator is not extensive (green part should not exist). Corrected closing (right).

This correction is included in the MAMBA closing operators (**closing** and **linearClose**).

2) The opening and closing by a doublet of points are not included in the MAMBA library. They can be defined as following:

```
from mamba import *

def doublePointOpen(imIn, imOut, dir, n, grid=DEFAULT_GRID, edge=FILLED):
  """
  Performs an opening by a doublet of points of size 'n' in direction 'dir'.
  'edge' is set to 'FILLED' by default.
  """

  doublePointErode(imIn, imOut, dir, n, edge=edge, grid=grid)
  doublePointDilate(imOut, imOut, transposeDirection(dir, grid=grid), n, grid=grid)

def doublePointClose(imIn, imOut, dir, n, grid=DEFAULT_GRID, edge=FILLED):
  """
```

```
    Performs a closing by a doublet of points of size 'n' in direction 'dir'.
    If 'edge' is set to 'EMPTY', the operation must be modified to remain extensive.
    """

    imWrk = imageMb(imIn)
    if edge==EMPTY:
        copy(imIn, imWrk)
    doublePointDilate(imIn, imOut, dir, n, grid=grid)
    doublePointErode(imOut, imOut, transposeDirection(dir, grid=grid), n, edge=edge,
    grid=grid)
    if edge==EMPTY:
        logic(imOut, imWrk, imOut, "sup")
```

3) Duality by complementation of opening and closing. It is possible to write:

$$(X)_B = (X \ominus \check{B}) \oplus B = (X^c \oplus \check{B})^c \oplus B$$
$$= ((X^c \oplus \check{B}) \ominus B)^c = \left[(X^c)^B\right]^c, \text{ Q.E.D.}$$

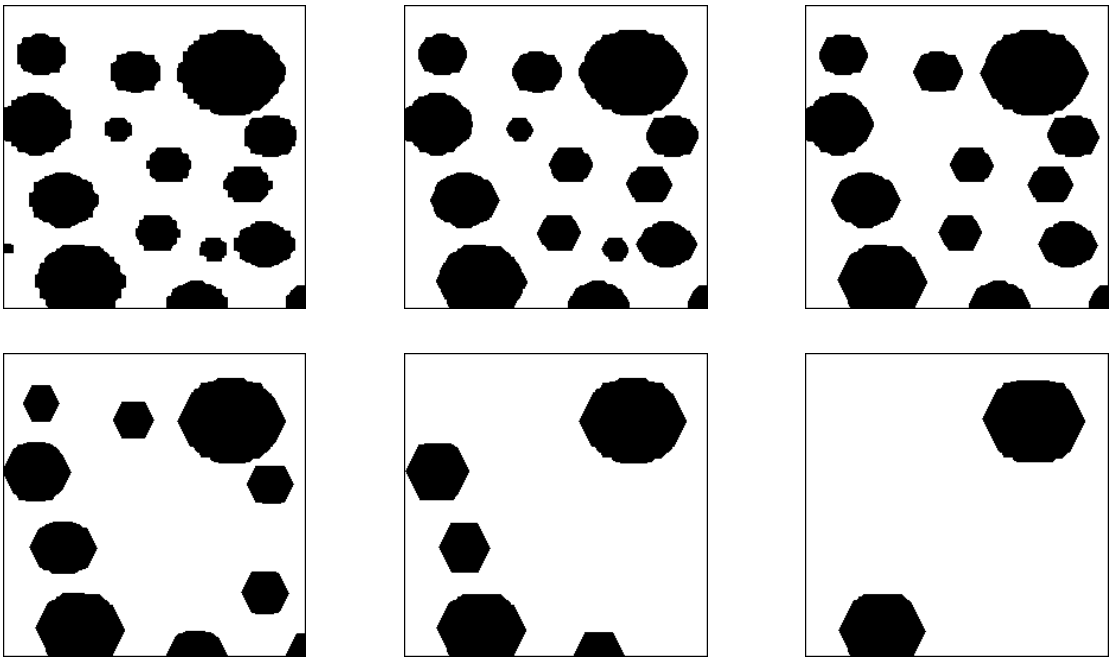
The opening by a doublet of point is not a size distribution (granulometry). Indeed, if you perform an opening with a doublet of points at a given distance, then another one with a doublet at a larger distance, the first operation may remove more points than the second one. To verify this, load image *particle2* in imbin1 then enter the following commands:

```
>>> doublePointOpen(imbin1, imbin2, 2, 30)
>>> computeVolume(imbin2)
2908L
>>> doublePointOpen(imbin1, imbin2, 2, 45)
>>> computeVolume(imbin2)
4250L
```

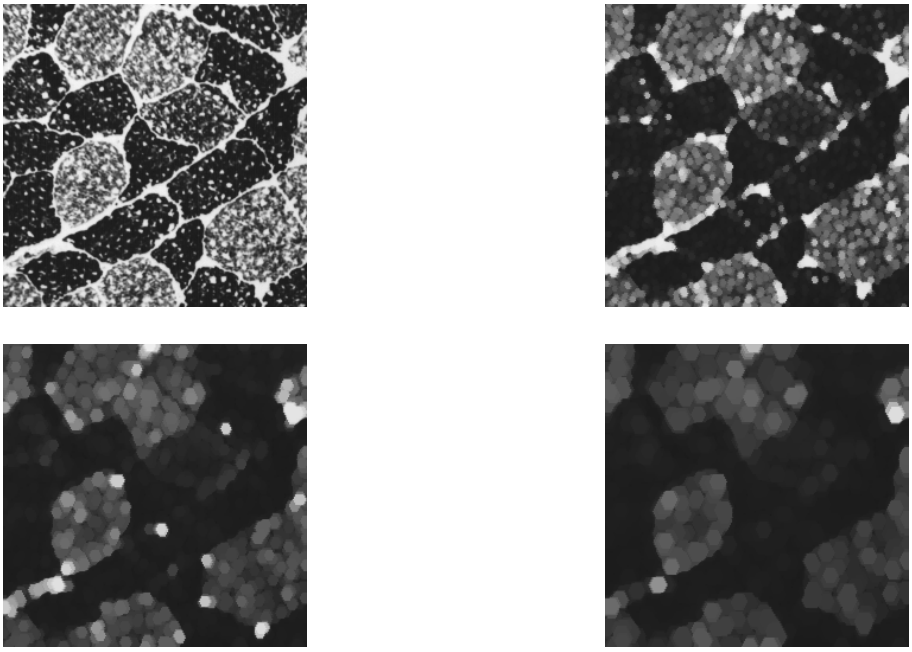
Initial particle2 image (left). Opening by an horizontal doublet of points at distance 30 (middle) and at distance 45 (right).

Exercise n° 2

1) Openings by convex homothetics as a size distribution (or granulometry)
The images below illustrate the use of the opening as a size distribution (granulometric) transformation. Transforms of increasing size act as a sieve, by removing more and more particles according to their size. Note also that the remaining ones are modified, their initial shape is not preserved.
Sizes distributions can also been applied on greytone images. The sieving and filtering effect is clearly visible on the *muscle* image where increasing openings tend to remove the white features in the image.



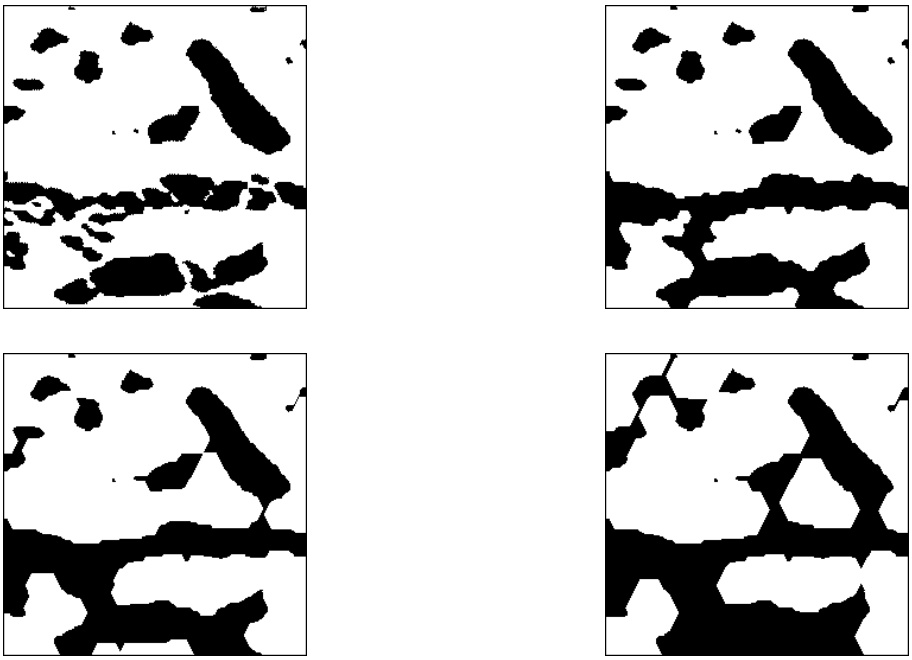
From top to bottom and from left to right: hexagonal openings of increasing sizes (5 by 5) of the balls image.



Successive openings of the muscle image show the progressive disappearance of the white dots.

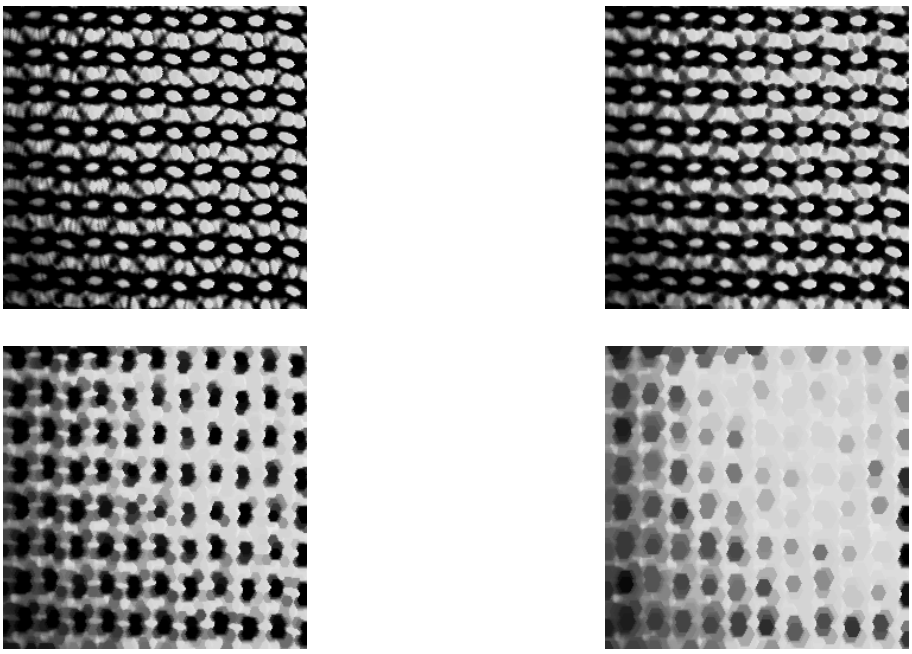
2) Size distribution by closing

The closing, which is the dual transformation with respect to the opening can be used to perform the size distribution of the space between particles. Thus, we use a property of the morphological notion of granulometry, which is to be defined without refering to the concept of particle (or connected component). Applied on binary images, the closing provides information on the spatial scattering of the particles by progressively connecting them.



Successive closings of the metal2 image. The particles are progressively connected and the clusters are emphasized.

Applied to a greyscale image, the closings filter the dark features (they fill them) and concatenate the white ones.

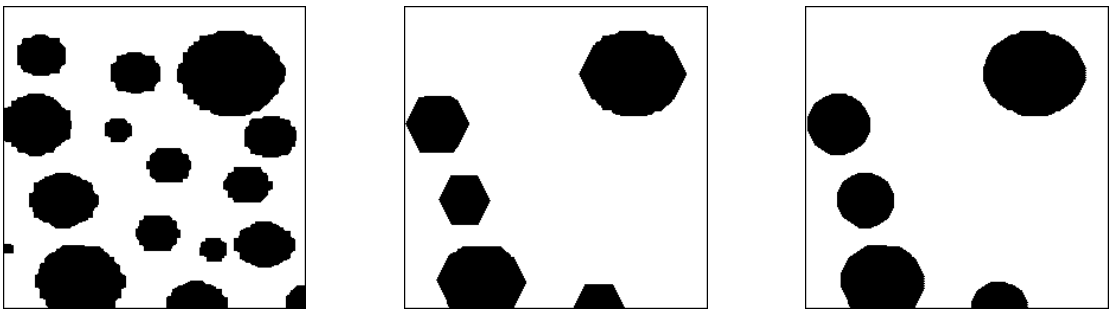


Successive hexagonal closings of the knitting image.

3) Influence of the shape on the opening

We already noticed the importance of the shape of the structuring element in the opening and closing operations. It is particularly obvious in the previous example where successive hexagonal openings were applied to the balls image. As their size increases, the remaining particles become more and more hexagon-shaped. To compare openings performed with hexagons and dodecagons, the dodecagonal opening must be defined. The MAMBA operator, **dodecagonalOpening** is designed as such:

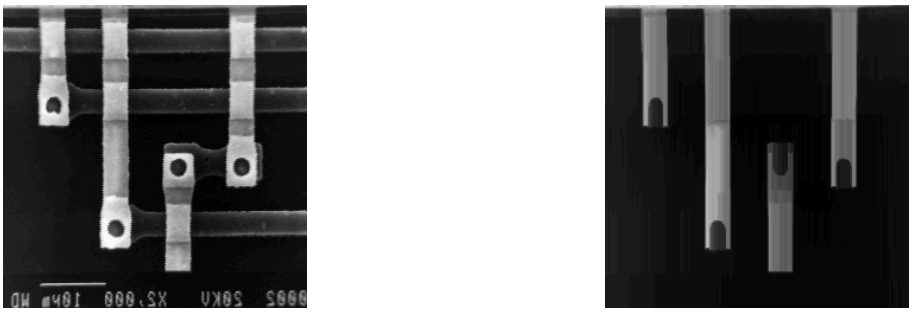
```
from mamba import *

def dodecagonalOpening(imIn, imOut, n=1, edge=FILLED):
  """
   Performs an opening operation on image 'imIn' with a dodecagon and puts the result in 'imOut'.
  'n' controls the size of the opening.
  The default edge is set to 'FILLED'. Note that the edge setting operates in the
  erosion only.
  """

  dodecagonalErode(imIn, imOut, n, edge=edge)
  dodecagonalDilate(imOut, imOut, n, edge=EMPTY)
```



Hexagonal (middle) and dodecagonal (right) openings of same size of the balls image (left).

Dodecagonal openings are smoother than hexagonal ones.
The shape of the structuring element plays an important role for selecting the appropriate features in an image. For instance, applying openings by vertical segments of great size on the *circuit* image preserves the vertical features whereas the horizontal ones are filtered:

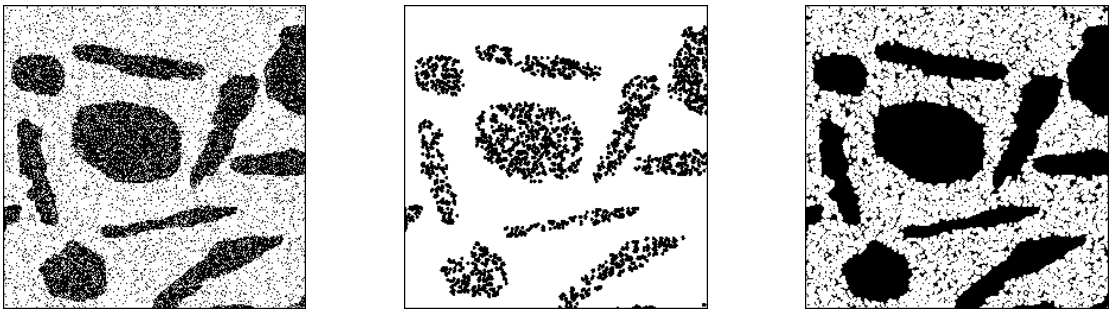
```
>>> linearOpen(im1, im2, 1, 50, grid=SQUARE)
```



Opening by a vertical segment of size 50 (right) of the circuit image (left). The vertical features are more or less preserved, the horizontal ones are filtered.

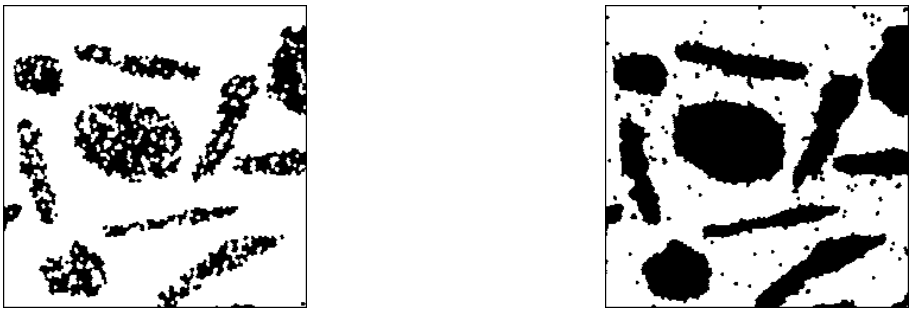
Exercise n° 3

1) Hexagonal opening and closing
The following images show the result of an hexagonal opening and closing on the image *noise*.



Hexagonal opening (middle) and closing (right) of the noise image (left).

2) Sequence of operations
In a sequence of operations, the order is important, as shown by the two images below.



Left image, hexagonal opening followed by an hexagonal closing. Right image, the order of operations is reversed.

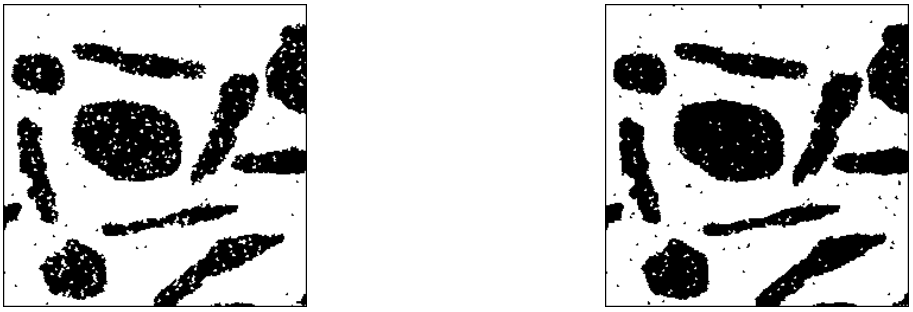
3) The triangle and 2x2 square structuring elements are already defined in MAMBA. They are named respectively **TRIANGLE** and **SQUARE2X2**. Triangular openings and closings are obtained by entering:

```
>>> opening(imbin1, imbin2, se=TRIANGLE)
>>> closing(imbin2, imbin2, se=TRIANGLE)
```

or:

```
>>> closing(imbin1, imbin2, se=TRIANGLE)
>>> opening(imbin2, imbin2, se=TRIANGLE)
```

for the reverse order.



Left image, triangular opening followed by a triangular closing. Right image, the order of operations is reversed.

The hexagonal filtering is stronger than the triangular one. The hexagonal opening always removes more points than the triangular one. However, its action is more rough. This is why, with triangular operators, the order of operations still matters, yet to a lesser extent.
In order to enhance filtering, the same operations should be performed with a transposed triangle.

It is possible to define filters where openings and closings of increasing sizes are alternated. These filters exist in the MAMBA library. They are named **alternateFilter** (a simple alternance opening/closing or closing/opening) and **fullAlternateFilter** (a sequence made of simple alternate filters of increasing sizes). Try them on the *noise* image loaded in imbin1 by entering:

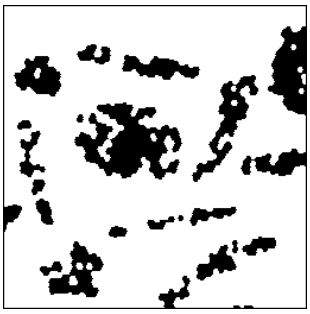
```
>>> fullAlternateFilter(imbin1, imbin2, 2, True)
>>> fullAlternateFilter(imbin1, imbin2, 2, False)
```

The first operation is a full alternate filter of size 2 starting with an hexagonal opening (the parameter 'openFirst' in the operator is set to True):

$$\varphi_2 \circ \gamma_2 \circ \varphi_1 \circ \gamma_1$$

In the second one, the first operation is a closing:

$$\gamma_2 \circ \varphi_2 \circ \gamma_1 \circ \varphi_1$$




Full alternate filter of size 2: left image, the filter starts with an hexagonal opening, right image, the filter starts with an hexagonal closing.

Exercise n° 4: Generalization of the notion of opening

1) The operation consisting in filling the holes of a binary image is a closing. The operation consisting in extracting the particles whose area is greater than N is an opening, called area opening. Conversely, the operation consisting in extracting the particles whose surface is smaller than N is not an opening (not increasing). The operation where the particles are contained in a bounding box larger than a given size (N, N) is an opening (sometimes called attribute opening).
The truncation of a function at level λ (all that is higher is reduced to value λ) is an opening (it is even a size distribution).
2) The transformation ψ which selects particles with at least one hole is not an opening, because the operation is not increasing. Take, for instance, a connected component X that does not contain a hole and take $Y \subset X$ equal to the interior contour of X. We have:

$$\Psi(X) = \varnothing \; ; \; \Psi(Y) = Y$$

That is:

$$\Psi(X) \subset \Psi(Y)$$

3) Let us prove that $\sup_i \gamma_i$ is an opening.

The transformation is increasing:

Let f < g.

$\forall i, \gamma_i(f) \leq \gamma_i(g)$ for γ_ι is an opening (hence increasing)

$\forall i, \gamma_i(f) \leq \sup_j \gamma_j(g)$

$\sup_i \gamma_i(f) \leq \sup_j \gamma_j(g)$ Q.E.D.

The transformation is anti-extensive:

$\forall i, \gamma_i(f) \leq f$

therefore $\left(\sup_i \gamma_i\right)(f) \leq f$ Q.E.D.

The transformation is idempotent:

Let us denote $\Psi = \sup_i \gamma_i$.

$\forall j, \Psi(f) \geq \gamma_j(f)$ by definition

$\forall j, \gamma_j(\Psi(f)) \geq \gamma_j(\gamma_j(f)) = \gamma_j(f)$

for γ_j is increasing and idempotent.

$\forall j, \left(\sup_k \gamma_k\right)(\Psi(f)) \geq \gamma_j(f)$ a fortiori.

then we have $\forall j, \Psi(\Psi(f)) \geq \gamma_j(f)$

then $\Psi(\Psi(f)) \geq \sup_j(\gamma_j(f))$

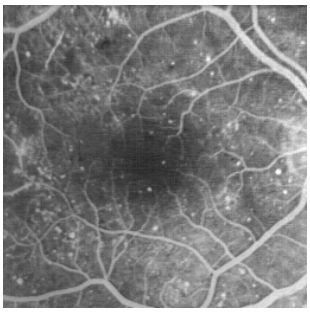
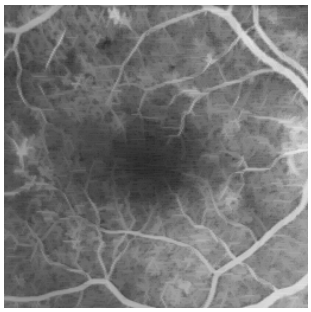
that is: $\Psi(\Psi(f)) \geq \Psi(f)$

hence $\Psi(\Psi(f)) = \Psi(f)$ since Ψ is anti-extensive. Q.E.D.

By a similar reasoning, we can prove that $\varphi = \inf_i \varphi_i$ is a closing.

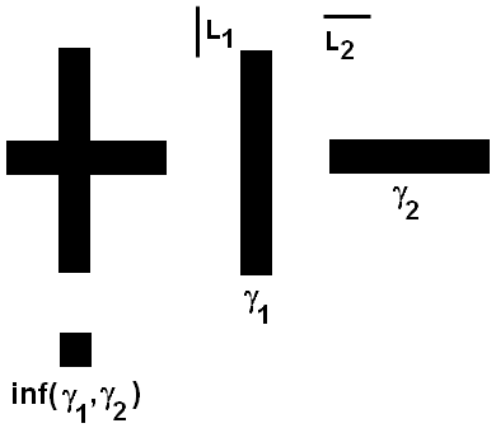
These transformations are named respectively **supOpen** and **infClose** in the MAMBA library:

```
>>> supOpen(im1, im2, 5)
```

Sup of linear openings of size 5 of the retina2 image. The white dots have been suppressed but the elongated blood vessels are preserved.

4) $\inf_i \gamma_i(f)$ is not an opening! Indeed, the transformation is increasing and anti-extensive but it is not idempotent. This is clearly shown in the following counter-example:



The inf of 2 openings is not an opening.

The openings γ_1 et γ_2 of the initial set (left cross) by L_1 and L_2 respectively are shown on the right. $\inf(\gamma_1, \gamma_2)$, intersection of the two openings, is the central square. The opening of this square by L_1 or L_2 is equal to the empty set.

5) The following images show the effect of isotropic closing compared with closing by intersection of linear closings. The difference between the two is particularly obvious in the case of elongated objects. The first transformation filters the objects according to their thickness, and the second is controlled by their elongation: an elongated object is preserved even if it is narrow.

```
>>> closing(im1, im2, 20)
>>> infClose(im1, im2, 20)
```

Initial image (left), hexagonal closing of size 20 (middle), sup of linear closings of size 20 (right).

6) We have already seen in a previous exercise (exercise n° 4, chapter 3) how to use the **label** and **getHistogram** operators to extract a particle from an image (provided that the image contains at most 255 particles). Therefore, an easy but slow solution would consist in extracting each particle and rejecting it if its area is less than N. But a more efficient approach is possible, based on the use of another MAMBA operator, **lookup**. To see how it works, load the *metal1* image in imbin1. Then enter the following commands:

```
>>> label(imbin1, im32_1)
38
>>> copyBytePlane(im32_1, 0, im1)
>>> histo = getHistogram(im1)
```

The binary image is labelled (we verify that it contains less than 256 connected components). Then the label image is converted into a greyscale (8-bit) one and its histogram is computed and stored in a list.

```
>>> histo[0] = 0
>>> area = 300
```

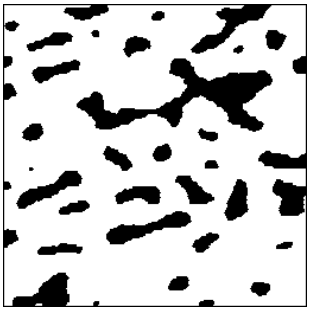
The first value of the histogram is forced to 0 as it corresponds to the area of the background. Then a variable 'area' is defined and set to 300. This variable contains the cutting value under which a particle is removed. The next command defines a new list 'areaThresh' where every histogram value (that is every area) is replaced by 0 if it is less than 'area' and by 255 if it is larger than or equal to 'area' (thanks to the list comprehension in Python, this can be achieved with a single coding line...):

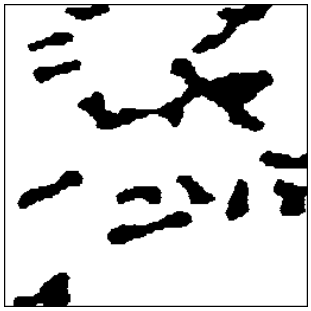
```
>>> areaThresh = [255*(x >= area) for x in histo]
```

You can print these two lists to compare them. Finally, enter:

```
>>> lookup(im1, im2, areaThresh)
```

lookup applies the look-up table 'areaThresh' to image im1 and stores the new image in im2. This operator replaces the grey value equal to i by a new value equal to areaThresh[i]. Thus, the particles with an area less than 'area' are replaced by 0 whilst the others are given the value 255. Extracting the latter particles is achieved by a simple thresholding.





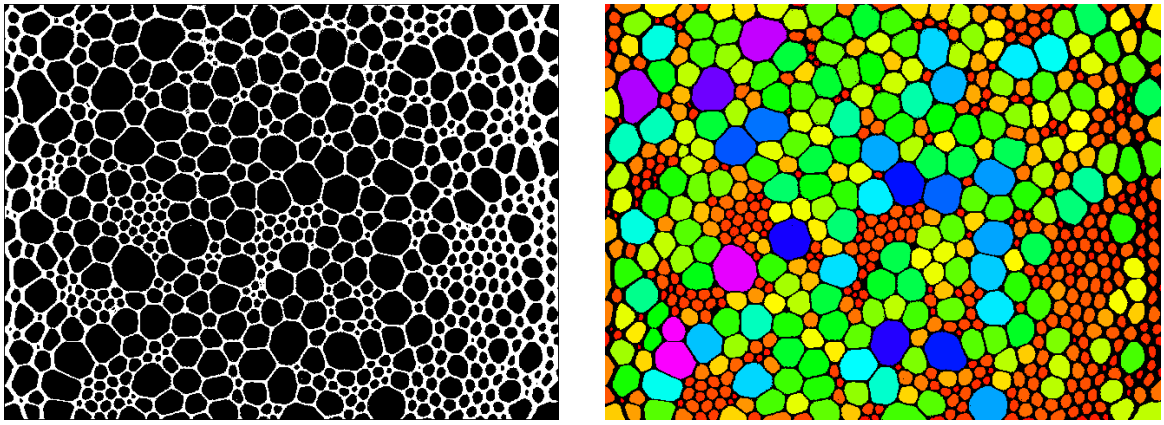
Particles of the metal1 image (left) are removed if their area is less than 300 (right).

This procedure, however, suffers from two drawbacks. Firstly, the lookup needs to be recalculated when changing the value of 'area'. Secondly, it does not work properly if the number of particles is higher than 255. Enhancing it is out of the scope of this exercise all the more so since the MAMBA library already contains the right operator. It is named **areaLabelling** and labels any particle of a set with a value equal to its area. The label image is obviously stored in a 32-bit image. Try this operator with the *binary_foam* image. Define a binary image of size (640, 480) and a 32-bit image of same size:

```
>>> imA = imageMb(640, 480, 1)
>>> imB = imageMb(imA, 32)
```

Then load the *binary_foam* image in imA and type:

```
>>> areaLabelling(imA, imB)
```



Area labelling of the left image. Each cell is labelled with a value equal to its area.

Exercise n° 5

1) Prove that G(λ) takes its values between 0 and 1.
Given:

$$G(\lambda) = 1 - \frac{F(\lambda)}{F(0)}$$

G(λ) is monotonous, since the opening is anti-extensive, with G(0) = 0. When $\lambda \to \infty$, $F(\lambda) \to 0$, $G(\lambda) \to 1$.
An estimate of G(λ) is given by:

$$G(\lambda) = \frac{mes(X \cap D') - mes(X_{\lambda B} \cap D')}{mes(X \cap D')}$$

or else:

$$G(\lambda) = \frac{mes[X/X_{\lambda B} \cap D']}{mes(X \cap D')} \quad (D' = D \ominus 2\lambda B)$$

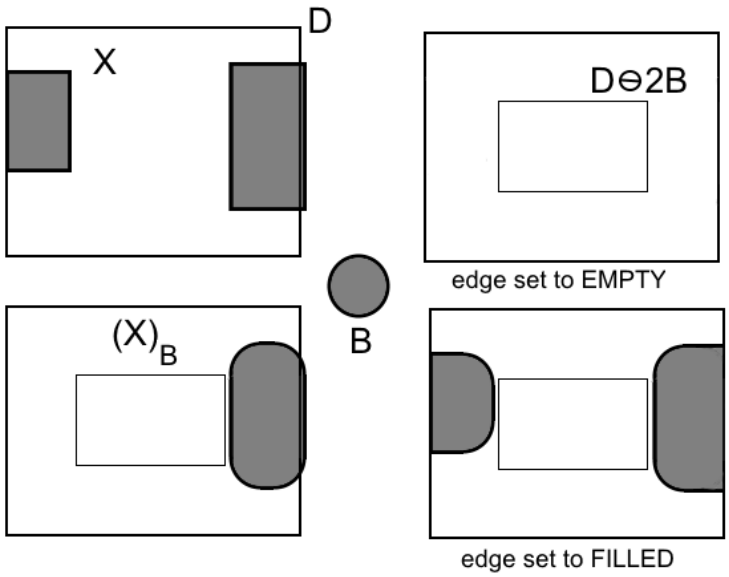
Remember that the measure of the ratio of the opened set X_λ is unbiased in the field D eroded by 2λB, as the opening is made of an erosion followed by a dilation.

2) Computation of G(n)
In a digital space, we have:

$$G(n) = \frac{mes[X/X_{nB} \cap D']}{mes(X \cap D')}$$

It may happen that, for high values of n, the denominator be equal to 0. But, as it is also the case for the numerator, G(n) is equal to 1. So the correct formula is:

$$G(n) = \frac{mes[X/X_{nB} \cap D']}{mes(X \cap D')} \text{ if } mes(X \cap D') \neq \varnothing$$
$$G(n) = 1 \text{ if not}$$



Upper left: The real set X is only known in the field D; lower left; true opening of X by B; upper right: result of the opening when edge is set to EMPTY; lower right: result of the opening when edge is set to FILLED.

Firstly, we can define the computation of G(n), named **granulometricMeasure**:

```
from mamba import *

def granulometricMeasure(imIn, n):
  """
  Granulometric measure (size distribution measure) of the binary image
  'imIn' after an hexagonal opening of size n.
  This function returns a real value between 0 and 1.
  """

  imWrk1 = imageMb(imIn, 1)
  imWrk2 = imageMb(imIn, 1)
  opening(imIn, imWrk1, n)
  diff(imIn, imWrk1, imWrk2)
  imWrk1.fill(1)
  erode(imWrk1, imWrk1, 2*n, edge=EMPTY)
  logic(imWrk2, imWrk1, imWrk2, "inf")
  measure1 = computeVolume(imWrk2)
  logic(imIn, imWrk1, imWrk2, "inf")
  measure2 = computeVolume(imWrk2)
  if measure2 <> 0:
     granulometry = float(measure1)/measure2
  else:
     granulometry = 0
  return granulometry
```

This operator returns a single measure corresponding to an opening of size n. Then, we can define the operator, named **granulometry**, allowing to compute this measure for a given range of values. This operator returns a list of real values:

```
def granulometry(imIn, size):
  """
  Computes the granulometry of binary image 'imIn' in the range (0, 'size' -1) by an
  hexagonal opening.
  """

  granuList =[]
  for i in range(size):
     mes = granulometricMeasure(imIn, i)
     granuList.append(mes)
  return granuList
```

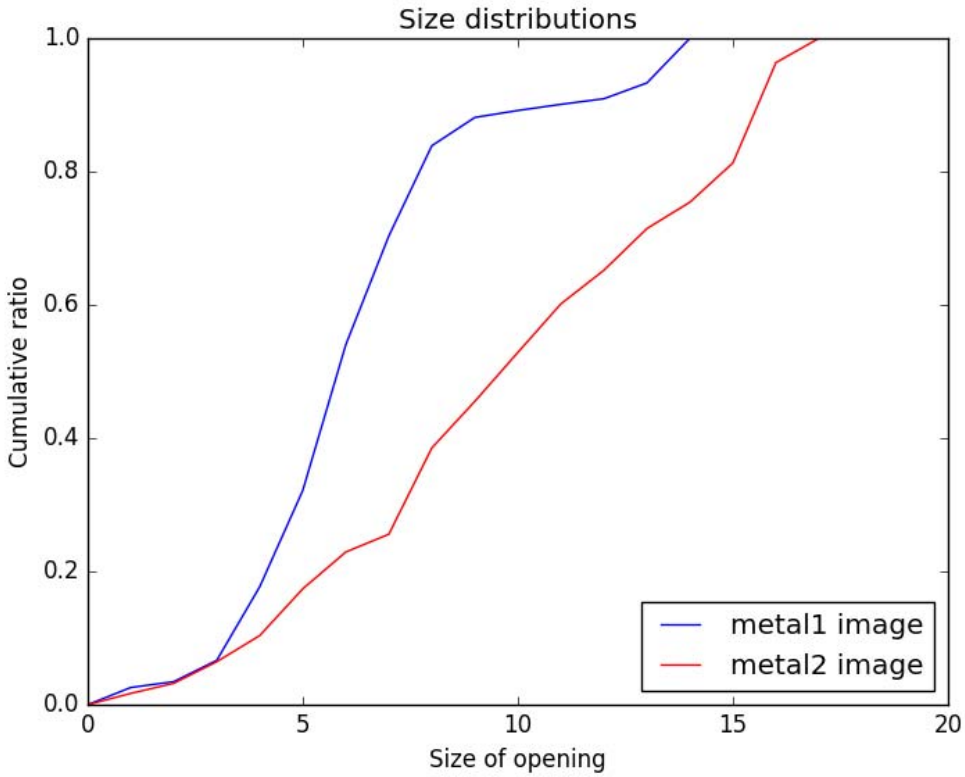
You can test this measure on the *metal1* image, loaded in imbin1:

```
>>> metal1 = granulometry(imbin1, 20)
```

The result is stored in the metal1 list. Then, load the *metal2* image and type:

```
>>> metal2 = granulometry(imbin1, 20)
```

to obtain the result in the metal2 list.



Size distributions (hexagonal openings) of the metal1 and metal2 images.

The two curves can be plotted:

```
>>> import matplotlib.pyplot as plt
>>> size = range(20)
>>> err = plt.xlabel("Size of opening")
```

```
>>> err = plt.ylabel("Cumulative ratio")
>>> err = plt.title("Size distributions")
>>> err = plt.plot(size, metal1, label="metal1 image", color="blue")
>>> err = plt.plot(size, metal2, label="metal2 image", color="red")
>>> err = plt.legend(loc="lower right")
>>> err =plt.show()
```

REFERENCES

[1] J.Serra: Courses on Mathematical Morphology, Opening, Closing (**http://cmm.ensmp.fr/~serra/cours/pdf/en/ch3en.pdf**)
This course gives a general presentation of the concepts of opening and closing. It is advised, before reading this chapter, to have a look on the previous ones (basic notions, erosion, dilation) also available at **http://cmm.ensmp.fr/~serra/acours.htm** to get familiar with the notations and the needed notions and concepts.

Chapter 5

MORPHOLOGICAL FILTERS

1 Reminder

Before we can extract any object from a greytone image, it is often necessary to enhance the image. The enhancement of an image is mainly obtained by a filtering operation. A morphological filter is a transformation ϕ which fulfils the two following properties:

(i) ϕ is increasing
(ii) ϕ is idempotent.

1.1 Sequential alternate filter

The white (resp. black) alternate sequential filter (ASF) consists, as the name indicates, in alternating morphological openings and closings (resp. closings and openings) of increasing size. Let γ_n be a size distribution and φ_n an anti-size distribution, the white alternate sequential filter of size n of a function f is defined by:

$$\phi_n(f) = \varphi_n\gamma_n\varphi_{n-1}\gamma_{n-1}...\varphi_2\gamma_2\varphi_1\gamma_1(f)$$

Similarly, the black alternating sequential filter of size n of f is defined by:

$$\psi_n(f) = \gamma_n\varphi_n\gamma_{n-1}\varphi_{n-1}...\gamma_2\varphi_2\gamma_1\varphi_1(f)$$

1.2 Morphological center

1.2.1 Definition

Let (Ψ_i) be a family of increasing transformations. Put:

$$\eta = \wedge\Psi_i \text{ et } \zeta = \vee\Psi_i$$

The center c of the family (Ψ_i) is:

$$c = (I \vee \eta) \wedge \zeta$$

(I represents the identity function).
The center is not a filter (it is not idempotent), but it is convergent when iterated and its limit is a filter.

1.2.2 Examples

1. for $(\Psi_i) = (\gamma\varphi, \varphi\gamma)$:

$$c(f) = [f \vee (\gamma\varphi(f) \wedge \varphi\gamma(f))] \wedge (\gamma\varphi(f) \vee \varphi\gamma(f))$$

2. for $(\Psi_i) = (\gamma\varphi\gamma, \varphi\gamma\varphi)$:

$$c(f) = [f \vee (\gamma\varphi\gamma(f) \wedge \varphi\gamma\varphi(f))] \wedge (\gamma\varphi\gamma(f) \vee \varphi\gamma\varphi(f))$$

This center is also called an automedian filter (its limit being a filter).

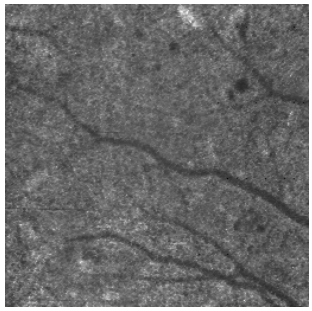
2 Contrasts, Reminder

Let η be an anti-extensive transformation and ξ an extensive transformation of a function f. A three-state contrast of primitives η and ξ is any transformation κ such that, for any f:
(i) $\kappa(f)(x)$ only depends on $\xi(f)(x)$, $\eta(f)(x)$ and $f(x)$ and on possible constants.
(ii) $\kappa(f)(x)$ can only take one of these three values (the choice depending on a decision rule).
Besides, if $\kappa(f)(x)$ cannot take the value $f(x)$, the contrast is said to be a two-state contrast.

EXERCISES

Exercise n° 1

1) You have already used alternate sequential filters (chapter 4, exercise n° 3) on the *noise* image by means of hexagonal and triangular openings and closings.
Continue this exercise:
- by increasing the number of iterations (size of the filter) with hexagonal and triangular structuring elements.
- by using different size intervals for the openings and closings (alternate filters with sizes increasing by steps larger than 1), with hexagonal and dodecagonal structuring elements.
2) Test the sequential alternating filters on greytone images by using the various openings and closings described up to now (hexagonal, dodecagonal, triangular, by sup of linear openings).
You can use the *retina3*, *burner* and *electrop* images (but also any image of your choice).



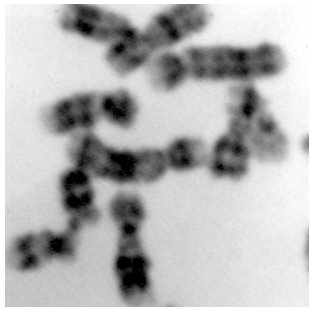
retina3

Exercise n° 2

Let κ be the contrast of primitives ξ and η and its decision rule the following:

$$\text{if } \xi(f)(x) - f(x) \leq f(x) - \eta(f)(x):$$
$$\kappa(f)(x) = \xi(f)(x)$$
$$\text{if not:}$$
$$\kappa(f)(x) = \eta(f)(x)$$

1) How many states is the contrast κ?



chromosomes

2) Program κ:
a) for η = hexagonal erosion of size n and ξ = hexagonal dilation of size n.
b) for η = morphological hexagonal opening of size n and ξ = morphological hexagonal closing of size n.
c) for η = hexagonal opening of size n and ξ = hexagonal closing of size 5*n.
Apply these transformations to the *chromosomes* image.
3) Verify that the contrast defined in 2-a is not idempotent and that the one defined in 2-b is idempotent.

Exercise n° 3

Let κ be a transformation defined by:

$$\kappa(f) = 3f - \gamma(f) - \varphi(f)$$

(γ is an opening, φ is a closing).
1) Is κ a contrast in the sense given above?
2) Program κ and apply it to the *chromosomes* image.

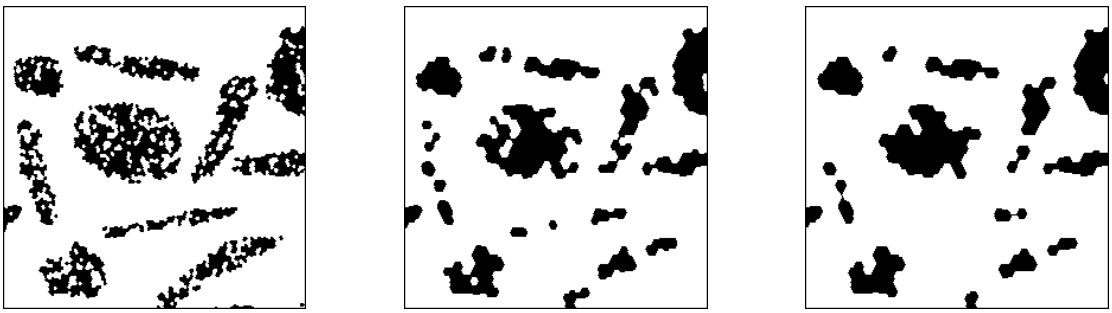
Exercise n° 4

1) Apply the automedian filter (which is not a filter according to the definition given above) defined with the doublet (φγ, γφ) to the *retina3* image.
2) program with MAMBA the automedian filter defined with the triplet (φγφ, γφγ).
3) See how many iterations of this last filter are required to reach idempotence with *burner*, *retina2* and *retina3* images.

SOLUTIONS

Exercise n° 1

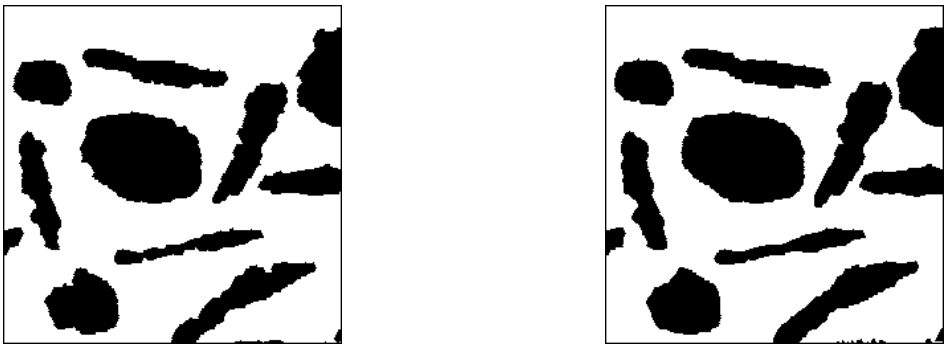
1) Use the already known operators **alternateFilter** and **fullAlternateFilter**. By increasing the size of the alternating sequential filter, details, either white or black, of increasing size gradually disappear. The main objects and shapes are preserved while the contours are smoothed.



White (start with an opening) alternate filters of noise image with hexagons of size 1 (left), 3 (middle) and 4 (right).

Black (start with a closing) alternate filters of noise image with hexagons of size 1 (left), 3 (middle) and 4 (right).

By using different sizes for openings and closings, the white pixels prevail over the black or vice-versa as the case may be.
If we use triangular structuring elements, the choice of an opening or a closing as first operator is less important: triangular filters are weaker than hexagonal ones, so the first step of the alternate filter is very similar in both cases.



Full alternate sequential filter of size 4 of the noise image with a triangular structuring element, starting with an opening (left) and a closing (right).

The operator **largeHexagonalAlternateFilter** allows to perform alternate filters with size intervals larger than 1. Entering this line:

```
>>> largeHexagonalAlternateFilter(imbin1, imbin2, 1, 6, 2, False)
```

applies the following filter to image imbin1:

$$\gamma_5 \circ \varphi_5 \circ \gamma_3 \circ \varphi_3 \circ \gamma_1 \circ \varphi_1$$

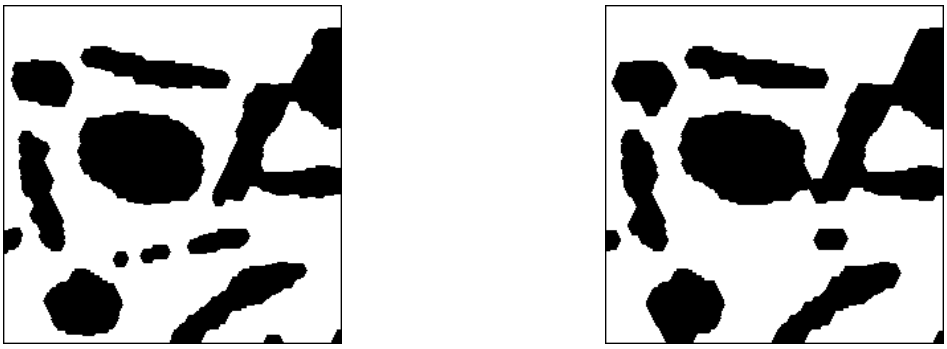
γ_i and φ_i being hexagonal openings and closings. The successive steps are 1, 3 (1 +2) and 5 (3 + 2).

In the same way, the command:

```
>>> largeHexagonalAlternateFilter(imbin1, imbin2, 1, 8, 3, False)
```

performs the following filter:

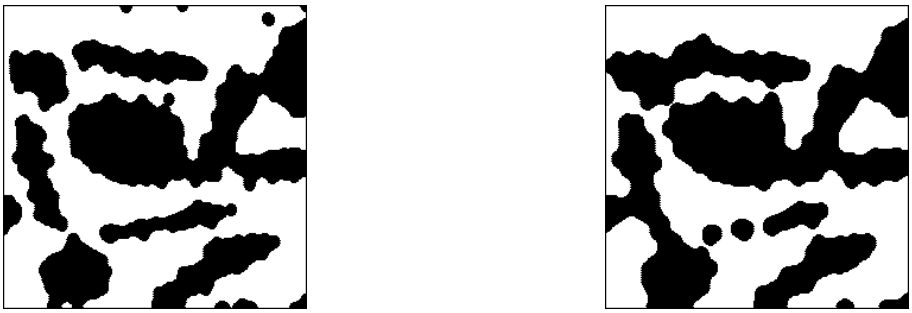
$$\gamma_7 \circ \varphi_7 \circ \gamma_4 \circ \varphi_4 \circ \gamma_1 \circ \varphi_1$$



Alternate filters starting by a closing of size 1 with a step equal to 2 (left) or to 3 (right).

You can also use dodecagonal structuring elements (but also octogonal ones). Try the **largeDodecagonalAlternateFilter**:

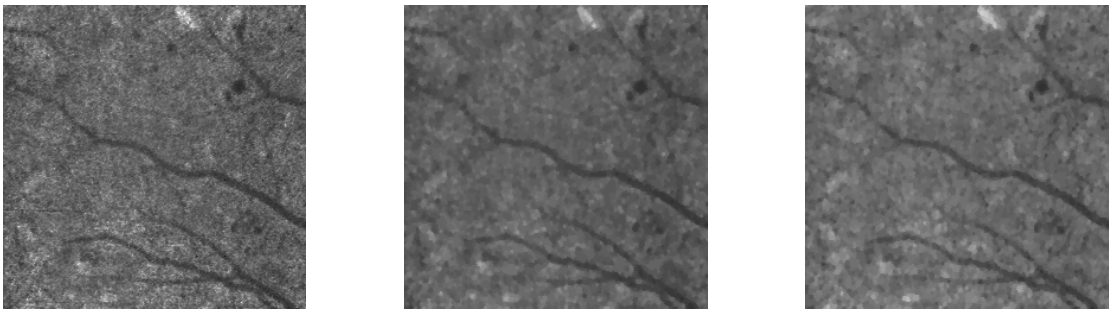
```
>>> largeDodecagonalAlternateFilter(imbin1, imbin2, 1, 6, 2, False)
>>> largeDodecagonalAlternateFilter(imbin1, imbin2, 1, 8, 3, False)
```



Alternate filters with dodecagons starting by a closing of size 1 with a step equal to 2 (left) or to 3 (right).

2) Alternate sequential filters with various types of openings and closings:
(a) opening and closing by hexagons, triangles and dodecagons
The **alternateFilter** and **fullAlternateFilter** already used with binary images can be applied on greytone images. Load the *retina3* image in im1 and enter:

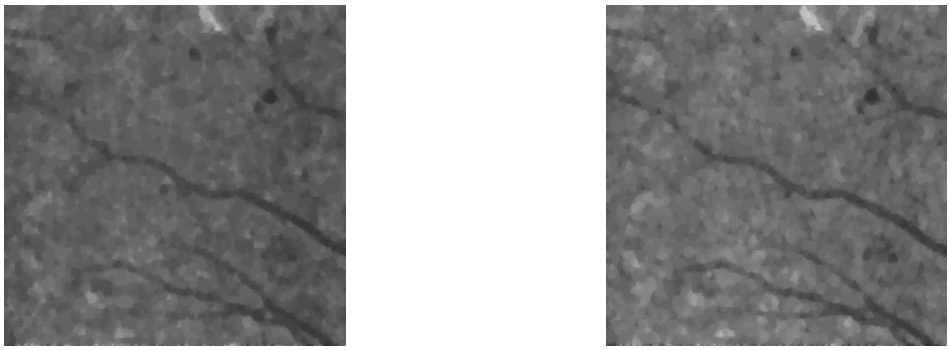
```
>>> alternateFilter(im1, im2, 1, True)
>>> alternateFilter(im1, im2, 1, False)
```



Hexagonal alternate filters of size 1 applied on retina3 image (left), starting with an opening (middle) and a closing (right).

You can try also the alternate filters with triangles:

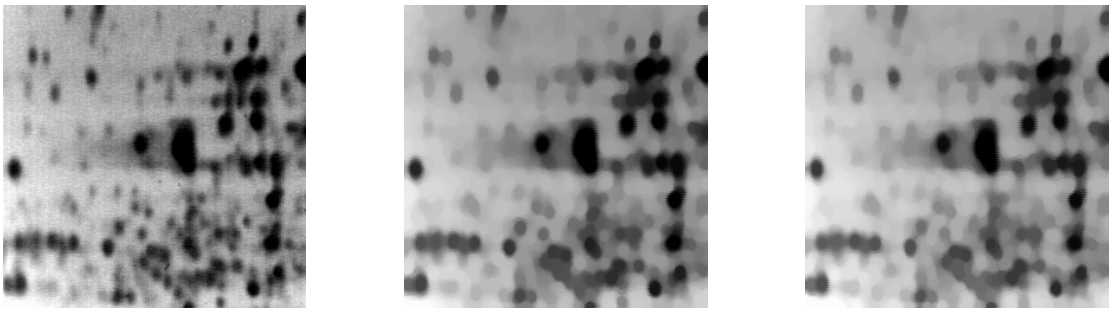
```
>>> alternateFilter(im1, im2, 3, True, se=TRIANGLE)
>>> alternateFilter(im1, im2, 3, False, se=TRIANGLE)
```



Triangular alternate filters of size 3 applied on retina3 image, starting with an opening (left) and a closing (right).

Dodecagonal filters can also be applied. Load the electrop image in im1 and enter these commands:

```
>>> largeDodecagonalAlternateFilter(im1, im2, 1, 5, 1, True)
>>> largeDodecagonalAlternateFilter(im1, im2, 1, 5, 1, False)
```



Full dodecagonal alternate filter of size 4 of the electrop image (left), starting with an opening (middle) and with a closing (right).

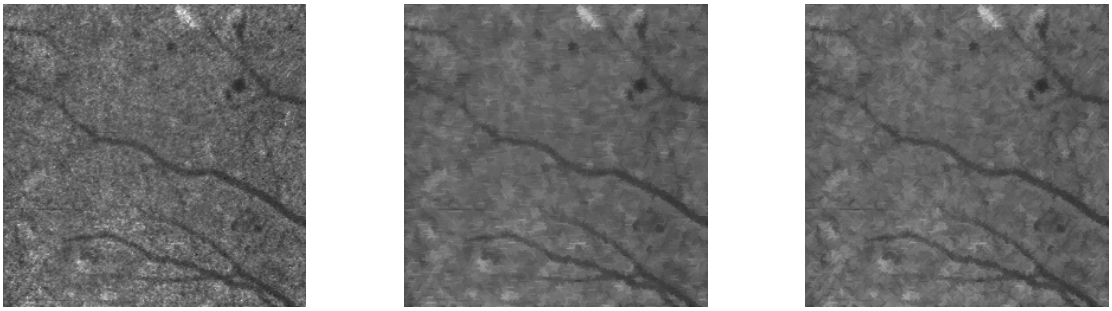
Note that all the filters using large structuring elements (**largeHexagonalAlternateFilter**, **largeDodecagonalAlternateFilter**, **largeSquareAlternateFilter**, etc.) are meant to reduce the computation time for large sizes (larger than 10). They are not optimised for small filter sizes.

(b) sup-opening of linear openings and inf-closing of linear closings

A sequential alternate filter using the sup of linear openings of the inf of linear closings already exists in MAMBA. It is named **linearAlternateFilter**. This filter can be used either with the hexagonal or square grids. Apply it on the *retina3* image loaded in im1:

```
>>> linearAlternateFilter(im1, im2, 4, True)
>>> linearAlternateFilter(im1, im2, 4, False, grid=SQUARE)
```

The first filter is applied on an hexagonal grid (three directions are used), the second one on a square grid (four directions used).



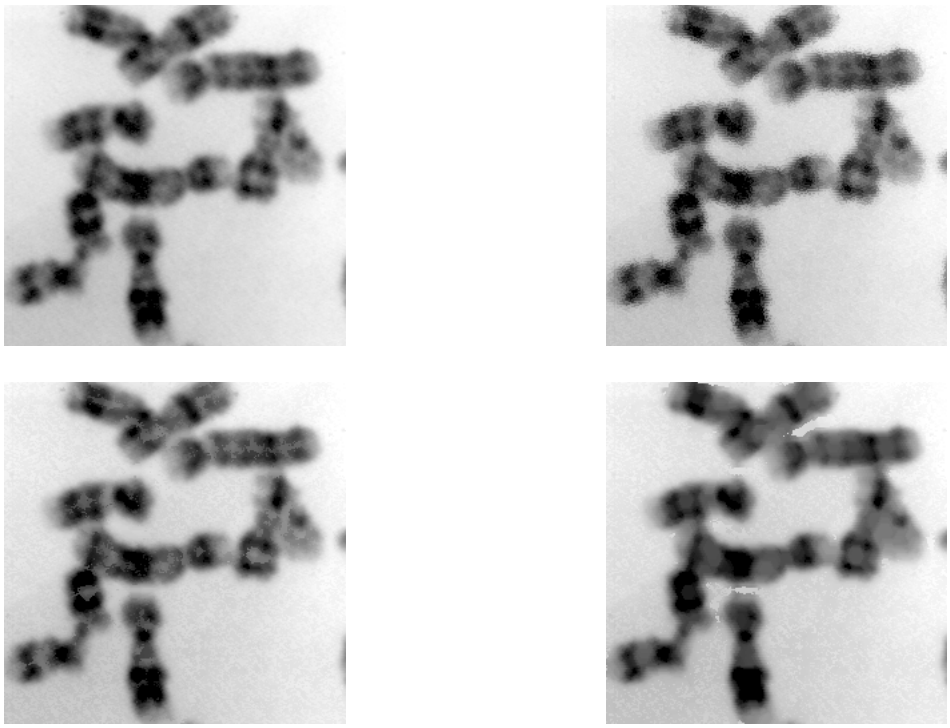
Linear filters applied on retina3 image (left): size 5 filter applied on the hexagonal grid and starting with a sup of openings (middle), filter of same size applied on the square grid and starting with an inf of closings (left).

Exercise n° 2

1) κ is a two-state contrast (obvious).
2) Programming κ
The operator **contrast** defined below uses one of the three combinations of transformations according to the value of the parameter **type**:

```
from mamba import *

def contrast(imIn, imOut, size, type):
    """
    Contrast operators applied on image 'imIn'. The result is in 'imOut'. The size
    of the operators is given by 'size'.
    'type' allows to select the type of operators:
    'type' < 1, the two contrast operators are erosion and dilation of size 'size'.
    'type' = 1, the two operators are opening and closing of size 'size'.
    'type' > 1, the first operator is an opening of size 'size', the second one
    is a closing of size 5*size.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)
    imWrk4 = imageMb(imIn)
    imMask = imageMb(imIn, 1)
    if type > 0:
        opening(imIn, imWrk1, size)
        size1 = size
        if type > 1:
            size1 = size * 5
        closing(imIn, imWrk2, size1)
    else:
        erode(imIn, imWrk1, size)
        dilate(imIn, imWrk2, size)
    sub(imIn, imWrk1, imWrk3)
    sub(imWrk2, imIn, imWrk4)
    generateSupMask(imWrk4, imWrk3, imMask, False)
    convertByMask(imMask, imOut, 0, computeMaxRange(imIn)[1])
    logic(imWrk1, imOut, imOut, "inf")
    negate(imMask, imMask)
```

```
convertByMask(imMask, imWrk4, 0, computeMaxRange(imIn)[1])
logic(imWrk2, imWrk4, imWrk4, "inf")
logic(imWrk4, imOut, imOut, "sup")
```



Contrast operators applied on the chromosomes image (upper left): dilation/erosion contrast of size 1 (upper right), opening/closing contrast of size 6 (lower left), non symmetric opening/closing contrast of size 5 (lower right).

3) In order to validate or invalidate the idempotence of the different contrasts defined above, enter the following commands:

```
>>> contrast(im1, im2, 1, 0)
>>> contrast(im2, im3, 1, 0)
>>> compare(im2, im3, im4)
(4, 0)
```

The contrast defined with erosion/dilation is not idempotent (the comparison returns the coordinates of the first different pixels).
On the contrary, enter:

```
>>> contrast(im1, im2, 1, 1)
>>> contrast(im2, im3, 1, 1)
>>> compare(im2, im3, im4)
(-1, -1)
```

In the case of a contrast by opening/closing, the operation seems to be idempotent (no modification of the result at the second iteration). This test, however, is not a proof of idempotence. So, let us prove the idempotence.

This contrast is defined by:

$$\kappa(f) = \varphi(f) \text{ if } \varphi(f) - f \leq f - \gamma(f)$$
$$\kappa(f) = \gamma(f) \text{ if not}$$

γ and φ being respectively an opening and a closing.
Let us calculate $\kappa \circ \kappa(f) = \kappa(f')$ with $f' = \kappa(f)$. f' can be equal to γ(f) or to φ(f).
We firstly consider the points of the image f where f' is equal to φ(f). In this case, we have:

$$\varphi(f') - f' = \varphi\varphi(f) - \varphi(f) = 0, \text{ as } \varphi \text{ is idempotent.}$$
$$f' - \gamma(f') = \varphi(f) - \gamma\varphi(f) \geq 0, \text{ as } \gamma \text{ is anti-extensive.}$$

Therefore:

$$\varphi(f') - f' \leq f' - \gamma(f')$$

So:

$$\kappa \circ \kappa(f) = \kappa(f') = \varphi(f') = \varphi\varphi(f) = \varphi(f) = \kappa(f)$$

We consider now the points of the image where f' is equal to γ(f). We have then:

$$\varphi(f') - f' = \varphi\gamma(f) - \gamma(f) \geq 0, \text{ as } \varphi \text{ is extensive.}$$
$$f' - \gamma(f') = \gamma(f) - \gamma\gamma(f) = 0, \text{ as } \gamma \text{ is idempotent.}$$

Then:

$$f' - \gamma(f') \leq \varphi(f') - f'$$

This large inegality can be split into two cases. Suppose that:

$$f' - \gamma(f') = \varphi(f') - f'$$

Then:

$$f' - \gamma(f') = \gamma(f) - \gamma\gamma(f) = 0 = \varphi(f') - f', \text{ hence } \varphi(f') = f'$$

By definition, we have:

$$\kappa(f') = \varphi(f')$$

So:

$$\kappa(f') = \kappa \circ \kappa(f) = \varphi(f') = f' = \gamma(f) = \kappa(f)$$

Suppose now that:

$$f' - \gamma(f') < \varphi(f') - f' \text{(strict inequality)}$$

Then:

$$\kappa(f') = \gamma(f') = \gamma\gamma(f) = \gamma(f) = \kappa(f)$$

We have finally:

$$\kappa \circ \kappa(f) = \kappa(f)$$

at any point of the image. κ is idempotent.

Exercise n° 3

1) κ may be written in a simpler way by remarking that:

$$\kappa(f) = 3f - \gamma(f) - \varphi(f) = f + (f - \gamma(f)) - (\varphi(f) - f)$$

The transform $f - \gamma(f)$ is called white top-hat transform. The transform $\varphi(f) - f$ is called black top-hat transform. These two transforms belong to a class of operators named residues (see chapter 7). The first transform allows to extract white and narrow features from the image whilst the second one extracts black and narrow features. These two transforms exist in the MAMBA library. They are named **whiteTopHat** and **blackTopHat**. As explained in the graphic below, the κ operator acts as a contrast enhancer. The white and narrow features are lightened and the black and narrow features are darkened.
If, for all f, we define η(f) and ξ(f) as:

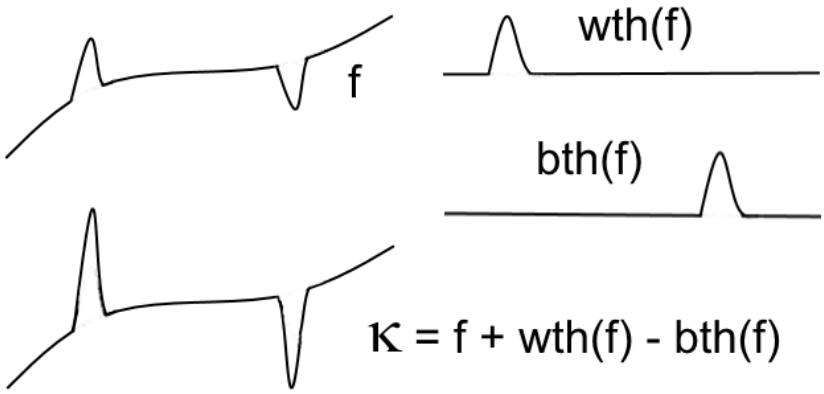
$$\eta(f) = \min(f, f + (f - \gamma(f)) - (\varphi(f) - f))$$
$$\xi(f) = \max(f, f + (f - \gamma(f)) - (\varphi(f) - f))$$

We have, $\forall f$:

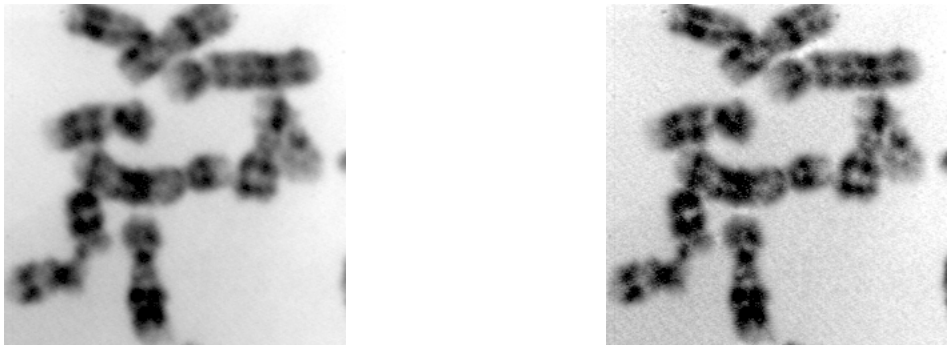
$$\eta(f) \leq f \text{ and } \xi(f) \geq f$$

Then η is anti-extensive and ξ extensive.
Therefore κ can be written as a contrast defined by:

$$\kappa(f)(x) = \eta(f)(x) \text{ if } \eta(f)(x) < f(x)$$
$$\kappa(f)(x) = \xi(f)(x) \text{ otherwise.}$$



Initial image (upper left), extraction of white features with the white top-hat (upper right), extraction of black features with a black top-hat (lower right), contrast enhancement obtained with the contrast operator (lower left).



Contrast by top-hat: left, original chromosomes image, (right) transform of size 4.

2) The corresponding operator, named **contrastByTopHat** is not included in the library. It can be defined as following (with an hexagonal structuring element):

```
from mamba import *

def contrastByTopHat(imIn, imOut, size):
    """
    Contrast by top-hat of size 'size' of 'imIn', result in 'imOut'.
    The final image can be identical to the initial one.
    For greyscale images, the arithmetic operations are truncated.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    copy(imIn, imWrk1)
    whiteTopHat(imIn, imWrk2, size)
    add(imIn, imWrk2, imOut)
```

```
blackTopHat(imWrk1, imWrk2, size)
sub(imOut, imWrk2, imOut)
```

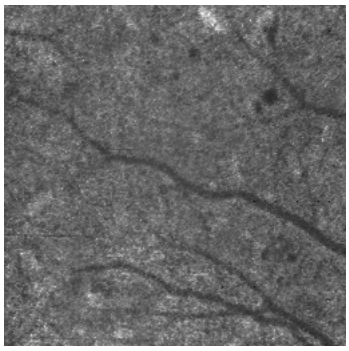
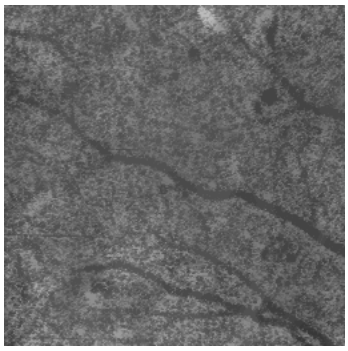
Exercise n° 4

1) The automedian filter defined by:

$$c(f) = [f \vee (\gamma\varphi(f) \wedge \varphi\gamma(f))] \wedge (\gamma\varphi(f) \vee \varphi\gamma(f))$$

is already included in the MAMBA library. It is named **autoMedian**. Try it with the following command (hexagonal structuring element), applied on the *retina3* image:

```
>>> autoMedian(im1, im2, 5)
```

Automedian with an hexagon of size 5 of the retina3 image (left).

