



# Mamba Image Library Python Reference

Automatically generated using pydoc

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>mamba</b>	<b>3</b>
2.1	Classes	3
2.1.1	imageMb	3
2.2	Functions	5
2.2.1	add(imIn1, imIn2, imOut)	5
2.2.2	addConst(imIn, v, imOut)	5
2.2.3	basinSegment(imIn, imMarker, grid=DEFAULT_GRID, max_level=256)	5
2.2.4	buildNeighbor(imMask, imInout, d, grid=DEFAULT_GRID)	5
2.2.5	checkEmptiness(imIn)	5
2.2.6	compare(imIn1, imIn2, imOut)	5
2.2.7	computeDistance(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY)	5
2.2.8	computeMaxRange(imIn)	6
2.2.9	computeRange(imIn)	6
2.2.10	computeVolume(imIn)	6
2.2.11	convert(imIn, imOut)	6
2.2.12	convertByMask(imIn, imOut, mFalse, mTrue)	6
2.2.13	copy(imIn, imOut)	6
2.2.14	copyBitPlane(imIn, plane, imOut)	6
2.2.15	copyBytePlane(imIn, plane, imOut)	6
2.2.16	copyLine(imIn, nIn, imOut, nOut)	6
2.2.17	cropCopy(imIn, posIn, imOut, posOut, size)	6
2.2.18	diff(imIn1, imIn2, imOut)	7
2.2.19	diffNeighbor(imIn, imInout, nb, grid=DEFAULT_GRID, edge=EMPTY)	7
2.2.20	divConst(imIn, v, imOut)	7
2.2.21	dualbuildNeighbor(imMask, imInout, d, grid=DEFAULT_GRID)	7
2.2.22	extractFrame(imIn, threshold)	7
2.2.23	generateSupMask(imIn1, imIn2, imOut, strict)	7
2.2.24	getDirections(grid=DEFAULT_GRID)	7
2.2.25	getHistogram(imIn)	7
2.2.26	getImageCounter()	7
2.2.27	getShowImages()	7
2.2.28	gridNeighbors(grid=DEFAULT_GRID)	8
2.2.29	hierarBuild(imMask, imInout, grid=DEFAULT_GRID)	8
2.2.30	hierarDualBuild(imMask, imInout, grid=DEFAULT_GRID)	8
2.2.31	hitOrMiss(imIn, imOut, cse0, cse1, grid=DEFAULT_GRID)	8
2.2.32	infFarNeighbor(imIn, imInout, nb, amp, grid=DEFAULT_GRID, edge=FILLED)	8
2.2.33	infNeighbor(imIn, imInout, nb, count, grid=DEFAULT_GRID, edge=FILLED)	8
2.2.34	label(imIn, imOut, lblow=1, lbhigh=256, grid=DEFAULT_GRID)	9
2.2.35	logic(imIn1, imIn2, imOut, log)	9
2.2.36	lookup(imIn, imOut, lutable)	9
2.2.37	mul(imIn1, imIn2, imOut)	9
2.2.38	mulConst(imIn, v, imOut)	9
2.2.39	negate(imIn, imOut)	9
2.2.40	rotateDirection(d, step=1, grid=DEFAULT_GRID)	9
2.2.41	setDefaultGrid(grid)	9
2.2.42	setImageIndex(index)	10
2.2.43	setShowImages(showThem)	10
2.2.44	shift(imIn, imOut, d, amp, fill, grid=DEFAULT_GRID)	10
2.2.45	sub(imIn1, imIn2, imOut)	10
2.2.46	subConst(imIn, v, imOut)	10
2.2.47	supFarNeighbor(imIn, imInout, nb, amp, grid=DEFAULT_GRID, edge=EMPTY)	10
2.2.48	supNeighbor(imIn, imInout, nb, count, grid=DEFAULT_GRID, edge=EMPTY)	10
2.2.49	threshold(imIn, imOut, low, high)	10
2.2.50	tidyDisplays(displayer=None)	11
2.2.51	transposeDirection(d, grid=DEFAULT_GRID)	11

2.2.52	watershedSegment(imIn, imMarker, grid=DEFAULT_GRID, max_level=256)	11
<b>3</b>	<b>mambaDraw</b>	<b>11</b>
3.1	Functions	11
3.1.1	drawBox(imOut, square, value)	11
3.1.2	drawCircle(imOut, circle, value)	11
3.1.3	drawFillCircle(imOut, circle, value)	11
3.1.4	drawLine(imOut, line, value)	11
3.1.5	drawSquare(imOut, square, value)	11
3.1.6	getIntensityAlongLine(imOut, line)	11
<b>4</b>	<b>mambaExtra</b>	<b>12</b>
4.1	Functions	12
4.1.1	Mamba2PIL(imIn)	12
4.1.2	PIL2Mamba(pilim, imOut)	12
4.1.3	dynamicThreshold(imIn)	12
4.1.4	hitormissPatternSelector(grid=DEFAULT_GRID)	12
4.1.5	mix(imInR, imInG, imInB)	12
4.1.6	split(pilimIn, imOutR, imOutG, imOutB)	12
4.1.7	superpose(imIn1, imIn2)	12
4.1.8	tagOneColorPalette(value, color)	12
<b>5</b>	<b>mambaComposed.contrasts</b>	<b>13</b>
5.1	Functions	13
5.1.1	blackTopHat(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	13
5.1.2	gradient(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	13
5.1.3	halfGradient(imIn, imOut, type='intern', n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	13
5.1.4	regularisedGradient(imIn, imOut, n, grid=DEFAULT_GRID)	13
5.1.5	supBlackTopHat(imIn, imOut, n, grid=DEFAULT_GRID)	13
5.1.6	supWhiteTopHat(imIn, imOut, n, grid=DEFAULT_GRID)	13
5.1.7	whiteTopHat(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	13
<b>6</b>	<b>mambaComposed.erodil</b>	<b>13</b>
6.1	Classes	14
6.1.1	structuringElement	14
6.2	Functions	15
6.2.1	conjugateHexagonalDilate(imIn, imOut, size, edge=EMPTY)	15
6.2.2	conjugateHexagonalErode(imIn, imOut, size, edge=FILLED)	15
6.2.3	dilate(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=EMPTY)	15
6.2.4	dodecagonalDilate(imIn, imOut, size, edge=EMPTY)	15
6.2.5	dodecagonalErode(imIn, imOut, size, edge=FILLED)	15
6.2.6	doublePointDilate(imIn, imOut, d, n, grid=DEFAULT_GRID, edge=EMPTY)	15
6.2.7	doublePointErode(imIn, imOut, d, n, grid=DEFAULT_GRID, edge=FILLED)	15
6.2.8	erode(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED)	16
6.2.9	linearDilate(imIn, imOut, d, n=1, grid=DEFAULT_GRID, edge=EMPTY)	16
6.2.10	linearErode(imIn, imOut, d, n=1, grid=DEFAULT_GRID, edge=FILLED)	16
6.2.11	octogonalDilate(imIn, imOut, size, edge=EMPTY)	16
6.2.12	octogonalErode(imIn, imOut, size, edge=FILLED)	16
<b>7</b>	<b>mambaComposed.erodilLarge</b>	<b>16</b>
7.1	Functions	16
7.1.1	largeDodecagonalDilate(imIn, imOut, size, edge=EMPTY)	16
7.1.2	largeDodecagonalErode(imIn, imOut, size, edge=FILLED)	16
7.1.3	largeHexagonalDilate(imIn, imOut, size, edge=EMPTY)	16
7.1.4	largeHexagonalErode(imIn, imOut, size, edge=FILLED)	17
7.1.5	largeLinearDilate(imIn, imOut, dir, size, grid=DEFAULT_GRID, edge=EMPTY)	17

7.1.6	largeLinearErode(imIn, imOut, dir, size, grid=DEFAULT_GRID, edge=FILLED)	17
7.1.7	largeOctogonalDilate(imIn, imOut, size, edge=EMPTY)	17
7.1.8	largeOctogonalErode(imIn, imOut, size, edge=FILLED)	17
7.1.9	largeSquareDilate(imIn, imOut, size, edge=EMPTY)	17
7.1.10	largeSquareErode(imIn, imOut, size, edge=FILLED)	17
<b>8</b>	<b>mambaComposed.filter</b>	<b>17</b>
8.1	Functions	17
8.1.1	alternateFilter(imIn, imOut, n, openFirst, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	17
8.1.2	autoMedian(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	17
8.1.3	fullAlternateFilter(imIn, imOut, n, openFirst, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	17
8.1.4	linearAlternateFilter(imIn, imOut, n, openFirst, grid=DEFAULT_GRID)	18
8.1.5	simpleLevelling(imIn, imMask, imOut, grid=DEFAULT_GRID)	18
8.1.6	strongLevelling(imIn, imOut, n, eroFirst, grid=DEFAULT_GRID)	18
<b>9</b>	<b>mambaComposed.geodesy</b>	<b>18</b>
9.1	Functions	18
9.1.1	build(imMask, imInout, grid=DEFAULT_GRID)	18
9.1.2	closeHoles(imIn, imOut, grid=DEFAULT_GRID)	18
9.1.3	dualBuild(imMask, imInout, grid=DEFAULT_GRID)	18
9.1.4	geodesicDilate(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	18
9.1.5	geodesicDistance(imIn, imMask, imOut, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	18
9.1.6	geodesicErode(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	19
9.1.7	maxima(imIn, imOut, h=1, grid=DEFAULT_GRID)	19
9.1.8	minima(imIn, imOut, h=1, grid=DEFAULT_GRID)	19
9.1.9	removeEdgeParticles(imIn, imOut, grid=DEFAULT_GRID)	19
<b>10</b>	<b>mambaComposed.measure</b>	<b>19</b>
10.1	Functions	19
10.1.1	computeArea(imIn, scale=(1.0, 1.0))	19
10.1.2	computeComponentsNumber(imIn, grid=DEFAULT_GRID)	19
10.1.3	computeConnectivityNumber(imIn, grid=DEFAULT_GRID)	19
10.1.4	computeDiameter(imIn, dir, scale=(1.0, 1.0), grid=DEFAULT_GRID)	19
10.1.5	computeFeretDiameters(imIn, scale=(1.0, 1.0))	20
10.1.6	computePerimeter(imIn, scale=(1.0, 1.0), grid=DEFAULT_GRID)	20
<b>11</b>	<b>mambaComposed.miscellaneous</b>	<b>20</b>
11.1	Functions	20
11.1.1	ceilingAdd(imIn1, imIn2, imOut)	20
11.1.2	ceilingAddConst(imIn, v, imOut)	20
11.1.3	drawEdge(imOut, thick=1)	20
11.1.4	floorSub(imIn1, imIn2, imOut)	20
11.1.5	floorSubConst(imIn, v, imOut)	20
11.1.6	isotropicDistance(imIn, imOut, edge=FILLED)	21
11.1.7	translate(imIn, imOut, deltaX, deltaY, v=0)	21
<b>12</b>	<b>mambaComposed.openclose</b>	<b>21</b>
12.1	Functions	21
12.1.1	buildClose(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	21
12.1.2	buildOpen(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))	21
12.1.3	close(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED)	21
12.1.4	infClose(imIn, imOut, n, grid=DEFAULT_GRID)	21
12.1.5	linearClose(imIn, imOut, dir, n, grid=DEFAULT_GRID, edge=FILLED)	21
12.1.6	linearOpen(imIn, imOut, dir, n, grid=DEFAULT_GRID, edge=FILLED)	21

