Mamba Image Examples

Nicolas BEUCHER
Serge BEUCHER

www.mamba-image.org

Except where otherwise **noted**, the Mamba Documentation Project is covered by the **Creative Commons Attribution 3.0 License** (see http://creativecommons.org/licenses/by/3.0/)

October 30, 2015

Contents

Contents **2**

1 Regarding examples **3**

2 Easy **4**
2.1 Closing holes in a grey scale image . . . 4
2.2 Conversion between PIL image format and Mamba image format . . . 5
2.3 Cropping and copying images . . . 6
2.4 Display in mamba3D . . . 7
2.5 Drawing in Mamba . . . 8
2.6 mambaDisplay.extra Dynamic thresholder usage . . . 9
2.7 Image properties . . . 10
2.8 mambaDisplay.extra interactive segment usage . . . 11
2.9 Messing about with display, part 1 : activation . . . 12
2.10 Messing about with display, part 2 : palette and shortcut . . . 14
2.11 Opening and creating images . . . 16
2.12 Pixels manipulation . . . 17
2.13 Squeezing an image . . . 18
2.14 mambaDisplay.extra superposer usage . . . 19

3 Moderate **20**
3.1 Arrow encoding of an image . . . 20
3.2 Color channels manipulations . . . 22
3.3 Contrast enhancement using top hat operators . . . 23
3.4 Counting and marking teeth of a notched wheel . . . 24
3.5 Creating Steiner polyhedrons in 3D . . . 25
3.6 Directional morphological operators in 12 or 16 directions . . . 27
3.7 Directional moving edge measurement . . . 30
3.8 Erosion and structuring elements . . . 32
3.9 Extraction of the first particle . . . 33
3.10 Extremities of particles . . . 34
3.11 Modulus and azimut of a directional gradient . . . 36
3.12 Morphological laplacian and zero-crossings detection . . . 39
3.13 Noise reduction using alternate filters . . . 41
3.14 Pellets separation in a 3D polyurethane foam . . . 43
3.15 Pixel neighbors counter . . . 45
3.16 Segmentation algorithms : examples and demo . . . 46
3.17 Simple and coarse extraction of micro-aneurisms in retina . . . 48
3.18 Sorting particles with different numbers of holes . . . 50
3.19 Valued minima of a greyscale image . . . 52
3.20 Watershed segmentation using gradient minima as markers . . . 53

4 Advanced **54**
4.1 Analysis of antibiograms . . . 54
4.2 Analysis of the distribution of boron fibers . . . 57
4.3 Area labelling and area opening . . . 60
4.4 Automated measurement in medical field using image analysis . . . 62
4.5 Automatic thresholding using gradient . . . 65
4.6 Blobs segmentation in an electrophoresis gel . . . 67
4.7 Color image segmentation with quasi-distance of the gradient . . . 69
4.8 Criterion opening, Feret diameter (bounding boxes) example . . . 71
4.9 Dislocations in an eutectic lamellar alloy . . . 73
4.10 Embedded text filtering . . . 77
4.11 Extraction of a stamped grid on a steel sheet . . . 79
4.12 Face recognition : connecting Mamba and OpenCV . . . 81
4.13 Fibers orientation coding . . . 83
4.14 Gaussian blur : connecting Mamba to numpy/scipy . . . 87
4.15 Particles (coffee grains) separation and counting . . . 89

4.16 Quality control of a clutch device 92
4.17 Regularised gradient and regularised minima 95
4.18 Segmentation of a heap of rocks for evaluating its size distribution 97
4.19 Topographic rendering of an image 99
4.20 Traffic lanes segmentation 101
4.21 Traffic measurement, vehicles detection and counting 104
4.22 Volume controlled flooding 107

1 Regarding examples

This document presents some examples that you can retrieve in the examples directory of the source files.

There are three types of examples regrouped in the following sections:

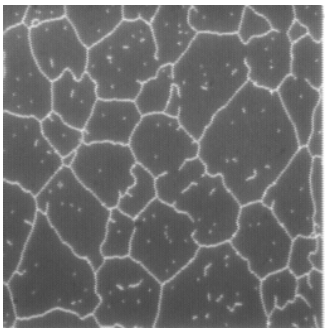
1. **easy**: These are mainly intended for beginners so they could refer to some basic examples to start their own code.
2. **moderate**: In these examples, more elaborated actions are performed and presented. They cover usage that may not be difficult to do provided that you read the documentation, but are more easily explained and understood with an example.
3. **advanced**: These are more complex examples that should be considered as demonstrators of the capabilities of the Mamba library.

Examples presented here may not correctly work. We tried to put a wide range of examples and produced this small selection after some efforts (some of them are non-trivial) but we may have broke them in the process of publishing them. If it so happens, try to see this as an exercise or e-mail us.

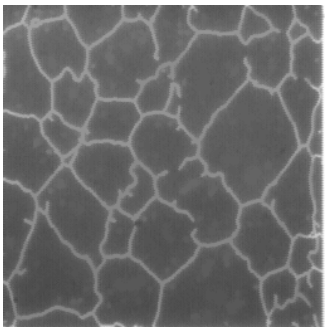
2 Easy

2.1 Closing holes in a grey scale image

This simple example shows that the holes closing procedure can be applied on a grey scale image without any problem. A hole is simply a region in the image surrounded by brighter pixels. In this example, the operator is used to clean an alumina grains image by removing the white inclusions inside the grains.



alumina.png



filtered_alumina.png

Here is the source code for this example (can be found in Closing holes in a greyscale image):

```
# Importing the mamba and mambComposed modules
from mamba import *

# Opening the initial image.
im = imageMb('alumina.png')

# Inverting the image (so that inclusions really correspond to holes).
negate(im, im)

# Applying the holes closing operator (note that the result can be stored in
# same image.
closeHoles(im, im)

# Inverting the image again.
negate(im, im)

# And saving the result.
im.save("filtered_alumina.png")
```

2.2 Conversion between PIL image format and Mamba image format

This example shows how to convert your Mamba image into a PIL image enabling you to use PIL functionnalities and how to convert the result back into a Mamba image.

Image source is http://commons.wikimedia.org/wiki/File:Ebenthal_Schlosswirt_20052010_06.jpg.

colorful.jpg

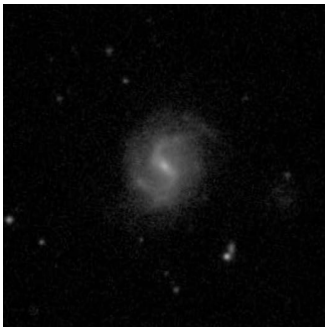
colorful_transposed.png

Here is the source code for this example (can be found in Conversion between PIL and Mamba):

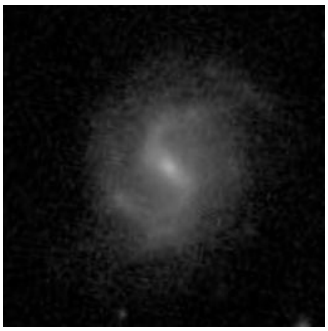
```
# Importing mamba
from mamba import *

# Opening an image and storing it in Mamba format
im = imageMb('colorful.jpg')

# Converting the image in PIL format
pilim = Mamba2PIL(im)

# Doing a bit of computation using the PIL library
pilim = pilim.transpose(Image.FLIP_LEFT_RIGHT)

# Then converting back into mamba format
PIL2Mamba(pilim, im)

# And saving it, notice that color disappeared
im.save("colorful_transposed.png")
```

2.3 Cropping and copying images

The cropCopy operator is very useful to modify the size of an image, to select regions of interest or to compose a new image by assembling elements coming from other images. In this example, this operator is simply used to reduce the size of the image as most of it is black and without interest. The image used in this example comes from the Galaxy Zoo web site. This site allows you to contribute to the classification of galaxies. Visit it at: http://www.galaxyzoo.org.



galaxy.png



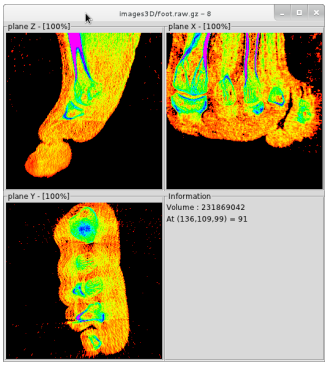
cropped_galaxy.png

Here is the source code for this example (can be found in Cropping and copying images):

```
# Importing the mamba and mambComposed modules
from mamba import *

# Opening the initial image.
imA = imageMb('galaxy.png')

# The galaxy visible in the image can be easily contained in a 256x256 image.
# The center of the galaxy is located at image coordinates (253, 256).
# Therefore, we calculate the initial position (upper left corner) of the
# cropping window (x = 253 - 128 and y = 256 - 128).
posin = (125, 128)

# Then we define a new image of size 256x256.
imB = imageMb(256, 256)

# We crop and copy the initial image into the new one.
cropCopy(imA, posin, imB, (0, 0), (256, 256))

# Then, we save the new image.
imB.save("cropped_galaxy.png")
```

2.4 Display in mamba3D

This example shows you how to select and activate the various displays available in mamba 3D package. The foot image data used can be downloaded at http://tc18.liris.cnrs.fr/code_data_set/3D_images.html. The file foot.raw.gz must be decoompressed and renamed foot.raw before use.



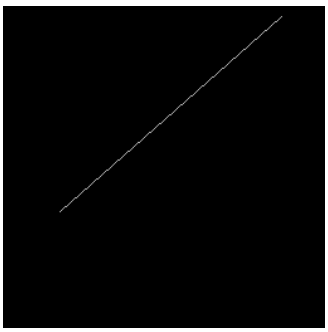
display_proj.png

Here is the source code for this example (can be found in Display in mamba3D):

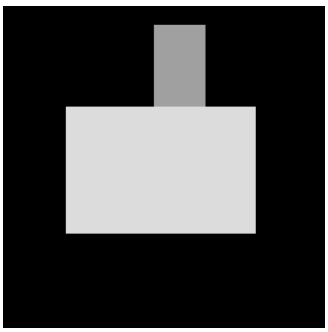
```
# Importing the mamba module
from mamba3D import *

# Creating an image and loading the data
im3D = image3DMb()
im3D.loadRaw("foot.raw")

# Activating the display. This will make a 3D window appear
im3D.show(mode="VOLUME")

# This line will activate a volume rendering display if you have VTK
# installed on your computer. You can rotate (holding your mouse left
# button while moving it), zoom in and out (same movement while holding
# the right button) and displace the image (this time by holding both the
# left button and the shift key on your keyboard while moving the mouse).
# This allows you to view more precisely any area of your image. You can
# restore the initial settings by pressing r (Although this does not seem
# to affect rotation).

# If you want to activate a more precise display or if you have not
# VTK installed (remind that VTK is not currently available with Python 3),
# you may want to use the plane projection display:
im3D.show() # (default display)

# You can navigate through planes moving your mouse while holding <ctrl> down.
# The mouse motion will allow you to change the plane following it (the
# displayer enables a red cross to indicate where you are in the image data).

# You can also alternatively display a 2D image extracted from the 3D image
im3D[128].show()
```

2.5 Drawing in Mamba

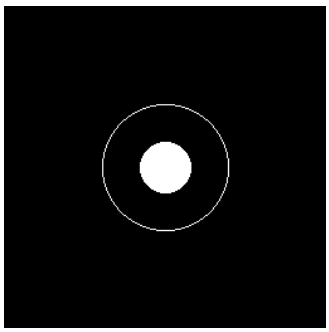
These examples show some of the drawing functions available in Mamba.



draw_line.png



draw_square.png



draw_circle.png

Here is the source code for this example (can be found in Drawing in Mamba):

```
# Importing the mamba module
from mamba import *

# Creating a default image, 256x256 8-bit image
im = imageMb()

# Drawing a line. The line is drawn according to the given
# 4 value tuple interpreted as (x_start, y_start, x_stop, y_stop).
# So, in our case, the line goes from point (45,163) to point (221,8)
drawLine(im, (45,163,221,8), 180)
im.save("draw_line.png")


im.reset()
# Drawing a square (actually a rectangle) filled with value 160.
# The middle tuple indicates from where to where the square extends
drawSquare(im, (120,15,160,95), 160)
# Drawing another square filled with value 220. Notice that this
# square overlaps the first one. The last drawn is always on top
# of the others
drawSquare(im, (50,80,200,180), 220)
im.save("draw_square.png")

im.reset()
# Drawing an empty circle centered in the middle of the image with a
# radius of 50 and white (255).
drawCircle(im, (128,128,50), 255)
# Then drawing a filled circle.
drawFillCircle(im, (128,128,20), 255)
im.save("draw_circle.png")
```

2.6 mambaDisplay.extra Dynamic thresholder usage

This is a small demonstration of the mambaDisplay.extra dynamic thresholder GUI. It allows you to search manually for the best threshold value.

snake.png

dynamicThreshold_snap.png

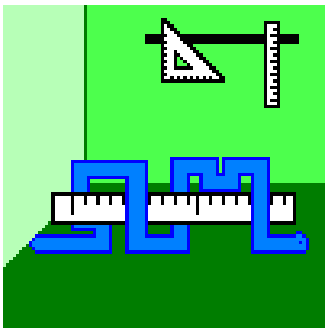
Here is the source code for this example (can be found in Dynamic thresholder usage):

```
# Importing mamba and mambaDisplay.extra
from mamba import *
import mambaDisplay.extra

# Opening an image
im = imageMb('snake.png')

# Calling the dynamic threshold GUI
# This will block script progression until user hit the close button
# You can change the threshold values by using keys q, s, w and x. Of course
# the display can be zoom in or out and has most of the functionalities
# available in the standard mamba display.
tvals = mambaDisplay.extra.dynamicThreshold(im)

# The value obtained can be then used to compute a binary
# image with the threshold value (upper and lower threshold)
thresh = imageMb(im, 1)
threshold(im, thresh, *tvals)
```

2.7 Image properties

This example shows you how to get image properties like size and depth. It also describes how the image constructor can be used to create images sharing the same properties.

snake.png



measure.png

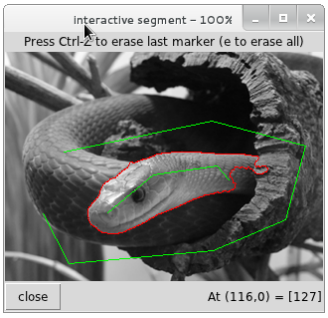
Here is the source code for this example (can be found in Image properties):

```
# Importing the mamba module
from mamba import *

# Loading an image
im = imageMb('snake.png')

# This image is 320x230 in 8-bit greyscale
print(im.getSize())
print(im.getDepth())

# Creating an image with the same properties
im1 = imageMb(im)
print(im1.getSize())
print(im1.getDepth())

# Creating an image with the same properties except for depth
im2 = imageMb(im, 1)
print(im2.getSize())
print(im2.getDepth()) # <- here the return is 1
```

2.8 mambaDisplay.extra interactive segment usage

This is a small demonstration of the mambaDisplay.extra interactive segmentation GUI. It allows you to segment an image using the watershed algorithm (processing the gradient image) with user-defined markers. Its behavior is very similar to the magic selection you can find in image processing programs such as GIMP.

snake.png



interseg_snap.png

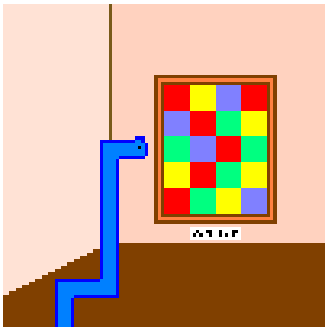
Here is the source code for this example (can be found in Interactive segment usage):

```
# Importing mamba and mambaDisplay.extra
from mamba import *
import mambaDisplay.extra

# Opening an image
im = imageMb('snake.png')
# The 32-bit image in which the segmentation result will be placed
imSeg = imageMb(im, 32)

# Calling the interactive segmentation GUI
#
# This will block script progression until you hit the close button.
# To create a marker simply click on the image. If you do that
# while pressing the control button on your keyboard, the marker
# will be a line between your previous click on the new one (creating a
# unique marker). The function return the list of markers.
#
# Of course the display can be zoom in or out and has most of the
# functionalities available in the standard mamba display.
print(mambaDisplay.extra.interactiveSegment(im, imSeg))
```

2.9 Messing about with display, part 1 : activation

This example shows you how to activate the display and presents some of the options related to the display. This example is meant to be replayed inside a Python console. If you run directly this script, nothing will happen.

snake.png



display.png

Here is the source code for this example (can be found in Messing about with display part1):

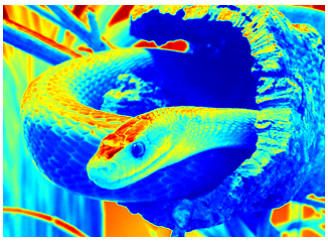
```
# Importing the mamba module
from mamba import *
import mambaDisplay

# Loading an image
im = imageMb('snake.png')

# Activating the display. This will make a window appear
im.show()

# You can deactivate the display at all time by either
# - Closing the window
# - Calling the appropriate method like this
im.hide()

# Reactivate it
im.show()

# You can update the display (if your image changed) by calling the following
# method
im.update()
# This action is automatically performed by every Mamba functions so it is
# likely you will not have to use it on your own

# The two last methods related to display allow you to freeze/unfreeze the
# display to prevent mamba updating it but keeping the window visible.
im.freeze() # <- the display is no longer updated
im.unfreeze() # <- the display will be updated (and returns to what the
# image looks like if there were modifications)

# Sometimes, you will have lots of images opened and displayed making it quite
# messy on your desktop and thus very difficult to read and analyze. To make
```

```
# sure displays are properly organised you can call the following function
mambaDisplay.tidyDisplays()
# The function does not do miracles (like increasing your screen resolution) so
# don't have too much expectations
```

2.10 Messing about with display, part 2 : palette and shortcut

This example shows you some options you can use to increase your display skills.

snake.png



snake_palette.png

Here is the source code for this example (can be found in Messing about with display part2):

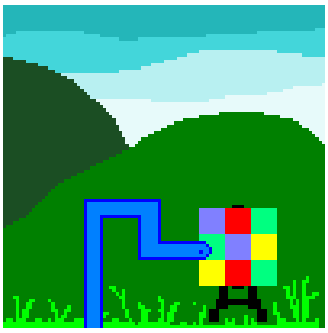
```
# Importing the mamba module
from mamba import *
import mambaDisplay

# Let's start with an image and an activated display
im = imageMb('snake.png')
im.show()

# You can create your own palette
your_palette = (
0, 0, 131, 0, 0, 135, 0, 0, 139, 0, 0, 143, 0, 0, 147, 0, 0, 151, 0, 0, 155, 0, 0,
159, 0, 0, 163, 0, 0, 167, 0, 0, 171, 0, 0, 175, 0, 0, 179, 0, 0, 183, 0, 0, 187,
0, 0, 191, 0, 0, 195, 0, 0, 199, 0, 0, 203, 0, 0, 207, 0, 0, 211, 0, 0, 215, 0, 0,
219, 0, 0, 223, 0, 0, 227, 0, 0, 231, 0, 0, 235, 0, 0, 239, 0, 0, 243, 0, 0, 247,
0, 0, 251, 0, 0, 255, 0, 3, 255, 0, 7, 255, 0, 11, 255, 0, 15, 255, 0, 19, 255, 0,
23, 255, 0, 27, 255, 0, 31, 255, 0, 35, 255, 0, 39, 255, 0, 43, 255, 0, 47, 255, 0,
51, 255, 0, 55, 255, 0, 59, 255, 0, 63, 255, 0, 67, 255, 0, 71, 255, 0, 75, 255, 0,
79, 255, 0, 83, 255, 0, 87, 255, 0, 91, 255, 0, 95, 255, 0, 99, 255, 0, 103, 255,
0, 107, 255, 0, 111, 255, 0, 115, 255, 0, 119, 255, 0, 123, 255, 0, 127, 255, 0,
131, 255, 0, 135, 255, 0, 139, 255, 0, 143, 255, 0, 147, 255, 0, 151, 255, 0, 155,
255, 0, 159, 255, 0, 163, 255, 0, 167, 255, 0, 171, 255, 0, 175, 255, 0, 179, 255,
0, 183, 255, 0, 187, 255, 0, 191, 255, 0, 195, 255, 0, 199, 255, 0, 203, 255, 0,
207, 255, 0, 211, 255, 0, 215, 255, 0, 219, 255, 0, 223, 255, 0, 227, 255, 0, 231,
255, 0, 235, 255, 0, 239, 255, 0, 243, 255, 0, 247, 255, 0, 251, 255, 0, 255, 255,
3, 255, 255, 7, 255, 251, 11, 255, 247, 15, 255, 243, 19, 255, 239, 23, 255, 235,
27, 255, 231, 31, 255, 227, 35, 255, 223, 39, 255, 219, 43, 255, 215, 47, 255, 211,
51, 255, 207, 55, 255, 203, 59, 255, 199, 63, 255, 195, 67, 255, 191, 71, 255, 187,
75, 255, 183, 79, 255, 179, 83, 255, 175, 87, 255, 171, 91, 255, 167, 95, 255, 163,
99, 255, 159, 103, 255, 155, 107, 255, 151, 111, 255, 147, 115, 255, 143, 119, 255,
139, 123, 255, 135, 127, 255, 131, 131, 255, 127, 135, 255, 123, 139, 255, 119, 143,
255, 115, 147, 255, 111, 151, 255, 107, 155, 255, 103, 159, 255, 99, 163, 255, 95,
167, 255, 91, 171, 255, 87, 175, 255, 83, 179, 255, 79, 183, 255, 75, 187, 255, 71,
191, 255, 67, 195, 255, 63, 199, 255, 59, 203, 255, 55, 207, 255, 51, 211, 255, 47,
215, 255, 43, 219, 255, 39, 223, 255, 35, 227, 255, 31, 231, 255, 27, 235, 255, 23,
239, 255, 19, 243, 255, 15, 247, 255, 11, 251, 255, 7, 255, 255, 3, 255, 255, 0, 255,
251, 0, 255, 247, 0, 255, 243, 0, 255, 239, 0, 255, 235, 0, 255, 231, 0, 255, 227,
0, 255, 223, 0, 255, 219, 0, 255, 215, 0, 255, 211, 0, 255, 207, 0, 255, 203, 0,
```

```
255, 199, 0, 255, 195, 0, 255, 191, 0, 255, 187, 0, 255, 183, 0, 255, 179, 0, 255,
175, 0, 255, 171, 0, 255, 167, 0, 255, 163, 0, 255, 159, 0, 255, 155, 0, 255, 151,
0, 255, 147, 0, 255, 143, 0, 255, 139, 0, 255, 135, 0, 255, 131, 0, 255, 127, 0,
255, 123, 0, 255, 119, 0, 255, 115, 0, 255, 111, 0, 255, 107, 0, 255, 103, 0, 255,
99, 0, 255, 95, 0, 255, 91, 0, 255, 87, 0, 255, 83, 0, 255, 79, 0, 255, 75, 0, 255,
71, 0, 255, 67, 0, 255, 63, 0, 255, 59, 0, 255, 55, 0, 255, 51, 0, 255, 47, 0, 255,
43, 0, 255, 39, 0, 255, 35, 0, 255, 31, 0, 255, 27, 0, 255, 23, 0, 255, 19, 0, 255,
15, 0, 255, 11, 0, 255, 7, 0, 255, 3, 0, 255, 0, 0, 251, 0, 0, 247, 0, 0, 243, 0,
0, 239, 0, 0, 235, 0, 0, 231, 0, 0, 227, 0, 0, 223, 0, 0, 219, 0, 0, 215, 0, 0,
211, 0, 0, 207, 0, 0, 203, 0, 0, 199, 0, 0, 195, 0, 0, 191, 0, 0, 187, 0, 0, 183,
0, 0, 179, 0, 0, 175, 0, 0, 171, 0, 0, 167, 0, 0, 163, 0, 0, 159, 0, 0, 155, 0,
0, 151, 0, 0, 147, 0, 0, 143, 0, 0, 139, 0, 0, 135, 0, 0, 131, 0, 0
)

# And then add it to the palettes available for display
mambaDisplay.addPalette("my palette", your_palette)

# If you save at this point, you can apply the palette through the appropriate
# argument
im.save('snake_palette.png', palette=your_palette)

# Now you can play with the display shortcuts
# Try pressing P, it rotates through all the palettes including yours
# Try zooming in or out with A and Z or with your mouse wheel
# Try Ctrl-R to reset the display
# Try pressing the right button of your mouse to spawn the contextual menu
```

2.11 Opening and creating images

This example shows various ways to open or create image using the constructor of the imageMb class. This example is meant to be replayed inside a Python console. If you run this script directly, nothing will happen.

snake.png



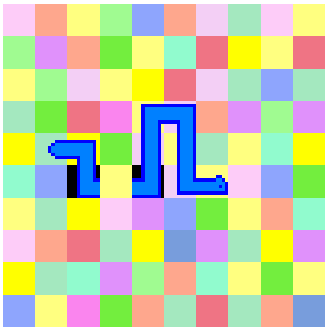
show.png

Here is the source code for this example (can be found in Open and creating images):

```
# Importing the mamba module
from mamba import *

# Opening an existing image (will be greyscale, 8-bit)
im1 = imageMb('snake.png')
# Creating a 8-bit image with the same size
im2 = imageMb(im1, 8)
# Then a binary image (still same size)
im3 = imageMb(im1, 1)
# And a 32-bit image (same size but do I need to be more specific ?)
im4 = imageMb(im1, 32)

# Opening an existing image but as binary image
im5 = imageMb('snake.png', 1)
# or as a 32-bit image
im6 = imageMb('snake.png', 32)
# For the last two examples, this might not be pertinent because of the
# properties of the example image.

# Creating an image with a specified size
im7 = imageMb(200, 340)
# In this example, the created image will have a corrected size of 256x340.

# Creating an image without option
im8 = imageMb()
# im8 is now a greyscale image (8-bit) of size 256x256. These are the default
# values when they are not overriden in the constructor.

# Images are automatically garbage collected by Python once there is no more
# reference pointing to it.
```

2.12 Pixels manipulation

In this example, we use the pixels manipulation methods.