2) The automedian filter defined by:

$$(f) = [f \vee (\gamma\varphi\gamma(f) \wedge \varphi\gamma\varphi(f))] \wedge (\gamma\varphi\gamma(f) \vee \varphi\gamma\varphi(f))$$

can be defined with the following MAMBA procedure, named **autoMedian2**:

```
from mamba import *

def autoMedian2(imIn, imOut, n):
    """
    Morphological automedian filter performed with an alternance
    closing/opening/closing and opening/closing/opening.
    """

    oco_im = imageMb(imIn)
    coc_im = imageMb(imIn)
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    alternateFilter(imIn, oco_im, n, True)
    opening(oco_im, oco_im, n)
    alternateFilter(imIn, coc_im, n, False)
    closing(coc_im, coc_im, n)
    copy(coc_im, imWrk1)
    logic(oco_im, imWrk1, imWrk1, "sup")
    copy(coc_im, imWrk2)
    logic(oco_im, imWrk2, imWrk2, "inf")
    copy(imIn, imOut)
    logic(imOut, imWrk2, imOut, "sup")
    logic(imOut, imWrk1, imOut, "inf")
```

3) In order to compare images, let us define an image comparator named **compareImages**. This operator returns the number of modified pixels in the two images:

```
def compareImages(imIn1, imIn2):
  """
  Compares the two images 'imIn1' and 'imIn2' and returns the number of
  modified pixels between them.
  """

  imWrk1 = imageMb(imIn1, 1)
  imWrk2 = imageMb(imIn2, 1)
  generateSupMask(imIn1, imIn2, imWrk1, True)
  generateSupMask(imIn2, imIn1, imWrk2, True)
  logic(imWrk1, imWrk2, imWrk1, "sup")
  pixdiff = computeVolume(imWrk1)
  return pixdiff
```

Then, with *burner* loaded in the im1 image, enter:

```
>>> autoMedian2(im1, im2, 1)
>>> compareImages(im1, im2)
23892L
>>> autoMedian2(im2, im1, 1)
>>> compareImages(im1, im2)
6441L
```

Each step returns the number of modified pixels. Iterate the process until you reach idempotence. Do the same with the *retina2* and *retina3* images. The following table indicates the number of modified pixels at each iteration for each image.

	burner	*retina2*	*retina3*
1	23 892	24 051	21 159
2	6 441	5 928	6 715
3	2 034	1 621	2 503
4	564	517	1 081
5	237	166	422
6	54	50	218
7	14	31	102
8	8	12	30
9	0	4	6
10	0	1	1
11	0	3	0
12	0	0	0

REFERENCES

[1] J.Serra: Courses on Mathematical Morphology, Morphological Filtering (**http://cmm.ensmp.fr/~serra/cours/pdf/en/ch4en.pdf**)

This course is devoted to the morphological filtering: concepts, main properties, derived notions, etc.

Chapter 6

GEODESY

1 Reminder, binary case

Given a set X, the geodesic distance between two points x and y of X is defined as the length of the shortest path L(x,y) included in X and joining these two points. We can now define balls of size λ in this metrics, and then the erosion and dilation of a set Y included in X by a geodesic ball B .
When one works on digitized sets, it can be shown that the elementary geodesic dilation is defined by:

$$D_X(Y) = (Y \oplus H) \cap X$$

Similarly, the elementary geodesic erosion is defined by:

$$E_X(Y) = X \cap [(Y \cup X^c) \ominus H]$$

H being an elementary digital ball (hexagon or square).

2 Greytone case

In the greytone case, the geodesic space under consideration may be either a set X or a function g (the latter case is the immediate generalization of the binary notion applied to sub-graphs). On digitized sets, the following definitions apply:
The geodesic dilation of f into the set X by an elementary hexagon centered in O is defined at any point x by:

$$D_X(f)(x) = \sup_{\substack{b \in H \\ x + \overrightarrow{Ob} \in X}} \left(f\left(x + \overrightarrow{Ob} \right) \right) = \sup\{f(x); x \in H_x \cap X\}$$

Likewise the erosion:

$$E_X(f)(x) = \inf_{\substack{b \in H \\ x + \overrightarrow{Ob} \in X}} \left(f\left(x + \overrightarrow{Ob} \right) \right) = \inf\{f(x); x \in H_x \cap X\}$$

The geodesic dilation of f under the function g by an elementary hexagon is defined by:

$$D_g(f) = \inf(f \oplus H, g)$$

Likewise the erosion of f over g:

$$E_g(f) = \sup(f \ominus H, g)$$

NB : Note that the duality is different from the one defined in the binary case. The duality used in the binary caseis the duality based on the complementation inside the geodesic space X. In the greytone case, the duality is defined with the image inversion (or negation): we simply replace f by m-f (where m=255 for 8-bit images and 2^{32} -1 for 32-bit images).

3 Geodesic reconstructions

Geodesic reconstructions are efficient tools for filtering, segmentation and so on. The geodesic reconstruction is defined as the limit of iterated geodesic dilations:

$$R_X(Y) = \lim_{n\to\infty} D_X(Y)^n$$
$$R_g(f) = \lim_{n\to\infty} D_g(f)^n$$

The dual reconstruction is defined as the infinite iteration of geodesic erosions:

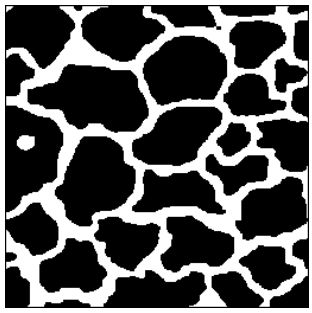
$$R^*_X(Y) = \lim_{n\to\infty} E_X(Y)^n \quad ; \quad R^*_g(f) = \lim_{n\to\infty} E_g(f)$$

EXERCISES

Exercise n° 1: Geodesic reconstructions

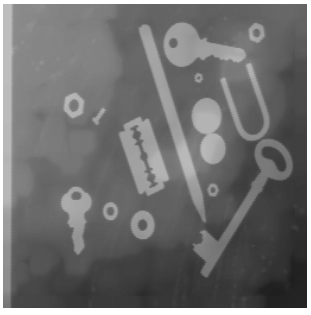
The **build, dualBuild, hierarBuild, hierarDualBuild** operators, which implement reconstructions of a function f with a function g as marker are already efficiently installed (with different algorithms) in MAMBA.

1) Let X be a set composed of several connected components $\{X_i\}$. A set Y included in X marks one or several connected components of X. Reconstruct the connected components of X marked by Y (these connected components consist of points x of X which are at a finite geodesic distance d(x,Y) from Y). Use the binary *alumine* image as mask and a sufficiently large erosion as marker.

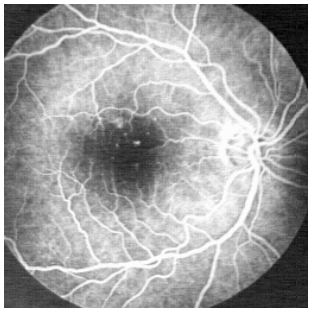


alumine

2) Test this reconstruction on the *tools* image by using as a marker an image entirely set to 0 except at a point (selected preferably at the location of an object) where the value is set to 255.



tools



retina1

3) Do the same operation on the *retina1* image, by placing the point on the blood lattice. Compare the speed of the different reconstructions of the blood vessels.

Exercise n° 2: Opening by erosion - reconstruction

1) Prove that the geodesic reconstruction of f (resp. X) by its erosion by the size n structuring element B is an opening (the center of the structuring element B must be a point of B).
2) Use the corresponding MAMBA operator, named **buildOpen**, as well as the dual closing, **buildClose** and compare them, on the *retina1*, *retina2* and *cat* images, to the classical opening and closing.

Exercise n° 3: Revisiting the individual analysis of particles

The individual analysis of particles has already been introduced in previous exercises (see exercise n°4, chapter 3 and exercise n°4, chapter 4). This technique consists in extracting each connected component of a set in order to perform some measure on it. The MAMBA **label** operator gives an efficient and fast way to achieve this extraction. It is also possible to to label each particle with the measured value as it is illustrated with the **areaLabelling** operator.
1) Verify that the transformation $\Psi(X)$ defined in this way:

$$X = \bigcup_{i=1}^{n} X_i, \text{ } X_i \text{ connected component of X}$$

$$\Psi_\lambda(X) = \cup X_j \text{ such that } mes(X_j) > \lambda$$

is a size distribution.
2) There exists another operator in the MAMBA library, named **measureLabelling**, which allows to label every connected component of an image with a value equal to the number of pixels of the second image which are contained in the connected component.
Design a procedure to label every particle of a binary image with its vertical diameter. Apply it to the *eutectic* image.
3) Consider a binary image X and a greyscale (8-bit) one. Can you design an algorithm for labelling any connected component of X with the maximum value taken by f inside his connected component, that is obtaining a function g such that:

$$g(x) = \max_{y \in X_i} f(y) \text{ for every } x \in X_i$$

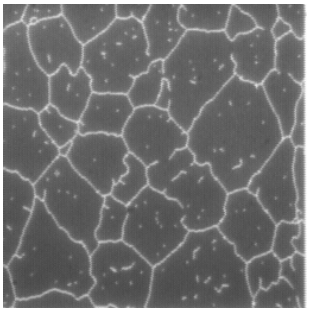
$$g(x) = 0 \text{ elsewhere}$$

Do the same with the minimum value. Try these transforms with the binary *tesselation* image and the greyscale *road1* image.

Exercise n° 4: Holes filling, objects cutting edges

1) Apply the geodesic reconstruction algorithm for suppressing the particles of a set X touching the edges of the field. What can be in particular the marking set Y? Application to the *grains2* image.
2) How can you fill the holes in the particles? Design an algorithm and test it on the *holes* and *gruyere* images.
3) On a greytone image, a hole may correspond to what is called a basin if the image is considered as a topographic surface. We can then imagine to fill up these holes, as would do rain water, the exceeding water spilling outside the limits of the image. Similarly to the binary case, design an algorithm for filling the holes on a greytone image and apply it to the *circuit* and *tools* images.

4) Can you find a way to eliminate the inclusions in the *alumine* image?



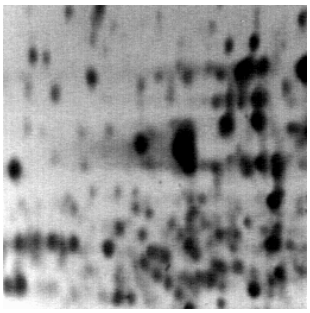
alumine

Exercise n° 5: Regional maxima and minima

Let p_0 ,...,p_n be the points of the image E. V(p_i) is the value of the image in p_i. p_0 p_1 ...p_n is called a path if, $\forall$i, p_{i+1} and p_i are neighbors. p_0 ...p_n is said to be strictly ascending if, $\forall$i, V(p_{i+1}) ≥ V(p_i) and V(p_0) < V(p_n). p_0 ...p_n is said to be strictly descending if, V(p_{i+1}) ≤ V(p_i) and V(p_0) > V(p_n).
A connected set X of the image E is a regional maximum if there exists no strictly ascending path coming from X:

$$\forall x \in X, \forall (p_i)_{i \in \langle 1 \ldots N \rangle} \in E, xp_0 \ldots p_N \text{ is not strictly ascending.}$$

1) Find an algorithm allowing to determine the regional maxima using the successive sections of the image and the binary reconstruction.
2) In practice, the level by level approach is not used. The following one is faster: 1 is subtracted from the image and the resulting image is reconstructed from the initial image. The regional maxima are located where the resulting image differs from the initial image. The dual algorithm generates the regional minima. These transformations exist in MAMBA and are named **maxima** and **minima**. Apply the **minima** operator to the *electrop* image. What do you observe? How can you explain this? Can you propose any enhancement?
3) The algorithmic method above allows to generalize the notions of regional minima and maxima. A point of the graph of a function belongs to a h-maximum if the height of any non descending path starting from this point is strictly less than h. The h-maxima are obtained by subtracting h from the initial image. The h-maxima are located where the resulting image differs from the initial image. The dual algorithm generates the h-minima. These h-maxima and h-minima are obtained with the same MAMBA operators by just changing the value of the height or depth parameters. Apply the **minima** operator with increasing depths to the filtered *electrop* image. Can you interpret the results, in particular when depths increase?

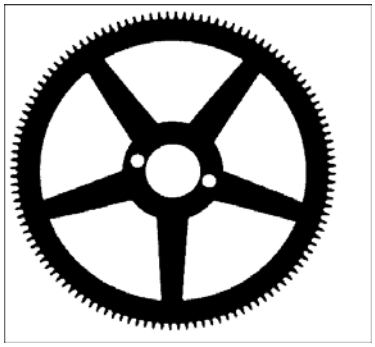


electrop

4) Two other operators are also available in the MAMBA library: **highMaxima** (**deepMinima**) and **maxDynamics** (**minDynamics**).
Try the **deepMinima** operator on the filtered electrop image (with the same parameters). What difference can you notice compared to the previous results obtained with the extended minima operator?
Try the **minDynamics** operator and interpret the results.

Exercise n° 6

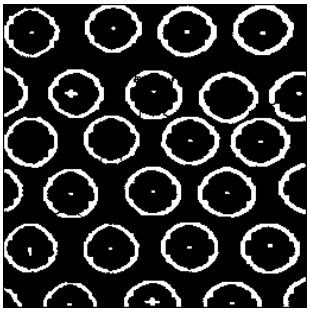
This exercise shows how a very simple transformation (opening here) combined with geodesic reconstructions can solve a problem of detection and counting of the teeth of a notched wheel when it is associated to a preliminary selection of the zone where these teeth should be.
Use image *wheel* to extract and count its teeth.



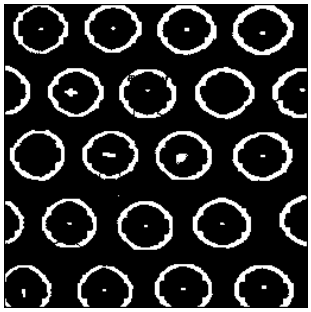
wheel

Exercise n° 7: Analysis of the distribution of boron fibers

This exercise illustrates the judicious use of measures for solving a problem of quality control.



boron1



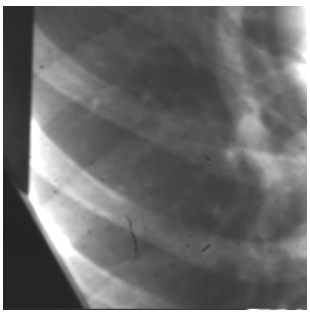
boron2

The *boron1* and *boron2* images represent boron fibers in a composite material. These fibers reinforce the mechanical resistance of the material. The resistance increases in proportion of the regularity of the fibers layout. Therefore, during industrial production, they are as far as possible placed according to a regular hexagonal network. However, irregularities occur.
The problem consists in quantifying these irregularities by means of an appropriate measurement. The images provided in the exercise are already thresholded (binary images).
1) Simplify the images in filling up the fibers (this is not easy at all!). The fibers that cut the edge of the image field are also to be filled.

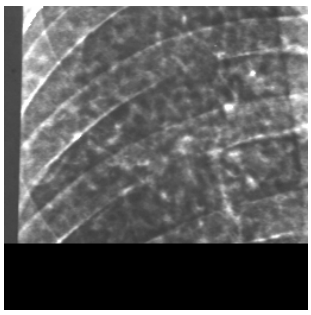
2) Perform hexagonal closings of increasing size and observe the result. Propose an elementary measure to quantify the connections between the fibers.
3) Compute the theoretical value of the specific connectivity number according to the size of the closing, in the case of a regular distribution. Derive a quantifier of the irregularity of the structure. Program it and apply it to the two example images.

Exercise n° 8: Size distribution of a greytone image

When a size distribution transformation is applied to a binary image, the computed size distribution curve (see chapter 4, exercise 5) may be considered as a texture index characterizing the size of the particles (case of the opening by a disk), the size of the pores separating the particles (closing by a disk) or else the main directions of the image (use of linear structuring elements).
A particularity of the morphological notion of size distribution is that it can be applied to non distinctive structures. The value which is then taken into account is, for example, the subtracted or added area for each operation of increasing size. The size distribution is then said to be “in measure” and is opposed to the size distribution “in number” which only applies to isolated particles.
The greytone size distribution is also an "in measure" size distribution but here, the measure more often bears on volumes than on areas. The size distribution curve will also be considered as a texture index but this time it characterizes a relief: shape, size, height of the volume structures contained in the sub-graph of the image.
We will try to use a greytone size distribution in order to characterize the presence of nodules (small white domes) on lung radiographs.
Let us recall the properties to be satisfied by a size distribution transformation.
Let X be the initial set (the sub-graph of an image in our case). Let Ψ be the sieving transformation applied to this set. We denote $\Psi_l(X)$ the part of X which has been retained by the sieve of size 1. If Ψ fulfills the three following rules, it may be considered as a transformation with good size distribution qualities:
1. The operation must be anti-extensive, in other words the part retained by the sieve must be a sub-set of X.
2. The operation must be increasing.
3. The operation must satisfy the following size criterion:
$\psi_{l_1}(\psi_{l_2}(X)) = \psi_{l_2}(\psi_{l_1}(X)) = \psi_{\sup(l_1,l_2)}(X)\ \forall l_1, l_2 > 0$
We know that the opening satisfies the three properties.



lung1



lung2

The images to be studied are *lung1* and *lung2*. *lung1* represents a sound lung, *lung2* a lung with a nodular texture. The nodules are small white dots distributed on the whole image.

1) Which kind of greytone transformation satisfying size distribution rules would allow to reveal the difference in texture between the two lung images?
- Program a size distribution function from this transformation.
- Does it effectively differentiate the two images?
- What do we learn from the position of the maximum of the size distribution curve?
- And from the range of this maximum?

2) What can be the use of a greytone opening by reconstruction? Program the size distribution based on openings by reconstruction and try to explain the differences between the two types of size distribution curves.

3) Indicate the third type of opening that can be used as a basic size distribution function to analyze these images? Why is the range of this curve much lower than the preceding?

4) Is it possible to apply a size distribution in number to a greytone image?

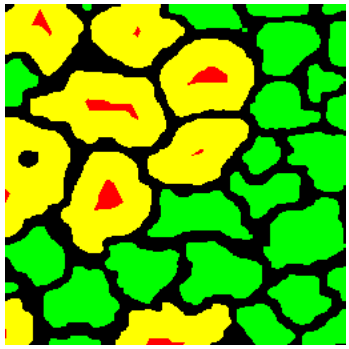
SOLUTIONS

Exercise n° 1: Geodesic reconstructions

1) Load the *alumine* image in imbin1 and type:

```
>>> erode(imbin1, imbin2, 20)
>>> build(imbin1, imbin2)
```

Note that, with binary images, the **build** operator must be used (the hierarBuild operator does not work with binary images). The marker image is replaced by the reconstructed image (here in imbin2). Note also that the marker is automatically intersected with the mask before the operation to force it to be included in the mask.



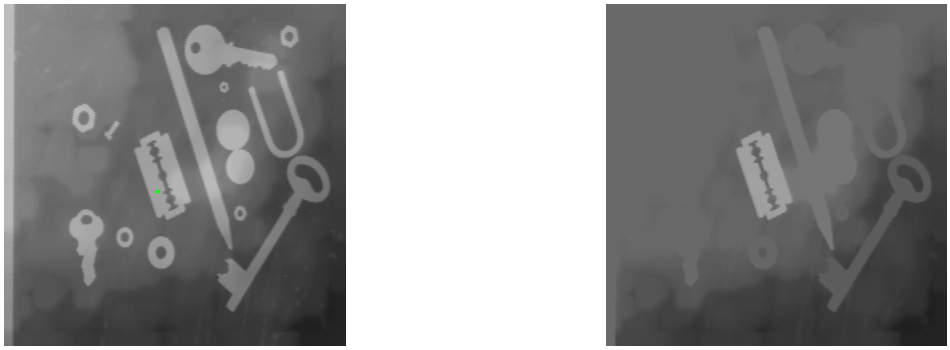
Geodesic reconstruction (in yellow) of the alumine image by a marker set (in red) obtained by a size 20 erosion of the initial image.

2) Greytone reconstruction

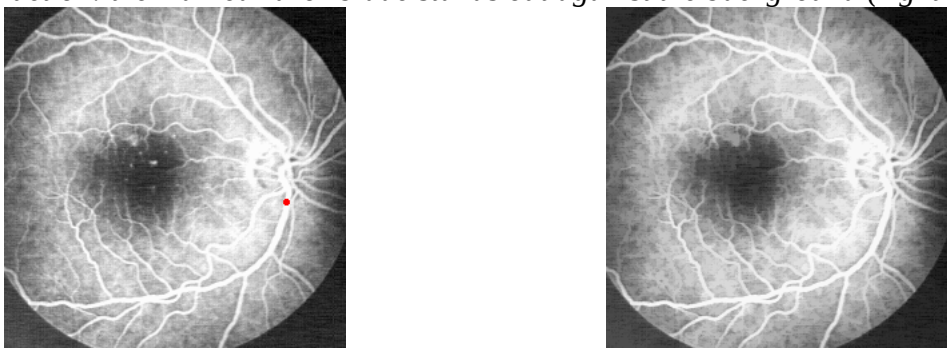
Load the *tools* image in im1, and enter the following commands (the first line insures that im2 is empty):

```
>>> im2.reset()
>>> im2.setPixel(255, (115, 140))
>>> hierarBuild(im1, im2)
```

The result is comparable to the binary reconstruction, only the marked object is reconstructed. Remark also that, if the marker image is higher than the mask image (it is the case here), it is automatically intersected with the mask image by the **hierarBuild** (or **build**) transform. In the cas of a dual reconstruction, an union (or sup) is performed.



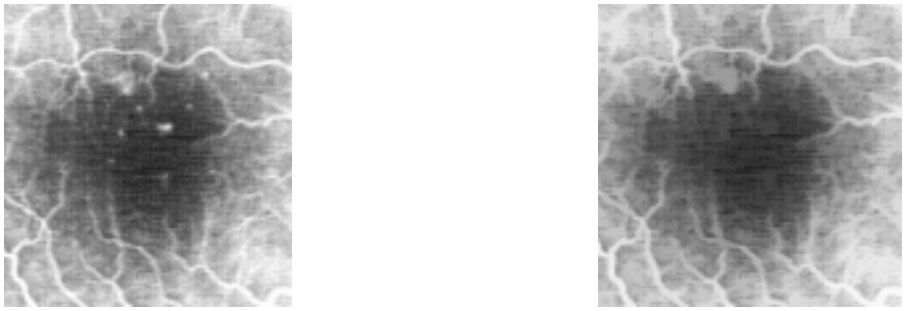
Initial tools image with the marker point in green (left image). Result of the geodesic reconstruction: the marked razor blade stands out against the background (right image).



The retina1 image and the marker (in red) in the vessels (left), result of the geodesic reconstruction (right).

2) The same effect can be observed on the *retina1* image (loaded in im1):

```
>>> im2.reset()
>>> im2.setPixel(255, (210, 145))
>>> hierarBuild(im1, im2)
```



View of the macula in the retina1 image (left), result of the geodesic reconstruction (right), the aneurisms are removed.

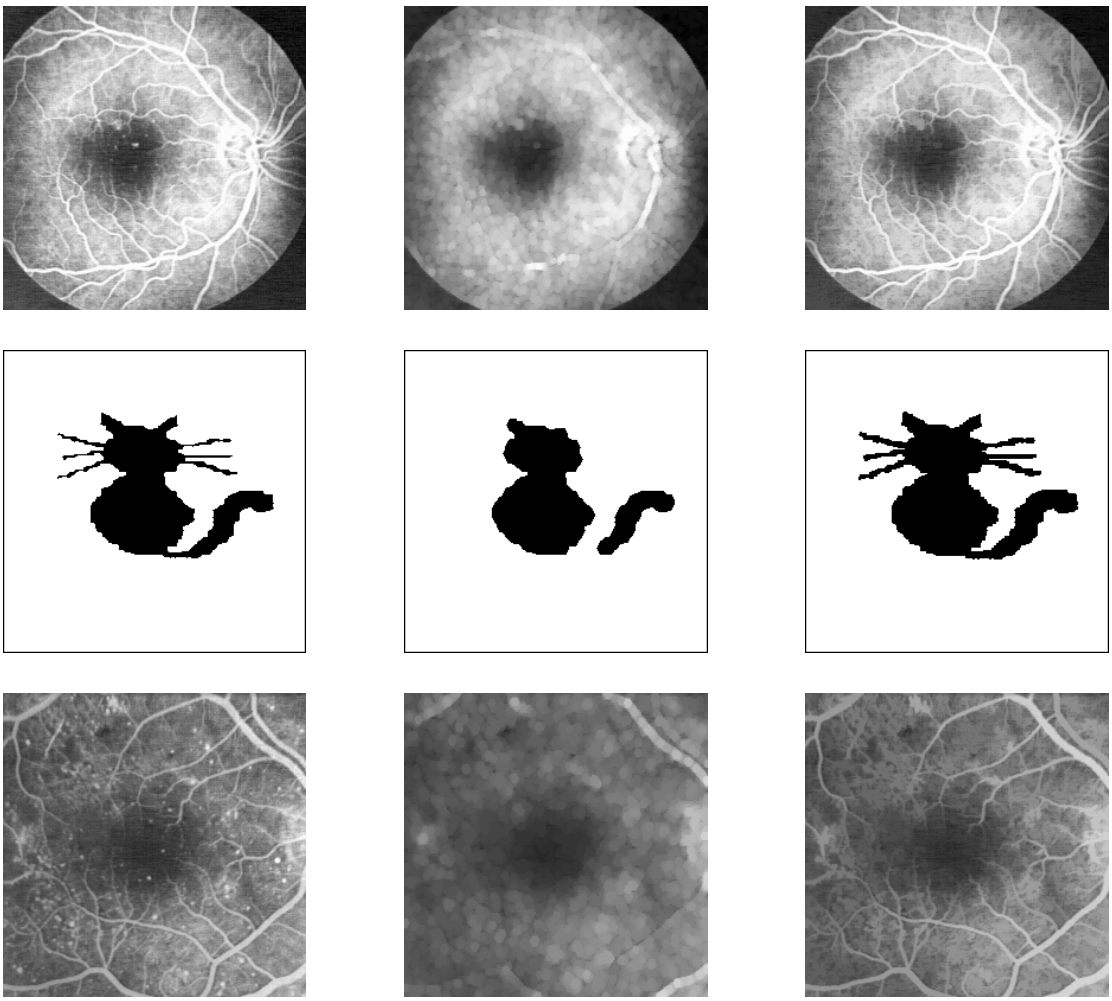
We notice that the white spots situated at the center of the image (a region called macula) are not reconstructed. Indeed, they are not connected to the vascular network. Remember that “x and y are connected” means: there exists a downward path joining a point x of the marker to

a point y of the mask. The non reconstructed white dots correspond to aneurisms in the retina, a quite serious pathology of the eye fundus.

Exercise n° 2: Opening by erosion - reconstruction

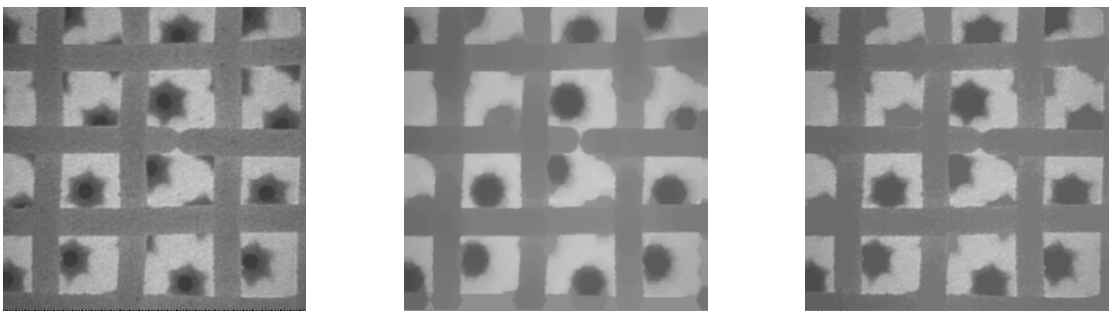
1) Let us prove that the opening by erosion-reconstruction is an opening.
Denote $\Psi_0(X) = X \ominus nH,\ \Psi_m(X) = (\Psi_{m-1}(X) \oplus H) \cap X$.
Ψ_m corresponds to the mth step of the geodesic dilation performed from the initial eroded set.
Ψ designates the limit of Ψ_m when $m \to \infty$, that is the opening by erosion-reconstruction.
<u>Ψ is increasing</u>:
Let $Y \subset X$.
$Y \ominus nH \subset X \ominus nH$ since the erosion is increasing.
$(Y \ominus nH) \oplus H \subset (X \ominus nH) \oplus H$ since the dilation is increasing.
$((Y \ominus nH) \oplus H) \cap Y \subset ((Y \ominus nH) \oplus H) \cap X \subset ((X \ominus nH) \oplus H) \cap X$ since $Y \subset X$.
Then $\Psi_1(Y) \subset \Psi_1(X)$. By iteration, $\Psi_m(Y) \subset \Psi_m(X)$. If Ψ(X) designates the limit of this iteration when $m \to \infty$, we have $\Psi(Y) \subset \Psi(X)$.
<u>Ψ is anti-extensive</u>:
Obvious (each step of the geodesic dilation is included in X).
<u>Ψ is idempotent</u>:
Let us prove that $\Psi(X) \ominus nH = X \ominus nH$.
We have $\Psi(X) \ominus nH \subset X \ominus nH$, since Ψ is anti-extensive and the erosion is increasing
Let us show that $X \ominus nH \subset \Psi(X) \ominus nH$.
For this, let us prove first that $\forall m' < m, \Psi_{m'}(X) \subset \Psi_m(X) \subset \Psi(X)$. As the origin of the structuring element H is inside H, the dilation is extensive. Then:
$\forall m' < m, \Psi_{m'}(X) \subset \Psi_{m'+1}(X) \subset \Psi_m(X)$. When $m \to \infty$, we have$\Psi_m(X) \subset \Psi(X)$.
Let us verify now that $\forall m \le n, \Psi_m(X) = (X \ominus nH) \oplus mH$.
Let us consider the case where $m \le n$. We have:
$(X \ominus nH) \oplus mH \subset (X \ominus nH) \oplus nH \subset X$
By successive iterations, we can write:
$\Psi_0(X) = X \ominus nH$
$\Psi_1(X) = (\Psi_0(X) \oplus H) \cap X = ((X \ominus nH) \oplus H) \cap X = ((X \ominus nH) \oplus H)$
$\Psi_2(X) = (\Psi_1(X) \oplus H) \cap X = ((X \ominus nH) \oplus 2H) \cap X = ((X \ominus nH) \oplus 2H)$
$\vdots$
$\Psi_m(X) = (\Psi_{m-1}(X) \oplus H) \cap X = ((X \ominus nH) \oplus mH) \cap X = ((X \ominus nH) \oplus mH)$
Finally, let us prove that $((X \ominus nH) \oplus nH) \ominus nH = X \ominus nH$.
$((X \ominus nH) \oplus nH) \ominus nH \subset X \ominus nH$ because the opening is anti-extensive.
.$((X \ominus nH) \oplus nH) \ominus nH \supset X \ominus nH$ because the closing is extensive (an erosion is always a closing).
When m = n, we have then:
$\Psi_n(X) \ominus nH = ((X \ominus nH) \oplus nH) \ominus nH = X \ominus nH$
Now consider the case where m > n. We have $\Psi n(X) \subset \Psi_m(X)$, then:
$\Psi_n(X) \ominus nH \subset \Psi_m(X) \ominus nH$ *and* $\Psi_n(X) \ominus nH = X \ominus nH$.
Then, $\forall m > n, X \ominus nH \subset \Psi_m(X) \ominus nH$.
When $m \to \infty$, we have $X \ominus nH \subset \Psi(X) \ominus nH$.
Therefore $X \ominus nH = \Psi(X) \ominus nH$.
Finally, performing $\Psi(\Psi(X))$ starts with an erosion of $\Psi(X)$ by nH followed by an iteration of geodesic dilations. But, as $\Psi(X) \ominus nH = X \ominus H$, we have $\Psi(\Psi(X)) = \Psi(X)$, Q.E.D.

2) Test **opening** and **buildOpen**, the two operators available in the MAMBA library. As you may notice, the classical **opening** is always coarser (it removes more features) than the **openBuild** operator.



From left to right: original image, hexagonal opening and erosion-reconstruction opening of same size. Top image (retina1), the size is equal to 2. Middle image (cat), the size is equal to 5. Bottom image (retina2): the size is equal to 3.

You can also compare the **closing** and **buildClose** operators, for example on the *burner* image.



burner image (left), hexagonal closing of size 8 (middle), dilation-dual reconstruction closing of same size (right).

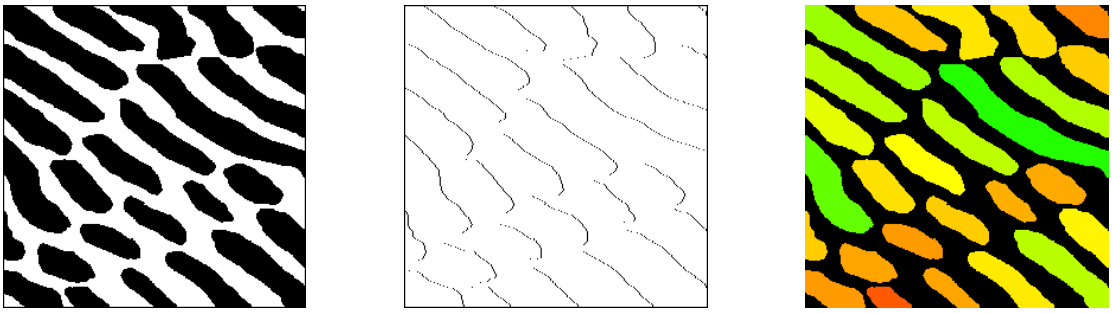
Exercise n° 3: Revisiting the individual analysis of particles

1) $\Psi_\lambda(X)$ is a size distribution (granulometry):
- It is idempotent: indeed, keep of a set only the connected components whose surface is higher than λ, then repeat the operation with the new set, the result remains unchanged.
- When $\lambda \geq \mu, \Psi_\lambda(X) \subset \Psi_\mu(X)$ (obvious).
- Finally: $\Psi_\lambda[\Psi_\mu(X)] = \Psi\mu[\Psi\lambda(X)] = \Psi_{\sup(\lambda,\mu)}(X)$.

2) The vertical diameter is equal (up to a scale factor which will be supposed set to 1) to the number of intercepts of the connected component in the horizontal direction (configuration 01). These intercepts can be obtained by the set difference between the initial image and its linear horizontal elementary erosion. The intercept set can then be used to label the initial particles by means of the **measureLabelling** operator:

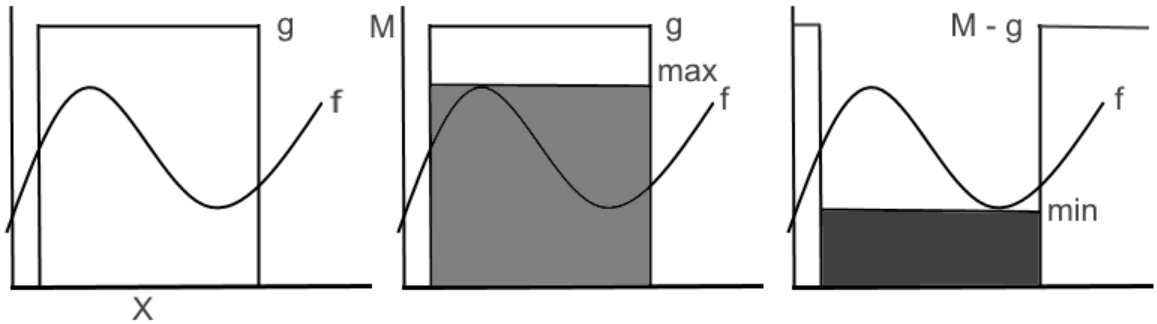
```
>>> linearErode(imbin1, imbin2, 2, edge=EMPTY)
>>> diff(imbin1, imbin2, imbin2)
>>> measureLabelling(imbin1, imbin2, im32_1)
```

Note that the depth of the labelled image is 32-bit.



eutectic image (left), its horizontal intercepts (middle) and labelling of each connected component with its vertical diameter (right).

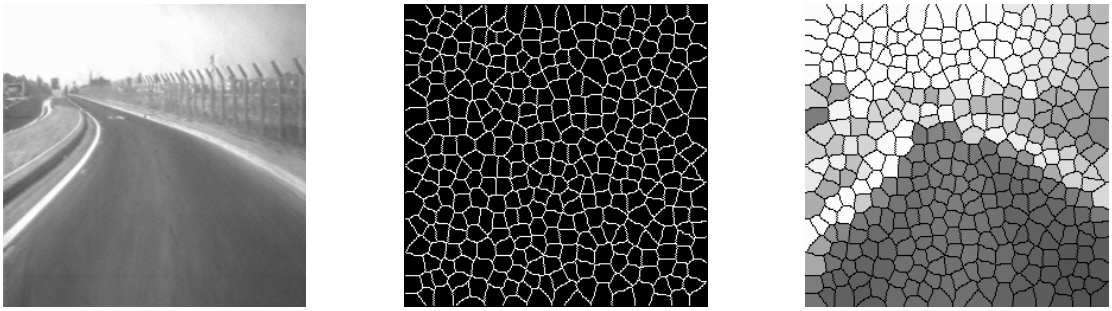
3) These labellings are performed rapidly by using the geodesic reconstructions, as explained in the figure below:



Labellings a set X by the maximum value (middle) or the minimum value (right) taken by the function f inside X. These labellings are obtained by a reconstruction of f under the indicator function g of X and by a dual reconstruction over the inverted function M - g.

Load the *tesselation* image in imbin1 and the *road1* image in im1 and type the following instructions:

```
>>> convert(imbin1, im2)
>>> copy(im1, im3)
>>> hierarBuild(im2, im3)
```

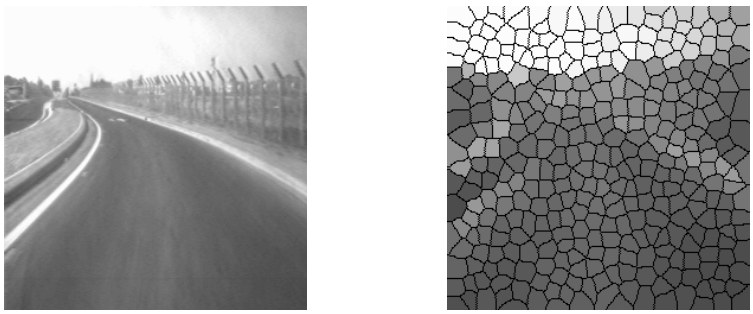


The maximum value of the road1 image (left) in each cell of the tesselation image (middle) is used to label it (right image).

Labelling the *tesselation* cells with the minimum of *road1* in each cell is performed by the following sequence of instructions:

```
>>> convert(imbin1, im2)
>>> negate(im2, im2)
>>> copy(im1, im3)
>>> hierarDualBuild(im2, im3)
>>> negate(im2, im2)
>>> logic(im3, im2, im3, "inf")
```

Note again that, in both cases, performing the inf between f and g (or the sup between f and M - g) is not necessary.



Result of the labelling of the tesselation image (right) with the minimum value of the road1 image (left) in each of its connected components.

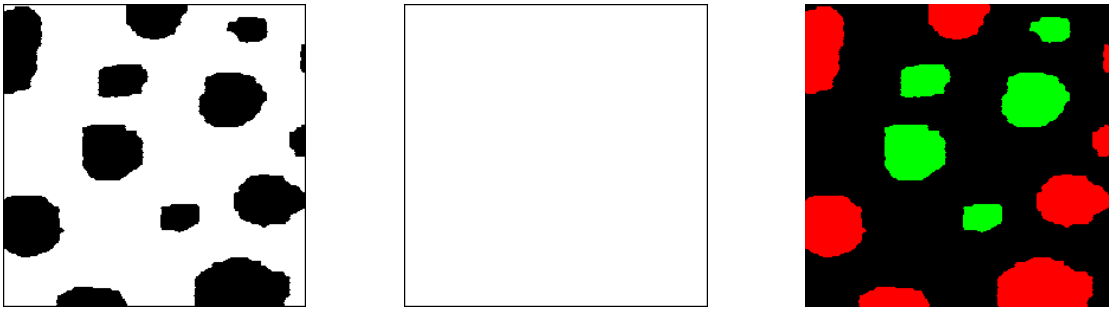
Exercise n° 4: Holes filling, objects cutting edges

1) Suppressing the particles which touch the field edges: the marking set consists of the intersection of the initial set and the field internal contour. Load the *grains2* image in imbin1 and type the following commands:

```
>>> imbin2.reset()
>>> dilate(imbin2, imbin2, edge=FILLED)
```

```
>>> build(imbin1, imbin2)
```

Note the trick for getting the edge in one step (use of a dilation of an empty image with the edge set to **FILLED**!).



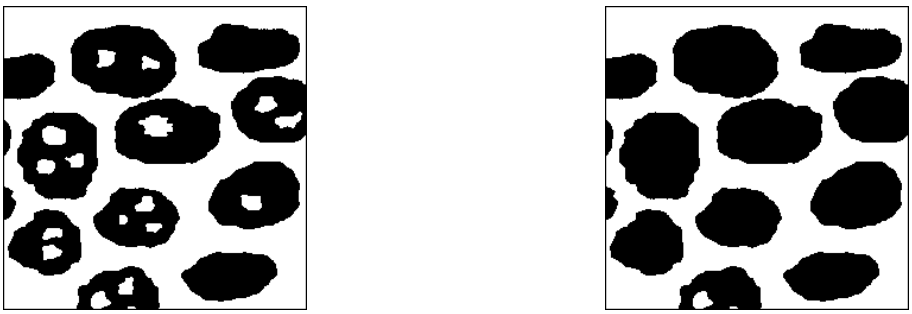
Removal of the objects which touch the edges of the field: initial image (left), marker set (field contour, middle), removed particles in red (right).

This operator already exists in the MAMBA library. It is named **removeEdgeParticles**.

2) Filling holes (binary image)
The initial set for the geodesic reconstruction is the complemented set. Then, the holes can be considered as connected components which cannot be accessed from the field edge. The marker is the same as in the preceding example and the reconstruction generates the background pixels which are connected to the edge. Load the initial image in imbin1 and type:

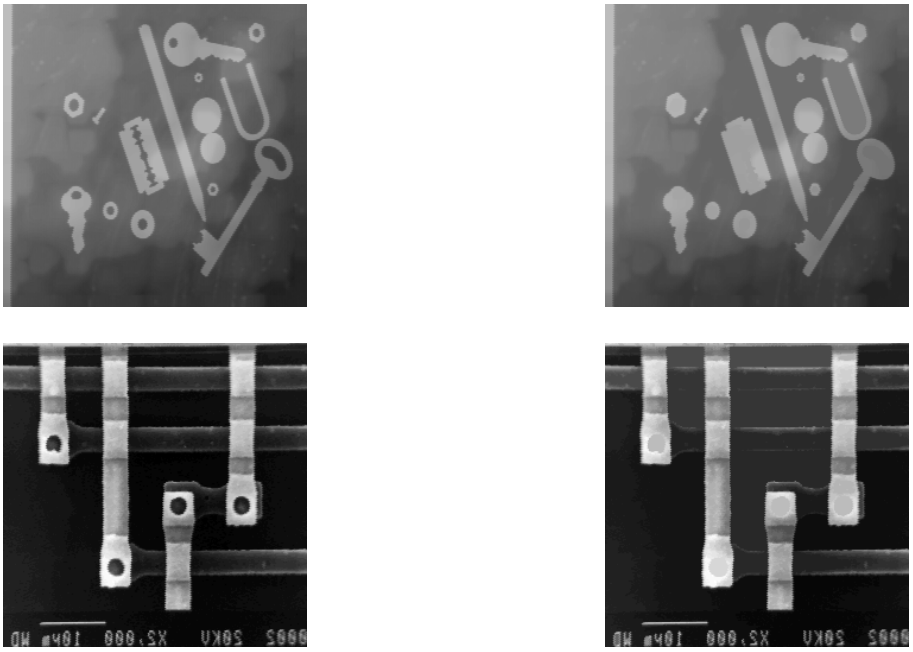
```
>>> negate(imbin1, imbin3)
>>> imbin2.reset()
>>> dilate(imbin2, imbin2, edge=FILLED)
>>> build(imbin3, imbin2)
>>> negate(imbin2, imbin2)
```

This operator, named **closeHoles**, exists in MAMBA.



Filling the holes of image grains2 (left). Note the distinctive feature at the bottom of the resulting image (right): in fact, the non filled parts are not holes according to the retained definition.

3) Filling holes in a greytone image uses exactly the same algorithm as the one described in the binary case. In fact, the same **closeHoles** operator can be used indifferently for binary or greytone images.



On the left, initial images (tools and circuit), on the right, result of the holes filling procedure.

4) The white inclusions inside the grains of the *alumine* image can be considered as "inverted holes". So, they can be removed with the **closeHoles** operator applied on the inverted image. The inital image is loaded in im1:

```
>>> negate(im1, im1)
>>> closeHoles(im1, im1)
>>> negate(im1, im1)
```

alumine image (left) and removal of the inclusions considered as white holes in the image (right).

Exercise n° 5: Regional maxima and minima

1) A section or threshold $X_i(f)$ at level i of a function f defined on E is made of the points of E such that:

$$X_i(f) = \{x \in E : f(x) \geq i\}$$

The geodesic reconstruction $R_X(Y)$ of a set X by a marker set Y ($Y \subset X$) is made of all the connected components of X which are marked by Y.

The maxima of a function f can be obtained by means of a geodesic reconstruction. Let us consider the various thresholds of f. A maximum of the function at altitude i (if it exists) will be a connected component of the threshold $X_i(f)$ of f containing no connected component of

any threshold $X_j(f)$ where $j > i$. Indeed, suppose that it is not true. Therefore, there exists a path connecting any point of the aforementioned connected component of $X_i(f)$ to any point of the connected component of $X_j(f)$ contained in the previous one. Let us take $j = i + 1$. Then path c_{xy} is the projection of a non descending path on the graph G of f. So, x cannot belong to a maximum, which proves the proposition.

So, a maximum at altitude i corresponds to a connected component of $X_i(f)$ which cannot be rebuilt by $X_{i+1}(f)$ and the set M (f) of all the maxima of f can be defined as:

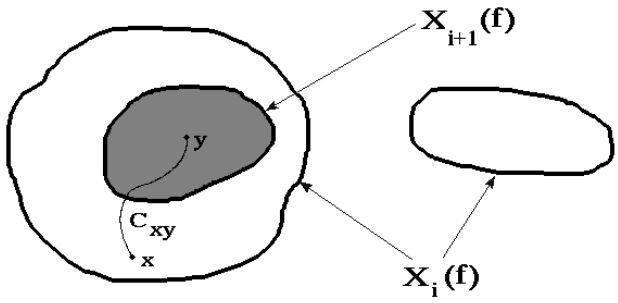
$$M(f) = \bigcup_i \left[X_i(f) \backslash R_{X_i(f)}[X_{i+1}(f)] \right]$$

2) The previous algorithm is very slow as all the sections of the image must be used sequentially. But the algorithm implemented in the **maxima** and **minima** operators uses a reconstruction $R_f(f-1)$ of the initial image f by the image f-1, which allows to process in parallel all the sections of the image.

Maxima of f are made of points of E where $f - R_f(f-1)$ is strictly positive. This function taking only two values 0 or 1 is also the indicator function $k_M(f)$ of the maxima of f.

Minima $m(f)$ of f can be obtained in a similar way by simply using the dual reconstruction R^*. We get:

$$k_{m(f)} = R_f^*(f+1) - f$$



The right connected component of X_i is a maximum as it cannot be rebuilt by any connected component of X_{i+1} (it does not contain such a component).

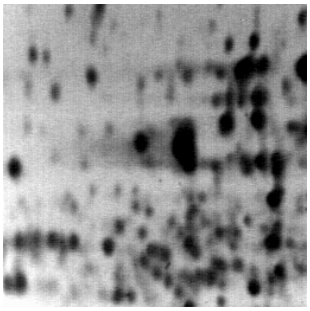
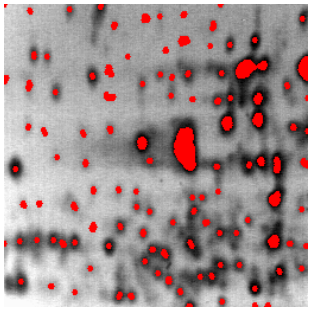
Let us detect the minima of the *electrop* image, loaded in im1:

```
>>> minima(im1, imbin1)
```

Minima (right) of the electrop image (left).

The *electrop* image contains a great number of minima, that are most often reduced to one point. This is typical of a noisy image. Then, we may apply the filters described above in order to suppress, partly at least, the noise from the image. Try different filters (openings, closings, alternate sequential filters). The alternate sequential filter of size 2, starting by an opening is quite efficient:

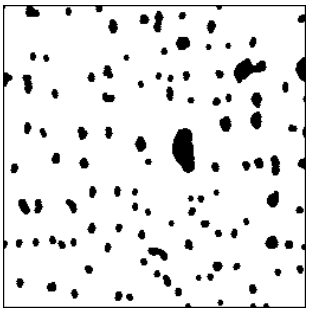
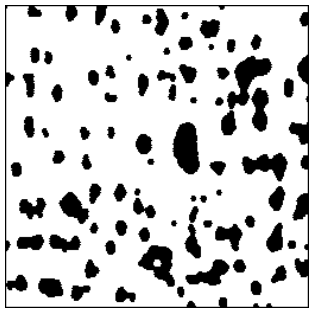
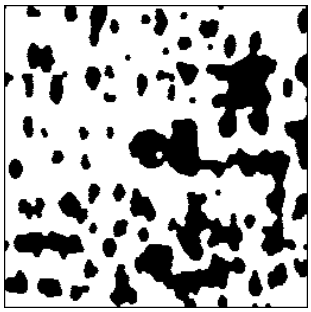
```
>>> alternateFilter(im1, im2, 2, True)
>>> minima(im2, imbin1)
```

Minima (right) of the electrop image (left) filtered with an alternate sequential filter of size 2 starting with an hexagonal opening.

3) Enter the following commands:

```
>>> alternateFilter(im1, im2, 2, True)
>>> minima(im2, imbin1, 10)
>>> minima(im2, imbin2, 50)
>>> minima(im2, imbin3, 100)
```

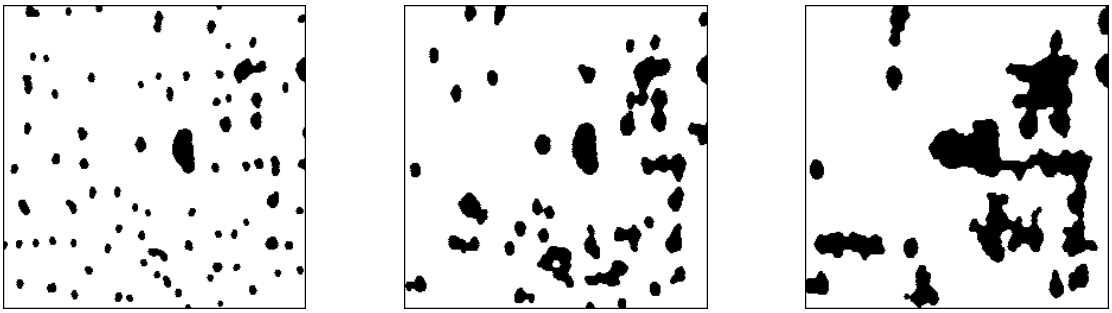




From left to right: extended minima of size 10, 50 and 100 extracted from the filtered electrop image.

As depth increases, the current binary resulting image always contains the previous one.The operator is extensive. It marks the regions of the initial image which are progressively arased by the geodesic reconstructions.

Try now the **deepMinima** operator:

```
>>> deepMinima(im2, imbin1, 10)
>>> deepMinima(im2, imbin2, 50)
>>> deepMinima(im2, imbin3, 100)
```

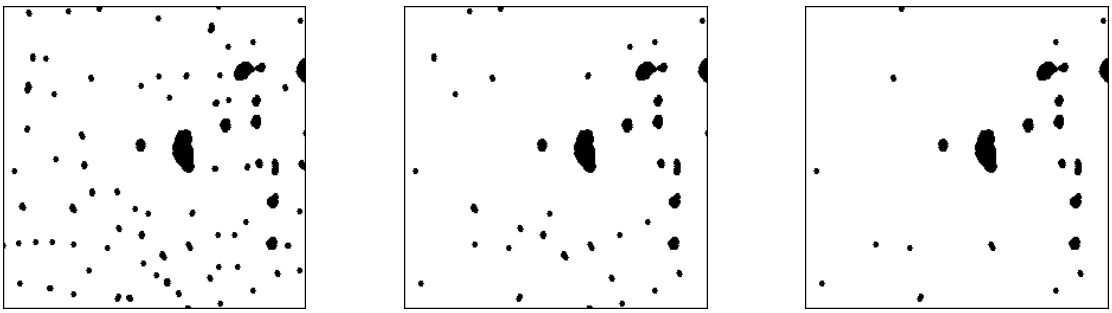


From left to right: deep minima of size 10, 50 and 100 extracted from the filtered electrop image.

This operator is no longer extensive. It preserves only he connected components of the extended minima which contain at least one minimum with a depth higher than or equal to h.

Finally, apply the **minDynamics** operator:

```
>>> minDynamics(im2, imbin1, 10)
>>> minDynamics(im2, imbin2, 50)
>>> minDynamics(im2, imbin3, 100)
```



From left to right: minima with a dynamics higher than 10, 50 and 100 extracted from the filtered electrop image.

This operator is anti-extensive: the minima with a dynamics higher than h' are always included in the minima with a dynamics higher than h when h' > h. The connected components preserved by the **minDynamics** transform of size h are the initial minima of the image whose depth is larger than or equal to h. In fact, the dynamics of a minimum is the only concept which allows to define clearly the depth of this minimum (or the height of a maximum with **maxDynamics**).

Exercise n° 6

The teeth of the wheel are extracted by means of two operators: an opening which removes the teeth (but also other features in the image) followed by the extraction of the outside of the image which corresponds to the region where the teeth are.
The size of the *wheel* image is (480, 472). First enter the following lines to define three working images. Note that the actual size of these images is (512, 472):

```
>>> imA = imageMb(480, 472, 1)
>>> imB = imageMb(imA)
>>> imC = imageMb(imA)
```

Then load *wheel* in imA and type the following commands:

```
>>> opening(imA, imB, 3)
>>> negate(imB, imC)
>>> removeEdgeParticles(imC, imB)
>>> diff(imC, imB, imC)
>>> logic(imA, imC, imC, "inf")
```

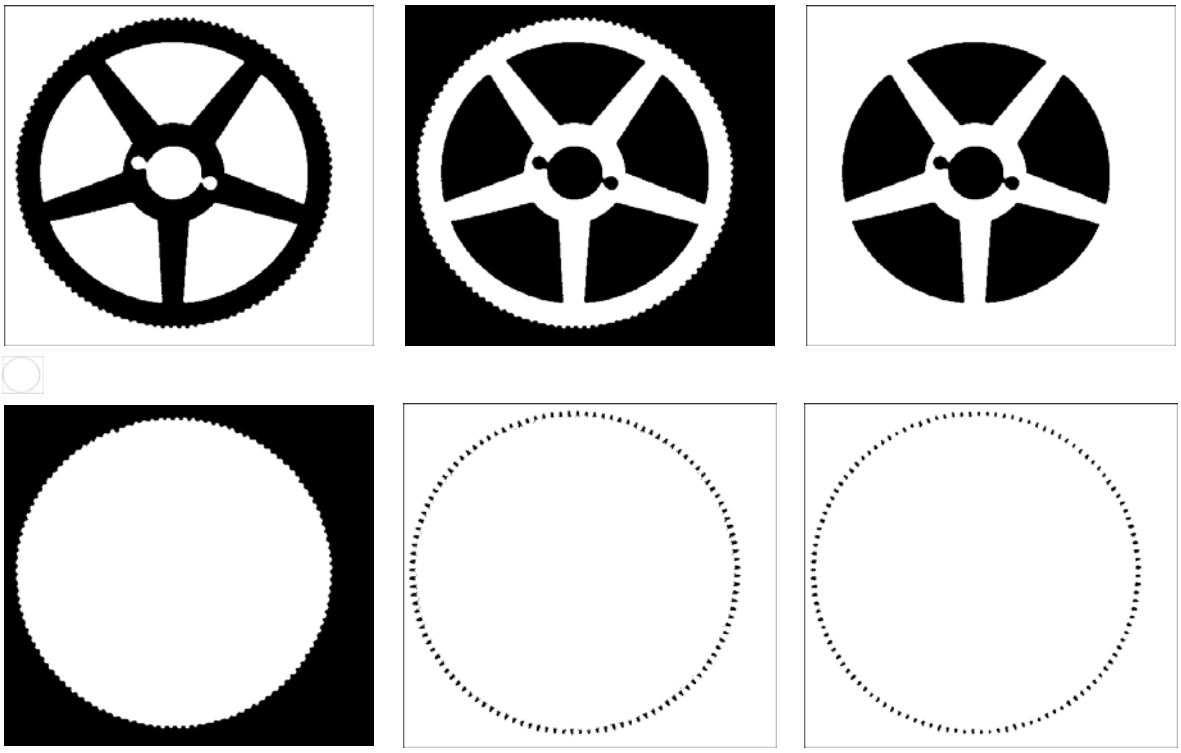
A final small opening removes the tiny defects which could remain (generally, they appear as isolated pixels):

```
>>> opening(imC, imC)
```

As the teeth are simply connected components, their number can be obtained easily with the connectivity number measure:

```
>>> computeConnectivityNumber(imC)
120L
```

120 teeth are detected.



From top to bottom and from left to right: opening of size 3 of the wheel image, negation of the result, removal of the connected component touching the edge (outside of the wheel), difference of the two last images (the outside of the opened comes back), intersection of the outside with the initial image and extracted teeth after a slight filtering.

Note that, instead of counting the teeth, the notches can be extracted, by the following procedure (*wheel* is loaded in imA again):

```
>>> negate(imA, imB)
```

```
>>> imC = imageMb(imA)
>>> removeEdgeParticles(imB, imC)
>>> diff(imB, imC, imB)
>>> closing(imB, imC, 3)
>>> logic(imA,imC, imC, "inf")
>>> opening(imC, imC)
>>> computeConnectivityNumber(imC)
120L
```

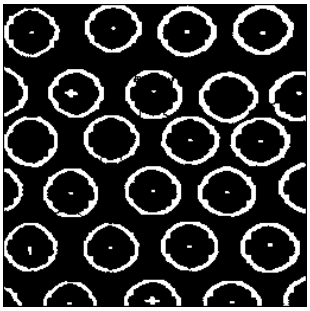
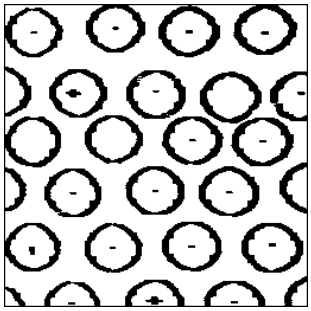
We have obviously as many teeth as notches.

Exercise n° 7: Analysis of the distribution of boron fibers

1) This exercise must be realised on the hexagonal grid. The simplification of the *boron1* image is performed in three steps :
- Connection of the fibers boundaries (using linear dilations).
- Filling of the interior of the fibers contained in the field.
- Lastly, filling of the fibers cutting the edge of the field.

a) Connection of the fibers boundaries
Perform the following operations after the *boron1* image has been loaded into imbin1:

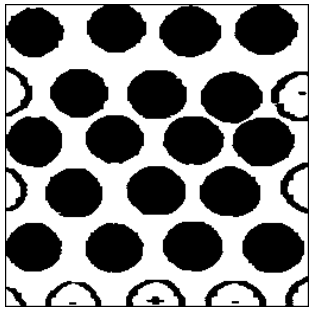
```
>>> negate(imbin1, imbin1)
>>> linearDilate(imbin1, imbin1, 5, 2)
```

Initial boron1 image (left) and horizontal linear dilation of size 2 of the complemented image (right).

b) The fibers are filled up by means of the **closeHoles** operator:

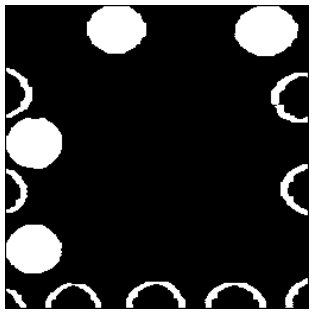
```
>>> closeHoles(imbin1, imbin2)
```



The boron fibers are partially filled up.

c) You can notice that the fibers cutting the edge of the field are not filled. Therefore, a more refined procedure must be designed to fill them. These fibers are first isolated with the operator **removeEdgeParticles** and extracted:

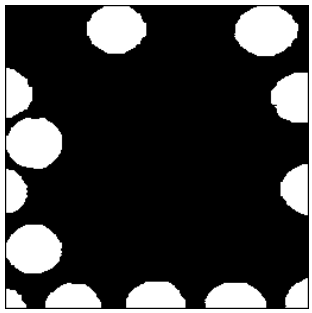
```
>>> removeEdgeParticles(imbin2, imbin3)
>>> diff(imbin2, imbin3, imbin3)
>>> negate(imbin3, imbin3)
```



Filling the exterior fibers (1st step).

We can now reconstruct the largest connected component of imbin3. In order to do so, we can generate a marker of this component by using a large erosion. The reconstruction provides the complemented image of the filled fibers which cut the edge of the field.

```
>>> erode(imbin3, imbin4, 20, edge=EMPTY)
>>> build(imbin3, imbin4)
```



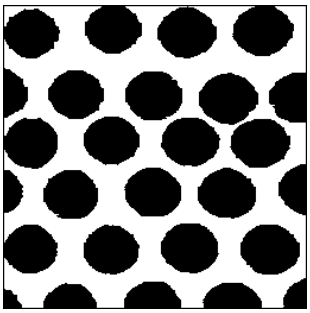
Filling the exterior fibers (2nd step).

All we have to do now is adding the imbin2 image to imbin4 (remind that imbin2 is inverted with respect to imbin4):

```
>>> diff(imbin4, imbin2, imbin2)
```

Finally, the imbin2 image has still to undergo a linear dilation of size 2 so as to counterbalance the first dilation used to connect the boundaries of the fibers. It is indeed a dilation since we are working on the complementary set. Moreover, a dilation, that is an extensive transformation, avoids edge effects. The linear dilation must use a transposed structuring element with respect to the one used in the initial linear dilation.

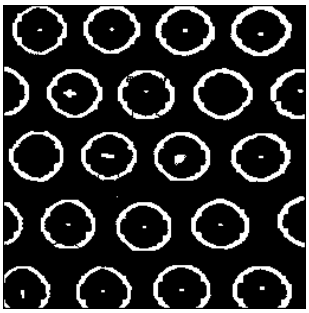
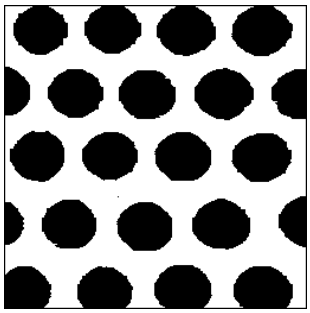
```
>>> linearDilate(imbin2, imbin2, 2, 2)
>>> negate(imbin2, imbin2)
```



Filling the exterior fibers (3rd step).

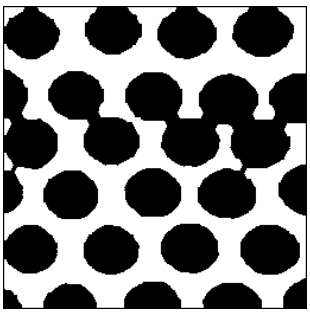
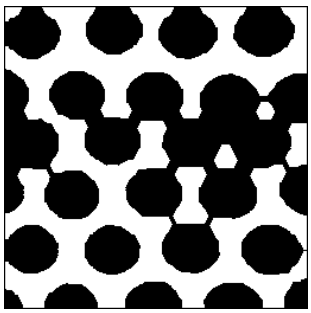
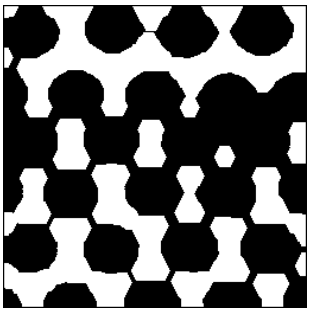
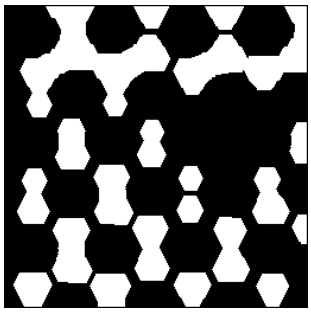
The whole procedure can be programmed in order to be used with the other boron image. Let us name it **closeFibers**:

```
from mamba import *

def closeFibers(imIn, imOut):
    """
    This operation cleans the original fibers image by closing their contours
    and by filling their interiors. Note that this operation also fills the
    fibers which are cutting the edges of the image.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)

    negate(imIn, imWrk1)
    # Linear dilation of size 2 in horizontal direction to connect
    # fiber contours.
    linearDilate(imWrk1, imWrk1, 5, 2)
    # Filling true holes (interior fibers)
    closeHoles(imWrk1, imWrk2)
    # Extracting all the fibers cutting the edges.
    removeEdgeParticles(imWrk2, imWrk3)
    diff(imWrk2, imWrk3, imWrk3)
    # The image is inverted and eroded to get a background marker.
    negate(imWrk3, imWrk3)
    erode(imWrk3, imOut, n=20, edge=EMPTY)
    # Reconstructing the background and adding fibers cutting the edges to
    # the previous ones.
    build(imWrk3, imOut)
    diff(imOut, imWrk2, imWrk2)
    # Another linear dilation in transposed direction to recover the initial
    # sizes of the fibers.
    linearDilate(imWrk2, imWrk2, 2, 2)
    # final result
    negate(imWrk2, imOut)
```

Filling the fibers of the boron2 image (left) with the closeFibers procedure (right).

2) Hexagonal closings of increasing sizes tend to connect boron fibers. This connection reduces the number of connected components and increases the number of holes. The connectivity number should then be reduced and this reduction must be more or less fast depending on the degree of regularity of the structure.

From left to right and from top to bottom: closings of increasing sizes (4, 6, 8 and 10) of the boron fibers.

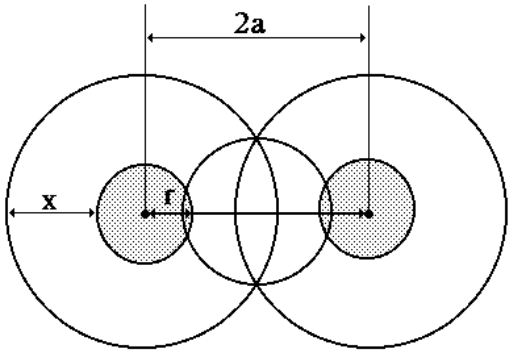
Therefore this measure, or more precisely its evolution in relation to the closing size is used to quantify the regularity of the layout. A procedure measuring the connectivity numbers of increasing closings, named **checkEvenness**, can be defined as following:

```
from mamba import *

def checkEvenness(imIn, maxSize):
    """
    Checks the regularity of the boron fibers arrangement by computing
    successive closings and by determining at each step the connectivity
    number of the closing. The variation of this measure from a positive
    to a negative value indicates the evenness of the arrangement. The
    more it is regular, the more this variation is fast and important.
    The successive connectivity numbers are returned in a list.
    """
```

```
imWrk = imageMb(imIn)
ncList =[]
for i in range(maxSize + 1):
    closing(imIn, imWrk, i)
    nc = computeConnectivityNumber(imWrk)
    ncList.append(nc)
return ncList
```

Note that as the closing size increases, the useful part of the image is reduced. Besides the fact that the measured connectivity number must be a specific number (related to the area unit), its calculation is likely to be highly imprecise in case of large closings. This is the reason why it is preferable to use geodesic closings in this example, the geodesic space being the whole field of measure. This option is the default one with closings (the edge is set to **FILLED**).

3) Assume the layout of the fibers is regular according to an hexagonal grid. Let r be the radius of the fibers and 2a the distance between their centers. Let N be the number of fibers in the field of analysis.
The fibers are all connected after a closing of size $x \geq (a^2 - r^2)/2r$. Indeed, in order that the closing connects contiguous fibers two by two, it is sufficient to be able to insert a disk of radius x at the junction of two particles.