12.1.7	<code>open(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED)</code>	22
12.1.8	<code>supOpen(imIn, imOut, n, grid=DEFAULT_GRID)</code>	22
<b>13</b>	<b>mambaComposed.residues</b>	<b>22</b>
13.1	Functions	22
13.1.1	<code>binarySkeletonByOpening(imIn, imOut1, imOut2, grid=DEFAULT_GRID, edge=FILLED)</code>	22
13.1.2	<code>binaryUltimateErosion(imIn, imOut1, imOut2, grid=DEFAULT_GRID, edge=FILLED)</code>	22
13.1.3	<code>fullRegularisedGradient(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	22
13.1.4	<code>quasiDistance(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	22
13.1.5	<code>skeletonByOpening(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	23
13.1.6	<code>ultimateBuildOpening(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	23
13.1.7	<code>ultimateErosion(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	23
13.1.8	<code>ultimateIsotropicOpening(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	23
13.1.9	<code>ultimateOpening(imIn, imOut1, imOut2, grid=DEFAULT_GRID)</code>	23
<b>14</b>	<b>mambaComposed.segment</b>	<b>23</b>
14.1	Functions	23
14.1.1	<code>fastSKIZ(imIn, imOut, grid=DEFAULT_GRID)</code>	23
14.1.2	<code>geodesicSKIZ(imIn, imMask, imOut, grid=DEFAULT_GRID)</code>	23
14.1.3	<code>markerControlledWatershed(imIn, imMarkers, imOut, grid=DEFAULT_GRID)</code>	24
14.1.4	<code>mosaic(imIn, imOut, imWts, grid=DEFAULT_GRID)</code>	24
14.1.5	<code>mosaicGradient(imIn, imOut, grid=DEFAULT_GRID)</code>	24
14.1.6	<code>valuedWatershed(imIn, imOut, grid=DEFAULT_GRID)</code>	24
<b>15</b>	<b>mambaComposed.sequence</b>	<b>24</b>
15.1	Classes	24
15.1.1	<code>sequenceMb</code>	24
15.2	Functions	26
15.2.1	<code>closeByCylinderSequence(sequence, height, section)</code>	26
15.2.2	<code>copySequence(sequenceIn, sequenceOut)</code>	26
15.2.3	<code>dilateByCylinderSequence(sequence, height, section)</code>	26
15.2.4	<code>erodeByCylinderSequence(sequence, height, section)</code>	26
15.2.5	<code>openByCylinderSequence(sequence, height, section)</code>	26
<b>16</b>	<b>mambaComposed.statistic</b>	<b>26</b>
16.1	Functions	26
16.1.1	<code>getMean(imIn)</code>	26
16.1.2	<code>getMedian(imIn)</code>	26
16.1.3	<code>getVariance(imIn)</code>	26
<b>17</b>	<b>mambaComposed.thinthick</b>	<b>26</b>
17.1	Classes	26
17.1.1	<code>doubleStructuringElement</code>	26
17.2	Functions	27
17.2.1	<code>binaryHMT(imIn, imOut, dse, edge=EMPTY)</code>	27
17.2.2	<code>blackClip(imIn, imOut, step=0, grid=DEFAULT_GRID)</code>	27
17.2.3	<code>computeSKIZ(imIn, imOut, grid=DEFAULT_GRID)</code>	27
17.2.4	<code>endPoints(imIn, imOut, grid=DEFAULT_GRID, edge=FILLED)</code>	27
17.2.5	<code>fullGeodesicThick(imIn, imMask, imOut, dse)</code>	28
17.2.6	<code>fullGeodesicThin(imIn, imMask, imOut, dse)</code>	28
17.2.7	<code>fullThick(imIn, imOut, dse)</code>	28
17.2.8	<code>fullThin(imIn, imOut, dse, edge=EMPTY)</code>	28
17.2.9	<code>geodesicThick(imIn, imMask, imOut, dse)</code>	28
17.2.10	<code>geodesicThin(imIn, imMask, imOut, dse)</code>	28
17.2.11	<code>homotopicReduction(imIn, imOut, grid=DEFAULT_GRID)</code>	28
17.2.12	<code>infThin(imIn, imOut, dse, edge=EMPTY)</code>	28
17.2.13	<code>multiplePoints(imIn, imOut, grid=DEFAULT_GRID)</code>	28
17.2.14	<code>rotatingGeodesicThick(imIn, imMask, imOut, dse)</code>	29

17.2.15 rotatingGeodesicThin(imIn, imMask, imOut, dse) . . . . .	29
17.2.16 rotatingThick(imIn, imOut, dse) . . . . .	29
17.2.17 rotatingThin(imIn, imOut, dse, edge=FILLED) . . . . .	29
17.2.18 supThick(imIn, imOut, dse) . . . . .	29
17.2.19 thick(imIn, imOut, dse) . . . . .	29
17.2.20 thickD(imIn, imOut, grid=DEFAULT_GRID) . . . . .	29
17.2.21 thickL(imIn, imOut, grid=DEFAULT_GRID) . . . . .	29
17.2.22 thickM(imIn, imOut, grid=DEFAULT_GRID) . . . . .	29
17.2.23 thin(imIn, imOut, dse, edge=EMPTY) . . . . .	30
17.2.24 thinD(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY) . . . . .	30
17.2.25 thinL(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY) . . . . .	30
17.2.26 thinM(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY) . . . . .	30
17.2.27 whiteClip(imIn, imOut, step=0, grid=DEFAULT_GRID, edge=FILLED) . . . . .	30

## List of Figures

## 1 Introduction

This document is the Mamba library Python reference.

It applies to version 1.0 .

It gives information regarding all the classes, functions and exceptions defined in the Python part of the Mamba library. This extends to all the basic functions found in Mamba but also to all the modules found in the `mambaComposed` package. The document also gives information regarding peripheral modules such as `mambaDraw`.

This document is intended for reference only. To learn more about Mamba, start with the Mamba user manual.

## 2 mamba

This is the main module of the Mamba Image library. It provides basic functions and classes needed for handling images and defining mathematical morphology transformations and algorithms. This module also contains image display functionalities and other user-friendly features.

### 2.1 Classes

#### 2.1.1 imageMb

Defines the `imageMb` class and its methods. All mamba images are represented by this class.

`__del__`(self)

`__init__`(self, \*args, \*\*kwargs) Constructor for a Mamba image object.

This constructor allows a wide range of possibilities for defining an image:

- `imageMb()`: without arguments will create an empty greyscale image.
- `imageMb(im)`: will create an image using the same size and depth than 'im'.
- `imageMb(depth)`: will create an image with the desired 'depth' (1, 8 or 32).
- `imageMb(path)`: will load the image located in 'path'.
- `imageMb(im, depth)`: will create an image using the same size than 'im' and the specified 'depth'.
- `imageMb(path, depth)`: will load the image located in 'path' and convert it to the specified 'depth'.
- `imageMb(width, height)`: will create an image with size 'width'x'height'.
- `imageMb(width, height, depth)`: will create an image with size 'width'x'height' and the specified 'depth'.

When not specified, the width and height of the image will be set to 256x256. The default depth is 8 (greyscale).

When loading an image from a file, please note that Mamba accepts all kinds of images (actually all the PIL supported formats). You can specify the RGB filter that will be used to convert a color image into a greyscale image by adding the `rgbfilter=<your_filter>` to the argument of the constructor.

`__str__`(self)

`convert`(self, depth) Converts the image depth to the given 'depth'. The conversion algorithm is identical to the conversion used in the `convert` function (see this function for details).

**fastSetPixel(self, value, position)** Sets the pixel at 'position' with 'value'. 'position' is a tuple holding (x,y).

This function will not update the display to enable a faster drawing. So make sure to call the `updateDisplay()` method once your drawing is finished, if you want to see the result.

**fill(self, v)** Completely fills the image with a given value 'v'. A zero value makes the image completely dark.

**freezeDisplay(self)** Called to freeze the display of the image. Thus the image may evolve but the display will not show these evolutions until the method `unfreezeDisplay` is called.

**getDepth(self)** Returns the depth of the image.

**getName(self)** Returns the name of the image.

**getPixel(self, position)** Gets the pixel value at 'position'. 'position' is a tuple holding (x,y). Returns the value of the pixel.

**getSize(self)** Returns the size (a tuple width and height) of the image.

**hideDisplay(self)** Called to hide the display associated to the image. If the display is hidden, the computations go faster.

**load(self, path, rgbfilter=None)** Loads the image in 'path' into the Mamba image.

The optional 'rgbfilter' argument can be used to specify how to convert a color image into a greyscale image. It is a sequence of 3 float values indicating the amount of red, green and blue to take from the image to obtain the grey value. By default, the color conversion uses the ITU-R 601-2 luma transform (see PIL documentation for details).

**reset(self)** Resets the image (all the pixels are put to 0). This method is equivalent to `im.fill(0)`.

**resetPalette(self)** Undefined the palette used to convert the image in color for display and save. The greyscale palette will be used then.

**save(self, path)** Saves the image at the corresponding 'path' using PIL library. The format is automatically deduced by PIL from the image name extension. Note that, if the image comes with a palette, the image is saved with this palette.

**setName(self, name)** Use this function to set the image 'name'.

**setPalette(self, pal)** Defines the palette used to convert the image in color for display and save. 'pal' may be `rainbow`, `inverted_rainbow`, `patchwork` or any user-defined palette.