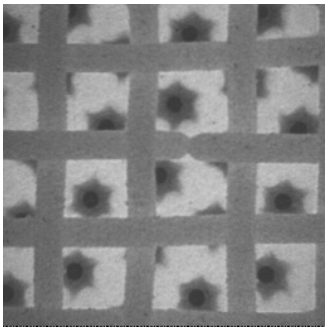
pixels.png

Here is the source code for this example (can be found in Pixels manipulation):

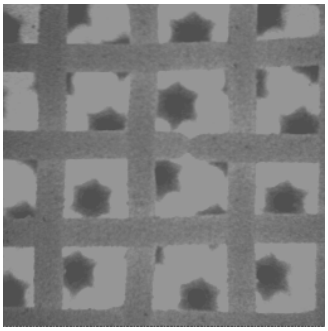
```
# Importing the mamba module
from mamba import *

# Creating a default image, 256x256 8-bit image
im = imageMb()

# Setting the pixel at (128,128) to value 50
im.setPixel(50, (128,128))
# Getting the pixel value at the same position
print(im.getPixel((128,128)))

# When trying to access a pixel outside image boundary an exception is raised
try:
    im.getPixel((256,256)) # position start at 0 to width-1 or height-1
except MambaError:
    print("error catch")

# The two methods to set a pixel vary on one aspect: the display update
im.show()
im.setPixel(255, (10,10)) #<- Pixel is set and visible in display
im.fastSetPixel(255, (20,20)) #<- Pixel is set but not visible
im.update() # Now it is
```

2.13 Squeezing an image

This very simple example shows how to use the conversion by mask to force the pixel values of an image to be in a given range. This can be useful when incrusting a mixed information in an image as it allows to let room for this information.



burner.png



squeezed_burner.png

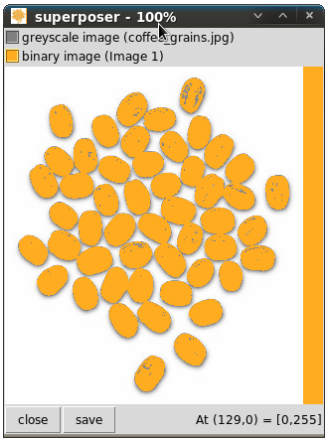
Here is the source code for this example (can be found in Squeezing an image):

```
# Importing the mamba module.
from mamba import *

# Defining the squeeze operator.
def squeeze(imIn, imOut, value1, value2):
    """
    Squeezes the pixel values of 'imIn' between 'value1' and 'value2'. If the
    pixel value is larger than 'value2', it is replaced by 'value2'. If it lower
    than 'value1', it is replaced by 'value1'. In-between values are unchanged.
    'imIn' and 'imOut' must have the same depth.
    """

    imWrk1 = imageMb(imIn, 1)
    imWrk2 = imageMb(imIn)
    # Replacing values lower than value1 by value1.
    threshold(imIn, imWrk1, 0, value1)
    convertByMask(imWrk1, imWrk2, 0, value1)
    logic(imIn, imWrk2, imOut, "sup")
    # Replacing values higher than value2 by value2.
    threshold(imOut, imWrk1, 0, value2)
    convertByMask(imWrk1, imWrk2, value2, computeMaxRange(imOut)[1])
    logic(imOut, imWrk2, imOut, "inf")

# Squeezing an image between 80 and 150.
im1 = imageMb('burner.png')
im2 = imageMb(im1)
squeeze(im1, im2, 80, 150)
# Saving the result.
im2.save('squeezed_burner.png')
```

2.14 mambaDisplay.extra superposer usage

This is a small demonstration of the mambaDisplay.extra superposer GUI. It allows you to display two images at the same time in a superposed manner.

coffee_grains.jpg



superpose_snap.png

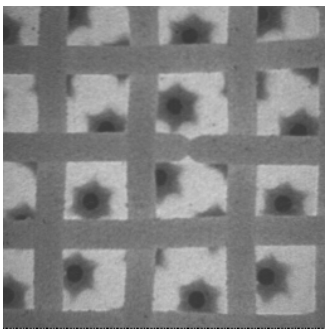
Here is the source code for this example (can be found in Superposer usage):

```
# Importing mamba and mambaDisplay.extra
from mamba import *
import mambaDisplay.extra

# Opening an image
im = imageMb('coffee_grains.jpg')
# To create a superposable image we will generate a binary threshold image
thresh = imageMb(im, 1)
threshold(im, thresh, 0, 128)

# Calling the superposer GUI
# This will block script progression until user hit the close button
# You can change the color in the legend.
# Of course the display can be zoom in or out and has most of the
# functionalities available in the standard mamba display.
mambaDisplay.extra.superpose(im, thresh)
```

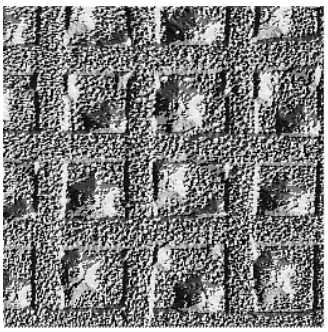
3 Moderate

3.1 Arrow encoding of an image

This example explains how a grey scale image can be arrowed. Arrowing an image consists in drawing arrows from each pixel to their corresponding neighbors if a given condition is fulfilled. Four conditions are possible: the center point value may be higher, higher or equal, lower or lower or equal to the neighbor value. The arrows are encoded by a numerical value equal to the sum of the powers of 2 of the corresponding direction. If, for instance, an arrow exists in directions 1, 4 and 5, the encoded value will be $(2 + 16 + 32) = 50$. This representation of neighborhood relationships is interesting for building various operators simulating propagation phenomena or for designing adaptive transforms (generalised geodesy).



burner.png



arrowedBurner.png

Here is the source code for this example (can be found in Arrow encoding of an image):

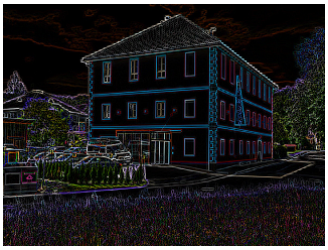
```
# Importing the mamba module.
from mamba import *

def arrowEncoding(imIn, imOut, oper, grid=DEFAULT_GRID):
    """
    This operator encodes the neighborhood of image 'imIn' and puts the result
    in greyscale image 'imOut'. The operator 'oper' may be one of the following:
    'sup', 'supeq', 'inf' or 'infeq'.
    The 'grid' may be an hexagonal or square one. In the first case, the range
    of encodings is [0, 63], in the second case, it is [0, 255].
    """

    imWrk = imageMb(imIn)
    imMask = imageMb(imIn, 1)
    imOut.reset()
    if oper in ["inf", "sup"]:
        strict = True
    elif oper in ["infeq", "supeq"]:
        strict = False
    else:
        strict = ""
        # This statement generates an error if 'oper' is not one of the
```

```
        # allowed ones.

    for i in range(gridNeighbors(grid)):
        dir = i + 1
        if oper in ["sup", "supeq"]:
            shift(imIn, imWrk, dir, 1, computeMaxRange(imIn)[1], grid=grid)
            generateSupMask(imWrk, imIn, imMask, strict)
        else:
            shift(imIn, imWrk, dir, 1, 0, grid=grid)
            generateSupMask(imIn, imWrk, imMask, strict)
        copyBitPlane(imMask, i, imOut)

# Loading the initial image.
im1 = imageMb('burner.png')

# Arrowing the image, result in im2. The operaton is the supremum (an arrow is
# drawn if the neighbor value is greater than the center value), the grid in
# use is the square one.
im2 = imageMb(im1)
arrowEncoding(im1, im2, "sup", grid=SQUARE)
# Storing the encoded image.
im2.save("arrowedBurner.png")
```

3.2 Color channels manipulations

This example shows you how to extract a specific color channel out of an image and how to reconstruct color images with three mamba images and the mix function. The colorful image was taken from wikimedia at http://commons.wikimedia.org/wiki/File:Ebenthal_Schlosswirt_20052010_06.jpg.

colorful.jpg



colorful_grad.png

Here is the source code for this example (can be found in Color channels manipulations):

```
# Importing mamba
from mamba import *

# Opening the image for each color channel (red, green and blue)
imRed = imageMb('colorful.jpg', rgbfilter=(1.0, 0.0, 0.0))
imGreen = imageMb('colorful.jpg', rgbfilter=(0.0, 1.0, 0.0))
imBlue = imageMb('colorful.jpg', rgbfilter=(0.0, 0.0, 1.0))

# We will perform a half gradient on each color channel
halfGradient(imRed, imRed)
halfGradient(imGreen, imGreen)
halfGradient(imBlue, imBlue)

# Then we recombine the three channel image into one using the
# mix function. This function actually returns a PIL image
pilim = mix(imRed, imGreen, imBlue)

# We save it
pilim.save("colorful_grad.png")
```

3.3 Contrast enhancement using top hat operators

This example presents a contrast enhancement technique based on the black top hat and white top hat operators.

snake.png

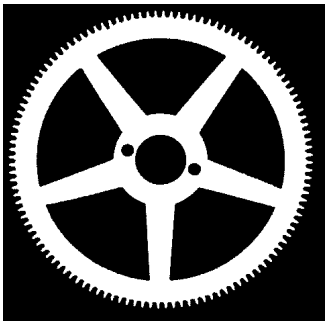
snake_contrast.png

Here is the source code for this example (can be found in Contrast enhancement using top hat operators):

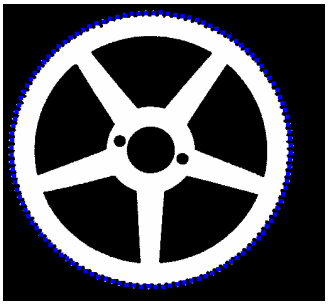
```
# Importing mamba
import mamba

def contrastEnhancer(imIn, imOut, n=1, se=mamba.DEFAULT_SE):
    """
    Increase the contrast of image 'imIn' and put the result in
    image 'imOut'. Parameter 'n' will control the size and 'se' the
    structuring element used by the top hat operators.
    """

    imWrk = mamba.imageMb(imIn)
    mamba.whiteTopHat(imIn, imWrk, n, se=se)
    mamba.add(imIn, imWrk, imOut)
    mamba.blackTopHat(imIn, imWrk, n, se=se)
    mamba.sub(imOut, imWrk, imOut)

im = mamba.imageMb("snake.png")
imContrast = mamba.imageMb(im)
contrastEnhancer(im, imContrast, 10)
imContrast.save("snake_contrast.png")
```

3.4 Counting and marking teeth of a notched wheel

This example shows how a very simple transformation (opening here) 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.



wheel.png



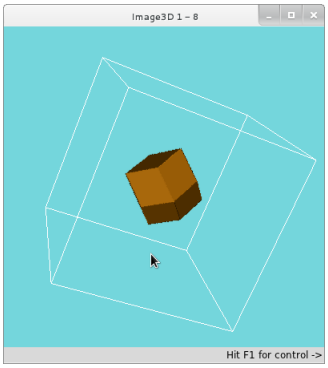
wheel_teeth.png

Here is the source code for this example (can be found in Counting and marking teeth of notched wheel):

```
# Importing mamba
import mamba
import mambaDisplay

im = mamba.imageMb("wheel.png", 1)
im1 = mamba.imageMb(im, 1)
im2 = mamba.imageMb(im, 1)

# Opening of image
mamba.opening(im, im1, 3)
# Selection of the outside region
mamba.negate(im1, im2)
mamba.removeEdgeParticles(im2, im1)
mamba.diff(im2, im1, im2)
# Extracting the wheel teeth
mamba.logic(im, im2, im2, "inf")
# Cleaning the image
mamba.opening(im2, im2)
# Counting and marking each tooth
mamba.thinD(im2, im1)
nb_teeth = mamba.computeVolume(im1)
print("Number of teeth: %d" % (nb_teeth))
mamba.dilate(im1, im1, 3, mamba.SQUARE3X3)
im1.convert(8)
im8 = mamba.imageMb(im, 8)
mamba.convert(im, im8)
mamba.subConst(im8, 1, im8)
mamba.logic(im8, im1, im8, "sup")
name = mambaDisplay.tagOneColorPalette(255, (0,0,255))
im8.save('wheel_teeth.png', palette=mambaDisplay.getPalette(name))
```

3.5 Creating Steiner polyhedrons in 3D

This example shows you how to construct Steiner polyhedrons with dilations. It shows 3D image creation, display and use in computations. This example is also a good introduction to the various 3D grids and their directions.



steiner3D.png

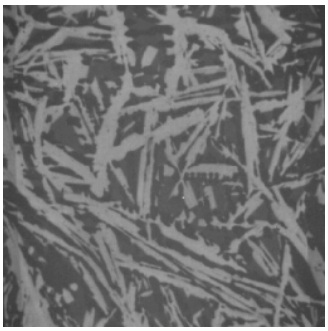
Here is the source code for this example (can be found in Creating Steiner polyhedrons in 3D):

```
# Importing mamba
from mamba3D import *
import mambaDisplay

def steiner3D(imOut, n, directions, grid=DEFAULT_GRID3D):
    """
    Steiner polyhedrons are built by dilating a central pixel along
    a given direction then dilating the result along another direction
    and repeating the process. 'directions' is the list of directions
    according to 'grid'. 'n' indicates the number of times the operation
    is repeated (the size of the resulting polyhedron). This is not
    a processing function, it is just meant to draw polyhedrons so there
    is only an output 'imOut'. The display is updated at the end.
    """
    imOut.reset()
    (w,h,l) = imOut.getSize()

    v = computeMaxRange3D(imOut)[1]
    imOut.setPixel(v, (w/2,h/2,l/2))

    ses = []
    for d in directions:
        ses.append(structuringElement3D([0,d],grid))

    for i in range(n):
        for se in ses:
            dilate3D(imOut, imOut, 1, se=se)
    imOut.update()

im3D = image3DMb()
im3D.show()
mambaDisplay.tagOneColorPalette(255, (255,150,0))
# We will choose orange to display the polyhedron. hit p to select
# the newly created palette.
# At this point, it is worth signaling that the result will look
# better in volume rendering (F2) using the isosurface process
# (Use <Tab> on the display to modify it).

# Drawing a rhombododecahedron
steiner3D(im3D, 30, [11,15,22,26], CUBIC)
# If you look at the display after this step you will see
# a strange blur. It is because the directions does not allow to fill all
```

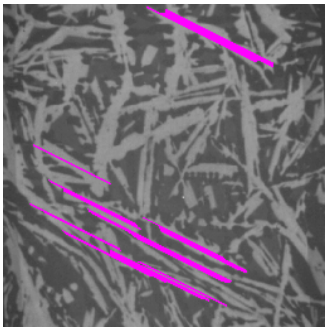
```
# the space so we close the 1 pixel gaps with CUBE3X3X3
closing3D(im3D, im3D, 1, CUBE3X3X3)
im3D.update()
```

3.6 Directional morphological operators in 12 or 16 directions

These transformations use linear structuring elements defined on standard or conjugate directions of the grid in use. These operators (erosion, dilation, opening, closing) are often useful to increase the accuracy of the detection and extraction of oriented objects in an image. The following operators can be used on an hexagonal grid (12 directions are available) or on a square one (16 directions are defined). The directions coding is the same in both cases: direction 1 corresponds to the same direction as in the grid in use (12 o'clock for the square grid and 1 o'clock for the hexagonal one), the numbering increasing clockwise up to 12 or 16.



ferrite.png



oriented_fibers.png

Here is the source code for this example (can be found in Directional morphological operators in 12 or 16 directions):

```
# Importing mamba
import mamba
import mambaDisplay.extra as mE

# This operator is used by the general directional erosion.
def conjugateDirectionalErode(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    Performs the erosion of image 'imIn' in the conjugate direction 'd' of the
    grid and puts the result in image 'imOut'. The images can be binary, greyscale
    or 32-bit images. 'size' is a multiple of the distance between two adjacent
    points in the conjugate directions. 'edge' is set to FILLED by default (can
    be changed).
    Note that this operator is not equivalent to successive erosions by a doublet
    of points.
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    mamba.copy(imIn, imOut)

    j3 = mamba.transposeDirection(d, grid=grid)
    j2 = mamba.rotateDirection(d, grid=grid)
    j1 = mamba.transposeDirection(j2, grid=grid)
    val = mamba.computeMaxRange(imIn)[1] * int(edge == mamba.FILLED)
    for i in range(size):
```

```
        mamba.copy(imOut, imWrk1)
        mamba.copy(imOut, imWrk2)
        mamba.linearErode(imWrk1, imWrk1, d, n=1, grid=grid, edge=edge)
        mamba.shift(imWrk1, imWrk1, j1, 1, val, grid=grid)
        mamba.logic(imWrk1, imOut, imWrk1, "inf")
        mamba.linearErode(imWrk2, imWrk2, j2, n=1, grid=grid, edge=edge)
        mamba.shift(imWrk2, imWrk2, j3, 1, val, grid=grid)
        mamba.logic(imWrk2, imOut, imWrk2, "inf")
        mamba.logic(imWrk1, imWrk2, imOut, "sup")

def directionalErode(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    General directional erosion of image 'imIn', result in 'imOut'. The images
    can have any format. The direction 'd' is coded clockwise from 1 to 12 or
    16. Direction 1 corresponds to 60 degrees on the hexagonal grid and 90
    degrees on the square grid. 'size' corresponds to twice the normal size
    when standard directions are used. 'edge' is set to FILLED by default.
    """

    if (d & 1) == 0:
        conjugateDirectionalErode(imIn, imOut, d/2, size, grid=grid, edge=edge)
    else:
        mamba.linearErode(imIn, imOut, (d+1)/2, size*2, grid=grid, edge=edge)

# similar operators are defined for the dilation.
def conjugateDirectionalDilate(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.EMPTY):
    """
    Performs the dilation of image 'imIn' in the conjugate direction 'd' of the
    grid and puts the result in image 'imOut'. The images can be binary, greyscale
    or 32-bit images. 'size' is a multiple of the distance between two adjacent
    points in the conjugate directions. 'edge' is set to EMPTY by default (can
    be changed).
    Note that the linear structuring element is not connected. Points connecting
    adjacent points in the conjugate directions are not present. This is normal
    if we want to insure that the result is identical when we iterate n size 1
    operations to get a size n one (the same remark applies to the erosion).
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    mamba.copy(imIn, imOut)

    j3 = mamba.transposeDirection(d, grid=grid)
    j2 = mamba.rotateDirection(d, grid=grid)
    j1 = mamba.transposeDirection(j2, grid=grid)
    val = mamba.computeMaxRange(imIn)[1] * int(edge == mamba.FILLED)
    for i in range(size):
        mamba.copy(imOut, imWrk1)
        mamba.copy(imOut, imWrk2)
        mamba.linearDilate(imWrk1, imWrk1, d, 1, grid=grid, edge=edge)
        mamba.shift(imWrk1, imWrk1, j1, 1, val, grid=grid)
        mamba.logic(imWrk1, imOut, imWrk1, "sup")
        mamba.linearDilate(imWrk2, imWrk2, j2, 1, grid=grid, edge=edge)
        mamba.shift(imWrk2, imWrk2, j3, 1, val, grid=grid)
        mamba.logic(imWrk2, imOut, imWrk2, "sup")
        mamba.logic(imWrk1, imWrk2, imOut, "inf")

def directionalDilate(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.EMPTY):
    """
    General directional dilation of image 'imIn', result in 'imOut'. The images
    can have any format. The direction 'd' is coded clockwise from 1 to 12 or
    16. Direction 1 corresponds to 60 degrees on the hexagonal grid and 90
    degrees on the square grid. 'size' corresponds to twice the normal size
    when standard directions are used. 'edge' is set to EMPTY by default.
```

```
    """

    if (d & 1) == 0:
        conjugateDirectionalDilate(imIn, imOut, d/2, size, grid=grid, edge=edge)
    else:
        mamba.linearDilate(imIn, imOut, (d+1)/2, size*2, grid=grid, edge=edge)

def directionalOpen(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    Directional opening of image 'imIn' defined by combining an erosion in
    direction 'd' followed by a dilation in the transposed direction (which
    depends on the grid in use). Result is put in 'imOut'.
    """

    directionalErode(imIn, imOut, d, size, grid=grid, edge=edge)
    j = (d + mamba.gridNeighbors(grid=grid) - 1) % (mamba.gridNeighbors(grid=grid) * 2) + 1
    directionalDilate(imOut, imOut, j, size, grid=grid)

def directionalClose(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    Directional closing of image 'imIn' defined by combining a dilation in
    direction 'd' followed by an erosion in the transposed direction (which
    depends on the grid in use). Result is put in 'imOut'.
    If 'edge' is set to 'EMPTY', the operation must be modified to remain extensive.
    """

    imWrk = mamba.imageMb(imIn)
    if edge==mamba.EMPTY:
        mamba.copy(imIn, imWrk)
    directionalDilate(imIn, imOut, d, size, grid=grid, edge=edge)
    j = (d + mamba.gridNeighbors(grid=grid) - 1) % (mamba.gridNeighbors(grid=grid) * 2) + 1
    directionalErode(imOut, imOut, j, size, grid=grid)
    if edge==mamba.EMPTY:
        mamba.logic(imOut, imWrk, imOut, "sup")

# Trying these operators on a fibers image.
# Loading the initial image.
im1 = mamba.imageMb('ferrite.png')
# Defining some working images.
im2 = mamba.imageMb(im1)
imbin1 = mamba.imageMb(im1, 1)
# Performing a directional opening of size 30 in direction 6.
# This should select fibers with a length greater than 60 in
# direction 165 degrees approximately (a square grid is used).
directionalOpen(im1, im2, 6, 30, grid=mamba.SQUARE)
# The markers (obtained by a simple threshold) of the corresponding
# fibers are superposed to the original image (some filtering is
# performed by removing fibers cutting the edges).
mamba.threshold(im2, imbin1, 100, 255)
mamba.removeEdgeParticles(imbin1, imbin1)
# The same operation is performed once again on the thresholded
# image to verify the markers sizes.
directionalOpen(imbin1, imbin1, 6, 30, grid=mamba.SQUARE)
# displaying the result.
mE.superpose(im1, imbin1)
```

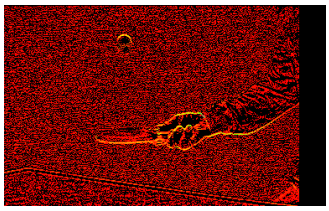
3.7 Directional moving edge measurement

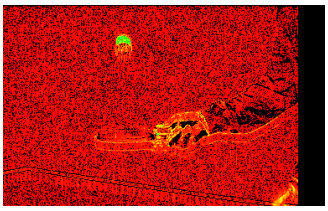
This example implements a directional MEM algorithm (also called T-Rex vision). This allows to identify the movement in a sequence of images using their gradients. Among other things, this example shows how to use sequenceMb. You can find a description of this algorithm in http://cmm.ensmp.fr/ beucher/publi/Ibpria_camera_ready.pdf.