The limit case allows to write:

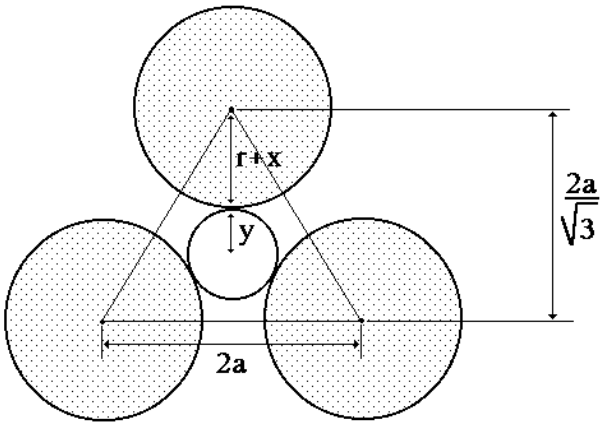
$$x^2 + a^2 = (x + r)^2, \text{ that is } x = (a^2 - r^2)/2r$$

Now, the image shows only one connected component. Calculating the number of holes is more complex. Indeed, in the stationary case, it is proved (in application of Euler formula) that the average number of holes is 2N. However, since we used geodesic transforms, this number does not correspond to the number of holes that could be measured on a regular layout. In fact, if the image contains n_1 rows of n_2 fibers, the number of holes generated by closing is approximately:

$$2(n_2 - 1)(n_1 - 1) = 2N - 2(n_1 + n_2) - 2$$

The connectivity number is then equal to :

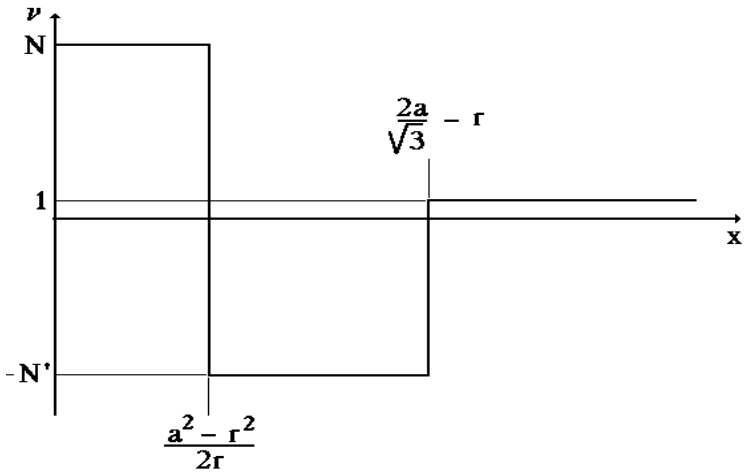
$$v = -(2N - 2(n_1 + n_2) - 3)$$



The greater the closing size increases, the more it fills the holes, until they are completely filled. This is achieved when the dilation closes the hole generated by the adjacency of three fibers, that is when:

$$r + x \geq \frac{2a}{\sqrt{3}} \text{ or } \quad x \geq \frac{2a}{\sqrt{3}} - r$$

Then the closed set is reduced to one connected component without hole. The connectivity number is then equal to 1. The theoretic curve of evolution of this number according to the closing size is the following:



In view of this curve, we can say that the major sign of the regularity of the structure is the sudden transition from highly positive values to highly negative values. The faster the transition, the greater the regularity of the fibers network. The tangent of the curve at the crossing point of the horizontal axis is then a possible quantizer of the degree of regularity.
The whole procedure is applied on the two images *boron1* and *boron2*. Don't forget to use an hexagonal grid. Start with the *boron1* image loaded in imbin1:

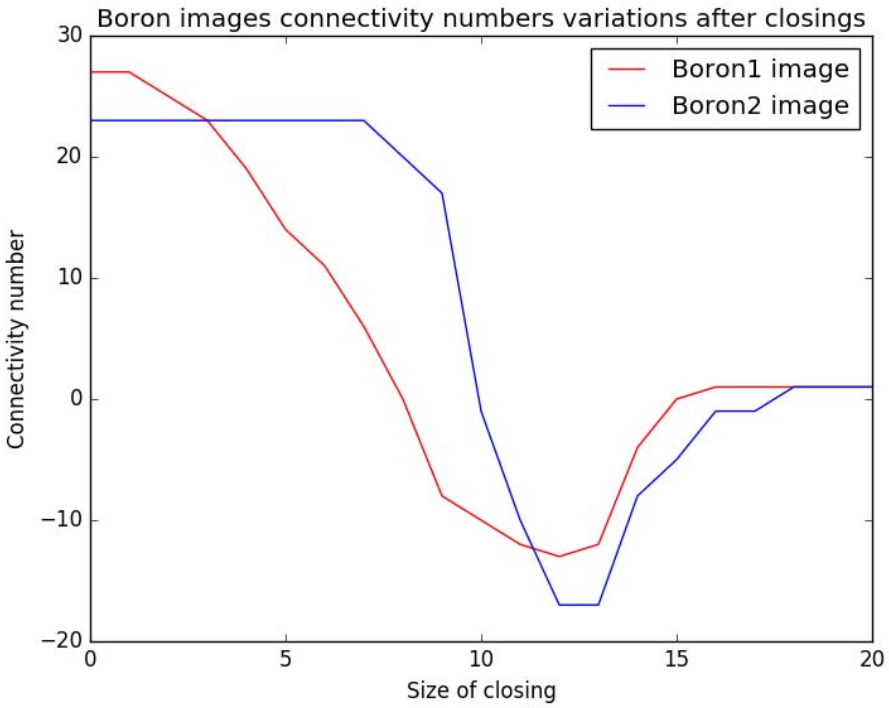
```
>>> closeFibers(imbin1, imbin2)
>>> ncList1 = checkEvenness(imbin2, 20)
```

The **ncList1** list contains the connectivity numbers measured for the first twenty closings. Then, perform the same operation for the *boron2* image, loaded in imbin1:

```
>>> closeFibers(imbin1, imbin2)
>>> ncList2 = checkEvenness(imbin2, 20)
```

The **ncList2** list contains the corresponding connectivity numbers. We can then plot the two curves. Enter the following commands:

```
>>> import matplotlib.pyplot as plt
>>> xs = range(21)
>>> ys = ncList1
>>> zs = ncList2
>>> err = plt.xlabel('Size of closing')
>>> err = plt.ylabel('Connectivity number')
>>> err = plt.title('Boron images connectivity numbers variations after closings')
>>> err = plt.plot(xs, ys, label='Boron1 image', color='red')
>>> err = plt.plot(xs, zs, label='Boron2 image', color='blue')
>>> err = plt.legend(loc='upper right')
>>> err = plt.show()
```



The slope of the blue curve (boron2) is steeper than the red one (boron1) indicating a better regularity of the fibers arrangement.

Exercise n° 8: Size distribution of a greytone image

1) We want to detect small dome-shaped structures. They are gradually eliminated by greytone openings of increasing size. Therefore, we can define a procedure, named **openSizeDistribution** for computing this granulometry in a range of sizes:

```
from mamba import *

def openSizeDistribution(imIn, sizeRange):
    """
```

```
    Computes the size distribution by hexagonal openings of the 'imIn' image
    in the range 'sizeRange'. The operator returns a list of values.
    """

    imWrk = imageMb(imIn)
    copy(imIn, imWrk)
    granuList = []
    oldVol = 0L
    for i in range(1, sizeRange + 1):
        opening(imIn, imWrk, i)
        sub(imIn, imWrk, imWrk)
        vol = computeVolume(imWrk)
        volInc = vol - oldVol
        granuList.append(volInc)
        oldVol = vol
    return granuList
```

The opening size is increased by 1 at each iteration. **SizeRange** represents the number of iterations. The value of the size distribution function for each level i corresponds to the difference in volume between the opened image of size (i - 1) and the opened image of size i.

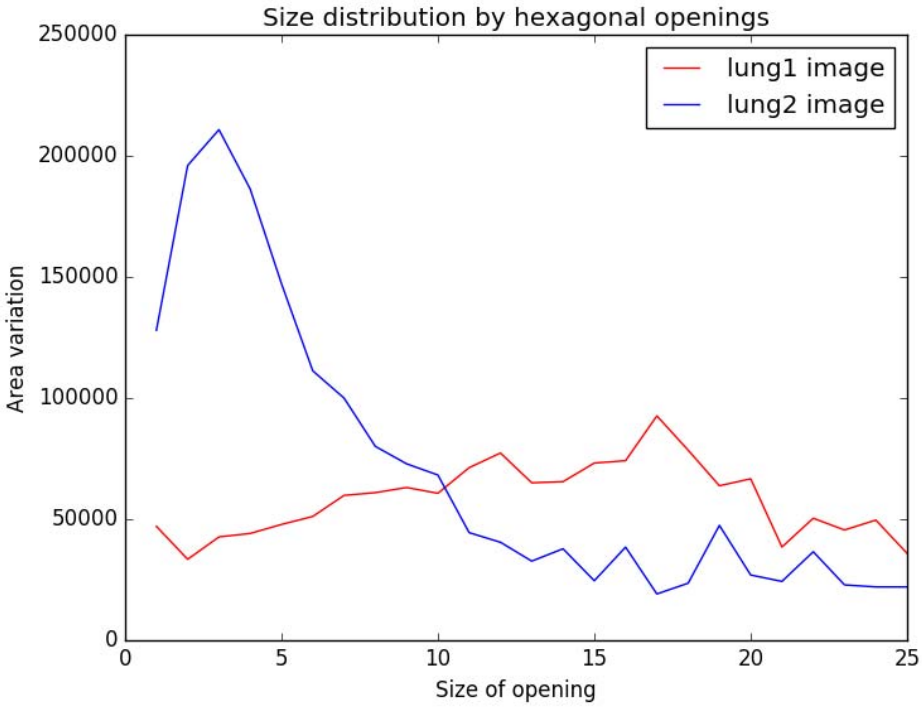
Let us apply this procedure on the two images *lung1* and *lung2* (loaded in im1 and im2):

```
>>> granuList1 = openSizeDistribution(im1, 25)
>>> granuList2 = openSizeDistribution(im2, 25)
```

Then, let us draw the two corresponding size distributions:

```
>>> import matplotlib.pyplot as plt
>>> xs = range(1, 26)
>>> ys = granuList1
>>> zs = granuList2
>>> err = plt.xlabel('Size of opening')
>>> err = plt.ylabel('Area variation')
>>> err = plt.title('Size distribution by hexagonal openings')
>>> err = plt.plot(xs, ys, label='lung1 image', color='red')
>>> err = plt.plot(xs, zs, label='lung2 image', color='blue')
>>> err = plt.legend(loc='upper right')
>>> err = plt.show()
```

It is indeed possible to differentiate the two types of texture. The radiography *lung1* does not contain any nodular structure and shows a smooth size distribution curve without any significant maximum for openings up to a given size (equal to 20 here). The size distribution curve of the radiography *lung2* shows a significant maximum for openings of small size. This maximum marks the disappearance of white nodules for openings of small size.
The position of the maximum gives an indication of the average size of the nodules scattered in the image.
It will not be possible to use the information provided by the range of the size distribution curve unless we make some assumptions concerning the volume characteristics of the nodules but it is out of the scope of this exercise.



Size distribution curves with hexagonal openings for the lung1 and lung2 images.

2) The opening by erosion-reconstruction is less active than a morphological opening. The morphological opening tends to eliminate the small irregularities of large structures. Indeed, if the irregularity cannot contain the structuring element, it is eliminated even if the structure is preserved. On the contrary, small irregular structures may be rebuilt by the geodesic reconstruction when they belong to a larger set, as it is the case with binary images (see exercise 2).
As nodules are regular structures with delimited contours, the opening by reconstruction allows a better discrimination between these nodules and other irregular structures. The size distribution based on openings by reconstruction is defined by the following procedure, named **buildOpenSizeDistribution**:

```
from mamba import *

def buildOpenSizeDistribution(imIn, sizeRange):
    """
    Computes the size distribution by openings by reconstruction of the
    'imIn' image in the range 'sizeRange'. The operator returns a list of values.
    """

    imWrk = imageMb(imIn)
    copy(imIn, imWrk)
    granuList = []
    oldVol = 0L
    for i in range(1, sizeRange + 1):
        buildOpen(imIn, imWrk, i)
        sub(imIn, imWrk, imWrk)
        vol = computeVolume(imWrk)
        volInc = vol - oldVol
        granuList.append(volInc)
```

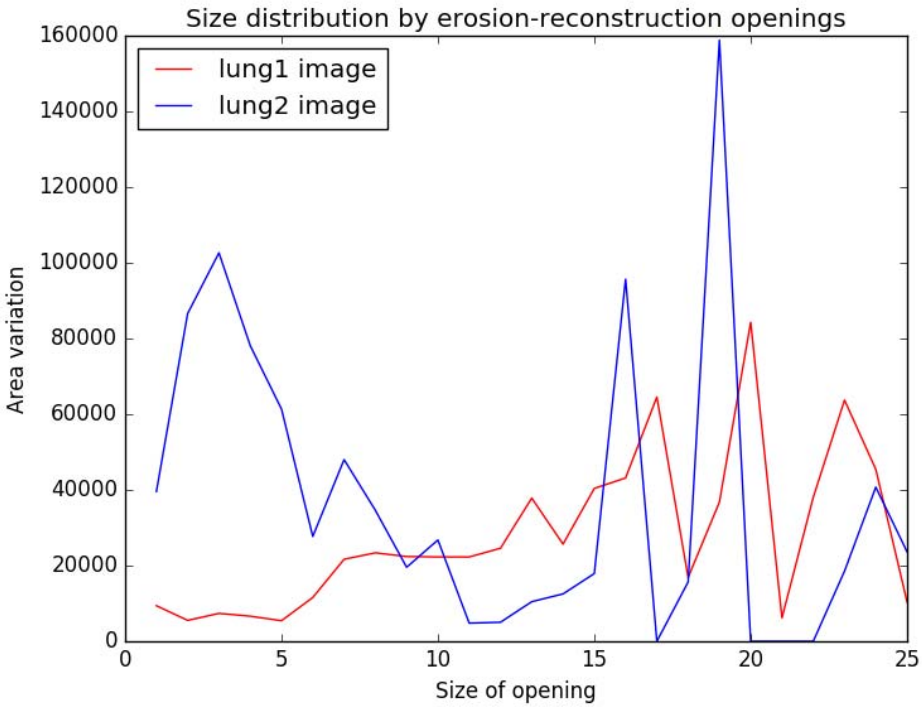
```
        oldVol = vol
    return granuList
```

Apply this operator on the two images:

```
>>> granuList1 = buildOpenSizeDistribution(im1, 25)
>>> granuList2 = buildOpenSizeDistribution(im2, 25)
```

Then, draw the two corresponding size distributions:

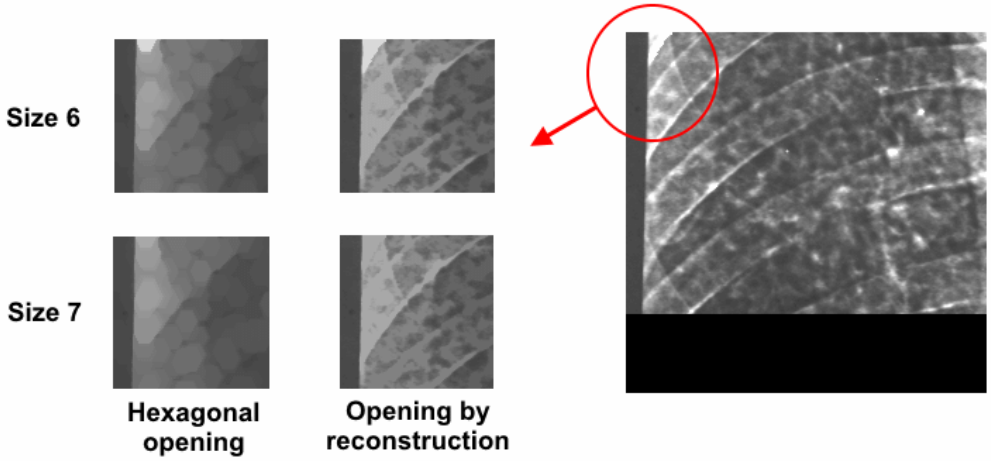
```
>>> import matplotlib.pyplot as plt
>>> xs = range(1, 26)
>>> ys = granuList1
>>> zs = granuList2
>>> err = plt.xlabel('Size of opening')
>>> err = plt.ylabel('Area variation')
>>> err = plt.title('Size distribution by erosion-reconstruction openings')
>>> err = plt.plot(xs, ys, label='lung1 image', color='red')
>>> err = plt.plot(xs, zs, label='lung2 image', color='blue')
>>> err = plt.legend(loc='upper left')
>>> err = plt.show()
```



Size distribution curves with erosion-reconstruction openings for the lung1 and lung2 images.

The range of the size distribution curve of the opening by erosion-reconstruction (RO in abbreviated form) is less extended than that of the morphological opening (MO) because the opening by erosion-reconstruction always eliminates less features than the classical opening. However since we work on differential distributions (difference of two openings of size i and i+1), it may happen that, for a given size of opening, the size distribution curve of the RO is

higher than that of the MO. However, the integral of the size distribution function by RO is always smaller than or equal to that of the morphological opening.
Some maxima may appear on the size distribution curve by reconstruction whereas they were not visible on the classical size distribution. This is due to the fact that a large structure may gradually disappear at each step of the morphological opening whereas it disappears suddenly when we use an opening by reconstruction. This phenomenon is noticeable on the *lung2* image. The white feature at the upper left part of the image is suddenly eliminated by the opening by reconstruction. If both types of size distribution are applied to this image up to size 20, a maximum appears on the size distribution by reconstruction whereas nothing indicates its disappearance on the other size distribution curve.



Difference between the hexagonal opening and the erosion-reconstruction opening observed on the left feature (red circle) in the lung2 image. The evolution between sizes 6 and 7 is progressive in the case of the hexagonal opening (the white feature size decreases gradually), whereas it is sudden with the erosion-reconstruction opening.

3) The nodules are almost round-shaped. Other white objects are present but their shape is more elongated. Using an approximately circular structuring element (hexagon or square) amounts to eliminate the volume structures according to the smallest width criterion. A narrow and elongated object will be eliminated at the same time as a circular object whose diameter is equal to the width of the elongated object. In order to detect the nodules while proceeding to the size distribution, it is preferable to use linear structuring elements. Then elongated objects will be eliminated by an opening of a size equal to their length.
The operator **supBuildOpen** performs linear openings in the three main directions of the hexagonal grid and then takes the sup of the three openings. Thus each structure that contains at least one elongated element is preserved. We know that the sup of three linear openings is an opening. Then, this opening is used as a marker for the geodesic reconstruction of the original image.

```
def buildSupOpen(imIn, imOut, size):
  """
  Performs on 'imIn' the opening by reconstruction with a marker made of a sup of
  linear openings. The result is put in 'imOut'.
  """

  imWrk = imageMb(imIn)
  supOpen(imIn, imWrk, size)
```

```
  hierarBuild(imIn, imWrk)
  copy(imWrk, imOut)
```

Then, this operator is used for computing the corresponding size distribution. It is named **buildSupOpenSizeDistribution**:

```
def buildSupOpenSizeDistribution(imIn, sizeRange):
  """
  Computes the size distribution by openings by reconstruction with
  a sup marker of the
  'imIn' image in the range 'sizeRange'. The operator returns a list of
  values.
  """

  imWrk = imageMb(imIn)
  copy(imIn, imWrk)
  granuList = []
  oldVol = 0L
  for i in range(1, sizeRange + 1):
     buildSupOpen(imIn, imWrk, i)
     sub(imIn, imWrk, imWrk)
     vol = computeVolume(imWrk)
     volInc = vol - oldVol
     granuList.append(volInc)
     oldVol = vol
  return granuList
```

Once again, it is applied on *lung1* and *lung2* images:

```
>>> granuList1 = buildSupOpenSizeDistribution(im1, 25)
>>> granuList2 = buildSupOpenSizeDistribution(im2, 25)
```

Then, the corresponding size distribution curves are drawn:

```
>>> xs = range(1,26)
>>> ys = granuList1
>>> zs = granuList2
>>> err = plt.xlabel('Size of linear opening')
>>> err = plt.ylabel('Area variation')
>>> err = plt.title('Size distribution by reconstruction from sup of linear openings')
>>> err = plt.plot(xs, ys, label='lung1 image', color='red')
>>> err = plt.plot(xs, zs, label='lung2 image', color='blue')
>>> err = plt.legend(loc='upper right')
>>> err = plt.show()
```

The range of this size distribution function is much smaller because, at each step of the opening, much less structures are eliminated. Indeed, the criterion is no longer: "can this structure contain an hexagon of diameter n?" but "can this structure contain a segment of length n?". Moreover, the next operation is a reconstruction, which allows to preserve small structures neighboring larger ones.

Size distribution curves with a geodesic reconstruction from a sup of linear openings for the lung1 and lung2 images.

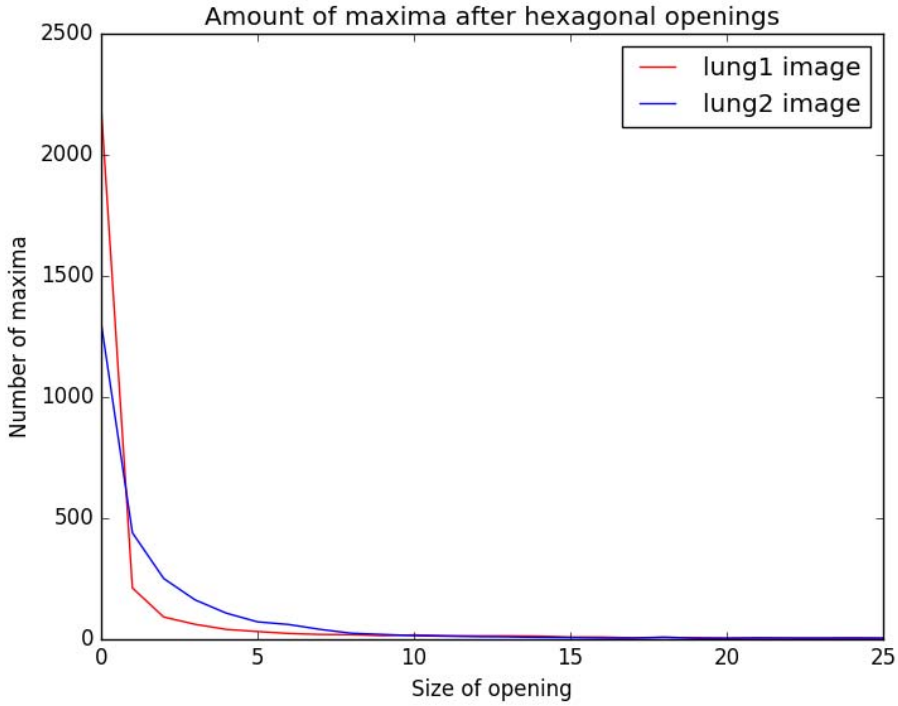
On the *lung2* radiography, the maximum characterizing the presence of nodules is shifted to the right side of the x-axis. This is due to the fact that the selecting criterion is now the longest radius contained in the object and that the nodules are not perfectly circular. If a non circular nodule has a radius of 7 pixels in one direction and a radius of 8 pixels in the opposite direction, it will disappear with an opening of size 7 using an hexagonal structuring element and an opening of size 8 with a sup of linear openings. Its position is then shifted to the right by one unit on the size distribution curve.

4) On a greytone image, it is possible to count its maxima and minima. These points are significant as they indicate the presence of a dome or a valley. Of course, this information is not sufficient to know the size and depth of these domes and valleys. Thus, impulse noise may cause maxima or minima but we cannot properly compare the latter to minima produced by large structures. However, it is interesting to follow the evolution of the number of maxima or minima after openings and closings. The **granulNumber** operator computes the number of maxima after openings of increasing sizes:

```
def granulNumber(imIn, sizeRange):
  """
  Computes the pseudo size distribution function provided by
  the number of maxima in the various openings in the range 'sizeRange'.
 The operator returns a list of values.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn, 1)
  imWrk3 = imageMb(imIn, 32)
  numberList = []
```

```
    for t in range(sizeRange+1):
        opening(imIn, imWrk1, i)
        maxima(imWrk1, imWrk2)
        nb = label(imWrk2, imWrk3)
        numberList.append(nb)
    return numberList
```

This measure is not a size distribution since it does not fulfil some properties required by a size distribution. In particular, it is not increasing. However, the resulting curve can be used as a texture index. The **granulNumber** operator is not differential. It measures the number of maxima and not its variation at each step of opening.



Number of maxima after openings in the lung1 and lung2 images.

The following command lines allow to compute and to draw the corresponding curves:

```
>>> numberList1 = granulNumber(im1, 25)
>>> numberList2 = granulNumber(im2, 25)
>>> xs = range(26)
>>> ys = numberList1
>>> zs = numberList2
>>> err = plt.xlabel('Size of opening')
>>> err = plt.ylabel('Number of maxima')
>>> err = plt.title('Amount of maxima after hexagonal openings')
>>> err = plt.plot(xs, ys, label='lung1 image', color='red')
>>> err = plt.plot(xs, zs, label='lung2 image', color='blue')
>>> err = plt.legend(loc='upper right')
>>> err = plt.show()
```

In *lung1* and *lung2* radiographs, we can see a very high number of maxima on the non filtered images. These maxima do not correspond to structures but to noise. For openings of

small size, it can be noticed that the number of maxima is greater in the lung2 image. This indicates the presence of nodules.

In conclusion, we saw that size distribution functions inform us on the presence of nodules in the image and on their mean size. In a wider context, a size distribution function may serve to characterize the texture of a greytone image. It is possible to define a large number of size distribution functions from transformations such as openings or closings. The relative efficiency of these functions always depends on the type of images on which they are applied. In our case, a set of varied images would be necessary to decide of the most appropriate function.

REFERENCES

[1] C.Lantuejoul, S.Beucher: On the use of the geodesic metric in image analysis
(**http://cmm.ensmp.fr/~beucher/publi/Use_of_Geodesic.pdf**)
This paper introduces the concept of geodesic transformation.

[2] S.Beucher: Geodesy and geodesic transformations
(**http://cmm.ensmp.fr/~beucher/publi/Course2008_Geodesy_SB_eng.pdf**)
These lecture slides (used in the Summer school of Mathematical Morphology) introduce the geodesic transforms.

[3] S.Beucher: Maxima and Minima: A Review
(**http://cmm.ensmp.fr/~beucher/publi/maxima-minima.pdf**)
The purpose of this document is just to provide additional information about the concepts of maxima and minima in MM. These concepts are often misleading and this document can be considered as a clarification of these notions.

Chapter 7

RESIDUES I

The following exercises introduce morphological transformations which belong to the class of residual operators. Among them, the gradient and top-hat operators are rather simple (they use simple primitives). Other operators use families of primitives. In this chapter, residues using erosions and openings will be addressed. The next chapter will be devoted to residues based on the Hit-or-Miss Transform (HMT) and on thinnings and thickenings.

1 Residual transforms, general definition

An elementary residual operator r is built with the difference of two operators ψ and ζ called primitives (set difference of algebraic difference):

r = ψ\ζ (binary case)

r = ψ - ζ (algebraic case)

Many morphological contrast operators belong to this class of operators.
More refined residual operators can be defined with families of primitives, $\{\psi_i\}$and $\{\zeta_i\}$. In this case, the residual transform is composed of a doublet of operators:

$$\theta = \sup_{i \in I}(\psi_i - \zeta_i)$$

$$q = \arg\max(\psi_i - \zeta_i) + 1$$

2 Gradients

2.1 Classical gradient

The gradient of a function f defined on $\mathbb{R}^2$ is defined as the vector:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

In the digital case, the first-order differences may be used to express the partial derivatives:

$$(\nabla_x f)(x,y) \approx f(x,y) - f(x-1,y)$$

$$(\nabla_y f)(x,y) \approx f(x,y) - f(x,y-1)$$

These expressions correspond to digital convolutions of f with the kernels [-1 1] and $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

We could also use the following differences as the expression of the partial derivatives:

$$(\nabla_{2x} f)(x,y) \approx f(x+1,y) - f(x-1,y)$$

$$(\nabla_{2y} f)(x,y) \approx f(x,y+1) - f(x,y-1)$$

They have the advantage of being centered in (x,y). These derivatives are performed with the digital convolutions of f by the kernels [-1 0 1] and $\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$.

In practice, other differences are also used, which may be expressed in terms of digital convolutions too. An abundant literature illustrates this subject. For example, the Sobel operator is given for the following convolutions:

$$\frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$

2.2 Morphological gradient

The morphological gradient of a function f defined on $\mathbb{R}^2$ or on a sub-set is given by:

$$g(f) = \lim_{\lambda \to 0} \frac{(f \oplus \lambda B) - (f \ominus \lambda B)}{2\lambda}$$

where λB denotes a ball of radius λ.
On a hexagonal grid, we get:

$$g(f) = \frac{(f \oplus H) - (f \ominus H)}{2}$$

where H is a hexagon of size 1. This gradient is equal to the gradient module of a function f continuously derivable. Most of the time, we use the function 2g (no division).
Half gradients can also be defined by:

$$g^+(f) = (f \oplus H) - f$$
$$g^-(f) = f - (f \ominus H)$$

The morphological gradient is a residual operator ($\psi = \delta$ and $\zeta = \varepsilon$).

3 Top-hat transform

The top-hat is a transformation that only applies to greytone images. The top-hat wth_λ of a function f is defined by:

$$wth_\lambda(f) = f - \gamma_\lambda(f) \quad \text{(white Tophat)}$$

where γ_λ is an opening of f with a structuring element of size λ.
Likewise, the conjugated top-hat bth_λ of a function f is defined by:

$$bth_\lambda(f) = \varphi_\lambda(f) - f \quad \text{(black Tophat)}$$

where φ_λ is a closing of f.
Black and white top-hats are also elementary residual operators.

4 Other residues

Many other residual transforms can be defined with various sequences of primitives: ultimate erosion, ultimate opening, skeletons by openings, etc. Some of these operators will be introduced in the exercises.

EXERCISES

Exercise n° 1

1) On the *road* image, apply the MAMBA **gradient** operator of size λ of a function f:

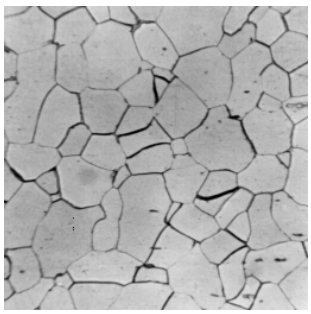
$$g_\lambda(f) = (f \oplus \lambda B) - (f \ominus \lambda B)$$

where λB denotes a ball of size λ (take here B = H, hexagon or square). This gradient is called thick gradient. The default value of λ is 1.
Prove and verify that the gradient of size λ is always greater than or equal to the gradient of size μ when $\lambda \geq \mu$.
What is the interest of using thick gradients and what is its drawback?
2) The top-hat $wth_n(f)$, named **whiteTopHat**, associated with the opening by a hexagon of size n and the conjugated black top-hat $bth_n(f)$, named **blackTopHat** are also available in MAMBA. Apply these transformations to the *circuit*, *electrop*, *grains3*, *retina1* and *retina2* images. Is the following property:

$$\forall \mu \geq \lambda,\ wth_\mu \geq wth_\lambda$$

true?
3) Note that these operations do not permit to discriminate aneurisms (small white spots) from blood vessels on the *retina1* and *retina2* images. Find a top-hat allowing such discrimination.

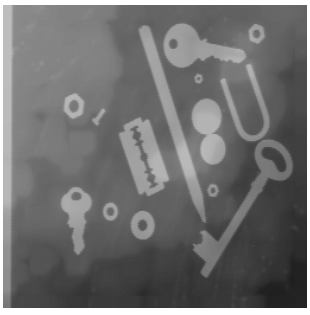
road



grains3

Exercise n° 2

The *tools* image has been acquired with a non uniform light. Propose an algorithm to get a uniform background. If this algorithm uses some parameters, indicate the rules which are used to set them.



tools

Exercise n° 3: Ultimate erosion of a (binary) set

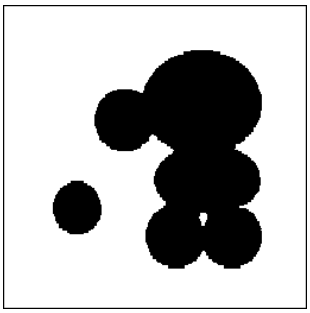
Let X be a set. The ultimate erosion of X is defined by:

$$U(X) = \bigcup_n \{x \in X \ominus nH; d_{X \ominus nH}(x, X \ominus (n+1)H) = +\infty\}$$

U(X) is then the set of the connected components resulting from the successive erosions of X that cannot be reconstructed after the erosion of the immediately larger size. This operator is a residual one, with:

$$\psi_i = \varepsilon_i \text{ and } \zeta_i = \gamma_{rec}(\varepsilon_i)$$

where $\varepsilon_i$is the erosion of size i of the initial set and $\gamma_{rec}(\varepsilon_i)$ the geodesic reconstruction of this eroded set by its elementary opening.
1) Apply this operator, named **binaryUltimateErosion**, to the *disks* image. Compute the number of disks in the aggregate.
2) Try to determine the limits of this transformation as a tool for separating particles.



disks

Exercise n° 4: Skeleton by maximal balls (binary case)

Let X be a set. A ball of radius λ included in X is said to be a maximal ball if and only if no ball of a radius strictly larger and containing the ball of radius λ can be found in X.

B_λ maximal: $B_\lambda \subset X$; there exists no B_μ , $\mu > \lambda$, $B_\lambda \subset B_\mu \subset X$

The locus S(X) of the centers of the maximal balls of X is called the skeleton of X. When X is defined on the hexagonal grid, the notion of maximal ball is replaced by that of maximal hexagon. The purpose of this exercise is to define a residual operator which corresponds to the skeleton S(X) of X.
1) Let $X \ominus nH$, be the erosion of size n of X. Prove that if x is a point of $X \ominus nH$ which does not belong to the open set $(X \ominus nH)_H$, this point is the center of a maximal hexagon of size n.
2) Derive the formula of the skeleton S(X) of X, expressed as a residual transform.
3) To any point x of S(X), can be associated the radius r(x) of the maximal hexagon centered in x. The function r(x) of support S(X) is called a quench function. Verify that the pair (S(X),r(x)) is sufficient to reconstruct the set X:

$$X = \bigcup_{x \in S(X)} H(x, r(x))$$

4) Try this operator, named **binarySkeletonByOpening**, and compare the skeleton of the set X with its ultimate erosion.
5) What are the drawbacks of this transformation in the digital case?

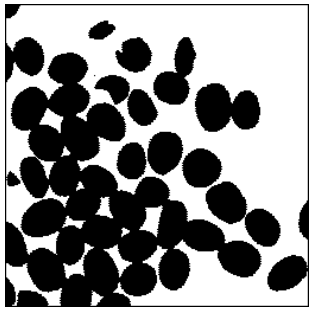
Exercise n° 5: Distance function and maxima

This exercise considers again the distance function introduced in chapter 2, exercise n° 7 and the corresponding procedure **computeDistance**.
1) A point x of a digital function f is said to be a local maximum if all the first neighbor points y of x verify the inequality:

$$f(y) \leq f(x)$$

Prove that the local maxima of the hexagonal distance function are the points of the skeleton by maximal balls. Illustrate it on the *coffee* image. How extract this skeleton from the distance function?

2) Prove that the regional maxima of the distance function are the ultimate eroded sets. Verify it on the *coffee* image. What can be the interest of the h-maxima of the distance function, with h > 1? Test it on the *coffee* image.



coffee

Exercise n° 6: Directional gradient

The morphological gradient is easy to implement and use. However, it provides only the gradient modulus of the gradient vector and not its orientation or azimuth. This azimuth can be computed by means of a residual transform built with directional gradients.

1) The morphological directional gradient of a function f in the direction a of the grid is defined by:

$$g^{\alpha}(f) = (f \oplus L^{\alpha}) - (f \ominus L^{\alpha})$$

where L^{α} is an elementary segment in direction α of the grid (6 directions for the hexagonal grid).

Define this operator with MAMBA. Name it **directionalGradient**. Limit your definition to the hexagonal grid. Apply it to the *fiber1* image.

2) A residual operator can be defined with the directional gradient. We can define:

$$\psi_i(f) = f \oplus L^i \; ; \; \zeta_i(f) = f \ominus L^i$$

where L^i is the elementary segment in direction i, $i \in I = [1, ..., 6]$ in the hexagonal grid. Then, the residue r_i is equal to:

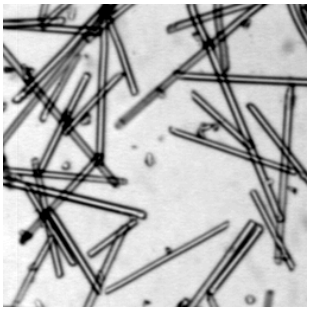
$$r_i = \psi_i - \zeta_i = g^i, \text{ directional gradient in direction i}$$

What is $g = \sup_{i \in I} g^i$?

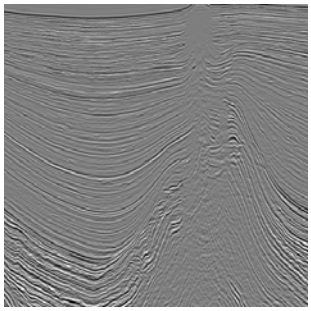
The azimuth az corresponds to the direction i where $g_i = g$. We have:

$$az = \arg \max_{i \in I} g^i$$

For sake of simplicity, we assume that the direction az is unique, which is not true in practice. Program this residual operator (named **vectorGradient**) with MAMBA and test it on the *seismic_section* image.



fiber1



seismic_section

3) What can you say about the exactness of this operator?

SOLUTIONS

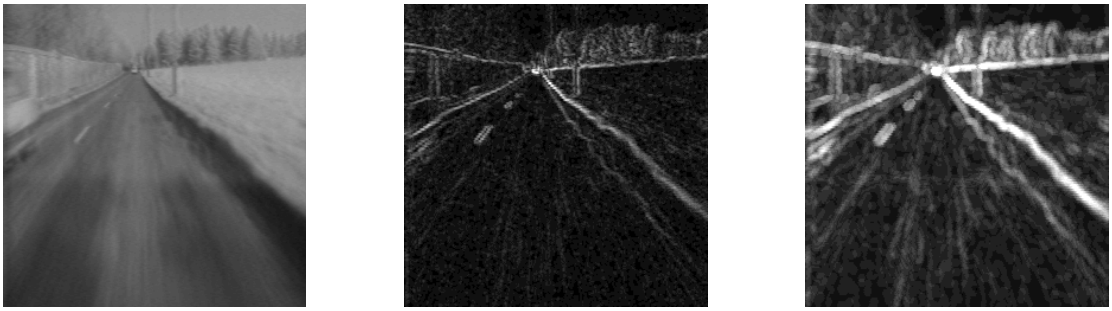
Exercise n° 1

1) Load the *road* image in im1 and type:

```
>>> gradient(im1, im2)
```

Then, perform a thick gradient of size 3 (result in im3):

```
>>> gradient(im1, im3, 3)
```



From left to right: initial road image, morphological gradient of size 1 and thick gradient of size 3 (the gradient images have been multiplied by 4 for a better contrast).

We write the thick gradient g_λ as:

$$g_\lambda(f) = \delta_\lambda(f) - \varepsilon_\lambda(f)$$

where δ_λ is a dilation with a ball λB and ε_λ, the dual erosion. Consider $\mu \geq \lambda$. We have:

$$\delta_\lambda(f(x)) = \sup_{y \in \lambda B_x} f(x), \ \ \varepsilon_\lambda(f(x)) = \inf_{y \in \lambda B_x} f(x)$$

Therefore, as $B_\lambda \subset B_\mu$, we have:

$$\delta_\lambda(f(x)) \leq \delta_\mu(f(x)), \forall x$$

(all the pixels contained in λB_x are also contained in μB_x).
We have also:

$$\varepsilon_\lambda(f(x)) \geq \varepsilon_\mu(f(x)), \forall x$$

Therefore:

$$g_\mu(f) = \delta_\mu(f) - \varepsilon_\mu(f) \geq \delta_\lambda(f) - \varepsilon_\mu(f) \geq \delta_\lambda(f) - \varepsilon_\lambda(f) = g_\lambda(f)$$

This inequality can be verified by using the **generateSupMask** operator:

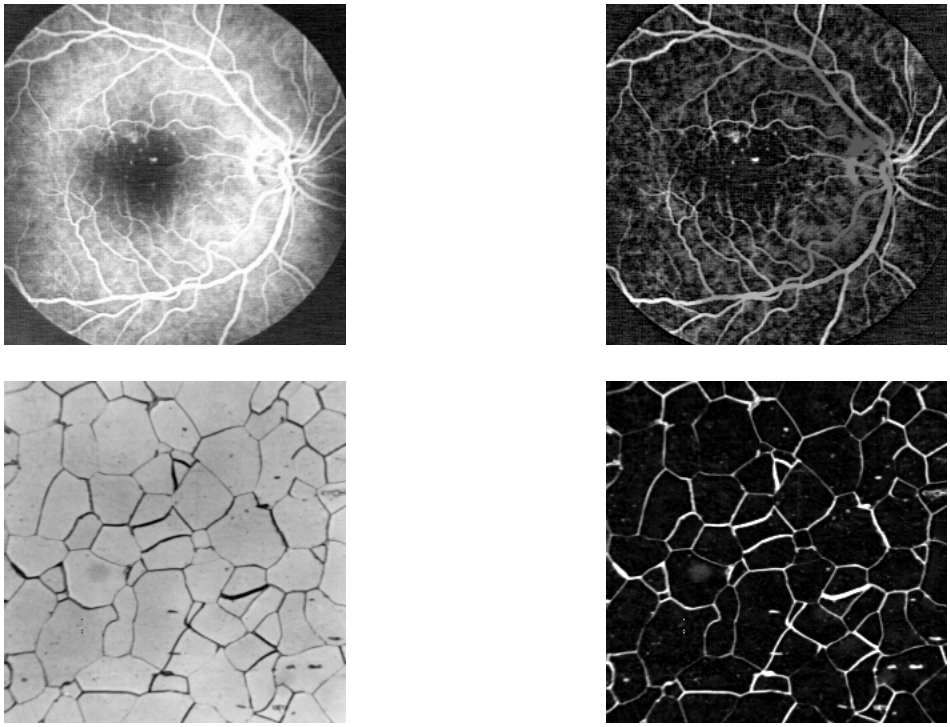
```
>>> generateSupMask(im3, im2, imbin1, False)
```

This operator puts a white pixel in imbin1 when the corresponding pixel value in im3 is greater than or equal to its value in im2. You can verify that the imbin1 image is entirely filled.
The interest of thick gradients lies in the fact that, when contours are fuzzy, the contrast measurement between the regions separated by these contours is better (the difference

between the sup and the inf is calculated on a larger support). Conversely, it is difficult to estimate the appropriate size of the thick gradient and when it is too large, close contours are likely to merge.

2) Load *retina1* in im1 and *grains3* in im2 and enter these commands:

```
>>> whiteTopHat(im1, im3, 10)
>>> blackTopHat(im2, im4, 8)
```



Top: white hexagonal top-hat of size 10 (right) of the retina1 image. Bottom: black top-hat of size 8 of the grains3 image.

We have:

$$wth_\lambda = I - \gamma_\lambda$$
$$wth_\mu = I - \gamma_\mu$$

$\forall \mu \geq \lambda$, $wth_\mu \geq wth_\lambda$ is true if and only if $\gamma_\lambda \geq \gamma_\mu$. This inequality is true when the opening is a size distribution or granulometry (see chapter 4, exercise n° 1).

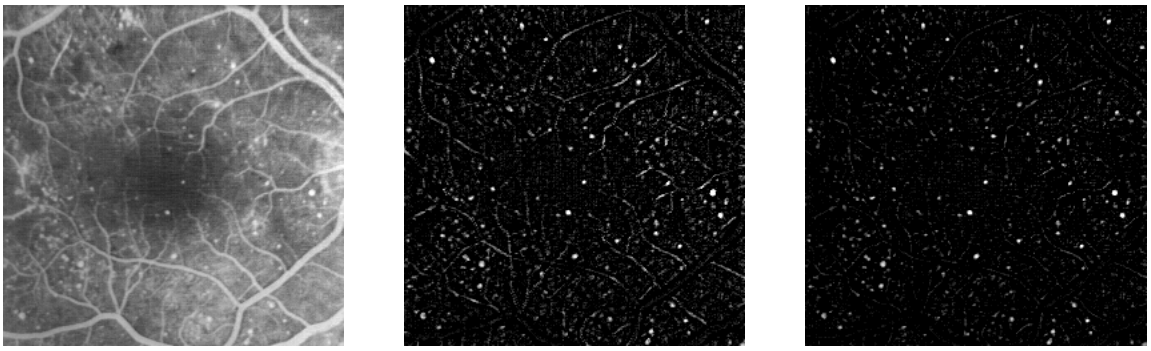
3) We already saw (chapter 4, exercise n° 4) that using the sup of linear openings allows to discriminate between the vessels and the aneurisms in the *retina2* image. Therefore, a top-hat transform based on the sup of linear openings (**supOpen**) can be designed. This operator exists in the MAMBA library and is named **supWhiteTopHat**. Load the *retina2* image in im1 and type:

```
>>> supWhiteTopHat(im1, im2, 5)
```

This operator provides a good discrimination. But this first result can be emphasized if a geodesic reconstruction using the sup of openings as marker is applied to the initial image. Doing so, the vessels are better rebuilt. Then the following subtraction produces a better

extraction of the aneurisms. Load the *retina2* image in im1 and enter the following commands:

```
>>> supOpen(im1, im2, 5)
>>> hierarBuild(im1, im2)
>>> sub(im1, im2, im2)
```



Retina2 image (left), top-hat with a sup of linear openings (middle) and top-hat when a geodesic reconstruction is performed before the subtraction (right). Some parts of vessels still remaining in the middle image have totally disappeared in the right one.

Although the result is not perfect, it is a good start for more sophisticated algorithms aiming at extracting aneurisms in eye fundus images.

Exercise n° 2

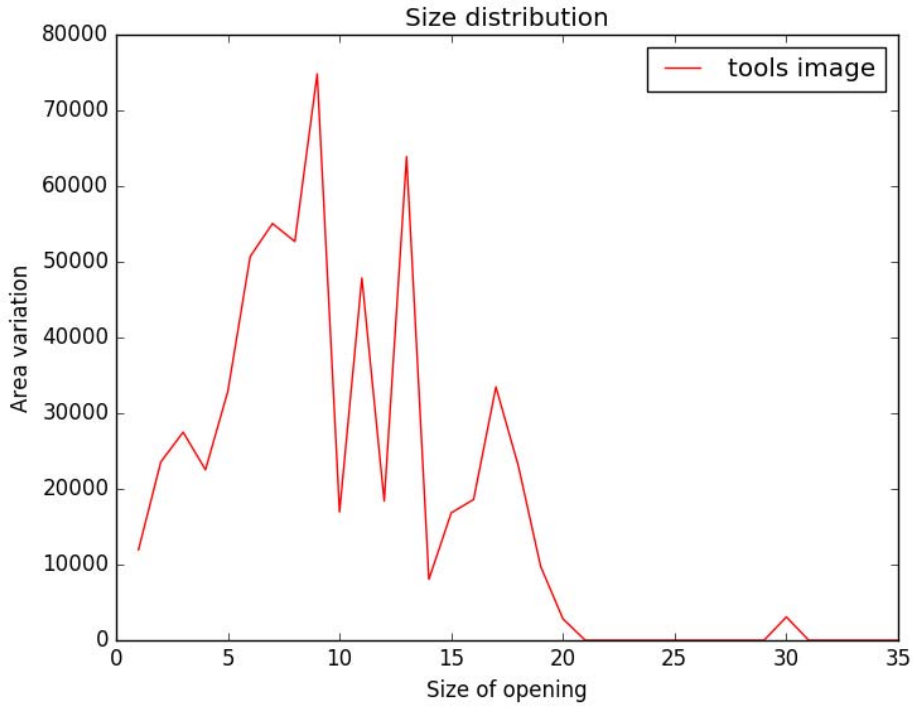
Several solutions are possible. However, the simplest and fastest one uses the top-hat transform applied to the initial image. It is compulsory to use an erosion-reconstruction opening otherwise the background would not be correctly taken into account. The size of the initial erosion must necessarily be larger than the maximal size of the objects contained in the *tools* image. A larger size does not modify significantly the result. The minimal size of this top-hat transform can be determined by computing the size distribution function which has been introduced in chapter 6, exercise 8 (operator **buildOpenSizeDistribution**). Let us compute this size distribution on the tools image loaded in im1:

```
>>> granuList = buildOpenSizeDistribution(im1, 35)
```

Then, we can plot it:

```
>>> import matplotlib.pyplot as plt
>>> xs = range(1, 36)
>>> ys = granuList
>>> err = plt.xlabel('Size of opening')
>>> err = plt.ylabel('Area variation')
>>> err = plt.title('Size distribution')
>>> err = plt.plot(xs, ys, label='tools image', color='red')
>>> err = plt.legend(loc='upper right')
>>> err = plt.show()
```

We can see that, for a size greater than 21, the area variation is negligible. There is no variation in the range [21, 29]. So, a top-hat by reconstruction of size 25 is convenient for obtaining a good background flattening.



Size distribution of the opening by reconstruction. No variation occurs between sizes 21 and 29. No object has been removed in this range.

Perform the following operation:

```
>>> buildOpen(im1, im2, 25)
>>> sub(im1, im2, im2)
```

Top-hat by reconstruction of size 25 of the tools image (the contrast has been enhanced).

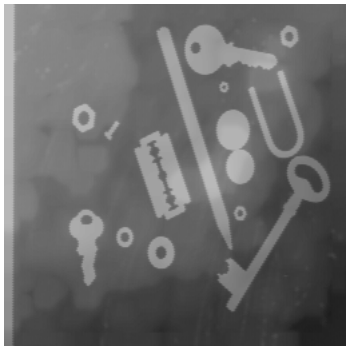
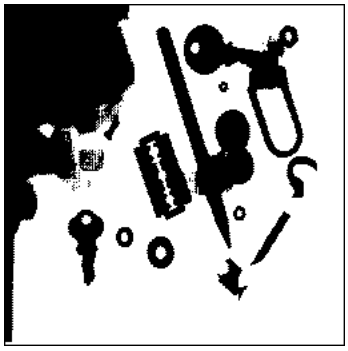
We can verify the efficiency of this operation by trying to threshold the initial image and the transformed one. For this, use the **dynamicThreshold** operator. Type:

```
>>> from mambaDisplay.extra import *
>>> dynamicThreshold(im1)
```

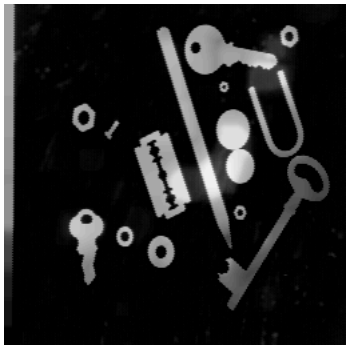
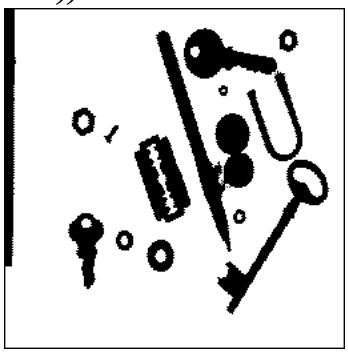
Try to threshold the tools image by hitting the <q> and <w> keys. Extracting the objects in the *tools* image by a simple threshold is not possible.
Try the same operator on the top-hat image:

```
>>> dynamicThreshold(im2)
```

Although the result is not perfect, it is nevertheless possible to extract the objects by thresholding.

No threshold value is convenient for extracting the objects of the tools image (the threshold level here is equal to (120, 255)).

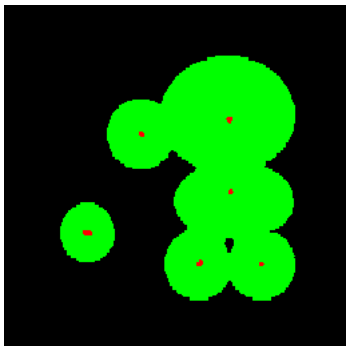
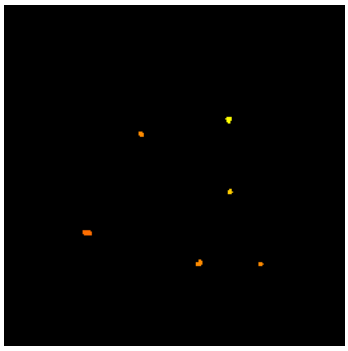



Threshold of the top-hat image at level (21, 255).

Exercise n° 3: Ultimate erosion of a (binary) set

1) Load the binary *disks* image in imbin1 and type:

```
>>> binaryUltimateErosion(imbin1, imbin2, im32_1)
>>> computeConnectivityNumber(imbin2)
6L
```

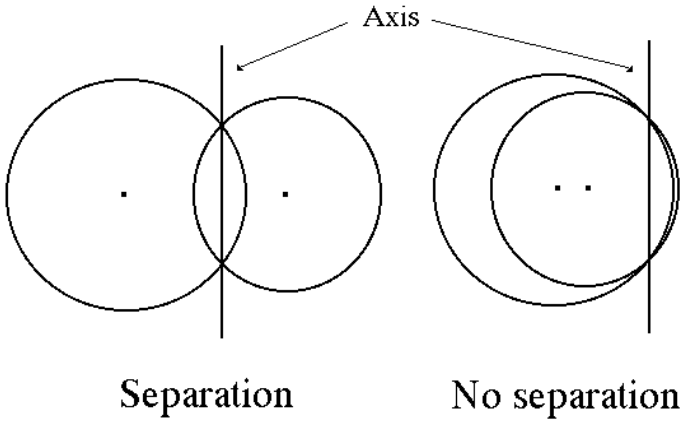



Ultimate erosion of the disks image (left, the result has been slightly dilated). The associated function (right image) labels each connected component with the radius of the corresponding disk (the rainbow palette has been used).

The number of disks is equal to 6. Note also that the **binaryUltimateErosion** operator requires two output images. The second one (which must be a 32-bit image) contains the connected components of the ultimate erosion labelled with the size of the erosion which

generated this connected component (size + 1 in fact to cope with the fact that some components may appear at the first iteration).

2) The separation by ultimate erosion provides satisfactory results only when the aggregate is composed of round particles which do not interpenetrate too much. In the simple case where the aggregate is made of disks, the ultimate erosion works if the radical axis (line passing through their two intersection points) of two connected disks separate their centers. When the particles are not perfectly circular but simply convex and more or less elongated, their separation by means of the ultimate erosion may be less efficient.



The separation of two disks by ultimate erosion fails when the disks are too intricated. The limit condition occurs when their centers are on the same side of the radical axis.

Exercise n° 4: Skeleton by maximal balls (binary case)

1) Let x be a point such that:

$$x \in (X \ominus nH) \text{ and } x \notin (X \ominus nH)_H$$

If x belongs to the eroded set of size n, x is by definition the center of a hexagon of size n included in X.

Assume that $x \in (X \ominus nH)_H$. x is then included in a hexagon of size 1 included in the eroded set $(X \ominus nH)$.

A size n dilation of this hexagon results in a hexagon of size n+1 included in X. Hence the hexagon of size n centered in x is covered by the hexagon of size n+1. Therefore it cannot be maximal. Conversely, let us assume that x, though not belonging to the open set$(X \ominus nH)_H$, is not the center of a maximal hexagon of size n. Then, it is included in a hexagon of size $m > n$covering the hexagon of size n centered in x. The erosion of this hexagon of size m by a hexagon of size n gives a hexagon of size $m - n \geq 1$which contains x. Then, $x \in (X \ominus nH)_H$, which is contradictory.

Therefore, we can state that a necessary and sufficient condition for a point x to be the center of a maximal hexagon of size n is that it belongs to the set:

$$(X \ominus nH)/(X \ominus nH)_H$$

2) According to what has just been stated, in order to obtain the skeleton S(X), it is sufficient to perform the preceding set difference for all the possible sizes of erosions, i.e.:

$$S(X) = \bigcup_{n=0}^{\infty} [(X \ominus nH)/(X \ominus nH)_H]$$

This operator is a residual transform with:

$\psi_i = \varepsilon_i$, hexagonal erosion of size i

$\zeta_i = \gamma(\varepsilon_i)$, hexagonal opening of size 1 of the erosion of size i

The associated function r defined on S(X) takes, at each point of S(X), the radius of the corresponding maximal ball (hexagon) centered at this point.
3) Let us start by proving that any point x of X belongs to the maximal ball.
Indeed, if this were not the case, then there would exist a ball centered in x and contained in X (the ball may be reduced to x itself). This ball is not included in any other ball included in X. (Otherwise this including ball would be itself either maximal, or included in a maximal ball). Therefore the ball centered in x is the maximal ball. x belongs to a maximal ball of X, which contradicts the hypothesis.

$$\forall x \in X,\ x \in \text{ maximal ball of X}$$

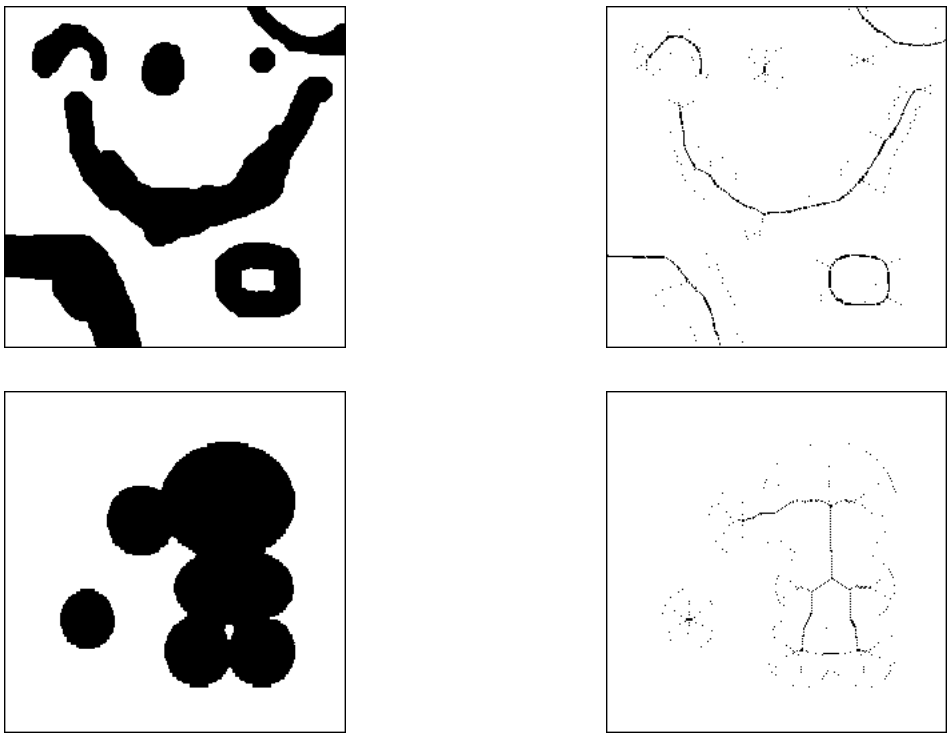
The union of all the maximal balls of X is then equal to X:

$$X = \bigcup_{x \in S(X)} H(x, r(x))$$

4) Use the *test_dist* and *disks* images loaded in imbin1:

```
>>> binarySkeletonByOpening(imbin1, imbin2, im32_1)
```

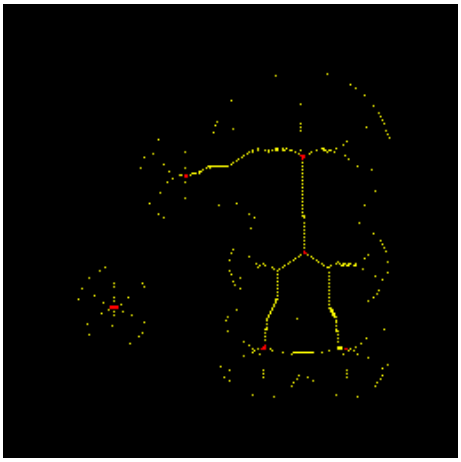
Here again, the quench function is put in the 32-bit image im32_1. Each pixel of the skeleton takes the value $r_i + 1$, where r_i is the radius of the maximal hexagon centered at i.



Initial images (left) and skeletons by maximal balls (right).

The ultimate erosion is always included in the skeleton by maximal balls. This can be verified by intersecting the ultimate erosion and the skeleton and by comparing this intersection with the ultimate erosion. The two images are identical:

```
>>> binarySkeletonByOpening(imbin1, imbin2, im32_1)
>>> binaryUltimateErosion(imbin1, imbin3, im32_2)
>>> logic(imbin2, imbin3, imbin2, "inf")
>>> compare(imbin2, imbin3, imbin4)
(-1, -1)
```



Skeleton (in yellow) of the disks image and its ultimate erosion (in red).

It can be easily verified that the ultimate erosion of a set is contained in the skeleton. Indeed, let x be a point of the ultimate erosion of X. Assume this point has appeared in the erosion of size N. $x \in (X \ominus nH)$ but $x \notin (X \ominus (n+1)H)$. This point x is at an infinite geodesic distance from $X \ominus (n+1)H$. In other words, the dilated set $[X \ominus (n+1)H] \oplus H$ does not encounter x. (since this dilated set represents the set of the points of the space which are at a geodesic distance smaller than or equal to 1 from $X \ominus (n+1)H$).
But:

$$[X \ominus (n+1)H] \oplus H = (X \ominus nH)_H$$

Then:

$$x \in (X \ominus nH)/(X \ominus nH)_H \Rightarrow x \in S(X)$$

5) The previous examples show the two main drawbacks of this operator: firstly, the skeleton is not connected (although there is no evidence that this skeleton should be connected) and, secondly, the use of hexagons instead of disks leads to a great number of points in the resulting image.

Exercise n° 5: Distance function and maxima

1) Let us prove the proposition. Let x be a point of the skeleton by hexagonal opening. x is the center of a maximal hexagon of size n. It results that dist(x) = n + 1.
Assume there exists a neighbor y of x such that dist(y) > dist(x). Then H_y^{n+1} (hexagon of center y of size n + 1) is included in X.
But $H_x^n \subset H_y^{n+1}$. Then, there exists an hexagon of size n + 1 that contains x and that is included in X, which contradicts the hypothesis according to which H_x^n is a maximal hexagon.
Then, $\forall$y, neighbor of x, dist(y) $\leq$ dist(x). x is then a local maximum of the distance function.
Conversely, let x be a local maximum of the hexagonal distance function. Let n = dist(x) - 1. H_x^n is the largest hexagon of center x included in X. Assume that H_x^n is not maximal, i.e.:

$$\exists H_y^{n'} \text{ such that } H_x^n \subset H_y^{n'} \text{ and } H_x^n \neq H_y^n$$

$H_y^{n'} \ominus H_x^n \neq \emptyset$ and it is a hexagon H_z^m. Moreover $m \geq 1$ because $H_y^{n'} \neq H_x^n$ and $x \in H_z^m$.
The erosion of H_z^m by a elementary hexagon is a hexagon H_z^{m-1} (possibly reduced to a single point) and $\forall y \in H_z^m/H_z^{m-1}$, there exists a neighbor of y in H_z^{m-1} (y belongs to the boundary of H_z^m and H_z^{m-1} is not empty). However, $x \notin H_z^m \ominus H$ (otherwise $H_x^{n+1} \subset X$, but H_x^n is the largest hexagon of center x included in X) and $H_z^m \ominus H \neq \emptyset$. then x has a neighbor y in H_z^{m-1}. It

results that $H_y^{n+1} \subset X$ and hence $dist(y) \geq n+2 = dist(x)+1$. This contradicts the assumption that x is a local maximum of the distance function. Then H_x^n is a maximal hexagon.

The local maxima of a function f can be easily extracted by remarking that, if a point x is a local maximum, the elementary hexagonal dilation of the function at this point is equal to the function itself. Indeed, all the values of f inside the hexagon centered at x are lower than or equal to f(x). Therefore, the dilation of f at point x, which is equal to the sup of all the values of the function inside H_x is equal to f(x). So, the local maxima correspond to the points x where $\delta_H(f(x)) - f(x) = 0$. Let us verify this equivalence on the *coffee* image loaded in imbin1. We compute the skeleton by maximal balls (result in imbin2):

```
>>> binarySkeletonByOpening(imbin1, imbin2, im32_1)
```

Then, we compute the distance function with **computeDistance**:

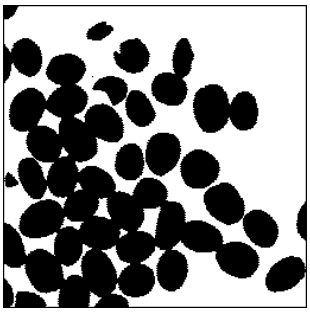
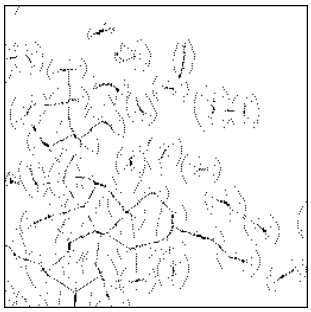
```
>>> computeDistance(imbin1, im32_2, edge=FILLED)
```

Note that the distance function is stored in a 32-bit image (im32_2) and that the edge has been set to **FILLED** in order to avoid edge effects with the particles touching the edges of the image. Then, we can extract the local maxima of the distance function as described above:

```
>>> dilate(im32_2, im32_1)
>>> sub(im32_1, im32_2, im32_1)
>>> threshold(im32_1, imbin3, 0, 0)
>>> logic(imbin1, imbin3, imbin3, "inf")
```

The last intersection with the initial image aims at removing the local maxima contained in the background. Finally, we can compare the two results:

```
>>> compare(imbin2, imbin3, imbin4)
(-1, -1)
```

coffee image and its skeleton by maximal hexagons.

It is also possible to obtain the skeleton by maximal balls (hexagons) of a set X by means of a top-hat transform applied to $dist_X$. Indeed, the threshold between n + 1 and 2^{32} - 1 of $dist_X$ is equal to the erosion of size n of X:

$$X \ominus nH = \{x : dist_X(x) > n\}$$

Performing an elementary opening of the distance function amounts to perform the same operation on every threshold of $dist_X$. Then, the top-hat transform, which corresponds to a set

difference $(X \ominus nH) \backslash (X \ominus nH)_H$ at each threshold of $dist_X - \gamma_H(dist_X)$, extracts the points of the skeleton by maximal balls of X. Load *coffee* in imbin1 and type:

```
>>> computeDistance(imbin1, im32_1, edge=FILLED)
>>> whiteTopHat(im32_1, im32_2, 1)
```

To finally obtain a binary set, threshold the 32-bit image im32_2:

```
>>> threshold(im32_2, imbin2, 1, 1)
```

2) An ultimate eroded set is made of connected components of the eroded set of size n that cannot be restored from the eroded set of size n+1, for all sizes n. But the eroded set of size n can be obtained by thresholding the distance function (see above). Then, the connected components of the ultimate erosion are, by definition, the maxima of the distance function. To verify this, load the *coffee* image in imbin1 and type:

```
>>> binaryUltimateErosion(imbin1, imbin2, im32_1)
```

Then:

```
>>> computeDistance(imbin1, im32_2, edge=FILLED)
>>> maxima(im32_2, imbin3)
```

Compare the two results:

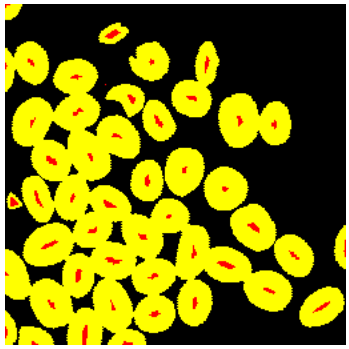
```
>>> compare(imbin2, imbin3, imbin4)
(-1, -1)
```

This operation is faster than the **binaryUltimateErosion** as it uses fast MAMBA operators. Moreover, the computation time is independent of the size of the set X. For this reason, this implementation of the ultimate erosion exists in MAMBA. It is named **ultimateErosion**.

In the *coffee* image, some grains are marked by more than one connected component of the ultimate erosion, which is not satisfying for counting and segmentation (particle separation) purposes. Using h-maxima can help to connect these markers.

```
>>> computeDistance(imbin1, im32_1, edge=FILLED)
>>> maxima(im32_1, imbin3, 2)
>>> maxima(im32_1, imbin2)
>>> computeConnectivityNumber(imbin2)
54L
>>> computeConnectivityNumber(imbin3)
48L
```

The number of connected components of the h-maxima, with h = 2 is less (48) than the number of initial maxima (54).



h-maxima of height 2 of the distance function of the coffee image.

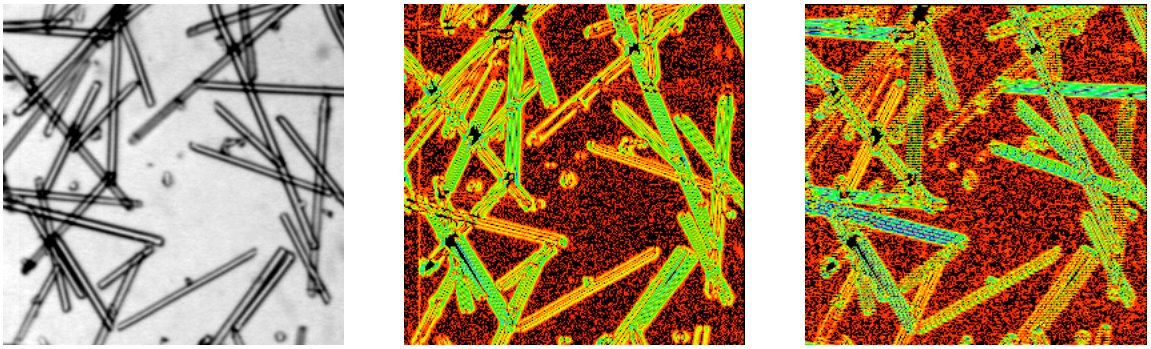
Exercise n° 6: Directional gradient

1) The operator defining elementary directional gradients of size 1 on the hexagonal grid is the following:

```
from mamba import *

def directionalGradient(imIn, imOut, d):
    """
    Basic directional gradient of size 1, computed on the hexagonal grid
    in direction 'd'.
    """

    imWrk = imageMb(imIn)
    linearDilate(imIn, imWrk, d)
    linearErode(imIn, imOut, d)
    sub(imWrk, imOut, imOut)
```

Load the *fiber1* image in im1 and type:

```
>>> directionalGradient(im1, im2, 2)
>>> directionalGradient(im1, im2, 1)
```



fiber1 image (left), directional gradient in direction 2 (middle) and directional gradient in direction 1 (right). The resulting images are enhanced and displayed with the rainbow palette.