**setPixel(self, value, position)** Sets the pixel at 'position' with 'value'. 'position' is a tuple holding (x,y).

**showDisplay(self)** Called to show the display associated to the image. Showing the display may significantly slow down your operations.

**unfreezeDisplay(self)** Called to unfreeze the display of the image. Thus the image display will be updated along with the modifications inside the image.

**updateDisplay(self)** Called when the display associated to the image must be updated (the image has changed).

## 2.2 Functions

### 2.2.1 `add(imIn1, imIn2, imOut)`

Adds 'imIn2' pixel values to 'imIn1' pixel values and puts the result in 'imOut'. The operation can be sum up in the following formula :

$$\text{imOut} = \text{imIn1} + \text{imIn2}.$$

You can mix formats in the addition operation (a binary image can be added to a greyscale image, etc...). However you must ensure that the output image is as deep as the deepest of the two added images.

The operation is also saturated for greyscale images (e.g. on a 8-bit greyscale image,  $255+1=255$ ). With 32-bit images, the addition is not saturated.

### 2.2.2 `addConst(imIn, v, imOut)`

Adds 'imIn' pixel values to value 'v' and puts the result in 'imOut'. The operation can be sum up in the following formula:

$$\text{imOut} = \text{imIn} + v$$

'imIn' and 'imOut' can be 8-bit or 32-bit images of same size and depth.

The operation is saturated (limited to 255) for greyscale images.

### 2.2.3 `basinSegment(imIn, imMarker, grid=DEFAULT_GRID, max_level=256)`

Segments greyscale image 'imIn' using the watershed algorithm. 'imMarker' is used both as the marker image (the wells from which the flooding proceeds) and as the output image. It is a 32-bit image. 'max\_level' can be used to limit the flooding process to a specific level (useful if you want to survey the flooding level by level).

'grid' will change the number of neighbors considered by the algorithm (HEXAGONAL is 6-Neighbors and SQUARE is 8-Neighbors).

The result is put inside 'imMarker'. The three first byte planes contain the actual segmentation (each segment has a specific label according to the original marker). This function only return catchment basins (no watershed line) and is faster than watershedSegment if you are not interested in the watershed line.

### 2.2.4 `buildNeighbor(imMask, imInout, d, grid=DEFAULT_GRID)`

Builds image 'imInout' in direction 'd' according to 'grid' using 'imMask' as a mask (the propagation is performed only in 'd' direction).

The function also returns the volume of the image 'imInout' after the build operation.

'grid' value can be HEXAGONAL or SQUARE.

### 2.2.5 `checkEmptiness(imIn)`

Checks if image 'imIn' is empty (i.e. completely black). Returns True if so, False otherwise.

'imIn' can be a 1-bit, 8-bit or 32-bit image.

### 2.2.6 `compare(imIn1, imIn2, imOut)`

Compares the two images 'imIn1' and 'imIn2'. The comparison is performed pixelwise by scanning the two images from top left to bottom right and it stops as soon as a pixel is different in the two images. The corresponding pixel in 'imOut' is set to the value of the pixel of 'imIn1'.

The function returns a tuple holding the position of the first mismatching pixel. The tuple value is (-1,-1) if the two images are identical.

'imOut' is not reset at the beginning of the comparison.

'imIn1', 'imIn2' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

### 2.2.7 `computeDistance(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY)`

Computes for each white pixel of binary 'imIn' the minimum distance to reach a connected component boundary while constantly staying in the set. The result is put in 32-bit 'imOut'.

The distance computation will be performed according to the 'grid' (HEXAGONAL is 6-Neighbors and SQUARE is 8-Neighbors). 'edge' can be FILLED or EMPTY.

### 2.2.8 computeMaxRange(imIn)

Returns a tuple with the minimum and maximum possible pixel values given the depth of image 'imIn'. The values are returned in a tuple holding the minimum and the maximum.

### 2.2.9 computeRange(imIn)

Computes the range, i.e. the minimum and maximum values, of image 'imIn'. The values are returned in a tuple holding the minimum and the maximum.

### 2.2.10 computeVolume(imIn)

Computes the volume of the image 'imIn', i.e. the sum of its pixel values. The computed integer value is returned by the function.

'imIn' can be a 1-bit, 8-bit or 32-bit image.

### 2.2.11 convert(imIn, imOut)

Converts the contents of 'imIn' to the depth of 'imOut' and puts the result in 'imOut'.

Only greyscale to binary and binary to greyscale conversion are supported. Value 255 in a greyscale image is considered as 1 in a binary one. All other values are transformed to 0. The reverse convention applies.

### 2.2.12 convertByMask(imIn, imOut, mFalse, mTrue)

Converts a binary image 'imIn' into a greyscale image (8-bit) or a 32-bit image and puts the result in 'imOut'. white pixels of 'imIn' are set to value 'mTrue' in the output image and the black pixels set to value 'mFalse'.

### 2.2.13 copy(imIn, imOut)

Copies 'imIn' image into 'imOut' image.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images. The images must have the same depth and size.

### 2.2.14 copyBitPlane(imIn, plane, imOut)

Inserts or extracts a bit plane. If 'imIn' is a binary image, it is inserted at 'plane' position in greyscale 'imOut'. If 'imIn' is a greyscale image, its bit plane at 'plane' position is extracted and put into binary image 'imOut'.

Plane values are 0 (LSB) to 7 (MSB).

### 2.2.15 copyBytePlane(imIn, plane, imOut)

Inserts or extracts a byte plane. If 'imIn' is a greyscale image, it is inserted at 'plane' position in 32-bit 'imOut'. If 'imIn' is a 32-bit image, its byte plane at 'plane' position is extracted and put into 'imOut'.

Plane values are 0 (LSByte) to 3 (MSByte).

### 2.2.16 copyLine(imIn, nIn, imOut, nOut)

Copies the line numbered 'nIn' of image 'imIn' into 'imOut' at line index 'nOut'.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images. The images must have the same depth and size.

### 2.2.17 cropCopy(imIn, posIn, imOut, posOut, size)

Copies the pixels of 'imIn' in 'imOut' starting from position 'posIn' (tuple x,y) in 'imIn' to position 'posOut' in 'imOut'. The size of the copy is controlled by 'size' (tuple w,h). The actual size will be computed so as not to exceed the images border.

The images must be of the same depth but can have different sizes. Only non binary images are accepted (8-bit or 32-bit).

### 2.2.18 `diff(imIn1, imIn2, imOut)`

Performs a set difference between 'imIn1' and 'imIn2' and puts the result in 'imOut'. The set difference will copy 'imIn1' pixels in 'imOut' if the corresponding pixel in 'imIn2' is lower and will write 0 otherwise:

`imOut = imIn1` if `imIn1 > imIn2` `imOut = 0` otherwise.

'imIn1', 'imIn2' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

### 2.2.19 `diffNeighbor(imIn, imInout, nb, grid=DEFAULT_GRID, edge=EMPTY)`

Performs a set difference operation between the 'imInout' image pixels and their neighbor 'nb' according to 'grid' in image 'imIn'. The result is put in 'imOut'.

'grid' value can be HEXAGONAL or SQUARE. 'edge' value can be EMPTY or FILLED.

If a neighboring point falls outside the image window, its value in the operation is defined by 'edge'. If 'edge' is EMPTY, its value is 0. If 'edge' is FILLED, its value equals the maximal allowed value according to the depth of 'imIn' image.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

### 2.2.20 `divConst(imIn, v, imOut)`

Divides 'imIn' pixel values by value 'v' and puts the result in 'imOut'. The operation can be sum up in the following formula:

`imOut = imIn / v` (or more accurately : `imIn = imOut * v + r`, r being the ignored remainder)

A zero value in 'v' will return an error. For a 8-bit image, v will be restricted between 1 and 255. You cannot use it with binary images.

### 2.2.21 `dualbuildNeighbor(imMask, imInout, d, grid=DEFAULT_GRID)`

Dual builds image 'imInout' in direction 'd' according to 'grid' using 'imMask' as a mask (the propagation is performed only in 'd' direction).

The function also returns the volume of the image 'imInout' after the build operation.

'grid' value can be HEXAGONAL or SQUARE.

### 2.2.22 `extractFrame(imIn, threshold)`

Extracts the smallest frame inside the image 'imIn' that includes all the pixels whose value is greater or equal to 'threshold'.

'imIn' can be a 8-bit or 32-bit image.

### 2.2.23 `generateSupMask(imIn1, imIn2, imOut, strict)`

Generates a binary mask image in 'imOut' where pixels are set to 1 when they are greater (strictly if 'strict' is set to True, greater or equal otherwise) in image 'imIn1' than in image 'imIn2'.

'imIn1' and 'imIn2' can be 1-bit, 8-bit or 32-bit images of same size and depth.

### 2.2.24 `getDirections(grid=DEFAULT_GRID)`

Returns a list containing all the possible directions available in 'grid' (set to DEFAULT\_GRID by default).

If the 'grid' value is incorrect, the function returns an empty list.

### 2.2.25 `getHistogram(imIn)`

Returns a list holding the histogram of the greyscale image 'imIn' (0 to 255).

### 2.2.26 `getImageCounter()`

Returns the number of images actually defined and allocated in the Mamba library. This function may be useful for debugging purposes.

### 2.2.27 `getShowImages()`

Returns the display status ('always\_show' value).

**2.2.28 gridNeighbors(grid=DEFAULT\_GRID)**

Returns the number of neighbors of a point in 'grid' (6 or 8).

If the 'grid' value is incorrect, the function returns 0.

**2.2.29 hierarBuild(imMask, imInout, grid=DEFAULT\_GRID)**

Builds image 'imInout' using 'imMask' as a mask. This function only works with greyscale images and uses a hierarchical list algorithm to compute the result.

'grid' will set the number of neighbors considered by the algorithm (HEXAGONAL is 6-Neighbors and SQUARE is 8-Neighbors).

This function is identical to build but it is faster. However, it works only with greyscale images.

**2.2.30 hierarDualBuild(imMask, imInout, grid=DEFAULT\_GRID)**

Builds (dual build) image 'imInout' using 'imMask' as a mask. This function only works with greyscale images and uses a hierarchical list algorithm to compute the result.

'grid' will set the number of neighbors considered by the algorithm (HEXAGONAL is 6-Neighbors and SQUARE is 8-Neighbors).