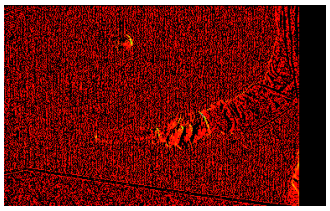
tennis/001.png

tennis/002.png

tennis/003.png


mcm_d1_n1.png


mcm_d1_n10.png


mcm_d3_n1.png

Here is the source code for this example (can be found in Directional moving edge measurement):

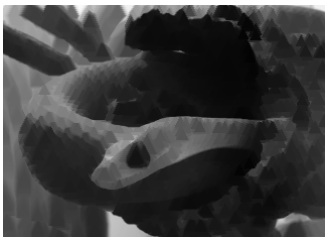
```
# Importing mamba and mamba3D
from mamba3D import *
from mamba import *
import mambaDisplay

def dirMEM(seqIn, imOut, d, n=1, grid=DEFAULT_GRID):
    """
    Directional MEM algorithm. This algorithm will indicates the
    boundaries of objects moving in the direction 'd'. To work it needs
    a sequence 'seqIn' with at least three images in it.
    """
    grad0 = imageMb(seqIn[0])
    grad1 = imageMb(seqIn[0])
    grad2 = imageMb(seqIn[0])
    imWrk1 = imageMb(seqIn[0])
    imWrk2 = imageMb(seqIn[0])

    # direction d converted into a structuring element
    sed = structuringElement([0,d], grid)

    # Computing the gradients of the first three images of the sequence
    halfGradient(seqIn[0], grad0, n=n, se=sed)
    halfGradient(seqIn[1], grad1, n=n, se=sed)
    halfGradient(seqIn[2], grad2, n=n, se=sed)

    # First we compute the absolute difference between the first two
    # gradients
    sub(grad0, grad1, imWrk1)
    sub(grad1, grad0, imWrk2)
    logic(imWrk1, imWrk2, imOut, "sup") # imOut = abs(grad1-grad0)
    # Then we compute the absolute difference between the last two
    # gradients
    sub(grad2, grad1, imWrk1)
    sub(grad1, grad2, imWrk2)
    logic(imWrk1, imWrk2, imWrk1, "sup") # imWrk1 = abs(grad1-grad0)
```

```
    # And last imOut is set to the inf of this two differences
    logic(imOut, imWrk1, imOut, "inf")

seq = sequenceMb("tennis")
im = imageMb(seq[0])

# Testing it with various direction and size
dirMEM(seq, im, d=1, n=1, grid=SQUARE)
im.save("mcm_d1_n1.png", palette=mambaDisplay.getPalette("rainbow"))
dirMEM(seq, im, d=1, n=10, grid=SQUARE)
im.save("mcm_d1_n10.png", palette=mambaDisplay.getPalette("rainbow"))
dirMEM(seq, im, d=3, n=1, grid=SQUARE)
im.save("mcm_d3_n1.png", palette=mambaDisplay.getPalette("rainbow"))
```

3.8 Erosion and structuring elements

This example shows you how to create a structuring element and how to use it with the erode function.

snake.png



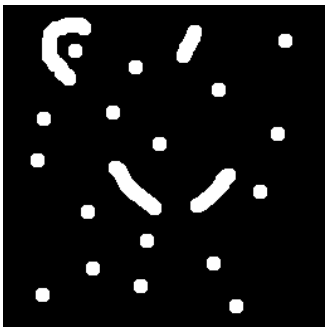
snake_ero10rt.png

Here is the source code for this example (can be found in Erosion and structuring elements):

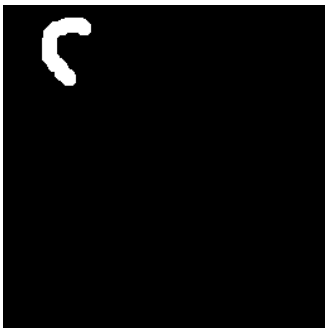
```
# Importing mamba
from mamba import *

# Opening and creating images
im1 = imageMb('snake.png')
im2 = imageMb(im1)

# Creating a reversed triangle (as opposed to the TRIANGLE found in Mamba
# source) structuring element
reverse_triangle = structuringElement([0,1,6], grid=HEXAGONAL)

# Performing an erosion of size 10 with the created structuring element
erode(im1, im2, 10, se=reverse_triangle)

im2.save("snake_ero10rt.png")
```

3.9 Extraction of the first particle

Extraction of the first particle (in scanning order) of a binary image.



particles.png



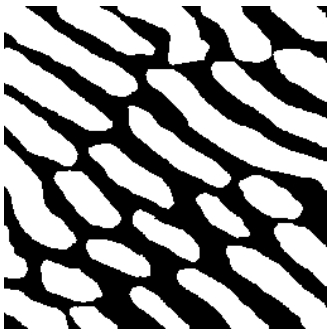
first_particle.png

Here is the source code for this example (can be found in Extraction of the first particle):

```
# Importing mamba
import mamba

def firstParticle(imIn, imOut, grid=mamba.DEFAULT_GRID):
    """
    Extraction of the first particle (in scanning order) of binary image imIn.
    The particle is put into image imOut and removed from imIn.

    If imIn is empty, imOut is also empty.
    imIn and imOut must be different.
    """

    imWrk = mamba.imageMb(imIn)
    mamba.compare(imIn, imWrk, imWrk)
    mamba.build(imIn, imWrk, grid=grid)
    mamba.diff(imIn, imWrk, imIn)
    mamba.copy(imWrk, imOut)

im = mamba.imageMb("particles.png", 1)
imFP = mamba.imageMb(im)
firstParticle(im, imFP)
imFP.save("first_particle.png")
```

3.10 Extremities of particles

This example illustrates the use of a geodesic distance function to extract the extremities of simply connected particles (without holes). Firstly, the centroid of each particle is obtained with an homotopic thinning with a D structuring element (this centroid is a sufficient approximation of the geodesic center). Then a geodesic distance function of each particle without its centroid is computed, the geodesic space being the particle itself. Finally, the extremities correspond to the maxima of this geodesic distance.



eutectic.png

hand.png

eutectic_extremities.png

hand_extremities.png

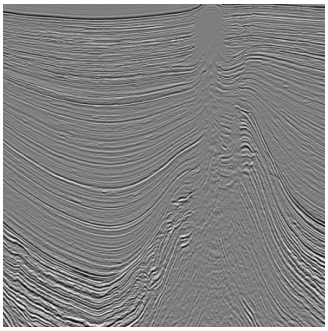
Here is the source code for this example (can be found in Extremities of particles):

```
# Importing the mamba modules
from mamba import *

def extremities(imIn, imOut, innerParticles=False, grid=DEFAULT_GRID):
    """
    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 and 'imOut' 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).
    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)
    imWrk2 = imageMb(imIn, 32)

    # Computation of the centroids.
    thinD(imIn, imWrk1, grid=grid)
    # These centroids are removed from the initial image (each particle contains
    # a hole.
    diff(imIn, imWrk1, imWrk1)
    # Computing the geodesic distance (the elementary structuring element is
    # determined by the grid in use).
```

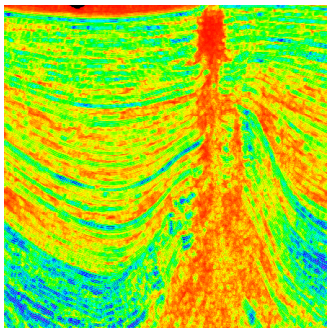
```
        se = structuringElement(getDirections(grid), grid)
        geodesicDistance(imWrk1, imIn, imOut, se=se)
        # The extremities correspond to the maxima.
        maxima(imOut, imWrk1, grid=grid)
        # Extremities on the edge are removed if 'innerParticles' is set to False.
        if not(innerParticles):
            removeEdgeParticles(imWrk1, imWrk1, grid=grid)
        # The extremities are given their corresponding distance to the center.
        convert(imWrk1, imWrk2)
        logic(imOut, imWrk2, imOut, "inf")

# The procedure is applied on two different images
# Opening and creating the first image
im1 = imageMb('eutectic.png')
imTemp = imageMb(im1, 1)
imResult = imageMb(im1, 32)
im1.convert(1)
# A small filtering is performed to remove irregularities.
opening(im1, imTemp)
# Extraction of the extremities. In this case, the particles are considered
# to be inside the window. So extremities are detected on the edge.
extremities(imTemp, imResult, innerParticles=True)
# The binary image of the extremities is saved (slightly dilated to make
# them visible).
threshold(imResult, imTemp, 1, computeMaxRange(imResult)[1])
dilate(imTemp, imTemp)
imTemp.save('eutectic_extremities.png')
# The same operation is performed on the second image. The only difference
# concerns 'innerParticles'. Here, the hand is supposed to extend outside
# the window.
im2 = imageMb('hand.png')
imTemp = imageMb(im2, 1)
imResult = imageMb(im2, 32)
im2.convert(1)
opening(im2, imTemp)
extremities(imTemp, imResult)
threshold(imResult, imTemp, 1, computeMaxRange(imResult)[1])
dilate(imTemp, imTemp)
imTemp.save('hand_extremities.png')
```

3.11 Modulus and azimut of a directional gradient

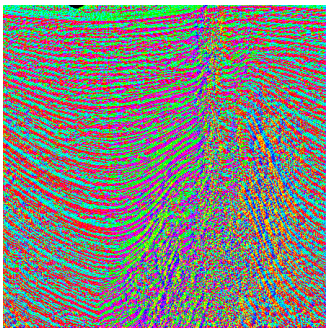
This example uses the linear operators defined in example M20 to compute a morphological gradient modulus and its azimut function. Directional gradients are calculated and the modulus corresponds to the supremum of all these directional gradients. The azimut corresponds to the direction where this supremum occurs. This example has been designed on the hexagonal grid but it could be adapted to the square grid without any difficulty.



seismic_section.png



gradient_modulus.png



gradient_azimut.png

Here is the source code for this example (can be found in Modulus and azimut of directional gradient):

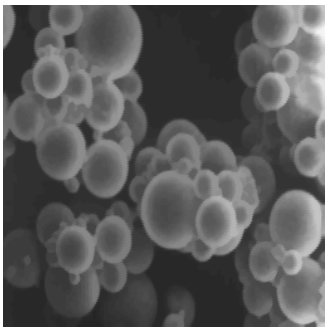
```
# Importing mamba
import mamba
import mambaDisplay

# Operators defined in example M20 are duplicated here.
def conjugateDirectionalErode(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    Performs the erosion of image 'imIn' in the conjugate direction 'd' of the
    grid and puts the result in image 'imOut'. The images can be binary, greyscale
    or 32-bit images. 'size' is a multiple of the distance between two adjacent
    points in the conjugate directions. 'edge' is set to FILLED by default (can
    be changed).
    Note that this operator is not equivalent to successive erosions by a doublet
    of points.
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    mamba.copy(imIn, imOut)

    j3 = mamba.transposeDirection(d, grid=grid)
    j2 = mamba.rotateDirection(d, grid=grid)
    j1 = mamba.transposeDirection(j2, grid=grid)
    val = mamba.computeMaxRange(imIn)[1] * int(edge == mamba.FILLED)
    for i in range(size):
        mamba.copy(imOut, imWrk1)
        mamba.copy(imOut, imWrk2)
```

```
        mamba.linearErode(imWrk1, imWrk1, d, n=1, grid=grid, edge=edge)
        mamba.shift(imWrk1, imWrk1, j1, 1, val, grid=grid)
        mamba.logic(imWrk1, imOut, imWrk1, "inf")
        mamba.linearErode(imWrk2, imWrk2, j2, n=1, grid=grid, edge=edge)
        mamba.shift(imWrk2, imWrk2, j3, 1, val, grid=grid)
        mamba.logic(imWrk2, imOut, imWrk2, "inf")
        mamba.logic(imWrk1, imWrk2, imOut, "sup")

def directionalErode(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    General directional erosion of image 'imIn', result in 'imOut'. The images
    can have any format. The direction 'd' is coded clockwise from 1 to 12 or
    16. Direction 1 corresponds to 60 degrees on the hexagonal grid and 90
    degrees on the square grid. 'size' corresponds to twice the normal size
    when standard directions are used. 'edge' is set to FILLED by default.
    """

    if (d & 1) == 0:
        conjugateDirectionalErode(imIn, imOut, d/2, size, grid=grid, edge=edge)
    else:
        mamba.linearErode(imIn, imOut, (d+1)/2, size*2, grid=grid, edge=edge)

# similar operators are defined for the dilation.
def conjugateDirectionalDilate(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.EMPTY):
    """
    Performs the dilation of image 'imIn' in the conjugate direction 'd' of the
    grid and puts the result in image 'imOut'. The images can be binary, greyscale
    or 32-bit images. 'size' is a multiple of the distance between two adjacent
    points in the conjugate directions. 'edge' is set to EMPTY by default (can
    be changed).
    Note that the linear structuring element is not connected. Points connecting
    adjacent points in the conjugate directions are not present. This is normal
    if we want to insure that the result is identical when we iterate n size 1
    operations to get a size n one (the same remark applies to the erosion).
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    mamba.copy(imIn, imOut)

    j3 = mamba.transposeDirection(d, grid=grid)
    j2 = mamba.rotateDirection(d, grid=grid)
    j1 = mamba.transposeDirection(j2, grid=grid)
    val = mamba.computeMaxRange(imIn)[1] * int(edge == mamba.FILLED)
    for i in range(size):
        mamba.copy(imOut, imWrk1)
        mamba.copy(imOut, imWrk2)
        mamba.linearDilate(imWrk1, imWrk1, d, 1, grid=grid, edge=edge)
        mamba.shift(imWrk1, imWrk1, j1, 1, val, grid=grid)
        mamba.logic(imWrk1, imOut, imWrk1, "sup")
        mamba.linearDilate(imWrk2, imWrk2, j2, 1, grid=grid, edge=edge)
        mamba.shift(imWrk2, imWrk2, j3, 1, val, grid=grid)
        mamba.logic(imWrk2, imOut, imWrk2, "sup")
        mamba.logic(imWrk1, imWrk2, imOut, "inf")

def directionalDilate(imIn, imOut, d, size, grid=mamba.DEFAULT_GRID, edge=mamba.EMPTY):
    """
    General directional dilation of image 'imIn', result in 'imOut'. The images
    can have any format. The direction 'd' is coded clockwise from 1 to 12 or
    16. Direction 1 corresponds to 60 degrees on the hexagonal grid and 90
    degrees on the square grid. 'size' corresponds to twice the normal size
    when standard directions are used. 'edge' is set to EMPTY by default.
    """
```

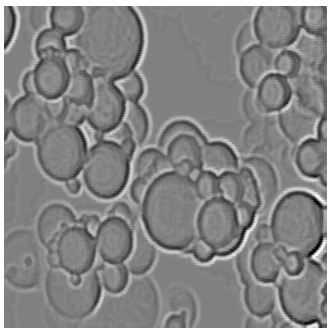
```
    if (d & 1) == 0:
        conjugateDirectionalDilate(imIn, imOut, d/2, size, grid=grid, edge=edge)
    else:
        mamba.linearDilate(imIn, imOut, (d+1)/2, size*2, grid=grid, edge=edge)

# We define the vectorial gradient procedure.
def vectorGradient(imIn, imModul, imAzim, size=1):
    """
    Computes modulus (result in 'imModul') and azimut (in 'imAzim') of image
    'imIn'. The 'size' of each directional gradient is set to 1 by default.
    This operator is defined on the hexagonal grid (12 directions are used).
    Note that this operator belongs to the residual operators class.
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    imWrk3 = mamba.imageMb(imIn)
    imWrk4 = mamba.imageMb(imIn, 1)

    mamba.copy(imIn, imWrk1)
    imModul.reset()
    imAzim.reset()
    for i in range(12):
        d = i + 1
        # Directional gradient obtained with linear erosions and dilations.
        directionalDilate(imWrk1, imWrk2, d, size)
        directionalErode(imWrk1, imWrk3, d, size)
        mamba.sub(imWrk2, imWrk3, imWrk2)
        # For each direction, maximal pixels are indexed in a mask image.
        mamba.generateSupMask(imWrk2, imModul, imWrk4, True)
        mamba.convertByMask(imWrk4, imWrk3, 0, d)
        # Modulus and azimut are calculated.
        mamba.logic(imWrk2, imModul, imModul, "sup")
        mamba.logic(imWrk3, imAzim, imAzim, "sup")

# Using this operator on a seismic section image.

# Reading the initial image and defining results images.
imA = mamba.imageMb('seismic_section.png')
imB = mamba.imageMb(imA)
imC = mamba.imageMb(imA)

# Calculating the vectorial gradient.
vectorGradient(imA, imB, imC)

# A palette for displaying the 12 possible directions is defined.
dirpal = (0, 0, 0, 255, 0, 0, 255, 128, 0, 255, 255, 0, 128, 255, 0, 0, 255, 0)
dirpal += (0, 255, 128, 0, 255, 255, 0, 128, 255, 0, 0, 255)
dirpal += (128, 0, 255, 255, 0, 255, 255, 0, 128)
dirpal += 243 * (0, 0, 0)
mambaDisplay.addPalette("direction palette", dirpal)

# Saving the results.
# Applying the new palette to the azimut image (and the classical
# rainbow palette to the modulus).
imB.save('gradient_modulus.png', palette=mambaDisplay.getPalette("rainbow"))
imC.save('gradient_azimut.png', palette=mambaDisplay.getPalette("direction palette"))
# The last line could have been
# imC.save('gradient_azimut.png', palette=dirpal)
```

3.12 Morphological laplacian and zero-crossings detection

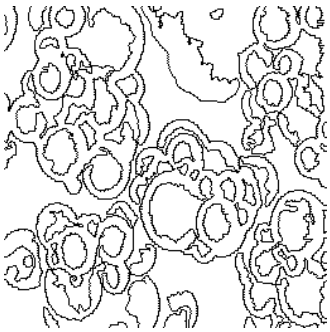
This example shows how to define the morphological equivalent of the Laplacian operator. This morphological Laplacian is normalised and limited to the range [0, 255]. The example also shows how the zero-crossings of this Laplacian (supposed to correspond to contours of salient objects) can be obtained by a SKIZ.



flocs.png



laplacian.png



zero-crossings.png

Here is the source code for this example (can be found in Morphological laplacian and zerocrossings detection):

```
# Importing mamba
from mamba import *

# The Laplacian will use the squeeze operator which has been introduced in
# example E12.
def squeeze(imIn, imOut, value1, value2):
    """
    Squeezes the pixel values of 'imIn' between 'value1' and 'value2'. If the
    pixel value is larger than 'value2', it is replaced by 'value2'. If it lower
    than 'value1', it is replaced by 'value1'. In-between values are unchanged.
    'imIn' and 'imOut' must have the same depth.
    """

    imWrk1 = imageMb(imIn, 1)
    imWrk2 = imageMb(imIn)
    # Replacing values lower than value1 by value1.
    threshold(imIn, imWrk1, 0, value1)
    convertByMask(imWrk1, imWrk2, 0, value1)
    logic(imIn, imWrk2, imOut, "sup")
    # Replacing values higher than value2 by value2.
    threshold(imOut, imWrk1, 0, value2)
    convertByMask(imWrk1, imWrk2, value2, computeMaxRange(imOut)[1])
    logic(imOut, imWrk2, imOut, "inf")

def laplacian(imIn, imOut, n=1, se=DEFAULT_SE):
    """
    Computes the morphological Laplacian (difference of outer and inner
    gradients) of greyscale image 'imIn' and puts the result in greyscale
```

```
    image 'imOut'. The result is normalised (squeezed) between 0 and 255.
    Negative values are shifted between 0 and 127, positive ones between
    129 and 255. The value 128 corresponds to the zeros of the Laplacian.
    The size 'n' of the Laplacian can be defined (default value is 1).
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn, 32)
    halfGradient(imIn, imWrk1, "extern", n=n, se=se)
    halfGradient(imIn, imWrk2, "intern", n=n, se=se)
    sub(imWrk1, imWrk2, imWrk3)
    addConst(imWrk3, 512, imWrk3)
    squeeze(imWrk3, imWrk3, 384, 639)
    subConst(imWrk3, 384, imWrk3)
    copyBytePlane(imWrk3, 0, imOut)

# Loading the initial image and defining the resulting image.
im1 = imageMb('flocs.png')
im2 = imageMb(im1)
# Performing a laplacian of size 3.
laplacian(im1, im2, 3)
# Storing the resulting image.
im2.save('laplacian.png')
# Defining other working images.
im3 = imageMb(im1)
imMarkers = imageMb(im1, 32)
imbin1 = imageMb(im1, 1)
imbin2 = imageMb(im1, 1)
# Determining the zero-crossings pixels.
imMarkers.reset()
# Mask of pixels corresponding to positive laplacian.
threshold(im2, imbin1, 129, 255)
# Removing small connected components.
buildOpen(imbin1, imbin1, 2)
# Giving them value equal to 2.
add(imMarkers, imbin1, imMarkers)
add(imMarkers, imbin1, imMarkers)
# Mask of negative pixels and filtering.
threshold(im2, imbin2, 0, 127)
buildOpen(imbin2, imbin2, 2)
# Generating the labels image...
add(imMarkers, imbin2, imMarkers)
# ...and the image to be processed.
logic(imbin1, imbin2, imbin1, "sup")
convertByMask(imbin1, im3, 1, 0)
# The SKIZ is obtained by means of a watershed transform.
basinSegment(im3, imMarkers)
# The boudaries between catchment basins correspond to
# zero-crossing pixels.
copyBytePlane(imMarkers, 0, im3)
halfGradient(im3, im3)
threshold(im3, imbin1, 0, 0)
# Saving the contours.
imbin1.save('zero-crossings.png')
```

3.13 Noise reduction using alternate filters

This example introduces a new full alternate filter based on buildOpen and buildClose operators. The new filter reduces efficiently noise while producing a much less fuzzy image than the classic full alternate filter.

lena_noisy.png

lena_FAF.png

lena_FBAF.png

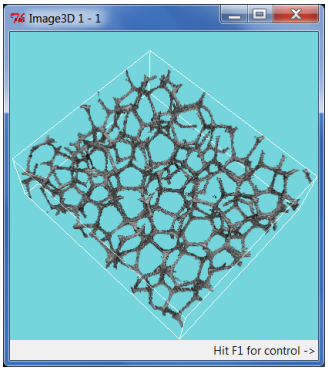
Here is the source code for this example (can be found in Noise reduction using alternate filters):

```
# Importing mamba
import mamba

def fullBuildAlternateFilter(imIn, imOut, n, openFirst, se=mamba.DEFAULT_SE):
    """
    Performs a full alternate filter operation (successive alternate filters of
    increasing sizes, from 1 to 'n') on image 'imIn' and puts the result
    in 'imOut'. 'n' controls the filter size. If 'openFirst' is True, the filter
    begins with an opening, a closing otherwise. This operator uses the
    buildOpen and buildClose operators instead of the classic open and close.
    """

    mamba.copy(imIn, imOut)
    for i in range(1,n+1):
        if openFirst:
            mamba.buildOpen(imOut, imOut, i, se=se)
            mamba.buildClose(imOut, imOut, i, se=se)
        else:
            mamba.buildClose(imOut, imOut, i, se=se)
            mamba.buildOpen(imOut, imOut, i, se=se)

im = mamba.imageMb("lena_noisy.png")
imFilter = mamba.imageMb(im)

# First we compute the result of the standard full alternate filter
# on the noisy lena image
mamba.fullAlternateFilter(im, imFilter, 3, True)
imFilter.save("lena_FAF.png")
# Then we compute the result of the reconstruction based full alternate filter
# on the noisy lena image
```

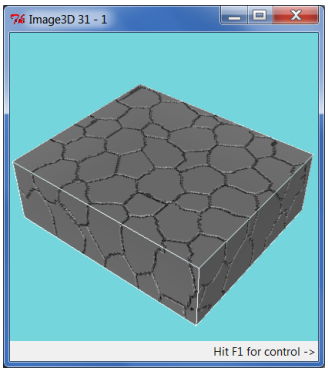
```
fullBuildAlternateFilter(im, imFilter, 3, True)
imFilter.save("lena_FBAF.png")
```

3.14 Pellets separation 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 availability in mamba3D module of operators which are the simple transposition of 2D operators. The initial 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 in exampleA2.py (coffee grains separation and counting).