2) The gradient modulus is equal to the sup of the directional gradients. The azimuth is coded in six directions and corresponds to the (first) direction where the directional gradient is maximum. The procedure **vectorGradient** is defined as following:

```
from mamba import *

def vectorGradient(imIn, imModul, imAzim):
  """
  Modulus and azimuth of the gradient of image 'imIn'. The modulus is
  stored in 'imModul' and the azimuth (6 directions are available) in
  image 'imAzim'.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  imWrk3 = imageMb(imIn)
  imWrk4 = imageMb(imIn, 1)
  imModul.reset()
  imAzim.reset()
  copy(imIn, imWrk1)
  for i in range(6):
     d = i+1
     directionalGradient(imWrk1, imWrk2, d)
     generateSupMask(imWrk2, imModul, imWrk4, True)
     convertByMask(imWrk4, imWrk3, 0, d)
     logic(imWrk2, imModul, imModul, "sup")
     logic(imWrk3, imAzim, imAzim, "sup")
```

The *seismic_section* image is loaded in the imA image, a 448x448 8-bit image. Two other images, imB and imC are also defined:

```
>>> imA = imageMb(448, 448)
>>> imB = imageMb(imA)
>>> imC = imageMb(imA)
```

Then, the **vectorGradient** operator is applied to imA. The modulus is put in imB and the azimuth is in imC:

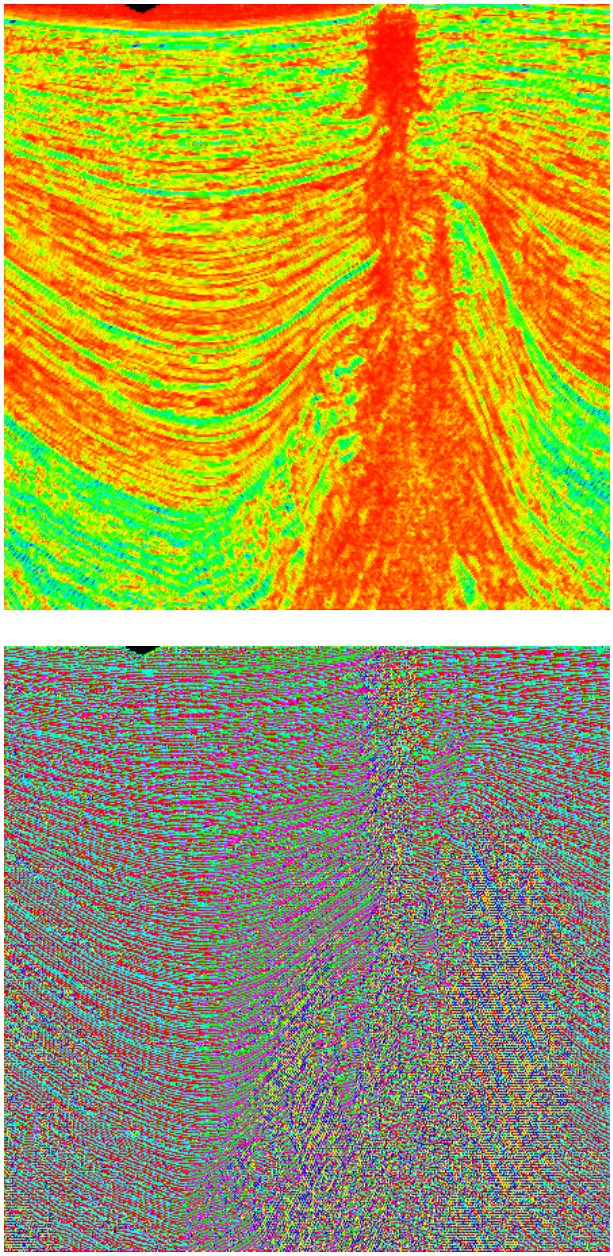
```
>>> vectorGradient(imA, imB, imC)
```

Finally, the modulus image is displayed with the rainbow palette. A new 6-color palette is defined for displaying this azimuth image:

```
>>> imB.show(palette="rainbow")
>>> dirpal = (0,0,0,255,0,0,255,255,0,0,255,0,0,255,255,0,0,255, 255,0,255)
>>> addPalette("direction palette", dirpal)
>>> imC.show(palette="direction palette")
```

3) The gradient modulus does not raise any problem. This modulus corresponds to the maximal directional gradient. On the contrary, the azimuth is not very accurate and robust. This is due to two facts. Firstly, the directional dilation and erosion allow to compute the maximum and minimum values on each directional segment but it does not indicate the

respective positions of these extrema. Therefore, the direction of the gradient vector is sometimes defined within about a 180° angle. Secondly, the retained direction corresponds to the first occurrence of the maximum value. But this maximum may appear for various directions.
Therefore, more sophisticated operators should be designed. This will be addressed in the next chapter.



Gradient modulus of the seismic_section image (top), gradient azimuth (bottom).

REFERENCES

[1] S.Beucher: Numerical residues
(**http://cmm.ensmp.fr/~beucher/publi/SB_NumRes_IVC_Rev1.pdf**)

The concept of residue is revisited in this paper with the description of classical residual transforms and the introduction of new numerical operators.

[2] S.Beucher: About a problem of definition of the geodesic erosion
(**http://cmm.ensmp.fr/~beucher/publi/GeodesicErode_eng.pdf**)
This note addresses a problem of definition of the geodesic erosion in the numerical case. It shows that the binary geodesic erosion does not correspond to the definition currently used in the numerical case. It also defines a new numerical geodesic erosion which really extends the binary operator.

Chapter 8

RESIDUES II

THINNINGS AND THICKENINGS

1 Introduction

The transformations developed in the following exercises are more sophisticated. According to our previous comparison, they are the true "machine-tools" of MM. They are at the meeting point of geodesy and homotopy. We have already handled geodesic transformations, therefore we shall only recall the notion of homotopy, and especially that of homotopic transformation.

2 Binary thinnings and thickenings, reminder

Let T = (T_1 ,T_2) be a two-phase structuring element. The hit-or-miss transformation of a set X by T is equal to:

$$X * T = (X \ominus \check{T}_1) \cap (X^c \ominus \check{T}_2)$$

The thickening of X by T is equal to:

$$X \odot T = X \cup (X * T)$$

and the thinning is defined by:

$$X \bigcirc T = X \backslash (X * T)$$

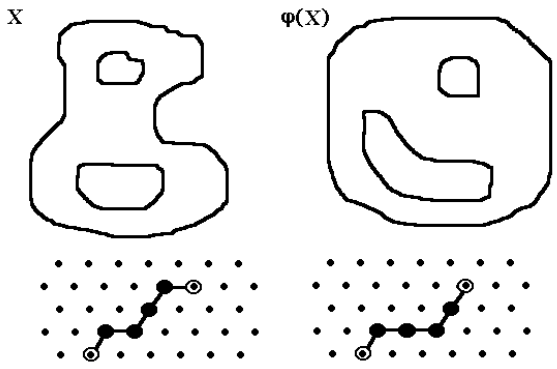
Thinning and thickening are both dual transformations:

$$(X^c \odot T)^c = [X^c \cup (X^c * T)]^c = X \cap (X^c * T)^c = X \setminus (X * T') = X \bigcirc T'$$

where T' = (T_2 ,T_1) is the transposed double structuring element.

3 Homotopy

Two paths of a set X are homotopic if it is possible to superpose one another by a sequence of continuous deformations. "Continuous" means without cut and that all intermediary paths are included in X.



Homotopic transform ($\varphi(X)$ and X can be superposed without tear) and homotopic paths (the second one can be obtained by a continuous displacement of the first one).

By extension, a transformation Ψ which preserves homotopy is said to be homotopic. Intuitively, a homotopic transformation Y = Ψ(X) transforms the set X into a set Y that can be superposed on X by a continuous deformation.
When the set X is digitized according to a (square or hexagonal) grid, matching paths is easy since any path can be defined as the concatenation of elementary edges. A homotopic transformation does not break any paths.

4 Greytone thinnings and thickenings

Let m(x) = $\sup_{y \in T_{2x}} f(y)$ and M(x) = $\inf_{y \in T_{1x}} f(y)$.
The thickening of f by T = (T_1 ,T_2) is defined by :

$$(f \odot T)(x) = M(x), \text{ iff } m(x) \le f(x) < M(x)$$
$$(f \odot T)(x) = f(x), \text{ if not.}$$

The thinning of f by T = (T_1 ,T_2) is defined by:

$$(f \bigcirc T)(x) = m(x), \text{ iff } m(x) < f(x) \le M(x)$$
$$(f \bigcirc T)(x) = f(x), \text{ if not.}$$

5 Zone of influence, SKIZ

Let us define the zone of influence of a connected component of a set. Let X be a set composed of several connected sub-sets:

$$X = \bigcup_i X_i$$

The zone of influence $Z(X_i)$ of the connected component X_i, is the set of the points closer (according to the distance d) to X_i than to any other connected component of X.

$$x \in Z(X_i) \Leftrightarrow d(x, X_i) < d(x, X_j), \forall j \neq i$$

The points of the space which do not belong to any influence zone are the points of the skeleton by influence zones, or SKIZ.

EXERCISES

Although all the transformations introduced or used in the following exercises can be defined on the hexagonal or square grids, for sake of simplicity, the hexagonal case will be used in most cases.

Exercise n° 1

The most interesting structuring elements, T = (T_1,T_2) are those defined on the elementary ball (hexagon or square):

$$T_1 \subset H,\ T_2 \subset H$$

We can even write: $T_1 \cap T_2 = \varnothing$. Indeed, T_1 and T_2 must have no common point if we want X * T to be different from an empty set.
1) Test the various operators available in MAMBA: **hitOrMiss**, **thin**, **thick**, **rotatingThin**, **rotatingThick**, **fullThin** and **fullThick**. These operators use the **doubleStructuringElement** class which allows to define double structuring elements T = (T_1,T_2). A tool named

hitormissPatternSelector included in the **mambaDisplay.extra** module can be used to define easily any double structuring element on the elementary square or hexagon. Try this tool and the various methods associated with these double structuring elements.
2) Apply these operators to binary images. In particular, perform the following operations:
- delete the white and black isolated points in the *noise* image.
- contour a set in only one transformation.

Exercise n° 2: Geodesic thickenings and thinnings

Let X be a set included in Z. The geodesic thickening of X by a structuring element T included in the elementary hexagon can be defined by:

$$(X \odot T)_Z = (X \odot T) \cap Z$$

1) Test this transformation (with the operators **geodesicThick**, **rotatingGeodesicThick** and **fullGeodesicThick** available in MAMBA).
2) Since thinning is the dual operation of thickening, geodesic thinning is defined by:

$$(X \bigcirc T)_Z = Z \backslash [(Z \backslash X) \bigcirc T']$$

where T' is the transposed double structuring element. Test also this transformation with the **geodesicThin**, **rotatingGeodesicThin** and **fullGeodesicThin** operators.

Exercise n° 3: Greytone thinnings and thickenings

1) Use the definition given above to program with MAMBA the greytone thinning and thickening (limit your operators to the greyscale images and the hexagonal grid).
2) Use the same template as the one used in the binary case to program the greytone rotating and full thinnings and thickenings.

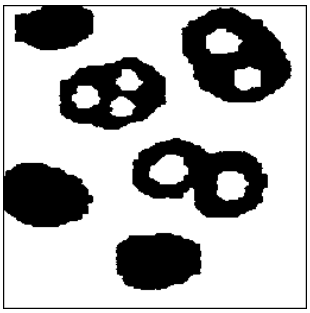
Exercise n° 4

1) Find all possible configurations (up to a rotation) for the neighborhood of a point in the hexagonal grid (there are 14 possibilities).
2) Prove that the relation of homotopy for two paths C_1 and C_2 with same origin and same extremity included in a set X is a relation of equivalence.
3) From this, deduce which of these configurations generate an homotopic thinning.

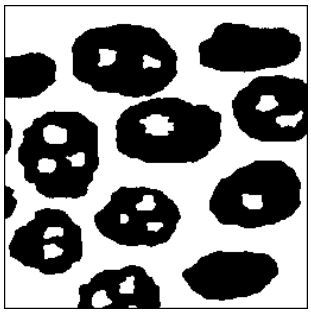
Exercise n° 5

1) Homotopic full thinnings with the double structuring elements L, M and D are available in MAMBA and are named **thinL**, **thinM** and **thinD**). Test them and describe their main properties and interests (use the *holes* and *gruyere* images).

L M D



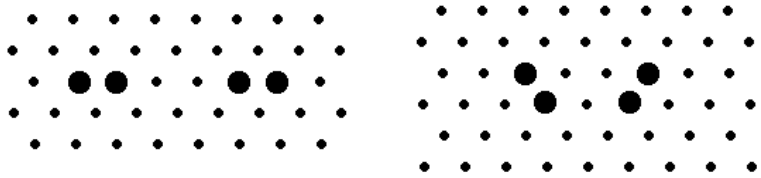
holes



gruyere

2) Find and test the transformations in the MAMBA library allowing to extract the characteristic points of these homotopic transforms:
- extremities
- n-uple points
- isolated points

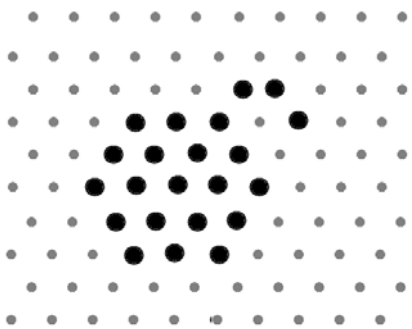
3) The direction of rotation and the starting orientation of the structuring elements used for the different thinnings are totally arbitrary. Consequently, more or less important variations occur in the resulting transformed set. These variations may even generate artefacts.



Generate the two following images above and apply to them a L thickening (also called L skeleton) and check the result. Is it possible to improve the algorithm?

Exercise n° 6

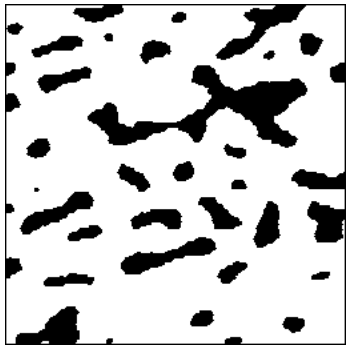
The purpose of this exercise is the analysis of the biases introduced when using the **thinD** operator to compute the geodesic center of a simply connected set and the design of some possible enhancements of the algorithm.
Each point of a set X that is simply connected (without holes) can be valued by the greatest geodesic distance between this point and any other point of X. Thus, we define a function on X, called geodesic distance function and denoted d. The maxima of this function define the extremities of X and it minimum (unique) is the geodesic center of the set.
1) Consider the following digital set X (on the hexagonal grid). Calculate its geodesic distance function.



2) Prove that, if x and y are two adjacent points of X, $|d(x) - d(y)| < 2$.
3) The D double structuring element (see previous exercise), used in homotopic thinnings, introduces some biases because it is used in rotative thinnings. We could expect better results regarding the position of the geodesic center if the various configurations composing this structuring element were used in parallel. However, we know that this transformation (inf of thinnings) is not homotopic. In order to overcome this problem, we can analyse in detail the different configurations of D. This structuring element can be splitted into three structuring elements:

	1	1		1	1		1	0
1	1	0	0	1	0	0	1	0
	0	0		0	0		0	0
	D_3			D_2			D_1	

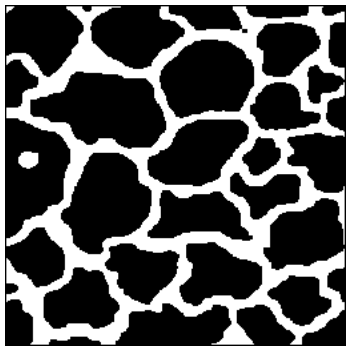
Start with D_3 configuration and try to find the possible values taken by d on the neighbor points when this value is equal to n on the central point. Deduce from this analysis which configurations could cause problems and propose a procedure to avoid these problems.
4) Perform the same analysis for the D_1 and D_2 double structuring elements and propose also solutions to the detected issues.
5) Use the previous results to design an algorithm generating the geodesic centers of a set composed of simply connected components. Compare this procedure with the **thinD** operator.
Use the *metal1* image.



metal1

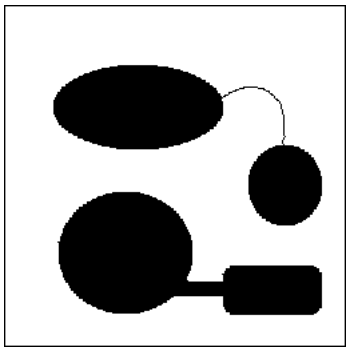
Exercise n° 7

1) Perform the SKIZ of the binary *alumine* image using the **computeSKIZ** operator available in MAMBA. Is the SKIZ an homotopic transform?

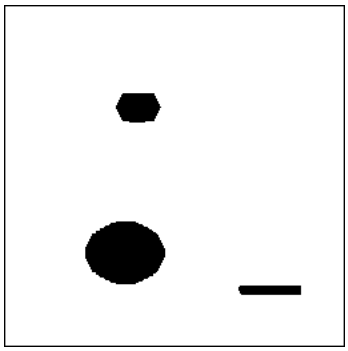


alumine

2) Test the geodesic thickening on the *test_thick* and *test_thick_marker* images. Perform a full geodesic thickening, with **fullGeodesicThick**, of the *test_thick_marker* image using the *test_thick image* as geodesic space. Try the transformation with the L and D double structuring elements. Comment the results.



test_thick



test_thick_marker

3) The geodesic zones of influence $Z(Y_i)$ of the connected components Y_i inside a set X $\left(Y = \bigcup_i Y_i\right)$ considered as a geodesic space are composed of the x of X closer (according to the geodesic distance d_X) to Y_i than to any other connected component of Y:

$$x \in Z(Y_i) \Leftrightarrow d_X(x, Y_i) < d_X(x, Y_j), \forall j \neq i$$

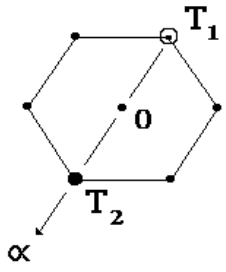
Use the **geodesicSKIZ** operator for obtaining the geodesic SKIZ of *test_thick_marker* inside the geodesic space defined by *test_thick*.

Exercise n° 8

We already defined some directional gradients by using linear dilations and erosions (chapter 7, exercise n° 6). However, these gradients are known to be biased, especially for the computation of the azimuth. In order to define more accurate operators, a new gradient can be defined in the direction α by means of thickenings and thinnings. The directional gradient in the direction α is defined by:

$$g_\alpha(f) = (f \odot T_\alpha) - (f \bigcirc T_\alpha)$$

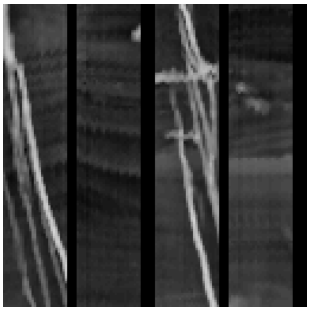
where T_1 and T_2 form the two-phase structuring element $T_\alpha = (T_1, T_2)_\alpha$.



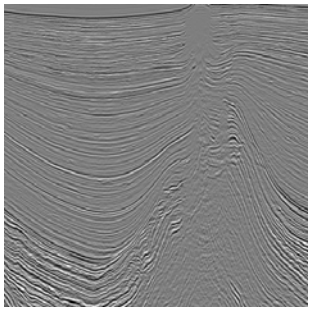
Double structuring element used in the directional gradient by thinning and thickening.

The gradient modulus is equal to the maximal directional gradient. But this maximal directional gradient may occur in several directions simultaneously. Then, we have to take into account all these maximal directions to compute a correct azimuth. The number of directions where the directional gradient may take its maximum value is limited. Indeed, three directions at most can be extracted in the hexagonal grid, four in the square grid.

1) Find all possible configurations of maximal directional gradients (up to a rotation, and in the hexagonal grid). Their number is equal to 5.
2) In case of non adjacent directions, the gradient is considered to be 0. In case of adjacent directions, a unique direction is selected, which is the mean direction of all present directions. Which configurations do we obtain (up to a rotation)? Their number is equal to 3. What is the resulting effect and how can it be avoided?



petrole

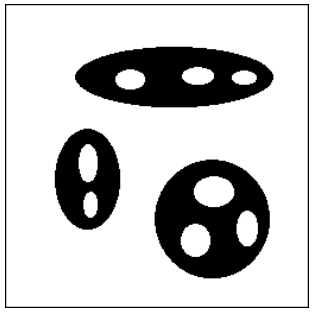


seismic_section

3) Program this new directional gradient based on thinnings and thickenings and apply it to the *petrole* and *seismic_section* images.

Exercise n° 9: Classification of particles

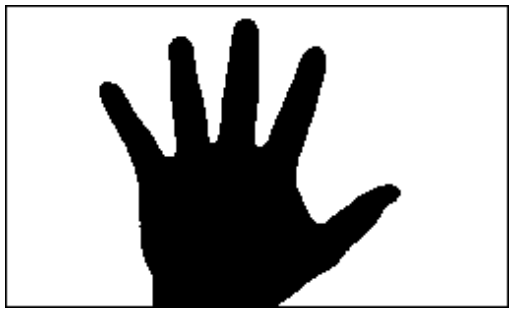
This problem is not related to a particular study but underlies numerous applications. It consists in separating particles or objects according to the number of holes they contain. This kind of problem is frequently addressed in industrial vision (discrimination of objects according to the number of their perforations), in automated character recognition (character classification), in bio-medical imagery (separation of cells with or without nucleus), etc. This separation is in fact a direct application of the notion of homotopy.
Using the *gruyere, holes, noise and two_three_holes* images, solve the following problems:
1) In the image, separate the particles without hole from the particles with holes.
2) Among the particles with holes, separate those with one and only one hole from those with more than one hole.
3) Among the particles with more than one hole, extract those with two holes only from those with three and more.
4) Is it possible to design algorithms that separate objects according to their number of holes, whatever this number?
5) Is the transform $\Psi_n(X)$ which extracts the particles of a set X with at least n holes an algebraic opening? And a size distribution?



two_three_holes

Exercise n° 10: Extremities of particles

This exercise is the continuation of exercise n° 6. It uses the geodesic distance function and, in particular, the geodesic center to extract the extremities of simply connected particles (without holes).
1) Compute the geodesic centers of a set X composed of simply connected components. Use the procedure defined in exercise n° 6 and apply it to the *eutectic* and *hand* images. You can also use the **thinD** operator to obtained the centroids instead of the true geodesic centers.
2) Compute the geodesic distance of the set X\{geodesic centers}, X itself being the geodesic space (the successive levels of this function correspond to successive geodesic erosions).
3) Extract the extremities of the particles (they correspond to the maxima of the geodesic distance). Compare the results obtained when using the true geodesic centers and the centroids.



hand

Exercise n° 11: Dislocations in eutectics

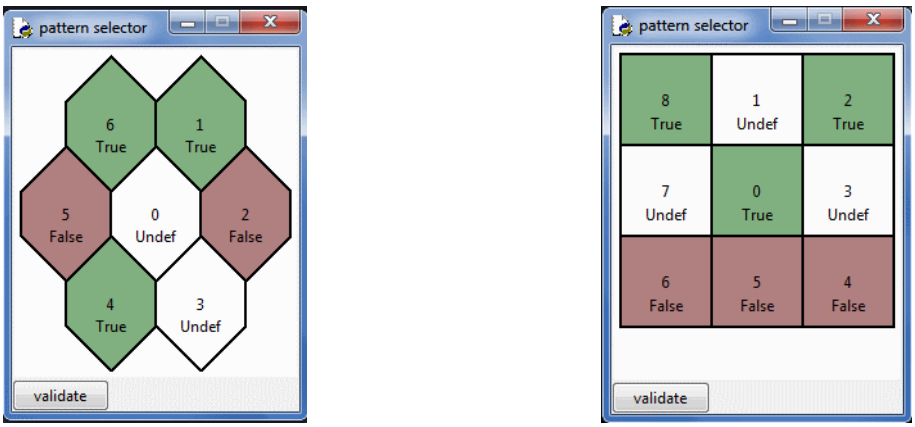
The *eutectic* image represents a lamellar eutectic material. This structure is characterized by a two-phase lamination, with here and there undesirable defects characterized by discontinuous lamellae. These discontinuities due to dislocations in the crystalline assembly of the material jeopardize its mechanical properties. The loss in resistance depends, among other things, on the length of these dislocations. Moreover, the size of the cells delimited by these dislocations is also important. You then have to find a sequence of operations for materializing these dislocations so as to be able to measure their length.
1) Use the previous exercise to extract the farthest extremities of the lamellae.
2) Try next to connect judiciously these extremities, so as to obtain an arc materializing each dislocation. Try also to extract the cells delimited by the dislocations.

eutectic

SOLUTIONS

Exercise n° 1

1) The **hitormissPatternSelector** is a tool which helps you to design easily double structuring elements. Just click once (for setting the point to one) or twice (for setting it to 0) in the corresponding window, then validate when ready. The operator will return the corresponding structuring element.



hitormissPatternSelector for the hexagonal and square grids. The pixels belonging to T_1 are in green, those belonging to T_2 are in red.

```
>>> from mambaDisplay.extra import *
>>> dse = hitormissPatternSelector()
>>> dse
DoubleStructuringElement(structuringElement([2, 5], mamba.HEXAGONAL),
structuringElement([1, 4, 6], mamba.HEXAGONAL))
```

Note that the first structuring element corresponds to T_2. Each structuring element can be extracted with the **getStructuringElement(ground)** method:

```
>>> s1 = dse.getStructuringElement(1)
>>> s1
structuringElement([1, 4, 6], mamba.HEXAGONAL)
>>> s2 = dse.getStructuringElement(0)
>>> s2
structuringElement([2, 5], mamba.HEXAGONAL)
```

The same tool also exists for the square grid:

```
>>> dse = hitormissPatternSelector(SQUARE)
>>> dse
doubleStructuringElement(structuringElement([4, 5, 6], mamba.SQUARE),
structuringElement([0, 2, 8], mamba.SQUARE))
```

rotatingThin and **rotatingThick** perform respectively successive thinnings and thickenings by rotating the double structuring element after each step. The operation stops when a complete rotation has been performed. The rotation is always clockwise.

fullThin and **fullThick** perform complete thinnings and thickenings by using successive rotations until idempotence.

2) Examples:
- Deletion of isolated points
Define the following double structuring element (with **hitormissPatternSelector**):

```
   0   0
 0   1   0
   0   0
```

```
>>> dse = hitormissPatternSelector()
>>> dse
doubleStructuringElement(structuringElement([1, 2, 3, 4, 5, 6], mamba.HEXAGONAL),
structuringElement([0], mamba.HEXAGONAL))
```

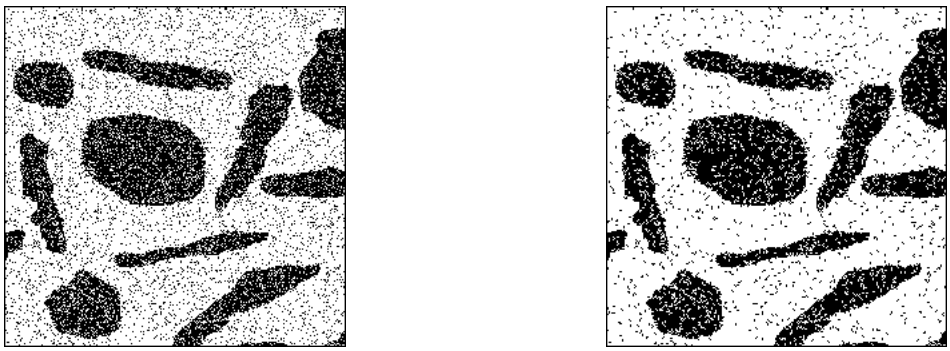
Then load the *noise* image into imbin1 and proceed as follows:

```
>>> thin(imbin1, imbin2, dse)
```

Similarly, define the following double structuring element and apply it to the previous resulting image:

```
   1   1
 1   0   1
   1   1
```

```
>>> dse = hitormissPatternSelector()
>>> dse
doubleStructuringElement(structuringElement([0], mamba.HEXAGONAL),
structuringElement([1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))
>>> thick(imbin2, imbin2, dse)
```



Removing black and white isolated points in the noise image.

- Contour detection
The contour C(X) of a set X corresponds to the points of X which do not belong to the eroded set:

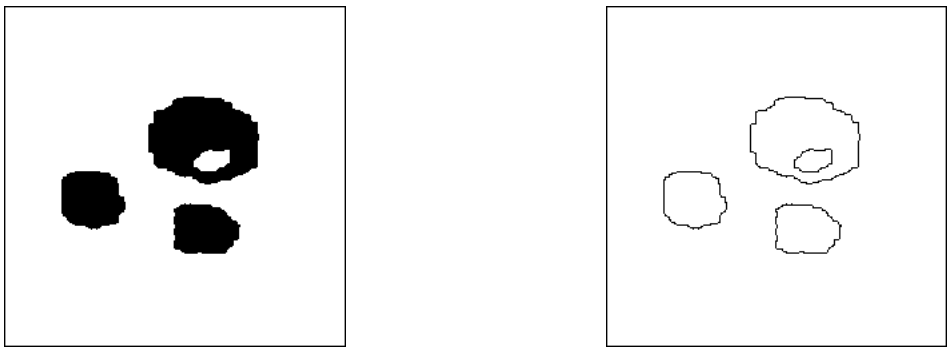
$$C(X) = X \backslash (X \ominus H) = X \cap (X \ominus H)^c$$

This can also be written:

$$C(X) = X \bigcirc T \text{ , with } T = (\varnothing, H)$$

Load the *objects* image in imbin1, define the corresponding double stucturing element and perform the thinning:

```
>>> dse = doubleStructuringElement(structuringElement([], mamba.HEXAGONAL),
structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))
>>> thin(imbin1, imbin2, dse)
```



Contour of a set obtained by a thinning operator.

Exercise n° 2: Geodesic thickenings and thinnings

1) **geodesicThick**, **rotatingGeodesicThick** and **fullGeodesicThick** are defined in MAMBA. Define the following double structuring element (T_1 is composed of the pixel in direction 1 and T_2 contains the pixel in the opposite direction):

```
>>> dse = doubleStructuringElement(structuringElement([4], mamba.HEXAGONAL),
structuringElement([1], mamba.HEXAGONAL))
```

Then, load *objects* in imbin1 and perform a size 30 dilation of the image in imbin2 which will be used as the geodesic space:

```
>>> dilate(imbin1, imbin2, 30)
```

Geodesic full thickening of the object image (in green) inside the dilated image (in yellow). Some points (in red) do not belong to the thickening because their neighborhood configuration does fit the double structuring elements used.

Then, perform a full geodesic thickening (until idempotence). Due to the configuration used, some black points appear in the final result.

```
>>> fullGeodesicThick(imbin1, imbin2, imbin3, dse)
```

2) A geodesic thinning is achieved by using a thickening by the dual structuring element, so as to avoid the problems due to edge effects already mentioned about geodesic erosions.

Exercise n° 3: Greytone thinnings and thickenings

In the definition of the greytone thinnings and thickenings, the operations $\sup_{y \in T_{2x}} f(y)$ and $\inf_{y \in T_{1x}} f(y)$ are respectively the dilation by T_2 and the erosion by T_1. A grey thinning or thickening is obtained by testing the inequalities between the dilation, the erosion and the initial image. This can be done with the **generateSupMask** operator. The pixels where the inequalities are fulfilled are given the value of the erosion (for the thickening) or the dilation (for the thinning). This is achieved with the **convertByMask** and **logic** operators. These operators (limited to greyscale images on the hexagonal grid) are defined as follows:

```
def greyThin(imIn, imOut, dse):
    """
    Grey thinning of the 'imIn' image by the double structuring element 'dse'.
    For sake of simplicity, this operator is defined only on the hexagonal grid and
    with greyscale (8-bit) images.
    """

    imDil = imageMb(imIn)
    imEro = imageMb(imIn)
    mask1 = imageMb(imIn, 1)
    mask2 = imageMb(imIn, 1)
    gmask = imageMb(imIn)
    dilate(imIn, imDil, se=dse.getStructuringElement(0))
    erode(imIn, imEro, se=dse.getStructuringElement(1))
    generateSupMask(imIn, imDil, mask1, True)
    generateSupMask(imEro, imIn, mask2, False)
    logic(mask1, mask2, mask1, "inf")
    convertByMask(mask1, gmask, 0, 255)
    logic(gmask, imDil, imDil, "inf")
    negate(gmask, gmask)
    logic(gmask, imIn, imOut, "inf")
    logic(imDil, imOut, imOut, "sup")

def greyThick(imIn, imOut, dse):
    """
    Grey thickening of the 'imIn' image by the double structuring element 'dse'.
    For sake of simplicity, this operator is defined only on the hexagonal grid and
    with greyscale (8-bit) images.
    """

    imDil = imageMb(imIn)
    imEro = imageMb(imIn)
    mask1 = imageMb(imIn, 1)
```

```
  mask2 = imageMb(imIn, 1)
  gmask = imageMb(imIn)
  dilate(imIn, imDil, se=dse.getStructuringElement(0))
  erode(imIn, imEro, se=dse.getStructuringElement(1))
  generateSupMask(imIn, imDil, mask1, False)
  generateSupMask(imEro, imIn, mask2, True)
  logic(mask1, mask2, mask1, "inf")
  convertByMask(mask1, gmask, 0, 255)
  logic(gmask, imEro, imEro, "inf")
  negate(gmask, gmask)
  logic(gmask, imIn, imOut, "inf")
  logic(imEro, imOut, imOut, "sup")
```

Rotating grey thinnings and thickenings are designed as follows (the double structuring element is rotated clockwise):

```
def rotatingGreyThin(imIn, imOut, dse):
  """
  Performs a complete rotation of  grey thinnings , the initial 'dse' double
  structuring element being turned one step clockwise after each thinning.
  At each rotation step, the previous result is used as input for the next
  thinning (chained thinnings). This operator works only on the hexagonal grid
  and on 8-bit images.
  """

  copy(imIn, imOut)
  for i in range(6):
    greyThin(imOut, imOut, dse)
    dse = dse.rotate()

def rotatingGreyThick(imIn, imOut, dse):
  """
  Performs a complete rotation of grey thickenings, the initial 'dse' double
  structuring element being turned one step clockwise after each thickening.
  At each rotation step, the previous result is used as input for the next
  thickening (chained thickenings). This operator works only with 8-bit images
  and on the hexagonal grid.
  """

  copy(imIn, imOut)
  for d in range(6):
    greyThick(imOut, imOut, dse)
    dse = dse.rotate()
```

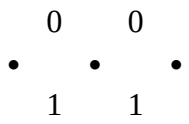
Finally, full grey thinnings and thickenings are programmed with the following operators:

```
def fullGreyThin(imIn, imOut, dse):
  """
  Performs a complete grey thinning of 'imIn' with the successive rotations of 'dse'
  (until idempotence) and puts the result in 'imOut'.
  Works with greyscale images and on the hexagonal grid.
  """

  copy(imIn, imOut)
```

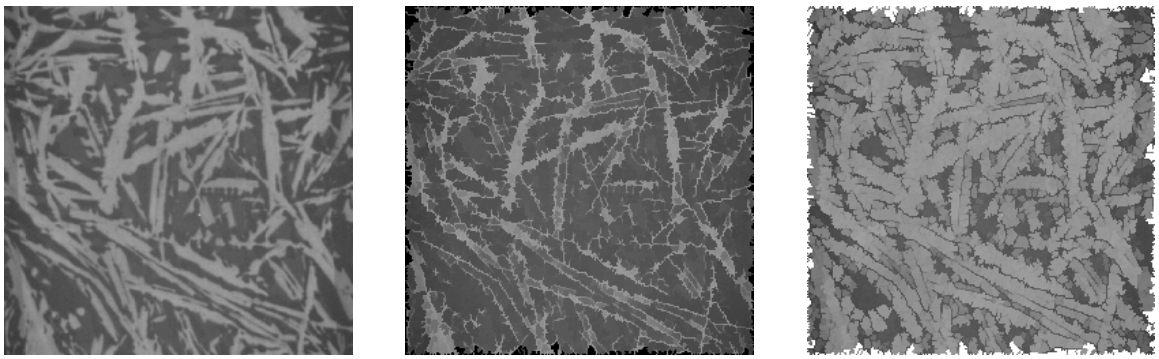
```
    v1 = computeVolume(imOut)
    v2 = 0
    while v1 != v2:
        v2 = v1
        rotatingGreyThin(imOut, imOut, dse)
        v1 = computeVolume(imOut)

def fullGreyThick(imIn, imOut, dse):
    """
    Performs a complete grey thickening of 'imIn' with the successive rotations of 'dse'
    (until idempotence) and puts the result in 'imOut'.
    Works with greyscale images and on the hexagonal grid.
    """

    copy(imIn, imOut)
    v1 = computeVolume(imOut)
    v2 = 0
    while v1 != v2:
        v2 = v1
        rotatingGreyThick(imOut, imOut, dse)
        v1 = computeVolume(imOut)
```

You can test these operators with some double structuring elements already existing in the MAMBA library. Use for instance the **hexagonalL** double structuring element on the *ferrite1* image, loaded in im1:

```
>>> hexagonalL
doubleStructuringElement(structuringElement([1, 6], mamba.HEXAGONAL),
structuringElement([3, 4], mamba.HEXAGONAL))
```




```
>>> fullGreyThick(im1, im2, hexagonalL)
>>> fullGreyThick(im1, im2, hexagonalL)
```

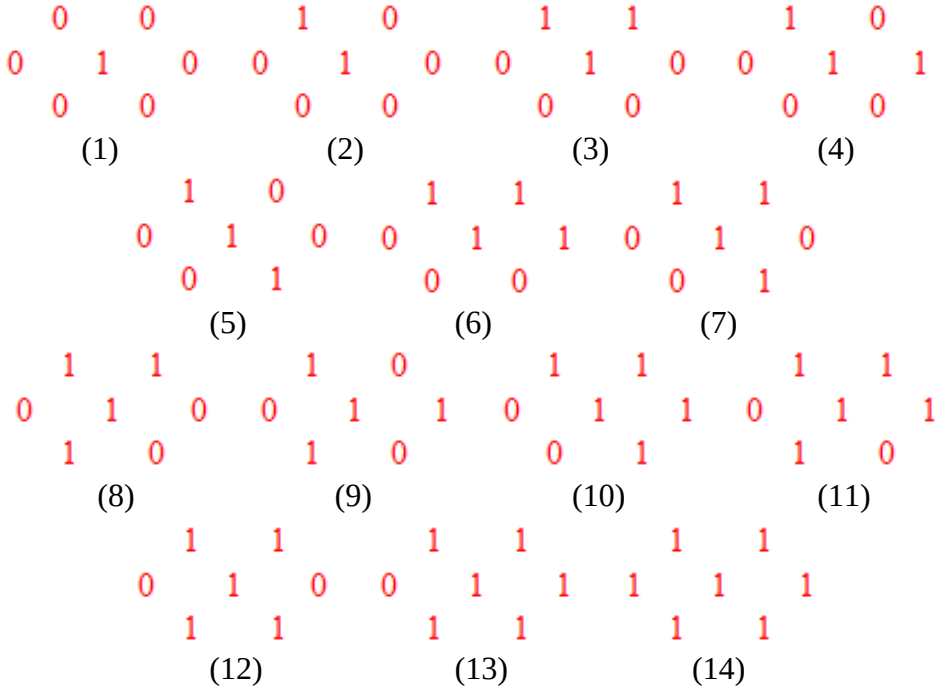


ferrite1 image (left), its full thinning with the hexagonalL structuring element (middle) and its full thickening with the same structuring element (right).

Most of the time, full thinning and thickening operators are not very useful as they produced results which are not understandable easily. Moreover, they are often slow as many iterations are needed before reaching idempotence.

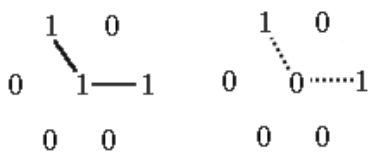
Exercise n° 4

1) Below are displayed the 14 configurations (the central point is assumed to be equal to 1):



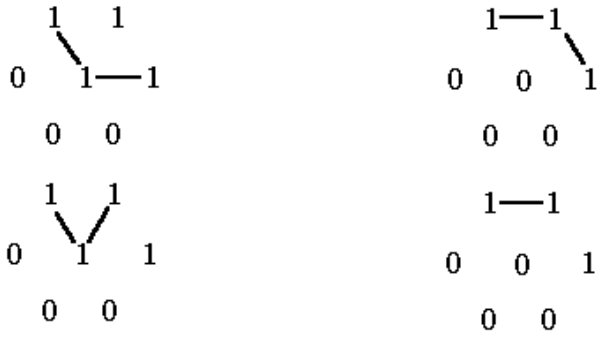
Possible neighborhood configurations of a point on the hexagonal grid (up to a rotation).

2) Obvious. The homotopy relation is an equivalence relation (that is a reflexive, symmetric and transitive relation).
3) The thinning by one of the 14 possible configurations replaces the central point 1 by 0. To extract the configurations which generate homotopic thinnings, we must verify that:
- any path which goes by the central point (this central point is not an extremity of the path) can be replaced, after thinning, by an equivalent path in the hexagonal neighborhood.
- the points which belong to the neighborhood are not modified by the thinning in such a way that the alternate paths could also be broken.
This procedure can be illustrated in the case of configuration n° 4, n° 6 and n° 10 for instance.
When thinning with the configuration n° 4, the only initial path is cut whilst the points on the neighborhood are not suppressed. This configuration does not produce an homotopic thinning.

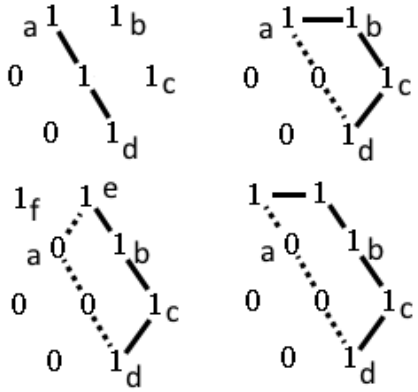


Thinning with configuration n° 4 removes the central point whilst the neighborhood points are not suppressed. The initial path being broken, the thinning is not homotopic.

Thinning with the configuration n° 6 does not break the initial paths. Therefore, this configuration produces an homotopic thinning.



Initial paths before thinning (left) and equivalent paths after thinning (right). The central point has been removed but a path joining the extremities of the initial path can still be defined because the neighborhood points remain after thinning.



Examples of possible path modifications with configuration n° 10. Points ((b) and (c) cannot be suppressed, allowing thus alternate paths when the central point is removed.

Analysing configuration n° 10 is more complex. We can notice first that, in this configuration, points (b) and (c) are never removed when the central point is replaced by 0, because their neighborhood does not match the double structuring element configuration (see figure above). However, points (a) and/or (d) may be removed. If these points are not removed, the path joining them through the central point is replaced by the path passing through (b) and (c). If, on the contrary, (a) is removed, this means that its neighborhood corresponds to the configuration n° 10. However, point (e) is not suppressed by thinning. So, any path starting from (e) and passing through (a) and the central point can be replaced by the path drawn in the figure (bottom right). It is the same if the initial path starts from point (f), when this later point is not suppressed by the thinning. If point (f) is also removed, the point which is just over it (point in direction 1 between (e) and (f)) remains unchanged and the initial path starting from this point, passing through (f) and joining (d) through the central point can still be replaced by a path passing through (e), (b) and (c) without cut.
Then, configuration n° 10 always produces an homotopic thinning.

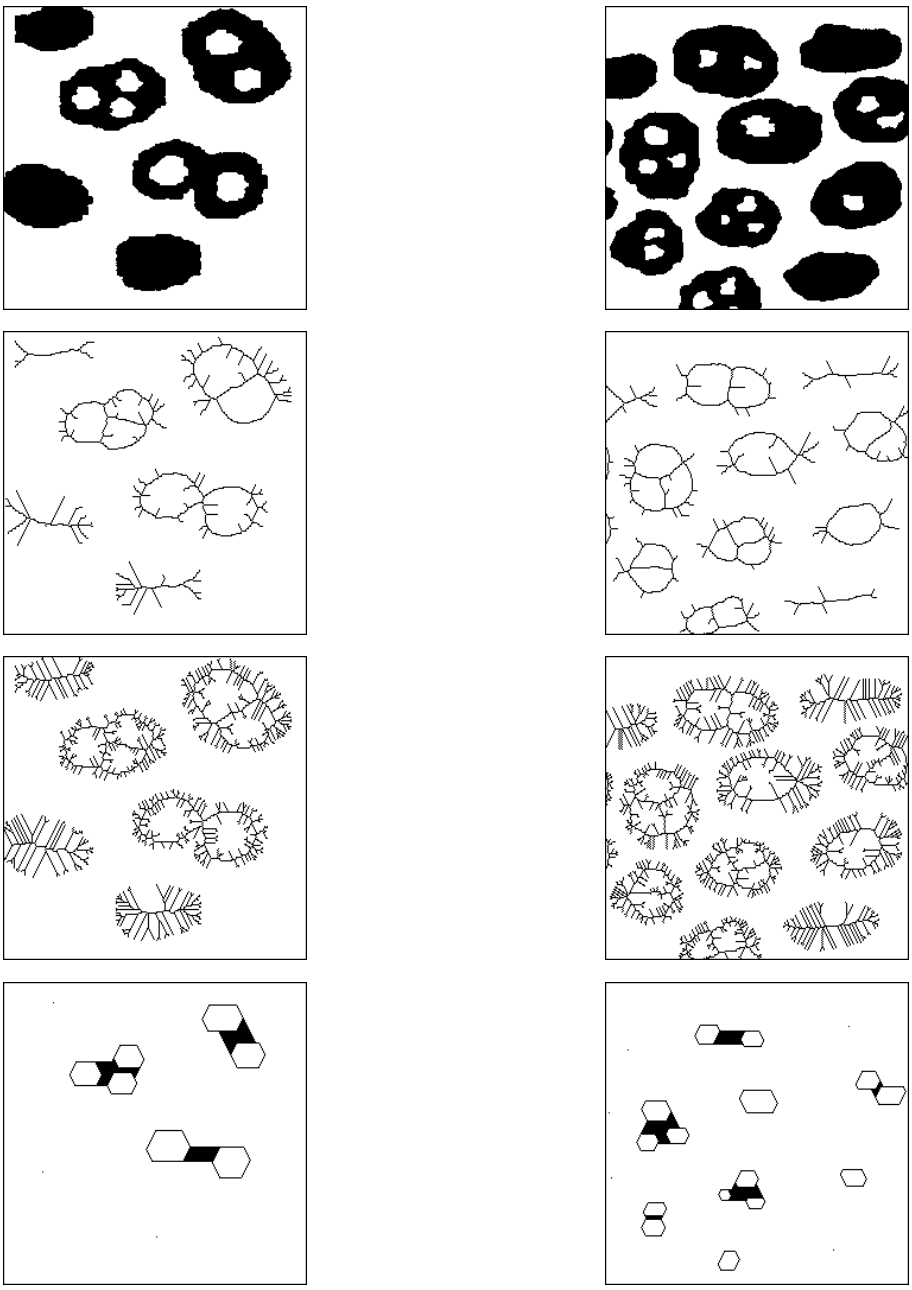
When performing the same analysis with the other configurations, we can see that only those where both T_1 and T_2 contain only one connected component generate homotopic thinning, namely: configurations 2, 3, 6, 10 and 13. The same configurations where the central point is equal to 0 generate homotopic thickenings.

Exercise n° 5

1) The L, M and D structuring elements are combinations of some of the above mentioned homotopic configurations (see previous exercise). For instance, L can be considered as the union of configuration n° 3, n° 6, n° 6 rotated counterclockwise and n° 10 also rotated counterclockwise. Then we have:

$$X \odot L = \bigcup_i (X \odot L_i) \text{ with } L = \{L_i\}$$

$\{L_i\}$ being the sequence of configurations listed above.



Homotopic full thinnings applied on the holes (left) and gruyere (right) images. From top to bottom: original image, thinL, thinM and thinD.

Similarly:

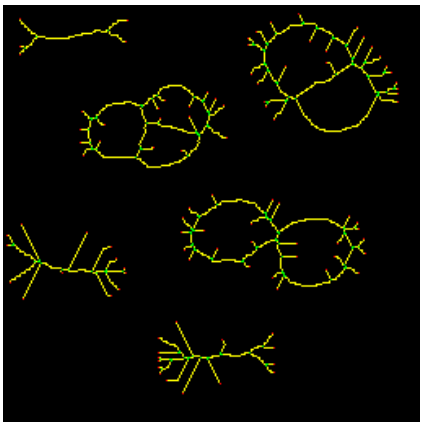
$$X \bigcirc L = \bigcap_i (X \bigcirc L_i)$$

It can be shown that these unions of thickenings or intersections of thinnings produce homotopic transforms, provided that the listed double structuring elements (with the right orientation) are used.
Note that, in general, unions of homotopic thickenings and intersections of homotopic thinnings (named **infThin** and **supThick** in MAMBA) are not homotopic.
Load the *holes* and *gruyere* images and test the **thinL**, **thinM** and **thinD** operators.
These three homotopic transformations give quite different results :
- **Mthin** produces a rather high number of dendrites and barbs (very sensitive to local irregularities).
- **Dthin** reduces any simply connected set to one point (this point exists in the resulting images although not very visible). On the contrary, when a connected components contains holes, they are preserved (remember that it is an homotopic transform) and some thick parts may remain.
- **Lthin** is the only transformation whose behavior corresponds to the intuitive idea one may have of the skeleton of a set. This is why this thinning is sometimes called L-skeleton.
Note that thickM is the dual transform of thinD and thickD, the dual transform of thinM. The double structuring element L is auto-dual.

2) Extracting characteristic points of the homotopic thinnings
The MAMBA operators available for extracting extremities and mutliple points are respectively **endPoints** and **multiplePoints**. On the hexagonal grid, for instance, extremities correspond to configurations n° 1, 2 and 3 (up to a rotation).

$$\begin{array}{ccc} & 0 & 0 \\ 0 & 1 & 0 \\ & 0 & 0 \end{array} \cup \begin{array}{ccc} & 1 & 0 \\ 0 & 1 & 0 \\ & 0 & 0 \end{array} \cup \begin{array}{ccc} & 1 & 1 \\ 0 & 1 & 0 \\ & 0 & 0 \end{array} = \begin{array}{ccc} & \bullet & \bullet \\ 0 & 1 & 0 \\ & 0 & 0 \end{array}$$

In order to simplify the structuring element, note that these three configurations can be merged in a single one (see above).



Skeleton (in yellow), its end points in red and its multiple points in green.

We already extracted isolated points (see exercise n° 1). They correspond to a hit-or-miss transform with configuration n° 1.

3) To generate the first test image into imbin1, do the following (use the method **setPixel**):

```
>>> imbin1.reset()
```

```
>>> imbin1.setPixel(1, (127, 127))
>>> imbin1.setPixel(1, (128, 127))
>>> imbin1.setPixel(1, (131, 127))
>>> imbin1.setPixel(1, (132, 127))
```

An for the second one in imbin2:

```
>>> imbin2.reset()
>>> imbin2.setPixel(1, (128, 127))
>>> imbin2.setPixel(1, (131, 127))
>>> imbin2.setPixel(1, (129, 128))
>>> imbin2.setPixel(1, (131, 128))
```

Then, perform the L-skeleton by thickening:

```
>>> thickL(imbin1, imbin3)
```

First test image (left) and result of the full L thickening (right).

Perform the same transformation on the second test image:

```
>>> thickL(imbin2, imbin4)
```

Second test image (left) and result of the full L thickening (right).

In both cases, the result is highly biased. The separation between the two connected components should be vertical. This bias comes from the fact that the thinning is not isotropic, since each structuring element is applied one direction after the other. Therefore, the first step of thickening tends to propagate the result in the first direction and the following thickenings increase the anisotropy.
Changing the direction of the first thickening does not reduce the bias, as shown in the following test. Instead of using the **hexagonalL** structuring element in the **thickL** operation,

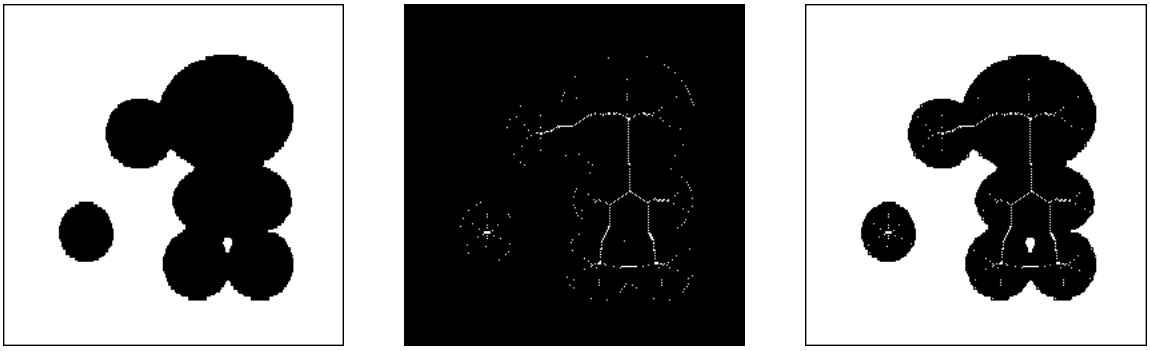
we can use a counterclockwise rotated structuring element. This can be done by means of the following commands:

```
>>> dse = hexagonalL
>>> dse = dse.rotate(-1)
>>> fullThick(imbin2, imbin3, dse)
```

Applied on the second test image, the result is not better.

Result of the thickL operation (right) applied on the second test image (left) when the starting direction is different.

More efficient algorithms exist. Some of them use structuring elements defined on hexagons of size 2 where all rotations can be used simultaneously. Another solution consists in using a geodesic homotopic thinning in combination with the skeleton by maximal balls. Let us see how to proceed with the following example.



Initial disk image (left), geodesic space (middle) and set to be thinned inside this geodesic space (right).

Load image *disks* into imbin1, then compute the skeleton by maximal balls (by openings):

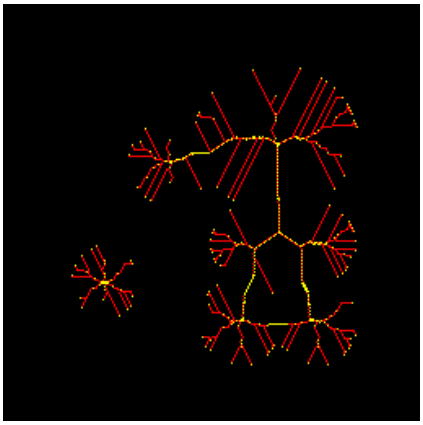
```
>>> skeletonByOpening(imbin1, imbin2, im32_1)
```

Inverting the result provides the geodesic space (in imbin3), while the set to be thinned is the initial set minus its skeleton points (in imbin4):

```
>>> negate(imbin2, imbin3)
>>> logic(imbin1, imbin3, imbin4, "inf")
```

We can then perform the full geodesic thinning with the L structuring element and add the initial skeleton by openings to the thinning to get the final result::

```
>>> fullGeodesicThin(imbin4, imbin3, imbin4, hexagonalL)
>>> logic(imbin2, imbin4, imbin4, "sup")
```



Connected skeleton: in yellow, centers of maximal hexagons, in red, connecting points added by the geodesic thinning.

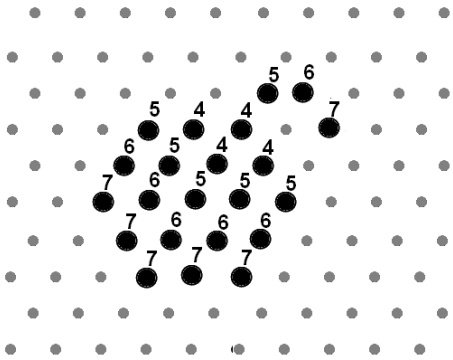
This skeleton is interesting as it contains the centers of the maximal hexagonal and it is also connected, which was not the case with the simple skeleton by opening.

Exercise n° 6

1) We can notice that the value taken at point x of X by the function d corresponds to the minimal size n of the geodesic dilation δ_X^n of x which fills entirely the set X:

$$\forall x \in X, \quad d(x) = n \text{ where } n = \inf(i) \text{ such that } \delta_X^i(x) = X$$

This function is given in the figure below. Its maxima correspond to the extremities of the particle and the minimum to the geodesic center. Note that, in the difgital case, these extrema are not reduced to a single pixel.



Geodesic distance function.

2) Consider two adjacent points x and y and suppose that d(x) = n and $d(y) \geq n+2$ (n > 1). We have:

$$\delta_X^n(x) = X \text{ and } \delta_X^{n+1}(y) \neq X$$

But:

$$\delta_X^1(y) \supset x$$

Thus:

$$\delta_X^{n+1}(y) \supset \delta_X^n(x) = X$$

The geodesic dilation of size (n + 1) of y fills X. Therefore d(y) cannot be greater than or equal to (n + 2).

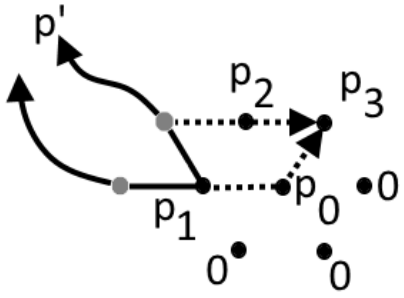
3) Let us consider the D_3 configuration. Assume that the value of d at the center point is equal to n. The value of d on the three neighbor points can possibly be equal to n, n - 1 or n + 1. However, among the 27 initial configurations, many of them can be discarded because either these configurations are identical to others up to a symmetry or they are not possible because the difference of the d function values on adjacent points is larger than 1. For instance, this configuration is not possible because the difference of the d values on two adjacent points is greater than 1:

$$\begin{array}{ccccc} & & n-1 & & n+1 \\ n-1 & & & n & & & 0 \\ & & 0 & & 0 \end{array}$$

This first filtering allows to eliminate 15 configurations. Among the 12 remaining ones, a second analysis suppresses some of them. For instance, this configuration is also impossible:

$$\begin{array}{ccccc} & & n-1 & & n-1 \\ n-1 & & & n & & & 0 \\ & & 0 & & 0 \end{array}$$

Indeed, let us consider the left neighbor point (p_1) where d is equal to (n - 1). The minimal path of length (n - 1) joining this point to its farthest point p' goes westward as shown in the figure below. But, in this case, p_3 is linked to p' by a minimal path of length n or (n + 1). Therefore, there exists a point p' of X at a geodesic distance greater than (n - 1) from p_3; d(p_3) cannot be equal to (n - 1).



The above configuration is not possible otherwise p_3 would necessarily be linked to p' by a minimal geodesic path of length greater than (n - 1).

This configuration, obviously, is also impossible (otherwise, d(p_0) could not be equal to n):

$$\begin{array}{ccccc} & & n+1 & & n+1 \\ n+1 & & & n & & & 0 \\ & & 0 & & 0 \end{array}$$

After this second sieving performed on these 12 configurations, only four of them remain possible (up to rotations and symmetries):

$$\begin{array}{ccccc} & n-1 & & n & \\ n-1 & & n & & 0 \\ & 0 & & 0 & \end{array} \qquad \begin{array}{ccccc} & n-1 & & n & \\ n & & n & & 0 \\ & 0 & & 0 & \end{array}$$

A B

$$\begin{array}{ccccc} & n & & n+1 & \\ n-1 & & n & & 0 \\ & 0 & & 0 & \end{array} \qquad \begin{array}{ccccc} & n & & n+1 & \\ n+1 & & n & & 0 \\ & 0 & & 0 & \end{array}$$

C D

We can notice that, for any of them, there always exists at least one neighbor point where d is equal to n, the value taken by the geodesic distance function in p_0.

We must now find configurations among these four remaining ones which can be problematic when thinnings with all the rotations of D_3 are applied in parallel, in particular, configurations where point p_2 may be removed at the same time as the central point p_0, which would break an homotopic path and also configurations where the neighbor points with d equal to n could be removed.

Let us see the first configuration (A). To remove p_2 with a thinning by D_3 (rotated), the following pattern must appear:

$$\begin{array}{ccccccc} & 0 & & 0 & & & \\ 0 & & \begin{matrix} p_2 \\ (n-1) \end{matrix} & & \begin{matrix} p_3 \\ (n) \end{matrix} & & \\ & \begin{matrix} p_1 \\ (n-1) \end{matrix} & & \begin{matrix} p_0 \\ (n) \end{matrix} & & 0 & \\ & & 0 & & 0 & & \end{array}$$

However, this is not possible because the pattern around p_2 corresponds to (after rotation and modification of the central value):

$$\begin{array}{ccccc} & n+1 & & n & \\ n+1 & & n & & 0 \\ & 0 & & 0 & \end{array}$$

But this configuration does not match any of the four above configurations. So, in configuration (A), p_2 is never removed whatever the direction of the D_3 structuring element. The same conclusion holds for the configuration (B).

Regarding points p_1 and p_3 in these configurations, if p_0 has been replaced by 0 after a thinning by D_3 in a given direction, no thinning in any other direction can remove p_1 or p_3.

$$\begin{array}{ccccc} & p_2 & & p_3 & \\ p_1 & & p_0 & & 0 \\ & 0 & & 0 & \end{array} \qquad \underset{\text{Direction of thinning}}{\nwarrow} \qquad \begin{array}{ccccc} & p_2 & & p_3 & \\ p_1 & & 0 & & 0 \\ & 0 & & 0 & \end{array}$$

After the removal of p_0 by thinning in the indicated direction, neither p_1 nor p_3 can be removed by thinning in any other direction.

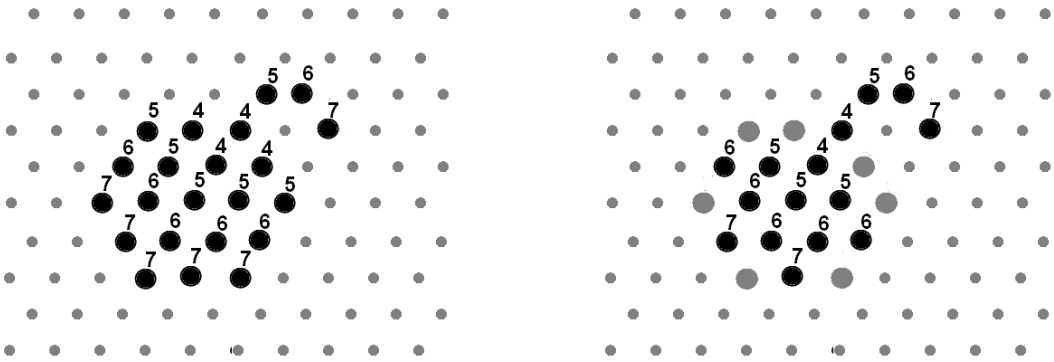
Contrary to configurations (A) and (B), in configurations (C) and (D), the points p_0 and p_2 may be removed simultaneously by thinnings with D_3 in opposite directions. We may have, for instance, the following configuration:

$$\begin{array}{ccccc} & 0 & & 0 & \\ 0 & & p_2 & & p_3 \\ & p_1 & & p_0 & & 0 \\ & & 0 & & 0 & \end{array}$$

where p_0 and p_2 are replaced by 0 after simultaneous thinnings by two double structuring elements in opposite directions. This removal must obviously be avoided because it would break possible homotopic paths joining p_1 and p_3.
This long analysis of the D_3 structuring element allows to design the following procedure which eliminates all the D_3 configurations (except those which are compulsory for preserving homotopy) so that the only remaining configurations are configurations D_2 and D_1:

For directions i = 1, 2, 3 (of the hexagonal grid):
    Perform hit-or-miss transforms of the set X with D_3 in direction i and i + 180°
    Perform the union of these two hit-or-miss transforms
    Extract isolated points of the union
    Remove isolated points from the set X (update the set)

This procedure must be iterated until idempotence. The resulting set has two properties:
- no D_3 configuration is present except those which are needed to preserve homotopy.
- The geodesic distance function of the resulting set is identical on the remaining points to the initial function.
As a consequence, the geodesic distance function of the resulting set is identical on the remaining points to the initial one and the geodesic center and the extremities of this resulting set are the same up to the possible removed points.
This procedure is not isotropic as the final result depends on the order of use of the three directions in the hit-or-miss operators. But this fact has no unfortunate consequence for the rest of the process.



Initial set and its geodesic distance function (left), result of the complete mixed thinning by D_3 double structuring elements (left). The grey points have been removed, but the geodesic distance function is not changed on the remaining points. The new extremities and the geodesic center are still embedded in the previous ones.

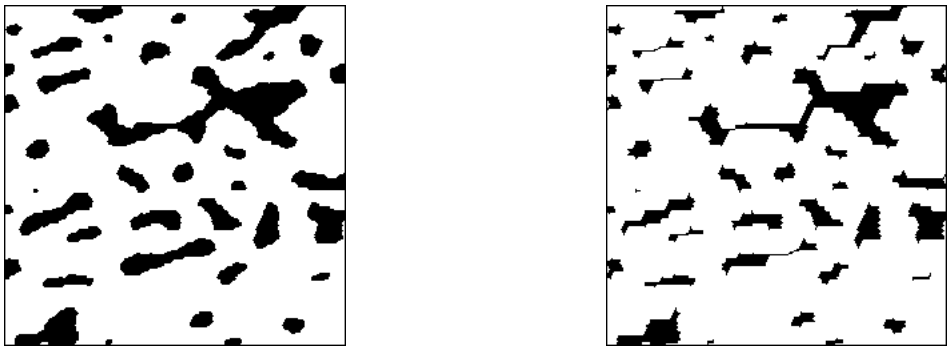
The complete transformation is achieved by the following operator named **thinByD3**:

```
def thinByD3(imIn, imOut):
    """
    Controlled thinning by the D3 structuring element using a mixture of
    rotational thinnings and parallel ones in order to preserve the extremities
```

```
    and the geodesic centers of the initial binary image 'imIn'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    copy(imIn, imOut)
    dse3 = doubleStructuringElement(structuringElement([2, 3, 4], HEXAGONAL),
    structuringElement([0, 1, 5, 6], HEXAGONAL))
    dse0 = doubleStructuringElement(structuringElement([1, 2, 3, 4, 5, 6], HEXAGONAL),
    structuringElement([0], HEXAGONAL))
    v1 = computeVolume(imOut)
    v2 = 0
    while v1 <> v2:
        v2 = v1
        for i in range(3):
            hitOrMiss(imOut, imWrk1, dse3)
            hitOrMiss(imOut, imWrk2, dse3.rotate(3))
            logic(imWrk1, imWrk2, imWrk1, "sup")
            hitOrMiss(imWrk1, imWrk2, dse0)
            diff(imOut, imWrk2, imOut)
            dse3 = dse3.rotate(1)
        v1 = computeVolume(imOut)
```

Load the *metal1* image in imbin1 and enter the following command:

```
>>> thinByD3(imbin1, imbin2)
```



metal1 image (left) and result of a first step of complete thinning using the thinByD3 operator (right). No D_3 configuration is present in the resulting image except those which are needed to preserve homotopy.

4) Analysing thinnings by double structuring elements D_1 and D_2 is simpler and easier. Concerning D_1, only the two following configurations are possible:

$$\begin{matrix} n-1 & 0 & \\ 0 & n & 0 \\ 0 & 0 & \end{matrix} \qquad \begin{matrix} & n & 0 \\ 0 & n & 0 \\ & 0 & 0 \end{matrix}$$

The first one corresponds to an extremity and the thinning does not raise any problem, except when n = 1. This configuration corresponds then to an isolated point. The second one is possible only when n = 1. In this case, the two points form an single connected component and will be removed by the inf of thinnings by the D_1 structuring elements. This connected component is a geodesic center.

Three configurations (up to rotations and symmetries) are possible with the D_2 double structuring element:

$$\begin{matrix} n-1 & n-1 & & n-1 & n & & n & n \\ 0 & n & 0 & 0 & n & 0 & 0 & n & 0 \\ 0 & 0 & & 0 & 0 & & 0 & 0 \end{matrix}$$

In the first configuration, the central point is an extremity and is removed by thinning. This is also the case in the second configuration (remember that no D_3 configuration exists). The third one is, as for D_1, possible only when n = 1. In this case, the three points are removed and, as for D_1, they form a single connected component and is a geodesic center.