This function is identical to build but it is faster. However, it works only with greyscale images.

**2.2.31 hitOrMiss(imIn, imOut, cse0, cse1, grid=DEFAULT\_GRID)**

Performs a binary Hit-or-miss operation on image 'imIn' using the coded structuring elements 'cse0' and 'cse1'. Result is put in 'imOut'.

WARNING! 'imIn' and 'imOut' must be different images.

'grid' value can be HEXAGONAL or SQUARE.

Structuring elements are integer values coding which directions must be taken into account. 'cse0' indicates which neighbor of the current pixel will be checked for a 0 value and 'cse1' those which will be evaluated for a 1 value. A neighbor in direction d is simply coded with the value 2\*\*d. If you want to use multiple directions, just add their codes to obtain the final structuring element coding. The edge is always set to EMPTY (therefore, there is 'edge' argument).

You can also find a helper function (hitormissPatternSelector) in the mambaExtra module.

**2.2.32 infFarNeighbor(imIn, imInout, nb, amp, grid=DEFAULT\_GRID, edge=FILLED)**

Performs a minimum operation between the 'imInout' image pixels and their neighbor 'nb' at distance 'amp' according to 'grid' in image 'imIn'. The result is put in 'imOut'.

'grid' value can be HEXAGONAL or SQUARE. 'edge' value can be EMPTY or FILLED.

If a neighboring point falls outside the image window, its value in the operation is defined by 'edge'. If 'edge' is EMPTY, its value is 0. If 'edge' is FILLED, its value equals the maximal allowed value according to the depth of 'imIn' image.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

**2.2.33 infNeighbor(imIn, imInout, nb, count, grid=DEFAULT\_GRID, edge=FILLED)**

Performs a minimum operation between the 'imInout' image pixels and their neighbor 'nb' according to 'grid' in image 'imIn'. The result is put in 'imOut'.

The operation is repeated 'count' times if the two images 'imIn' and 'imInout' are referring to the same data. Otherwise, 'count' is not taken into account.

'grid' value can be HEXAGONAL or SQUARE. 'edge' value can be EMPTY or FILLED.

If a neighboring point falls outside the image window, its value in the operation is defined by 'edge'. If 'edge' is EMPTY, its value is 0. If 'edge' is FILLED, its value equals the maximal allowed value according to the depth of 'imIn' image.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

### 2.2.34 `label(imIn, imOut, lblow=1, lbhigh=256, grid=DEFAULT_GRID)`

Labels the binary image 'imIn' and puts the result in 32-bit image 'imOut'. Returns the number of connected components found by the labeling algorithm. The labelling will be performed according to the 'grid' (HEXAGONAL is 6-Neighbors and SQUARE is 8-Neighbors).

'lblow' and 'lbhigh' are used to restrain the possible values in the lower byte of 'imOut' pixel values. these values (and all their multiples of 256) are then reserved for another use (see Mamba User Manual for further details).

### 2.2.35 `logic(imIn1, imIn2, imOut, log)`

Performs a logic operation between the pixels of images 'imIn1' and 'imIn2' and put the result in 'imOut'. The logic operation to be performed is indicated through argument 'log'. The allowed logical operations in 'log' are :

"and", "or", "xor", "inf" or "sup".

"and" performs a bitwise AND operation, "or" a bitwise OR and "xor" a bitwise XOR. "inf" calculates the minimum and "sup" the maximum between corresponding pixel values.

'imIn1', 'imIn2' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

### 2.2.36 `lookup(imIn, imOut, lutable)`

Converts the greyscale image 'imIn' using the look-up table 'lutable' and puts the result in greyscale image 'imOut'.

'lutable' is a list containing 256 values with the first one corresponding to 0 and the last one to 255.

### 2.2.37 `mul(imIn1, imIn2, imOut)`

Multiplies 'imIn2' pixel values with 'imIn1' pixel values and put the result in 'imOut'. The operation can be sum up in the following formula :

$$\text{imOut} = \text{imIn1} * \text{imIn2}$$

You can mix formats in the multiply operation (a binary image can be multiplied with a greyscale image, etc...). However you must ensure that the output image is as deep as the deepest of the two input images.

The operation is also saturated for greyscale images (e.g. on a greyscale image  $255 * 255 = 255$ ).

### 2.2.38 `mulConst(imIn, v, imOut)`

Multiplies 'imIn' pixel values with value 'v' and puts the result in 'imOut'. The operation can be sum up in the following formula:

$$\text{imOut} = \text{imIn} * v$$

The operation is saturated for greyscale images. You cannot use it with binary images.

### 2.2.39 `negate(imIn, imOut)`

Negates the image 'imIn' and puts the result in 'imOut'.

The operation is a binary complement for binary images and a negation for greyscale and 32-bit images.

### 2.2.40 `rotateDirection(d, step=1, grid=DEFAULT_GRID)`

Calculates the value of the new direction starting from direction 'd' after 'step' rotations (default value 1). If 'step' is positive, rotations are performed clockwise. They are counterclockwise if 'step' is negative. Calculation is made according to the grid. Direction 0 is taken into account (and always unchanged).

### 2.2.41 `setDefaultGrid(grid)`

This function will change the value of the default grid used in each operator that needs to specify one.

'grid' must be either HEXAGONAL or SQUARE.

You can of course manually change the variable DEFAULT\_GRID by yourself. Using this function is however recommended if you are not sure of what you are doing.

#### 2.2.42 `setImageIndex(index)`

Sets the image index used for naming to a given value 'index'

#### 2.2.43 `setShowImages(showThem)`

Activates automatically the display for new images when 'showThem' is set to True.

#### 2.2.44 `shift(imIn, imOut, d, amp, fill, grid=DEFAULT_GRID)`

Shifts image 'imIn' in direction 'd' of the 'grid' over an amplitude of 'amp'. The emptied space is filled with 'fill' value. The result is put in 'imOut'.

'grid' value can be HEXAGONAL or SQUARE and is set to DEFAULT\_GRID by default.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

#### 2.2.45 `sub(imIn1, imIn2, imOut)`

Subtracts 'imIn2' pixel values to 'imIn1' pixel values and put the result in 'imOut'. The operation can be sum up in the following formula :

$$\text{imOut} = \text{imIn1} - \text{imIn2}$$

You can mix formats in the subtraction operation (a binary image can be subtracted to a greyscale image, etc...). However you must ensure that the output image is as deep as the deepest of the two subtracted images.

The operation is also saturated for grey-scale images (e.g. on a grey scale image 0-1=0) but not for 32-bit images.

#### 2.2.46 `subConst(imIn, v, imOut)`

Subtracts 'v' value to 'imIn' pixel values and puts the result in 'imOut'. The operation can be sum up in the following formula:

$$\text{imOut} = \text{imIn} - v$$

'imIn' and 'imOut' can be 8-bit or 32-bit images of same size and depth.

The operation is saturated (lower limit is 0) for greyscale images.

#### 2.2.47 `supFarNeighbor(imIn, imInout, nb, amp, grid=DEFAULT_GRID, edge=EMPTY)`

Performs a maximum operation between the 'imInout' image pixels and their neighbor 'nb' at distance 'amp' according to 'grid' in image 'imIn'. The result is put in 'imOut'.

'grid' value can be HEXAGONAL or SQUARE. 'edge' value can be EMPTY or FILLED.

If a neighboring point falls outside the image window, its value in the operation is defined by 'edge'. If 'edge' is EMPTY, its value is 0. If 'edge' is FILLED, its value equals the maximal allowed value according to the depth of 'imIn' image.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

#### 2.2.48 `supNeighbor(imIn, imInout, nb, count, grid=DEFAULT_GRID, edge=EMPTY)`

Performs a maximum operation between the 'imInout' image pixels and their neighbor 'nb' according to 'grid' in image 'imIn'. The result is put in 'imOut'.

The operation is repeated 'count' times if the two images 'imIn' and 'imInout' are referring to the same data. Otherwise, 'count' is not taken into account.

'grid' value can be HEXAGONAL or SQUARE. 'edge' value can be EMPTY or FILLED.

If a neighboring point falls outside the image window, its value in the operation is defined by 'edge'. If 'edge' is EMPTY, its value is 0. If 'edge' is FILLED, its value equals the maximal allowed value according to the depth of 'imIn' image.

'imIn' and 'imOut' can be 1-bit, 8-bit or 32-bit images of same size and depth.

#### 2.2.49 `threshold(imIn, imOut, low, high)`

Performs a threshold operation over image 'imIn'. The result is put in binary image 'imOut'.

All the pixels that have a strictly lower value than 'low' or strictly higher than 'high' are set to false. Otherwise they are set to true.

'imIn' can be a 8-bit or 32-bit image.

### 2.2.50 tidyDisplays(*displayer=None*)

Tidies the displayed images. This function will try to optimize, given the actual screen size, the position of the images so that every one may be visible (not always) possible if many images are displayed).

### 2.2.51 transposeDirection(*d*, *grid=DEFAULT\_GRID*)

Calculates the transposed (opposite) direction value of direction 'd' (corresponds to a rotation of gridNeighbors/2 steps).

### 2.2.52 watershedSegment(*imIn*, *imMarker*, *grid=DEFAULT\_GRID*, *max\_level=256*)

Segments greyscale image 'imIn' using the watershed algorithm. 'imMarker' is used both as the marker image (the wells from which the flooding proceeds) and as the output image. It is a 32-bit image. 'max\_level' can be used to limit the flooding process to a specific level (useful if you want to survey the flooding level by level).

'grid' will change the number of neighbors considered by the algorithm (HEXAGONAL is 6-Neighbors and SQUARE is 8-Neighbors).

The result is put inside 'imMarker'. The three first byte planes contain the actual segmentation (each region has a specific label according to the original marker). The last plane represents the actual watershed line (pixels set to 255).

## 3 mambaDraw

This module defines functions to draw inside mamba images. Drawing functions include lines, squares, ... The module also provides functions to extract pixel information.

### 3.1 Functions

#### 3.1.1 drawBox(*imOut*, *square*, *value*)

Draws a box (empty square) in 'imOut' using the tuple 'square' containing 4 values (upper left and down right corners (x1,y1,x2,y2)) using 'value' to set the pixels.

#### 3.1.2 drawCircle(*imOut*, *circle*, *value*)

Draws a circle in 'imOut' using the tuple 'circle' containing 3 values (center and radius (x,y,r)) using 'value' to set the pixels.