foam3D.png



segmented_pellets.png

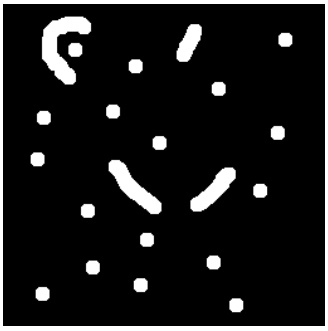
Here is the source code for this example (can be found in Pellets separation in 3D polyurethane foam):

```
# Importing the mamba and mamba3D modules.
from mamba import *
from mamba3D import *

# Importing the initial image and converting it to binary.
imA = image3DMb('foam')
imA.convert(1)
# The initial image is filtered (holes are closed and a reconstruction opening
# is performed to remove small artifacts which could be present in th 3D volume.
imB = image3DMb(imA)
closeHoles3D(imA, imB)
imC = image3DMb(imA)
supOpen3D(imB, imC, 2)
build3D(imB, imC)
# 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)
# We verify that the maximum distance is less than 256 so that the result can
# be transferred into a greyscale image.
tMax = computeRange3D(imE)[1]
```

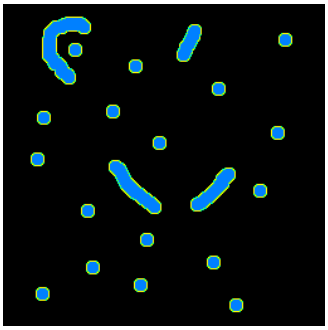
```
print("maximum size of pellets: %d" % (tMax))
imF = image3DMb(imA, 8)
copyBytePlane3D(imE, 0, imF)
# The distance function is filtered in order to keep its most significant
# maxima (markers of the pellets). Their number gives the number of pellets.
imG = image3DMb(imA, 8)
opening3D(imF, imG, 3)
maxima3D(imG, imB)
# The distance function is inverted.
negate3D(imF, imF)
# labelling the markers and printing the number of pellets.
nbPellets = label3D(imB, imE)
print("Number of polyurethane pellets : %d" % (nbPellets))
# A 3D watershed of the inverted distance function is performed.
watershedSegment3D(imF, imE)
copyBytePlane3D(imE, 3, imG)
imH = image3DMb(imA)
# The watershed is displayed. Watershed surfaces correspond to the separations
# between the pellets.
threshold3D(imG, imH, 0, 0)
imH.show(mode="VOLUME")
```

3.15 Pixel neighbors counter

This example shows how to use the hit-or-miss operator along with the double structuring element class in Mamba to count the number of neighbors set to true each true pixel has.



particles.png



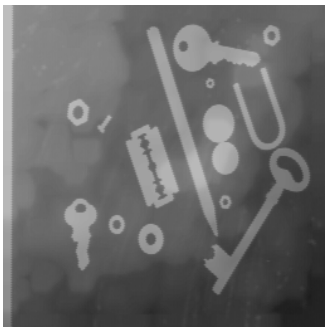
particles_nbg.png

Here is the source code for this example (can be found in Pixel neighbors counter):

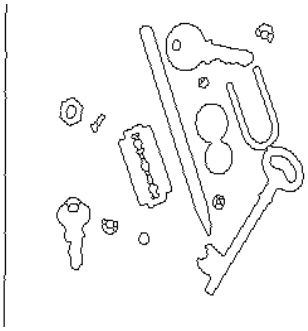
```
# Importing the mamba package
import mamba

def neighborCounter(imIn, imOut, grid=mamba.DEFAULT_GRID):
    """
    For each pixel set to true in the binary image 'imIn', this function
    counts its neighbor set to true and puts the result in 'imOut'. The
    neighbors are selected according to 'grid'.
    """

    imWrk = mamba.imageMb(imIn)
    imOut.reset()

    for d in mamba.getDirections(grid)[1:]:
        dse = mamba.doubleStructuringElement([],[0,d],grid)
        mamba.hitOrMiss(imIn, imWrk, dse)
        mamba.add(imOut, imWrk, imOut)

im = mamba.imageMb("particles.png", 1)
imCount = mamba.imageMb(im, 8)
neighborCounter(im, imCount, mamba.SQUARE)
# Creating a specific palette to see more clearly the possible values
# wich range from 0 to 8 for a square grid.
palette = (0,0,0,255,0,0,0,255,0,255,128,0,0,255,128,255,255,0,0,255,255,128,255,0,0,128,255)+247
imCount.save("particles_nbg.png", palette=palette)
```

3.16 Segmentation algorithms : examples and demo

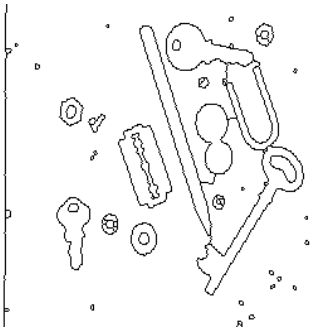
This example shows the effect of the various segmentation algorithms embedded in the hierarchies module. These algorithms can produce quite impressive results but their usage is a bit tricky. This example is intended as a showroom of their results and to illustrate some of their properties (quality, speed, ...). Keep in mind that the results presented here are only valid for our test image and may be completely different on other images. See http://cmm.ensmp.fr/ beucher/publi/Unified_Segmentation.pdf and http://cmm.ensmp.fr/ beucher/publi/P-Algorithm_SB_BM.pdf for more information on these algorithms.



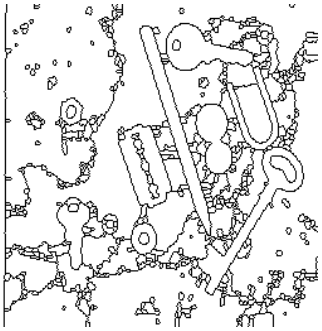
tools.png



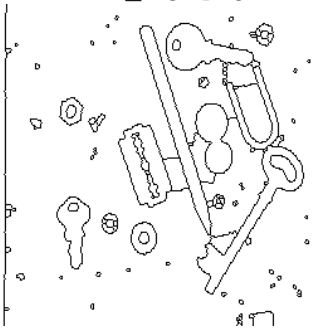
tools_segP.png



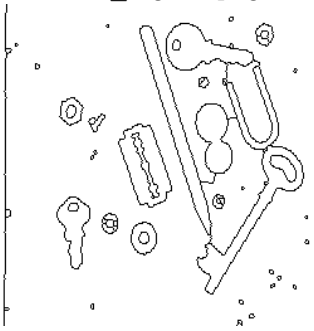
tools_segEW.png



tools_segStd_g20.png



tools_segStd_g15.png



tools_segStd_g10.png

tools_segExt.png

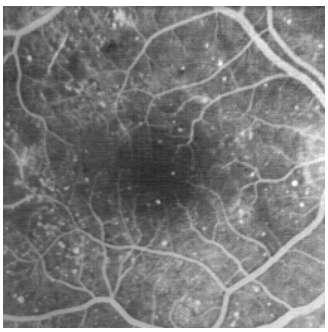
Here is the source code for this example (can be found in Segmentation algorithms examples and demo):

```
# Importing mamba
from mamba import *

im = imageMb("tools.png")

im1 = imageMb(im)
im2 = imageMb(im)
im3 = imageMb(im)
im4 = imageMb(im, 1)
im5 = imageMb(im)

# First computing the valued watershed of our image gradient
gradient(im, im1)
```

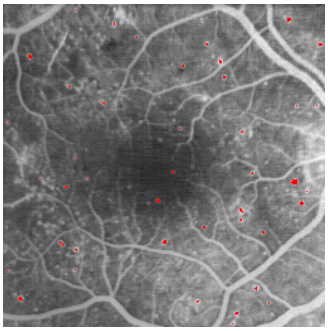
```
valuedWatershed(im1, im2)

# Enhanced Waterfalls
#####################
n = enhancedWaterfalls(im2, im3)
threshold(im3, im4, 0, n-1)
print("Enhanced waterfalls, levels = %d" % (n))
im4.save("tools_segEW.png")

# Standard algorithm
####################
# With a gain of 2 (default), it does not produce very better results
# than the enhanced waterfalls.
n = standardSegment(im2, im3)
threshold(im3, im4, 0, n-1)
print("Standard algorithm, levels = %d" % (n))
im4.save("tools_segStd_g20.png")
# Let's try with a smaller gain, like 1.5
# It produces what seems to be a better result
n = standardSegment(im2, im3, gain=1.5)
threshold(im3, im4, 0, n-1)
print("Standard algorithm, levels = %d" % (n))
im4.save("tools_segStd_g15.png")
# So why not lower the gain again ?
# Well if the result seems equal to the enhanced it's because
# standardSegment with a 1.0 gain is equal to
# enhancedWaterfalls.
n = standardSegment(im2, im3, gain=0.5)
threshold(im3, im4, 0, n-1)
print("Standard algorithm, levels = %d" % (n))
im4.save("tools_segStd_g10.png")
# Regarding the gain, you should note that a greater gain will
# produce a more segmented image. Gains below 1.0 produces the
# same result as 1.0.

# P algorithm
#############
# Note that in this case, the P algorithm produces the best result.
# It is however one of the slowest (more hierarchical levels).
n = segmentByP(im2, im3)
threshold(im3, im4, 0, n-1)
print("P algorithm, levels = %d" % (n))
im4.save("tools_segP.png")

# Extended algorithm (experimental)
#######################################
# This algorithm is allowing you to control the gain for
# each pixel of your image. This can produce interesting
# result such as segmenting only the object inside
# a certain focus area (objects outside or too big to fit
# are removed).
(w, h) = im.getSize()
im5.fill(1)
drawFillCircle(im5, (w/2, h/2, w/4), 2)
mul(im5, im2, im5)
n = extendedSegment(im2, im5, im3)
threshold(im3, im4, 0, n-1)
print("Extended algorithm, levels = %d" % (n))
im4.save("tools_segExt.png")
```

3.17 Simple and coarse extraction of micro-aneurisms in retina

Micro-aneurisms are abnormal inflations of blood vessels in the retina (eye fundus). They appear as white dots in the retina image. This example shows a simple extraction technique based on a Top-Hat transform built with linear openings and geodesic reconstructions.



retina.png



aneurisms.png

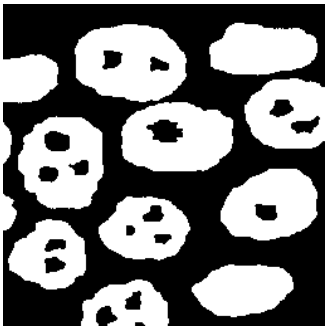
Here is the source code for this example (can be found in Simple and coarse extraction of microaneurisms in retina):

```
# Importing the mamba modules
from mamba import *
import mambaDisplay

# Top-hat operator built with linear openings and a geodesic reconstruction.
def buildSupWhiteTopHat(imIn, imOut, size):
    """
    Top-hat operator applied on image 'imIn' obtained by a supremum of linear
    openings of size 'size' and a reconstruction. The result is put in 'imOut'.
    This operator extracts white objects with a thickness less than 2*'size'
    in all directions.
    """
    imWrk = imageMb(imIn)
    supOpen(imIn, imWrk, size)
    build(imIn, imWrk)
    sub(imIn, imWrk, imOut)

# Reading the initial image.
imIn = imageMb('retina.png')

# Working image and result image.
imWrk = imageMb(imIn)
aneurisms = imageMb(imIn, 1)

# Extraction of aneurisms.
buildSupWhiteTopHat(imIn, imWrk, 10)
# Thresholding of the image (the value of the threshold is half the maximum
# grey value in the top-hat image.
```

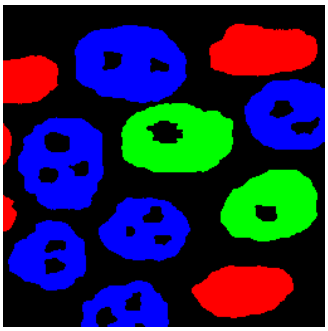
```
t = computeRange(imWrk)[1]/2
threshold(imWrk, aneurisms, t, 255)

# Superposing the result to the original image and saving the result.
multiSuperpose(imIn, aneurisms)
name = mambaDisplay.tagOneColorPalette(255, (255,0,0))
imIn.save('aneurisms.png', palette=mambaDisplay.getPalette(name))
```

3.18 Sorting particles with different numbers of holes

This example uses different morphological operators (geodesic reconstruction, SKIZ) to separate particles with no hole (simply connected particles) from particles with one hole then with more than one hole. More sophisticated procedures exist to sort any particle according to its number of holes. But these procedures are out of the scope of this example.



gruyere.png



sorted_particles.png

Here is the source code for this example (can be found in Sorting particles with different numbers of holes):

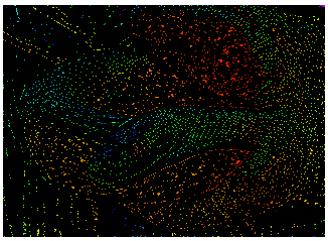
```
# Importing the mamba modules
from mamba import *
from mambaDisplay.extra import *

# Reading the initial image
imIn = imageMb('gruyere.png')
imIn.convert(1)
imWrk1 = imageMb(imIn)
imWrk2 = imageMb(imIn)
imWrk3 = imageMb(imIn)

# Particles without holes are sorted. Holes are extracted and they are used
# (after an elementary dilation) as markers for building particles which
# contain at least one hole.
closeHoles(imIn, imWrk1)
diff(imWrk1, imIn, imWrk1)
dilate(imWrk1, imWrk1)
build(imIn, imWrk1)
# Now, imWrk1 contains only particles with hole(s)

# Separating now particles with a single hole from particles with more
# than one hole.
negate(imWrk1, imWrk2)
fastSKIZ(imWrk2, imWrk2)
negate(imWrk2, imWrk2)
# Particles with several holes are marked by multiple points in the SKIZ.
multiplePoints(imWrk2, imWrk2)
build(imWrk1, imWrk2)
```

```
# The results can be seen by the superposer.
diff(imIn, imWrk1, imWrk3)
logic(imWrk2, imWrk3, imWrk3, "sup")
superpose(imWrk1, imWrk3)
```

3.19 Valued minima of a greyscale image

This example is a small variation of the minima function found in the mamba extrema module. It shows that there are various possibilities to attain the same result in Mamba.

snake.png



snake_valmin.png

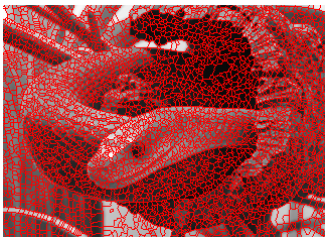
Here is the source code for this example (can be found in Valued minima of greyscale image):

```
# Importing mamba
from mamba import *
import mambaDisplay

def valuedMinima(imIn, imOut, h=1, grid=DEFAULT_GRID):
    """
    Computes the valued minima of 'imIn' using a dual build operation and
    puts the result in 'imOut'.

    'h' can be used to define the minima depth. Grid used by the dual build
    operation can be specified by 'grid'.

    Only works with greyscale images as input and output.
    """

    imWrk = imageMb(imIn)
    imWrk_bin = imageMb(imIn, 1)
    copy(imIn, imWrk)
    addConst(imWrk, h, imWrk)
    hierarDualBuild(imIn, imWrk, grid=grid)
    sub(imWrk, imIn, imWrk)
    threshold(imWrk, imWrk_bin, 1, computeMaxRange(imIn)[1])
    mul(imWrk_bin, imIn, imOut)

# Opening and creating images
im1 = imageMb('snake.png')
im2 = imageMb(im1)

# Computing the valued minima
valuedMinima(im1, im2)

im2.save("snake_valmin.png", palette=mambaDisplay.getPalette("rainbow"))
```

3.20 Watershed segmentation using gradient minima as markers

In this example, the watershed line is computed on the gradient image using its minima as a marker image. The resulting image is then superposed to the original to show the possibilities offered by some of the palette extra function.

snake.png



snake_watmin.png

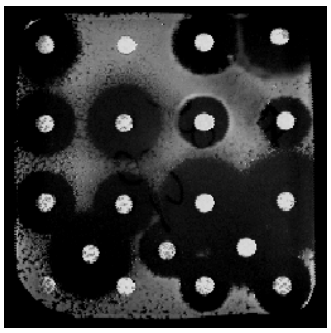
Here is the source code for this example (can be found in Watershed segmentation using gradient minima as markers):

```
# Importing the mamba module
from mamba import *
import mambaDisplay

def watershedByGradMinima(imIn, imOut, grid=DEFAULT_GRID):
    """
    Computes the watershed line of imIn using its gradient minima as a marker
    image
    """
    grad = imageMb(imIn, 8)
    marker = imageMb(imIn, 32)
    minbin = imageMb(imIn, 1)
    gradient(imIn, grad)
    minima(grad, minbin, grid=grid)
    label(minbin, marker, grid=grid)
    watershedSegment(grad, marker, grid=grid)
    copyBytePlane(marker, 3, imOut)

# Opening and creating images
im1 = imageMb('snake.png')
im2 = imageMb(im1)
im3 = imageMb(im1)

# Computing the watershed by minima marker
watershedByGradMinima(im1, im2)

# For a better view the watershed line is superposed on the original image
subConst(im1, 1, im3)
# The watershed line is set to 255 and the basins are empty in im2.
logic(im3, im2, im3, "sup")
# a special palette is created
name = mambaDisplay.tagOneColorPalette(255, (255,0,0))
im3.save("snake_watmin.png", palette=mambaDisplay.getPalette(name))
```

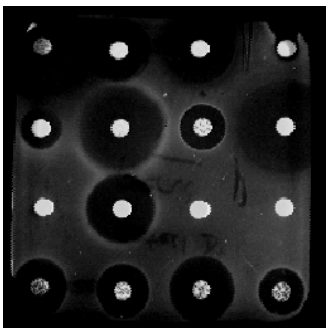
4 Advanced

4.1 Analysis of antibiograms

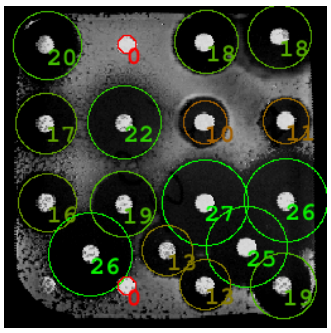
Antibiograms are a methodology used to determine the efficiency of various antibiotics against a specific bacteria. In a Petri dish where the bacteria has been cultivated, some discs containing antibiotics are scattered on. After a few days, some halos appear around the discs. The halo radius is a direct measurement of the antibiotics efficiency against the bacteria. The purpose of this example is to show how the radius of each halo can be determined automatically.



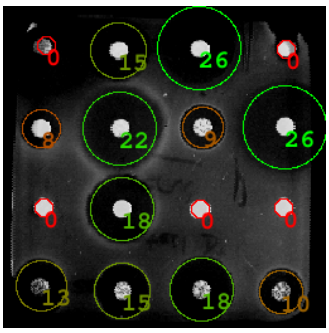
antibio1.png



antibio2.png



anti1_result.png



anti2_result.png

Here is the source code for this example (can be found in Analysis of antibiograms):

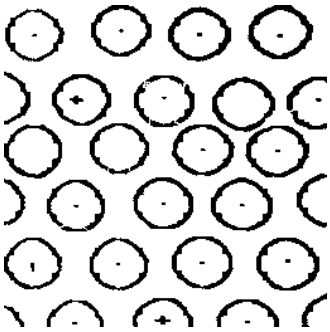
```
# Importing mamba
from mamba import *
from PIL import ImageDraw
from PIL import ImageFont

# The antibiotics are incorporated to the Petri dish using discs
# which appear with a certain size on our images
DISC_SIZE = 7

def drawingResults(imIn, results, path):
    """
    This function draws the result of the antibiograms measurement.
    """
    # First looking for the best result
    max_value = 0
    for (x,y,value) in results:
        max_value = max(max_value, value)
    # Now drawing the result
    pilim = Mamba2PIL(imIn).convert("RGB")
    draw = ImageDraw.Draw(pilim)
    font = ImageFont.truetype("FreeMonoBold.ttf", 20)
    for (x,y,value) in results:
        width, height = font.getsize(str(value))
        red = (255*(max_value-value))/max_value
        green = (255*value)/max_value
```

```
            color = "rgb(%d,%d,0)" % (red,green)
            r = value+DISC_SIZE
            draw.ellipse((x-r,y-r,x+r,y+r), outline=color)
            draw.text((x, y), str(value), color, font=font)
        pilim.save(path)

# Defining the operator which detects the antibiotics discs and measures the
# halos in the antibiograms images.

def checkAntibiograms(imIn, discCenters, halosRadii):
    """
    This procedure extracts the centers of the antibiotics discs scattered
    in image 'imIn' and puts them in 'discCenters' binary image. The radius
    of the corresponding halo is put in 'halosRadii' 32-bit image (the radii
    could be larger than 255).
    """
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)
    imWrk4 = imageMb(imIn, 1)
    imWrk5 = imageMb(imIn, 1)
    imWrk6 = imageMb(imIn, 32)

    # We work on a filtered image.
    alternateFilter(imIn, imWrk1, 3, openFirst=False)
    # The discs are removed by erosion (the size of the discs is constant and known).
    erode(imWrk1, imWrk2, DISC_SIZE)
    # A reconstruction gives the grey level of the unaltered microbe culture.
    imWrk3.fill(255)
    build(imWrk3, imWrk2)
    t = computeRange(imWrk2)[1]
    # This intermediary background is subtracted from the initial filtered image.
    sub(imWrk1, imWrk2, imWrk3)
    # The result is thresholded (a threshold equal to 30 is a good value to insure
    # that the noise in the image will not be taken into account).
    threshold(imWrk3, imWrk4, 30, 255)
    # The centers of the discs are built.
    thinD(imWrk4, discCenters)
    # The center points are stored in a 8-bit image.
    convert(discCenters, imWrk2)
    # Building the unaltered part of the antibiogram.
    dilate(imWrk2, imWrk3, 10)
    sub(imWrk1, imWrk3, imWrk3)
    build(imWrk1, imWrk3)
    # This unaltered microbe culture image is thresholded. The
    # threshold value is half its grey value. We obtain the halos.
    threshold(imWrk3, imWrk5, 0, t/2)
    # the distance function of this set is computed.
    computeDistance(imWrk5, imWrk6)
    # The center points are stored in a 32-bit image.
    for i in range(4):
        copyBytePlane(imWrk2, i, halosRadii)
    # They are valued with the radius of their corresponding halo (the size
    # of the disc is subtracted).
    logic(halosRadii, imWrk6, halosRadii, "inf")
    floorSubConst(halosRadii, DISC_SIZE, halosRadii)

# Reading the initial image.
im1 = imageMb('antibio1.png')
# Defining working images.
imWrk1 = imageMb(im1, 32)
imWrk2 = imageMb(im1, 1)
imWrk3 = imageMb(im1, 1)
```

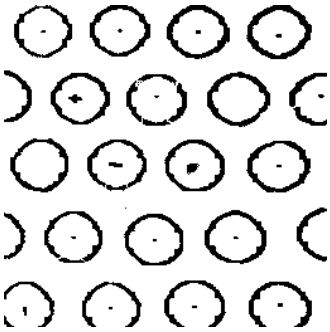
```
# Processing the image.
checkAntibiograms(im1, imWrk3, imWrk1)
# Each center and its corresponding radius are extracted.
# The radius value is printed in the result image.
results = []
while computeVolume(imWrk3) != 0:
    imWrk2.reset()
    x, y = compare(imWrk3, imWrk2, imWrk2)
    value = imWrk1.getPixel((x, y))
    diff(imWrk3, imWrk2, imWrk3)
    results.append((x,y,value))
# The result is saved.
drawingResults(im1, results, 'anti1_result.png')

# Processing the second image.
im1 = imageMb('antibio2.png')
checkAntibiograms(im1, imWrk3, imWrk1)
results = []
while computeVolume(imWrk3) != 0:
    imWrk2.reset()
    x, y = compare(imWrk3, imWrk2, imWrk2)
    value = imWrk1.getPixel((x, y))
    diff(imWrk3, imWrk2, imWrk3)
    results.append((x,y,value))
# Saving the result.
drawingResults(im1, results, 'anti2_result.png')
```

4.2 Analysis of the distribution of boron fibers

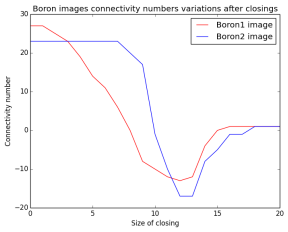
This example illustrates the judicious use of connectivity measures for solving a problem of quality control. The images represent boron fibers in a composite material. These fibers reinforce the mechanical resistance of the material. The resistance increases in proportion of the evenness of the fibers 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. This example illustrates the use of the matplotlib library with Mamba (to run this example, you must install this library).



boron1.png



boron2.png



fibers_connectivity.png

Here is the source code for this example (can be found in Analysis of the distribution of boron fibers):