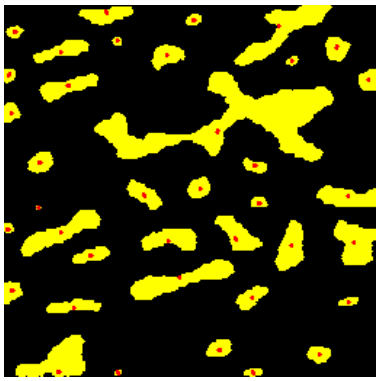
5) The previous analysis allows to design a procedure to extract the geodesic centers of the simply connected components of an image. The successive steps of the procedure are iterated until idempotence:
- The **thinBy3D** operator is applied to the set.
- An intersection of thinnings by all the rotations of D_2 and D_1 is performed on the previous result. The connected components which have been suppressed after this step correspond to geodesic centers.
- The geodesic reconstruction of the previous result (**thinBy3D** operator) using the last step (intersection of thinnings) does not rebuild the removed geodesic centers. A set difference extracts them and they can be reintroduced in the last result before a new iteration of the whole procedure.

This procedure, iterated until idempotence, is realised by the **geodesicCenter** operator:

```
def geodesicCenter(imIn, imOut):
  """
  Computes the geodesic centers of the connected components of 'imIn' by using
  successive thinnings by D3, then D2 and D1 until idempotence.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  copy(imIn, imOut)
  dse1 = doubleStructuringElement(structuringElement([1, 2, 3, 4, 5], HEXAGONAL),
  structuringElement([0, 6], HEXAGONAL))
  dse2 = doubleStructuringElement(structuringElement([2, 3, 4, 5], HEXAGONAL),
  structuringElement([0, 1, 6], HEXAGONAL))
  v1 = computeVolume(imOut)
  v2 = 0
  while v1 <> v2:
     v2 = v1
     thinByD3(imOut, imOut)
     infThin(imOut, imWrk1, dse2)
     infThin(imOut, imWrk2, dse1)
     logic(imWrk1, imWrk2, imWrk1, "inf")
     copy(imWrk1, imWrk2)
     build(imOut, imWrk2)
     diff(imOut, imWrk2, imWrk2)
     logic(imWrk1, imWrk2, imOut, "sup")
     v1 = computeVolume(imOut)
```

Load image metal1 in imbin1 and enter:

```
>>> geodesicCenter(imbin1, imbin2)
```

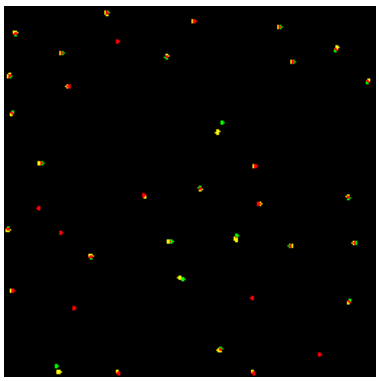


Geodesic centers (in red, after a dilation) of the metal1 image (in yellow) obtained by the geodesicCenter operator.

This operator can be compared to the thinning with the double structuring element D producing a single point (named centroid) per particle. Use again the *metal1* image and enter the following commands:

```
>>> geodesicCenter(imbin1, imbin2)
>>> thinD(imbin1, imbin3)
```

The results are displayed in the following image. The differences are important when the connected components are elongated with significant variations in thickness. However, in many cases, the bias is not so important. This is why the **thinD** operator is very often used as it is simpler and faster. Remind also that these operators work only on the hexagonal grid. There exists in MAMBA an operator adapted to the square grid for extracting centroids. It is named **homotopicReduction** and it is more complex than the hexagonal one.

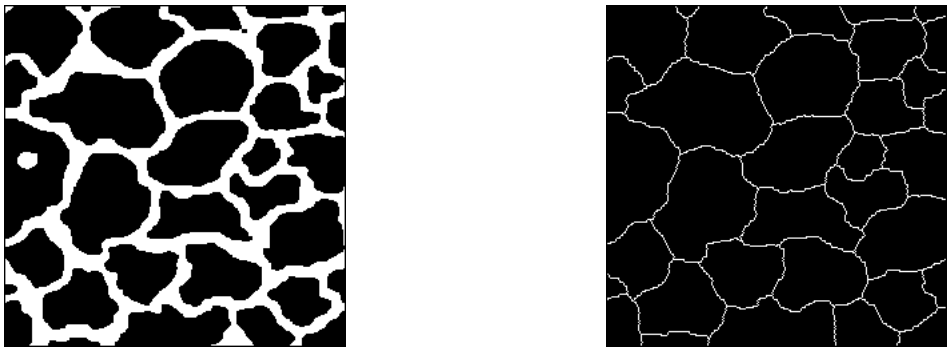


Geodesic centers obtained with the geodesicCenter procedure in yellow, with the thinD in green (centroids). When both centers are superposed, they appear in red.

Exercise n° 7

1) Load the *alumine* image in imbin1 and enter:

```
>>> computeSKIZ(imbin1, imbin2)
```

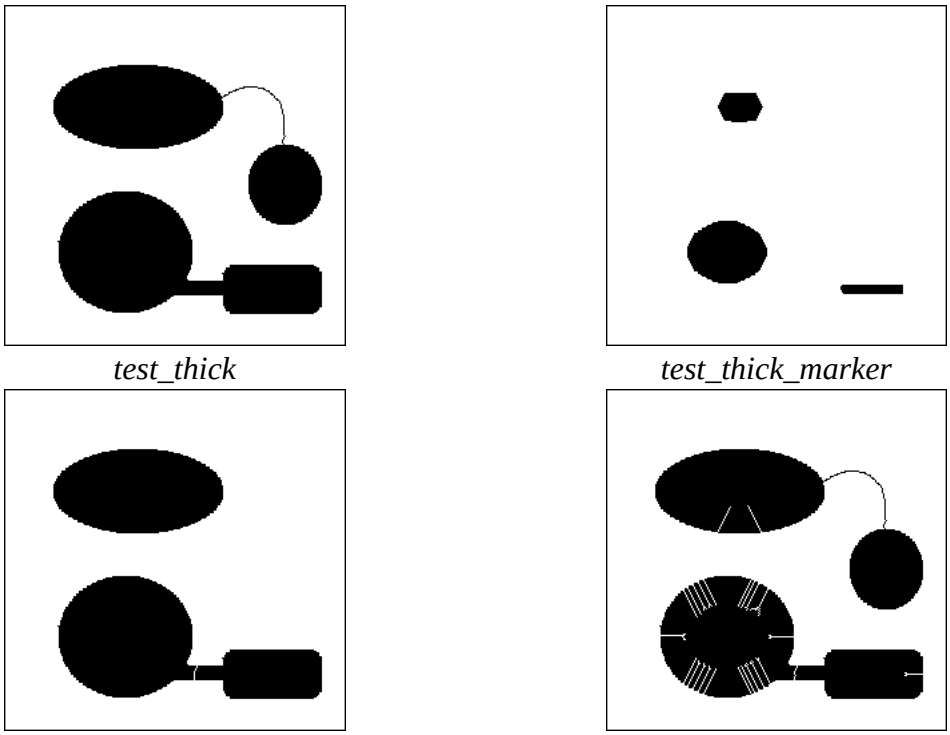


alumine image (left) and its SKIZ (right).

This transformation, as shown in the example, is not homotopic: the hole in the left particle is filled after the SKIZ.

2) Load the *test_thick* image in imbin1 and the *test_thick_marker* image in imbin2, then enter the following commands successively:

```
>>> fullGeodesicThick(imbin2, imbin1, imbin3, hexagonalL)
>>> fullGeodesicThick(imbin2, imbin1, imbin3, hexagonalD)
```



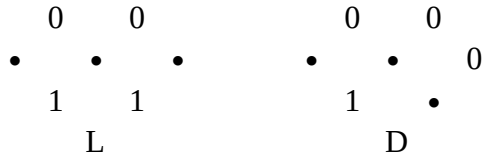
test_thick *test_thick_marker*

Result of the geodesic thickening when the L structuring element is used (left image) and when the D structuring element is used (right image).

You may also display the imbin3 image before launching the transformations in order to see the propagation of the thickening with the two structuring elements:

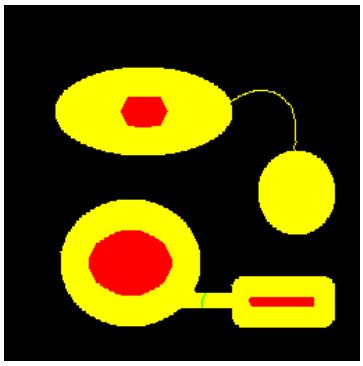
```
>>> imbin3.show()
```

Using the L structuring element does not allow to invade the regions without thickness of the geodesic set. On the contrary, this propagation is possible when using the D structuring element. However, this structuring element produces many barbs.



3) As it has been shown previously, computing the geodesic SKIZ with thickenings require to use carefully a judicious combination of structuring elements in order to avoid the biases and defects introduced by these transformations. This operator fortunateley already exists in MAMBA. It is not based on thickenings but on the watershed transform (see next chapter). It is named **geodesicSKIZ**. Try it with *test_thick* and *test_thick_marker* respectively loaded in imbin1 and imbin2:

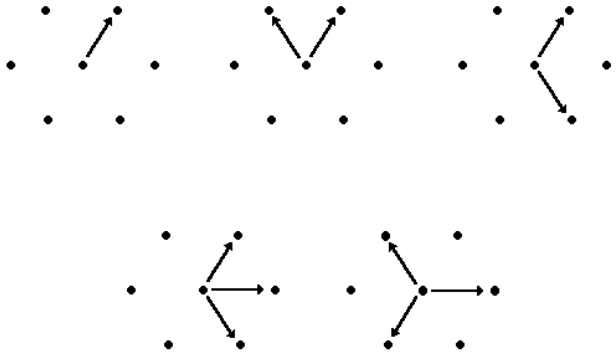
```
>>> geodesicSKIZ(imbin2, imbin1, imbin3)
```



Geodesic SKIZ (the boundary between the zones of influence is in green) of test_thick_marker (in red) inside the geodesic space test_thick (in yellow).

Exercise n° 8

1) As already mentioned in exercise n°6 in the previous chapter, computing the gradient azimuth requires to compute the directional gradients $g^{\alpha}(f)$ for all the directions α that can be exhibited on the digitization grid. Then, the gradient azimuth is the direction α that corresponds to the highest directional gradient. Since the processing is made on hexagonal grid, there are six possible directions for the azimuth. However, several directions of directional gradients may happen to be maximal values. This phenomenon is annoying as the gradient vector is unique at any point of the image. It is therefore necessary to correct the rough image of the azimuths obtained by simply detecting the direction(s) of highest directional gradient. In order to do so, a first transformation allows to detect all the directions for which the directional gradient is maximum.
Computing these gradients by thickening/thinning implies that the directional gradients can be maximal in three directions at most. The set of the possible configurations of maximal directional gradients (up to the rotations) is given below:

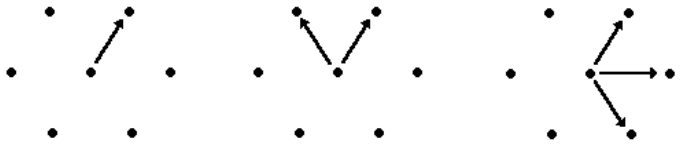


Set of all the configurations where the directional gradient can be maximal (its value is the same for all the concerned directions).

The procedure for computing this directional gradient with thickenings and thinnings, named **thinThickGradient**, is given below:

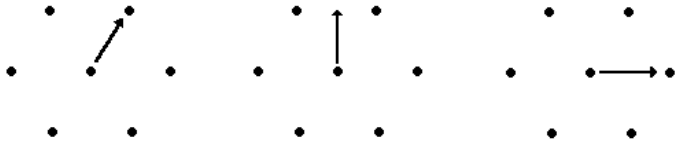
```
def thinThickGradient(imIn, imOut, dir):
    """
    Computes the modulus of the directional gradient in direction 'dir' of 'imIn' and
    stores the result in 'imOut'. A greyscale thickening and a greyscale thinning are
    used.
    """

    imWrk = imageMb(imIn)
    se = structuringElement([dir], HEXAGONAL)
    dse = doubleStructuringElement(se.transpose(), se)
    greyThick(imIn, imWrk, dse)
    greyThin(imIn, imOut, dse)
    sub(imWrk, imOut, imOut)
```

2) If the maximal gradient directions are not adjacent, then the gradient is considered to be zero. Otherwise a unique direction is chosen, which is the mean directions of all present directions. The initial sorting is illustrated in the following figure:



Among the above possible configurations, only these three correspond to a non zero directional gradient.

The second configuration is quite interesting, as, in this case, the mean direction is one of the conjugated directions of the grid. This is the reason why the azimuth of the final gradient is coded on twelve directions instead of six. Each direction is coded by a numerical value in the range [0,12].



Final azimuths retained from the above configurations. The second one is defined on conjugated directions.

3) To program the complete gradient (modulus and azimuth), we define first a procedure named **roughVectorGrad**. This procedure computes the modulus of the gradient and generates an image coding all the directions where the directional gradient is maximal. This coding stores in the bit plane (i -1) all the pixels where the gradient is maximal in the direction i. For instance, if , for a given pixel, the gradient is maximal in directions 1, 3 and 4, the value of the returned function is equal to 13 ($2^0+2^2+2^3$). This procedure is defined as follows:

```
def roughVectorGrad(imIn, imOut1, imOut2):
  """
  Initial vectorial gradient of the 'imIn' image. 'imOut1' contains the modulus
  with some erroneous values (non zero) where the azimuth should be egal to zero.
  'imOut2' contains a first computation of the azimuth: all the directions where
  the modulus is maximal are stored in the various bit planes of 'imOut2'.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  imMask = imageMb(imIn, 1)
  imOut1.reset()
  imOut2.reset()
  for i in range(6):
     dir = i + 1
     thinThickGradient(imIn, imWrk1, dir)
     generateSupMask(imWrk1, imOut1, imMask, True)
     convertByMask(imMask, imWrk2, 255, 0)
     logic(imOut2, imWrk2, imOut2, "inf")
     generateSupMask(imWrk1, imOut1, imMask, False)
     logic(imWrk1, imOut1, imOut1, "sup")
     copyBitPlane(imMask, i, imOut2)
```

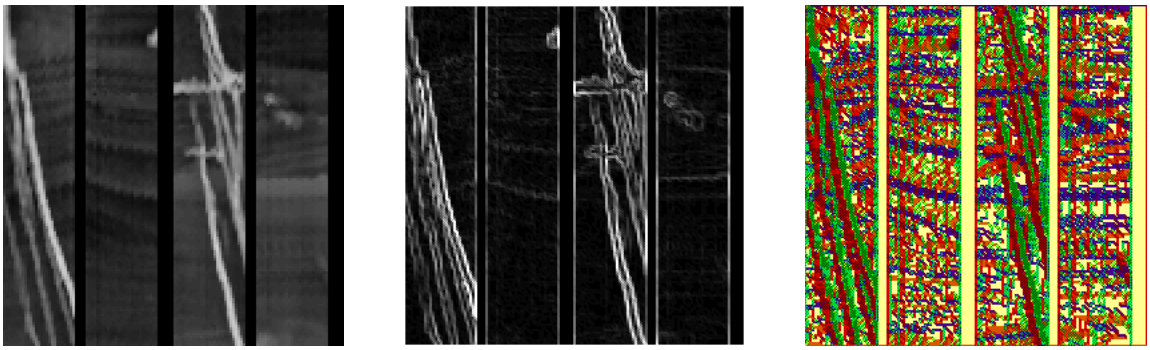


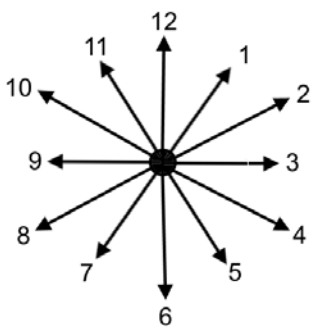
Image petrole (left), gradient modulus provided by the roughVectorGrad (middle), azimuth directions coded by the same procedure and colored with the patchwork palette (right).

Note that the result is not correct yet, not only for the azimuth but also for the modulus as this component of the gradient may be different of 0 simply because non adjacent directions may have returned equal non zero values.
This is why this intermediary procedure must be followed by a final one, named **vectorialGradient**, which generates the right directions of the azimuth and correct the wrong values of the modulus by replacing them by 0.
The generation and coding of the final directions uses a look-up table. This look-up table is a list which contains at index i (coding of the previous directions) the coding of the new gradient direction. For example, an initial coding equal to 7 (directions 1, 2 and 3) will be equal to 3 (direction 2) after correction. This look-up table, named **GradLut** is defined by the **defineGradLut** procedure:

```
def defineGradLut():
    """
    Generation of the look-up table correcting the initial coding of the azimuths
    provided by the roughVectorGrad function.
    """

    gradLut = [0 for i in range(256)]
    for i in range(6):
        j = (2 ** i)
        gradLut[j] = (2 * i) + 1
        j = j + (2 ** (i + 1)) % 63
        gradLut[j] = 2 * (i + 1)
        j = j + (2 ** (i + 2)) % 63
        gradLut[j] = (2 * (i + 1) + 1) % 12
    return gradLut
```

Launch the procedure to get the look-up table:

```
>>> defineGradLut()
[0, 1, 3, 2, 5, 0, 4, 3, 7, 0, 0, 0, 6, 0, 5, 0, 9, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 7, 0, 0, 0, 11, 12, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 11, 0, 0, 0, 0, 0, 0, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```



Coding of the 12 directions of azimuth.

Twelve directions are thus defined, the odd directions correspond to the main directions of the hexagonal grid and the even directions to the conjugated directions.
The final gradient operator, named **vectorialGradient**, uses this look-up table to code correctly the directions of the azimuth and replaces by zero the values of the modulus where the azimuth is also equal to zero:

```
def vectorialGradient(imIn, imOut1, imOut2):
  """
  Vectorial gradient of the 'imIn' image. 'imOut' contains the gradient modulus and
  'imOut2' contains the gradient azimuths coded by twelve directions.
  """

  imWrk = imageMb(imIn)
  imMask = imageMb(imIn, 1)
  gradLut = defineGradLut()
  roughVectorGrad(imIn, imOut1, imWrk)
  lookup(imWrk, imOut2, gradLut)
  threshold(imOut2, imMask, 0, 0)
  convertByMask(imMask, imWrk, 255, 0)
  logic(imOut1, imWrk, imOut1, "inf" )
```

This gradient can be applied to the seismic_section image which has already been used in exercise n°6 in the previous chapter. Load it in the imA image, a 448x448 8-bit image. Two other images, imB and imC are also defined:

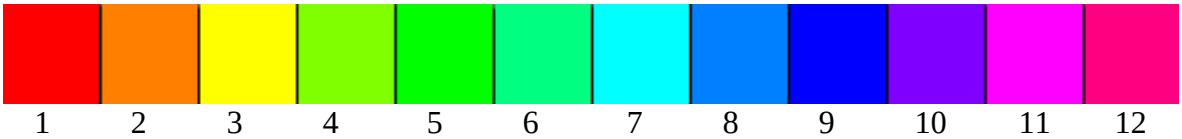
```
>>> imA = imageMb(448, 448)
>>> imB = imageMb(imA)
>>> imC = imageMb(imA)
```

Then, the **vectorialGradient** operator is applied to imA. The modulus is put in imB and the azimuth is in imC:

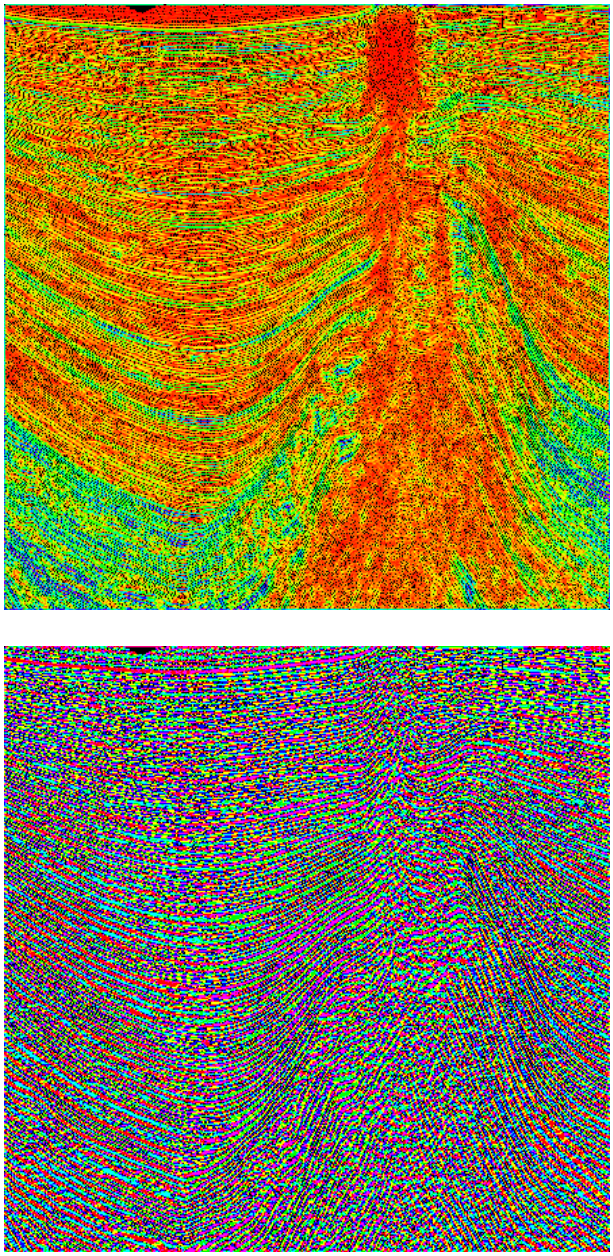
```
>>> vectorialGradient(imA, imB, imC)
```

The modulus image can be displayed again with the rainbow palette. For the azimuth, as 12 directions are now available, a new palette can be defined to display it:

```
>>> imB.show(palette="rainbow")
>>> dir12pal = (0,0,0, 255,0,0, 255,128,0, 255,255,0, 128,255,0, 0,255,0, 0,255,128,
0,255,255, 0,128,255, 0,0,255, 128,0,255, 255,0,255, 255,0,128)
>>> addPalette("direction12 palette", dir12pal)
>>> imC.show(palette="direction12 palette")
```



Color palette used to display the twelve directions of the azimuth in the vectorialGradient procedure.



Gradient modulus of the seismic_section image (top), gradient azimuth (bottom) obtained with the vectorialGradient operator.

This gradient (modulus and azimuth) can be compared to the previous one where simple linear dilations and erosions were used with 6 directions instead of 12. The azimuth is more accurate in this latter case.

Exercise n° 9: Classification of particles

Although this problem is simple to be expressed, it may be quite complex to be solved if fast, elegant and universal solutions are awaited. However, a simple, universal (but not fast) solution exists. It is based on the individual analysis of particles already introduced in chapter 3, exercise n°4. Implementing this solution is performed as follows:
- The initial image is labelled with the **label** operator which returns the number of connected components in the image.
- The ith particle is extracted by thresholding the labelled image in the range [i, i].

- Measuring the connectivity number ν of the connected component gives its number of holes n_T with the formula $n_T = 1 - \nu$.
- Any particle with a number of holes equal to a given value n is added to the output image.

The MAMBA implementation of the procedure named **holesSieving** is the following:

```
def holesSieving(imIn, imOut, n):
    """
    Puts in imOut all the connected components of imIn which contains exactly n holes.
    """

    imWrk0 = imageMb(imIn, 32)
    imWrk1 = imageMb(imIn)
    imOut.reset()
    # Initial image labelling
    nParticles = label(imIn, imWrk0)
    for i in range(1, nParticles+1):
        # each particle is extracted and its connectivity number is measured
        threshold(imWrk0, imWrk1, i, i)
        nc = computeConnectivityNumber(imWrk1)
        # if the number of holes is equal to n, the particle is added to the output image
        if (n==(1 - nc)):
            logic(imWrk1, imOut, imOut, "sup")
```

This solution is quite slow as it depends of the number of particles in the image. Looking for faster and more elegant solutions needs to analyse this problem step by step. So, let us see first how simply connected particles can be extracted.

1) Extracting simply connected components

A first solution is based on the use of homotopic transforms. A second one uses the geodesic reconstruction.

The first solution uses the **thinD** operator. We know that, when a set is simply connected, **thinD** reduces it to a single point. So, these single points can be extracted and used as markers to rebuild the simply connected components (refer to the first exercise in this chapter to see how to extract isolated points).

Load for instance the *gruyere* image in imbin1 and enter the following commands:

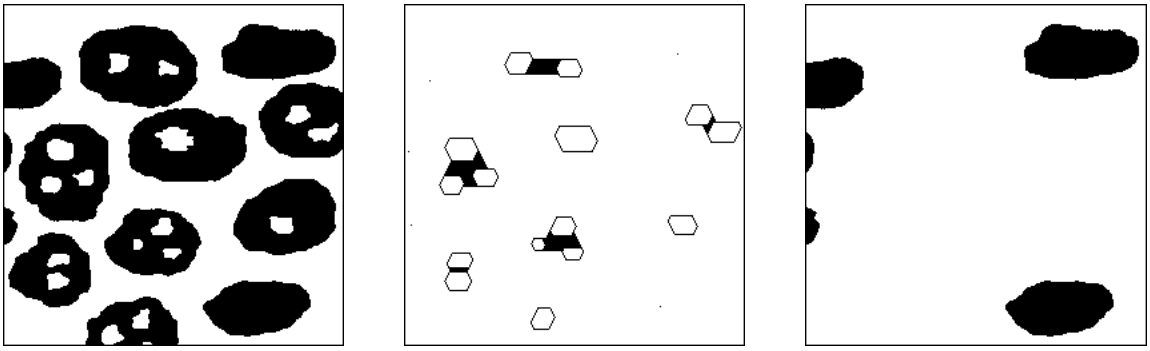
```
>>> thinD(imbin1, imbin2)
>>> dse = doubleStructuringElement(structuringElement([1, 2, 3, 4, 5, 6],
HEXAGONAL),structuringElement([0], HEXAGONAL))
>>> hitOrMiss(imbin2, imbin3, dse)
>>> build(imbin1, imbin3)
```

The imbin3 image contains only the simply connected components of the initial image.

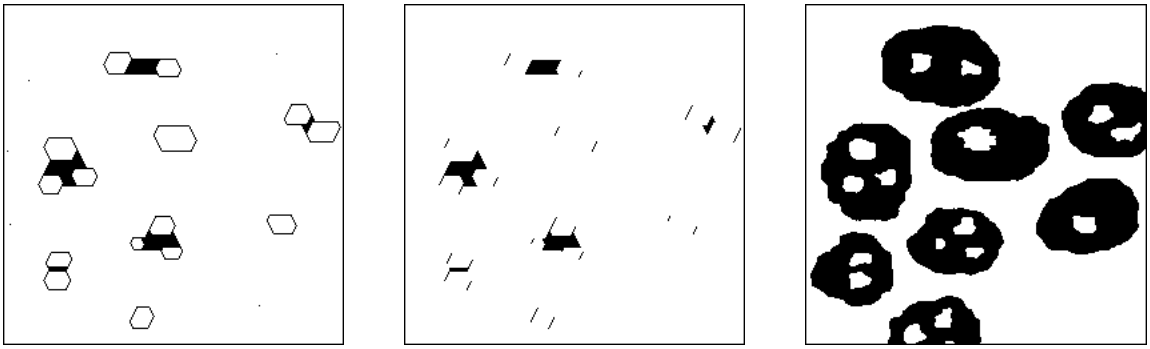
Note that these isolated points can also be removed with an elementary linear erosion (in any direction). It is compulsory to use a linear erosion (with the edge set to **EMPTY** to be sure to remove isolated points which are near the edge) and not an hexagonal one because this latter one would be likely to remove also markers of non simply connected components. Enter the following commands:

```
>>> thinD(imbin1, imbin2)
>>> linearErode(imbin2, imbin3, 1, edge=EMPTY)
```

```
>>> build(imbin1, imbin3)
```

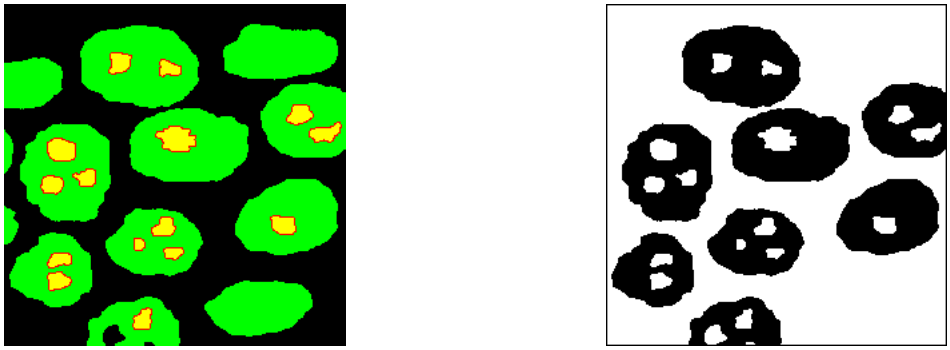


From left to right: initial image, thining with D and geodesic reconstruction of the initial image by the isolated points (simply connected particles).



Thinning with D (left), linear erosion in direction 1 - but any direction can be used (middle), reconstruction of the particles with hole(s) with the previous erosion as marker image (right).

The second approach consists in rebuilding the connected components of the image which are marked by the holes. These holes are extracted first and their elementary dilation marks the particles which contain them. These marked particles can then be reconstructed.



Left: initial image (green), holes (yellow), dilation of the holes generating markers (red). Right: result of the geodesic reconstruction.

Use again the *gruyere* image loaded in imbin1 with the following commands:

```
>>> closeHoles(imbin1, imbin2)
>>> diff(imbin2, imbin1, imbin2)
>>> dilate(imbin2, imbin2)
>>> build(imbin1, imbin2)
```

Believing that these two approaches produce the same result would be an error. Indeed, rather complex structures may appear where holes contain connected components which may, themselves contain holes and so on. Contrary to the *gruyere* image which is simple, the *homotopy* image is an example of such a complex structure. This structure can be represented as a tree, called homotopy tree. The root of the tree corresponds to the background, the first branches correspond to the connected components adjacent to the background. The next branches correspond to the holes adjacent to the previous connected components, etc. This homotopy tree can also be represented by an image where each connected component is given a label corresponding to its position in the tree.

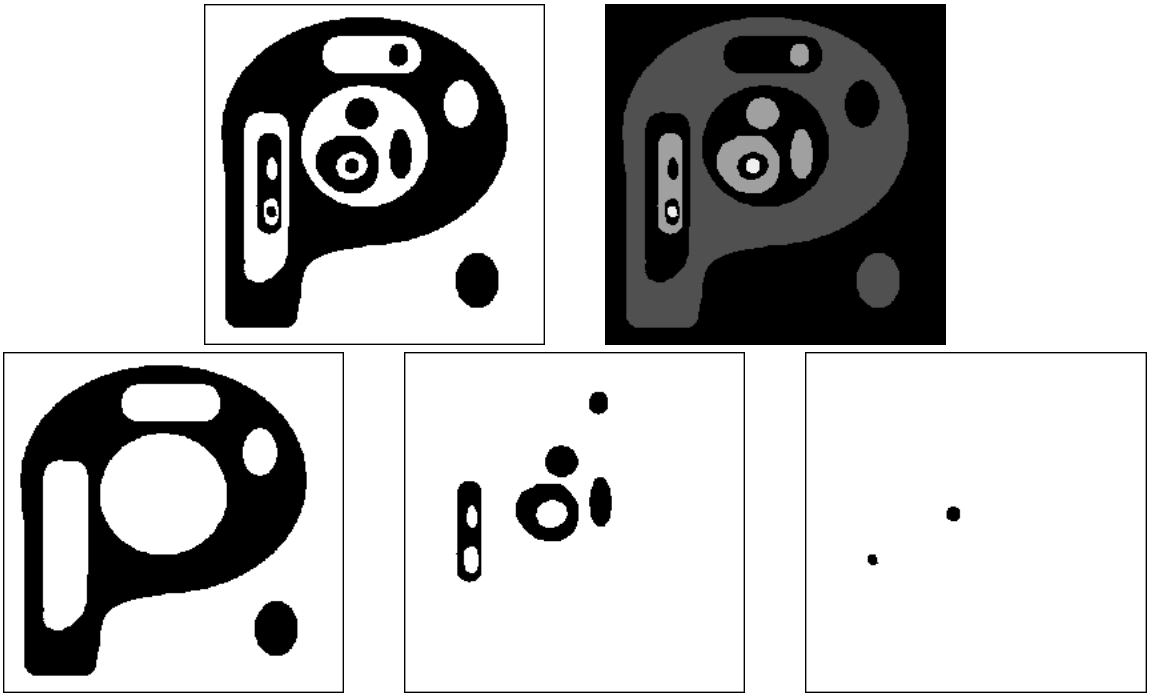
Left: homotopy image. Middle: homotopy tree represented by an image; the connected components of same color belong to the same level of homotopy. Right: corresponding homotopy tree; the black branches represent holes.

The following procedure allows to generate the homotopy image:

```
def homotopyTreeBuild(imIn, imOut):
    """
    Builds the homotopy tree of the initial binary set imIn. The result is stored
    in image imOut. Each grey level corresponds to a hierarchy level of the homotopy
    tree. Extracting each level i of embedding of particles and holes of the initial
    set consists simply in a [i, i] thresholding of imOut.
    """

    imWrk0 = imageMb(imIn)
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn, 8)
    copy(imIn, imWrk0)
    v = computeVolume(imIn)
    imOut.reset()
    i = 0
    # Loop performed until no connected component remains.
    while v!=0:
        i = i+1
        # Background extraction.
        closeHoles(imWrk0, imWrk1)
        negate(imWrk1, imWrk1)
        # Connected components adjacent to the background are rebuilt.
        dilate(imWrk1, imWrk1)
        build(imWrk0, imWrk1)
        # These particles are at level i in the homotopy tree.
        # They are removed from the current set, giving access to the next level (in imWrk0).
        diff(imWrk0, imWrk1, imWrk0)
```

```
v = computeVolume(imWrk0)
# The level-i particles are labelled with i and added to the label image.
convertByMask(imWrk1, imWrk2, 0, i)
add(imOut, imWrk2, imOut)
```

Each level of hierarchy i of the homotopy tree can easily be obtained by a simple thresholding of the homotopy image at value i.



Top left, original image. Top right, homotopy image obtained by the homotopyTreeBuild procedure (its contrast has been enhanced). Bottom: the successive hierarchies of homotopy obtained by thresholding the homotopy image.

Therefore, the second approach must be applied on each homotopy level.

2) Extracting particles with a single hole

Among many other possibilities, the geodesic SKIZ can be used to solve this problem. Consider the initial set. We first close and extract its holes. Then, we keep only the non simply connected components by rebuilding them from the set without holes. This set is considered as the geodesic space used to perform the geodesic SKIZ of the holes inside it. This geodesic SKIZ separates into several parts the connected components which contain at least two holes. So we just have to rebuild the particles marked by the boundaries of the geodesic SKIZ to obtain the particles with more than one hole.

The different steps of this procedure are given below. The initial *gruyere* image is in imbin1:

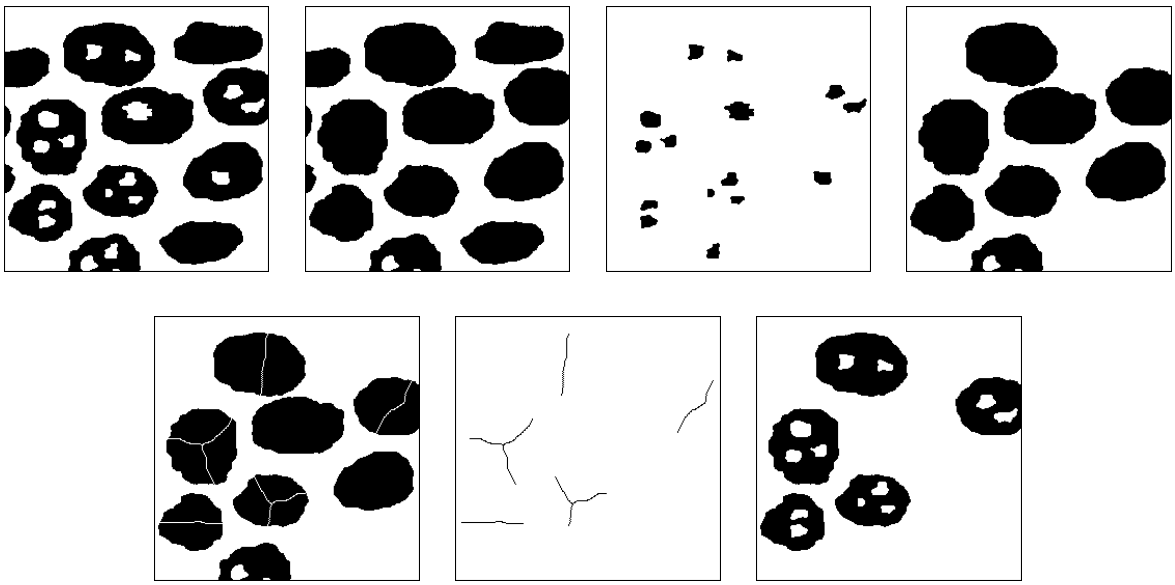
```
>>> closeHoles(imbin1, imbin2)
>>> diff(imbin2, imbin1, imbin3)
>>> copy(imbin3, imbin4)
>>> build(imbin2, imbin4)
```

The holes are filled in imbin2 and stored in imbin3 and imbin4. Then, the particles with hole(s) are extracted in imbin4 (after filling).

```
>>> geodesicSKIZ(imbin3, imbin4, imbin3)
```

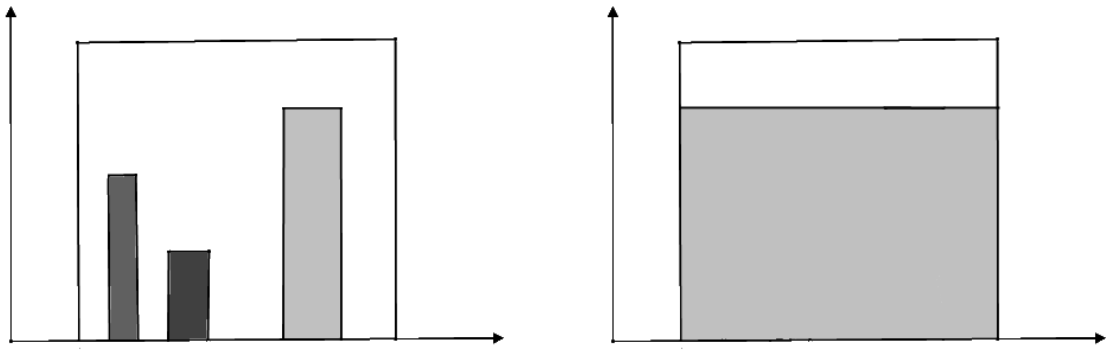
```
>>> diff(imbin4, imbin3, imbin3)
>>> build(imbin1, imbin3)
```

The geodesic SKIZ of the holes inside imbin4 is stored in imbin3. Then, the boundaries of the geodesic SKIZ are extracted and used as markers to rebuild the particles containing at least two holes.
These steps are illustrated below.



From left to right and from top to bottom: initial image, image with its holes filled, extracted holes, particles with holes (after filling), geodesic SKIZ of the holes in the previous image, boundaries of the SKIZ, geodesic reconstruction of the particles containing at least two holes.

We can also use the geodesic reconstruction in a different way by defining an operator named **closeOneHole** which will close one and only one hole per connected component with holes. This operator, however, works only with particles with a single level of the homotopy tree. If it is not the case, this operator must be applied to each level of the homotopy tree. This operator is built as following:
- Holes of the initial set are filled.
- These holes are extracted and labelled: each hole is therefore assigned a unique numerical value.
- The initial set with its holes filled is converted a two-level (0 and 2^{32} - 1) 32-bit function.
- This function is rebuilt with the hole-labelled image as marker.
- Each connected component of the reconstructed image will then take the label value of only one hole contained in it (it is the maximum label of all the holes contained in the connected component). This hole can be extracted with the **generateSupMask** operator which generates a binary mask corresponding to all the pixels of which their value in the first image is greater than or equal to their value in the second one. The union of this mask and the original image allows to fill one and only one hole in each connected component (see the figure below). Then, connected components with one hole are now simply connected and can be extracted with the procedure described previously.



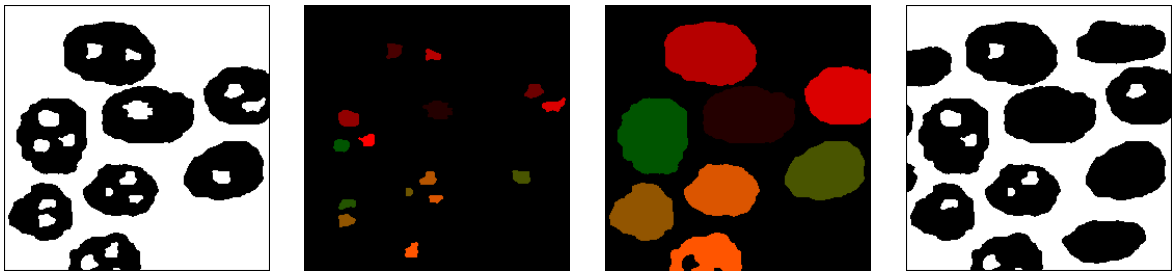
On the left, a particle contains three holes with three different labels. On the right, result of the geodesic reconstruction of the initial image (particle with its holes filled) by the labelled image. Only one hole share the same label. Pixels with the same value in the two images correspond to the mask of the hole.

The **closeOneHole** operator is defined below:

```
def closeOneHole(imIn, imOut):
    """
    This procedure allows to close one and only one hole in each connected component
    of the binary image imIn. When a connected component has no hole, it remains
    unchanged in the final image stored in imOut.
    This algorithm requires a single level of homotopy in the initial image.
    """

    imWrk0 = imageMb(imIn, 32)
    imWrk1 = imageMb(imIn, 32)
    imWrk2 = imageMb(imIn, 32)
    imWrk3 = imageMb(imIn)
    imWrk4 = imageMb(imIn)
    # The holes are filled in imWrk3 and extracted in imWrk4.
    closeHoles(imIn, imWrk3)
    diff(imWrk3, imIn, imWrk4)
    # The holes are labelled.
    nb = label(imWrk4, imWrk0)
    # The image with filled holes is converted in 32-bit.
    convertByMask(imWrk3, imWrk1, 0, computeMaxRange(imWrk1)[1])
    # Geodesic reconstruction of the label image.
    copy(imWrk0, imWrk2)
    build(imWrk1, imWrk2)
    # The holes with same label as the built image are extracted...
    generateSupMask(imWrk0, imWrk2, imWrk4, False)
    logic(imWrk4, imWrk3, imWrk3, "inf")
    # ... and filled.
    logic(imIn, imWrk3, imOut, "sup")
```

Test it with the *gruyere* image loaded in imbin1:

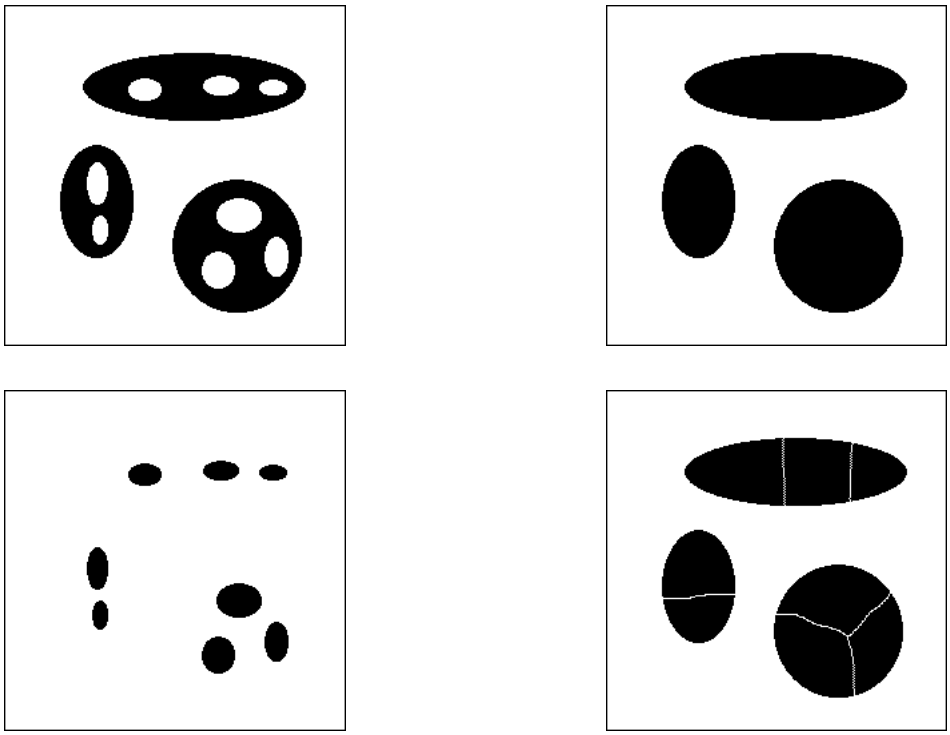
```
>>> closeOneHole(imbin1, imbin2)
```



From left to right: particles containing at least one hole, labelled holes, reconstruction of the filled particles with the label image; this reconstruction assigns to each particle the label of only one hole included in the particle. Extracting pixels with identical values in the label image and in the reconstruction produces a mask of the holes to be removed. The result of the filling of a single hole is given in the right image.

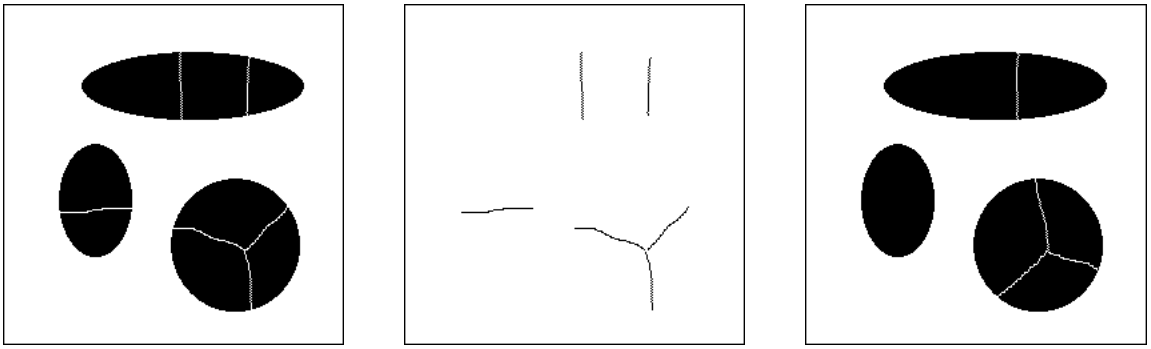
3) Extracting particles with only two holes

Separating particles with two holes from those with more than two holes is even more complex when an approach based on homotopic thinnings is used. However, a solution using the geodesic SKIZ can be designed once again. Let us consider a set made of particles with two holes or more. Let us fill in the holes and perform the geodesic SKIZ of the holes in the geodesic space made of this set without holes. The result obtained with the *gruyere* image would prompt to use the multiple points of the geodesic SKIZ as markers ot particles with more than two holes. But this would be an error as shown in the following example with the *two_three_holes* image.



The geodesic SKIZ of the holes does not produce necessarily multiple points when the number of holes is greater than 2 as illustrated in this example. Upper left: initial image. Upper right: geodesic space (initial image without holes). Lower left: holes. Lower right: geodesic SKIZ of the holes in the geodesic space.

The geodesic SKIZ may be without multiple points when the particle contains three holes. However, when the possible multiple points are removed, only particles with one and only one hole contain a single arc in the geodesic SKIZ.



Previous geodesic SKIZ (left), boundaries of the SKIZ without their multiple points (middle), geodesic SKIZ of the simple arcs (right). Only the particles with two holes are marked by a single arc.

So, we just have to perform the geodesic SKIZ of the preceding arcs. When a single arc is present in a connected component, the geodesic SKIZ is empty. On the contrary, it is not the case when more than two holes are present. This last geodesic SKIZ can therefore be used as a marker for reconstructing particles with more than two holes. Note also that this algorithm does not work when the holes organisation is complex with an homotopy tree containing more than one branch.

The list of the successive operations for obtaining the awaited result is given below. Remember that the procedure must be applied on an initial image containing particles with at least two holes. Load this image in imbin1 (this image can be obtained by applying on the *gruyere* image the solution described in the previous section) and enter the following commands:

```
>>> closeHoles(imbin1, imbin2)
>>> diff(imbin2, imbin1, imbin3)
```

The holes are extracted and stored in imbin3.

```
>>> geodesicSKIZ(imbin3, imbin2, imbin4)
>>> diff(imbin2, imbin4, imbin3)
```

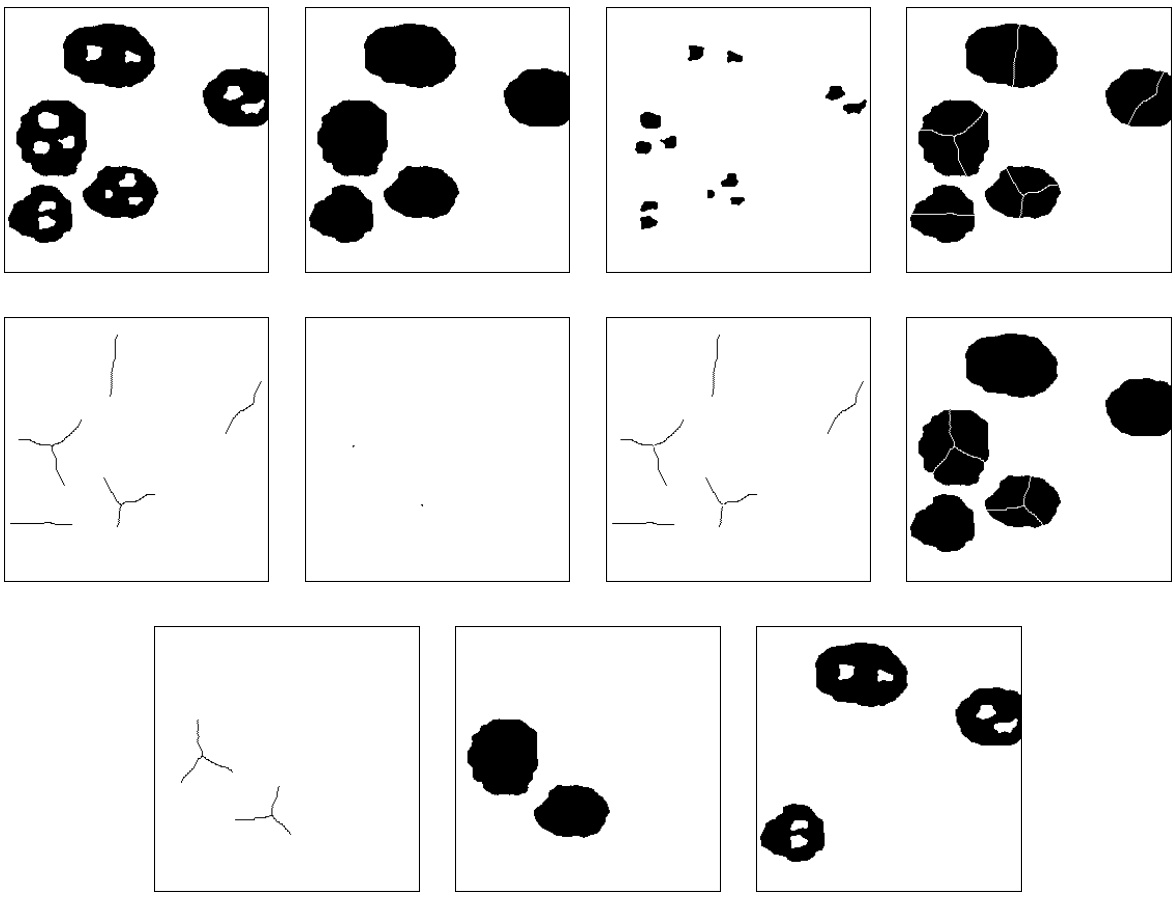
The geodesic SKIZ of the holes is performed in the geodesic space defined by the particles without their holes. The boundaries are stored in imbin3.

```
>>> multiplePoints(imbin3, imbin4)
>>> diff(imbin3, imbin4, imbin4)
```

The multiple points are extracted and removed. imbin4 contains only simple arcs.

```
>>> geodesicSKIZ(imbin4, imbin2, imbin3)
>>> diff(imbin2, imbin3, imbin3)
>>> build(imbin2, imbin3)
>>> diff(imbin1, imbin3, imbin3)
```

The boundaries of the geodesic SKIZ of these simple arcs are used to rebuild the particles which contain more than two holes. Finally, imbin3 contains particles with only two holes.



From left to right and from top to bottom: initial image (particles with at least two holes), particles without holes, holes, geodesic SKIZ of the holes, boundaries of the SKIZ, multiple points, simple arcs of the SKIZ, second geodesic SKIZ of the simple arcs, boundaries of the second SKIZ, reconstructed particles (particles with more than two holes), final result (particles with two holes).

The approach based on geodesic reconstructions is, a contrario, easy as we just need to iterate twice the closeOneHole operator. Particles without holes after this iteration contained initially two holes.

4) Designing an algorithm able to extract any particle with n holes or more
Remind that this algorithm already exists. It has been defined at the beginning of this exercise. It is the **holesSieving** operator. Fortunately, a faster solution exists. This solution works for all kinds of sets, even when their corresponding homotopy tree is complex. It is based on the **measureLabelling** operator which has already been used to label each connected component of a set with its area (see exercise n°3, chapter 6). This operator labels each connected component with a value equal to the number of points of another set contained in the connected component.
With this procedure, we can label each connected component of a set with the number of holes contained in this component. We know that the number of holes n_T of a connected component is linked to its connectivity number ν by the following relation:

$$n_T = 1 - \nu$$

In order to avoid to label connected components without hole with the 0 value, the labelling will be made with the value $n_T + 1 = 2 - v$.
On the hexagonal grid (the whole exercise is solved on this grid only), the connectivity number is obtained by counting the number of $\begin{matrix} 0 & 0 \\ & 1 \end{matrix}$ and $\begin{matrix} & 0 \\ 1 & 1 \end{matrix}$ configurations in the connected component according to the formula:

$$v = n\begin{pmatrix} 0 & 0 \\ & 1 \end{pmatrix} - n\begin{pmatrix} & 0 \\ 1 & 1 \end{pmatrix} = n_1 - n_2$$

The points presenting these two configurations can be extracted with a hit-or-miss transformation. The two sets obtained with these two hit-or-miss transforms can be used to label the initial set X with the values n_1 and n_2 correponding to each connected component with the ***measureLabelling*** operator. Then, the two label images can be subtracted and the value 2 can be added to each non zero value to obtain a new label image where each connected component will take a value equal to the number of holes contained in it plus 1:

$$n_T + 1 = 2 + n_2 - n_1$$

The procedure named **holesLabelling** is the following:

```
def holesLabelling(imIn, imOut):
  """
  Labels each particle in imIn with a value equal to its number of holes +1.
  The result is put in 32-bit image imOut.
  """

  # Working images.
  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn, 32)
  # Structuring elements
  dse1 = doubleStructuringElement([1,6],[0],HEXAGONAL)
  dse2 =  doubleStructuringElement([1],[0,2],HEXAGONAL)
  # Initializing the label image with 2.
  convertByMask(imIn, imOut, 0, 2)
  # Determining the 2nd configurations in the connectivity number calculation.
  # and adding their number to the label image.
  hitOrMiss(imIn, imWrk1, dse2)
  measureLabelling(imIn, imWrk1, imWrk2)
  add(imOut, imWrk2, imOut)
  # Determining the 1st configurations and subtracting them to get the
  # number of holes + 1.
  hitOrMiss(imIn, imWrk1, dse1)
  measureLabelling(imIn, imWrk1, imWrk2)
  sub(imOut, imWrk2, imOut)
```

Test this operator on the noise image, loaded in imbin1:

```
>>> holesLabelling(imbin1, im32_1)
```

Then, threshold the label image at the value taken by the particle in the middle of the image. This value is equal to 485 (move the mouse cursor over it):

```
>>> threshold(im32_1, imbin2, 485, 485)
```

Compute the connectivity number:

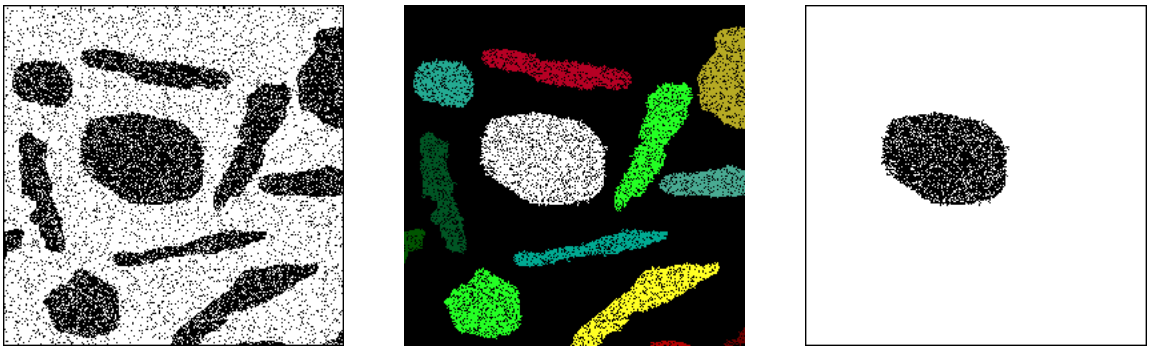
```
>>> computeConnectivityNumber(imbin2)
-483L
```

We have:

$$v = 1 - n_T = -483$$

Therefore the number of holes is equal to 484 and the label value is 485.

Note that the origin of the structuring elements have been chosen judiciously so that the result of the hit-or-miss transform be always included in the initial set.



Initial noise image (left), labelling of the connected components by their number of holes + 1 (middle), Extraction by thresholding of the connected component with a label equal to 485 (right). Its connectivity number is equal to -483. This connected component contains 484 holes.

5) This transformation is obviously anti-extensive (there are no more particles in the transformed set than in the initial one) and idempotent (all the connected components of $\Psi_n(X)$ contains at least n holes and no particle is removed when the operator is applied again). But this transformation is not increasing. Indeed, let us consider a connected component X_1 containing n holes and another connected component X_2 containing X_1 but without hole (X_2 can be equal to X_1 without its holes). We have then:

$$X_1 \subset X_2$$
$$\psi_n(X_1) = X_1 \text{ and } \psi_n(X_2) = \varnothing$$

So:

$$\psi_n(X_1) \supset \psi_n(X_2)$$

This transformation is not an opening.

Now, it is only a matter of perspective... If we consider that the elements of the working space are not somposed of pixels but of connected components, a subset X of this space is defined as a non ordered sequence of connected components X_i. A set X_1 in this space is included in a set X_2 if and only if all the connected components belonging to X_1 belong also to X_2. In this (infinite) space made of all the possible connected components, ψ_n is increasing, anti-extensive and idempotent. It is an opening... Moreover, it is a granulometry (size distribution). We have:

$$\forall m > n,\ \psi_m \circ \psi_n = \psi_n \circ \psi_m = \psi_{\sup(n,m)} = \psi_m$$

Exercise n° 10: Extremities of particles

1) Load the *eutectic* image in imbin1 and type:

```
>>> geodesicCenter(imnin1, imbin2)
```

You can also use **thinD**:

```
>>> thinD(imbin1, imbin2)
```

When the connected components are elongated, there is not a great difference between the positions of the geodesic centers and the centroids.

Geodesic centers of the eutectic image (left), centroids (right).

This difference is more important when the connected component is less elongated as it is the case for the *hand* image. Try the two operators. You can define new binary images imbinA and imbinB, their size been equal to the size of the *hand* image:

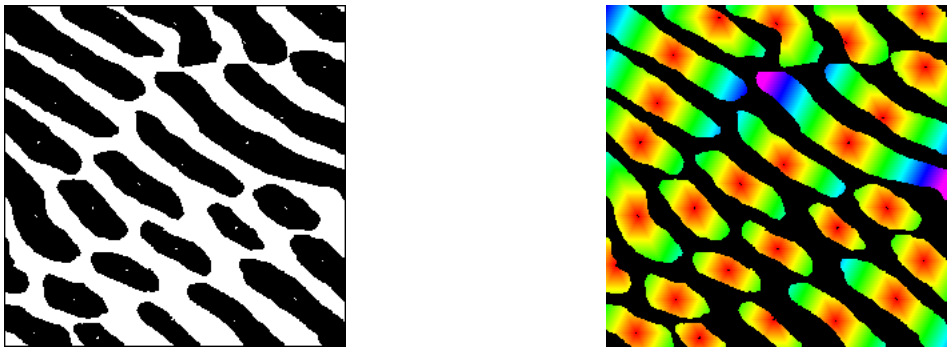
```
>>> imbinA = imageMb(256, 154, 1)
>>> imbinB = imageMb(imbinA)
```

Geodesic center of the hand image (left), centroid (right). Due to the palm of the hand, the position of the two centers is quite different.

2) The **geodesicDistance** operator exists in MAMBA. Enter the following commands (the initial image is in imbin1, the geodesic center in imbin2):

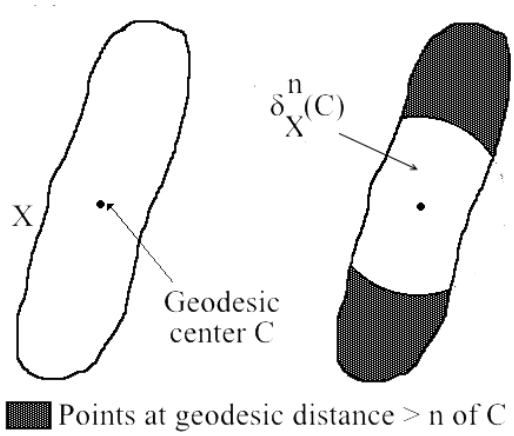
```
>>> diff(imbin1, imbin2, imbin2)
>>> geodesicDistance(imbin2, imbin1, im32_1)
```

The geodesic distance is stored in a 32-bit image. The maximal values of this geodesic distance correspond to the extremities of the particles.

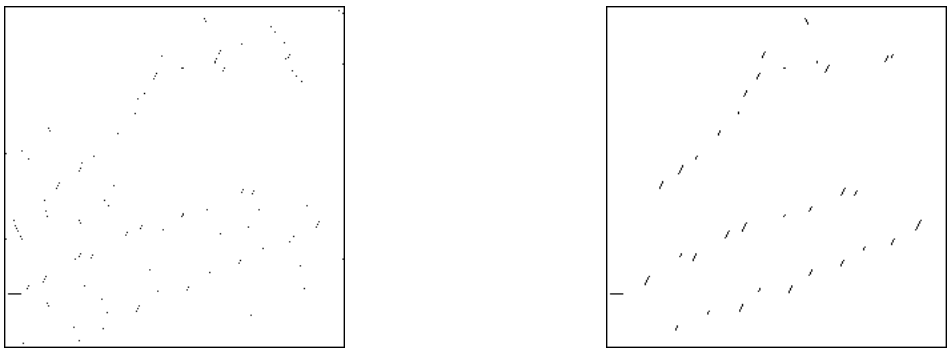


Initial set made of the original image without the geodesic centers (left), geodesic distance of this set (right, rainbow palette used).

3) The extremities are the maxima of the geodesic distance function. In fact, extremities correspond to the ultimate geodesic erosion of the initial set minus its geodesic centers (see exercise n°5, chapter 7).



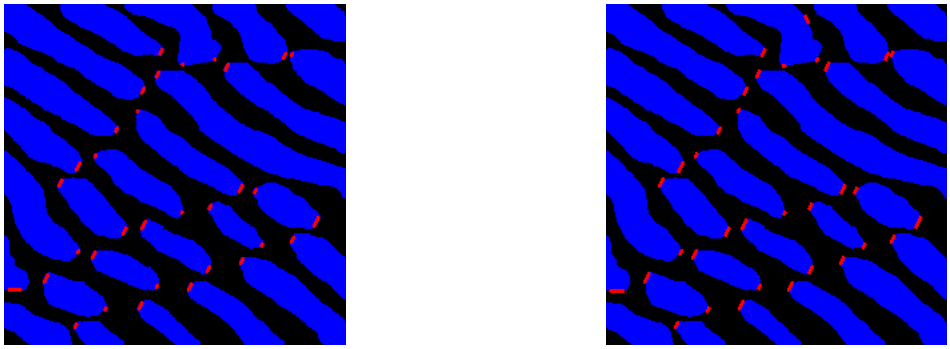
The successive geodesic erosions of the set X\C (or geodesic dilations of C) correspond to successive levels of the geodesic distance function.



Extremities of the eutectic image (maxima of the geodesic distance) on the left. On the right, extremities of the same image after an opening.

The result, however, is noisy, due to the fact that the boundaries of the set are often not smooth enough. So, a good practice consists in applying the same algorithm on the opened image. An elementary opening is, most of the time, sufficient to reduce the noise.

It is also possible to open the geodesic distance function before extracting its maxima. The following images show the effect of these two filterings applied to the *eutectic* image.



Detection of the extremities of the opened eutectic image (right) compared to the detection by extracting the maxima of the opened geodesic distance function (left).

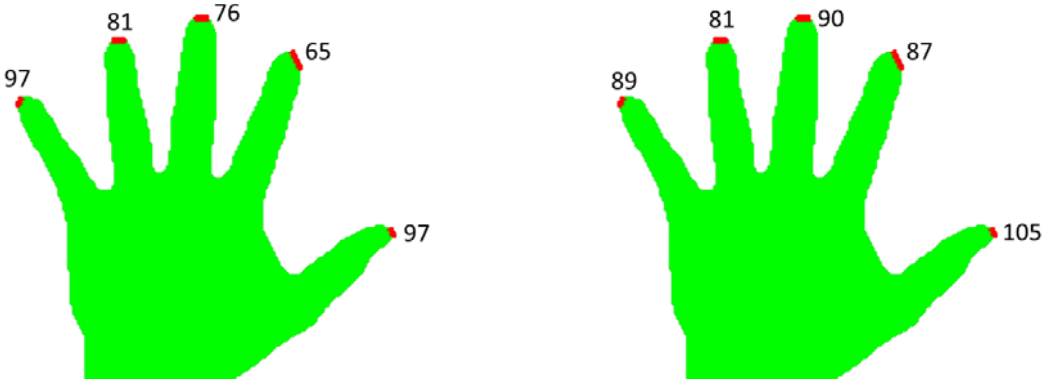
An operator named **extremities** can be defined to extract the extremities of the imIn image. These extremities are stored in the 32-bit image imOut1. Each one is valued with its geodesic distance to the geodesic center. The 32-bit image imOut2 contains the geodesic distance function. The parameter **InnerParticles** indicates if the connected components are considered to be completely included in the image field (it is then set to True and the extremities of the particles touching the edge are prexerved) or not (it is set to False - its default value- and the particles are supposed to extent outside the image window).