#### 3.1.3 drawFillCircle(*imOut*, *circle*, *value*)

Draws a filled circle in 'imOut' using the tuple 'circle' containing 3 values (center and radius (x,y,r)) using 'value' to set the pixels.

#### 3.1.4 drawLine(*imOut*, *line*, *value*)

Draws a line in 'imOut' using the tuple 'line' containing 4 values (starting and ending points (x1,y1,x2,y2)) using 'value' to set the pixels.

This function uses the Bresenham algorithm.

#### 3.1.5 drawSquare(*imOut*, *square*, *value*)

Draws a square in 'imOut' using the tuple 'square' containing 4 values (upper left and down right corners (x1,y1,x2,y2)) using 'value' to set the pixels.

#### 3.1.6 getIntensityAlongLine(*imOut*, *line*)

Returns in a list the intensity profile along a line in 'imOut' using the tuple 'line' containing 4 values (starting and ending points (x1,y1,x2,y2)).

This function uses the Bresenham algorithm.

## 4 mambaExtra

This module defines specific functions and classes which can be considered as extras for the Mamba image library. They provide optional display methods, interactive tools and bridges for exchanging images between the Mamba and PIL libraries.

### 4.1 Functions

#### 4.1.1 Mamba2PIL(imIn)

Creates and returns a PIL image using the Mamba image 'imIn'.

If the mamba image uses a palette, it will be integrated inside the PIL image.

#### 4.1.2 PIL2Mamba(pilim, imOut)

The PIL image 'pilim' is used to load the Mamba image 'imOut'.

#### 4.1.3 dynamicThreshold(imIn)

Opens a separate display in which you can dynamically perform a threshold operation over image 'imIn'.

Once the close button is pressed, the result of the dynamic threshold is returned. This result is a tuple (low, high) used to obtain the image displayed using the threshold operation from mamba.

While the window is opened, you can increase or decrease the low level using keys Q and W respectively. The high level is modified by the keys S (increasing) and X (decreasing).

#### 4.1.4 hitormissPatternSelector(grid=DEFAULT\_GRID)

Helps the user to create patterns for the Hit-or-Miss operator defined in the Mamba module.

The function returns inside a tuple the structuring elements 'es0' and 'es1' (in that order) used as entry in the hitOrMiss function.

You can select the desired grid for the pattern selector. If not specified, the function will use the grid currently in use.

Example with the hitOrMiss function : `hitOrMiss(imIn, imOut, *hitormissPatternSelector())`

#### 4.1.5 mix(imInR, imInG, imInB)

Mixes mamba images 'imInR' (red channel), 'imInG' (green channel) and 'imInB' (blue channel) into a color image. The function returns a PIL image.

#### 4.1.6 split(pilimIn, imOutR, imOutG, imOutB)

Splits a color PIL image 'pilimIn' into its three color channels (Red, Green and Blue) and puts the three resulting images into 'imOutR', 'imOutG' and 'imOutB' respectively.

#### 4.1.7 superpose(imIn1, imIn2)

Draws images 'imIn1' and 'imIn2' in a common display.

If both images are binary, the display is a combination of their pixel values, i.e. black where the pixel is black in both images, blue (default color) if the pixel is set in both images, green (default color) if the pixel is set only in 'imIn1' and red (default color) if it is only set in 'imIn2'.

If one image is greyscale and the other is binary, the binary image is redrawn over the greyscale image in purple (default color).

Image superposition is not possible if both images are greyscale.

The default colors can be changed while displaying by clicking the corresponding color box in the caption above the display window. A color palette will appear where a new color can be selected.

#### 4.1.8 tagOneColorPalette(value, color)

Creates a palette that tags a specific 'value' inside an image with a given 'color', a tuple (red, green, blue), while the rest of the image stays in greyscale.

## 5 `mambaComposed.contrasts`

This module provides a set of functions to perform morphological contrast operators (gradient, top-hat transform,...) using mamba. it works with `imageMb` instances as defined in mamba.

### 5.1 Functions

#### 5.1.1 `blackTopHat(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Performs a black Top Hat operation on 'imIn' and puts the result in 'imOut'. This operator extracts from 'imIn' the dark objects thinner than  $2*n+1$ .

The structuring element used is defined by 'se' ('DEFAULT\_SE' by default).

#### 5.1.2 `gradient(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Computes the morphological gradient of image 'imIn' and puts the result in 'imOut'. The thickness can be controlled using parameter 'n' (1 by default). The structuring element used by the erosion and dilation is defined by 'se' (DEFAULT\_SE by default).

#### 5.1.3 `halfGradient(imIn, imOut, type='intern', n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Computes the half morphological gradient of image 'imIn' and puts the result in 'imOut'.

'type' indicates if the half gradient should be internal or external. Possible values are : "extern" : dilation(imIn) - imIn "intern" : imIn - erosion(imIn)

The thickness can be controlled using parameter 'n' (1 by default). The structuring element used by the erosion or the dilation is defined by 'se'.

#### 5.1.4 `regularisedGradient(imIn, imOut, n, grid=DEFAULT_GRID)`

Computes the regularized gradient of image 'imIn' of size 'n'. The result is put in 'imOut'. A regularized gradient of size 'n' extracts in the image contours thinner than 'n' while avoiding false detections.

This operation is only valid for omnidirectional structuring elements.

#### 5.1.5 `supBlackTopHat(imIn, imOut, n, grid=DEFAULT_GRID)`

Performs a black Top Hat operation with the infimum of directional openings on 'imIn' and puts the result in 'imOut'. This operator partly extracts from 'imIn' the dark objects whose extension in at least one direction of 'grid' is smaller than 'n'.

#### 5.1.6 `supWhiteTopHat(imIn, imOut, n, grid=DEFAULT_GRID)`

Performs a white Top Hat operation with the supremum of directional openings on 'imIn' and puts the result in 'imOut'. This operator partly extracts from 'imIn' the bright objects whose extension in at least one direction of 'grid' is smaller than 'n'.

#### 5.1.7 `whiteTopHat(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Performs a white Top Hat operation on 'imIn' and puts the result in 'imOut'. This operator extracts from 'imIn' the bright objects thinner than  $2*n+1$ .

The structuring element used is defined by 'se' ('DEFAULT\_SE' by default).

## 6 `mambaComposed.erodil`

This module provides a set of functions to perform erosions and dilations using Mamba base functions. It works with `imageMb` instances as defined in Mamba.

## 6.1 Classes

### 6.1.1 structuringElement

This class allows to define simple structuring elements with points belonging to the elementary neighborhood of the origin point. Points in use are defined by their direction, according to the grid in use (hexagonal or square one). All the used directions are put in a in a direction list. The central point (direction 0) may or may not belong to the structuring element.

Example:

```
>>>HEXAGON = structuringElement ([0,1,2,3,4,5,6], mamba.HEXAGONAL)
```

defines a structuring element named HEXAGON on the hexagonal grid and containing the origin point and all the six neighboring points in directions 1 to 6 of the grid.

`__cmp__(self, otherSE)`

`__init__(self, directions, grid)` Structuring element constructor. A structuring element is defined by the couple 'directions' (given in an ordered list) and 'grid'. You cannot defines a structuring element that holds a direction more than once.

You can look at the predefined structuring elements to get examples of how to make yours.

`__repr__(self)`

`getDirections(self, withoutZero=False)` Returns a copy of the directions used by the structuring element. if 'withoutZero' is set to True the returned direction list will not include direction 0 (useful for some operators, such as erode or dilate, where direction 0 modifies the initial conditions).

Example:

```
>>>SQUARE3X3.getDirections()
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

`getGrid(self)` Returns the grid associated with the structuring element.

Example:

```
>>>HEXAGON.getGrid()
HEXAGONAL
```

`hasZero(self)` Returns True if the central point (0) is included in the direction list.

`rotate(self, step=1)` Rotates the structuring element 'step' times. When step is positive, rotation is clockwise. When step is negative, rotation is counterclockwise. When step is equal to zero, there is no rotation.

Example:

```
>>>SEGMENT.getDirections()
[0, 1]
>>>SEGMENT.rotate().getDirections()
[0, 2]
```

`setAs(self, se)` Copies the attributes (directions, grid) of 'structuringElement' into the structuring element which this method is applied to. This method is mainly used to modify the default structuring element.

Example:

```
>>>DEFAULT_SE.setAs(SQUARE3X3)
```

modifies the default structuring element to a square one. The erosions and dilations now will be performed with a square.

Warning! Although it is perfectly allowed, it is not wise to use setAs method with other pre-defined structuring elements. For instance, if you type:

```
>>>HEXAGON.setAs(SQUARE3X3)
```

the structuring element HEXAGON will be superseded by SQUARE3X3. This may have unwanted consequences.

**transpose(self)** Structuring element transposition (symmetry around the origin). Basically, it corresponds to a 3-steps rotation on an hexagonal grid, a 4-steps on a square one.

Example:

```
>>>TRIANGLE. getDirections ()
[0, 3, 4]
>>>TRIANGLE. transpose (). getDirections ()
[0, 1, 6]
```

## 6.2 Functions

### 6.2.1 conjugateHexagonalDilate(imIn, imOut, size, edge=EMPTY)

Dilation by a conjugate hexagon (hexagon turned by 30 degrees). Be aware that the size of operation corresponds to twice the size of the regular hexagon: a conjugate hexagon of size 1 is inscribed in a regular hexagon of size 2.

### 6.2.2 conjugateHexagonalErode(imIn, imOut, size, edge=FILLED)

Erosion by a conjugate hexagon (hexagon turned by 30 degrees).

Be aware that the size of operation corresponds to twice the size of the regular hexagon: a conjugate hexagon of size 1 is inscribed in a regular hexagon of size 2.

### 6.2.3 dilate(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=EMPTY)

This operator performs a dilation, using the structuring element 'se' (set by default as DEFAULT\_SE), of image 'imIn' and puts the result in 'imOut'. The operation is repeated 'n' times (default is 1). This operator assumes an 'EMPTY' edge by default.

This operator always considers that the origin of the structuring element in use is at position 0 even if this point does not belong to it.

### 6.2.4 dodecagonalDilate(imIn, imOut, size, edge=EMPTY)

Dilation by a dodecagon (hexagonal grid). This operation is the result of an hexagonal dilation followed by a dilation by a conjugate hexagon. The respective sizes of the hexagon and of the conjugate hexagon are calculated in order that the final dodecagon be as isotropic as possible.

### 6.2.5 dodecagonalErode(imIn, imOut, size, edge=FILLED)

Erosion by a dodecagon (hexagonal grid). This operation is the result of an hexagonal erosion followed by an erosion by a conjugate hexagon. The respective sizes of the hexagon and of the conjugate hexagon are calculated in order that the final dodecagon be as isotropic as possible.

### 6.2.6 doublePointDilate(imIn, imOut, d, n, grid=DEFAULT\_GRID, edge=EMPTY)

This operator performs a dilation of 'imIn' using a double point as a structuring element. To build the double point, the first point is considered in position (0,0) and the second is built using a shift 'n' times in the direction 'd' + 180 degrees (transposed direction) of 'grid'. The result is put in imOut. The direction d is selected according to the grid in use (DEFAULT\_GRID).

Note that this operator is just an alias of the operator supFarNeighbor.

### 6.2.7 doublePointErode(imIn, imOut, d, n, grid=DEFAULT\_GRID, edge=FILLED)

This operation performs an erosion of 'imIn' using a double point as a structuring element. To build the double point, the first point is considered in position (0,0) and the second is built using a shift 'n' times in the conjugate direction 'd' + 180 degrees of 'grid'. The result is put in imOut.

This operator is an alias of the infFarNeighbor operator.

### 6.2.8 `erode(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED)`

This operator corresponds, for erosion, to the dilation operator. It performs an erosion using the default structuring element of image 'imIn' and puts the result in 'imOut'. The operation is repeated 'n' times (default is 1). This function will assume a FILLED edge unless specified otherwise using 'edge'.

This operator always considers that the origin of the structuring element in use is at position 0 even if this point does not belong to it.

### 6.2.9 `linearDilate(imIn, imOut, d, n=1, grid=DEFAULT_GRID, edge=EMPTY)`

Dilation by a segment in direction 'd' of image 'imIn', result in 'imOut'. The operation is repeated 'n' times (default is 1). This function will assume an EMPTY edge unless specified otherwise using 'edge'. The directions are defined according to the grid in use.

### 6.2.10 `linearErode(imIn, imOut, d, n=1, grid=DEFAULT_GRID, edge=FILLED)`

Performs an erosion in direction 'd' of image 'imIn' and puts the result in 'imOut'. The operation is repeated 'n' times (default is 1). This function will assume a FILLED edge unless specified otherwise using 'edge'.

### 6.2.11 `octogonalDilate(imIn, imOut, size, edge=EMPTY)`

Dilation by an octogon (square grid). This operation is the result of a dilation by a square followed by a dilation by a diamond. The respective sizes of the square and of the diamond are calculated in order that the final octogon be as isotropic as possible.

### 6.2.12 `octogonalErode(imIn, imOut, size, edge=FILLED)`

Erosion by an octogon (square grid). This operation is the result of an erosion by a square followed by an erosion by a diamond (conjugate square). The respective sizes of the square and of the diamond are calculated in order that the final octogon be as isotropic as possible.

## 7 `mambaComposed.erodilLarge`

This module provides a set of functions performing erosions and dilations with large structuring elements. They are built with special shift operators written in C, together with special 'infFarNeighbor' and 'supFarNeighbor' functions.

### 7.1 Functions

#### 7.1.1 `largeDodecagonalDilate(imIn, imOut, size, edge=EMPTY)`

Dilation by large dodecacagons (hexagonal grid). Basically, it is the same operation as the previous one where classical dilations have been replaced by dilations by large structuring elements, and where a "partial" dilation by a conjugate hexagon is used.

#### 7.1.2 `largeDodecagonalErode(imIn, imOut, size, edge=FILLED)`

Erosion by large dodecacagons (hexagonal grid). Basically, it is the same operation as the previous one where classical erosions have been replaced by erosions by large structuring elements, and where a "partial" erosion by a conjugate hexagon is used.

#### 7.1.3 `largeHexagonalDilate(imIn, imOut, size, edge=EMPTY)`

Dilation by large hexagons using dilations by large segments and the Steiner decomposition property of the hexagon. Edge effects are corrected by dilations with transposed decompositions combined with sup operators.

This operator is quite complex to avoid edge effects.

#### 7.1.4 `largeHexagonalErode(imIn, imOut, size, edge=FILLED)`

Erosion by large hexagons using erosions by large segments and the Steiner decomposition property of the hexagon. Edge effects are corrected by erosions with transposed decompositions combined with inf operations (see documentation for further details).

This operator is quite complex to avoid edge effects.

#### 7.1.5 `largeLinearDilate(imIn, imOut, dir, size, grid=DEFAULT_GRID, edge=EMPTY)`

Dilation by a large segment in direction 'dir' in a reduced number of iterations. Uses the dilations by doublets of points (supposed to be faster, thanks to an enhanced shift operator).

#### 7.1.6 `largeLinearErode(imIn, imOut, dir, size, grid=DEFAULT_GRID, edge=FILLED)`

Erosion by a large segment in direction 'dir' in a reduced number of iterations. Uses the erosions by doublets of points (supposed to be faster, thanks to an enhanced shift operator).

#### 7.1.7 `largeOctogonalDilate(imIn, imOut, size, edge=EMPTY)`

Dilation by a large octogon (square grid). This operation uses dilations by large squares and large diamonds previously defined.

#### 7.1.8 `largeOctogonalErode(imIn, imOut, size, edge=FILLED)`

Erosion by a large octogon (square grid). This operation uses erosions by large squares and large diamonds previously defined.

#### 7.1.9 `largeSquareDilate(imIn, imOut, size, edge=EMPTY)`

Dilation by a large square using dilations by large segments and the Steiner decomposition property of the square.

No edge effects are likely to happen with a square structuring element.

#### 7.1.10 `largeSquareErode(imIn, imOut, size, edge=FILLED)`

Erosion by a large square using erosions by large segments and the Steiner decomposition property of the square.

No edge effects are likely to happen with a square structuring element.

## 8 `mambaComposed.filter`

This module provides a set of functions to perform morphological filtering operations using mamba. it works with imageMb instances as defined in mamba.

### 8.1 Functions

#### 8.1.1 `alternateFilter(imIn, imOut, n, openFirst, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Performs an alternate filter operation of size 'n' on image 'imIn' and puts the result in 'imOut'. If 'openFirst' is True, the filter begins with an opening, a closing otherwise.

#### 8.1.2 `autoMedian(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Morphological automedian filter performed with alternate sequential filters.

#### 8.1.3 `fullAlternateFilter(imIn, imOut, n, openFirst, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

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.

#### 8.1.4 `linearAlternateFilter(imIn, imOut, n, openFirst, grid=DEFAULT_GRID)`

Performs an alternate filter operation on image 'imIn' with openings and closings by segments of size 'n' (supremum of openings and infimum of closings) and puts the result in 'imOut'. If 'openFirst' is True, the filter begins with an opening, a closing otherwise.

#### 8.1.5 `simpleLevelling(imIn, imMask, imOut, grid=DEFAULT_GRID)`

Performs a simple levelling of image 'imIn' controlled by image 'imMask' and puts the result in 'imOut'. This operation is composed of two geodesic reconstructions. This filter tends to level regions in the image of homogeneous grey values.

#### 8.1.6 `strongLevelling(imIn, imOut, n, eroFirst, grid=DEFAULT_GRID)`

Strong levelling of 'imIn', result in 'imOut'. 'n' defines the size of the erosion and dilation of 'imIn' in the operation. If 'eroFirst' is true, the operation starts with an erosion, it starts with a dilation otherwise.

This filter is stronger (more efficient) than simpleLevelling. However, the order of the initial operations (erosion and dilation) matters.

## 9 `mambaComposed.geodesy`

This module provides a set of functions to perform geodesic computations using Mamba based functions. It includes build and dualbuild operations, geodesic erosion and dilation, computation of maxima and minima... it works with imageMb instances as defined in mamba.

### 9.1 Functions

#### 9.1.1 `build(imMask, imInout, grid=DEFAULT_GRID)`

Builds image 'imInout' using 'imMask' as a mask. This operator performs the geodesic reconstruction of 'imInout' inside the mask image and puts the result in the same image.

This operator uses a recursive implementation of the reconstruction.

This function will use the mamba default grid unless specified otherwise in 'grid'.

#### 9.1.2 `closeHoles(imIn, imOut, grid=DEFAULT_GRID)`

Close holes in image 'imIn' and puts the result in 'imOut'. This operator works on binary and greytone images. In this case, however, it should be used cautiously.

#### 9.1.3 `dualBuild(imMask, imInout, grid=DEFAULT_GRID)`

Builds (dual build) image 'imInout' using 'imMask' as a mask. This operator performs the geodesic dual reconstruction (by erosions) of 'imInout' inside the mask image and puts the result in the same image.

This operator uses a recursive implementation of the reconstruction.

This function will use the mamba default grid unless specified otherwise in 'grid'.

#### 9.1.4 `geodesicDilate(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Performs a geodesic dilation of image 'imIn' inside 'imMask'. The result is put inside 'imOut', 'n' controls the size of the dilation. 'se' specifies the type of structuring element used to perform the computation (DEFAULT\_SE by default).

This transformation works for binary and greytone images.

Warning! 'imMask' and 'imOut' must be different.