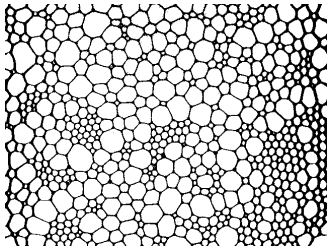
```
# Importing the mamba module
from mamba import *

# Importing the plot function from matplotlib
import matplotlib.pyplot as plot

def closeFibers(imIn, imOut):
    """
    This operation cleans the original fibers image by closing their contours
    and by closing their interiors. Note that this operation also fill 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 ans 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

# The whole procedure is applied on two different images
# Warning! this example works only on the hexagonal grid.
setDefaultGrid(HEXAGONAL)
# Opening and creating the first image
im1 = imageMb('boron1.png')
imtemp = imageMb(im1, 1)
im1.convert(1)
# Closing holes in the fibers.
closeFibers(im1, imtemp)
# Then, the connectivity numbers of the successive closings are computed
# and stored in a list.
maxSize = 20
ncList1 = checkEvenness(imtemp, maxSize)
# The same operation is performed on the second image
im2 = imageMb('boron2.png')
imtemp = imageMb(im2, 1)
im2.convert(1)
# Closing holes in the fibers
closeFibers(im2, imtemp)
# Then, the connectivity numbers of the successive closings are computed
# and stored in a list.
maxSize = 20
ncList2 = checkEvenness(imtemp, maxSize)
# Plotting the two curves with mathplotlib and showing them.
xs = range(maxSize + 1)
ys = ncList1
zs = ncList2
plot.xlabel('Size of closing')
plot.ylabel('Connectivity number')
plot.title('Boron images connectivity numbers variations after closings')
plot.plot(xs, ys, label='Boron1 image', color='red')
plot.plot(xs, zs, label='Boron2 image', color='blue')
plot.legend(loc='upper right')
# The slope of the curves is an indicator of the regularity of the arrangement.
```

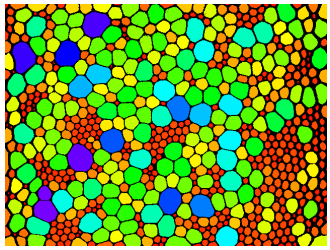
```
# A perfectly regular arrangement would exhibit a sudden transition between
# positive connectivity numbers and negative ones. The transition for the
# boron2 image is steeper than for the boron1 image as the arrangement of fibers
# is more regular in this latter image.
plot.savefig('fibers_connectivity.png')
```

4.3 Area labelling and area opening

This example explains how particles can be labelled by their area with a fast algorithm which does not use a particle analysis. It uses look-up tables to achieve this. Getting instantaneously all the particles with an area lower than a given value is made by a simple threshold of the result image. This operator allows to obtain area openings of the initial image. This operator is applied to an image of bubbles in Champagne wine. Initial image, courtesy of 0. Lordereau, Dept of Physics, Rennes University).



binary_foam.png



area_labelling.png

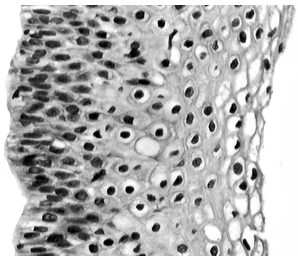
Here is the source code for this example (can be found in Area labelling and area opening):

```
# Importing mamba.
from mamba import *
import mambaDisplay

# The area labelling procedure is defined.
def areaLabelling(imIn, imOut):
    """
    Labelling of each particle of the binary image 'imIn' with the value of its
    area. The result is put is the 32-bit image 'imOut'.
    """

    # Working images.
    imWrk1 = imageMb(imIn, 32)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn, 8)
    imWrk4 = imageMb(imIn, 8)
    imWrk5 = imageMb(imIn, 8)
    imWrk6 = imageMb(imIn, 32)

    # Output image is emptied.
    imOut.reset()
    # Labelling the initial image.
    nbParticles = label(imIn, imWrk1)
    # Defining output LUTs.
    outLuts = [[0 for i in range(256)] for i in range(4)]
    while nbParticles > 0:
        # particles with labels between 1 and 255 are extracted.
        threshold(imWrk1, imWrk2, 0, 255)
        convert(imWrk2, imWrk3)
        copyBytePlane(imWrk1, 0, imWrk4)
        logic(imWrk3, imWrk4, imWrk3, "inf")
        # The histogram is computed.
```

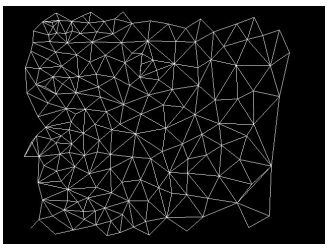
```
            histo = getHistogram(imWrk3)
            # The same operation is performed for the 255 particles.
            for i in range(1, 256):
                # The area of each particle is obtained from the histogram.
                value = histo[i]
                j = 3
                # The area value is splitted in powers of 256 and stored in the four
                # output LUTs.
                while j >= 0:
                    n = 2 ** (8 * j)
                    outLuts[j][i] = value / n
                    value = value % n
                    j -= 1
            # each LUT is used to label each byte plane of a temporary image with the
            # corresponding value.
            for i in range(4):
                lookup(imWrk3, imWrk5, outLuts[i])
                copyBytePlane(imWrk5, i, imWrk6)
            # The intermediary result is accumulated in the final image.
            logic(imOut, imWrk6, imOut, "sup")
            # 255 is subtracted from the initial labelled image in order to process
            # the next 255 particles.
            floorSubConst(imWrk1, 255, imWrk1)
            nbParticles -= 255

# This operator is applied to an image of bubbles.
# Reading and converting the initial image.
imA = imageMb('binary_foam.png')
imA.convert(1)

# Result image.
imB = imageMb(imA, 32)

# Labelling by area.
areaLabelling(imA, imB)

# Normalizing the final result in the range (0, 255)
# to be able to display it in a greyscale image (it's
# just a trick..).
imC = imageMb(imA, 32)
imD = imageMb(imA, 8)
# The maximum area is 2067 (determined by computeRange).
# So, a division by 9 is sufficient.
divConst(imB, 9, imC)
copyBytePlane(imC, 0, imD)
# Saving the result.
imD.save('area_labelling.png', palette=mambaDisplay.getPalette("rainbow"))
```

4.4 Automated measurement in medical field using image analysis

This example demonstrates how mathematical morphology and Mamba can be used to perform relevant automated measurements on medical images. This example implements part of the algorithm described in paper http://www.diagnosticpathology.org/content/5/1/7. Courtesy of Michel Jondet.



medical.png



delaunay.jpg

Here is the source code for this example (can be found in Automated measurement in medical field using image analysis):

```
# Importing mamba and associates
from mamba import *

# Reusing functions from previous examples
def firstParticle(imIn, imOut, grid=mamba.DEFAULT_GRID):
    imWrk = mamba.imageMb(imIn)
    mamba.compare(imIn, imWrk, imWrk)
    build(imIn, imWrk, grid=grid)
    mamba.diff(imIn, imWrk, imIn)
    mamba.copy(imWrk, imOut)

def autoThreshold(imIn, imOut):
    grad = imageMb(imIn)
    wrk = imageMb(imIn)
    level = imageMb(imIn, 1)
    gradient(imIn, grad)
    histo = getHistogram(imIn)
    distri = []
    for i in range(256):
        if histo[i]!=0:
            threshold(imIn, level, i, i)
            mul(level, grad, wrk)
            vol = computeVolume(wrk)/histo[i]
            distri.append(vol)
        else:
            distri.append(0)
    sd = sum(distri)
    sr = distri[0]
    threshval = 0
    while(sr<(sd/2)):
        threshval += 1
        sr += distri[threshval]
    threshold(imIn, imOut, threshval, 255)
```

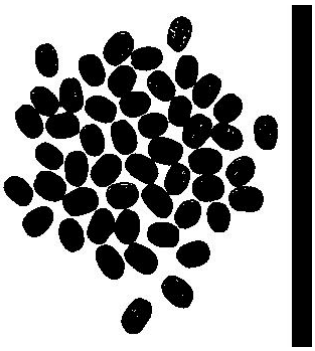
```
    return threshval

def segmentGrains2(imIn, imOut):
    imMarker = imageMb(imIn, 1)
    imDist = imageMb(imIn, 32)
    imDist8 = imageMb(imIn, 8)
    imWTS32 = imageMb(imIn, 32)
    imWTS = imageMb(imIn, 8)
    computeDistance(imIn, imDist, edge=FILLED)
    computeRange(imDist)
    copyBytePlane(imDist, 0, imDist8)
    negate(imDist8, imDist8)
    minima(imDist8, imMarker)
    dilate(imMarker, imMarker, 2)
    nb = label(imMarker, imWTS32)
    watershedSegment(imDist8, imWTS32)
    copyBytePlane(imWTS32, 3, imWTS)
    negate(imWTS, imWTS)
    copyBytePlane(imWTS32, 0, imOut)
    logic(imOut, imWTS, imOut, "inf")
    convert(imIn, imWTS)
    logic(imOut, imWTS, imOut, "inf")
    return nb

def drawDelaunay(imIn, imOut):
    """
    Draws Delaunay triangles using neighbors particles.
    Returns the ridges length in a list.
    """
    imZero = imageMb(imIn)
    imZero.reset()
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn)
    imWrk4 = imageMb(imIn)

    list_ridges = []
    copy(imIn, imWrk1)
    while computeVolume(imWrk1) != 0:
        firstParticle(imWrk1, imWrk2)
        dilate(imWrk2, imWrk3, 2)
        build(imWrk1,imWrk3)
        logic(imWrk3, imWrk2, imWrk3, "sup")
        thinD(imWrk3, imWrk4)
        negate(imWrk2, imWrk2)
        diff(imWrk4,imWrk2,imWrk2)
        diff(imWrk4,imWrk2,imWrk3)
        (x1,y1) = compare(imWrk2, imZero, imZero)
        imZero.reset()
        while computeVolume(imWrk3) != 0:
            firstParticle(imWrk3, imWrk4)
            (x2,y2) = compare(imWrk4, imZero, imZero)
            imZero.reset()
            drawLine(imWrk4,(x1,y1,x2,y2),1)
            a = computeVolume(imWrk4)
            add(imWrk4, imOut, imOut)
            list_ridges.append(a)

    return list_ridges


im = imageMb("medical.png")

nuclei = imageMb(im, 1)
```

```
skiz = imageMb(im, 1)
voronoi = imageMb(im, 1)
delaunay = imageMb(im, 1)

imbin1 = imageMb(im, 1)
imbin2 = imageMb(im, 1)
imbin3 = imageMb(im, 1)
im1 = imageMb(im, 8)
im2 = imageMb(im, 8)

# Finding the nuclei
##########################################################################################
autoThreshold(im, imbin1)
# Instead of autoThreshold you could ask the user to select the best threshold
# image (making sure the nuclei are in black) .
# Using his keyboard (with w,q,x and s) to adjust the threshold value to its
# liking
#threshold(im,imbin1,*dynamicThreshold(im))
negate(imbin1, imbin1)
removeEdgeParticles(imbin1, imbin1) # border artefact removal
closeHoles(imbin1, imbin1) # Closes holes in nuclei
opening(imbin1, imbin1, 2) # Removes small artefacts
# Segmentations : returns the number of nuclei
nb_nuclei = segmentGrains2(imbin1, im1)
threshold(im1, imbin1, 1, 255)
copy(imbin1, nuclei)
nuclei.show()
#nuclei.save("nuclei_segmentation.jpg")

# Building the voronoi tesselation, skiz and geodesic center
##########################################################################################
drawEdge(imbin2)
add(imbin2, imbin1, imbin1)
fastSKIZ(imbin1, imbin2)
removeEdgeParticles(imbin2, imbin2)
copy(imbin2, skiz)
skiz.show()
#skiz.save("skiz.jpg")
thinD(imbin2,imbin3)
sub(imbin2, imbin3, imbin2)
copy(imbin2, voronoi)
voronoi.show()
#voronoi.save("voronoi.jpg")

# Delaunay triangulation
##########################################################################################
edges = drawDelaunay(skiz, delaunay)
delaunay.show()
delaunay.save("delaunay.jpg")

# Reporting all the measurements
##########################################################################################
print("number of nuclei %d" % (nb_nuclei))
print("nuclei mean area %d" % (computeVolume(nuclei)/nb_nuclei))
print("cytoplasmic mean area %d" % (computeVolume(voronoi)/nb_nuclei))
print("nucleo-cytoplasmic ratio %d" % (computeVolume(voronoi)/computeVolume(nuclei)))
print("mean edge length %d" % (sum(edges)/len(edges)))
```

4.5 Automatic thresholding using gradient

This example shows how to compute an automatic threshold value using the gradient, histogram and volume computations.

coffee_grains.jpg



threshold_coffee_grains.jpg

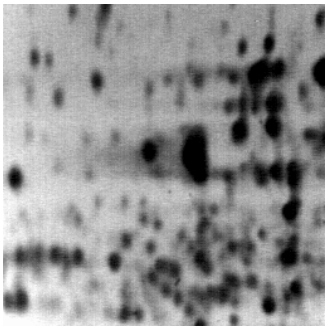
Here is the source code for this example (can be found in Automatic thresholding using gradient):

```
# Importing mamba
from mamba import *

def autoThreshold(imIn, imOut):
    """
    Computes an automatic threshold image using the gradient.
    This function works well with greyscale images displaying two
    highly contrasted sets.
    It produces a binary image that sort of *segment* the two sets in two.
    """

    grad = imageMb(imIn)
    wrk = imageMb(imIn)
    level = imageMb(imIn, 1)
    # First the gradient is computed
    gradient(imIn, grad)

    # Then the histogram
    histo = getHistogram(imIn)

    distri = []
    for i in range(256):
        # First no point at looking at a particular value if there is no
        # pixel in it.
        if histo[i]!=0:

            # for each each possible pixel value, we extract the pixels
            # in imIn with that value
```

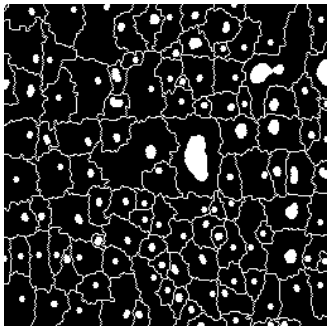
```
                threshold(imIn, level, i, i)
                # then we compute the volume of their corresponding pixels
                # in the gradient image (normalised by the number of
                # pixels)
                mul(level, grad, wrk)
                vol = computeVolume(wrk)/histo[i]
                # The volume is added to a distribution function
                distri.append(vol)
            else:
                distri.append(0)

        # Finding the median of the distribution
        sd = sum(distri)
        sr = distri[0]
        threshval = 0
        while(sr<(sd/2)):
            threshval += 1
            sr += distri[threshval]

        # Final computation
        threshold(imIn, imOut, threshval, 255)

        return threshval

im = imageMb("coffee_grains.jpg")
imbin = imageMb(im, 1)

# Computing the automatic threshold image
autoThreshold(im, imbin)
# And saving it
# The resulting image has a black band on the right because
# Mamba padded the original image so its width is a multiple of 64
# for performance reasons. This has a no impact on the quality of the result.
# Also notice that the result is not a very good segmentation as different
# coffee grains are connected in a unique set.
imbin.save("threshold_coffee_grains.jpg")
```

4.6 Blobs segmentation in an electrophoresis gel

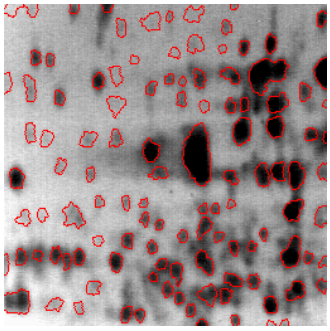
This (historical) example is one of the first applications of the marker- controlled watershed segmentation. The initial image is a two-dimensional electrophoresis gel. This technique allows to separate and identify proteins. This example illustrates the different steps of the blobs segmentation.



electrop.png



finalMarkers.png



blobsContours.png

Here is the source code for this example (can be found in Blobs segmentation in an electrophoresis gel):

```
# Importing mamba
from mamba import *
import mambaDisplay

# Reading the initial image.
imIn = imageMb('electrop.png')

# Defining working images.
imWrk1 = imageMb(imIn)
blobsMarkers = imageMb(imIn, 1)
backgroundMarker= imageMb(imIn, 1)
imWrk2 = imageMb(imIn, 1)
finalMarkers = imageMb(imIn, 1)
blobsContours = imageMb(imIn, 1)

# The initial image is filtered with an alternate filter of size 2.
alternateFilter(imIn, imWrk1, 2, True)

# The minima of this filtered image can be used as markers of the blobs.
minima(imWrk1, blobsMarkers)

# Then, we must generate a background marker. This marker can be obtained by
# a marker-controlled watershed of the initial image.
markerControlledWatershed(imIn, blobsMarkers, imWrk1)
threshold(imWrk1, backgroundMarker, 1, 255)

# The two sets of markers are merged. We must however insure that they are
# separated by at least one pixel.
dilate(backgroundMarker, imWrk2)
```

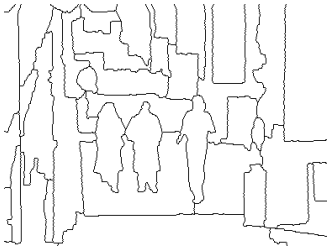
```
diff(blobsMarkers, imWrk2, blobsMarkers)
logic(blobsMarkers, backgroundMarker, finalMarkers, "sup")
finalMarkers.save('finalMarkers.png')
# The contours of the blobs are obtained by a marker-controlled watershed of
# the gradient image.
gradient(imIn, imWrk1)
markerControlledWatershed(imWrk1, finalMarkers, imWrk1)
threshold(imWrk1, blobsContours, 1, 255)

# Superposing the result to the original image and saving the result.
multiSuperpose(imIn, blobsContours)
name = mambaDisplay.tagOneColorPalette(255,(255,0,0))
imIn.save('blobsContours.png', palette=mambaDisplay.getPalette(name))
```

4.7 Color image segmentation with quasi-distance of the gradient

This example illustrates the use of the quasi-distance operator applied on a color gradient to segment a scene. It also uses the Python Imaging Library (PIL) and the Mamba split operator to extract the three channels of a color image.

The gallery image is coming from the EC Funded CAVIAR (Context Aware Vision using Image-based Active Recognition) project/IST 2001 37540, found at URL: http://homepages.inf.ed.ac.uk/rbf/CAVIAR/.

gallery.png



binary_segmentation.png

segmented_gallery.png

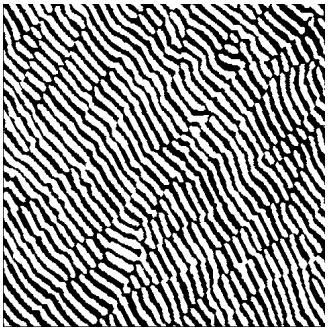
Here is the source code for this example (can be found in Color image segmentation with quasidistance of gradient):

```
# Importing PIL Image module and mamba
from PIL import Image
from mamba import *

# Opening the image (PIL format) and splitting it into three color channels
pilim = Image.open('gallery.png')
imRed = imageMb(pilim.size[0], pilim.size[1], 8)
imGreen = imageMb(imRed)
imBlue = imageMb(imRed)
split(pilim, imRed, imGreen, imBlue)

# We will perform a thick gradient on each color channel (contours in original
# picture are more or less fuzzy) and we add all these gradients
gradIm = imageMb(imRed)
imWrk1 = imageMb(imRed)
gradIm.reset()
gradient(imRed, imWrk1, 2)
add(imWrk1, gradIm, gradIm)
gradient(imGreen, imWrk1, 2)
add(imWrk1, gradIm, gradIm)
gradient(imBlue, imWrk1, 2)
add(imWrk1, gradIm, gradIm)

# Then we invert the gradient image and we compute its quasi-distance
qDist = imageMb(gradIm, 32)
negate(gradIm, gradIm)
quasiDistance(gradIm, imWrk1, qDist)

# The maxima of the quasi-distance are extracted and filtered (too close maxima,
# less than 6 pixels apart, are merged)
imWrk2 = imageMb(imRed)
```

```
imMark = imageMb(gradIm, 1)
copyBytePlane(qDist, 0, imWrk1)
subConst(imWrk1, 3, imWrk2)
build(imWrk1, imWrk2)
maxima(imWrk2, imMark)

# The marker-controlled watershed of the gradient is performed
imWts = imageMb(gradIm)
label(imMark, qDist)
negate(gradIm, gradIm)
watershedSegment(gradIm, qDist)
copyBytePlane(qDist, 3, imWts)

# The segmented binary and color image are stored
logic(imRed, imWts, imRed, "sup")
logic(imGreen, imWts, imGreen, "sup")
logic(imBlue, imWts, imBlue, "sup")
pilim = mix(imRed, imGreen, imBlue)
pilim.save('segmented_gallery.png')
negate(imWts, imWts)
imWts.save('binary_segmentation.png')
```

4.8 Criterion opening, Feret diameter (bounding boxes) example

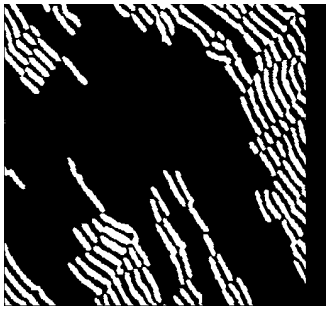
This example introduces another criterion opening: the bounding boxes opening. The feretDiameterOpening operator allows to label each connected component of a binary set with its Feret horizontal or vertical diameter. Then, sorting each component included in a bounding box of a given size is performed by a simple thresholding of the resulting images.



binary_eutectic.png

H_Feret_label.png

V_Feret_label.png



bbox_filter.png

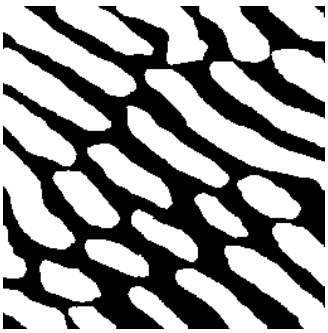
Here is the source code for this example (can be found in Criterion opening Feret diameter example):

```
# Importing the mamba module
from mamba import *
import mambaDisplay

def feretDiameterOpening(imIn, imOut, direc):
    """
    The Feret diameter of each connected component of the binary image 'imIn'
    is computed and its value labels the corresponding component. The labelled
    image is stored in the 32-bit image 'imOut'.
    If 'direc' is 'vertical', the vertical Feret diameter is computed. If it is
    set to 'horizontal', the corresponding diameter is used.
    """

    imWrk1 = imageMb(imIn, 1)
    imWrk2 = imageMb(imIn, 32)
    imWrk3 = imageMb(imIn, 32)
    imWrk4 = imageMb(imIn, 32)

    imWrk1.fill(1)
    if direc == "horizontal":
        dir = 7
    elif direc == "vertical":
        dir = 1
    else:
        dir = -1
        # The above statement generates an error ('direc' is not horizontal or
        # vertical.
    # An horizontal or vertical distance function is generated.
    linearErode(imWrk1, imWrk1, dir, grid=SQUARE, edge=EMPTY)
```

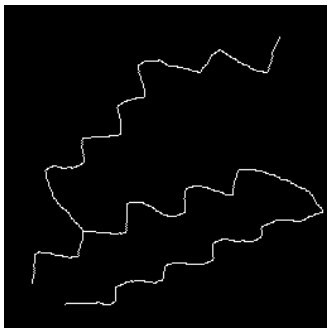
```
        computeDistance(imWrk1, imOut, grid=SQUARE, edge=FILLED)
        addConst(imOut, 1, imOut)
        # Each particle is valued with the distance.
        convertByMask(imIn, imWrk2, 0, computeMaxRange(imWrk3)[1])
        logic(imOut, imWrk2, imWrk3, "inf")
        # The valued image is preserved.
        copy(imWrk3, imWrk4)
        # Each component is labelled by the maximal coordinate.
        build(imWrk2, imWrk3)
        # Using the dual reconstruction, we label the particles with the
        # minimal ccordinate.
        negate(imWrk2, imWrk2)
        logic(imWrk2, imWrk4, imWrk4, "sup")
        dualBuild(imWrk2, imWrk4)
        # We subtract 1 because the selected coordinate must be outside the particle.
        subConst(imWrk4, 1, imWrk4)
        negate(imWrk2, imWrk2)
        logic(imWrk2, imWrk4, imWrk4, "inf")
        # Then, the subtraction gives the Feret diameter.
        sub(imWrk3, imWrk4, imOut)

# Reading the initial image.
im1 = imageMb('binary_eutectic.png', 1)
im2 = imageMb(im1, 32)
im3 = imageMb(im1, 32)
# Labelling the connected components with the horizontal Feret diameter.
feretDiameterOpening(im1, im2, "horizontal")
# Same operation with the vertical Feret diameter.
feretDiameterOpening(im1, im3, "vertical")
# Coloring the results for a better view.
# Saving the results.
im2.save('H_Feret_label.png', palette=mambaDisplay.getPalette("rainbow"))
im3.save('V_Feret_label.png', palette=mambaDisplay.getPalette("rainbow"))
# Example of extraction of the connected components included in a square box
# of size 100.
temp1 = imageMb(im1)
temp2 = imageMb(im1)
threshold(im2, temp1, 1, 100)
threshold(im3, temp2, 1, 100)
logic(temp1, temp2, temp1, "inf")
# Saving the result.
temp1.save('bbox_filter.png')
```