```
def extremities(imIn, imOut1, imOut2, innerParticles=False):
    """
    This operation performs the computation of the extremities (maxima of a
    geodesic distance function) and puts the result in imOut.
    'imIn' must be a binary image, 'imOut1' is a 32-bit image containing the
    extremities and their geodesic distance to the centroid (allowing thus to
    sort them according to their distance to the center) and imOut2 a 32-bit
    image containing the entire geodesic distance from the geodesic centers.
    If 'innerParticles' is set to False(default), the particles touching the
    edge are considered as extending outside the image window. Therefore, no
    extremity is detected on the edge of the image. If 'innerParticles' is True,
    all the particles are supposed to be included in the image, so extremities
    may appear on the edge.
    """

    imWrk1 = imageMb(imIn)
    # Computation of the geodesic centers.
    geodesicCenter(imIn, imWrk1)
    # These centers are removed from the initial image (each particle contains
    # a hole).
    diff(imIn, imWrk1, imWrk1)
    # Computing the geodesic distance
    geodesicDistance(imWrk1, imIn, imOut2)
    # The extremities correspond to the maxima.
    maxima(imOut2, imWrk1)
    # Extremities on the edge are removed if 'innerParticles' is set to False.
    if not(innerParticles):
        removeEdgeParticles(imWrk1, imWrk1)
    # The extremities are given their corresponding distance to the center.
    convert(imWrk1, imOut1)
    logic(imOut2, imOut1, imOut1, "inf")
```

It is possible to use the centroid (obtained by the thinD operator) instead of the geodesic center to generate the geodesic distance and extract its maxima. The extremities obtained from these centroids are often not different from the extremities produced with the geodesic center, even when their respective locations are different. You can verify this with the *hand* image:

Extremities (slightly dilated) of the hand image. The result is the same if the geodesic center or the centroid is used.

However, if we consider the values of the geodesic distance on these extremities, they are far from been the same, as shown in the following figure. It is likely that at least two extremities should be at the same distance from the center point. These extremities correspond to the farthest extremities of the set. It is actually the case when the geodesic center is used (extremities of the thumb and the little finger in the *hand* image). But it is not true when the centroid is used.

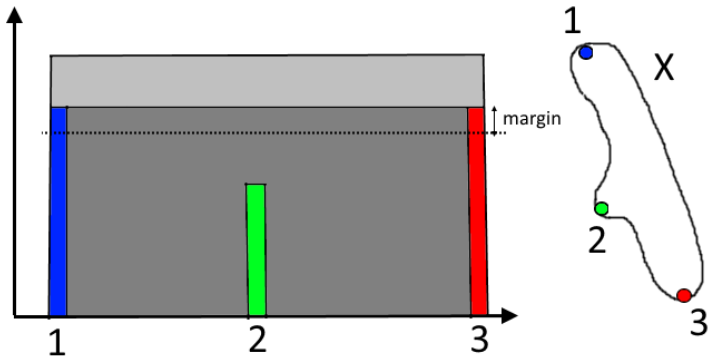


Left image: values at the extremities of the geodesic distance from the geodesic center. Right image: values of the geodesic distance from the centroid.

Exercise n° 11: Dislocations in eutectics

This exercise is interesting as it shows that it is possible to build, by an appropriate sequence of operations, features which are not visible at the first sight.
1) We saw in the previous exercise how to extract the extremities of particles. However, as illustrated in the *hand* image, these extremities are not necessarily the farthest ones, that is the extremities at the greatest distance from the geodesic center. We need, for solving the current problem, to extract these farthest extremities. To do this, we design a new operator named **extremePoints**. This operator is based on the same approach as the one used to close one and only one hole in exercise n° 9 (**closeOneHole** procedure) . The extremities of the particles are extracted and valued as explained previously. Then, they are used to rebuild by a

geodesic reconstruction the particles which are, by this means, valued with the distance from the geodesic center of the farthest extremities. These extremities are extracted with the **generateSupMask** operator masking the extremities which have te same value as the rebuilt image.



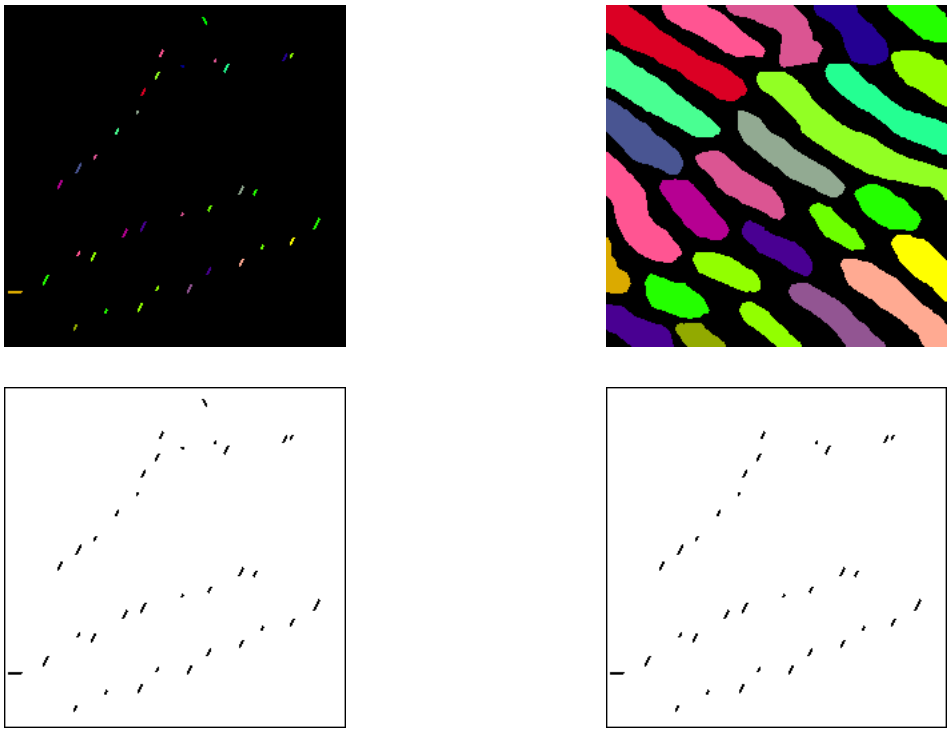
The particle X has 3 extremities. In light grey, indicator function of X. In dark grey, result of the geodesic reconstruction of the indicator function by the valued extremities. The extremities 1 and 3 are at the same height as the reconstructed image. The margin parameter allows to select extremities at a lower height.

The operator, named **extremePoints**, is the following:

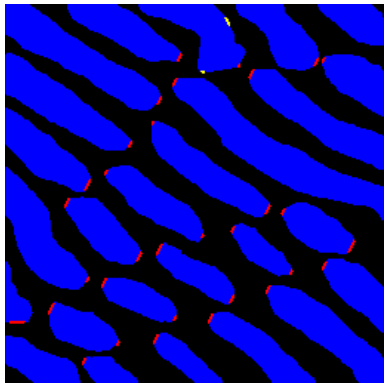
```
def extremePoints(imIn, imOut, margin=0):
    """
    This operator is a refinement of the 'extremities' operator. It determines
    points of the connected components of the binary set 'imIn' which are the
    farthest extremity points. The 32-bit image 'imOut' contains these extreme
    points valued with their distance to the geodesic center. 'margin' is a
    parameter which allows to take into account an extremity even if its distance
    to the center is not maximal, provided that it is not lower than this
    maximal value minus 'margin'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn, 32)
    imWrk3 = imageMb(imIn, 32)
    imWrk4 = imageMb(imIn, 32)
    geodesicCenter(imIn, imWrk1)
    diff(imIn, imWrk1, imWrk1)
    geodesicDistance(imWrk1, imIn, imWrk2)
    maxima(imWrk2, imWrk1)
    convert(imWrk1, imWrk3)
    logic(imWrk2, imWrk3, imWrk3, "inf")
    convert(imIn, imWrk4)
    copy(imWrk3, imOut)
    hierarBuild(imWrk4, imOut)
    floorSubConst(imOut, margin, imOut)
    generateSupMask(imWrk3, imOut, imWrk1, False)
    logic(imWrk1, imIn, imWrk1, "inf")
    removeEdgeParticles(imWrk1, imWrk1)
    convert(imWrk1,imWrk3)
    logic(imWrk2, imWrk3, imOut, "inf")
```

The result of the operation is stored in in 32-bit image which contains the farthest extremities of the particles, valued with their respective distance to the geodesic center. The extremities touching the edge of the image have been removed.

Note that a parameter, **margin**, is used in this operator. Its purpose is to cope with possible differences of valuation of the farthest extremities due to parity biases. This parameter is set to 0 by default.



From top to bottom and from left to right: valued extremities of the eutectic image, labelling of the lamellae by the greatest geodesic distance of the points from the geodesic center (corresponds to half the length of the particle), extremities and extreme points.



Extreme points of the eutectic image in red. In yellow, extremities which are not farthest points.

As the difference between the two sets is not obvious, the image below shows a colored superposition of the extremities and the extreme points.

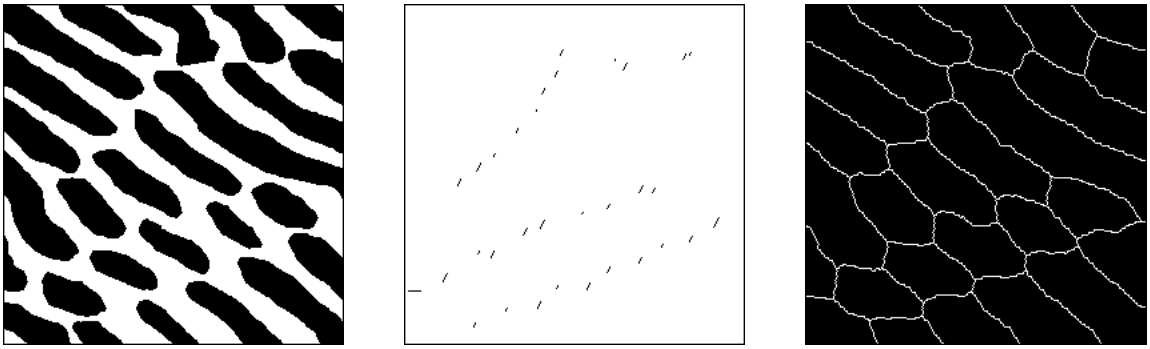
2) Two different procedures will be described to extract the dislocations. The first one tries to connect the extreme points to build lines corresponding to the dislocations. This connection

uses dilations. To achieve this, two problems must be addressed: avoiding to connect extreme points belonging to the same lamella and determining the size of these dilations.
Let us describe the different operations used in this first procedure. The initial *eutectic* image is stored in imbin1 and a small opening is performed in order to smooth the boundaries. Then the extreme points are extracted, stored in imbin2, and valued by their distance to the geodesic center in im32_1:

```
>>> opening(imbin1, imbin1)
>>> extremePoints(imbin1, im32_1)
>>> threshold(im32_1, imbin2, 1, computeMaxRange(im32_1)[1])
```

Then, the influence zones of the lamellae are built in imbin3:

```
>>> computeSKIZ(imbin1, imbin3)
```



Original opened image (left), extreme points (middle) and influence zones of the lamellae (right).

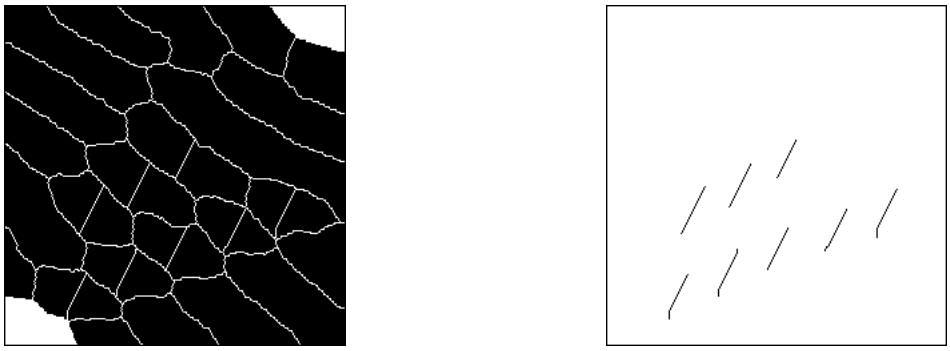
The next step consists in defining a separation zone between the extreme points belonging to the same lamella. First of all, a median boundary is obtained by a geodesic SKIZ of the extreme points inside the lamella. This median boundary is extracted and the artefacts due to edge effects are removed. The result is put in imbin4:

```
>>> geodesicSKIZ(imbin2, imbin3, imbin4)
>>> diff(imbin3, imbin4, imbin4)
>>> removeEdgeParticles(imbin4, imbin4)
```

Then, we would like to perform geodesic dilations of these boundaries inside the influence zones, the size of each dilation being proportional to the length of the influence zone. To achieve this, we define a new operator, named **geodesicAdaptiveDilate** which perform a geodesic dilation of each point of a 32-bit image, its size being given by its value in the image. The final result is a binary image made of the union of all these dilations. The procedure is defined as follows:

```
def geodesicAdaptiveDilate(imIn, imMask, imOut):
  """
  This operator performs a binary adaptive dilation. The 32-bit image
  'imIn' indicates for each pixel the size of the geodesic dilation (by the
  default structuring element) which will be applied on it. The geodesic
  mask is defined by the binary image 'imMask'. The result of the dilation is
```

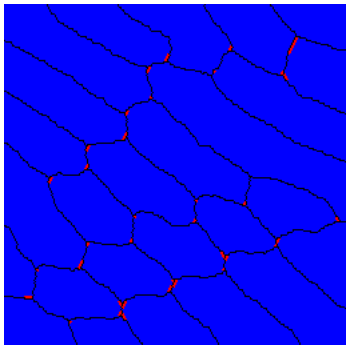
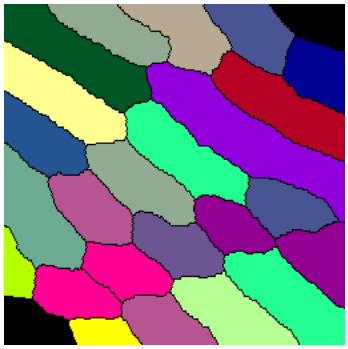
```
put in the binary image 'imOut'. It is called adaptive because its size is
given locally for each pixel by the value of this pixel in 'imIn'.
"""

imWrk1 = imageMb(imIn)
imWrk2 = imageMb(imIn)
imWrk3 = imageMb(imIn)
convert(imMask, imWrk1)
copy(imIn, imWrk2)
v1 = 0
v2 = computeVolume(imWrk2)
# At each step, the dilated image is decreased. So each pixel value
# indicates how many steps of dilation remain. When the image volume
# does not change, the process is finished.
while v2 > v1:
    v1 = v2
    geodesicDilate(imWrk2, imWrk1, imWrk3)
    floorSubConst(imWrk3, 1, imWrk3)
    logic(imWrk2, imWrk3, imWrk2, "sup")
    v2 = computeVolume(imWrk2)
threshold(imWrk2, imOut, 1, computeMaxRange(imIn)[1])
```



New boundaries in the influence zones (left), the same boundaries after their extraction and the removal of the edge artefacts (right).

The separation lines are given a value equal to ¼ of the length of the corresponding influence zone. Therefore, the adaptive dilation will partition the zone into three regions: two of them correspond to the extreme points, their length been equal to ¼ of the total length and the middle region with a length equal to ½ of the total length. For that, the influence zones are valued with the valuation of their extreme points. This valuation is also equal to half their length. The result is stored in the 32-bit image im32_3 (which has been created before):

```
>>> im32_3 = imageMb(im32_1)
>>> convert(imbin3, im32_2)
>>> extremePoints(imbin3, im32_3)
>>> hierarBuild(im32_2, im32_3)
```

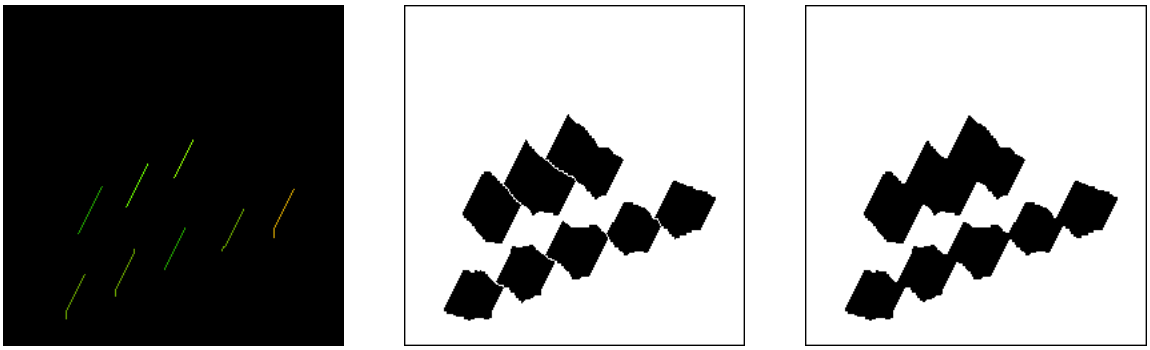
Left image: extreme points (in red) of the influence zones (in blue). Right image: labelling of the influence zones with half their respective length.

Then, this valuation is divided by 2 (each influence zone has a value equal to ¼ of its length) and the corresponding boundaries are given this valuation, stored in the previously defined 32-bit image im32_4:

```
>>> im32_4 = imageMb(im32_1)
>>> divConst(im32_3, 2, im32_3)
>>> convert(imbin4, im32_4)
>>> logic(im32_3, im32_4, im32_4, "inf")
```

The geodesic adaptive dilation of logic the im32_4 image is performed. The result, stored in the previously defined binary image imbin5, is closed in order to connect the adjacent connected components and opened to smooth the result:

```
>>> imbin5 = imageMb(imbin1)
>>> geodesicAdaptiveDilate(im32_4, imbin3, imbin5)
>>> closing(imbin5, imbin5)
>>> opening(imbin5, imbin5)
```



Valued inside boundaries (left), their geodesic adaptive dilation (middle), final separation zones after closing and smoothing (right).

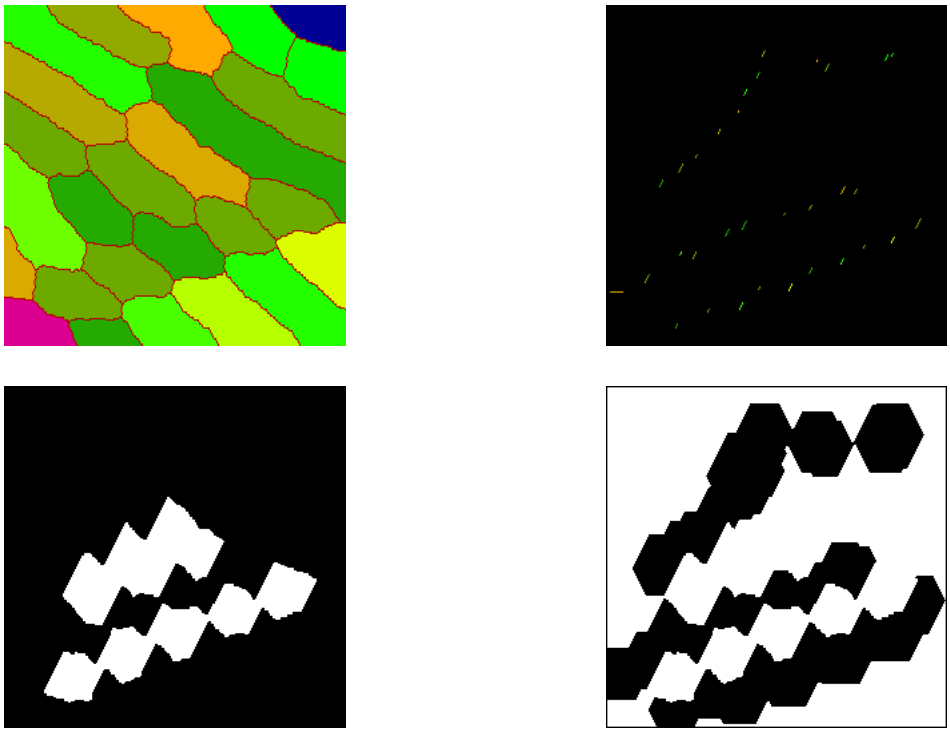
We must now define the size of the dilations which will be applied to the extreme points. This size will be equal to half the thickness of the influence zones of the lamellae +2, so as to be sure that these dilations will reach the influence zones of the neighbor points. The ½ thickness is equal to the maximum of the distance function. The following operations produce this valuation stored in the im32_3 image:

```
>>> computeDistance(imbin3, im32_3, edge=FILLED)
>>> hierarBuild(im32_2, im32_3)
```

```
>>> addConst(im32_3, 2, im32_3)
```

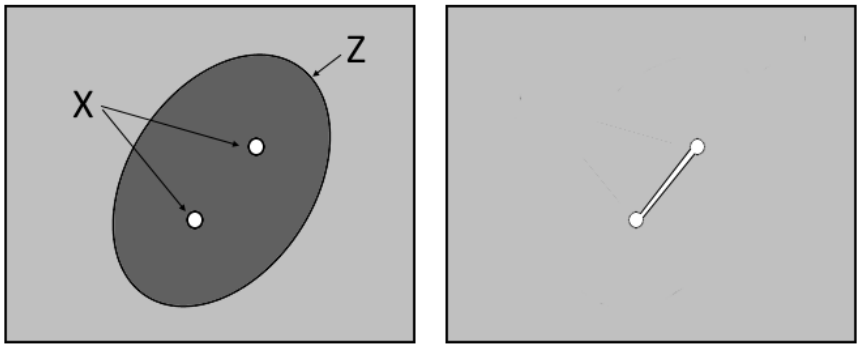
Note that the distance function is computed with the edge set to **FILLED** in order to avoid errors in the calculation of the thickness of the lamellae touching the edge. Then, we can value each extreme point with this image and perform their geodesic adaptive dilation, the geodesic space being the outside of the separation region defined previously in imbin5. The result can be put in imbin4, after a small closing connecting the close connected components:

```
>>> convert(imbin2, im32_4)
>>> negate(imbin5, imbin5)
>>> logic(im32_4, im32_3, im32_4, "inf")
>>> geodesicAdaptiveDilate(im32_4, imbin5, imbin4)
>>> closing(imbin4, imbin4)
```



Top left: influence zones valued with half their thickness + 2. Top right: extreme points valued with the previous image. Bottom left: geodesic space used for the adaptive dilation (the separation zones are excluded). Bottom right: result of the adaptive dilation after closing.

Various possibilities are available to connect the extreme points and to draw the dislocation lines. The method proposed here uses the **fullGeodesicThin** operator applied with homotopic structuring elements. The following figure explains how it works.



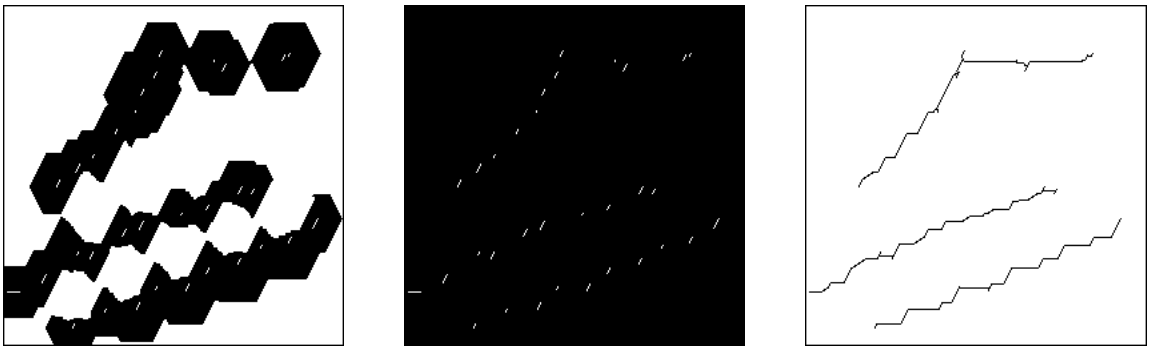
Consider the set X made of two connected components. The geodesic space is the entire field minus X. The set to be thinned is the set Z\X. Performing a full geodesic thinning using all the directions of the homotopic structuring elements until idempotence produces a connection between the two initial connected components.

Let us apply this operator. First, the set Z is built. It corresponds to the previous dilation (in imbin4) minus the extreme points (in imbin2). Moreover, in order to avoid edge effects during the geodesic thinning, the connected components touching the edge are separated from this edge by removing a small border around the image (border generated in imbin6 with a dilation of an empty set!). The geodesic space is simply the complementation of imbin2. Here are the operations:

```
>>> imbin6.reset()
>>> dilate(imbin6, imbin6, edge=FILLED)
>>> diff(imbin4, imbin6, imbin6)
>>> negate(imbin2, imbin3)
>>> diff(imbin6, imbin2, imbin5)
```

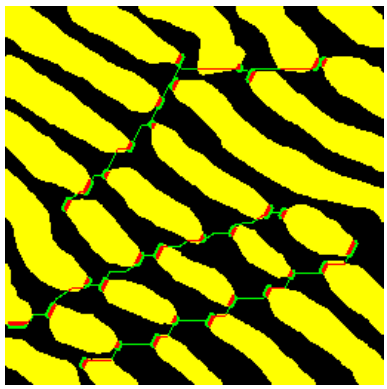
Then a full geodesic thinning is performed. Note that, for obtaining thin dislocation lines, a thinning with D must be followed by a thinning with M (a more elegant way of doing would consist in defining a full geodesic thinning where these structuring elements are used in parallel). Finally, the initial extreme points are added to the result:

```
>>> fullGeodesicThin(imbin5, imbin3, imbin6, hexagonalD)
>>> fullGeodesicThin(imbin6, imbin3, imbin6, hexagonalM)
>>> logic(imbin2,imbin6, imbin6, "sup")
```



Set to be thinned made of the adaptive dilation minus the extreme points (left), geodesic space corresponding to the complementation of the extreme points (middle), result of the succesive full geodesic thinnings by the structuring elements D and M (right).

The following image shows the initial set, the extreme points and the dislocation lines.

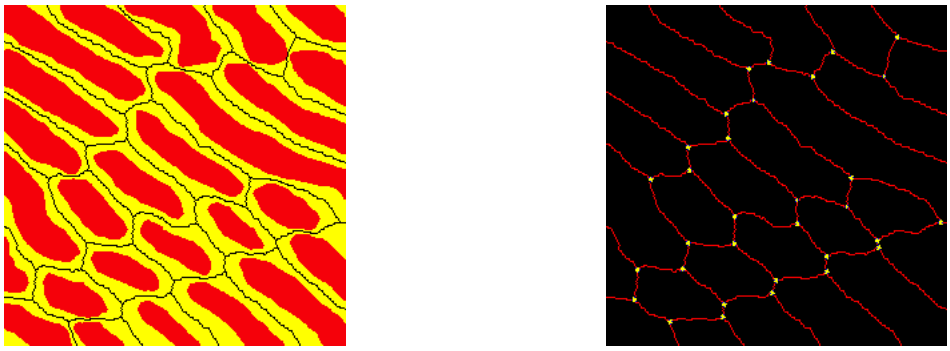


Initial eutectic image (in yellow), extreme points of the lamellae (in red) and dislocation lines (in green).

The second procedure is different and is based on the use of the skeleton by influence zones of the lamellae. We shall try to emphasize the dislocation lines by selecting among the arcs of the skeleton by influence zones those which can or cannot belong to these dislocations. Let us load the initial *eutectic* image in imbin1 and perform a small opening to smooth it. Then, we build the influence zones in imbin2, the multiple points of the skeleton in imbin3 and the simple arcs in imbin4:

```
>>> opening(imbin1, imbin1)
>>> computeSKIZ(imbin1, imbin2)
>>> negate(imbin2, imbin4)
>>> multiplePoints(imbin4, imbin3)
>>> removeEdgeParticles(imbin3, imbin3)
>>> diff(imbin4, imbin3, imbin4)
```

Note that the multiple points touching the edge are removed.



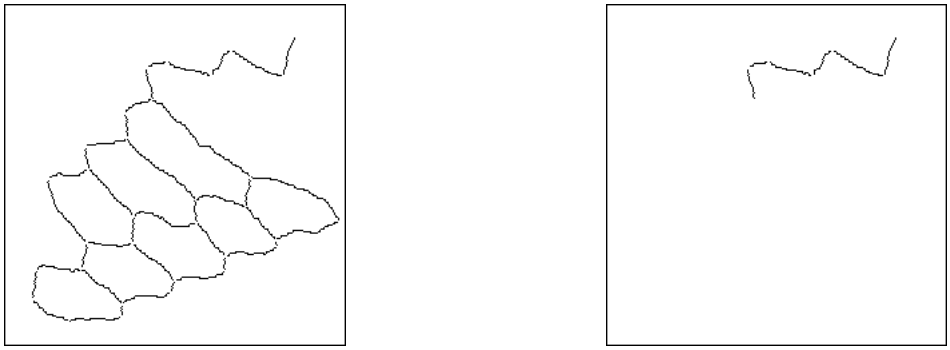
Left: lamellae of the eutectic image (red) and their influence zones (yellow). Right: simple arcs of the SKIZ (red) and multiple points (slightly dilated in yellow).

The simple arcs touching the edge are then removed because they separate lamellae which are not entirely included in the image window. Therefore, it is not possible to insure that these arcs belong to dislocations:

```
>>> removeEdgeParticles(imbin4, imbin4)
```

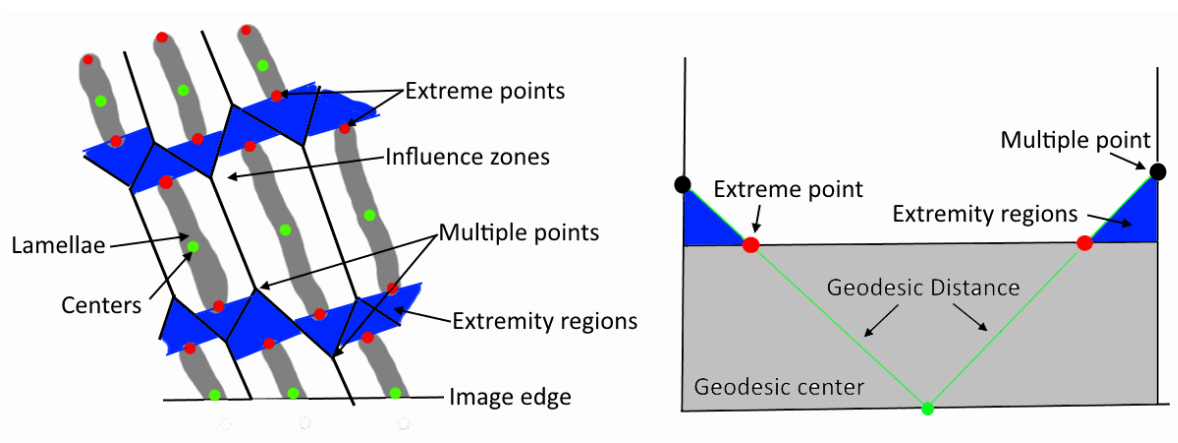
Similarly, the simple arcs which are not either surronding or inside the inner influence zones are belonging to dislocations. The inner influence zones are stored in imbin2 whilst the preserved arcs are stored in imbin5 (this working binary image has been defined previously):

```
>>> imbin5 = imageMb(imbin1)
>>> removeEdgeParticles(imbin2, imbin2)
>>> dilate(imbin2, imbin5)
>>> diff(imbin4, imbin5, imbin5)
```



Arcs touching the edge are removed (left), arcs not surrounding or inside inner influence zones are extracted, they belong to dislocation lines (right).

The next step consists in defining regions at the extremities of the inner influence zones which will be used as masks in the process of selecting arcs belonging to the dislocations. These masks are built as described in the following figure.



The geodesic centers and the extreme points of the lamellae are extracted. The geodesic distance of the centers inside the influence zones is computed. The influence zones are labelled with the extreme points and are assigned the maximal distance of these points inside each influence zone. The mask of all the points where the geodesic distance is higher than or equal to the labelled image defines the extremity regions.

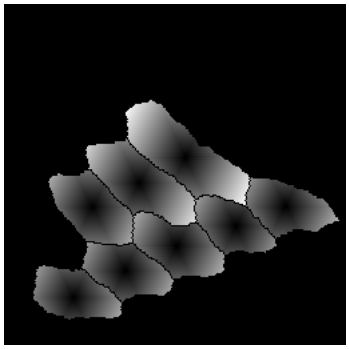
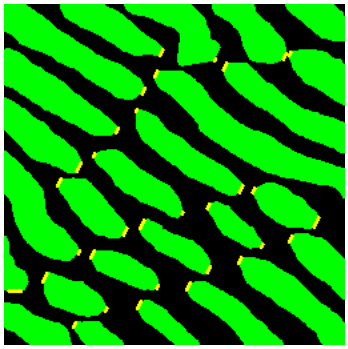
Let us perform the different steps of this construction. Another working image imbin6 is defined. The geodesic centers of the lamellae are stored in imbin6 and the geodesic distance of these centers inside the inner zones of influence is defined in im32_1:

```
>>> imbin6 = imageMb(imbin1)
>>> geodesicCenter(imbin1, imbin6)
>>> diff(imbin2, imbin6, imbin6)
```

```
>>> geodesicDistance(imbin6, imbin2, im32_1)
```

The valued extreme points of the lamellae are defined in im32_2:

```
>>> extremePoints(imbin1, im32_2)
```

Left: geodesic distance of inner zones of influence. Right: extreme points of all the particles (in yellow).

We define two new 32-bit working images im32_3 and im32_4:

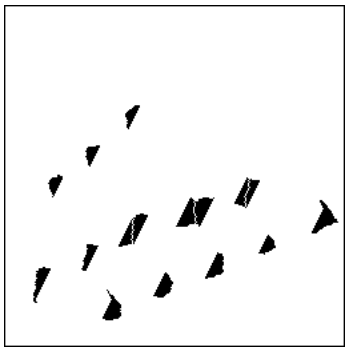
```
>>> im32_3 = imageMb(im32_1)
>>> im32_4 = imageMb(im32_1)
```

The inner influence zones are labelleded with the valued extreme points. The result is stored in im32_4:

```
>>> convert(imbin2, im32_3)
>>> copy(im32_2, im32_4)
>>> hierarBuild(im32_3, im32_4)
```

This image is compared to the geodesic distance in order to generate the extremity regions in imbin6:

```
>>> generateSupMask(im32_1, im32_4, imbin6, False)
>>> logic(imbin6, imbin2, imbin6, "inf")
```

Left image: inner influence zones labelled with the extreme points distances. Right image: the extremity regions corresponding to the pixels where the geodesic distance is higher than or equal to the labelled image.

The points adjacent of multiple points in the inner influence zones are put in a new working image imbin7:

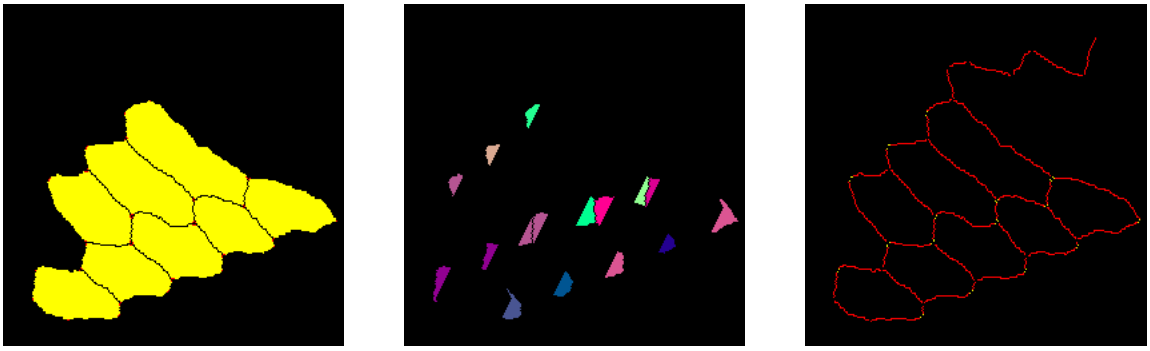
```
>>> imbin7 = imageMb(imbin1)
>>> dilate(imbin3, imbin7)
>>> logic(imbin2, imbin7, imbin7, "inf")
```

These points are valued with the geodesic distance and the result is stored in im32_1. Then, the extremity zones are labelled with the maximal value taken by the adjacent points. The result is stored in im32_3:

```
>>> convert(imbin7, im32_2)
>>> logic(im32_1, im32_2, im32_1, "inf")
>>> convert(imbin6, im32_4)
>>> copy(im32_1, im32_3)
>>> hierarBuild(im32_4, im32_3)
```

Then, the farthest point among the points adjacent to the multiple points is extracted and put in imbin2:

```
>>> generateSupMask(im32_1, im32_3, imbin2, False)
>>> logic(imbin2, imbin7, imbin2, "inf")
```



Left image: inner influence zones (yellow) and points adjacent to the multiple points (red). Middle image: labelling of the extremity regions with the maximal value of the geodesic distance on the previous points. Right image: simple arcs (red) and points adjacent to the multiple points at a maximal distance from the geodesic center (yellow).

Then, the connected components adjacent to the multiple points and containing the point(s) at maximal distance from the geodesic center are rebuilt:

```
>>> build(imbin7, imbin2)
```

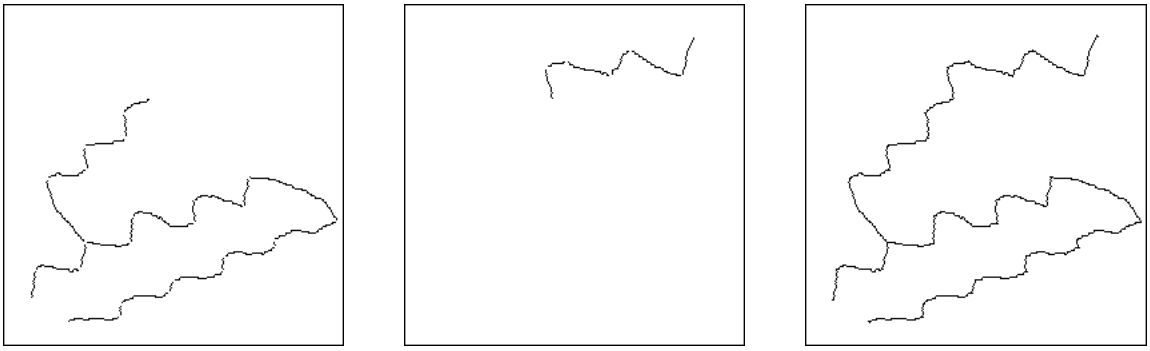
The simple arcs which are adjacent to these connected components correspond to dislocations. They are put in imbin4:

```
>>> dilate(imbin2, imbin2)
>>> build(imbin4, imbin2)
```

The arcs previously extracted and stored in imbin5 together with the multiple points are added:

```
>>> logic(imbin2, imbin5,imbin2, "sup")
```

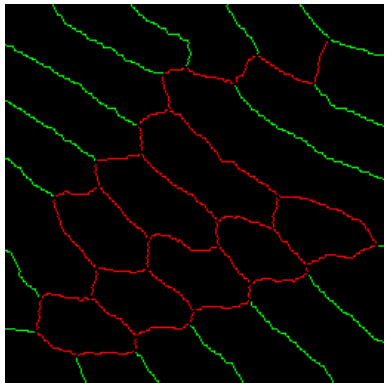
```
>>> logic(imbin2, imbin3,imbin2, "sup")
```



Left: dislocations extracted by the last procedure. Middle: dislocations previously detected. Right: final dislocation extraction.

The cells containing the stacked lamellae can also be defined. This is done by adding the boundaries connecting the edge and the end points of the dislocation lines to these dislocations. The different steps of the procedure are given below.
The simple arcs touching the edge are stored in imbin5:

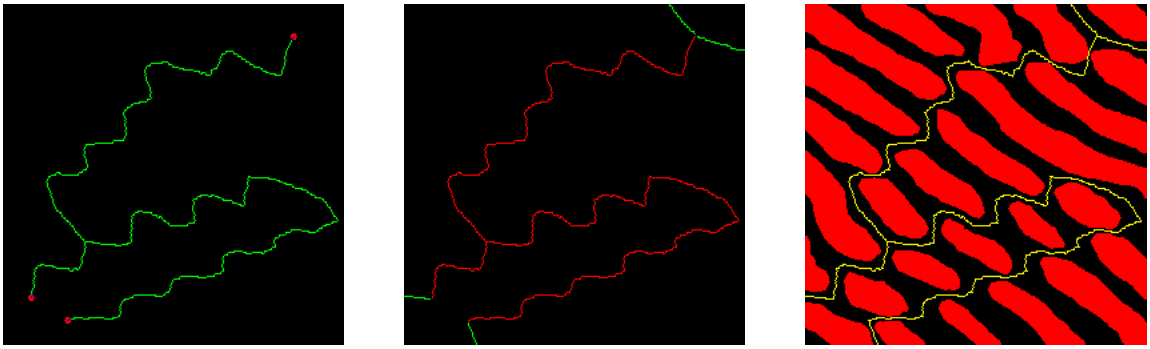
```
>>> computeSKIZ(imbin1, imbin4)
>>> negate(imbin4, imbin4)
>>> diff(imbin4, imbin3, imbin4)
>>> removeEdgeParticles(imbin4, imbin5)
>>> diff(imbin4, imbin5, imbin5)
```



Inner simple arcs in red and simple arcs touching the edge in green.

The dislocation lines stored in imbin2 are clipped (adding the multiple points may have introduced unwanted end points) and the remaining end points, corresponding to the extremities of the lines are extracted in imbin6. These end points are dilated and used to reconstruct the simple arcs connected to them and touching the edge. These reconstructed arcs are added to the dislocation lines and the result is put in imbin6:

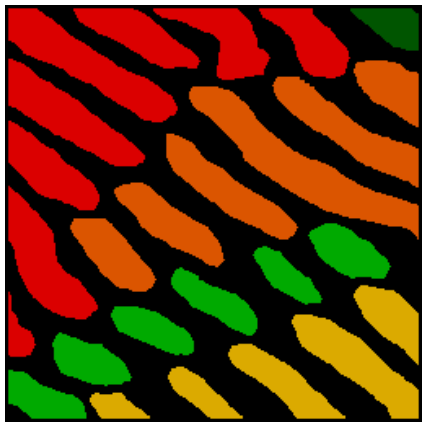
```
>>> whiteClip(imbin2, imbin6, step=1)
>>> endPoints(imbin6, imbin6)
>>> dilate(imbin6, imbin6, 2)
>>> build(imbin5, imbin6)
>>> logic(imbin6, imbin2, imbin6, "sup")
```



Left: end points (red) of the dislocations (green). Middle: the simple arcs touching the edge and connected to the end points are added. Right: boundaries of the cells containing stacked lamellae.

The lamellae belonging to the same cell can share the same label. To do this, enter the following commands:

```
>>> negate(imbin6, imbin6)
>>> erode(imbin6, imbin6, 2, edge=EMPTY)
>>> nbCells = label(imbin6, im32_1)
>>> convert(imbin1, im32_2)
>>> logic(im32_1, im32_2, im32_2, "inf")
```



Lamellae belonging to the same cells have the same color.

Note that the lamella at the upper right is ambiguous. Therefore, it has been given a different color.

REFERENCES

[1] S.Beucher: Residues
(**http://cmm.ensmp.fr/~beucher/publi/Course2008_Residues_SB_eng.pdf**)
These lecture slides introduce various binary and numerical residual transforms. Some of them have already been used in chapter 6. But this document also describe the thinnings and thickenings operators.

[2] S.Beucher: Digital skeletons in euclidean and geodesic spaces
(**http://cmm.ensmp.fr/~beucher/publi/cncskel.pdf**)

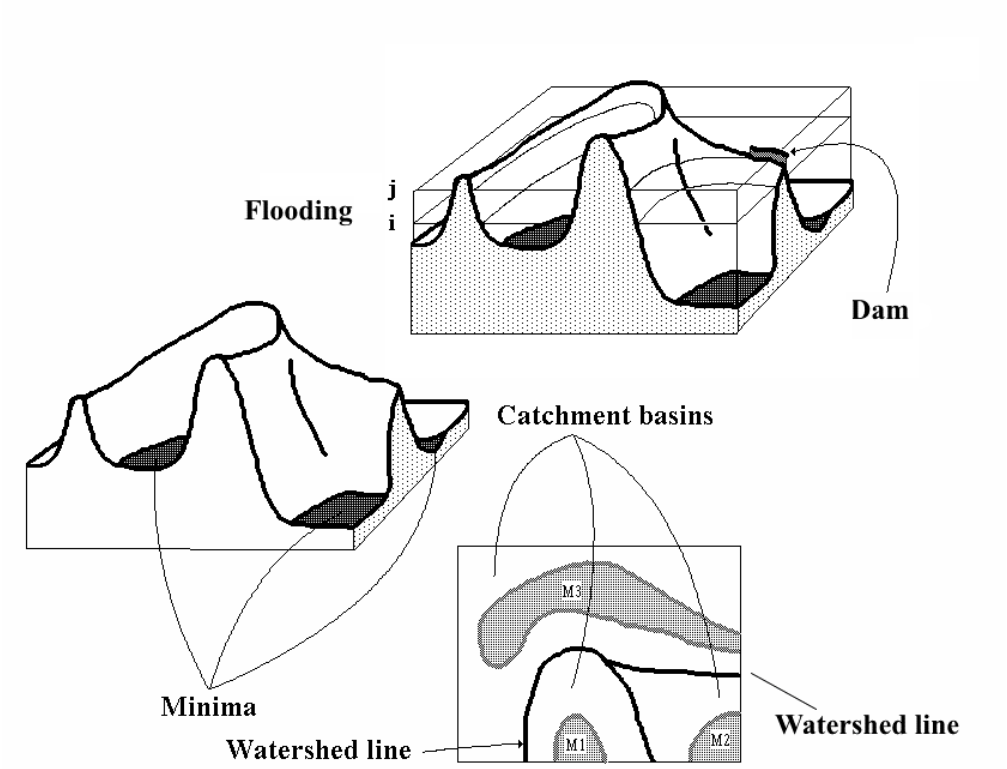
This paper describes the relationships between the skeleton by maximal balls and the homotic skeleton realised with thinnings. 3D skeletons are also addressed.

Chapter 9

SEGMENTATION

1 Watershed transformation

The watershed transformation is widely used for image segmentation. It is often applied on gradient images but not only. This transformation may be seen as a flooding process. The greytone image f is considered as a topographic surface in which holes are pierced in every regional minimum. The topographic surface is progressively plunged into water and dams are constructed each time that the waters coming from two distinct regional minima are on the point to merge. At the end of the flooding process, the dams correspond to the watershed of f, and they delimit the catchment basins of f.



The topographic surface (left image) of the initial function is flooded from its minima. Merging lakes are separated by a dam (upper right image). When the process is complete, the image is partitioned in catchment basins containing one minimum separated by the watershed lines (lower right image).

Early implementations of the watershed transform were based on floodings section by section of the image f. Modern and fast implementations (in particular the one available in MAMBA) use hierarchical queues.

2 Marker-controlled watershed

The marker-controlled watershed transform is a variant of the watershed transform where the flooding process is initiated from a set of previously defined markers instead of the minima of the function. The classical watershed transform may therefore be seen as a marker-controlled watershed where the markers are the minima. By the way, this is how this operator is realised in MAMBA.

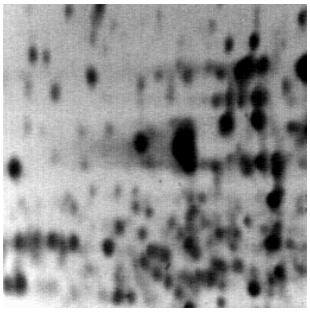
3 Hierarchical segmentation, waterfalls transformation

The waterfalls transformation is an operator based on the watershed which performs hierarchical segmentations of an image. A hierarchical segmentation extracts and merge regions of the image and builds a hierarchy where the salient regions are progressively detected. This approach is useful, in particular, when the definition of good markers to be used in a marker-controlled watershed is difficult, not to say impossible.
The MAMBA library contains different versions of these hierarchical segmentation algorithms.

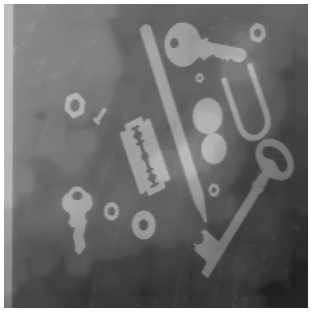
EXERCISES

Exercise n° 1

1) Practice the watershed transforms available in the MAMBA library (**watershedSegment** and **basinSegment**). Apply them to the *electrop* image and to its gradient.



electrop



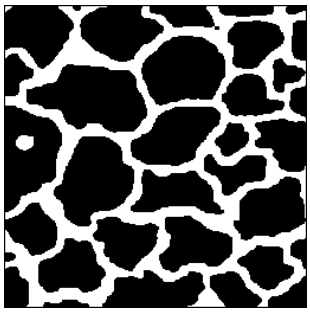
tools

2) An interactive segmentation tool is also available in MAMBA (**interactiveSegment**). Practice it (on *tools* image for instance) and try to show the influence of the position of markers on the segmentation.

Exercise n° 2: Return on the SKIZ and geodesic SKIZ operator

The SKIZ and geodesic SKIZ operators have already been introduced (see chapter 8, exercise n° 7). Until now, they were computed by means of homotopic thickening operators. However, these operators can also be defined with the watershed transform.
1) Prove that the SKIZ of a set can be obtained by a watershed with a judicious use of the watershed transform and the distance function. Do you think that computing the distance function is necessary to get the result.

2) This fast SKIZ operator is already available in MAMBA (**fastSKIZ**). Compare on the *test_skiz* and *alumine* images the result obtained by this operator with the result obtained with the classical thickening (**computeSKIZ**). What about the speed of these two operators?
3) Define similarly an algorithm for computing the geodesic SKIZ (this operator is defined in MAMBA and is named **geodesicSKIZ**). Use the *test_geodSKIZ image*. Take its holes as initial set and the image without holes as geodesic space.



alumine

test_SKIZ

test_geodSKIZ

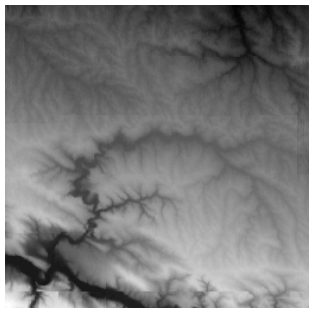
Exercise n° 3

The two-dimensional electrophoresis is a technique for separating and identifying proteins. The migration of the proteins on the gel depends on their molecular weight and on their electric charge. The purpose of this exercise is to extract the contour of each spot of proteins visible in the *electrop* image.
1) Detect the regional minima of the image. What is your conclusion? Which transformations can we apply to the image to enhance the result? Detect the new minima. From now on, we shall work on the filtered image.
2) Compute the morphological gradient (with the **gradient** operator). Detect the gradient minima. Perform the watershed transform. Is the result satisfactory?
3) Try to obtain a better result. To do so, Define markers located inside the spots and also markers for the background as well.
- What can we take as markers of the spots?
- As background markers?
Perform the watershed transform controlled by this new set of markers.

Exercise n° 4: Catchment basins in a digital elevation model

This exercise is devoted to the extraction of the catchment basins of a digital elevation model. The *relief* image represents a digital elevation model (DEM). In this model, the altitudes of a (true) topographic surface are sampled at the nodes of a regular grid. The purpose of this study is to define an algorithm for extracting the catchment basins from the topography. This may appear to be easy since it is precisely the purpose of the watershed transform available in MAMBA. However you will see that this application is more difficult than it seems because of the noise present in the image.
1) Extract the regional minima from the *relief* image. Comment the result. Segment the *relief* image into its different catchment basins and verify that one and only one catchment basin is associated to each minimum.
2) As regional minima inside the DEM cannot correspond to outlets of the hydrographic network, define a procedure to suppress these regional minima. Display the mask of the pixels modified by this procedure. What are the properties of this transformation?

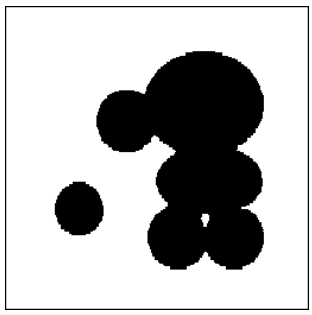
3) Compute the marker-controlled watershed transformation of the *relief* image, using as markers the minima which have not been removed.
4) Starting from any point of the topographic surface, how could you build the partial catchment basin related to this point, that is the set of points which can be flooded by water coming from this particular point?
5) What strategy would you adopt if there really existed significant closed depressions on the topographic surface (such as volcanic craters for instance)?



relief

Exercise n° 5: Separation of particles (first approach)

You have already defined the ultimate erosion (chapter 7, exercises n° 3 and 5) of the *disks* image. Using the same image, try to design an algorithm to separate the disks.
1) Use the geodesic SKIZ of the ultimate eroded sets in the initial set. Is the segmentation satisfactory? Analyse the causes of the problem.
2) We saw that the ultimate erosions correspond to maxima of the distance function. They also can be considered as markers of the disks to be segmented. From these two observations, design an algorithm based on the watershed transform to segment correctly the disks (use the **computeDistance** operator).



disks

Exercise n° 6: Separation of particles (second example)

The previous exercise showed how to use the watershed transform associated with the distance function to segment overlapping or touching objects. It also presented an ideal case of segmentation which works immediately, without the need of any pre-processing. This is not always the case as will prove the next example.
1) Apply the segmentation algorithm seen previously on the *coffee* image. What can you observe? Why?
2) Show how the filtering of the distance function and an appropriate definition of the **edge** parameter allow to overcome these difficulties.

Exercise n° 7: Road segmentation

The purpose of this exercise is to extract the roadway in different road images. This corresponds to the first step of the road segmentation procedure used to initiate the process on a sequence of road images.
1) Use the hierarchical segmentation operator named **enhancedWaterfalls** to segment the *route* image.
2) Apply the same algorithm to the *road* image. Is the result satisfying? Explain why.
3) Use a thick gradient for computing the initial valued watershed. Then, perform a hierarchical segmentation. Can you explain why you get a better result. Apply the same algorithm to the *road1* and *road2* images.
4) Extract two markers, one for the road and the other for the outside, and use them to segment the road with the **markerControlledWatershed** operator.

route

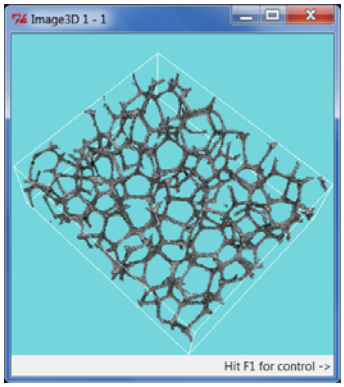
road

road1

road2

Exercise n° 8: Pellets segmentation in a 3D polyurethane foam

This 3D segmentation example shows that a process which has been designed for 2D images can be applied directly to 3D images thanks to the availability in MAMBA of a **mamba3D** module which contains operators transposing many 2D operators to 3D images.
Warning! This exercise is the only one, in the exercise book, which uses **mamba3D** operations. To take advantage of this exercise, it is advised to use the VTK 3D display functionalities available in the MAMBA library. See the user manual for the installation procedure and the restrictions.

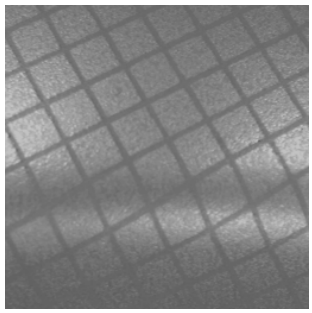


foam

The initial 3D *foam* image represents a foam made of polyurethane pellets. However, these pellets are so compressed that the separation walls between them have disappeared. Only corners at the junction of adjacent pellets are still visible. Nevertheless, it is possible to rebuild the separation walls with a procedure which is, in 3D, similar, indeed identical, to the approach used to segment coffee grains presented in the previous exercise.
1) Use the display operators in **mamba3D** to display the initial image.
2) Transpose in 3D the segmentation process defined for the coffee grains by using the corresponding 3D operators available in the 3D module.

Exercise n° 9: Stamped grid in steel

A regular grid has been printed on a steel sheet before its stamping. This example shows how the crossing points of this grid after stamping can be extracted. The new position of each point indicates its displacement during stamping and therefore the degree of stress exerted locally on the steel sheet.
1) Design a filtering procedure applied on the *steel_sheet* image to extract sufficiently good markers of the grid cells (we suggest to use levellings filters. Don't try to attain perfection, multiple markers in a single cell are allowed).
2) Use the watershed operator to extract the grid.
3) Extract the crossing points of the grid.



steel_sheet

Exercise n° 10: Analysis of a burner

The *burner* image represents a detail of a gas heating appliance. The infrared emitter consists of:
- a ceramic plate with cylindric holes opening onto the surface by cavities forming truncated cones with a hexagonal base.

- a raster formed of longitudinal and transversal metallic wires.

From a morphological point of view, the interesting structures are the following:

- The raster which appears on the image under the form of linear horizontal and vertical structures in light grey tone.
- The hexagonal structures a little darker.
- The black spots located inside the hexagons. Both spots and hexagons may be partially hidden by the raster.
- The background of the image which presents higher grey levels (white tone).

The purpose of this exercise is to extract a mask of the raster and to be able to position correctly (with respect to the grid) the center of the visible spots.

1) What kind of grid is it interesting to use?

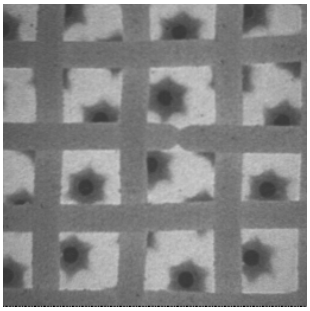
2) Use morphological filters in order to reduce the granular aspect of the image without modifying the structures of interest. Do you note a significant difference between the filters based on closing-opening and those based on opening-closing?

3) Extract the raster.

4) Segment also the hexagonal sructures. Use the marker-controlled watershed transform to produce a partition of the image in three regions: the raster, the hexagonal structures and the background.

5) Extract the black spots. Use the fact that these black spots correpond to significant minima inside the hexagonal structures.

6) Try to position as accurately as possible the centers of the visible spots.



burner

SOLUTIONS

Exercise n° 1

1) The two watershed operators available in MAMBA, named **watershedSegment** and **basinSegment**, are marker-controlled watershed operators. Both require two input images: the image to be segmented (either 8-bit or 32-bit) and a 32-bit image containing the labelled markers used as sources of flooding. When the markers are the minima of the image, we obtain a classical watershed transform. The difference between the two watershed operators is that the first one computes the watershed lines and the labelled catchment basins (each one takes a label equal to the label of its included marker) and the second one computes only the catchment basins (it is therefore a little bit faster). The result of the transformation is put in the image which contained the markers. In both cases, the catchment basins are stored in the three lower byte planes of the image (this means that a maximum of 2^{24} - 1 catchment basins

is allowed). For the **watershedSegment** operator, the watershed lines are stored in the upper byte plane. They take the value 255.
Load the *electrop* image in im1, compute its gradient in im2 and the minima of the gradient in imbin1:

```
>>> gradient(im1, im2)
>>> minima(im2, imbin1)
```

Then label these minima and put the result in im32_1:

```
>>> label(imbin1, im32_1)
3943
```

As indicated, 3943 sources of flooding appear in the gradient image. Copy the label image in im32_2 (it will be use again later). Then perform the watershed transform of the gradient (with apparent watershed lines):

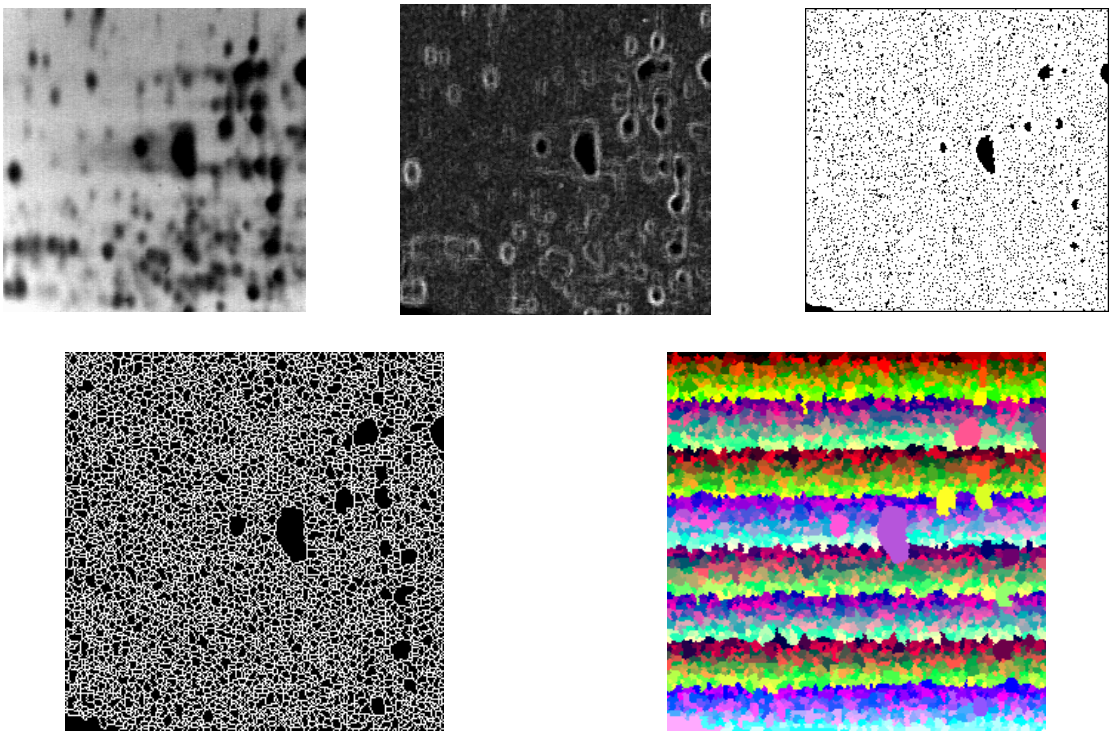
```
>>> watershedSegment(im2, im32_1)
```

The watershed lines can be stored in im3 as a 8-bit image or in imbin2 as a binary image:

```
>>> copyBytePlane(im32_1, 3, im3)
>>> copyBitPlane(im32_1, 31, imbin2)
```

You can also compute a watershed transform without lines (use the labelled minima previously copied in im32_2):

```
>>> basinSegment(im2, im32_2)
```



Initial electrop image (upper left), gradient of the image (upper middle), minima of the gradient (upper right), watershed lines of the gradient (lower left) and catchment basins (lower right).

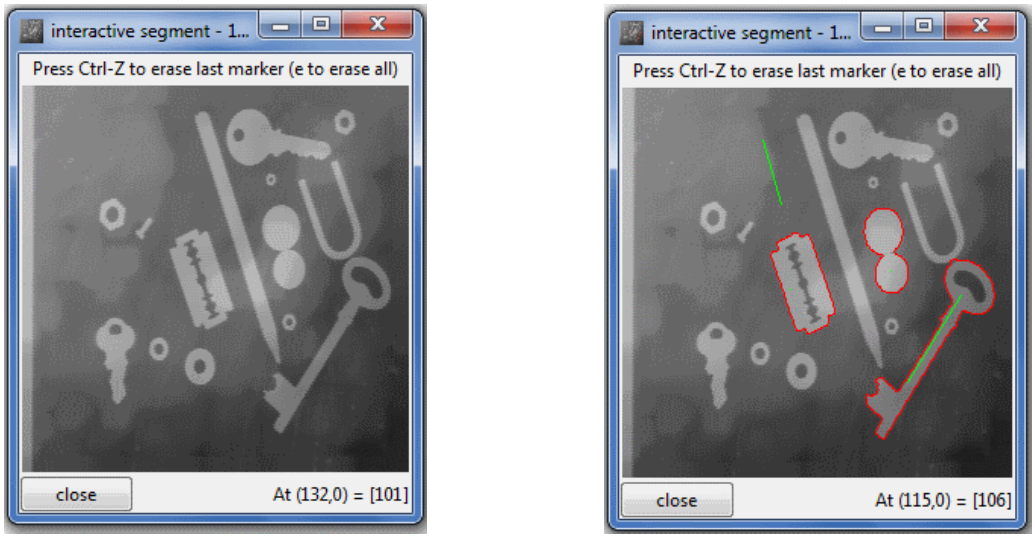
The transformation produces an important over-segmentation. This is due to the large number of initial markers (3943). These markers, sources of flooding, are due to the noise in the image and in its gradient and they will generate as many catchment basins.

2) The **interactiveSegment** operator is located in the **mambaDisplay.extra** module which can be imported as follows:

```
>>> from mambaDisplay.extra import *
```

This operator needs two image arguments: the image to be segmented (which can be 8-bit or 32-bit) and the 32-bit image where the result of the segmentation is stored. When launched, a display window opens where you can define markers by clicking the left mouse button. Markers can be simple points, segments or chains of segments. A segment or a chain are defined by a first click to enter the location of the first extremity followed by a second click while pressing at the same time the keyboard <control> key. Continuing to press this <control> key allows to concatenate segments. The markers appear in green in the display. The watershed transform of the image gradient is performed in real time and is displayed in red as soon as at least two markers have been defined.
Load the *tools* image in im1 and type:

```
>>> interactiveSegment(im1, im32_1)
```



Interactive segmentation display (left), result of the segmentation after the definition of 4 markers, one in each segmented object and one in the background (right).

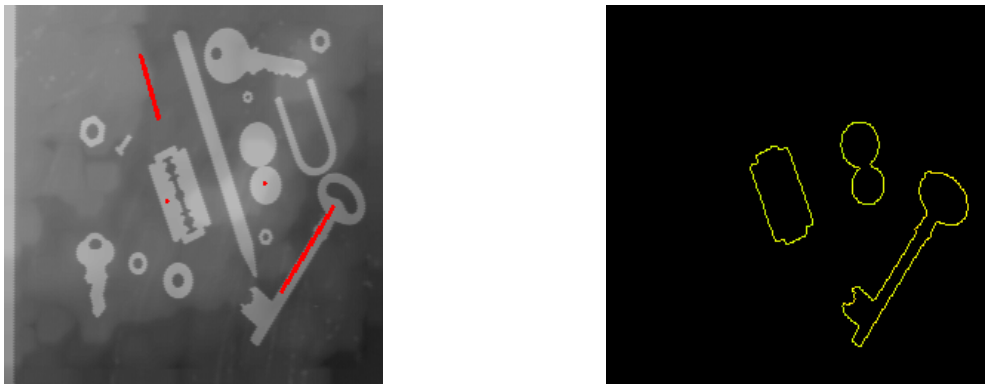
To leave the interactive segmentation tool, press on the <close> button. The watershed transform is stored in im32_1 and a list of coordinates of the marker points is then returned:

```
[(176, 120), (92, 34, 104, 76), (110, 132), (222, 136, 187, 193)]
```

This list of coordinates can be used to draw the markers in a binary image (imbin1) and to verify that using them to control the watershed of the gradient image produces the same result. Be careful however to perform a closing of the marker set before using it. Indeed the **drawLine** operator using the square grid, the segment markers will be disconnected on the hexagonal grid (there exists another way to cope with this problem which does not need this closing step):

```
>>> imbin1.reset()
>>> imbin1.setPixel(1, (176, 120))
>>> imbin1.setPixel(1, (110, 132))
>>> drawLine(imbin1,(92, 34, 104, 76), 1)
>>> drawLine(imbin1,(222, 136, 187, 193), 1)
>>> gradient(im1, im2)
>>> closing(imbin1, imbin1)
>>> markerControlledWatershed(im2, imbin1, im3)
```

The **markerControlledWatershed** operator is applied on a greyscale image (8-bit), it uses directly a binary set of markers and return in a greyacale image the valued watershed, that is a watershed where each point takes the value of the initial image.



Left image: original tools image and the marker set (slightly dilated). Left image: marker-controlled valued watershed of the gradient.

You can verify also that it is of the outmost importance to correctly define the markers. If a marker is badly located (for instance, if it is crossing the boundary of a region), the result of the segmentation is generally not satisfactory.

Only two markers have been used. Both are in the background. Therefore, the image is divided into two catchment basins and the watershed line is not very helpful.

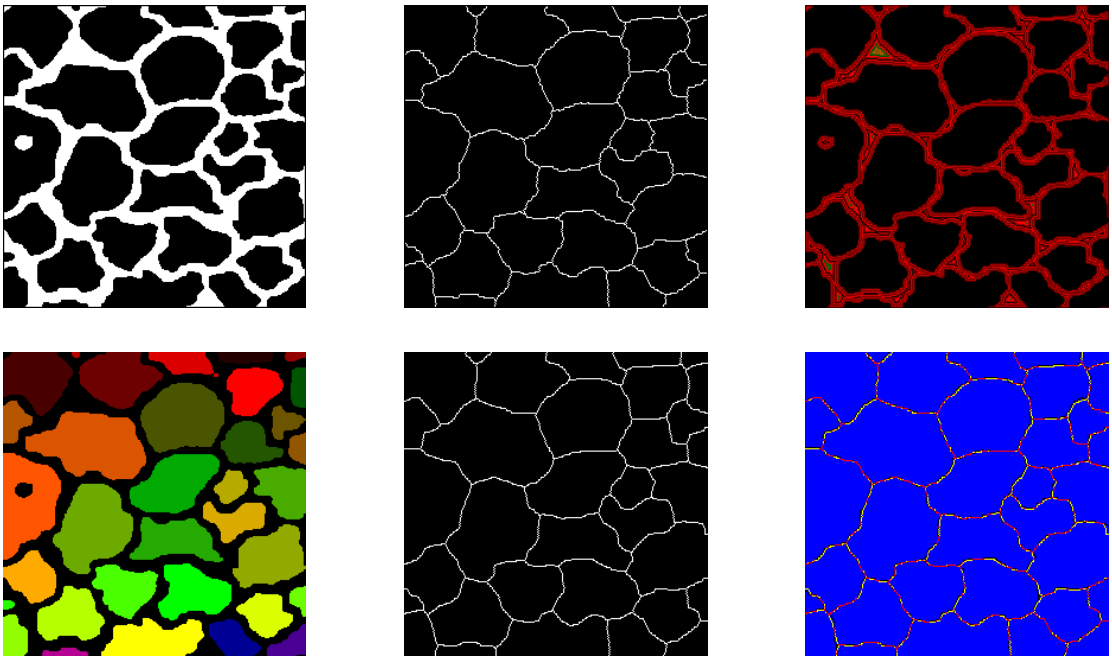
Exercise n° 2: Return on the SKIZ and geodesic SKIZ operators

1) Consider a set X and the distance function $d(X^c)$ of X^c, complement of X. The influence zones of the connected components of X appears to be nothing but the catchment basins of the watershed of d. Let us verify this with the *alumine* image loaded in imbin1. We compute first the SKIZ with thickenings which will be compared to the SKIZ obtained with the watershed transform:

```
>>> computeSKIZ(imbin1, imbin2)
```

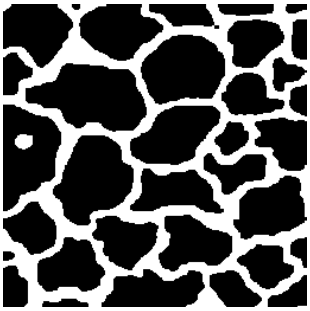
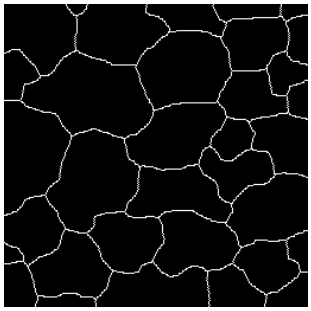
Then, we compute the distance function of the complementary set and perform its watershed (the markers are the connected components of the *alumine* image):