#### 9.1.5 `geodesicDistance(imIn, imMask, imOut, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Computes the geodesic distance function of a set in 'imIn'. This distance function uses successive geodesic erosions of 'imIn' performed in the geodesic space defined by 'imMask'. The result is stored in 'imOut'. Be sure to use an image of sufficient depth as output.

This geodesic distance is quite slow as it is performed by successive geodesic erosions.

### 9.1.6 `geodesicErode(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))`

Performs a geodesic erosion of image 'imIn' inside 'imMask'. The result is put inside 'imOut', 'n' controls the size of the erosion. 'se' specifies the type of structuring element used to perform the computation (DEFAULT\_SE by default).

The geodesic erosion is realised using the fact that the dilation is the dual operation of the erosion.

Warning! 'imMask' and 'imOut' must be different.

Note that this operation is different for binary and greytone images.

### 9.1.7 `maxima(imIn, imOut, h=1, grid=DEFAULT_GRID)`

Computes the maxima of 'imIn' using a build operation and puts the result in 'imOut'.

'h' can be used to define the maxima height. Grid used by the build operation can be specified by 'grid'.

Only works with greyscale images as input. 'imOut' must be binary.

### 9.1.8 `minima(imIn, imOut, h=1, grid=DEFAULT_GRID)`

Computes the 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. 'imOut' must be binary.

### 9.1.9 `removeEdgeParticles(imIn, imOut, grid=DEFAULT_GRID)`

Removes particles (connected components) touching the edge in image 'imIn'. The resulting image is put in image 'imOut'. Although this operator may be used with greytone images, it should be considered with caution.

## 10 `mambaComposed.measure`

This module provides a set of functions which perform measure operations on mamba images. It works with `imageMb` instances as defined in `mamba`.

### 10.1 Functions

#### 10.1.1 `computeArea(imIn, scale=(1.0, 1.0))`

Calculates the area of the binary image 'imIn'. 'scale' is a tuple containing the horizontal scale factor (distance between two adjacent horizontal points) and the vertical scale factor (distance between two successive lines) of image 'imIn' (default is 1.0 for both). The result is a float (when default values are used, the result value is identical to the `computeVolume` operator).

Note that, with hexagonal grid, the "scale" default values do not correspond to an isotropic grid (where triangles would be equilateral).

Beware, if the input image 'imIn' is not a binary image, the function raises an error.

#### 10.1.2 `computeComponentsNumber(imIn, grid=DEFAULT_GRID)`

Computes the number of connected components in image 'imIn'. The result is an integer value.

#### 10.1.3 `computeConnectivityNumber(imIn, grid=DEFAULT_GRID)`

Computes the connectivity number (Euler-Poincare constant) of image 'imIn'. The result is an integer number.

Beware, if the input image 'imIn' is not a binary image, the function raises an error.

#### 10.1.4 `computeDiameter(imIn, dir, scale=(1.0, 1.0), grid=DEFAULT_GRID)`

Computes the diameter (diametral variation) of binary image 'imIn' in direction 'dir'. 'scale' is a tuple defining the horizontal and vertical scale factors (default is 1.0).

Beware, if the input image 'imIn' is not a binary image, the function raises an error.

### 10.1.5 `computeFeretDiameters(imIn, scale=(1.0, 1.0))`

computes the global Feret diameters (horizontal and vertical) of binary image 'imIn' and returns the result in a tuple (hDf, vDf). These diameters correspond to the horizontal and vertical dimensions of the smallest bonding box containing all the particles of 'imIn'

### 10.1.6 `computePerimeter(imIn, scale=(1.0, 1.0), grid=DEFAULT_GRID)`

Computes the perimeter of all particles in binary image 'imIn' according to the Cauchy-Crofton formula. 'scale' is a tuple defining the horizontal and vertical scale factors (default is 1.0).

The edge of the image is always set to 'EMPTY'.

Beware, if the input image 'imIn' is not a binary image, the function raises an error.

## 11 `mambaComposed.miscellaneous`

This module regroups functions/operators that could not be regrouped with other operators because of their unique nature or other peculiarity. As such, it regroups some utility functions.

### 11.1 Functions

#### 11.1.1 `ceilingAdd(imIn1, imIn2, imOut)`

Adds image 'imIn2' to image 'imIn1' and puts the result in 'imOut'. If  $imIn1 + imIn2$  is larger than the maximal possible value in imOut, the result is truncated and limited to this maximal value.

Although it is possible to use a 8-bit image for imIn2, it is recommended to use the same depth for all the images.

Note that this operator is mainly useful for 32-bit images, as the result of the addition is always truncated for 8-bit images.

#### 11.1.2 `ceilingAddConst(imIn, v, imOut)`

Adds a constant value 'v' to image 'imIn' and puts the result in 'imOut'. If  $imIn + v$  is larger than the maximal possible value in imOut, the result is truncated and limited to this maximal value.

Note that this operator is mainly useful for 32-bit images, as the result of the addition is always truncated for 8-bit images.

#### 11.1.3 `drawEdge(imOut, thick=1)`

Draws a frame around the edge of 'imOut' whose value equals the maximum range value and whose thickness is given by 'thick' (default 1).

#### 11.1.4 `floorSub(imIn1, imIn2, imOut)`

subtracts image 'imIn2' from image 'imIn1' and puts the result in 'imOut'. If  $imIn1 - imIn2$  is negative, the result is truncated and limited to 0.

Although it is possible to use a 8-bit image for imIn2, it is recommended to use the same depth for all the images.

Note that this operator is mainly useful for 32-bit images, as the result of the subtraction is always truncated for 8-bit images.

#### 11.1.5 `floorSubConst(imIn, v, imOut)`

Subtracts a constant value 'v' to image 'imIn' and puts the result in 'imOut'. If  $imIn - v$  is negative, the result is truncated and limited to 0.

Note that this operator is mainly useful for 32-bit images, as the result of the subtraction is always truncated for 8-bit images.

### 11.1.6 isotropicDistance(imIn, imOut, edge=FILLED)

Computes the distance function of a set in 'imIn'. This distance function uses dodecagonal erosions and the grid is assumed to be hexagonal. The procedure is quite slow but the result is more aesthetic. This operator also illustrates how to perform successive dodecagonal operations of increasing sizes.

### 11.1.7 translate(imIn, imOut, deltaX, deltaY, v=0)

Performs the translation of image 'imIn' by a vector ('deltaX', 'deltaY') and puts the result in 'imOut'. It is possible to fill the void regions of the translated image with the value 'v' (default 0).

## 12 mambaComposed.openclose

This module provides a set of functions to perform opening and closing operations using mamba. it works with imageMb instances as defined in mamba. All the closing and opening operation defined in this module use erosion, dilation and build functions with user-defined edge settings. The functions define a default edge which can be changed (see the modules erodil and geodesy).

### 12.1 Functions

#### 12.1.1 buildClose(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))

Performs a closing by dual reconstruction operation on image 'imIn' and puts the result in 'imOut'. 'n' controls the size of the closing.

#### 12.1.2 buildOpen(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL))

Performs an opening by reconstruction operation on image 'imIn' and puts the result in 'imOut'. 'n' controls the size of the opening.

#### 12.1.3 close(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED)

Performs a closing operation on image 'imIn' and puts the result in 'imOut'. 'n' controls the size of the closing and 'se' the structuring element used.

The default edge is set to 'FILLED'. If 'edge' is set to 'EMPTY', the operation is slightly modified to avoid errors (non extensivity).

#### 12.1.4 infClose(imIn, imOut, n, grid=DEFAULT\_GRID)

Performs the infimum of directional closings. A black particle is preserved if its length is larger than 'n' in at least one direction.

This operator is a closing. The image edge is set to 'FILLED' in order to take into account particles touching the edge (they are supposed not to extend outside the image window).

When square grid is used, the size in oblique directions are reduced to be similar to the horizontal and vertical size.

#### 12.1.5 linearClose(imIn, imOut, dir, n, grid=DEFAULT\_GRID, edge=FILLED)

Performs a closing by a segment of size 'n' in direction 'dir'.

If 'edge' is set to 'EMPTY', the operation must be modified to remain extensive.

#### 12.1.6 linearOpen(imIn, imOut, dir, n, grid=DEFAULT\_GRID, edge=FILLED)

Performs an opening by a segment of size 'n' in direction 'dir'.

'edge' is set to 'FILLED' by default.

### 12.1.7 `open(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED)`

Performs an opening operation on image 'imIn' and puts the result in 'imOut'. 'n' controls the size of the opening and 'se' the structuring element used.

The default edge is set to 'FILLED'. Note that the edge setting operates in the erosion only.

### 12.1.8 `supOpen(imIn, imOut, n, grid=DEFAULT_GRID)`

Performs the supremum of directional openings. A white particle is preserved (but not entirely) if its length is larger than 'n' in at least one direction.

This operator is an opening. The image edge is set to 'EMPTY' in order to take into account particles touching the edge (they are considered as entirely included in the image window).

When square grid is used, the size in oblique directions are reduced to be similar to the horizontal and vertical size.

## 13 `mambaComposed.residues`

This module provides a set of functions to perform residual operations using mamba. It works with `imageMb` instances as defined in mamba. A residual transformation is built by subtracting two sequences of primitive operators to get residues and by computing the supremum of these residues. The position in the sequence where this maximum occurs is also computed (it is called associated function and is generally a 32-bit image). These residues are defined on binary and greytone images.

### 13.1 Functions

#### 13.1.1 `binarySkeletonByOpening(imIn, imOut1, imOut2, grid=DEFAULT_GRID, edge=FILLED)`

Skeleton by openings (maximal balls skeleton) of binary image 'imIn'. 'imOut1' contains the skeleton points (centers of maximal balls) and 'imOut2' contains the associated function (that is the radius of each maximal ball included in the initial set).

The operation is fast because it is computed through the use of the distance function of 'imIn' (skeleton points can be obtained by a Top Hat transform on the distance function).

The edge is set to 'FILLED' by default.

#### 13.1.2 `binaryUltimateErosion(imIn, imOut1, imOut2, grid=DEFAULT_GRID, edge=FILLED)`

Ultimate erosion of binary image 'imIn'. 'imOut1' contains the ultimate eroded set and 'imOut2' contains the associated function (that is the height of each connected component of the ultimate erosion).

An ultimate erosion is composed of the union of the last connected components of the successive erosions of the initial set. The associated function provides the size of the corresponding erosion.

Depth of 'imOut1' is 1, depth of 'imOut2' is 32.

The operation is fast because it is computed using the distance function of 'imIn' (the ultimate erosion is identical to the maxima of this distance function).

The edge is set to 'FILLED' by default.

#### 13.1.3 `fullRegularisedGradient(imIn, imOut1, imOut2, grid=DEFAULT_GRID)`

Full regularised morphological gradient of image 'imIn'. This operator is a residual transform which uses the regularised gradient of size *i* as a residue. The range of sizes *i* is limited to 16, as beyond this value, the residue is most of the time equal to 0.