4.9 Dislocations in an eutectic lamellar alloy

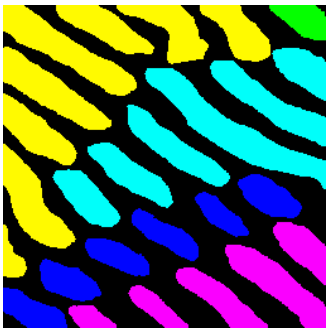
The problem addressed here is a problem of detection of dislocations in an eutectic lamellar alloy. This structure is characterised by a two-phase lamination with some discontinuities due to dislocations in the cristalline assembly. These dislocations induce loss of strength which is all the more important as their lengths are important. Moreover, the size of the cells delimited by these dislocations is also important. This example shows that it is possible to build, by an appropriate sequence of operations, features which are not visible at the first sight.



eutectic.png



eutectic_disloc.png



eutectic_cells.png

Here is the source code for this example (can be found in Dislocations in an eutectic lamellar alloy):

```
# Importing mamba
from mamba import *
import mambaDisplay

# The extremities extractor designed in exampleM14 will also be necessary.
# It is duplicated below.

def extremities(imIn, imOut, innerParticles=False, grid=DEFAULT_GRID):
    """
    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 and 'imOut' 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).
    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)
    imWrk2 = imageMb(imIn, 32)

    # Computation of the centroids.
    thinD(imIn, imWrk1, grid=grid)
    # These centroids are removed from the initial image (each particle contains
```

```
    # a hole.
    diff(imIn, imWrk1, imWrk1)
    # Computing the geodesic distance (the elementary structuring element is
    # determined by the grid in use).
    se = structuringElement(getDirections(grid), grid)
    geodesicDistance(imWrk1, imIn, imOut, se=se)
    # The extremities correspond to the maxima.
    maxima(imOut, imWrk1, grid=grid)
    # Extremities on the edge are removed if 'innerParticles' is set to False.
    if not(innerParticles):
        removeEdgeParticles(imWrk1, imWrk1, grid=grid)
    # The extremities are given their corresponding distance to the center.
    convert(imWrk1, imWrk2)
    logic(imOut, imWrk2, imOut, "inf")

# To solve this problem, we need to define two new functions.
def geodesicAdaptiveDilate(imIn, imMask, imOut):
    """
    This operator performs a binary adaptive dilation. The greyscale 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)
        subConst(imWrk3, 1, imWrk3)
        logic(imWrk2, imWrk3, imWrk2, "sup")
        v2 = computeVolume(imWrk2)
    threshold(imWrk2, imOut, 1, 255)

def extremePoints(imIn, imOut):
    """
    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 result is put in 'imOut'.
    Note that this operator needs that the connected components be inside the
    image and not touching the edges (a more refine operation could be designed
    but it is useless in this case).
    """

    imWrk1 = imageMb(imIn, 32)
    imWrk2 = imageMb(imIn, 8)
    imWrk3 = imageMb(imIn, 8)
    imWrk4 = imageMb(imIn, 8)
    imWrk5 = imageMb(imIn)
    imWrk6 = imageMb(imIn)
    imWrk7 = imageMb(imIn)
    # Extraction of extremities of initial image.
    extremities(imIn, imWrk1)
    convert(imIn, imWrk2)
```

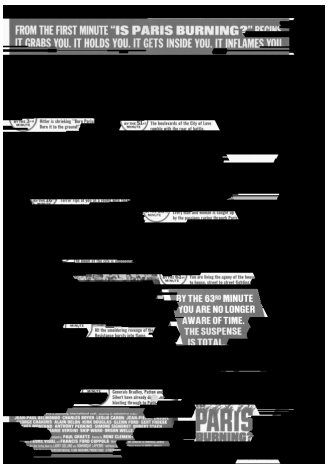
```
        # Valued extremities are copied in 8-bit images.
        copyBytePlane(imWrk1, 0, imWrk3)
        copy(imWrk3, imWrk4)
        # Each particle is valued with the maximum extremity (may be more than one).
        build(imWrk2, imWrk4)
        # Farthest extremity is extracted.
        generateSupMask(imWrk3, imWrk4, imWrk5, False)
        logic(imWrk5, imIn, imWrk5, "inf")
        convert(imWrk5, imWrk2)
        logic(imWrk3, imWrk2, imWrk2, "inf")
        # This extremity is dilated by a geodesic dilation of size equal to its
        # value.
        geodesicAdaptiveDilate(imWrk2, imIn, imWrk6)
        # determination of points which are not reached yet.
        diff(imIn, imWrk6, imWrk6)
        # Remaining extremities are selected.
        convert(imWrk6, imWrk2)
        logic(imWrk3, imWrk2, imWrk3, "inf")
        # They are copied in a binary image.
        threshold(imWrk3, imWrk7, 1, 255)
        copy(imWrk3, imWrk4)
        # The process is run again to extract the farthest extremity among those
        # which remain (it is possible that the result be empty).
        build(imWrk2, imWrk4)
        generateSupMask(imWrk3, imWrk4, imWrk6, False)
        logic(imWrk6, imWrk7, imWrk6, "inf")
        # This extremity is added to the previous sone(s).
        logic(imWrk5, imWrk6, imOut, "sup")

# Reading and converting the initial image.
im1 = imageMb('eutectic.png')
im1.convert(1)

# working images.
imWrk1 = imageMb(im1)
imWrk2 = imageMb(im1)
imWrk3 = imageMb(im1)
imWrk4 = imageMb(im1)
imWrk5 = imageMb(im1)
imWrk6 = imageMb(im1)
imWrk7 = imageMb(im1)
imWrk8 = imageMb(im1)
imWrk9 = imageMb(im1)
imWrk10 = imageMb(im1, 32)
imWrk11 = imageMb(im1, 8)
imWrk12 = imageMb(im1, 8)

# Performing the SKIZ of the lamellae.
fastSKIZ(im1, imWrk1)
# Extracting multiple points.
negate(imWrk1, imWrk2)
multiplePoints(imWrk2, imWrk3)
removeEdgeParticles(imWrk3, imWrk3)
# Separatng each arc of the boundaries.
diff(imWrk2, imWrk3, imWrk4)
# Arcs touching the edges are removed. They do not belong to dislocations.
removeEdgeParticles(imWrk4, imWrk5)
# Closed cells are rebuilt.
logic(imWrk5, imWrk3, imWrk2, "sup")
closeHoles(imWrk2, imWrk6)
diff(imWrk6, imWrk2, imWrk6)
# Isolated arcs (not surrounding cells) are kept separately.
dilate(imWrk6, imWrk2)
diff(imWrk5, imWrk2, imWrk7)
```

```
# Farthest extremities of these closed cells are extracted.
extremePoints(imWrk6, imWrk8)
# these extremities should mark arcs belonging to dislocations. These
# arcs are rebuilt.
dilate(imWrk8, imWrk2)
build(imWrk5, imWrk2)
# They are added to the isolated arcs. Multiple points are also introduced
# again.
logic(imWrk2, imWrk7, imWrk9, "sup")
logic(imWrk9, imWrk3, imWrk9, "sup")
# A small clipping allows to remove remaining barbs after the reintroduction
# of the multiple points.
whiteClip(imWrk9, imWrk9, step=1)
# The result image is saved.
imWrk9.save('eutectic_disloc.png')

# We can also build the cells containing stacked lamellae. This is done
# by adding boundaries connecting the edge and the end points of the previous
# dislocations image.
diff(imWrk4, imWrk5, imWrk5)
endPoints(imWrk9, imWrk2)
dilate(imWrk2, imWrk2, 2)
build(imWrk5, imWrk2)
logic(imWrk9, imWrk2, imWrk2, "sup")
# the lamellae belonging to the same cell are labelled and displayed in the
# same color.
negate(imWrk2, imWrk2)
erode(imWrk2, imWrk2, n=2, edge=EMPTY)
fastSKIZ(imWrk2, imWrk2)
nCells = label(imWrk2, imWrk10)
copyBytePlane(imWrk10, 0, imWrk11)
v = 256/nCells
mulConst(imWrk11, v, imWrk11)
convert(im1, imWrk12)
logic(imWrk11, imWrk12, imWrk11, "inf")
# The result image is saved (with different colors for the different cells).
# Note that the lamella at the upper right is ambiguous. Therefore, it has
# been given a different color.
imWrk11.save('eutectic_cells.png', palette=mambaDisplay.getPalette("rainbow"))
```

4.10 Embedded text filtering

Image indexation is quite a complex job. Being able to find text in an image may help categorize it. This example implements a very simple filtering algorithm that makes text stands out. If you are interested in the subject, visit http://cmm.ensmp.fr/ marcoteg/ImagEval.htm.

is-paris-burning.jpg



filter-text.png

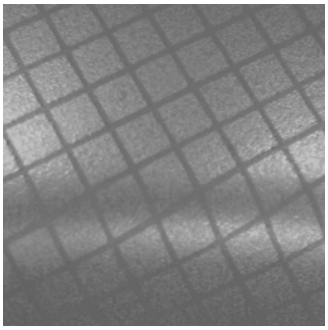
Here is the source code for this example (can be found in Embedded text filtering):

```
# Importing mamba and associates
from mamba import *

def textFiltering(imIn, imOut, textSize=16, textDir="HORZ"):
    """
    Filters image 'imIn' so that text comes out more brillant in 'imOut'.
    The text size inside the image must be identified with 'textSize', default
    value is 16. The direction of the text is given with 'textDir', possible
    values are "HORZ" for horizontal text or "VERT" for vertical text.
    """
    imWrk1 = imageMb(imIn)
    imWrk32 = imageMb(imIn, 32)
    imWrkb = imageMb(imIn, 1)

    if textDir=="VERT":
        dir1 = structuringElement([0,1], SQUARE)
    else:
        dir1 = structuringElement([0,3], SQUARE)
```

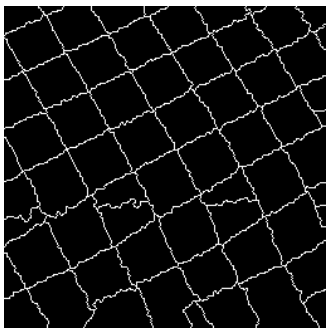
```
        ultimateOpening(imIn, imWrk1, imWrk32)
        whiteTopHat(imWrk32, imWrk32, textSize)
        closing(imWrk32, imWrk32, textSize*2, dir1)
        opening(imWrk32, imWrk32, textSize*10, dir1)
        closeHoles(imWrk32, imWrk32)
        threshold(imWrk32, imWrkb, 1, 0xffffffff)
        mul(imIn, imWrkb, imOut)

im = imageMb("is-paris-burning.jpg")
imText = imageMb(im)

textFiltering(im, imText, textSize=8)

imText.save("filter-text.png")
```

4.11 Extraction of a stamped grid on a steel sheet

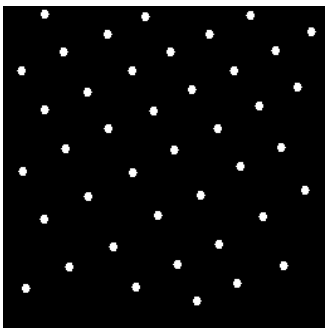
A regular grid has been printed on a steel sheet before its stamping. This example shows how the crossing points of this grid after stamping can be extracted. The new position of each point indicates its displacement during stamping and therefore the degree of stress exerted locally on the steel sheet.



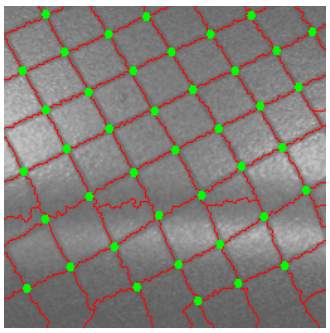
steel_sheet.png



grid_image.png



grid_crossings.png



grid_result.png

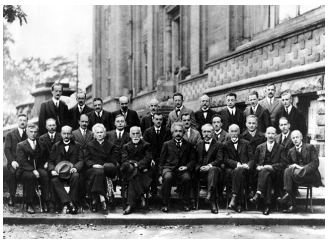
Here is the source code for this example (can be found in Extraction of stamped grid on steel sheet):

```
# Importing mamba
from mamba import *

# The markers of the grid cells are extracted by successive filterings.
def extractMarkers(imIn, imOut):
    """
    This procedure extracts markers of the cells of the printed grid in image
    'imIn' and puts the result in binary image 'imOut'.

    """
    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).
    strongLevelling(imIn, imWrk1, 2, eroFirst=False)
    maxima(imWrk1, imOut)
    # The same filtering is iterated for increasing sizes. Only the inner
    # maxima are preserved and added to the markers image.
    for i in range(4, 16, 2):
        strongLevelling(imIn, imWrk1, i, eroFirst=False)
        maxima(imWrk1, imWrk2)
        removeEdgeParticles(imWrk2, imWrk2)
        logic(imWrk2, imOut, imOut, "sup")

# Reading the image.
im1 = imageMb('steel_sheet.png')
# Defining working images.
imWrk1 = imageMb(im1, 1)
imWrk2 = imageMb(im1)
```

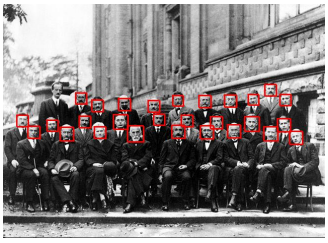
```
imWrk3 = imageMb(im1, 32)
imWrk4 = imageMb(im1, 1)
imWrk5 = imageMb(im1, 1)
imWrk6 = imageMb(im1, 1)
crossings = imageMb(im1, 1)

# Extracting the markers of the cells in the image.
extractMarkers(im1, imWrk1)
# Performing the marker-controlled watershed transform of the inverted
# initial image.
negate(im1, imWrk2)
label(imWrk1, imWrk3)
watershedSegment(imWrk2, imWrk3)
copyBytePlane(imWrk3, 3, imWrk2)
threshold(imWrk2, imWrk1, 1, 255)
# Saving the grid image.
imWrk1.save('grid_image.png')
# Crossing points extraction.
multiplePoints(imWrk1, imWrk4)
dilate(imWrk4, imWrk4, 2)
# The crossing points which are not at the intersection of 4 cells
# are removed. This procedure is applied separately on each dot.
negate(imWrk1, imWrk1)
while computeVolume(imWrk4) != 0:
    imWrk5.reset()
    p = compare(imWrk4, imWrk5, imWrk5)
    build(imWrk4, imWrk5)
    dilate(imWrk5, imWrk6)
    diff(imWrk6, imWrk5, imWrk6)
    logic(imWrk6, imWrk1, imWrk6, "inf")
    if computeConnectivityNumber(imWrk6) == 4:
        logic(imWrk5, crossings, crossings, "sup")
    diff(imWrk4, imWrk5, imWrk4)
# The remaining dots are reduced to a point and slightly dilated (for
# a better view).
thinD(crossings, crossings)
dilate(crossings, crossings, 3)
# The result is saved.
crossings.save('grid_crossings.png')

# Superposing the various results to the original image and saving the result.
negate(imWrk1, imWrk1)
multiSuperpose(im1, imWrk1, crossings)
pal = ()
for i in range(254):
    pal += (i,i,i)
pal += (255,0,0)
pal += (0,255,0)
im1.save('grid_result.png', palette=pal)
```

4.12 Face recognition : connecting Mamba and OpenCV

This example shows how to transform a Mamba image into an OpenCV image and reverse. This is used to apply a face recognition algorithm offered in OpenCV to our image. The picture used to try our algorithm was taken during the Solvay conference in 1927 (http://en.wikipedia.org/wiki/Solvay_Conference)



Solvay_conference_1927.jpg



faces_of_physic.jpg

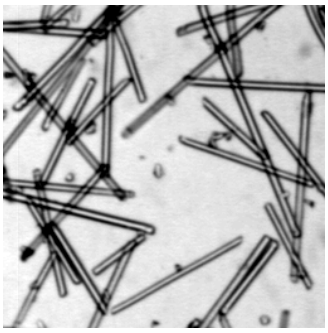
Here is the source code for this example (can be found in Face recognition connecting Mamba and OpenCV):

```
# Importing mamba
from mamba import *
import mambaDisplay

# Importing opencv.
import cv

# Defining the image conversions from mamba to openCv and the reverse.
def Mamba2OpenCV(imIn):
    """
    Creates an OpenCV image from the mamba image 'imIn'. The function only
    works with 8-bit images (other cases will return None).
    """
    if imIn.getDepth()!=8:
        return None

    (w, h) = imIn.getSize()
    cvImOut = cv.CreateImage((w,h), cv.IPL_DEPTH_8U, 1)
    cv.SetData(cvImOut, imIn.extractRaw())

    return cvImOut

def OpenCV2Mamba(cvImIn, imOut):
    """
    Loads the data contained in the openCV image 'cvImOut' into mamba image
    'imOut'. Images size and depth must be identical for the function to
    operate (the function will quit silently otherwise).
    """
    if cvImIn.depth!=cv.IPL_DEPTH_8U:
        return
    if imOut.getDepth()!=8:
        return
    if cv.GetSize(cvImIn)!=imOut.getSize():
        return
    imOut.loadRaw(cvImIn.tostring())
```

```
# Modify this line so that it points to your local installation of the
# cascade file
FACE_CASCADE_PATH = '/usr/share/OpenCV/haarcascades/haarcascade_frontalface_alt.xml'

def detectFaces(imIn):
    """
    This function detects faces inside Mamba image 'imIn' by first converting
    it into an OpenCV image and then applying the appropriate functions.
    The function returns a tuple containing the coordinates of the bounding
    box around all the faces in the image.
    """

    cvim = Mamba2OpenCV(imIn)
    size = cv.GetSize(cvim)

    # Preparing the image.
    storage = cv.CreateMemStorage(0)
    cv.EqualizeHist(cvim, cvim)
    # Detecting objects (faces).
    cascade = cv.Load(FACE_CASCADE_PATH)
    faces = cv.HaarDetectObjects(cvim,
                                 cascade,
                                 storage,
                                 1.1, 2,
                                 cv.CV_HAAR_DO_CANNY_PRUNING,
                                 (20, 20))

    faces_bb = ()
    for face in faces:
        face_coord = face[0]
        faces_bb += ((face_coord[0],
                      face_coord[1],
                      face_coord[0]+face_coord[2],
                      face_coord[1]+face_coord[3]),)

    return faces_bb

# Applying the operator on the Solvay conference image.
imIn = imageMb("Solvay_conference_1927.jpg")
imOut = imageMb(imIn)
imbin = imageMb(imIn, 1)
faces = detectFaces(imIn)
# Drawing a box around each detected face.
for f in faces:
    drawBox(imbin, f, 1)
    drawBox(imbin, map(lambda x : x+1, f), 1)
    drawBox(imbin, map(lambda x : x-1, f), 1)
copy(imIn, imOut)
multiSuperpose(imOut, imbin)
name = mambaDisplay.tagOneColorPalette(255, (255,0,0))
imOut.save("faces_of_physic.jpg", palette=mambaDisplay.getPalette(name))
```

4.13 Fibers orientation coding

This example shows how to use the directional operators defined in example entitled "Directional morphological operators in 12 or 16 directions" to detect and code the orientation of fibers. This application is performed in two steps. The first step consists in determining the maximum elongation of each fiber in a given direction. This is performed by a directional ultimate opening. In the second step, the direction which, at each point, corresponds to the maximum elongation is given to this point. This example is applied to a binary image and on an hexagonal grid (6 directions are encoded).



fiber.png

coded_directions.png

Here is the source code for this example (can be found in Fibers orientation coding):

```
# Importing mamba
import mamba

# The basic operators of example M20 are used again.
def conjugateDirectionalErode(imIn, imOut, d, size,
                              grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    Performs the erosion of image 'imIn' in the conjugate direction 'd' of the
    grid and puts the result in image 'imOut'. The images can be binary, greyscale
    or 32-bit images. 'size' is a multiple of the distance between two adjacent
    points in the conjugate directions. 'edge' is set to FILLED by default (can
    be changed).
    Note that this operator is not equivalent to successive erosions by a doublet
    of points.
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    mamba.copy(imIn, imOut)

    j3 = mamba.transposeDirection(d, grid=grid)
    j2 = mamba.rotateDirection(d, grid=grid)
    j1 = mamba.transposeDirection(j2, grid=grid)
    val = mamba.computeMaxRange(imIn)[1] * int(edge == mamba.FILLED)
    for i in range(size):
        mamba.copy(imOut, imWrk1)
```

```
        mamba.copy(imOut, imWrk2)
        mamba.linearErode(imWrk1, imWrk1, d, n=1, grid=grid, edge=edge)
        mamba.shift(imWrk1, imWrk1, j1, 1, val, grid=grid)
        mamba.logic(imWrk1, imOut, imWrk1, "inf")
        mamba.linearErode(imWrk2, imWrk2, j2, n=1, grid=grid, edge=edge)
        mamba.shift(imWrk2, imWrk2, j3, 1, val, grid=grid)
        mamba.logic(imWrk2, imOut, imWrk2, "inf")
        mamba.logic(imWrk1, imWrk2, imOut, "sup")

def directionalErode(imIn, imOut, d, size,
                     grid=mamba.DEFAULT_GRID, edge=mamba.FILLED):
    """
    General directional erosion of image 'imIn', result in 'imOut'. The images
    can have any format. The direction 'd' is coded clockwise from 1 to 12 or
    16. Direction 1 corresponds to 60 degrees on the hexagonal grid and 90
    degrees on the square grid. 'size' corresponds to twice the normal size
    when standard directions are used. 'edge' is set to FILLED by default.
    """

    if (d & 1) == 0:
        conjugateDirectionalErode(imIn, imOut, d/2, size, grid=grid, edge=edge)
    else:
        mamba.linearErode(imIn, imOut, (d+1)/2, size*2, grid=grid, edge=edge)

# similar operators are defined for the dilation.
def conjugateDirectionalDilate(imIn, imOut, d, size,
                               grid=mamba.DEFAULT_GRID, edge=mamba.EMPTY):
    """
    Performs the dilation of image 'imIn' in the conjugate direction 'd' of the
    grid and puts the result in image 'imOut'. The images can be binary, greyscale
    or 32-bit images. 'size' is a multiple of the distance between two adjacent
    points in the conjugate directions. 'edge' is set to EMPTY by default (can
    be changed).
    Note that the linear structuring element is not connected. Points connecting
    adjacent points in the conjugate directions are not present. This is normal
    if we want to insure that the result is identical when we iterate n size 1
    operations to get a size n one (the same remark applies to the erosion).
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)
    mamba.copy(imIn, imOut)

    j3 = mamba.transposeDirection(d, grid=grid)
    j2 = mamba.rotateDirection(d, grid=grid)
    j1 = mamba.transposeDirection(j2, grid=grid)
    val = mamba.computeMaxRange(imIn)[1] * int(edge == mamba.FILLED)
    for i in range(size):
        mamba.copy(imOut, imWrk1)
        mamba.copy(imOut, imWrk2)
        mamba.linearDilate(imWrk1, imWrk1, d, 1, grid=grid, edge=edge)
        mamba.shift(imWrk1, imWrk1, j1, 1, val, grid=grid)
        mamba.logic(imWrk1, imOut, imWrk1, "sup")
        mamba.linearDilate(imWrk2, imWrk2, j2, 1, grid=grid, edge=edge)
        mamba.shift(imWrk2, imWrk2, j3, 1, val, grid=grid)
        mamba.logic(imWrk2, imOut, imWrk2, "sup")
        mamba.logic(imWrk1, imWrk2, imOut, "inf")

def directionalDilate(imIn, imOut, d, size,
                      grid=mamba.DEFAULT_GRID, edge=mamba.EMPTY):
    """
    General directional dilation of image 'imIn', result in 'imOut'. The images
    can have any format. The direction 'd' is coded clockwise from 1 to 12 or
    16. Direction 1 corresponds to 60 degrees on the hexagonal grid and 90
```

```
    degrees on the square grid. 'size' corresponds to twice the normal size
    when standard directions are used. 'edge' is set to EMPTY by default.
    """

    if (d & 1) == 0:
        conjugateDirectionalDilate(imIn, imOut, d/2, size, grid=grid, edge=edge)
    else:
        mamba.linearDilate(imIn, imOut, (d+1)/2, size*2, grid=grid, edge=edge)

# Definition of the linear ultimate opening.
def linearUltimateOpen(imIn, imOut, d, grid=mamba.DEFAULT_GRID):
    """
    This operator performs the ultimate directional opening of the binary image
    'imIn' in direction 'd' and puts the result in the 32-bit image 'imOut'.
    The value of 'imOut' at each point of 'imIn' corresponds to the length of the
    intercept passing through this point.
    """

    imWrk1 = mamba.imageMb(imIn)
    imWrk2 = mamba.imageMb(imIn)

    size = 0
    imOut.reset()
    mamba.copy(imIn, imWrk1)
    while mamba.computeVolume(imWrk1) != 0:
        size += 1
        directionalErode(imWrk1, imWrk1, d, 1, grid=grid, edge=mamba.EMPTY)
        td = (d + mamba.gridNeighbors(grid) - 1) % (mamba.gridNeighbors(grid) * 2) + 1
        directionalDilate(imWrk1, imWrk2, td, size, grid=grid)
        mamba.add(imOut, imWrk2, imOut)

#
def directionalCoding(imIn, imOut, grid=mamba.DEFAULT_GRID):
    """
    Coding of the direction which, at each point of 'imIn', corresponds to the
    direction of the maximal intercept through this point. The result of this
    coding is put in the grey scale image 'imOut'.
    6 directions are coded on the hexagonal grid, 8 on the square one.
    """
    imWrk1 = mamba.imageMb(imIn, 32)
    imWrk2 = mamba.imageMb(imIn, 32)
    imWrk3 = mamba.imageMb(imIn)
    imWrk4 = mamba.imageMb(imIn, 8)

    imOut.reset()
    imWrk1.reset()
    for i in range(mamba.gridNeighbors(grid=grid)):
        d = i + 1
        linearUltimateOpen(imIn, imWrk2, d, grid=grid)
        mamba.generateSupMask(imWrk2, imWrk1, imWrk3, True)
        mamba.logic(imWrk2, imWrk1, imWrk1, "sup")
        mamba.convertByMask(imWrk3, imWrk4, 0, d)
        mamba.logic(imWrk4, imOut, imOut, "sup")

# Trying these operators on a fiber image.
# Loading the initial image.
im1 = mamba.imageMb('fiber.png')
# Defining some working images.
im2 = mamba.imageMb(im1)
imbin1 = mamba.imageMb(im1, 1)
# A binary image is defined by a top-hat operator, followed by
# a thresholding and a filtering.
mamba.blackTopHat(im1, im2, 10)
mamba.threshold(im2, imbin1, 50, 255)
```

```
mamba.buildClose(imbin1, imbin1, 2)
# The directions encoding is performed.
directionalCoding(imbin1, im2)
# A palette for displaying the 6 possible directions is defined.
dirpal = (0, 0, 0, 255, 0, 0, 0, 255, 0, 0, 0, 255, 255, 255, 0, 0, 255, 255, 255, 0, 255)
for i in range(249):
    dirpal = dirpal + (0, 0, 0)
# The result is saved with the palette.
im2.save('coded_directions.png', palette=dirpal)
```