```
>>> negate(imbin1, imbin3)
>>> computeDistance(imbin3, im32_1)
>>> n = label(imbin1, im32_2)
>>> watershedSegment(im32_1, im32_2)
```



From top to bottom and from left to right: initial alumine image, SKIZ obtained by thickenings, distance function of the complementary initial image, labelling of the initial image, SKIZ obtained by watershed of the distance function, comparison of the two SKIZ: apart some local variations due to different choices of initial directions of propagation, the two SKIZ are similar.

The two SKIZ are very similar. The differences are due to the fact that the thickenings use successive directions of the structuting elements whereas the watershed transform is based on hierarchical queues which sort the pixels differently.

Watershed (right image) of the indicator function of the initial complemented set (left image).

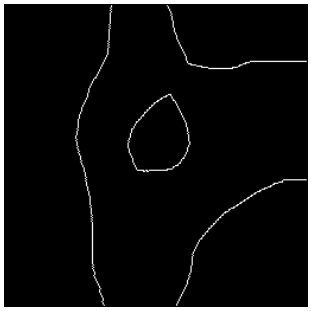
It is, in fact, not necessary to compute the distance function, as the propagation of the flooding is performed directly by the watershed operator based on hierarchical queues. We just need to perform the watershed of the inverted initial image (after it has been converted to a greyscale image):

```
>>> convert(imbin1, im1)
>>> negate(im1, im1)
>>> markerControlledWatershed(im1, imbin1, im2)
```

2) We already compared the SKIZ obtained by thickenings and by the watershed transform on the *alumine* image and saw that there was not a significant difference between the two. But it is not true when the two operators are applied on the *test_SKIZ* image. Load it in imbin1 and enter:

```
>>> computeSKIZ(imbin1, imbin2)
>>> fastSKIZ(imbin1, imbin3)
```





Initial image (left), SKIZ obtained by thickenings (middle), SKIZ obtained by watershed of the distance function (right). A false separation appears in the SKIZ by thickenings. This boundary is included inside the same influence zone.

If we compare the influence zone of the U-shaped component in the two results, we remark that a boundary appears inside this influence zone when the SKIZ is computed with thickenings. This boundary is obviously wrong as it separates points which belong to the same influence zone. This false boundary does not appear in the SKIZ obtained by the watershed transform. The occurrence of this artefact can be explained by the fact that propagations performed by thickenings are whitout memory (it is not possible to know that two propagation fronts are coming from the same source) whereas the initial labelling of the sources in the watershed transform allows to assess the common origin of two apparently distinct propagation fronts.

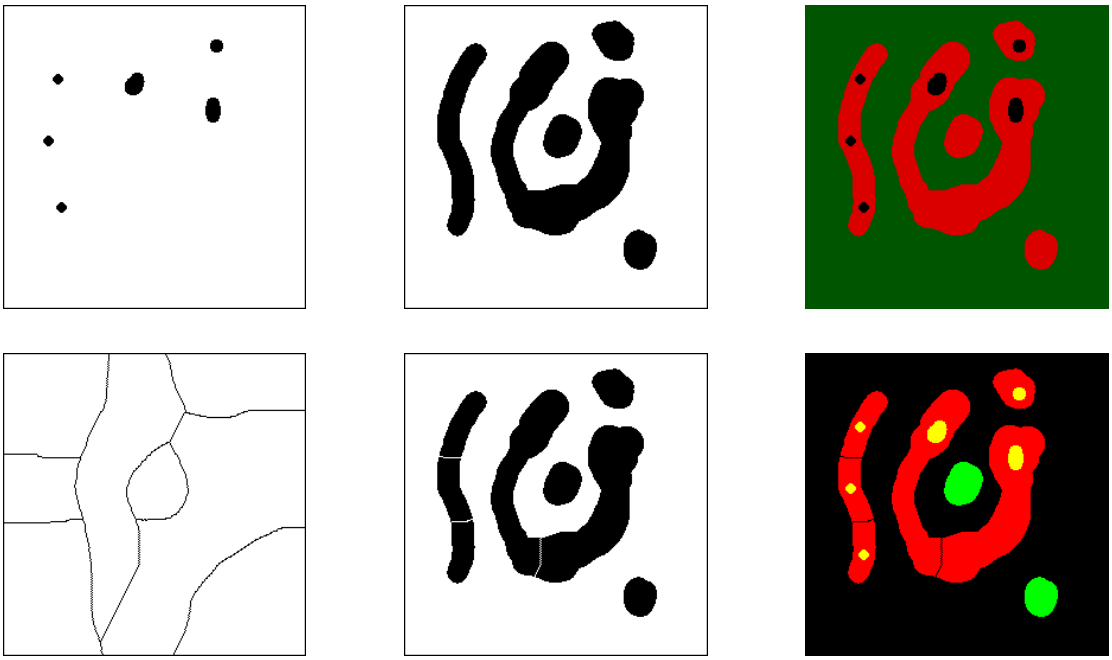
The **fastSKIZ** operator is therefore better than the **computeSKIZ** one. By the way, the former one is faster than the latter one as the computation time is shorter (no iteration of successive thickenings is made). Moreover, this computation time does not depend of the complexity of the initial set.

3) An appropriate labelling of the initial set, the geodesic space and the background allows to define a function on which a watershed transform can be performed to build the influence zones. Let us see how it works.
Load the *test_geodSKIZ* image in imbin1 and close its holes in imbin2. This image will be considered as the geodesic space. Then, the holes are stored in imbin1:

```
>>> closeHoles(imbin1, imbin2)
>>> diff(imbin2, imbin1, imbin1)
```

We can then define a new greyscale image in im1. The points belonging to the geodesic space are given the value 1, whilst the points in the background receive a value equal to 2. Finally, the points belonging to the initial set are set to 0:

```
>>> convertByMask(imbin2, im1, 2, 1)
>>> sub(im1, imbin1, im1)
```



Upper left: initial set (holes of the test_geodSKIZ image). Upper middle: geodesic space (test_geodSKIZ image without holes). Upper right: function used for the watershed, black points take value 0, red ones take value 1 and green ones take value 2. Lower left: watershed transform of the previous function. Lower middle: influence zones together with the connected components of the geodesic space at an infinite geodesic distance from the initial set. Lower right: in yellow, initial set, in red influence zones in the geodesic space, in green connected components which do not belong to any influence zone.

We perform the watershed transform of this image in im2 and we threshold it to get the binary watershed lines in imbin3, which are finally removed from the geodesic space:

```
>>> valuedWatershed(im1, im2)
>>> threshold(im2, imbin3, 1, 255)
>>> diff(imbin2, imbin3, imbin3)
```

However, the set stored in imbin3 contains more than the influence zones as some connected components of imbin2 are at an infinite geodesic distance from imbin1. They do not belong to the influence zones and must be removed by means of a geodesic reconstruction and a set difference:

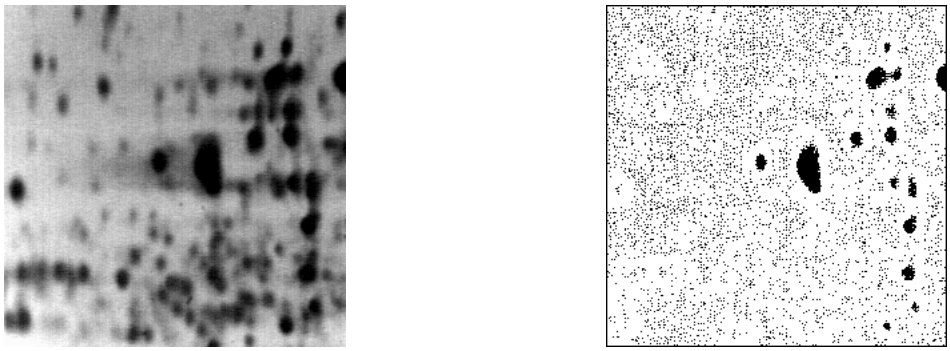
```
>>> copy(imbin1, imbin4)
>>> build(imbin2, imbin4)
>>> logic(imbin3, imbin4, imbin3, “inf”)
```

The final influence zones are stored in imbin3.
This procedure is effectively applied in the MAMBA **geodesicSKIZ** operator. Thanks to the use of a geodesic reconstruction and a watershed transform, the operator is fast and its computation time is independent of the complexity of the geodesic space.

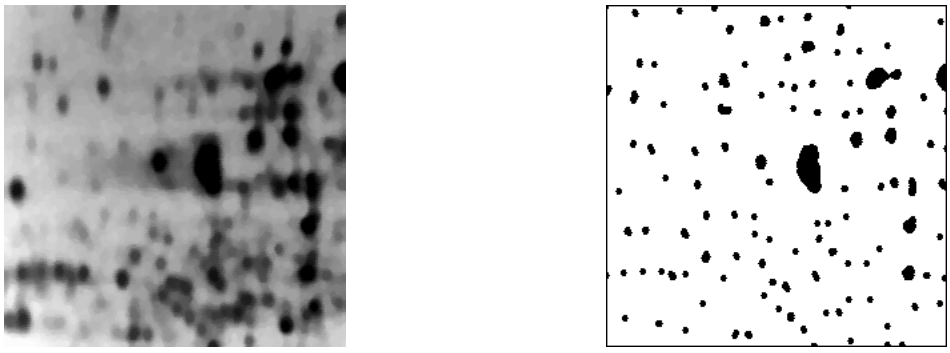
Exercise n° 3

1) Load the *electrop* image into im1 and type:

```
>>> minima(im1, imbin1)
```



Original electrop image (left) and its minima (right).



Filtered image (left) and its minima (right).

There is a great number of minima, indicating that the *electrop* image is very noisy. It seems that working directly with this image is not wise. So, this image is slightly filtered with an

alternate filter of size two (opening first). The minima of the filtered image are more significant. Type the following commands:

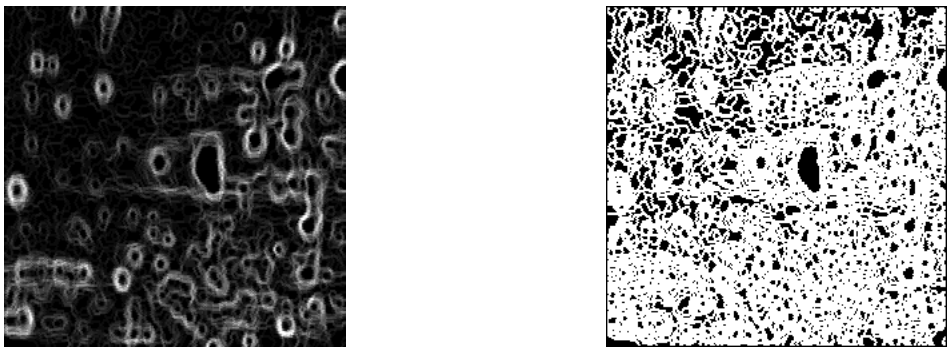
```
>>> opening(im1, im1)
>>> alternateFilter(im1, im1, 2, True)
>>> minima(im1, imbin1)
```

2) Compute the morphological gradient of the filtered image and detect its minima:

```
>>> gradient(im1, im2)
>>> minima(im2, imbin2)
```

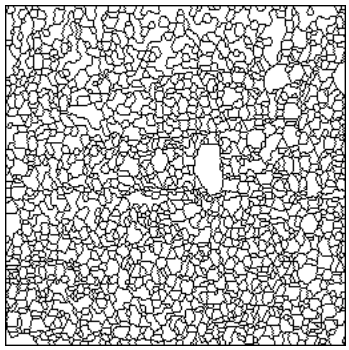
Despite the fact that the initial image as been filtered, its gradient still exhibits numerous minima (compare with the result given in exercise n° 1). Indeed, any sufficiently large flat zone (at least three pixels) is likely to generate a minimum of gradient. Moreover, it is not sure that increasing the size of the initial filtering would improve the result.
The watershed of the gradient image is realised with the **valuedWatershed** operator. The watershed lines are extracted by thresholding:

```
>>> valuedWatershed(im2, im3)
>>> threshold(im3, imbin1, 1, 255)
```



Gradient of the filtered image (left - its contrast has been increased) and its minima (right).

As awaited, the result is extremely "over-segmented", each minimum of the gradient being associated with a catchment basin. Note, however, that the contours of the blobs belong to the watershed lines but they are buried among a lot of non relevant ones.

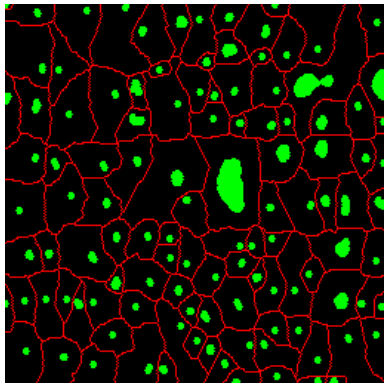


Watershed of the gradient image and over-segmentation.

3) Improving segmentation can be achieved by introducing relevant markers for the blobs but also for the background. Each spot must be marked only once. The minima detected in the above filtered image seem to be perfectly convenient.
Defining a good marker for the background is more difficult. For this, it is necessary to define correctly where is the background. It can be stated that a pixel belongs to the background when it is at the farthest distance from the markers of the blobs. Therefore, the SKIZ of the spot markers could be used. But, the background pixels also need to be as bright as possible. However, the SKIZ does not take into account the actual grey values of the image. This is why it is preferable to perform the watershed of the image. We are sure now that the marker of the background corresponds to the brightest pixels which are far from the blobs. Proceed as follows:

```
>>> valuedWatershed(im1, im3)
>>> threshold(im3, imbin2, 1, 255)
```

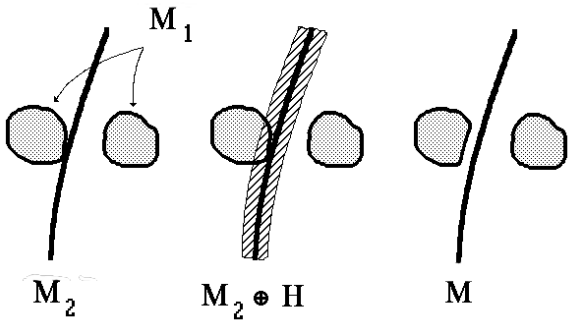
The binary watershed lines stored in imbin2 will be the marker of the background and the minima of the initial filtered image stored in imbin1 are the markers of the blobs. Both of them can be gathered to perform a marker-controlled watershed of the gradient image.



Markers of the blobs in green, marker of the background in red.

However, we must be sure that the background marker does not touch any marker of the blobs. For this, the pixels belonging to the blobs markers which fall inside the elementary dilation of the background marker are removed.

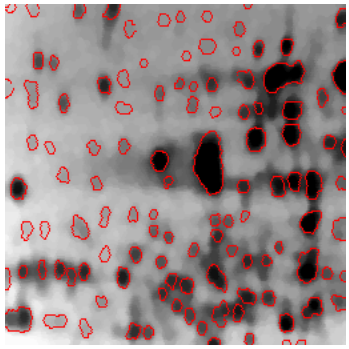
```
>>> dilate(imbin2, imbin3)
>>> diff(imbin1, imbin3, imbin3)
>>> logic(imbin2, imbin3, imbin3, "sup")
```



Preventing that markers of the blobs (M_1) touch the background marker (M_2) by removing points of M_1 which are inside the elementary dilation of M_2. M is the final marker set.

Finally, the contours of the blobs are obtained by a marker-controlled watershed of the gradient image:

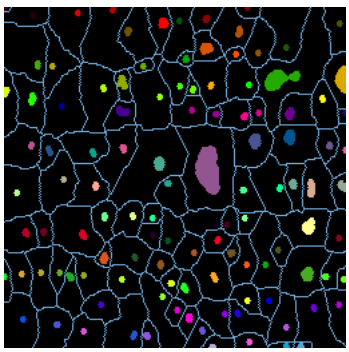
```
>>> markerControlledWatershed(im2, imbin3, im4)
>>> threshold(im4, imbin4, 1, 255)
```

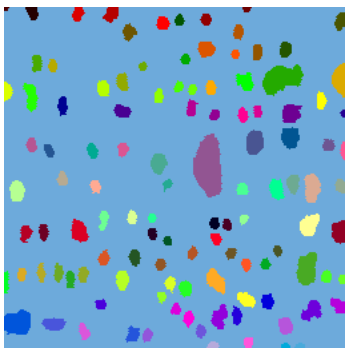


Contours of the blobs.

It is also possible to cope with the touching markers issue by defining differently the label image used by the watershed transform and by using the **basinSegment** operator instead of the **watershedSegment** one. Firstly, the blob markers in imbin1 are labelled. Their number is returned by the **label** operator. Secondly, the marker of the background in imbin2 is valued with the number of blobs + 1. Then, the two images are added to give the final label image. So, even if some blob markers touch the background marker, they have not the same label and the **basinSegment** operator will not merge them:

```
>>> nBlobs =label(imbin1, im32_1)
>>> convertByMask(imbin2, im32_2, 0, nBlobs+1)
>>> logic(im32_1, im32_2, im32_1, "sup")
>>> basinSegment(im2, im32_1)
```



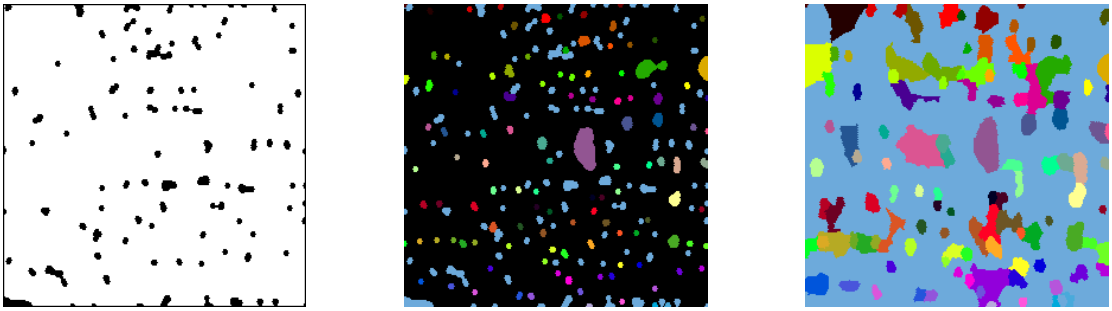


Left: labelling of all the markers, touching markers have different labels. Right: watershed segmentation obtained with the basinSegment operator.

Finally, note that the definition of the background marker can be changed. Instead of using the watershed of the initial filtered image, one can mark the background with the maxima of the image (as the blobs were marked with the minima). Despite the fact that many maxima are extracted, they can act as a unique source of flooding if we give them the same label value. This can be done by the following commands:

```
>>> maxima(im1, imbin3)
>>> nBlobs = label(imbin1, im32_1)
```

```
>>> convertByMask(imbin3, im32_2, 0, nBlobs+1)
>>> logic(im32_1, im32_2, im32_1, "sup")
>>> basinSegment(im2, im32_1)
```



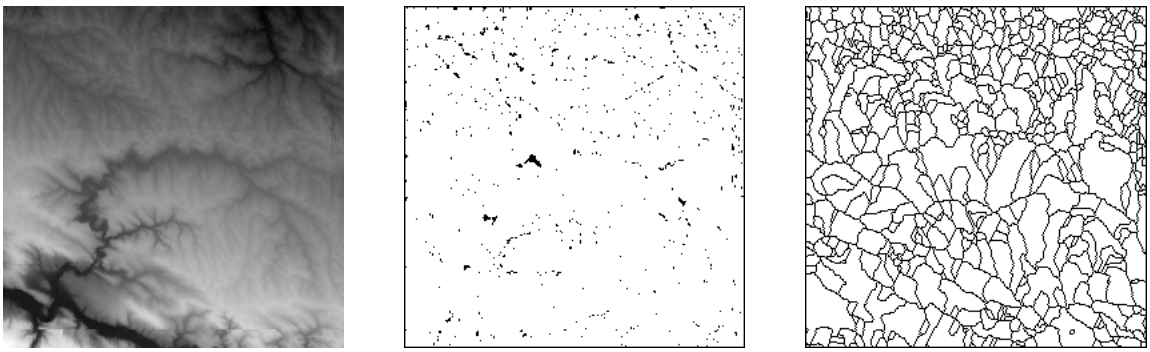
Maxima of the initial filtered image (left), label image where all the maxima have the same label (middle), result of the gradient watershed (right).

Changing the definition and the location of the markers may deeply modify the segmentation. Here, some blobs may be adjacent (which was not allowed with the previous background marker). This segmentation emphasizes the blob clusters.

Exercise n° 4: Catchment basins in a digital elevation model

1) Load the *relief* image into im1 and type:

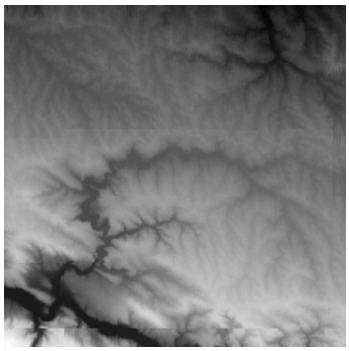
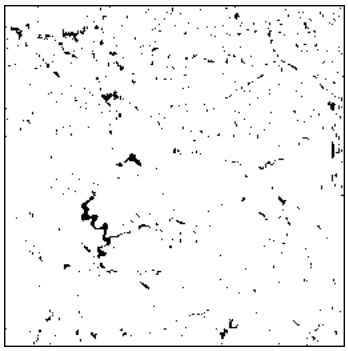
```
>>> minima(im1, imbin1)
>>> valuedWatershed(im1, im2)
>>> threshold(im2, imbin2, 1, 255)
```



Left: initial relief image. Middle: minima. Right: watershed transform.

As awaited from the large number of minima, the watershed transform produces many catchment basins. These minima come from local depressions and dips in the DEM. The presence of these undesirable regional minima within the digital elevation model induces a well-known over-segmentation. In fact, if there was no closed depression, the only regional minima of the DEM would correspond to river outlets and should appear on the field border of the image. Unfortunately, it is not the case.

2) A simple way to remove these unwanted minima consists in filling the dips which are considered as holes (see chapter 6, exercise n° 4).

Initial image after holes closing (left). At the first sight, there is no significant difference with the original image. However, the comparison of the two images shows that many dips have been filled (right).

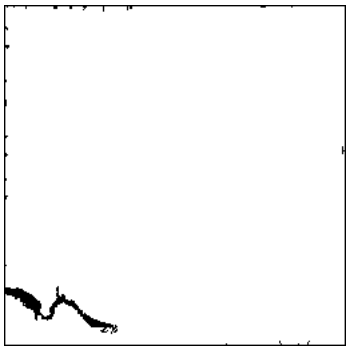
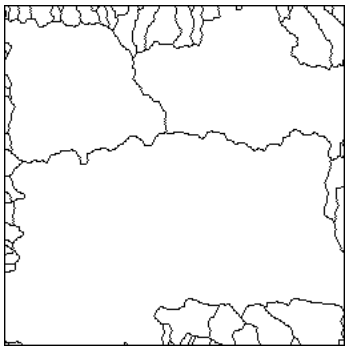
Apply the **closeHoles** operator to the initial image, compare the result with the initial image (with the **generateSupMask** operator) and extract the minima:

```
>>> closeHoles(im1, im2)
>>> minima(im2, imbin2)
>>> generateSupMask(im2, im1, imbin3, True)
```

We already know that this operation is an algebraic closing: it is idempotent, increasing and extensive.

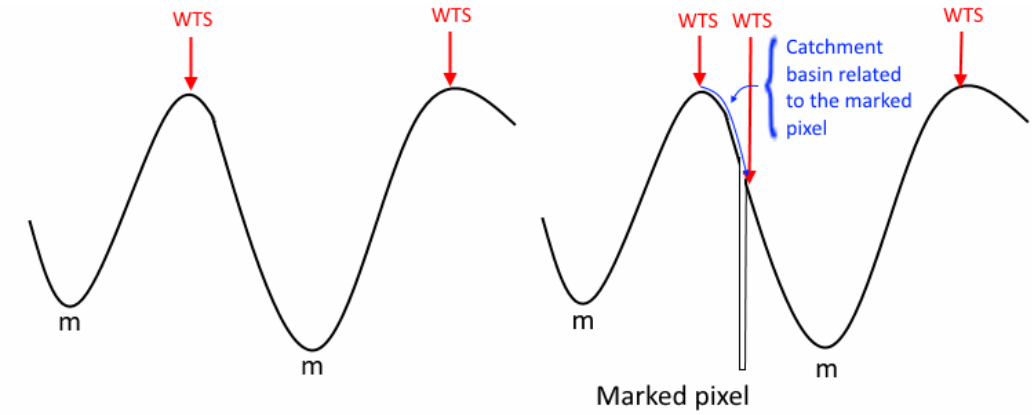
3) We just have to apply the **valuedWatershed** operator to the im2 image containing the relief image without its depressions:

```
>>> valuedWatershed(im2, im3)
>>> threshold(im3, imbin3, 1, 255)
```

The minima of the filtered relief image are touching the edge of the image and correspond to real outlets (left). Watershed transform of the filtered image (right).

4) Determining the points of the topographic surface which are flooded from a given point is simple, as explained in the following figure. All you have to do is to define this point as a flooding source. A new catchment basin is generated containing all the points which are flooded by this new source.
Try this procedure by entering the following commands. The filtered DEM image (without depressions) is copied in im3 and the chosen pixel will be located at (60, 140):

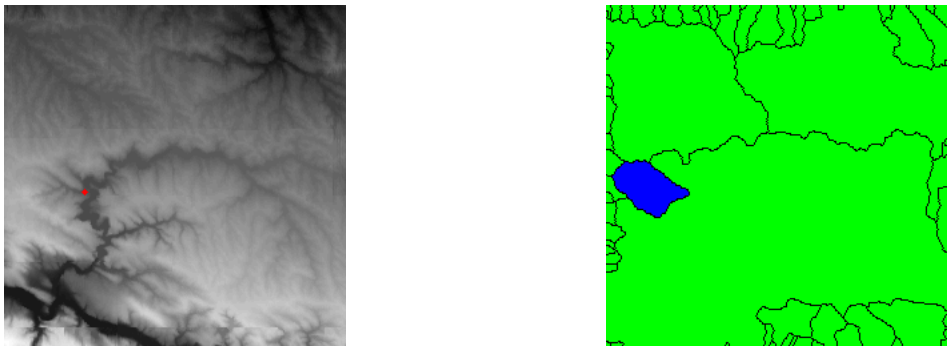


A new source of flooding is created at a specific position by simply replacing the pixel value by 0. A well is generated. When the flooding coming from the other source included in the same catchment basin reaches the point, a new watershed line appears, delimiting the catchment basin related to the point.

```
>>> copy(im2, im3)
>>> im3.setPixel(0, (60, 140))
>>> valuedWatershed(im3, im4)
>>> threshold(im4, imbin1, 1, 255)
```

The new catchment basin is extracted by reconstruction in imbin2:

```
>>> negate(imbin1, imbin1)
>>> imbin2.reset()
>>> imbin2.setPixel(1, (60, 140))
>>> build(imbin1, imbin2)
```



DEM and the selected point in red (left). Catchment basin associated withthe point in blue (right).

5) Taking into account some specific depressions in the DEM is easy. Once the depressions which will be kept are defined and marked, just add these new markers to the outlet markers before performing the watershed transform.

Exercise n° 5: Separation of particles (first approach)

1) The skeleton by geodesic influence zones of the ultimate eroded set gives the following image (the *disks* image is in imbin1):

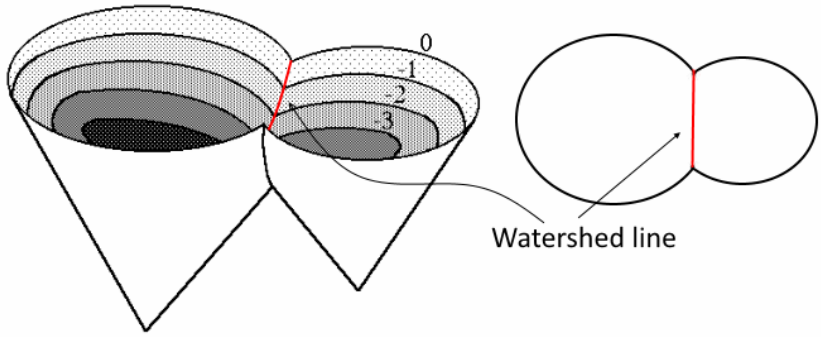
```
>>> binaryUltimateErosion(imbin1, imbin2, im32_1)
```

```
>>> geodesicSKIZ(imbin2, imbin1, imbin3)
```

Initial disk image (left) and geodesic influence zones (in red) of the components of the ultimate erosion (yellow).

The segmentation is not satisfactory, because the procedure does not take into account when the different components of the ultimate eroded set appear. The propagation by the geodesic thickening starts at the same time for all the connected components of the ultimate erosion whereas they appear for different sizes of erosion. Therefore, the separations between the particles are not correctly located.

2) We know that the ultimate erosion corresponds to the maxima of the distance function of the initial set. Let us consider the inverted distance function (see figure below). The connected components of the ultimate erosion are then the minima of this inverted distance function and the separations between the particles are its watershed lines.



The separation between two particles (right) corresponds to the watershed line of the inverted distance function of the initial set (left).

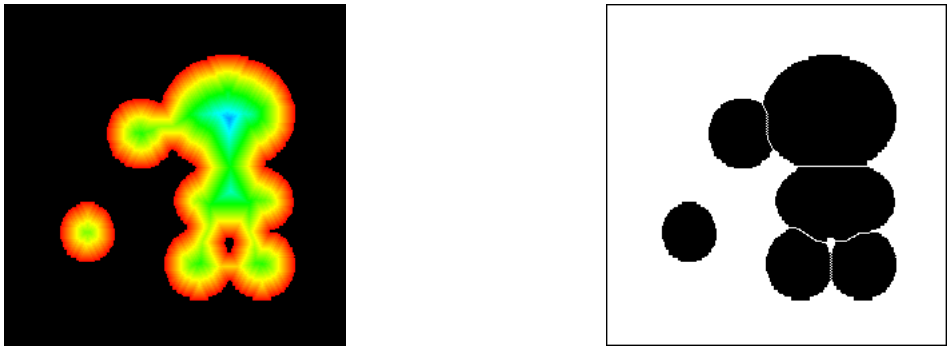
First, we can design an operator, named **distanceWatershed**, which performs the watershed transform of an inverted distance function:

```
def distanceWatershed(imIn, imOut):
    """
    Watershed transform of a distance function loaded in 'imIn', which
    is a 32-bit image.
    This operator inverts the distance function (inversion with a limitation of the
    range of values to reduce the computaation time).
    The result is stored in the binary image 'imOut'.
    """

    imWrk1 = imageMb(imIn)
```

```
imWrk2 = imageMb(imIn)
imWrk3 = imageMb(imIn, 1)
imWrk4 = imageMb(imIn, 1)
maxima(imIn, imWrk3)
maxVal = computeRange(imIn)[1]
imWrk1.fill(maxVal)
sub(imWrk1, imIn, imWrk1)
nParticles = label(imWrk3, imWrk2)
watershedSegment(imWrk1, imWrk2)
copyBitPlane(imWrk2, 31, imWrk3)
threshold(imIn, imWrk4, 1, maxVal)
diff(imWrk4, imWrk3, imOut)
```

Note that, in this procedure, the distance function d, defined in a 32-bit image, is not inverted with the **negate** operator. We prefer to use its maximum value **maxVal** and compute the new function (maxVal - d). This way of doing allows to restrict the function in the range [0, maxVal] whereas the inverted function would be in the range [0, 2^{32}-1], which would induce a very long computation time for the watershed transform.
Then, we can define the **segmentParticles** operator:

```
def segmentParticles(imIn, imOut):
    """
    Segmentation of the touching particles contained in the binary image
    'imIn'. This segmentation performs the watershed of the inverted
    distance function of the image.
    The result is stored in the binary image 'imOut'.
    """

    imWrk1 = imageMb(imIn, 32)
    computeDistance(imIn, imWrk1)
    distanceWatershed(imWrk1, imOut)
```



Distance function of the disk image (left), segementation by the watershed of the inverted distance function (right).

Load the *disk* image in imbin1 and type:

```
>>> segmentParticles(imbin1, imbin2)
```

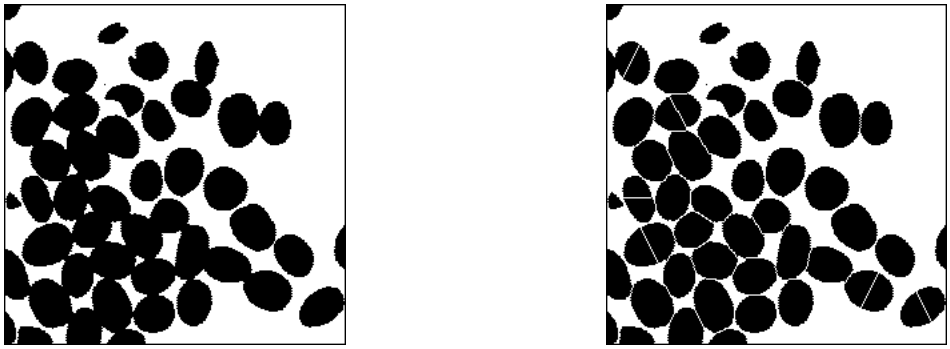
The separation lines are now correctly located.

Exercise n° 6: Separation of particles (second example)

1) Load the coffee image in imbin1 and apply the **segmentParticles** to it:

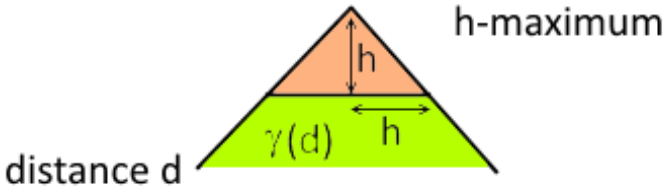
```
>>> segmentParticles(imbin1, imbin2)
```

The **segmentParticles** operator, applied to the coffee image, let some defects appear. First of all, some grains are over-segmented. These grains are marked by more than one connected component of the ultimate erosion, which explains why they are cut into several pieces. This phenomenon has already been observed previously (see chapter 7, exercise n° 5). It is due to irregularities of the distance function.
Another small defect is visible at the left side of the image border. A coffee grain cutting the edge and touching another grain inside the image field is not correctly segmented.



Original coffee image (left) and result of the segmentation by the segmentParticles operator (right). Some coffee grains are over-segmented. Note also that a grain on the upper left side is badly segmented.

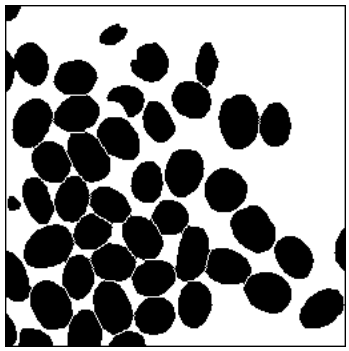
2) The multiple markers issue on the coffee image has already been addressed at chapter 7, exercise n°5. We showed that extracting the h-maxima of height 2 produced connected markers. The same operator could be used here to prevent the over-segmentation of some coffee grains. An equivalent solution consists in performing an opening of size 2 of the distance function. Indeed, the slope of a distance function is equal to 1 (the difference between two adjacent points is at most equal to 1 - the function is said to be 1-Lipchitzian).



If d is a distance function, extracting the h-maxima of d and performing an opening of size h are equivalent operations.

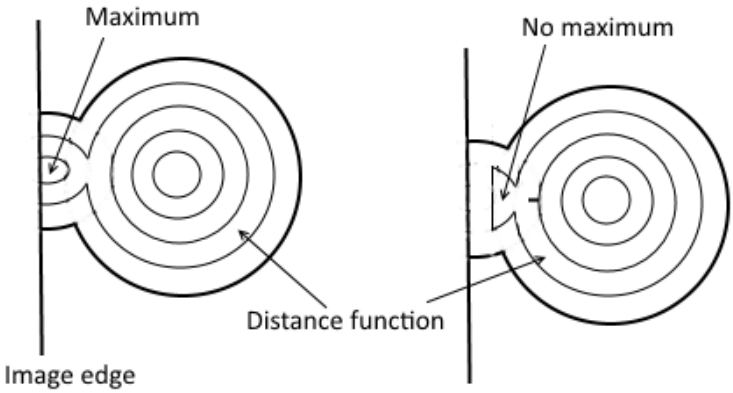
Load the *coffee* image in imbin1 and type:

```
>>> computeDistance(imbin1, im32_1)
>>> opening(im32_1, im32_1, 2)
>>> distanceWatershed(im32_1, imbin2)
```

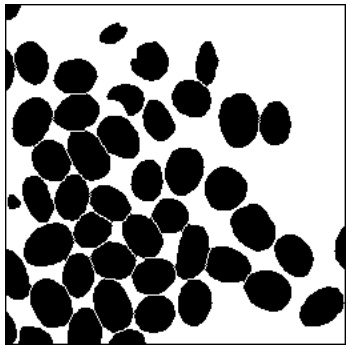


Segmentation of the coffee grains without over-segmentation.

The over-segmentation has disappeared. But a grain on the left side of the image is still not correctly segmented. This is due to the fact that the distance function has been computed with the edge parameter set to **EMPTY** (its default value).



When the parameter edge is set to EMPTY in the computeDistance operator, no maximum appear (right image), whereas this maximum is present when edge is set to FILLED (left image).



Final segmentation of the coffee image without defects (over-segmentation and edge effects).

The final segmentation is obtained with:

```
>>> computeDistance(imbin1, im32_1, edge=FILLED)
>>> opening(im32_1, im32_1, 2)
>>> distanceWatershed(im32_1, imbin2)
```

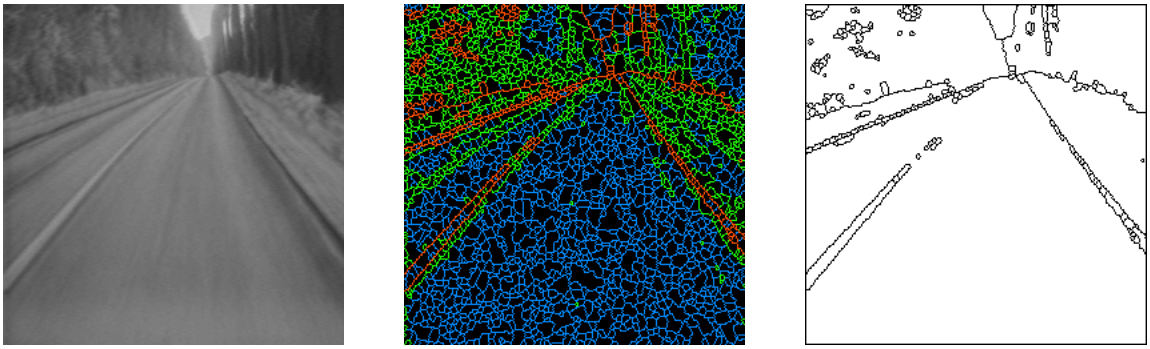
Exercise n° 7: Road segmentation

1) When it is not easy or feasible to extract the objects to be segmented in an image, it is often interesting to use hierarchical segmentation tools.
The road can be defined as a sufficiently large and quite homogeneous region in the foreground of the image. The enhanced waterfalls operator takes an initial valued watershed image and merges the catchment basins which are inside salient regions hence generating different levels of hierarchy. The operator returns the number of levels found. The last one contains the most salient regions.
Load the *route* image and try this operator on the valued watershed of the gradient image. Type:

```
>>> gradient(im1, im2)
>>> valuedWatershed(im2, im3)
>>> enhancedWaterfalls(im3, im4)
3
```

The operator returns the value 3. Three levels of hierarchy were found and stacked in the greyscale image im4. To get the last one, simply threshold it in the range [3, 255]:

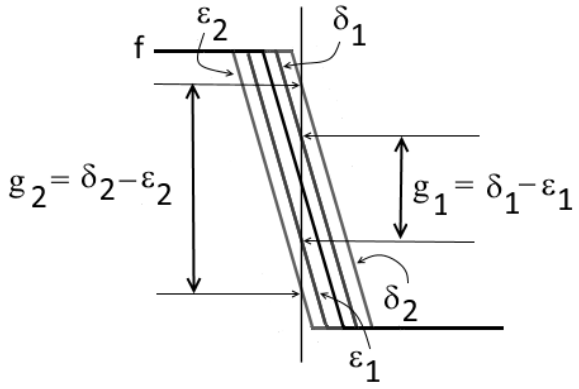
```
>>> threshold(im4, imbin1, 3, 255)
```



Initial route image (left), result of the enhanced waterfalls segmentation, each level of hierarchy is represented with a different color (middle), extraction of the highest level of hierarchy (right).

The last segmentation contains a large region corresponding to the road which can be extracted and used to build a marker.

2) However, if we apply the same algorithm to the *road* image, the result is not so good (see below). This is due to the bad quality of the image, which is low-contrasted and fuzzy (the camera was aboard a running car).
If a gradient of size 1 is used, when a contour is not sharp, its value is lower than the true variation of luminance between the two regions separated by the contour. As this variation corresponds to the value of the watershed, it is very important that this value be as close as possible to the real one so that the enhanced waterfalls transformation be able to assess correctly the saliency of this contour in the hierarchy.

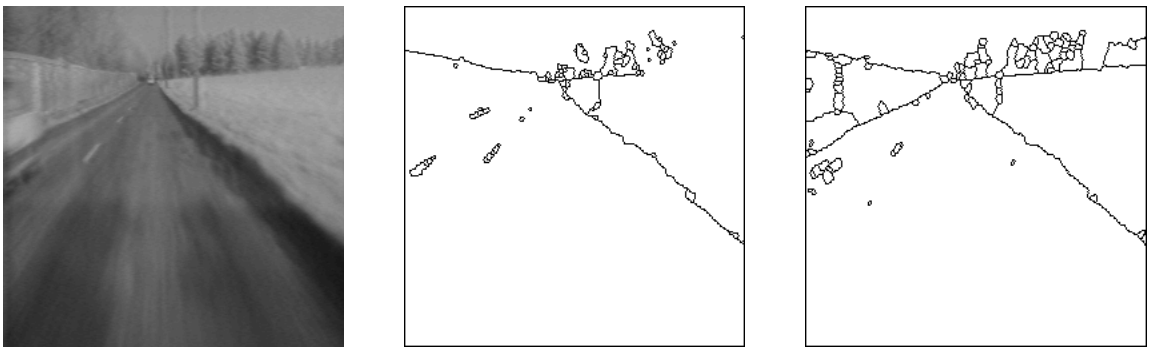


Value g_1 of the gradient when elementary erosion and dilation are used. Value g_2 corresponding to a thick gradient of size 2.

3) So, try again the algorithm with a thick gradient of size 2:

```
>>> gradient(im1, im2, 2)
>>> valuedWatershed(im2, im3)
>>> enhancedWaterfalls(im3, im4)
3
>>> threshold(im4, imbin1, 3, 255)
```

As explained above, the gradient values are better determined with a thick gradient. Therefore, the hierarchies computed by the enhanced waterfalls transform better extract the salient regions.
The highest hierarchical level (equal to 3) produces a good segmentation of the road.



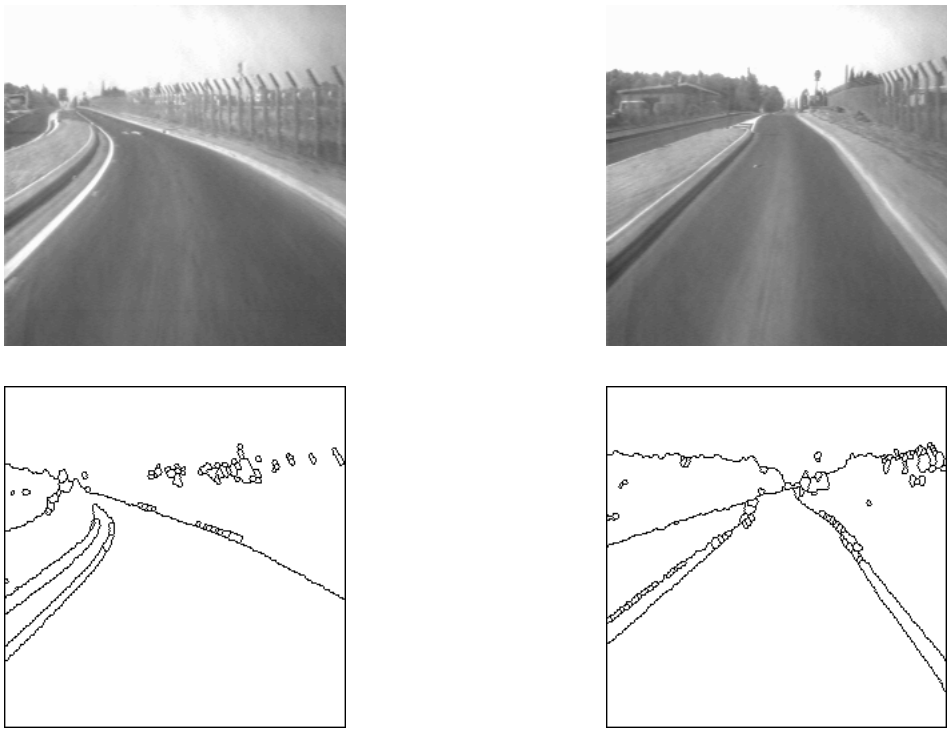
Initial road image (left). Highest level of hierarchy (4) with a gradient of size 1 (middle). Highest level (3) with a gradient of size 2.

You can also test this algorithm on the *road1* and *road2* images.

4) As mentioned above, the road should correspond to the catchment basin which is at the foreground of the image. So, to define a road marker together with an outside one, different steps are used and described below.
Load the *road1* image in im1 and get the hierarchical segmentation (catchment basins) in imbin1:

```
>>> gradient(im1, im2, 2)
>>> valuedWatershed(im2, im3)
>>> enhancedWaterfalls(im3, im4)
```

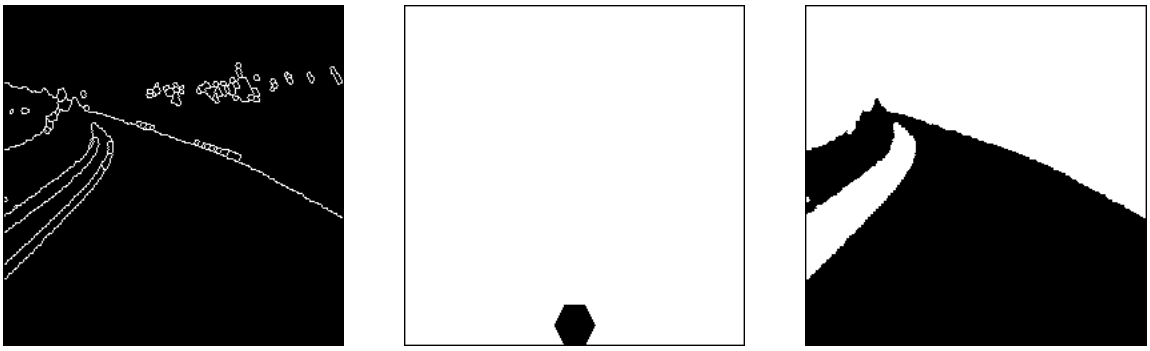
```
4
>>> threshold(im4, imbin1, 0, 3)
```



Hierarchical segmentation (highest level) of the road1 (left) and road2 (right) images.

Then, we can define a first marker in imbin2 made of a dilated point located in the middle and bottom of the image (we verify its size with the **getSize** method):

```
>>> imbin2.reset()
>>> im1.getSize()
(256, 256)
>>> imbin2.setPixel(1, (128, 240))
>>> dilate(imbin2, imbin2, 15)
```



Hierarchical segmentation of the road1 image (left), marker of the road (middle) and extracted catchment basin (right).

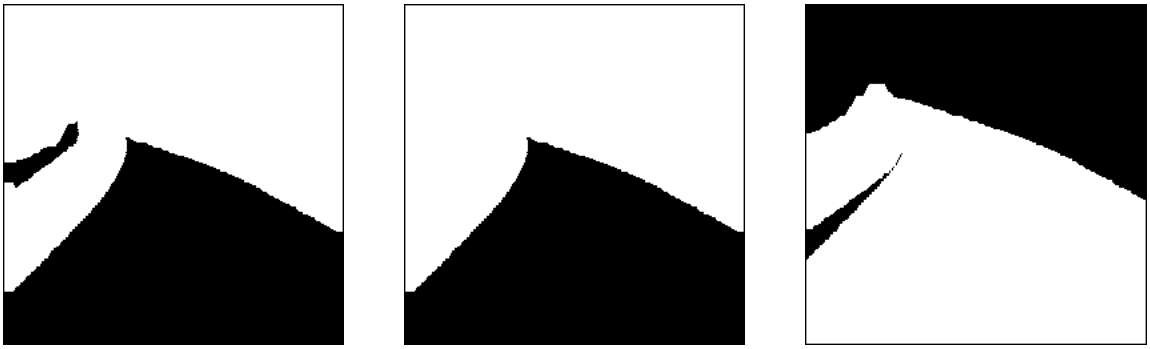
This marker is duplicated in imbin3 (the same marker will be used later on) and used to extract the foreground catchment basin (in imbin3). This catchment basin is filtered. Its holes are filled:

```
>>> copy(imbin2, imbin3)
>>> build(imbin1, imbin3)
```

```
>>> closeHoles(imbin3, imbin3)
```

This catchment basin is eroded with a sufficiently large erosion (size 10). This erosion will ensure that the road marker will fall inside the road and will not run into its boundary and the outside. Moreover, as we want that the road marker be composed of a single component, we perform again a reconstruction of the eroded set to keep this connected component as marker. The connected components which are not rebuilt are considered to be markers of the outside.

```
>>> erode(imbin3, imbin4, 10)
>>> build(imbin4, imbin2)
>>> diff(imbin4, imbin2, imbin4)
```



Erosion of the road catchment basin (left), extraction of the connected component marked by the initial marker (middle), erosion of size 10 of the complementary of the road catchment basin (right).

The final outside marker is obtained by eroding (same size of erosion) the complementary part of the road catchment basin (stored in imbin3) and by adding to this set the connected components which have not been rebuilt in the previous step.

```
>>> negate(imbin3, imbin3)
>>> erode(imbin3, imbin3, 10)
>>> logic(imbin3, imbin4, imbin3, "sup")
```

The non rebuilt marker in the previous step is added to the previous step (left). This set defines the outside marker in red whilst the green connected component is the marker of the road (right).

The entire procedure can be realised by the following **roadMarkerExtract** operator. The initial image is in im1 and the 32-bit image im2 contains the labelled markers:

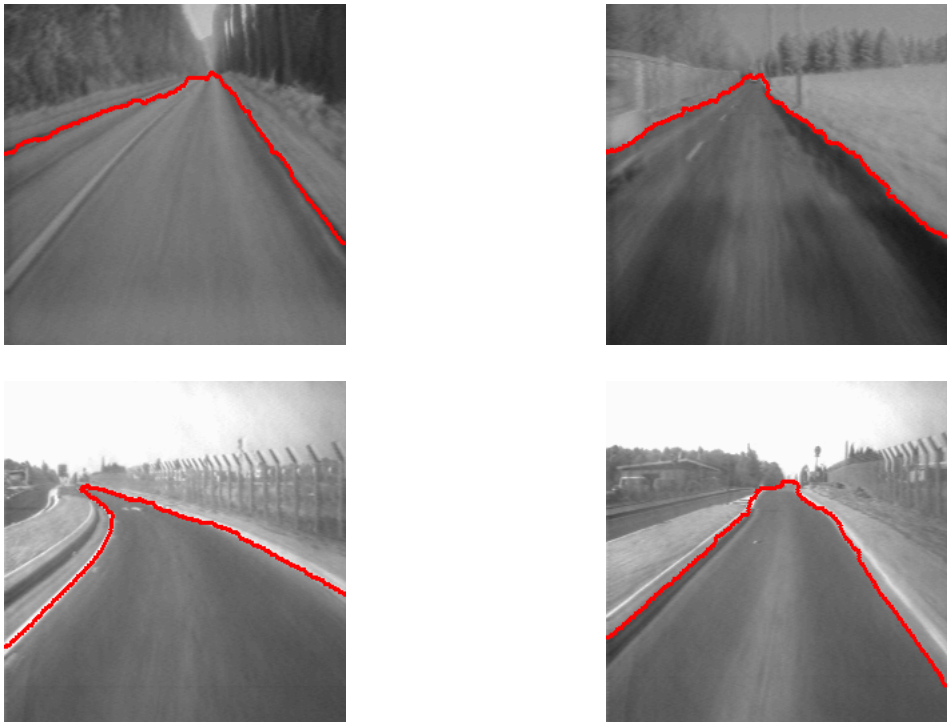
```
def roadMarkersExtract(im1, im2):
  """
  Extraction of the road and outside markers from the segmentation obtained
  by the higher level of hierarchy of the enhanced waterfalls transform.
  'im1' contains the initial catchment basins. 'im2' is a 32-bit image containing the
  labelled markers for the road and for the outside.
  """

  imWrk1 = imageMb(im1)
  imWrk2 = imageMb(im1)
  imWrk3 = imageMb(im1)
  imWrk1.reset()
  imSize = im1.getSize()
  xPos = imSize[0]//2
  yPos = imSize[1]-15
  imWrk1.setPixel(1, (xPos, yPos))
  dilate(imWrk1, imWrk1, 15)
  copy(imWrk1, imWrk2)
  build(im1, imWrk2)
  closeHoles(imWrk2, imWrk2)
  erode(imWrk2, imWrk3, 10)
  build(imWrk3, imWrk1)
  diff(imWrk3, imWrk1, imWrk3)
  negate(imWrk2, imWrk2)
  erode(imWrk2, imWrk2, 10)
  logic(imWrk2, imWrk3, imWrk2, "sup")
  convertByMask(imWrk1, im2, 0, 2)
  add(im2, imWrk2, im2)
```

Finally, the complete road segmentation can be achieved with the **roadSegment** operator:

```
def roadSegment(im1, im2):
  """
  Segmentation of the road in image 'im1'. The result is put in
  the binary image 'im2'.
  The algortihm uses the above road markers extractor.
  """

  imWrk1 = imageMb(im1)
  imWrk2 = imageMb(im1)
  imWrk3 = imageMb(im1, 1)
  imWrk4 = imageMb(im1, 32)
  gradient(im1, imWrk1, 2)
  valuedWatershed(imWrk1, imWrk2)
  nbHier = enhancedWaterfalls(imWrk2, imWrk1)
  threshold(imWrk1, imWrk3, 0, nbHier -1)
  roadMarkersExtract(imWrk3, imWrk4)
  watershedSegment(imWrk1, imWrk4)
  copyBitPlane(imWrk4, 31, im2)
```

Test it with the *route*, *road*, *road1* and *road2* images:

```
>>> roadSegment(im1, imbin1)
```



From top to bottom and from left to right, result of the road segmentation algorithm applied to the route, road, road1 and road2 images.

Note that his road segmentation algorithm is far from being universal. The use of enhanced waterfalls gives good results only if the road is really a salient region with a sufficient contrast. The extraction of markers could be more robust, a deeper insight of the shape and size of these markers should be performed. Anyway, this exercise only aims at showing the capabilities of the morphological segmentation tools.

Exercise n° 8: Pellets segmentation in a 3D polyurethane foam

1) The *foam* image is a 3D image stored in the **Mamba_Images/3D** directory. This image is in the **foam** directory and is composed of 74 2D image sections. Define properly the working directory with the following commands:

```
>>> import os
>>> os.chdir("c:/Mamba_Images/3D")
```

Then import the **mamba** and **mamba3D** modules, load the *foam* image into the 3D imA image and convert this image into a binary one:

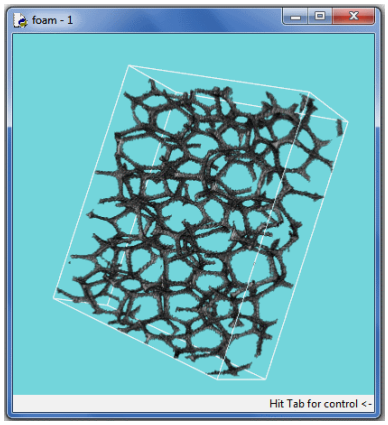
```
>>> from mamba import *
>>> from mamba3D import *
>>> imA = image3DMb('foam')
>>> imA.convert(1)
```

The size of the image is given by:

```
>>> ImA.getSize()
(192, 224, 74)
```

If the VTK Visualisation Tool Kit has been properly installed (read carefully the MAMBA user manual about the installation procedure and its restrictions), you can display this image in 3D with the show() method by setting the mode to **VOLUME**:

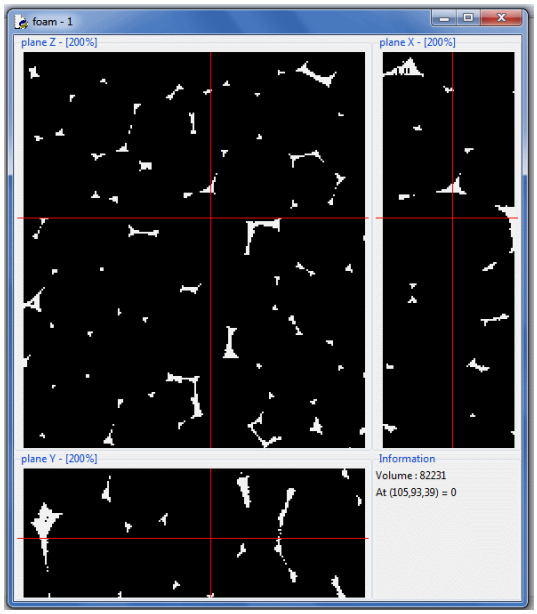
```
>>> imA.show(mode="VOLUME")
```



3D display in VOLUME mode (VTK installed).

You can also use the **PROJECTION** mode (which is the default mode):

```
>>> imA.show(mode="PROJECTION")
```



In PROJECTION mode, the three projections of the 3D image are displayed and it is possible to navigate inside it by moving the mouse cursor.

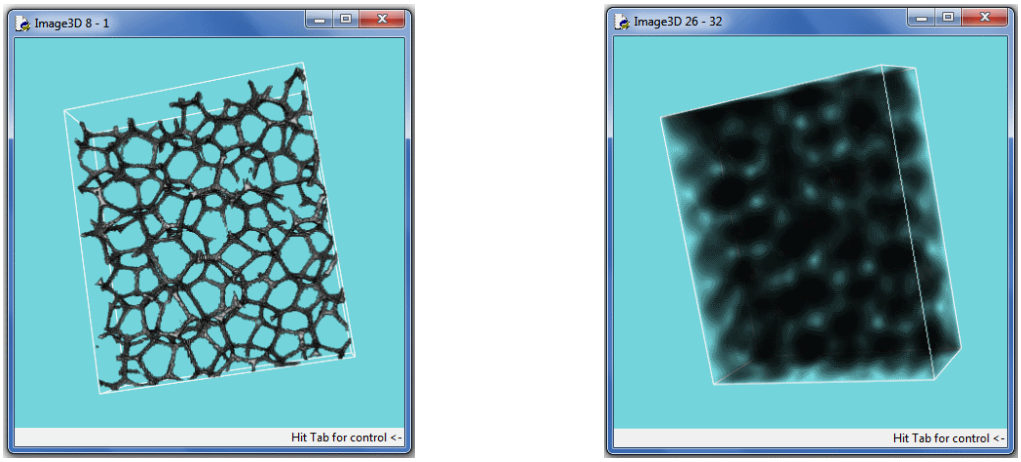
2) Although we are working with 3D images, the segmentation process is similar to the one used for separating coffee grains in the exercise n° 6. Indeed, most of the 2D operators also exist in the **mamba3D** package (their name is generally the same as in 2D with simply the suffix 3D added).
In the first step, the initial image is filtered: holes are closed and a reconstruction opening is performed to remove small artefacts which could be present in the 3D volume. A sup of directional linear openings of size 2 is used. Note that the default grid is the Face Centered

Cubic (FCC) grid in **mamba3D**. The elementary structuring element is a cuboctahedron (each pixel has 12 neighbors).

```
>>> imB = image3DMb(imA)
>>> closeHoles3D(imA, imB)
>>> imC = image3DMb(imA)
>>> supOpen3D(imB, imC, 2)
>>> build3D(imB, imC)
```

Then, the filtered image is inverted and its 3D distance function is performed.

```
>>> imD = image3DMb(imA)
>>> negate3D(imC, imD)
>>> imE = image3DMb(imA, 32)
>>> computeDistance3D(imD, imE, edge=FILLED)
```



Initial foam image (left) and distance function of the complementary image (right).

Contrary to the 2D case where the inverted distance function was decremented in order to reduce the computation time of the watershed transform, we prefer here to verify that the maximal value of the distance function is less than 256 so that it is possible to transfer it into a greyscale (8-bit) 3D image before inverting it.

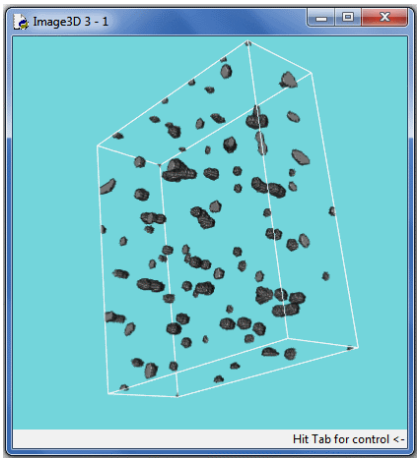
```
>>> computeRange3D(imE)[1]
32
```

The maximal distance is equal to 32. So, the lowest byte plane of the imE image can be transferred into the 8-bit imF image:

```
>>> imF = image3DMb(imA, 8)
>>> copyBytePlane3D(imE, 0, imF)
```

The distance function is filtered in order to keep its most significant maxima (markers of the pellets).