4.14 Gaussian blur : connecting Mamba to numpy/scipy

Numpy and scipy are powerful Python API to perform scientific computations. Numpy provides a multidimensional array object and a very complete range of functions for fast operations on it such as convolution (see http://www.scipy.org/). In this example, we show how to translate a Mamba image into an array and back and how to use it to perform a Gaussian blur.

snake.png

blurred_snake.png

Here is the source code for this example (can be found in Gaussian blur connecting Mamba to numpy):

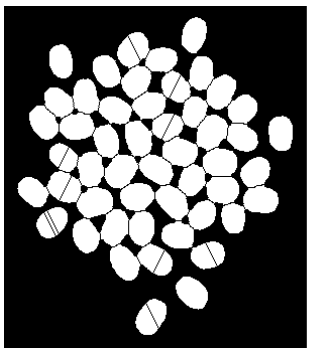
```
# Importing mamba and associates
from mamba import *
# Importing the numpy and scipy packages
# You need to have them installed on your computer
import numpy as np
import scipy.signal

def getArrayFromImage(imIn):
    """
    Creates an 2D array containing the same data as in 'imIn'. Only
    works for greyscale and 32-bit images. Returns the array.
    """
    if imIn.getDepth()==8:
        dtype = np.uint8
    elif imIn.getDepth()==32:
        dtype = np.uint32
    else:
        import mambaCore
        raiseExceptionOnError(mambaCore.ERR_BAD_DEPTH)

    (w,h) = imIn.getSize()
    # First extracting the raw data out of image imIn
    data = imIn.extractRaw()
    # creating an array with this data
    # At this step this is a one-dimensional array
    array1D = np.fromstring(data, dtype=dtype)
    # Reshaping it to the dimension of the image
    array2D = array1D.reshape((h,w))
    return array2D

def fillImageWithArray(array, imOut):
    """
    Fills image 'imOut' with the content of two dimensional 'array'. Only
    works for greyscale and 32-bit images.
```

```
    """
    # Checking depth
    if (imOut.getDepth()==8 and array.dtype != np.uint8) or \
       (imOut.getDepth()==32 and array.dtype != np.uint32) or \
       (imOut.getDepth()==1):
        import mambaCore
        raiseExceptionOnError(mambaCore.ERR_BAD_DEPTH)

    # image size
    (wi,hi) = imOut.getSize()
    # array size
    (ha,wa) = array.shape

    # Checking the sizes
    if wa!=wi or ha!=hi:
        import mambaCore
        raiseExceptionOnError(mambaCore.ERR_BAD_SIZE)

    # Extracting the data out of the array and filling the image with it
    data = array.tostring()
    imOut.loadRaw(data)

def gaussianBlur(imIn, imOut):
    """
    This function computes a Gaussian blur on 'imIn' using the scipy convolution
    with the appropriate kernel. The result is put into 'imOut'.
    """
    # Getting an array out of the snake image
    array = getArrayFromImage(imIn)
    # We then perform a convolution (here a gaussian blur)
    kernel = np.array( [ [ 0, 0, 5, 0, 0],
                         [ 0,11,16,11, 0],
                         [ 5,16,24,16, 5],
                         [ 0,11,16,11, 0],
                         [ 0, 0, 5, 0, 0] ])
    conv_array = scipy.signal.convolve2d(array, kernel, mode='same')
    # At this point conv_array contains value outside the rang of imOut.
    # Thus we rescale it.
    conv_array = conv_array - conv_array.min()
    conv_array = (conv_array * 255)/conv_array.max()
    # Changing its dtype to the original dtype
    conv_array = conv_array.astype(array.dtype)
    # Filling imOut with the result
    fillImageWithArray(conv_array, imOut)

im = imageMb("snake.png")
im1 = imageMb(im)
gaussianBlur(im, im1)
im1.save("blurred_snake.png")
```

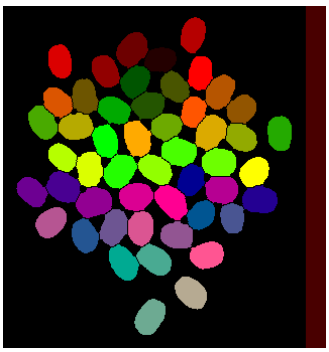
4.15 Particles (coffee grains) separation and counting

This example shows how to separate touching or slightly overlapping particles (coffee grains in this example) in order to measure and count them. This example also illustrates how some small enhancements may deeply increase the quality of the final result.

coffee_grains.jpg



segmented_grains_1.png



segmented_grains_2.png

Here is the source code for this example (can be found in Particles separation and counting):

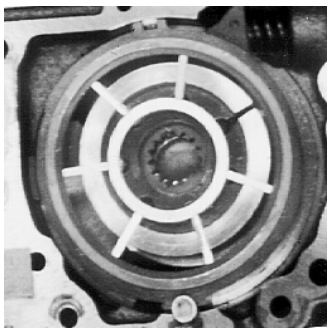
```
# Importing mamba
from mamba import *
import mambaDisplay

def autoThreshold(imIn, imOut):
    # Informations regarding this function can be found in previous
    # example.
    grad = imageMb(imIn)
    wrk = imageMb(imIn)
    level = imageMb(imIn, 1)
    gradient(imIn, grad)
    histo = getHistogram(imIn)
    distri = []
    for i in range(256):
        if histo[i]!=0:
            threshold(imIn, level, i, i)
            mul(level, grad, wrk)
            vol = computeVolume(wrk)/histo[i]
            distri.append(vol)
        else:
            distri.append(0)
    sd = sum(distri)
    sr = distri[0]
    threshval = 0
    while(sr<(sd/2)):
        threshval += 1
        sr += distri[threshval]
```

```
    threshold(imIn, imOut, threshval, 255)
    return threshval

def segmentGrains1(imIn, imOut):
    """
    Segments a binary image 'imIn' displaying grains using the distance
    function and puts the result in binary image 'imOut'.
    """
    imDist = imageMb(imIn, 32)
    imDist8 = imageMb(imIn, 8)
    imWTS = imageMb(imIn, 8)

    # Segmentation of the grains with the distance function. Firstly, we
    # compute the distance function (note the edge programming)
    computeDistance(imIn, imDist, edge=FILLED)

    # We verify (with computeRange) that the distance image is lower than
    # 256 and we copy it into a greyscale (8-bit) image
    computeRange(imDist)
    copyBytePlane(imDist, 0, imDist8)

    # The distance function is inverted and its valued watershed is computed
    negate(imDist8, imDist8)
    valuedWatershed(imDist8, imWTS)

    # The watershed lines are extracted (by thresholding) and subtracted from the
    # initial image to give the first result
    threshold(imWTS, imOut, 1, 255)
    diff(imIn, imOut, imOut)

def segmentGrains2(imIn, imOut):
    """
    Segments a binary image 'imIn' displaying grains using the distance
    function and puts the results in greyscale image 'imOut'.
    This segmentation uses a specific marker image and watershed segmentation
    to obtain better results than the previous function.
    This function returns the number of grains counted.
    """
    imMarker = imageMb(imIn, 1)
    imDist = imageMb(imIn, 32)
    imDist8 = imageMb(imIn, 8)
    imWTS32 = imageMb(imIn, 32)
    imWTS = imageMb(imIn, 8)

    # Segmentation of the grains with the distance function. Firstly, we
    # compute the distance function (note the edge programming)
    computeDistance(imIn, imDist, edge=FILLED)

    # We verify (with computeRange) that the distance image is lower than
    # 256 and we copy it into a greyscale (8-bit) image
    computeRange(imDist)
    copyBytePlane(imDist, 0, imDist8)

    # The distance function is inverted and its valued watershed is computed
    negate(imDist8, imDist8)

    # Computing a marker image
    minima(imDist8, imMarker)
    dilate(imMarker, imMarker, 2)

    # Then, we compute the watershed of the inverted distance function controlled
    # by this marker set (note the number of connected components given by the
    # labelling operator; they should correspond to the number of grains)
    nb = label(imMarker, imWTS32)
```

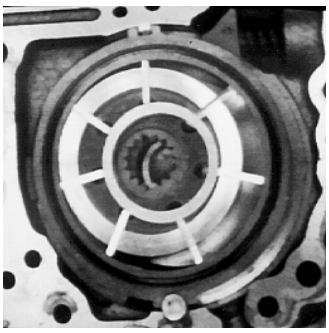
```
    watershedSegment(imDist8, imWTS32)

    # We build the labelled catchment basins
    copyBytePlane(imWTS32, 3, imWTS)
    negate(imWTS, imWTS)
    copyBytePlane(imWTS32, 0, imOut)
    logic(imOut, imWTS, imOut, "inf")

    # Then, we obtain the final (and better) result. Each grain is labelled
    convert(imIn, imWTS)
    logic(imOut, imWTS, imOut, "inf")

    return nb

# Loading the initial image and creating some working images
im = imageMb("coffee_grains.jpg")
imthresh = imageMb(im, 1)

# Using the function describes in exampleA1.py we compute the automatic
# threshold of our initial image
autoThreshold(im, imthresh)
# A bit of tidying
opening(imthresh, imthresh, 2)
negate(imthresh, imthresh)

# Computing the first segmentation of the coffee grains
imSeg1 = imageMb(im, 1)
segmentGrains1(imthresh, imSeg1)
# We store this result
imSeg1.save("segmented_grains_1.png")

# This first result is not perfect as some grains are badly segmented. This
# over-segmentation is due to parity problems affecting some maxima of the
# distance function. In order to cope with these defects, we use a marker
# controlled watershed of the inverted distance function. The markers are
# generated by a dilation of the minima of the inverted distance function

# Computing the second segmentation of the coffee grains
imSeg2 = imageMb(im, 8)
nb = segmentGrains2(imthresh, imSeg2)
# We store this result
# The palette can be changed before saving the result
imSeg2.save("segmented_grains_2.png", palette=mambaDisplay.getPalette("patchwork"))
print("number of grains : %d" % (nb))
# The result should be 50, however due to the black band on the right
# the function counts 51 coffee grains.
```

4.16 Quality control of a clutch device

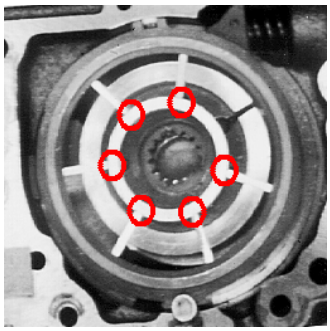
This example explains how a clutch device can be checked by computer vision. Two pieces are controlled in the process: a star-shaped spring at the center of the device and a seal surrounding the casing. The first one can be damaged (missing branches) and the second one can be missing. This example shows how the regions where these pieces are supposed to be can be emphasized by a geometrical and topological approach.



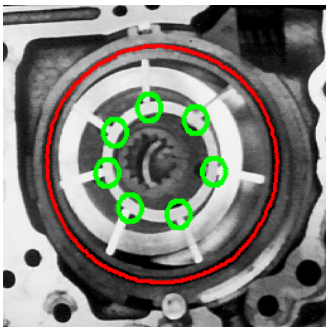
motor1.png



motor2.png



motor1_check.png



motor2_check.png

Here is the source code for this example (can be found in Quality control of clutch device):

```
# Importing mamba
from mamba import *

# The main checking routine.
def checkClutch(imIn, springBranches, sealZone):
    """
    this operator checks the spring and the seal of a clutch device in 'imIn'.
    'springBranches' contains the branches of the spring which have been
    detected and 'sealZone' contains the region where the seal should be. This
    latter image is empty if the seal is present.
    This procedure returns a 2-item 'check' list. 'check[0]' indicates if the
    spring is OK (1) or not (0). 'check[1]' indicates if the seal is missing (0)
    or not (1).
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn, 1)
    imWrk4 = imageMb(imIn,1)
    imWrk5 = imageMb(imIn, 1)
    imWrk6 = imageMb(imIn, 1)
    imWrk7 = imageMb(imIn)
    check = [0, 0]
    # The initial image is filtered.
    buildClose(imIn, imWrk1, 2)
    buildOpen(imWrk1, imWrk1, 2)
    # copy for latter use.
    copy(imWrk1, imWrk7)
```

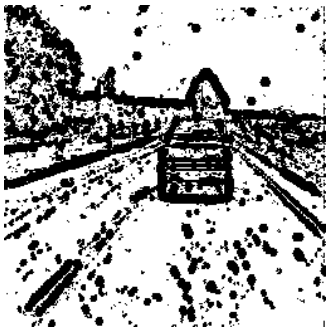
```
    # Firstly, we check the spring.
    # Parts of the casing touching the edges are removed.
    removeEdgeParticles(imWrk1, imWrk1)
    # The center region of the clutch is extracted.
    closeHoles(imWrk1, imWrk2)
    sub(imWrk2, imWrk1, imWrk2)
    # The central region is extracted by an opening by reconstruction.
    threshold(imWrk2, imWrk3, 10, 255)
    buildOpen(imWrk3, imWrk4, 20)
    # Looking for the branches of the spring. Firstly, the spaces between
    # the branches are extracted.
    dilate(imWrk4, imWrk5, 15)
    logic(imWrk3, imWrk5, imWrk6, "inf")
    dilate(imWrk6, imWrk3, 10)
    diff(imWrk5, imWrk3, imWrk5)
    logic(imWrk5, imWrk6, imWrk5, "sup")
    # The center region and 7 spaces between the branches should appear
    # if the spring is correct.
    check[0] = (computeConnectivityNumber(imWrk5) == 8)
    # This part of the procedure is just here to generate a binary image
    # showing the junctions of the different branches of the spring (not
    # necessary for the checking). These junctions correspond to multiple
    # points of the SKIZ of the previous image.
    fastSKIZ(imWrk5, imWrk5)
    negate(imWrk5, imWrk5)
    multiplePoints(imWrk5, imWrk5)
    removeEdgeParticles(imWrk5, imWrk5)
    # Eachmultiple point is reduced to a single point by thinning.
    thinD(imWrk5, springBranches)
    # The junctions are surrounded by a circle for a better view.
    dodecagonalDilate(springBranches, imWrk5, 9)
    dilate(imWrk5, springBranches, 2)
    diff(springBranches, imWrk5, springBranches)
    dilate(springBranches, springBranches)
    #
    # Secondly, the seal is checked.
    # The region of the image where the seal should be is generated from the
    # central region extracted above.
    dodecagonalDilate(imWrk4, imWrk5, 45)
    dodecagonalDilate(imWrk5, imWrk6, 30)
    diff(imWrk6, imWrk5, imWrk5)
    convert(imWrk5, imWrk2)
    # The corresponding region in the initial filtered image is selected.
    logic(imWrk7, imWrk2, imWrk7, "inf")
    # A black Top-Hat operator is applied (should emphasize the seal location
    # if missing.
    blackTopHat(imWrk7, imWrk2, 5)
    # The result is thresholded (a more robust threshold when applied to a
    # Top-Hat transform).
    threshold(imWrk2, imWrk3, 15, 255)
    # The binary image is cleaned (small holes filled).
    buildClose(imWrk3, imWrk3)
    # A SKIZ indicates if the seal is missing.
    negate(imWrk3, imWrk3)
    fastSKIZ(imWrk3, sealZone)
    negate(sealZone, sealZone)
    # The missing seal location is emphasized for display.
    dilate(sealZone, sealZone)
    # If the seal is missing, the sealZone image is not empty.
    check[1] = (computeVolume(sealZone) == 0)
    return check

# Reading the first image.
im1 = imageMb('motor1.png')
```

```
# Defining result images
spring = imageMb(im1, 1)
seal = imageMb(im1, 1)

# Checking the clutch.
check = checkClutch(im1, spring, seal)
# Printing the result and setting the color of the spring branches locations:
# green, all the branches are here, red, some branches are missing.
print("First image checking:")
if check[0]:
    print("Spring is OK.")
    pal1 = (0, 255, 0)
else:
    print("Spring is defective.")
    pal1 = (255, 0, 0)
if check[1]:
    print("Seal is OK.")
    pal2 = (0, 255, 0)
else:
    print("Seal is missing.")
    pal2 = (255, 0, 0)
# Superposing the seal and spring locations to the original image and saving
# the result.
multiSuperpose(im1, seal, spring)
pal = ()
for i in range(254):
    pal += (i,i,i)
pal += pal2
pal += pal1
im1.save('motor1_check.png', palette=pal)

# Reading the next image.
im1 = imageMb('motor2.png')

# Resetting the result images
spring.reset()
seal.reset()

# Checking it.
check = checkClutch(im1, spring, seal)
print("Second image checking:")
if check[0]:
    print("Spring is OK.")
    pal1 = (0, 255, 0)
else:
    print("Spring is defective.")
    pal1 = (255, 0, 0)
if check[1]:
    print("Seal is OK.")
    pal2 = (0, 255, 0)
else:
    print("Seal is missing.")
    pal2 = (255, 0, 0)
# Storing the resulting images.
multiSuperpose(im1, seal, spring)
pal = ()
for i in range(254):
    pal += (i,i,i)
pal += pal2
pal += pal1
im1.save('motor2_check.png', palette=pal)
```

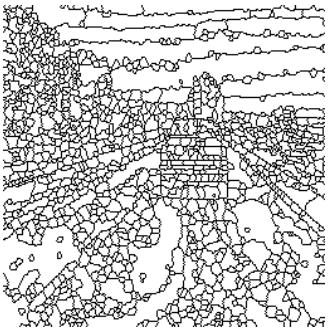
4.17 Regularised gradient and regularised minima

This example shows how regularised gradients can be used to define better markers for image segmentation. More details about the algorithm presented here can be found in http://cmm.ensmp.fr/ beucher/publi/SB_these.pdf, pages 63-65. A marker-controlled gradient watershed segmentation is performed to show the benefit of this operator compared to the use of a classical gradient.

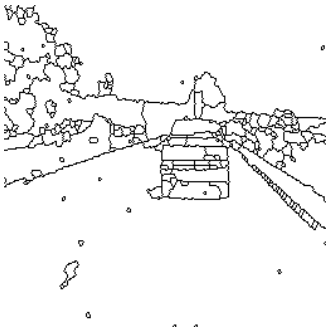
car.png



reg_minima.png



car_contours1.png



car_contours2.png

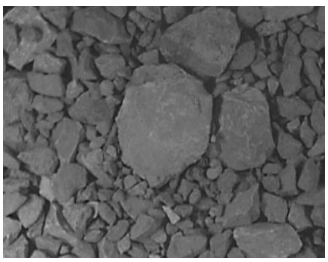
Here is the source code for this example (can be found in Regularised gradient and regularised minima):

```
# Importing mamba
from mamba import *

# Defining a regularised minima extractor.
def regularisedMinima(imIn, imOut, maxSize=8):
    """
    This operator performs regularised gradients of image 'imIn' for sizes in
    the range [1, maxSize - 1]. The influence zone of the regularised gradient
    is taken into account by dilating it (dilation of size i - 1). Then the
    supremum of all the dilated gradients is computed. This allows to absorb
    contours of small height which are in the vicinity of more contrasted ones.
    At the end, the minima of the supremum are extracted and stored in the
    binary image 'imOut'.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk2.reset()
    for i in range(1, maxSize + 1):
        regularisedGradient(imIn, imWrk1, i)
        n = i - 1
        dilate(imWrk1, imWrk1, n)
        logic(imWrk2, imWrk1, imWrk2, "sup")
    minima(imWrk2, imOut)

# Testing the algorithm.
# Reading the initial image.
im1 = imageMb('car.png')
```

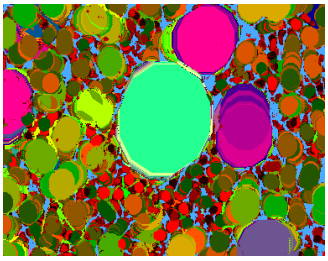
```
imMarkers = imageMb(im1, 1)

# Extracting the regularised minima. Applying the operator in the range (1, 4)
# is sufficient.
regularisedMinima(im1, imMarkers, 4)
# Storing the result.
imMarkers.save('reg_minima.png')
# Computing the classical gradient.
im2 = imageMb(im1)
gradient(im1, im2)
# Performing the classical gradient watershed segmentation.
im3 = imageMb(im1)
valuedWatershed(im2, im3)
imbin = imageMb(im1, 1)
threshold(im3, imbin, 0, 0)
# Saving the result.
imbin.save('car_contours1.png')
# Using the previous markers to perform a marker-controlled segmentation.
im32 = imageMb(im1, 32)
nb = label(imMarkers, im32)
watershedSegment(im2, im32)
copyBytePlane(im32, 3, im3)
threshold(im3, imbin, 0, 0)
# Saving this segmentation.
imbin.save('car_contours2.png')
```

4.18 Segmentation of a heap of rocks for evaluating its size distribution

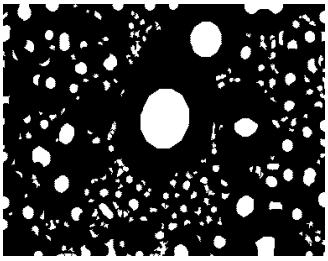
Measuring the size distribution of heaps of rocks is an important task for a quarry manager. This information is crucial to optimize the efficiency of rocks crushers. The best and fastest way to get this information is by means of an image of the rocks taken from the skip of a lorry or from a conveyor belt. However, obtaining the size distribution from the image is far from being easy. This example shows that the use of the ultimate opening operator allows to get not so bad results.