```
>>> imG = image3DMb(imA, 8)
>>> opening3D(imF, imG, 3)
>>> maxima3D(imG, imB)
```



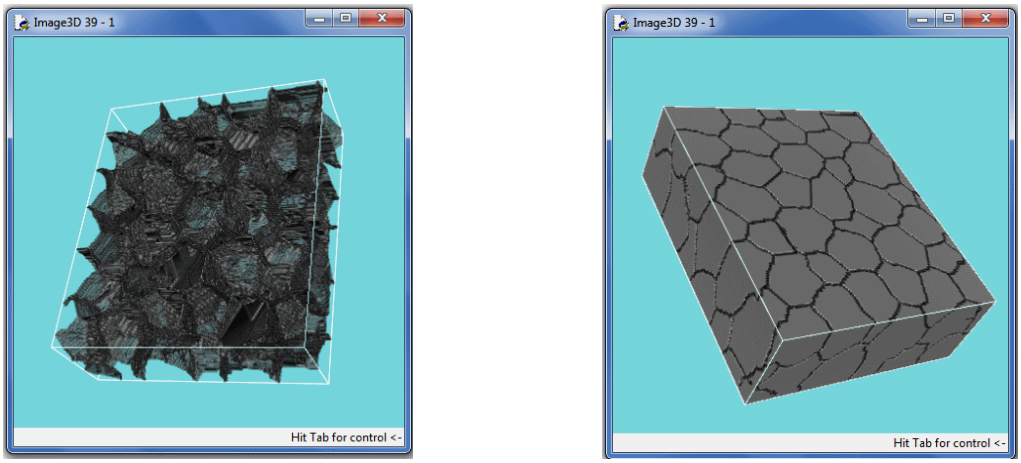
Maxima of the opened distance function.

The distance function is inverted and the markers are labelled. The number of labels corresponds to the number of pellets:

```
>>> negate3D(imF, imF)
>>> label3D(imB, imE)
94
```

Finally, a 3D watershed of the inverted distance function is performed:

```
>>> watershedSegment3D(imF, imE)
>>> imH = image3DMb(imA)
>>> copyBitPlane3D(imE, 31, imH)
```



The septa between the pellets are rebuilt (left). The separated pellets are visible in the inverted image (right).

The result of the watershed transform can be displayed. Inverting the image shows the separated pellets:

```
>>> imH.show(mode="VOLUME")
>>> negate3D(imH, imH)
```

As mentioned earlier, this exercise has proved that working on 3D images is not an issue with **mamba3D**.

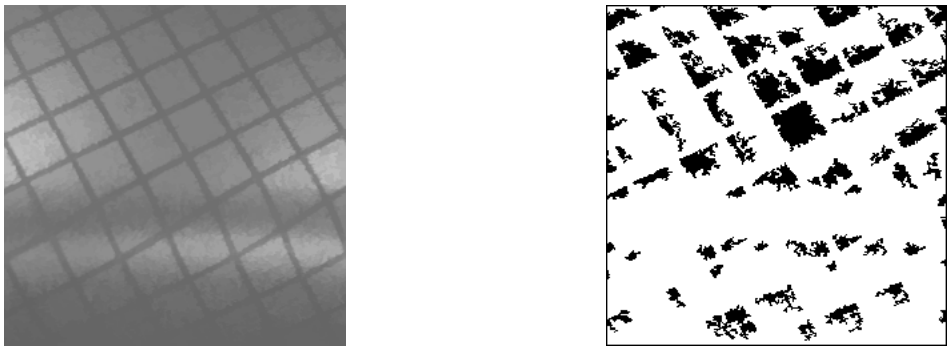
Exercise n° 9: Stamped grid in steel

1) Levellings are filtering operators which flatten the images. Two simple implementations of these filters are available in MAMBA: **simpleLevelling** and **strongLevelling**. Both of them use geodesic reconstructions. The simple levelling is symmetrical while the order of the initial operations (erosion or dilation) can be set up in the strong levelling.
Try the **strongLevelling** operator of size 2 on the initial *steel_sheet* image and extract the maxima of the filtered image:

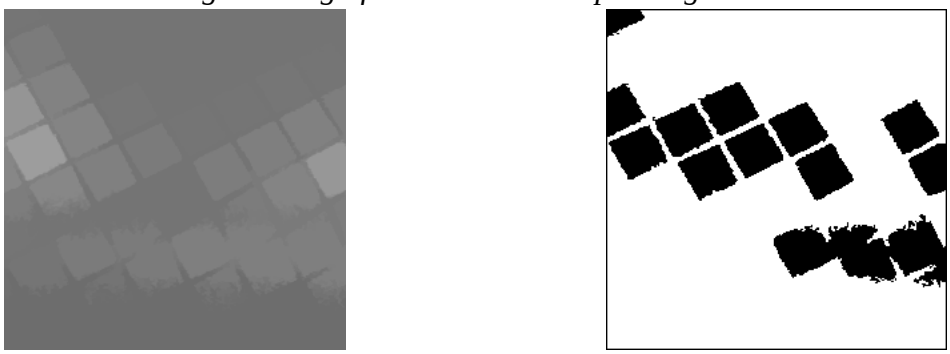
```
>>> strongLevelling(im1, im2, 2, eroFirst=False)
>>> maxima(im2, imbin1)
```

Some cells are correctly marked but some others are missed. Conversely, if the size of the levelling increases, some cells are better detected, but others are concatenated:

```
>>> strongLevelling(im1, im2, 14, eroFirst=False)
>>> maxima(im2, imbin1)
```



Strong levelling of size 2 and corresponding maxima.



Strong levelling of size 14 and corresponding maxima. Some cell markers are connected to the edge.

Therefore, a procedure named **extractGridMarkers** can be designed, where increasing sizes of levellings are used. At each step (except for the first one), the maxima which touch the edges are removed. The others are added (union) to the final result. The procedure ends when no inner maximum remains:

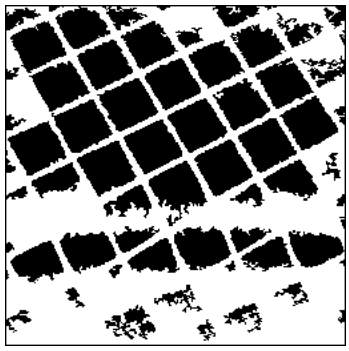
```
def extractGridMarkers(imIn, imOut):
    """
    This procedure extracts markers of the cells of the grid printed on the
    steel sheet (image 'imIn') and puts the result in binary image 'imOut'.
```

```
    The procedure uses the extraction of the maxima of successive strong
    levellings. At each step after the first one, the connected components
    of the maxima which touch the edge are removed. The process stops
    when no marker is available.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn, 1)
    # A first filtering with a strong levelling produces first seeds for the
    # markers (maxima of the filtered image).
    i = 2
    strongLevelling(imIn, imWrk1, i, eroFirst=False)
    maxima(imWrk1, imOut)
    v1 = computeVolume(imOut)
    # The same filtering is iterated for increasing sizes. Only the inner
    # maxima are preserved and added to the markers image. When no inner
    # marker is available, the process ends.
    while v1 <> 0:
        i += 2
        strongLevelling(imIn, imWrk1, i, eroFirst=False)
        maxima(imWrk1, imWrk2)
        removeEdgeParticles(imWrk2, imWrk2)
        logic(imWrk2, imOut, imOut, "sup")
        v1 = computeVolume(imWrk2)
```

Load the steel_sheet image in im1 and test the procedure:

```
>>> extractGridMarkers(im1, imbin1)
```

The cells are marked. Some cells contain more than one marker. They correspond to cells which are not homogeneous.



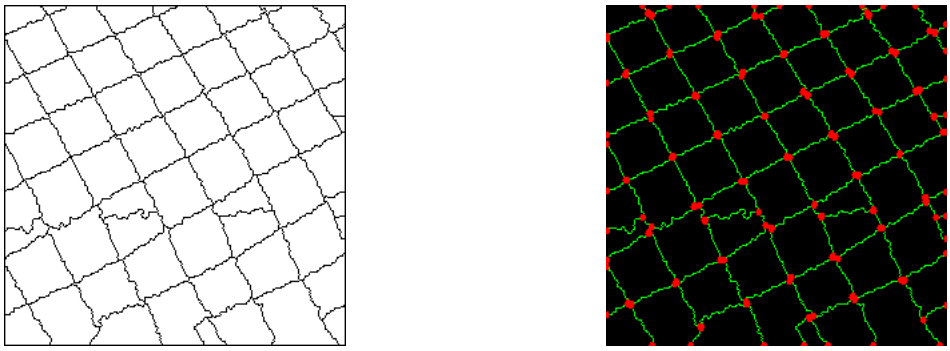
Markers of the grid cells.

Now, the marker-controlled watershed of the inverted initial image (not its gradient!) is performed. The binary watershed lines are stored in imbin2:

```
>> negate(im1, im2)
>>> label(imbin1, im32_1)
65
>>> watershedSegment(im2, im32_1)
>>> copyBitPlane(im32_1, 31, imbin2)
```

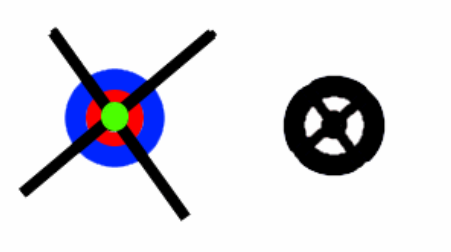
As indicated above, some cells are over-segmented. The multiple points of the watershed transform are extracted. Close points are connected by a dilation of size 2 followed by an intersection with the watershed lines in order to restrain the multiple points to the watershed:

```
>>> multiplePoints(imbin2, imbin3)
>>> dilate(imbin3, imbin4, 2)
>>> logic(imbin2, imbin4, imbin4, "inf")
```



Watershed transform of the inverted steel_sheet image (left) and its multiple points (right).

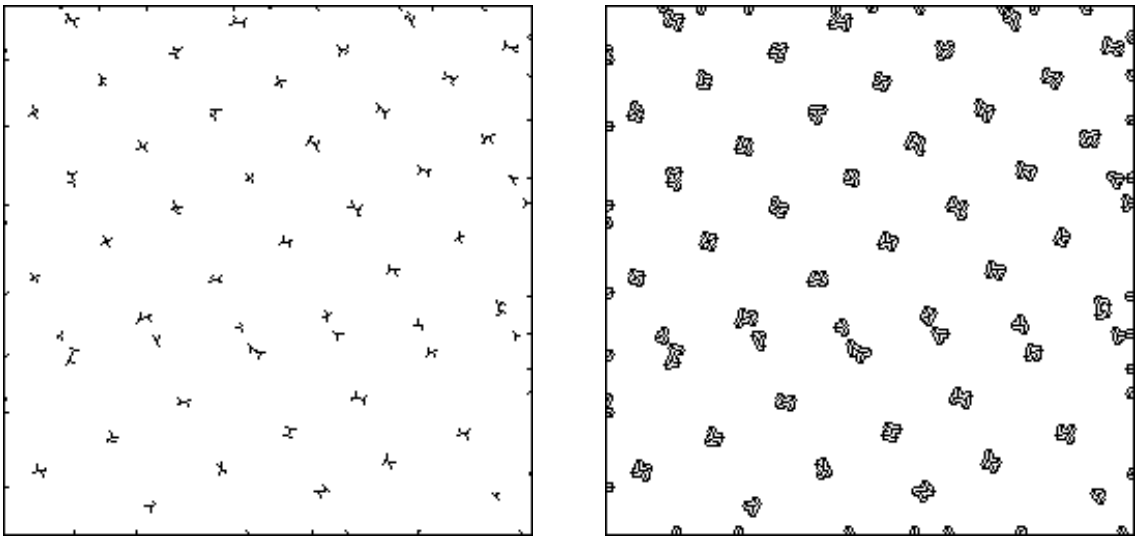
Among the detected multiple points, only those which are adjacent to four cells of the grid are relevant. It could be possible to sort these points individually. However, we prefer to design a general algorithm using a tool which has already been defined in the exercise n° 9, chapter 8: **holesLabelling**. The following illustration explains this algorithm which consists in dilating the multiple points twice and in removing from the second dilation the points added by the first one.



The multiple point (in green) is dilated twice. The points of the first dilation belonging to the cells (in red) are removed from the second dilation. The rsulting connected component contains as many holes as there are neighboring cells.

The following sequence implements the algorithm:

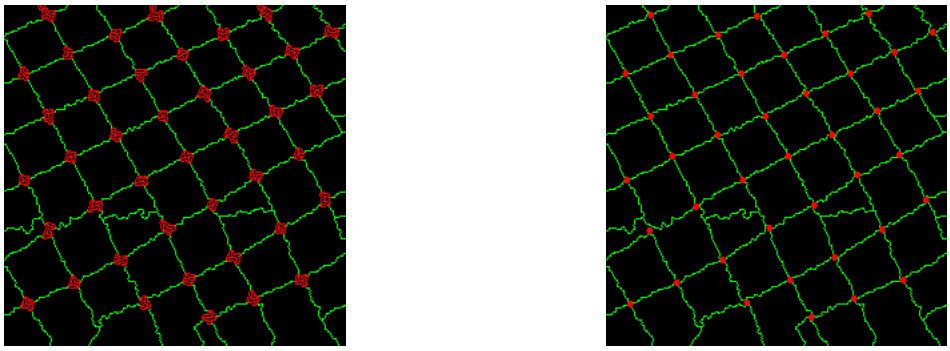
```
>>> dilate(imbin4, imbin1)
>>> imbin5 = imageMb(imbin1)
>>> dilate(imbin1, imbin5)
>>> diff(imbin1, imbin2, imbin3)
>>> diff(imbin5, imbin3, imbin3)
```



Multiple points after a slight concatenation (left), result of the algorithm which generates holes inside the dilated multiple points.

The **holesLabelling** procedure labels each connected component with its number of holes + 1. The relevant crossings of the grid are obtained by thresholding the labelled image at [5,5], value which corresponds to the number of holes (4) + 1:

```
>>> holesLabelling(imbin3, im32_1)
>>> threshold(im32_1, imbin5, 5, 5)
```



A [5,5] threshold extracts the relevant multiple points (left) which can be thinned to refine their position (right).

Exercise n° 10: Analysis of a burner

1) As the raster is oriented in vertical and horizontal directions, using a square grid is more convenient. Indeed, on the hexagonal grid, it is more difficult to generate the vertical direction (it requires to perform combinations of transformations in the 60° and 120° directions). In MAMBA, the square grid is defined with the **setDefaultGrid** operator:

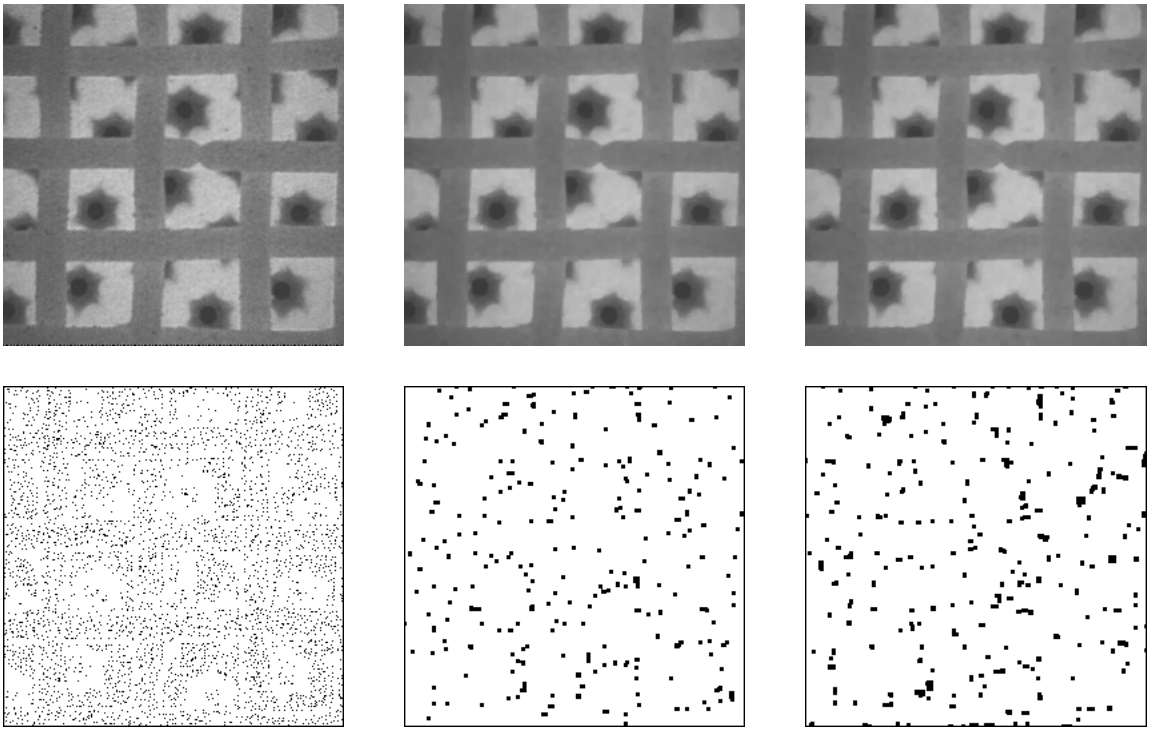
```
>>> setDefaultGrid(SQUARE)
```

When an operator uses a structuring element, the grid corresponding to this structuring element is automatically used. For instance a **SQUARE3X3** structuring element is used on a square grid.

2) The alternate sequential filters (ASF) with opening-closing or closing-opening by a square structuring element of size 1 give satisfactory results. There is no significant difference between these two types of filters. Load the *burner* image in im1 and type:

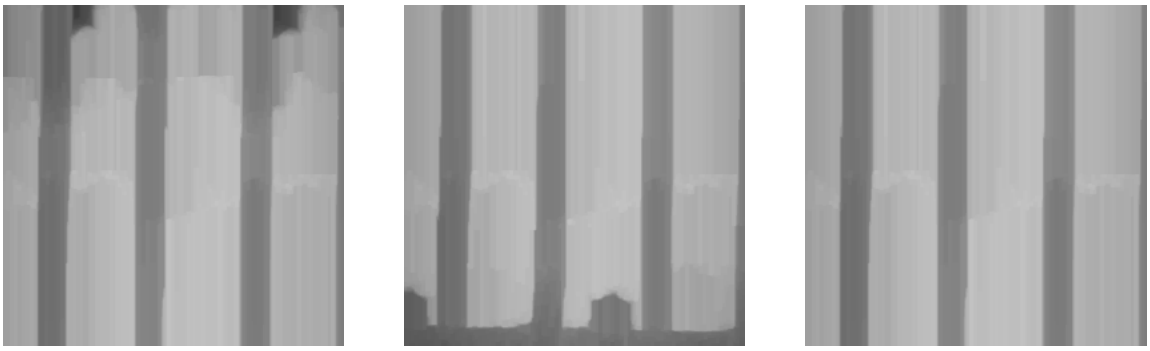
```
>>> alternateFilter(im1, im2, 1, True, se=SQUARE3X3)
>>> alternateFilter(im1, im3, 1, False, se=SQUARE3X3)
```

You can assess the efficiency of these filters by extracting the maxima of the original image and of the filtered ones. If both filters are very efficient to reduce noise, there is no big difference in the number of maxima between the two results.



Left: original image (top) and its maxima (bottom). Middle: alternate sequentiel filter of size 1 starting with an opening (top) and its maxima (bottom). Right: ASF with a closing first (top) and its maxima (bottom).

3) The raster is extracted by means of the following steps.



Vertical closing in direction 1 (left), in direction 5 (middle). Sup of the two closings (right).

First of all, linear vertical closings of large size are performed. Note that, to avoid edge effects, the two vertical directions (1 and 5) must be used. The sup of these two closings produces a satisfactory result:

```
>>> linearClose(im2, im3, 1, 100, grid=SQUARE)
>>> linearClose(im2, im4, 5, 100, grid=SQUARE)
>>> logic(im3, im4, im3, "sup")
```

The same procedure is applied in the horizontal direction (note the definition of a new greyscale image in im5):

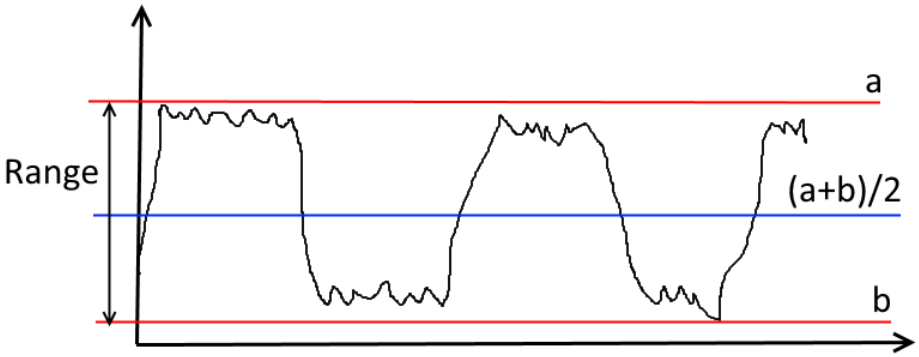
```
>>> linearClose(im2, im4, 3, 100, grid=SQUARE)
>>> im5 = imageMb(im1)
>>> linearClose(im2, im5, 7, 100, grid=SQUARE)
>>> logic(im4, im5, im4, "sup")
```

Finally, the infimum of the two preceding results provides a pretty good extraction of the raster:

```
>>> logic(im3, im4, im3, "inf")
```

Extraction of the horizontal components of the grid (left), final result (right).

As the image is almost a two-phased image, a binary raster can be obtained by a simple threshold in the middle of the grey values range.



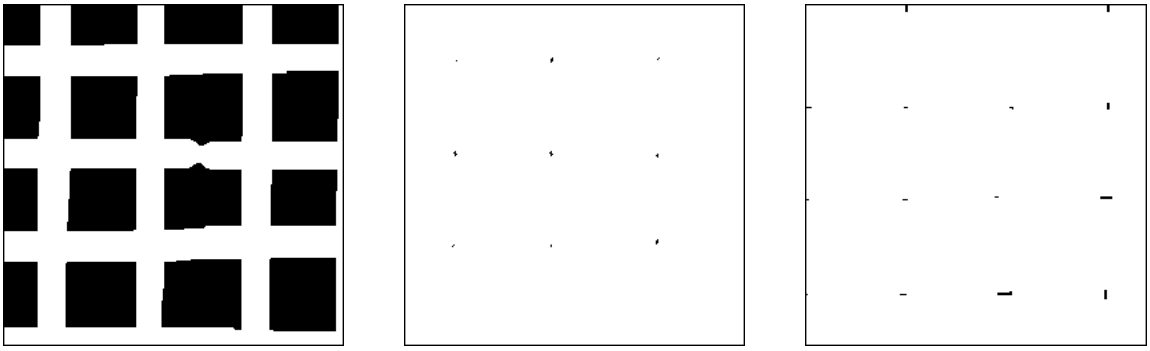
Principle of the automatic thresholding of a two-phased image; the extremal grey values a and b are computed and a threshold can be applied in the middle of the range.

Apply this to the raster image:

```
>>> computeRange(im3)
(109, 196)
>>> threshold(im3, imbin3, 0, 152)
```

However, a more refined extraction can be obtained by a watershed transform applied on the grey raster image. For this, markers of the raster can be defined by an ultimate erosion of the previous coarse binary raster image, while markers of the background are obtained with an ultimate erosion of the complementary binary set. Using ultimate erosions insure that the markers are centered on the regions to be segmented.

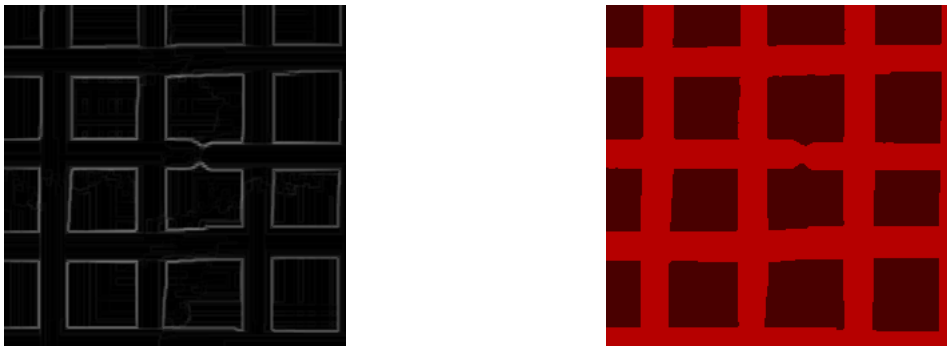
```
>>> ultimateErosion(imbin3, imbin1, im32_1)
>>> negate(imbin3, imbin3)
>>> ultimateErosion(imbin3, imbin2, im32_1)
```



First binary image of the raster obtained by a simple thresholding (left). Markers of the raster (middle) and of the background (right) obtained by ultimate erosions.

Then, a label image is built: the raster markers are given a label value equal to 2 while the label of the background markers is equal to 1. The watershed of the gradient of the grey raster image is performed with the **basinSegment** operator (no boundary is generated):

```
>>> im32_1.reset()
>>> add(im32_1, imbin1, im32_1)
>>> add(im32_1, imbin1, im32_1)
>>> add(im32_1, imbin2, im32_1)
>>> gradient(im3, im4)
>>> basinSegment(im4, im32_1)
```



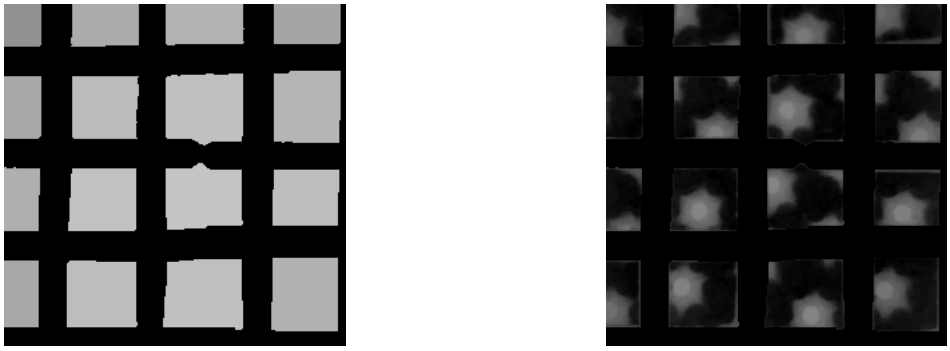
Gradient of the raster image (left) and result of the watershed transform (right).

The markers of the raster (in imbin1) and the gradient of the raster image are preserved as they will be use for the extraction of the other features. The gradient is stored in the im5 image:

```
>>> copy(im4, im5)
```

4) Extraction of the hexagonal structures
The previous segmentation can be used to restrain the search of the hexagonal structures inside the background region. This increases the robustness of the process.
A greyscale image of the background (labelled 1 in the previous segmentation) is generated (each background pixel takes the value 255) and used as mask in a geodesic reconstruction. each cell of the background is valued with its maximal grey value. Subtracting the initial filtered image from this reconstructed image extracts the hexagonal structures:

```
>>> threshold(im32_1, imbin2, 1, 1)
>>> convert(imbin2, im4)
>>> copy(im2, im3)
>>> hierarBuild(im4, im3)
>>> sub(im3, im2, im4)
```



Each background cell is valued with its maximal grey value (left). Subtracting the initial filtered image extracts the hexagonal structures.

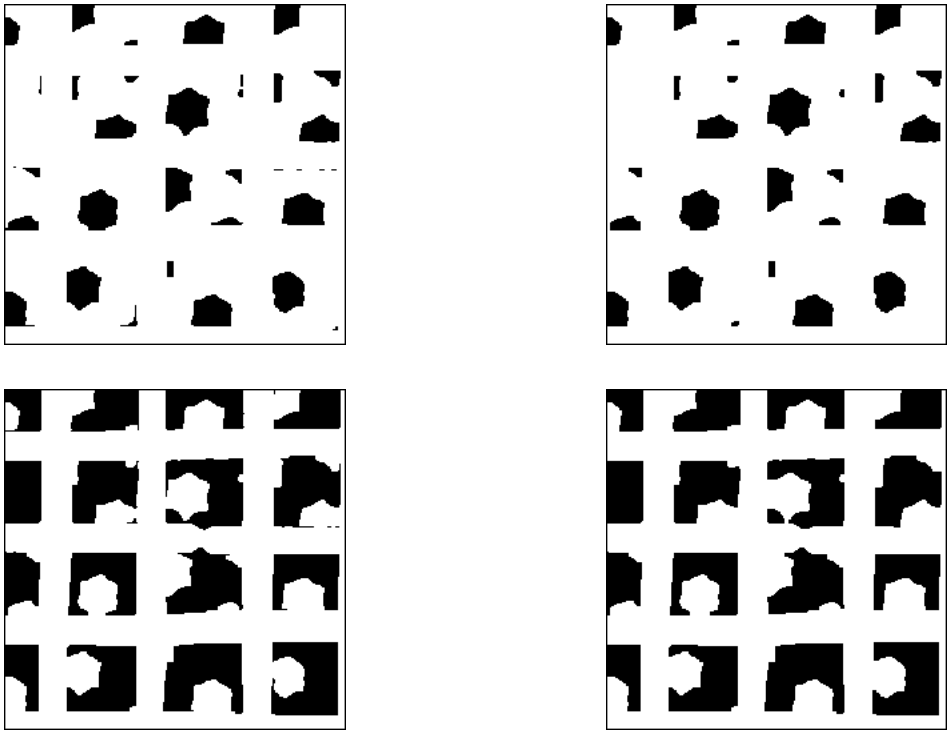
Here again, a first binary image of the hexagonal structures can be obtained by a simple thresholding of the im4 image in the middle of its range of grey values:

```
>>> computeRange(im4)
(0, 134)
>>> threshold(im4, imbin3, 67, 255)
```

A more refined segmentation can also be performed by a marker-controlled watershed. The process is identical to the one used for the raster segmentation. The binary sets corresponding to the hexagonal structures and to the background obtained by thresholding and complementation are opened in order to remove small defects at the boundary of the background.:

```
>>> opening(imbin3, imbin4, se=SQUARE3X3)
>>> diff(imbin2, imbin4, imbin3)
>>> opening(imbin3, imbin3, se=SQUARE3X3)
```

The ultimate erosion of these filtered sets is performed. The components of the ultimate erosion can be used as markers of the hexagonal structures and of the background. We could now build the label image used by the watershed transformation.



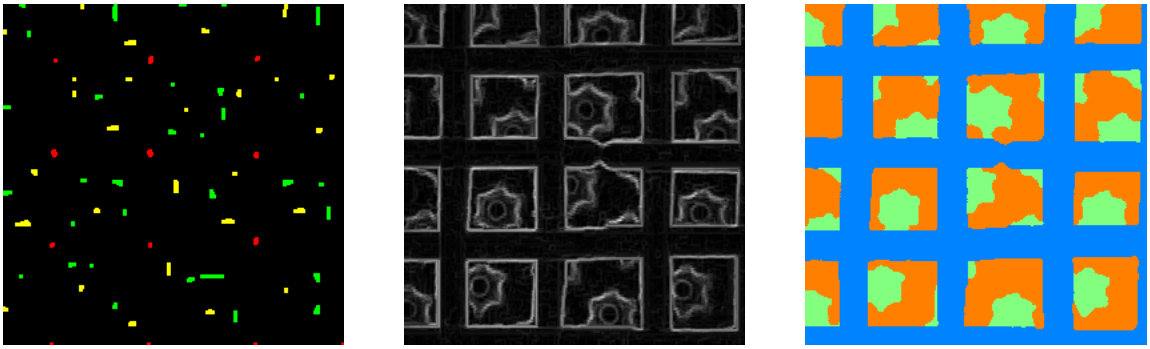
Hexagonal structures extracted by thresholding (upper left), their opening (upper right), background (lower left) and its opening (lower right).

However, as we want also to obtain the segmentation of the raster, we must add to the label image the markers of the raster which have been previously generated and stored in the imbin1 image (the label is equal to 3 for the markers of the hexagonal structures, equal to 2 for the background markers and to 1 for the raster markers) :

```
>>> ultimateErosion(imbin3, imbin5, im32_2)
>>> convertByMask(imbin5, im32_1, 0, 3)
>>> ultimateErosion(imbin4, imbin5, im32_2)
>>> add(im32_1, imbin5, im32_1)
>>> add(im32_1, imbin5, im32_1)
>>> add(im32_1, imbin1, im32_1)
```

Moreover, the marker-controlled watershed is applied to the supremum of the gradient of the initial filtered image (stored in im2) and of the gradient of the raster image (stored in im5). This procedure allows to better define the boundaries between the raster and the hexagonal structures:

```
>>> gradient(im2, im3)
>>> logic(im3, im5, im5, "sup")
>>> basinSegment(im5, im32_1)
```



Left: markers of the background (in green), of the hexagonal structures (in yellow) and of the raster (in red). Middle: supremum of the gradient images of the initial filtered image and of the raster image. Right: watershed segmentation.

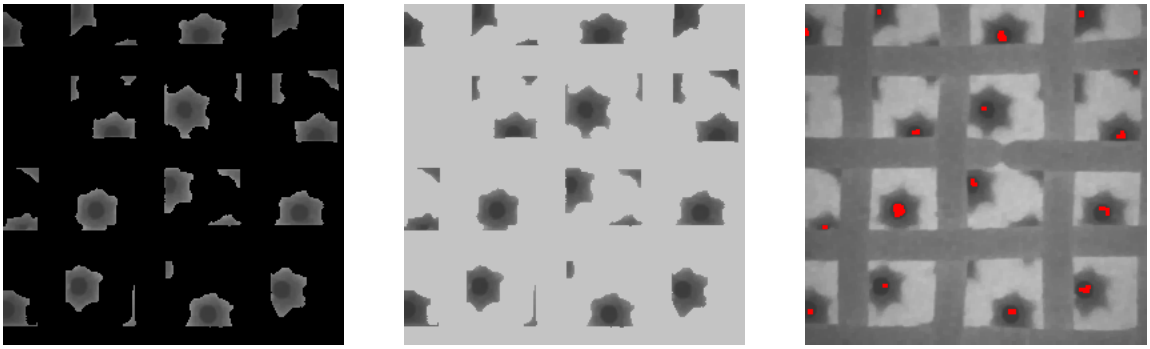
5) Extraction of the black spots

To extract the spots, we start by building an image where the spots will be the most significant minima. We first extract the region containing the hexagonal structures. Then we fill the outside with the maximal grey value in the initial filtered image (it corresponds to the background). This filling is achieved by a geodesic reconstruction:

```
>>> threshold(im32_1, imbin3, 2, 2)
>>> convert(imbin3, im3)
>>> logic(im2, im3, im4, "inf")
>>> negate(im3, im5)
>>> copy(im2, im3)
>>> hierarBuild(im5, im3)
>>> logic(im3, im4, im3, "sup")
```

The range of grey values in this image gives the maximum depth of the minima (the spots). These spots can be marked by the **minDynamics** operator. Minima with a dynamics greater than or equal to 120 (that is approximately 5/6 of the maximum depth) are extracted and thinned. They constitute the markers of the spots:

```
>>> computeRange(im3)
(53, 196)
>>> minDynamics(im3, imbin4, 120)
>>> thinD(imbin4, imbin4)
```



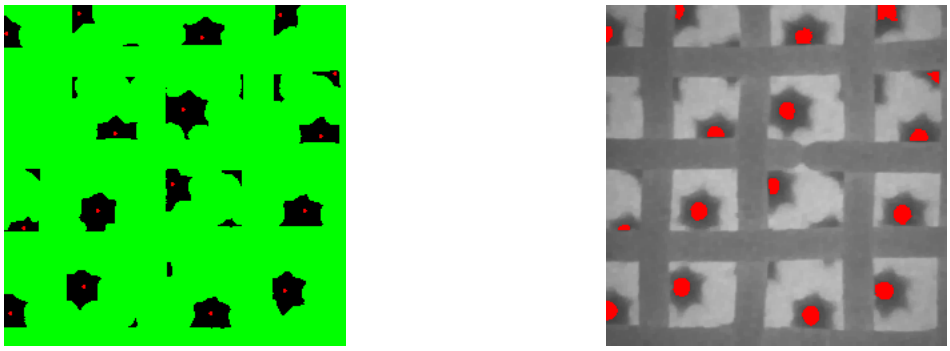
Extracted hexagonal structures (left), filling of the outside (middle), minima with a high dynamics (right, in red).

The outside marker is composed of the union of the raster and of the dilated background. A size 2 dilation of the background set is performed to insure that the outside marker overrides the boundaries of the hexagonal structures and that the watershed flooding will not be blocked by these boundaries:

```
>>> threshold(im32_1, imbin3, 3, 3)
>>> imbin3.show()
>>> dilate(imbin3, imbin3, 2)
>>> threshold(im32_1, imbin2, 1, 1)
>>> logic(imbin3, imbin2, imbin3, "sup")
```

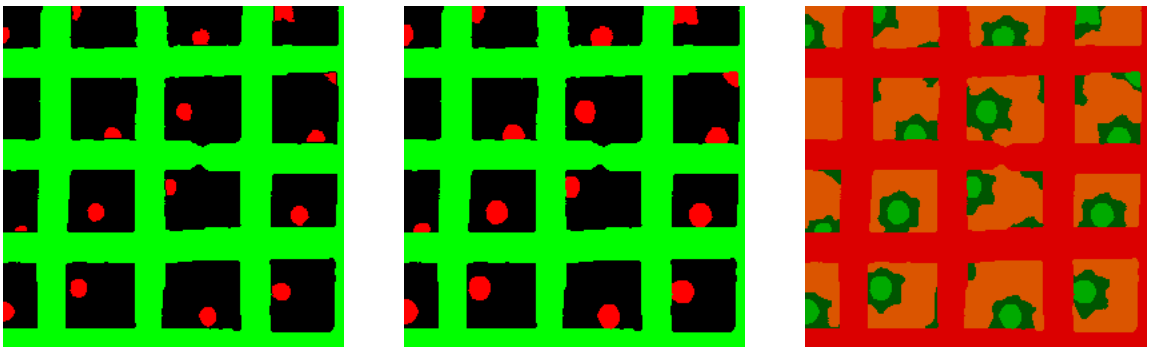
The watershed transform of the gradient of the inital image is realised, controlled by these markers:

```
>>> im32_2.reset()
>>> add(im32_2, imbin3, im32_2)
>>> add(im32_2, imbin3, im32_2)
>>> add(im32_2, imbin4, im32_2)
>>> gradient(im2, im5)
>>> basinSegment(im5, im32_2)
```



Left: markers of the spots (red) and of the outside (green). Right: extraction of the spots by the watershed transform.

6) Due to biases in the watershed transform, some spots are not adjacent to the raster even though they should be. To overcome this, a geodesic dilation of the spots inside the background cells can be performed:



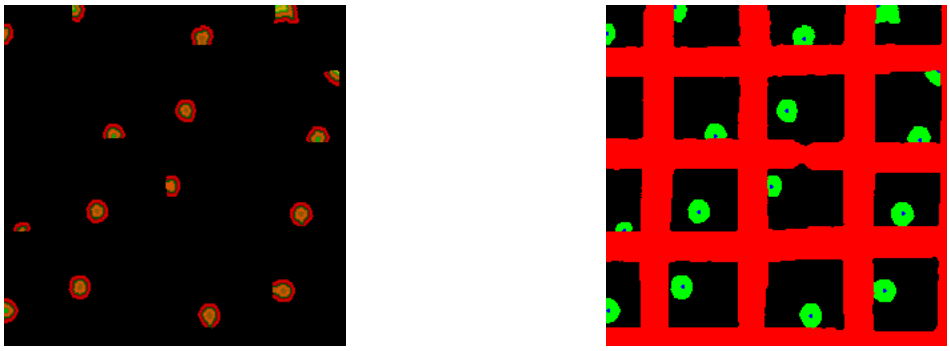
Initial spots (left), spots after a geodesic dilation (middle), complete segmentation showing the raster, the hexagonal structures and the spots.

```
>>> threshold(im32_2, imbin5, 1, 1)
```

```
>>> threshold(im32_1, imbin4, 1, 1)
>>> negate(imbin4, imbin3)
>>> geodesicDilate(imbin5, imbin3, imbin2, 2)
```

A better localisation of the centers of the spots which are touching or partly hidden by the raster can be obtained if the distance function of the set made of the union of the spots and the raster is computed first. Then, the extraction of the maxima of the restriction of this distance function to the spots (filtered by an elementary opening) allows to better locate the centers of the spots. A **thinD** operator reduces these centers to a single point:

```
>>> logic(imbin2, imbin4, imbin1, "sup")
>>> computeDistance(imbin1, im32_2, edge=FILLED)
>>> convert(im32_2, im3)
>>> convert(imbin2, im4)
>>> opening(im3, im3)
>>> logic(im3, im4, im4, "inf")
>>> maxima(im4, imbin3)
>>> thinD(imbin3, imbin3)
```



Restriction to the spots of the distance function of the union of the spots and the raster (left). Spots in green, raster in red and centers of the spots in blue (right).

REFERENCES

{1] S.Beucher: The watershed transformation applied to image segmentation
(**http://cmm.ensmp.fr/~beucher/publi/pfefferkorn.pdf**)
This paper and the following introduce the concepts of watersheds and of waterfalls.

{2] S.Beucher, F.Meyer: The Morphological approach of segmentation
(**http://cmm.ensmp.fr/~beucher/publi/SB_watershed.pdf**)

[3] N.Beucher, S.Beucher: Hierarchical Queues: general description and implementation in MAMBA Image library
(**http://cmm.ensmp.fr/~beucher/publi/HQ_algo_desc.pdf**)
This paper describes thoroughly the hierarchical queues (HQ) which are implemented in the MAMBA image library. The use of these HQ to realize watershed transforms and geodesic reconstructions is fully explained. The extension of these morphological operators to 32-bit images is also described.

[4] S.Beucher, B.Marcotegui: P algorithm, a dramatic enhancement of the waterfall transformation
(**http://cmm.ensmp.fr/~beucher/publi/P-Algorithm_SB_BM.pdf**)
This document describes an efficient enhancement of the waterfall algorithm, a hierarchical segmentation algorithm defined from the watershed transformation. The first part of the document recalls the definition of the waterfall algorithm, its various avatars as well as its links with the geodesic reconstruction. The second part starts by analyzing the different shortcomings of the algorithm and introduces several strategies to palliate them. Two enhancements are presented, the first one named standard algorithm and the second one, P algorithm. The different properties of P algorithm are analyzed. This analysis is detailed in the last part of the document. The performances of the two algorithms, in particular, are addressed and their analogies with perception mechanisms linked to the brightness constancy phenomenon are discussed.

[5] S.Beucher: Towards a unification of waterfalls, standard and P algorithms
(**http://cmm.ensmp.fr/~beucher/publi/Unified_Segmentation.pdf**)
This document is an extension of the above paper. It comes back to the waterfalls, standard and P algorithms to propose a general segmentation scheme which covers and unifies these different processes.

[6] S.Beucher, X.Yu: Road recognition in complex traffic situations
(**http://cmm.ensmp.fr/~beucher/publi/ifac.pdf**)
This paper explains how to use the watershed transform fotr road segmentation.

Conclusion

CONGRATULATIONS!

If you arrived to this point after a thorough study of this manual, you deserve congratulations! This exercise handbook contains approximately fifty exercises and some of them are quite difficult. Even if you just skimmed through them, we hope that you could find ideas for designing solutions to your own image analysis applications. Moreover, we are sure now that you are convinced that using Mathematical Morphology does not simply consist in applying basic operators on binary images. The various tools available in the MAMBA Image library can be used efficiently for image segmentation, feature extraction and characterization. These operators are fast, they work on images of various sizes and depths. It is also possible to build your own tools as this was done in fact in many exercises.

Be aware, however, that these exercises have unveil a small part of the capabilities of Mathematical Morphology in general and of the MAMBA library in particular. If the algorithmic chapter has been put forward, many other aspects have not been tackled: color images (although the operators allowing to work on these images exist in MAMBA), 3D images (except in one exercise; remind that **mamba3D** contains many 3D operators), morphology on graphs (although powerful operators dealing with partitions, a dual representation of graphs, exist in MAMBA), stochastic morphology (random sets, random models, simulations), etc.

Nevertheless, if this exercise handbook makes you want to go further, it has fulfilled its purpose.

Annex

MAMBA SOURCES

This annex contains the source listing of the Python file **Mamba_solutions.py**. This file collects all the MAMBA operators defined in the exercises.

```
# This file contains the various mamba functions providing the solutions of
# the exercises contained in the "Mathematical Morphology Exercises with Mamba"
# document (version 2).

from mamba import *

def covariance(imIn, dir, sizeRange, grid=DEFAULT_GRID):
  """
  This operator calculates the covariance of image 'imIn' in the direction
  'dir' of 'grid' for all tha sizes in 'size_range'.
  It returns a list of real values.
  """

  imField = imageMb(imIn, 1) # Binary measurement field
  imWrk = imageMb(imIn) # Working image
  covarList = [] # Initializing the covariance list
  for i in range(sizeRange):
    doublePointErode(imIn, imWrk, dir, i, grid=grid, edge=EMPTY)
    v1 = computeVolume(imWrk)
    # Performing the erosion by a doublet of points and measuring
    # its area. Note the setting of edge!
    imField.fill(1)
    doublePointErode(imField, imField, dir, i, grid=grid, edge=EMPTY)
    v2 = computeVolume(imField)
    # Generating the current unbiased measurement mask and measuring
    # its area.
    covarValue = float(v1) / v2
    covarList.append(covarValue)
  return covarList

def particleBoundingBoxes(imIn):
  """
  This operator returns a list containing the horizontal and vertical dimensions
  (also called Feret diameters) of all the particles contained in the binary image
  'imIn'.
  """

  # Defining working images
  imWrk = imageMb(imIn)
  imWrk32 = imageMb(imIn, 32)
  # Initializing the list of results
  feretDiametersList = []
  # Labelling the binary image. nb contains the number of particles
  nb = label(imIn, imWrk32)
  # This loop calculates the size of the bounding box for each connected component
  for i in range(nb):
    threshold(imWrk32, imWrk, i+1, i+1)
    box = computeFeretDiameters(imWrk)
    feretDiametersList.append(box)
```

```
    # Returning the list of results
    return feretDiametersList

def doublePointOpen(imIn, imOut, dir, n, grid=DEFAULT_GRID, edge=FILLED):
    """
    Performs an opening by a doublet of points of size 'n' in direction 'dir'.
    'edge' is set to 'FILLED' by default.
    """

    doublePointErode(imIn, imOut, dir, n, edge=edge, grid=grid)
    doublePointDilate(imOut, imOut, transposeDirection(dir, grid=grid), n, grid=grid)

def doublePointClose(imIn, imOut, dir, n, grid=DEFAULT_GRID, edge=FILLED):
    """
    Performs a closing by a doublet of points of size 'n' in direction 'dir'.
    If 'edge' is set to 'EMPTY', the operation must be modified to remain extensive.
    """

    imWrk = imageMb(imIn)
    if edge==EMPTY:
        copy(imIn, imWrk)
    doublePointDilate(imIn, imOut, dir, n, grid=grid)
    doublePointErode(imOut, imOut, transposeDirection(dir, grid=grid), n, edge=edge, grid=grid)
    if edge==EMPTY:
        logic(imOut, imWrk, imOut, "sup")

def dodecagonalOpening(imIn, imOut, n=1, edge=FILLED):
    """
    Performs an opening operation on image 'imIn' with a dodecagon and puts the result in 'imOut'.
    'n' controls the size of the opening.
    The default edge is set to 'FILLED'. Note that the edge setting operates in the
    erosion only.
    """

    dodecagonalErode(imIn, imOut, n, edge=edge)
    dodecagonalDilate(imOut, imOut, n, edge=EMPTY)

def granulometricMeasure(imIn, n):
    """
    Granulometric measure (size distribution measure) of the binary image
    'imIn' after an hexagonal opening of size n.
    This function returns a real value between 0 and 1.
    """

    imWrk1 = imageMb(imIn, 1)
    imWrk2 = imageMb(imIn, 1)
    opening(imIn, imWrk1, n)
    diff(imIn, imWrk1, imWrk2)
    imWrk1.fill(1)
    erode(imWrk1, imWrk1, 2*n, edge=EMPTY)
    logic(imWrk2, imWrk1, imWrk2, "inf")
    measure1 = computeVolume(imWrk2)
    logic(imIn, imWrk1, imWrk2, "inf")
    measure2 = computeVolume(imWrk2)
    if measure2 <> 0:
        granulometry = float(measure1)/measure2
    else:
        granulometry = 0
    return granulometry
```

```
def granulometry(imIn, size):
  """
  Computes the granulometry of binary image 'imIn' in the range (0, 'size' -1) by an hexagonal
  opening.
  """

  granuList =[]
  for i in range(size):
     mes = granulometricMeasure(imIn, i)
     granuList.append(mes)
  return granuList

def contrast(imIn, imOut, size, type):
  """
  Contrast operators applied on image 'imIn'. The result is in 'imOut'. The size
  of the operators is given by 'size'.
  'type' allows to select the type of operators:
  'type' < 1, the two contrast operators are erosion and dilation of size 'size'.
  'type' = 1, the two operators are opening and closing of size 'size'.
  'type' > 1, the first operator is an opening of size 'size', the second one
  is a closing of size 5*size.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  imWrk3 = imageMb(imIn)
  imWrk4 = imageMb(imIn)
  imMask = imageMb(imIn, 1)
  if type > 0:
     opening(imIn, imWrk1, size)
     size1 = size
     if type > 1:
        size1 = size * 5
     closing(imIn, imWrk2, size1)
  else:
     erode(imIn, imWrk1, size)
     dilate(imIn, imWrk2, size)
  sub(imIn, imWrk1, imWrk3)
  sub(imWrk2, imIn, imWrk4)
  generateSupMask(imWrk4, imWrk3, imMask, False)
  convertByMask(imMask, imOut, 0, computeMaxRange(imIn)[1])
  logic(imWrk1, imOut, imOut, "inf")
  negate(imMask, imMask)
  convertByMask(imMask, imWrk4, 0, computeMaxRange(imIn)[1])
  logic(imWrk2, imWrk4, imWrk4, "inf")
  logic(imWrk4, imOut, imOut, "sup")

def contrastByTopHat(imIn, imOut, size):
  """
  Contrast by top-hat of size 'size' of 'imIn', result in 'imOut'.
  The final image can be identical to the initial one.
  For greyscale images, the arithmetic operations are truncated.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  copy(imIn, imWrk1)
  whiteTopHat(imIn, imWrk2, size)
```

```
    add(imIn, imWrk2, imOut)
    blackTopHat(imWrk1, imWrk2, size)
    sub(imOut, imWrk2, imOut)

def autoMedian2(imIn, imOut, n):
    """
    Morphological automedian filter performed with an alternance
    closing/opening/closing and opening/closing/opening.
    """

    oco_im = imageMb(imIn)
    coc_im = imageMb(imIn)
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    alternateFilter(imIn, oco_im, n, True)
    opening(oco_im, oco_im, n)
    alternateFilter(imIn, coc_im, n, False)
    closing(coc_im, coc_im, n)
    copy(coc_im, imWrk1)
    logic(oco_im, imWrk1, imWrk1, "sup")
    copy(coc_im, imWrk2)
    logic(oco_im, imWrk2, imWrk2, "inf")
    copy(imIn, imOut)
    logic(imOut, imWrk2, imOut, "sup")
    logic(imOut, imWrk1, imOut, "inf")

def compareImages(imIn1, imIn2):
    """
    Compares the two images 'imIn1' and 'imIn2' and returns the number of
    modified pixels between them.
    """

    imWrk1 = imageMb(imIn1, 1)
    imWrk2 = imageMb(imIn2, 1)
    generateSupMask(imIn1, imIn2, imWrk1, True)
    generateSupMask(imIn2, imIn1, imWrk2, True)
    logic(imWrk1, imWrk2, imWrk1, "sup")
    pixdiff = computeVolume(imWrk1)
    return pixdiff

def closeFibers(imIn, imOut):
    """
    This operation cleans the original fibers image by closing their contours
    and by filling their interiors. Note that this operation also fills the
    fibers which are cutting the edges of the image.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)

    negate(imIn, imWrk1)
    # Linear dilation of size 2 in horizontal direction to connect fibers
    # contours.
    linearDilate(imWrk1, imWrk1, 5, 2)
    # Filling true holes (interior fibers)
    closeHoles(imWrk1, imWrk2)
    # Extracting all the fibers cutting the edges.
    removeEdgeParticles(imWrk2, imWrk3)
```

```
    diff(imWrk2, imWrk3, imWrk3)
    # The image is inverted and eroded to get a background marker.
    negate(imWrk3, imWrk3)
    erode(imWrk3, imOut, n=20, edge=EMPTY)
    # Reconstructing the background and adding fibers cutting the edges to
    # the previous ones.
    build(imWrk3, imOut)
    diff(imOut, imWrk2, imWrk2)
    # Another linear dilation in transposed direction to recover the initial
    # sizes of the fibers.
    linearDilate(imWrk2, imWrk2, 2, 2)
    # final result
    negate(imWrk2, imOut)

def checkEvenness(imIn, maxSize):
    """
    Checks the regularity of the boron fibers arrangement by computing
    successive closings and by determining at each step the connectivity
    number of the closing. The variation of this measure from a positive
    to a negative value indicates the evenness of the arrangement. The
    more it is regular, the more this variation is fast and important.
    The successive connectivity numbers are returned in a list.
    """

    imWrk = imageMb(imIn)
    ncList =[]
    for i in range(maxSize + 1):
        closing(imIn, imWrk, i)
        nc = computeConnectivityNumber(imWrk)
        ncList.append(nc)
    return ncList

def openSizeDistribution(imIn, sizeRange):
    """
    Computes the size distribution by hexagonal openings of the 'imIn' image
    in the range 'sizeRange'. The operator returns a list of values.
    """

    imWrk = imageMb(imIn)
    copy(imIn, imWrk)
    granuList = []
    oldVol = 0L
    for i in range(1, sizeRange + 1):
        opening(imIn, imWrk, i)
        sub(imIn, imWrk, imWrk)
        vol = computeVolume(imWrk)
        volInc = vol - oldVol
        granuList.append(volInc)
        oldVol = vol
    return granuList

def buildOpenSizeDistribution(imIn, sizeRange):
    """
    Computes the size distribution by openings by reconstruction of the
    'imIn' image in the range 'sizeRange'. The operator returns a list of values.
    """

    imWrk = imageMb(imIn)
    copy(imIn, imWrk)
```

```
    granuList = []
    oldVol = 0L
    for i in range(1, sizeRange + 1):
        buildOpen(imIn, imWrk, i)
        sub(imIn, imWrk, imWrk)
        vol = computeVolume(imWrk)
        volInc = vol - oldVol
        granuList.append(volInc)
        oldVol = vol
    return granuList

def buildSupOpen(imIn, imOut, size):
    """
    Performs on 'imIn' the opening by reconstruction with a marker made of a sup of
    linear openings. The result is put in 'imOut'.
    """

    imWrk = imageMb(imIn)
    supOpen(imIn, imWrk, size)
    hierarBuild(imIn, imWrk)
    copy(imWrk, imOut)

def buildSupOpenSizeDistribution(imIn, sizeRange):
    """
    Computes the size distribution by openings by reconstruction with a sup marker of the
    'imIn' image in the range 'sizeRange'. The operator returns a list of values.
    """

    imWrk = imageMb(imIn)
    copy(imIn, imWrk)
    granuList = []
    oldVol = 0L
    for i in range(1, sizeRange + 1):
        buildSupOpen(imIn, imWrk, i)
        sub(imIn, imWrk, imWrk)
        vol = computeVolume(imWrk)
        volInc = vol - oldVol
        granuList.append(volInc)
        oldVol = vol
    return granuList

def granulNumber(imIn, sizeRange):
    """
    Computes the pseudo size distribution function provided by the number of maxima
    in the various openings in the range 'sizeRange'. The operator returns a list of
    values.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn, 1)
    imWrk3 = imageMb(imIn, 32)
    numberList = []
    for i in range(sizeRange+1):
        opening(imIn, imWrk1, i)
        maxima(imWrk1, imWrk2)
        nb = label(imWrk2, imWrk3)
        numberList.append(nb)
    return numberList
```

```
def directionalGradient(imIn, imOut, d):
    """
    Basic directional gradient of size 1, computed on the hexagonal grid in direction 'd'.
    """

    imWrk = imageMb(imIn)
    linearDilate(imIn, imWrk, d)
    linearErode(imIn, imOut, d)
    sub(imWrk, imOut, imOut)

def vectorGradient(imIn, imModul, imAzim):
    """
    Modulus and azimuth of the gradient of image 'imIn'. The modulus is stored in
    'imModul' and the azimuth (6 directions are available) in image 'imAzim'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)
    imWrk4 = imageMb(imIn, 1)
    imModul.reset()
    imAzim.reset()
    copy(imIn, imWrk1)
    for i in range(6):
        d = i+1
        directionalGradient(imWrk1, imWrk2, d)
        generateSupMask(imWrk2, imModul, imWrk4, True)
        convertByMask(imWrk4, imWrk3, 0, d)
        logic(imWrk2, imModul, imModul, "sup")
        logic(imWrk3, imAzim, imAzim, "sup")

def greyThin(imIn, imOut, dse):
    """
    Grey thinning of the 'imIn' image by the double structuring element 'dse'.
    For sake of simplicity, this operator is defined only on the hexagonal grid and
    with greyscale (8-bit) images.
    """

    imDil = imageMb(imIn)
    imEro = imageMb(imIn)
    mask1 = imageMb(imIn, 1)
    mask2 = imageMb(imIn, 1)
    gmask = imageMb(imIn)
    dilate(imIn, imDil, se=dse.getStructuringElement(0))
    erode(imIn, imEro, se=dse.getStructuringElement(1))
    generateSupMask(imIn, imDil, mask1, True)
    generateSupMask(imEro, imIn, mask2, False)
    logic(mask1, mask2, mask1, "inf")
    convertByMask(mask1, gmask, 0, 255)
    logic(gmask, imDil, imDil, "inf")
    negate(gmask, gmask)
    logic(gmask, imIn, imOut, "inf")
    logic(imDil, imOut, imOut, "sup")

def greyThick(imIn, imOut, dse):
    """
    Grey thickening of the 'imIn' image by the double structuring element 'dse'.
    For sake of simplicity, this operator is defined only on the hexagonal grid and
    with greyscale (8-bit) images.
```

```
    """

    imDil = imageMb(imIn)
    imEro = imageMb(imIn)
    mask1 = imageMb(imIn, 1)
    mask2 = imageMb(imIn, 1)
    gmask = imageMb(imIn)
    dilate(imIn, imDil, se=dse.getStructuringElement(0))
    erode(imIn, imEro, se=dse.getStructuringElement(1))
    generateSupMask(imIn, imDil, mask1, False)
    generateSupMask(imEro, imIn, mask2, True)
    logic(mask1, mask2, mask1, "inf")
    convertByMask(mask1, gmask, 0, 255)
    logic(gmask, imEro, imEro, "inf")
    negate(gmask, gmask)
    logic(gmask, imIn, imOut, "inf")
    logic(imEro, imOut, imOut, "sup")

def rotatingGreyThin(imIn, imOut, dse):
    """
    Performs a complete rotation of  grey thinnings , the initial 'dse' double
    structuring element being turned one step clockwise after each thinning.
    At each rotation step, the previous result is used as input for the next
    thinning (chained thinnings). This operator works only on the hexagonal grid
    and on 8-bit images.
    """

    copy(imIn, imOut)
    for i in range(6):
        greyThin(imOut, imOut, dse)
        dse = dse.rotate()

def rotatingGreyThick(imIn, imOut, dse):
    """
    Performs a complete rotation of grey thickenings, the initial 'dse' double
    structuring element being turned one step clockwise after each thickening.
    At each rotation step, the previous result is used as input for the next
    thickening (chained thickenings). This operator works only with 8-bit images
    and on the hexagonal grid.
    """

    copy(imIn, imOut)
    for d in range(6):
        greyThick(imOut, imOut, dse)
        dse = dse.rotate()

def fullGreyThin(imIn, imOut, dse):
    """
    Performs a complete grey thinning of 'imIn' with the successive rotations of 'dse'
    (until idempotence) and puts the result in 'imOut'.
    Works with greyscale images and on the hexagonal grid.
    """

    copy(imIn, imOut)
    v1 = computeVolume(imOut)
    v2 = 0
    while v1 != v2:
        v2 = v1
        rotatingGreyThin(imOut, imOut, dse)
```

```
    v1 = computeVolume(imOut)

def fullGreyThick(imIn, imOut, dse):
  """
  Performs a complete grey thickening of 'imIn' with the successive rotations of 'dse'
  (until idempotence) and puts the result in 'imOut'.
  Works with greyscale images and on the hexagonal grid.
  """

  copy(imIn, imOut)
  v1 = computeVolume(imOut)
  v2 = 0
  while v1 != v2:
    v2 = v1
    rotatingGreyThick(imOut, imOut, dse)
    v1 = computeVolume(imOut)

def thinByD3(imIn, imOut):
  """
  Controlled thinning by the D3 structuring element using a mixture of
  rotational thinnings and parallel ones in order to preserve the extremities
  and the geodesic centers of the initial binary image 'imIn'.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  copy(imIn, imOut)
  dse3 = doubleStructuringElement(structuringElement([2, 3, 4], HEXAGONAL),
  structuringElement([0, 1, 5, 6], HEXAGONAL))
  dse0 = doubleStructuringElement(structuringElement([1, 2, 3, 4, 5, 6], HEXAGONAL),
  structuringElement([0], HEXAGONAL))
  v1 = computeVolume(imOut)
  v2 = 0
  while v1 <> v2:
    v2 = v1
    for i in range(3):
      hitOrMiss(imOut, imWrk1, dse3)
      hitOrMiss(imOut, imWrk2, dse3.rotate(3))
      logic(imWrk1, imWrk2, imWrk1, "sup")
      hitOrMiss(imWrk1, imWrk2, dse0)
      diff(imOut, imWrk2, imOut)
      dse3 = dse3.rotate(1)
    v1 = computeVolume(imOut)

def geodesicCenter(imIn, imOut):
  """
  Computes the geodesic centers of the connected components of 'imIn' by using
  successive thinnings by D3, then D2 and D1 until idempotence.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn)
  copy(imIn, imOut)
  dse1 = doubleStructuringElement(structuringElement([1, 2, 3, 4, 5], HEXAGONAL),
  structuringElement([0, 6], HEXAGONAL))
  dse2 = doubleStructuringElement(structuringElement([2, 3, 4, 5], HEXAGONAL),
  structuringElement([0, 1, 6], HEXAGONAL))
  v1 = computeVolume(imOut)
  v2 = 0
```

```
    while v1 <> v2:
        v2 = v1
        thinByD3(imOut, imOut)
        infThin(imOut, imWrk1, dse2)
        infThin(imOut, imWrk2, dse1)
        logic(imWrk1, imWrk2, imWrk1, "inf")
        copy(imWrk1, imWrk2)
        build(imOut, imWrk2)
        diff(imOut, imWrk2, imWrk2)
        logic(imWrk1, imWrk2, imOut, "sup")
        v1 = computeVolume(imOut)

def thinThickGradient(imIn, imOut, dir):
    """
    Computes the modulus of the directional gradient in direction 'dir' of 'imIn' and
    stores the result in 'imOut'. A greyscale thickening and a greyscale thinning are
    used.
    """

    imWrk = imageMb(imIn)
    se = structuringElement([dir], HEXAGONAL)
    dse = doubleStructuringElement(se.transpose(), se)
    greyThick(imIn, imWrk, dse)
    greyThin(imIn, imOut, dse)
    sub(imWrk, imOut, imOut)

def roughVectorGrad(imIn, imOut1, imOut2):
    """
    Initial vectorial gradient of the 'imIn' image. 'imOut1' contains the modulus
    with some erroneous values (non zero) where the azimuth should be egal to zero.
    'imOut2' contains a first computation of the azimuth: all the directions where
    the modulus is maximal are stored in the various bit planes of 'imOut2'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imMask = imageMb(imIn, 1)
    imOut1.reset()
    imOut2.reset()
    for i in range(6):
        dir = i + 1
        thinThickGradient(imIn, imWrk1, dir)
        generateSupMask(imWrk1, imOut1, imMask, True)
        convertByMask(imMask, imWrk2, 255, 0)
        logic(imOut2, imWrk2, imOut2, "inf")
        generateSupMask(imWrk1, imOut1, imMask, False)
        logic(imWrk1, imOut1, imOut1, "sup")
        copyBitPlane(imMask, i, imOut2)

def defineGradLut():
    """
    Generation of the look-up table correcting the initial coding of the azimuths
    provided by the roughVectorGrad function.
    """

    gradLut = [0 for i in range(256)]
    for i in range(6):
        j = (2 ** i)
        gradLut[j] = (2 * i) + 1
```

```
        j = j + (2 ** (i + 1)) % 63
        gradLut[j] = 2 * (i + 1)
        j = j + (2 ** (i + 2)) % 63
        gradLut[j] = (2 * (i + 1) + 1) % 12
    return gradLut

def vectorialGradient(imIn, imOut1, imOut2):
    """
    Vectorial gradient of the 'imIn' image. 'imOut' contains the gradient modulus and
    'imOut2' contains the gradient azimuths coded by twelve directions.
    """

    imWrk = imageMb(imIn)
    imMask = imageMb(imIn, 1)
    gradLut = defineGradLut()
    roughVectorGrad(imIn, imOut1, imWrk)
    lookup(imWrk, imOut2, gradLut)
    threshold(imOut2, imMask, 0, 0)
    convertByMask(imMask, imWrk, 255, 0)
    logic(imOut1, imWrk, imOut1, "inf" )

def holesSieving(imIn, imOut, n):
    """
    Puts in imOut all the connected components of imIn which contains exactly n holes.
    """

    imWrk0 = imageMb(imIn, 32)
    imWrk1 = imageMb(imIn)
    imOut.reset()
    # Initial image labelling
    nParticles = label(imIn, imWrk0)
    for i in range(1, nParticles+1):
        # each particle is extracted and its connectivity number is measured
        threshold(imWrk0, imWrk1, i, i)
        nc = computeConnectivityNumber(imWrk1)
        # if the number of holes is equal to n, the particle is added to the output image
        if (n==(1 - nc)):
            logic(imWrk1, imOut, imOut, "sup")

def homotopyTreeBuild(imIn, imOut):
    """
    Builds the homotopy tree of the initial binary set imIn. The result is stored
    in image imOut. Each grey level corresponds to a hierarchy level of the homotopy
    tree. Extracting each level i of embedding of particles and holes of the initial
    set consists simply in a [i, i] thresholding of imOut.
    """

    imWrk0 = imageMb(imIn)
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn, 8)
    copy(imIn, imWrk0)
    v = computeVolume(imIn)
    imOut.reset()
    i = 0
    # Loop performed until no connected component remains.
    while v!=0:
        i = i+1
        # Background extraction.
        closeHoles(imWrk0, imWrk1)
```

```
    negate(imWrk1, imWrk1)
    # Connected components adjacent to the background are rebuilt.
    dilate(imWrk1, imWrk1)
    build(imWrk0, imWrk1)
    # These particles are at level i in the homotopy tree.
    # They are removed from the current set, giving access to the next level (in imWrk0).
    diff(imWrk0, imWrk1, imWrk0)
    v = computeVolume(imWrk0)
    # The level-i particles are labelled with i and added to the label image.
    convertByMask(imWrk1, imWrk2, 0, i)
    add(imOut, imWrk2, imOut)

def closeOneHole(imIn, imOut):
  """
  This procedure allows to close one and only one hole in each connected component
  of the binary image imIn. When a connected component has no hole, it remains
  unchanged in the final image stored in imOut.
  This algorithm requires a single level of homotopy in the initial image.
  """

  imWrk0 = imageMb(imIn, 32)
  imWrk1 = imageMb(imIn, 32)
  imWrk2 = imageMb(imIn, 32)
  imWrk3 = imageMb(imIn)
  imWrk4 = imageMb(imIn)
  # The holes are filled in imWrk3 and extracted in imWrk4.
  closeHoles(imIn, imWrk3)
  diff(imWrk3, imIn, imWrk4)
  # The holes are labelled.
  nb = label(imWrk4, imWrk0)
  # The image with filled holes is converted in 32-bit.
  convertByMask(imWrk3, imWrk1, 0, computeMaxRange(imWrk1)[1])
  # Geodesic reconstruction of the label image.
  copy(imWrk0, imWrk2)
  build(imWrk1, imWrk2)
  # The holes with same label as the built image are extracted...
  generateSupMask(imWrk0, imWrk2, imWrk4, False)
  logic(imWrk4, imWrk3, imWrk3, "inf")
  # ... and filled.
  logic(imIn, imWrk3, imOut, "sup")

def holesLabelling(imIn, imOut):
  """
  Labels each particle in imIn with a value equal to its number of holes +1.
  The result is put in 32-bit image imOut.
  """

  # Working images.
  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn, 32)
  # Structuring elements
  dse1 = doubleStructuringElement([1,6],[0],HEXAGONAL)
  dse2 =  doubleStructuringElement([1],[0,2],HEXAGONAL)
  # Initializing the label image with 2.
  convertByMask(imIn, imOut, 0, 2)
  # Determining the 2nd configurations in the connectivity number calculation.
  # and adding their number to the label image.
  hitOrMiss(imIn, imWrk1, dse2)
  measureLabelling(imIn, imWrk1, imWrk2)
```

```
    add(imOut, imWrk2, imOut)
    # Determining the 1st configurations and subtracting them to get the
    # number of holes + 1.
    hitOrMiss(imIn, imWrk1, dse1)
    measureLabelling(imIn, imWrk1, imWrk2)
    sub(imOut, imWrk2, imOut)

def extremities(imIn, imOut1, imOut2, innerParticles=False):
    """
    This operation performs the computation of the extremities (maxima of a
    geodesic distance function) and puts the result in imOut.
    'imIn' must be a binary image, 'imOut1' is a 32-bit image containing the
    extremities and their geodesic distance to the centroid (allowing thus to
    sort them according to their distance to the center) and imOut2 a 32-bit
    image containing the entire geodesic distance from the geodesic centers.
    If 'innerParticles' is set to False(default), the particles touching the
    edge are considered as extending outside the image window. Therefore, no
    extremity is detected on the edge of the image. If 'innerParticles' is True,
    all the particles are supposed to be included in the image, so extremities
    may appear on the edge.
    """

    imWrk1 = imageMb(imIn)
    # Computation of the geodesic centers.
    geodesicCenter(imIn, imWrk1)
    # These centers are removed from the initial image (each particle contains
    # a hole).
    diff(imIn, imWrk1, imWrk1)
    # Computing the geodesic distance
    geodesicDistance(imWrk1, imIn, imOut2)
    # The extremities correspond to the maxima.
    maxima(imOut2, imWrk1)
    # Extremities on the edge are removed if 'innerParticles' is set to False.
    if not(innerParticles):
        removeEdgeParticles(imWrk1, imWrk1)
    # The extremities are given their corresponding distance to the center.
    convert(imWrk1, imOut1)
    logic(imOut2, imOut1, imOut1, "inf")

def extremePoints(imIn, imOut, margin=0):
    """
    This operator is a refinement of the 'extremities' operator. It determines
    points of the connected components of the binary set 'imIn' which are the
    farthest extremity points. The 32-bit image 'imOut' contains these extreme
    points valued with their distance to the geodesic center. 'margin' is a
    parameter which allows to take into account an extremity even if its distance
    to the center is not maximal, provided that it is not lower than this
    maximal value minus 'margin'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn, 32)
    imWrk3 = imageMb(imIn, 32)
    imWrk4 = imageMb(imIn, 32)
    geodesicCenter(imIn, imWrk1)
    diff(imIn, imWrk1, imWrk1)
    geodesicDistance(imWrk1, imIn, imWrk2)
    maxima(imWrk2, imWrk1)
    convert(imWrk1, imWrk3)
    logic(imWrk2, imWrk3, imWrk3, "inf")
```

```
    convert(imIn, imWrk4)
    copy(imWrk3, imOut)
    hierarBuild(imWrk4, imOut)
    floorSubConst(imOut, margin, imOut)
    generateSupMask(imWrk3, imOut, imWrk1, False)
    logic(imWrk1, imIn, imWrk1, "inf")
    removeEdgeParticles(imWrk1, imWrk1)
    convert(imWrk1,imWrk3)
    logic(imWrk2, imWrk3, imOut, "inf")

def geodesicAdaptiveDilate(imIn, imMask, imOut):
    """
    This operator performs a binary adaptive dilation. The 32-bit image
    'imIn' indicates for each pixel the size of the geodesic dilation (by the
    default structuring element) which will be applied on it. The geodesic
    mask is defined by the binary image 'imMask'. The result of the dilation is
    put in the binary image 'imOut'. It is called adaptive because its size is
    given locally for each pixel by the value of this pixel in 'imIn'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)
    convert(imMask, imWrk1)
    copy(imIn, imWrk2)
    v1 = 0
    v2 = computeVolume(imWrk2)
    # At each step, the dilated image is decreased. So each pixel value
    # indicates how many steps of dilation remain. When the image volume
    # does not change, the process is finished.
    while v2 > v1:
        v1 = v2
        geodesicDilate(imWrk2, imWrk1, imWrk3)
        floorSubConst(imWrk3, 1, imWrk3)
        logic(imWrk2, imWrk3, imWrk2, "sup")
        v2 = computeVolume(imWrk2)
    threshold(imWrk2, imOut, 1, computeMaxRange(imIn)[1])

def segmentParticles(imIn, imOut):
    """
    Segmentation of the touching particles contained in the binary image
    'imIn'. This segmentation performs the watershed of the inverted
    distance function of the image.
    The result is stored in the binary image 'imOut'.
    """

    imWrk1 = imageMb(imIn, 32)
    imWrk2 = imageMb(imIn, 32)
    imWrk3 = imageMb(imIn)
    computeDistance(imIn, imWrk1, edge=FILLED)
    maxima(imWrk1, imWrk3)
    maxVal = computeRange(imWrk1)[1]
    imWrk2.fill(maxVal)
    sub(imWrk2, imWrk1, imWrk1)
    nParticles = label(imWrk3, imWrk2)
    watershedSegment(imWrk1, imWrk2)
    copyBitPlane(imWrk2, 31, imWrk3)
    diff(imIn, imWrk3, imOut)
```

```
def roadMarkersExtract(im1, im2):
  """
  Extraction of the road and outside markers from the segmentation obtained
  by the higher level of hierarchy of the enhanced waterfalls transform.
  'im1' contains the initial catchment basins. 'im2' is a 32-bit image containing the
  labelled markers for the road and for the outside.
  """

  imWrk1 = imageMb(im1)
  imWrk2 = imageMb(im1)
  imWrk3 = imageMb(im1)
  imWrk1.reset()
  imSize = im1.getSize()
  xPos = imSize[0]//2
  yPos = imSize[1]-15
  imWrk1.setPixel(1, (xPos, yPos))
  dilate(imWrk1, imWrk1, 15)
  copy(imWrk1, imWrk2)
  build(im1, imWrk2)
  closeHoles(imWrk2, imWrk2)
  erode(imWrk2, imWrk3, 10)
  build(imWrk3, imWrk1)
  diff(imWrk3, imWrk1, imWrk3)
  negate(imWrk2, imWrk2)
  erode(imWrk2, imWrk2, 10)
  logic(imWrk2, imWrk3, imWrk2, "sup")
  convertByMask(imWrk1, im2, 0, 2)
  add(im2, imWrk2, im2)

def roadSegment(im1, im2):
  """
  Segmentation of the road in image 'im1'. The result is put in
  the binary image 'im2'.
  The algortihm uses the above road markers extractor.
  """

  imWrk1 = imageMb(im1)
  imWrk2 = imageMb(im1)
  imWrk3 = imageMb(im1, 1)
  imWrk4 = imageMb(im1, 32)
  gradient(im1, imWrk1, 2)
  valuedWatershed(imWrk1, imWrk2)
  nbHier = enhancedWaterfalls(imWrk2, imWrk1)
  threshold(imWrk1, imWrk3, 0, nbHier -1)
  roadMarkersExtract(imWrk3, imWrk4)
  watershedSegment(imWrk1, imWrk4)
  copyBitPlane(imWrk4, 31, im2)

def extractGridMarkers(imIn, imOut):
  """
  This procedure extracts markers of the cells of the grid printed on the
  steel sheet (image 'imIn') and puts the result in binary image 'imOut'.
  The procedure uses the extraction of the maxima of successive strong
  levellings (until size 14). At each step after the first one, the connected
  components of the maxima which touch the edge are removed.
  """

  imWrk1 = imageMb(imIn)
  imWrk2 = imageMb(imIn, 1)
```

```
# A first filtering with a strong levelling produces first seeds for the
# markers (maxima of the filtered image).
i = 2
strongLevelling(imIn, imWrk1, i, eroFirst=False)
maxima(imWrk1, imOut)
v1 = computeVolume(imOut)
# The same filtering is iterated for increasing sizes. Only the inner
# maxima are preserved and added to the markers image. When no inner
# marker is available, the process ends.
while v1 <> 0:
    i += 2
    strongLevelling(imIn, imWrk1, i, eroFirst=False)
    maxima(imWrk1, imWrk2)
    removeEdgeParticles(imWrk2, imWrk2)
    logic(imWrk2, imOut, imOut, "sup")
    v1 = computeVolume(imWrk2)
```