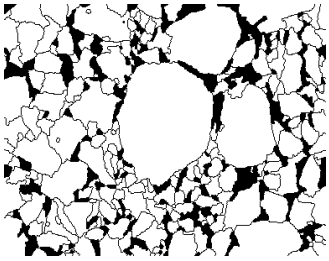
rocks.png



rocks_granu.png



rocks_markers.png



rocks_segment.png

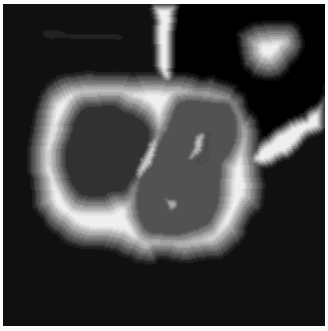
Here is the source code for this example (can be found in Segmentation of heap of rocks for evaluating its size distribution):

```
# Importing mamba
from mamba import *
import mambaDisplay

# Reading the image.
im1 = imageMb('rocks.png')

# Defining working images.
imWrk0 = imageMb(im1)
imWrk1 = imageMb(im1)
imWrk2 = imageMb(im1, 32)
imWrk3 = imageMb(im1)
imWrk4 = imageMb(im1)
imWrk5 = imageMb(im1, 1)
imWrk6 = imageMb(im1)
imWrk7 = imageMb(im1)
imWrk8 = imageMb(im1, 1)
imWrk9 = imageMb(im1, 1)

# The initial image is filtered by an alternate filter by reconstruction.
buildOpen(im1, imWrk0)
buildClose(imWrk0, imWrk0)
# The ultimate opening residual operator is applied to the image. In order
# to obtain results of better quality, operators with dodecagons are used.
ultimateIsotropicOpening(imWrk0, imWrk1, imWrk2)
# The granulometric image is extracted.
copyBytePlane(imWrk2, 0, imWrk3)
# This image is saved with a color palette.
imWrk3.save('rocks_granu.png', palette=mambaDisplay.getPalette("patchwork"))
# The flat zones of the granulometric image are extracted (these zones have
```

```
# a gradient equal to zero).
gradient(imWrk3, imWrk4)
threshold(imWrk4, imWrk5, 1, 255)
negate(imWrk5, imWrk5)
# Holes in these zones are filled (they correspond to artifacts).
closeHoles(imWrk5, imWrk5)
# The real size of these flat zones is determined by a dodecagonal distance
# function.
isotropicDistance(imWrk5, imWrk2, edge=FILLED)
copyBytePlane(imWrk2, 0, imWrk6)
# We add one to correct the bias brought by the gradient.
addConst(imWrk6, 1, imWrk6)
# The real size is compared to the size given by the granulometric function.
# When the real size is less than half the size of the granulometric function,
# the corresponding flat zone cannot be considered as a marker of a block.
divConst(imWrk3, 2, imWrk7)
generateSupMask(imWrk6, imWrk7, imWrk8, False)
# The extracted markers are filtered with an alternate filter by reconstruction
# to connect the closest ones and to remove the smallest ones.
buildClose(imWrk8, imWrk9)
buildOpen(imWrk8, imWrk9)
# The result is saved.
imWrk9.save('rocks_markers.png')

# In this second part, we shall use the previous markers to segment the blocks
# in the heap of rocks.

# The gradient of the original filtered image is computed.
gradient(imWrk0, imWrk1)
# In order to get a better result, we must also introduce markers for the
# background. These markers correspond the highest flat zones of the
# granulometric function.
t = computeRange(imWrk3)[1]
threshold(imWrk3, imWrk5, t, 255)
# A small correction is applied to insure that blocks markers and background
# markers are not touching each other.
dilate(imWrk5, imWrk8)
diff(imWrk9, imWrk8, imWrk9)
# To be sure that background markers mark only the background, they are reduced
# to a point (after filling of their holes).
closeHoles(imWrk5, imWrk5)
thinD(imWrk5, imWrk5)
# The rocks markers are labelled.
nbStones = label(imWrk9, imWrk2)
print("Number of stones : %d" % (nbStones))
# We add 1 to the label values to let room for the background marker.
add(imWrk2, imWrk9, imWrk2)
# The background marker is added (label 1). Note that all the connected
# components of the background marker share the same label value.
add(imWrk2, imWrk5, imWrk2)
# The watershed of the gradient is performed.
watershedSegment(imWrk1, imWrk2)
copyBytePlane(imWrk2, 3, imWrk4)
threshold(imWrk4, imWrk8, 0,0)
# The background marker is used to build the catchment basins corresponding
# to the background. Then, they are removed from the segmented image.
build(imWrk8, imWrk5)
diff(imWrk8, imWrk5, imWrk5)
# The segmented image is saved.
imWrk5.save('rocks_segment.png')
```

4.19 Topographic rendering of an image

This example illustrates the use of morphological operators for performing a topographic rendering of a grey scale image. To achieve this, we use the arrow encoding operator defined in example M18.



topography.png

rendering.png

Here is the source code for this example (can be found in Topographic rendering of an image):

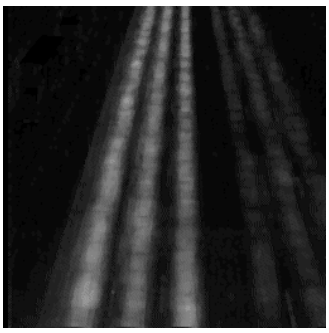
```
# Importing the mamba module.
from mamba import *

# The arrowing operator is used in this example (see exampleM18.py).
def arrowEncoding(imIn, imOut, oper, grid=DEFAULT_GRID):
    """
    This operator encodes the neighborhood of image 'imIn' and puts the result
    in greyscale image 'imOut'. The operator 'oper' may be one of the following:
    'sup', 'supeq', 'inf' or 'infeq'.
    The 'grid' may be an hexagonal or square one. In the first case, the range
    of encodings is [0, 63], in the second case, it is [0, 255].
    """

    imWrk = imageMb(imIn)
    imMask = imageMb(imIn, 1)
    imOut.reset()
    if oper in ["inf", "sup"]:
        strict = True
    elif oper in ["infeq", "supeq"]:
        strict = False
    else:
        strict = ""
        # The above statement generates an error if 'oper' is not one of the
        # allowed ones.
    for i in range(gridNeighbors(grid)):
        dir = i + 1
        if oper in ["sup", "supeq"]:
            shift(imIn, imWrk, dir, 1, computeMaxRange(imIn)[1], grid=grid)
            generateSupMask(imWrk, imIn, imMask, strict)
        else:
```

```
            shift(imIn, imWrk, dir, 1, 0, grid=grid)
            generateSupMask(imIn, imWrk, imMask, strict)
        copyBitPlane(imMask, i, imOut)

# Topographic rendering transform.
def shadowing(imIn, imOut, dir, brightness, height=(3,1)):
    """
    This operator takes the initial image 'imIn' and generates in image 'imOut'
    a topographic rendering of it. To enhance this rendering, a shadowing effect
    is superposed to the topography. The height of the sun is given by a tuple
    'height'. The two values correspond to the tangent of the angle of the sun
    rays. 'dir' is the direction of the light and 'brightness' indicates its
    intensity (contrast between sunny and shady regions).
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(imIn, 1)
    copy(imIn, imWrk2)
    copy(imIn, imWrk1)
    # This part of the process aims at defining the region where the
    # shadow appears. This is done by calculating the residue between
    # the dilation of the image by an oblique segment until idempotence
    # and the original image.
    while computeVolume(imWrk1) !=0:
        shift(imWrk1, imWrk1, dir, height[1], 0)
        subConst(imWrk1, height[0], imWrk1)
        logic(imWrk2, imWrk1, imWrk2, "sup")
    # imWrk3 contains the shadow region.
    sub(imWrk2, imIn, imWrk1)
    threshold(imWrk1, imWrk3, 1, 255)
    closing(imWrk3, imWrk3)
    # The topography is built by arrowing (on hexagonal grid).
    arrowEncoding(imIn, imWrk1, "sup")
    mulConst(imWrk1, 4, imWrk1)
    # A constant value is added to the pixels outside the shadow.
    convertByMask(imWrk3, imWrk2, brightness, 0)
    add(imWrk1, imWrk2, imOut)

# Defining a new color palette, the sepia palette (old photographs colors).
# This color palette will be used with the topographic rendering image.
sepia = ()
for i in range(64):
    sepia = sepia + (4 * i, 2 * i, 0)
for i in range(64):
    sepia = sepia + (255, 2 * (i + 64), 0)
for i in range(128):
    sepia = sepia + (255, 255, 2 * i)

# Reading the initial image.
im1 = imageMb('topography.png')
im2 = imageMb(im1)
# Defining the new palette.
# Performing the rendering process. The sun is supposed to shine from the
# north east.
shadowing(im1, im2, 4, 25)
# Saving the result.
im2.save('rendering.png', palette=sepia)
```

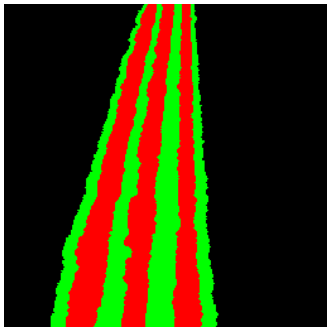
4.20 Traffic lanes segmentation

This example illustrates the use of two images of a traffic scene to segment the traffic lanes. Firstly, the mean of successive difference images is used to define lanes markers. This image indicates the regions of the scene where some motion occurs. The lanes markers are then dilated to generate the road mask. The size of the dilation depends of the vertical coordinate of each pixel in order to take into account the apparent metrics due to the perspective view. Secondly, the lanes are segmented. This segmentation is performed by a watershed of a distance function generated from the ground markings extracted from the mean image of the scene (moving objects have been removed).

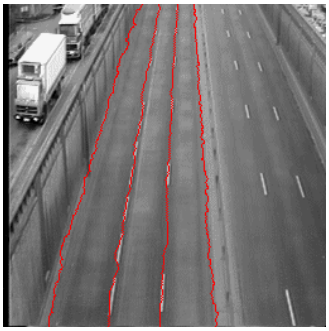
traffic_sum.png



traffic_diff.png



lanes_road_masks.png



segmented_lanes.png

Here is the source code for this example (can be found in Traffic lanes segmentation):

```
# Importing mamba
from mamba import *
from mambaDisplay.extra import *

# Special distance function generator.
def geodesicGreyDistance(imIn, imMask, imOut):
    """
    This operator computes a distance function from the greyscale image 'imIn'
    by performing a conic dilation until idempotence. Moreover, this dilation
    is a geodesic one, controlled by the 'imMask' image. The result is put in
    'imOut'.
    """
    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)

    copy(imIn, imWrk1)
    convert(imMask, imWrk2)
    prev_vol = 0
    logic(imIn, imWrk2, imOut, "inf")
    vol = computeVolume(imOut)
    while vol > prev_vol:
        prev_vol = vol
        dilate(imOut, imWrk1)
        subConst(imWrk1, 1, imWrk1)
        logic(imWrk1, imOut, imOut, "sup")
```

```
        logic(imOut, imWrk2, imOut, "inf")
        vol = computeVolume(imOut)

# Loading initial images.
im1 = imageMb('traffic_sum.png')
im2 = imageMb('traffic_diff.png')
# Defining resulting images.
# This image will contain the lanes markers.
im3 = imageMb(im1, 1)
# This image will contain the mask of the road.
im4 = imageMb(im1, 1)
# This image will contain the segmented lanes.
im5 = imageMb(im1, 1)
# Working images.
imWrk1 = imageMb(im1)
imWrk2 = imageMb(im1, 1)
imWrk3 = imageMb(im1, 32)
imWrk4 = imageMb(im1, 32)
imWrk5 = imageMb(im1, 32)
imWrk6 = imageMb(im1)

# Marking the lanes and generating the road mask.
# Filtering the mean of differences.
buildOpen(im2, imWrk1, 3)
# Thresholding it (the threshold value is equal to half the maximum grey value).
thrval = computeRange(imWrk1)[1] / 2
threshold(imWrk1, imWrk2, thrval, 255)
# The image is filtered and gives the lanes markers.
closing(imWrk2, im3)
# The distance function of the complementary set is computed. Note that the
# edge is set to 'filled'.
negate(im3, imWrk2)
computeDistance(imWrk2, imWrk3, edge=FILLED)
# Generating two markers at right and left of the image window. Note that the
# linear dilations are performed with a FILLED edge (quite unusual...).
imWrk4.reset()
linearDilate(imWrk4, imWrk4, 2, edge=FILLED)
linearDilate(imWrk4, imWrk4, 5, edge=FILLED)
# Rebuilding parts of the distance function at left and right.
build(imWrk3, imWrk4)
diff(imWrk3, imWrk4, imWrk4)
# Filling an image with the maximum value of the remaining parts of the distance
# function on every horizontal line of the image. This is performed by means of
# directional horizontal reconstructions. 'val' is just used to swallow the
# unused value returned by 'buildNeighbor'.
imWrk5.reset()
negate(imWrk5, imWrk5)
val = buildNeighbor(imWrk5, imWrk4, 2)
val = buildNeighbor(imWrk5, imWrk4, 5)
# The pixel values of the initial distance function which are lower than or
# equal to the previous mask image correspond to the road mask image. This
# approach allows to perform an adaptive dilation on each line in order to take
# the perspective into account (generalised distance).
generateSupMask(imWrk4, imWrk3, im4, False)
# Displaying the result by superposing the road and the lanes markers.
superpose(im4, im3)

# Segmenting the lanes.
# Extracting the ground markings (their thickness is known, hence the parameter
# value equal to 8).
whiteTopHat(im1, imWrk1, 8)
# The zone separating the lanes is extracted.
diff(im4, im3, imWrk2)
# Computing a special quasi-distance from the ground markings.
```

```
geodesicGreyDistance(imWrk1, imWrk2, imWrk6)
# Segmenting the lanes with a watershed of this quasi-distance controlled by the
# lanes markers. The watershed lines pass through the ground markings.
markerControlledWatershed(imWrk6, im3, imWrk1)
# Generating the boundaries of the lanes.
threshold(imWrk1, imWrk2, 1, 255)
diff(im4, imWrk2, imWrk2)
dilate(imWrk2, im4)
diff(im4, imWrk2, im5)
# Displaying the result.
superpose(im1, im5)
```

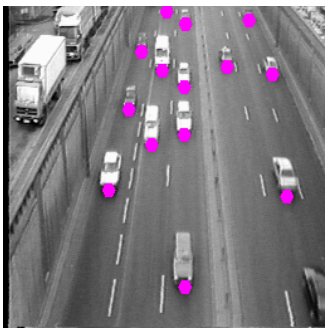
4.21 Traffic measurement, vehicles detection and counting

This example is the sequel of example A21. It shows how to detect vehicles in a traffic image. Although the process uses a simple feature to characterize a vehicle (namely the shadow in front of each car), it is rather robust. The mean image of example A21 is used again and is subtracted from the current image. Then the maxima of the resulting image which are sufficiently high are extracted as vehicles markers. The other interest of this example comes from the fact that an adaptive linear opening is defined to filter the previous minima. The size of the filtering varies according to the position of each pixel in the image. This size is controlled by an image which has been generated with the mask of the road obtained in example A21.

traffic.png

traffic_sum.png

traffic_road_mask.png


detected_vehicles.png

Here is the source code for this example (can be found in Traffic measurement vehicles detection and counting):

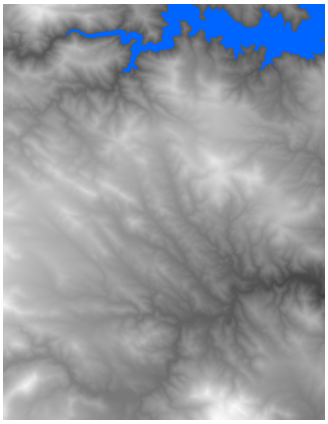
```
# Importing mamba
from mamba import *
from mambaDisplay.extra import *

# Defining the extended maxima extractor.
def deepMaxima(imIn, imOut, d):
    """
    This function extracts the maxima (or domes) of the greyscale image 'imIn'
    whose depth (or height) is higher than 'd'. The result is put in binary
    image 'imOut'.
    (Note that this operator is different from the 'maxima' operator).
    """

    imWrk = imageMb(imIn)
    subConst(imIn, d, imWrk)
    hierarBuild(imIn, imWrk)
    maxima(imWrk, imOut)

# Defining an adaptive linear opening.
def adaptiveHorizontalOpen(imIn, scaleIm, imOut):
    """
    This operator performs an horizontal linear opening of binary image 'imIn'
    and puts the result in binary image 'imOut'. The size of the opening is given
```

```
    for each pixel of 'imIn' by the value of the corresponding pixel in image
    'scaleIm'.
    This operator belongs to the generalized geodesic operators class.
    """

    imWrk1 = imageMb(imIn)
    imWrk2 = imageMb(imIn)
    imWrk3 = imageMb(scaleIm)
    imWrk4 = imageMb(imIn)
    copy(imIn, imWrk1)
    copy(scaleIm, imWrk3)
    # Computing an adaptive linear erosion. After each step of erosion, image
    # scaleIm is decremented. Only the pixels of imIn for which scaleIm is not
    # equal to zero are eroded. The other pixels are unchanged.
    while computeVolume(imWrk3) != 0:
        linearErode(imWrk1, imWrk2, 2, edge=EMPTY)
        threshold(imWrk3, imWrk4, 1, 255)
        logic(imWrk2, imWrk4, imWrk2, "inf")
        diff(imWrk1, imWrk4, imWrk1)
        logic(imWrk1, imWrk2, imWrk1, "sup")
        subConst(imWrk3, 1, imWrk3)
    # Linear horizontal dilation performed with a linear geodesic reconstruction.
    buildNeighbor(imIn, imWrk1, 2)
    copy(imWrk1, imOut)

# The vehicles extraction process is now defined.
# Firstly, the scale image used by the aadaptive opening is built from the mask
# of the road.
imb1 = imageMb('traffic_road_mask.png')
imb1.convert(1)
imb2 = imageMb(imb1)
imb3 = imageMb(imb1)
imW = imageMb(imb1, 32)
im1 = imageMb(imb1, 8)
im2 = imageMb(imb1, 8)
# A vertical marker is defined in image imb2.
linearDilate(imb2, imb2, 2, edge=FILLED)
# The right border of the image is rebuilt and added to the marker.
negate(imb1, imb3)
build(imb3, imb2)
logic(imb1, imb2, imb2, "sup")
# The distance function of this set is computed and transferred in a greyscale
# image. The restriction of this distance to the mask of the road is calculated.
computeDistance(imb2, imW, edge=FILLED)
copyBytePlane(imW, 0, im1)
convert(imb1, im2)
logic(im1, im2, im1, "inf")
# A linear reconstruction labels each line of the image window with the width
# of the road.
im2.fill(255)
# v is used to swallow the area returned by buildNeighbor (not used).
v = buildNeighbor(im2, im1, 2)
v = buildNeighbor(im2, im1, 5)
# The traffic image and the mean image are loaded.
im2 = imageMb('traffic.png')
im3 = imageMb('traffic_sum.png')
# Their difference (limited to zero) is performed.
sub(im3, im2, im3)
# The height of the maxima which are considered as salient ones is calculated.
# This height corresponds to half the range of values of the difference image.
height = computeRange(im3)[1]/2
# The salient maxima are extracted.
deepMaxima(im3, imb2, height)
# Holes in the maxima are closed.
```

```
closeHoles(imb2, imb2)
# The real width of the road is approximately equal to 9 meters. So, by dividing
# the im1 image by 9, each pixel of the resulting image is given a value equal
# to the number of iterations needed to achieve a linear erosion of size equal
# to one meter (the minimal width of a vehicle is about 1.5 m).
divConst(im1, 9, im3)
# An adaptive opening controlled by the previously defined scale function is
# performed.
adaptiveHorizontalOpen(imb2, im3, imb3)
# A small closing allows to connect close connected components of markers.
closing(imb3, imb3)
# Each vehicle marker is reduced to a point and dilated to enhance its display.
thinD(imb3, imb3)
dilate(imb3, imb3, 5)
# The result is displayed.
superpose(im2, imb3)
```

4.22 Volume controlled flooding

Watershed operators in Mamba can be used to compute flooding effects on Digital Elevation Model (DEM). In this example, we show how to create a volume controlled flooding operator. This is a very brutal approach as the low-level operators found in Mamba are controlled by water level rather than volume. This example also uses a DEM file that you can found at http://www.mapmart.com/samples.aspx (NED 10 meter).



flood.png

Here is the source code for this example (can be found in Volume controlled flooding):

```
# Importing mamba and associates
from mamba import *
import mambaDisplay

def downscale(imIn, imOut):
    # Reuse from previous example
    imWrk = imageMb(imIn)
    (mi, ma) = computeRange(imIn)
    subConst(imIn, mi, imWrk)
    mulConst(imWrk, 255, imWrk)
    divConst(imWrk, ma-mi, imWrk)
    copyBytePlane(imWrk, 0, imOut)

def computeFloodVolume(imDEM, imFloodArea):
    """
    Computes and returns the volume of *water* needed to flood completely
    the area described by 'imFloodArea' in image 'imDEM'.
    """
    imWrk1 = imageMb(imDEM)
    imWrk2 = imageMb(imDEM)
    mul(imDEM, imFloodArea, imWrk1)
    (mi, ma) = computeRange(imWrk1)
    imWrk2.fill(ma)
    mul(imWrk2, imFloodArea, imWrk2)
    sub(imWrk2, imWrk1, imWrk1)
    return computeVolume(imWrk1)

def volumeControlledFlood(imDEM, imFlood, targetVolume, grid=DEFAULT_GRID):
    """
    This computes the flood area in imDEM given a certain amount of water
    as given by 'targetVolume'. The flood starting point must be given
    in 'imFlood'. This image will also contains the resulting flooded
    area. The function returns the level reached by water and the actual
    volume needed to perform the flood (greater or equal to targetVolume).
    """
    imWrk1 = imageMb(imFlood)
    imWrk2 = imageMb(imFlood, 1) # will be used to represent the flooded area

    (mi, ma) = computeRange(imDEM)
```

```
    (mit, mat) = computeRange(imFlood)

    # First we check if the targetVolume will flood the
    # maximum level. In this case the flooded area is the whole image
    imWrk2.fill(1)
    vol = computeFloodVolume(imDEM, imWrk2)
    if vol<targetVolume:
        # A complete flooding is not sufficient to reach
        # target volume. We stop here and return the maximum level
        # and the associated volume
        return (ma+1, vol)

    # Using a dichotomy approach, we determine the level for which
    # the flood volume becomes equal or greater to target volume.
    inc = 100
    level = mi
    vol = 0
    while inc>0:
        while vol<targetVolume and level<ma+1:
            copy(imFlood, imWrk1)
            basinSegment(imDEM, imWrk1, max_level=level, grid=grid)
            threshold(imWrk1, imWrk2, 1, mat+1)
            vol = computeFloodVolume(imDEM, imWrk2)
            # next level
            level += inc
        # Changing the level to the previous level for which the flood volume
        # was below the target volume
        level -= 2*inc
        # Decreasing increment for better precision
        inc = inc/10

    copy(imWrk1, imFlood)
    return (level+2, vol)

imDEM = imageMb("NED10Meter.tif")
imFlood = imageMb(imDEM)

# Our flood starting point
imFlood.setPixel(255, (1337,81))

# Computing the flood area, level and actual volume needed to reach
# it with a control volume of 20000000. This is not of course a 'real'
# volume but you can compute its actual physical meaning if you have all
# the information regarding your DEM (such as grid resolution and pixel
# value metrics). Here we picked a SQUARE grid because it is most
# likely going to represent the grid used in the DEM.
print(volumeControlledFlood(imDEM, imFlood, 20000000, grid=SQUARE))

# Displaying the result
imDEM_8 = imageMb(imDEM, 8)
imFlood_8 = imageMb(imDEM, 8)
downscale(imDEM,imDEM_8)
copyBytePlane(imFlood, 0, imFlood_8)
logic(imDEM_8, imFlood_8, imDEM_8, "sup")
name = mambaDisplay.tagOneColorPalette(255,(0,100,255))
imDEM_8.save("flood.png", palette=mambaDisplay.getPalette(name))
```