Warning! 'imOut2' is a greyscale image (depth equal to 8).

#### 13.1.4 `quasiDistance(imIn, imOut1, imOut2, grid=DEFAULT_GRID)`

Quasi-distance of image 'imIn'. 'imOut1' contains the residues image and 'imOut2' contains the quasi-distance (associated function).

The quasi-distance of a greytone image is made of a patch of distance functions of some almost flat regions in the image. When the image is a simple indicator function of a set, the quasi-distance and the distance function are identical.

Depth of 'imOut1' is the same as 'imIn', depth of 'imOut2' is 32.

### 13.1.5 skeletonByOpening(imIn, imOut1, imOut2, grid=DEFAULT\_GRID)

General skeleton by openings working on greytone image 'imIn'. 'imOut1' contains the skeleton function and 'imOut2' contains the associated function.

This skeleton corresponds to the centers of maximal cylinders included in the set under the graph of the image 'imIn'.

Depth of 'imOut1' is the same as 'imIn', depth of 'imOut2' is 32.

The edge is always set to 'FILLED'.

### 13.1.6 ultimateBuildOpening(imIn, imOut1, imOut2, grid=DEFAULT\_GRID)

Ultimate opening by build of image 'imIn'. 'imOut1' contains the ultimate opening whereas 'imOut2' contains the granulometric function.

This ultimate opening is obtained by using successive openings by build.

Depth of 'imOut1' is the same as 'imIn', depth of 'imOut2' is 32.

### 13.1.7 ultimateErosion(imIn, imOut1, imOut2, grid=DEFAULT\_GRID)

General ultimate erosion working on greytone image 'imIn'. 'imOut1' contains the ultimate eroded function and 'imOut2' contains the associated function.

This ultimate erosion can be applied to greytone images.

Depth of 'imOut1' is the same as 'imIn', depth of 'imOut2' is 32.

The edge is always set to 'FILLED'.

### 13.1.8 ultimateIsotropicOpening(imIn, imOut1, imOut2, grid=DEFAULT\_GRID)

Ultimate opening of image 'imIn' with more isotropic structuring elements. Dodecagons are used on hexagonal grid, octogons on square grid. 'imOut1' contains the ultimate opening whereas 'imOut2' contains the granulometric function.

Depth of 'imOut1' is the same as 'imIn', depth of 'imOut2' is 32.

### 13.1.9 ultimateOpening(imIn, imOut1, imOut2, grid=DEFAULT\_GRID)

Ultimate opening of image 'imIn'. 'imOut1' contains the ultimate opening whereas 'imOut2' contains the granulometric function.

Ultimate opening is obtained by using successive openings by hexagons or squares as primitive functions depending of the grid in use.

Depth of 'imOut1' is the same as 'imIn', depth of 'imOut2' is 32.

## 14 mambaComposed.segment

This module provides a set of functions to perform segmentation operations using mamba. it works with imageMb instance as defined in mamba.

### 14.1 Functions

#### 14.1.1 fastSKIZ(imIn, imOut, grid=DEFAULT\_GRID)

Fast skeleton by zones of influence of binary image 'imIn'. Result is put in binary image 'imOut'. The transformation is faster as it uses the watershed transform by hierarchical queues.

#### 14.1.2 geodesicSKIZ(imIn, imMask, imOut, grid=DEFAULT\_GRID)

Geodesic skeleton by zones of influence of binary image 'imIn' inside the geodesic mask 'imMask'. The result is in binary image 'imOut'.

### 14.1.3 markerControlledWatershed(imIn, imMarkers, imOut, grid=DEFAULT\_GRID)

Marker-controlled watershed transform of greytone image 'imIn'. The binary image 'imMarkers' contains the markers which control the flooding process. 'imOut' contains the valued watershed.

### 14.1.4 mosaic(imIn, imOut, imWts, grid=DEFAULT\_GRID)

Builds the mosaic image of 'imIn' and puts the results into 'imOut'. The watershed line (pixel values set to 255) is stored in the greytone image 'imWts'. A mosaic image is a simple image made of various tiles of uniform grey values. It is built using the watershed of 'imIn' gradient and original markers made of gradient minima which are labelled by the maximum value of 'imIn' pixels inside them.

### 14.1.5 mosaicGradient(imIn, imOut, grid=DEFAULT\_GRID)

Builds the mosaic-gradient image of 'imIn' and puts the result in 'imOut'. The mosaic-gradient image is built by computing the differences of two mosaic images generated from 'imIn', the first one having its watershed lines valued by the suprema of the adjacent catchment basins values, the second one been valued by the infima.

### 14.1.6 valuedWatershed(imIn, imOut, grid=DEFAULT\_GRID)

Returns the valued watershed of greyscale image 'imIn' into greyscale image 'imOut'. Each pixel of the watershed lines is given its corresponding value in initial image 'imIn'.

## 15 mambaComposed.sequence

This module provides classes, methods and functions to perform morphological computations over sequences of images using mamba. This module can be considered as a restricted 3-D extension of mamba.

### 15.1 Classes

#### 15.1.1 sequenceMb

A sequence of images is represented by an instance of this class.

**\_\_getitem\_\_ (self, key)** Handles direct acces to the image inside the sequence.

**\_\_init\_\_ (self, \*args, \*\*kwargs)** Constructor for a mamba image sequence. A sequence is defined by its images width and height and by its length (the number of images in it).

There is a wide range of possibilities :

- `sequenceMb()` : without arguments will create an empty greyscale image sequence.
- `sequenceMb(seq)` : will create a sequence using the same size, depth and length than sequence 'seq'.
- `sequenceMb(im)` : will create a sequence using the same size and depth than image 'im'.
- `sequenceMb(depth)` : will create an image sequence with the desired 'depth' (1, 8 or 32) for the mamba images.
- `sequenceMb(path)` : will load the image sequence located in 'path', see the load method.
- `sequenceMb(seq, depth)` : will create a sequence using the same size than sequence 'seq' and the specified 'depth'.

- `sequenceMb(im, length)` : will create a sequence using the same size than image 'im' and the specified 'length'.
- `sequenceMb(path, depth)` : will load the image sequence located in 'path' and convert it to the specified 'depth'.
- `sequenceMb(width, height, length)` : will create an image sequence with size 'width'x'height' and 'length'.
- `sequenceMb(width, height, length, depth)` : will create an image sequence with size 'width'x'height', 'depth' and 'length'.

When not specified, the width, height and length of the sequence will be set to 256. The default depth is 8 (greyscale).

When loading an image sequence make sure all the images have the same size.

**`__iter__`(self)** Makes a mamba image sequence iterable.

**`fill`(self, v)** Fills all the images in the sequence with value 'v' A zero value makes the image completely dark.

**`getDepth`(self)** Returns the depth of the sequence.

**`getLength`(self)** Returns the length of the sequence.

**`getName`(self)** Returns the name of the sequence.

**`getSize`(self)** Returns the size (tuple with width, height) of the sequence.

**`hideAllImages`(self)** Deactivates the image display for all the images in the sequence.

**`hideImage`(self, index)** Deactivates the image display for image at 'index' in the sequence.

**`load`(self, path, rgbfilter=None)** Loads a sequence of image as found in directory 'path'.

To be valid a sequence of images must be composed of at least 'length' images to be able to fill the sequence. Their file names must be of the form XXX.ext where XXX is a number on three digits and ext is an image file extension (like jpg or png). The sequence will be read in increasing order (001 then 002 and so on). However it is not mandatory that the numbers follow each other (001 then 005 is legal). You can also mix image formats (001.bmp then 002.jpg is legal) but you should make sure that files that are not images (txt, pdf, ...) are not named following that pattern.

**`next`(self)** The next method for the iteration.

**`reset`(self)** Reset the sequence (all the pixels are put to 0).

**`resetPalette`(self)** Undefined the palette to use to convert the images in color for display and save. The images will be grey scale.

**`setPalette`(self, pal)** Defines the palette to use to convert the images in color for display and save. Apply to all the images in the sequence.

**`showAllImages`(self)** Activates the image display for all the images in the sequence.

**`showImage`(self, index)** Activates the image display for image at 'index' in the sequence.

## 15.2 Functions

### 15.2.1 `closeByCylinderSequence(sequence, height, section)`

Closing using the dilation and erosion by a cylinder.

### 15.2.2 `copySequence(sequenceIn, sequenceOut)`

Copies the content of 'sequenceIn' into 'sequenceOut'. The copy is stopped by the smallest sequence.

### 15.2.3 `dilateByCylinderSequence(sequence, height, section)`

Dilates the 'sequence' using a cylinder with an hexagonal section of size  $2x'section'$  and a height of  $2x'height'$ . The sequence is modified by this function.

### 15.2.4 `erodeByCylinderSequence(sequence, height, section)`

Erodes the 'sequence' using a cylinder with an hexagonal section of size  $2x'section'$  and a height of  $2x'height'$ . The sequence is modified by this function.

### 15.2.5 `openByCylinderSequence(sequence, height, section)`

Opening using the dilation and erosion by a cylinder.

## 16 `mambaComposed.statistic`

This module provides a set of functions to compute statistical values inside a mamba image.

### 16.1 Functions

#### 16.1.1 `getMean(imIn)`

Returns the average value (float) of the pixels of 'imIn' (which must be a greyscale image).

#### 16.1.2 `getMedian(imIn)`

Returns the median value of the pixels of 'imIn'.

The median value is defined as the first pixel value for which at least half of the pixels are below it. 'imIn' must be a greyscale image.

#### 16.1.3 `getVariance(imIn)`

Returns the pixels variance (estimator without bias) of image 'imIn' (which must be a greyscale image)..

## 17 `mambaComposed.thinthick`

This module contains morphological Hit-or-Miss, thinning and thickening operators of the Mamba Image library, together with various homotopic and geodesic functions derived from these operators. They use `imageMb` image instances as defined in the `mamba` module.

### 17.1 Classes

#### 17.1.1 `doubleStructuringElement`

This class allows to define a doublet of structuring elements used in a coded format by Hit-or-Miss, thin and thick operations and their corresponding methods. The coding corresponds to the output of the 'hitormissPatternSelector' tool available in `mambaExtra` module.

**\_\_init\_\_(self, \*args)** Double structuring element constructor. A double structuring element is defined by the first (background points) and second (foreground points) structuring elements.

You can define it in two ways:

- `doubleStructuringElement(se0, se1)`: where 'se0' and 'se1' are

instances of the class structuring element found in `erodil` module. These structuring elements must be defined on the same grid.

- `doubleStructuringElement(dse0, dse1, grid)`: where 'dse0' and

'dse1' are direction lists and 'grid' defines the grid on which the two structuring elements are defined.

If the constructor is called with inappropriate arguments, it raises a `ValueError` exception.

**\_\_repr\_\_(self)**

**flip(self)** Flips the doublet of structuring elements. Flipping corresponds to a swap: the doublet (se0, se1) becomes (se1, se0).

**getCSE(self)** Returns the coded values corresponding to the background and foreground structuring elements se0 and se1 in a tuple so they can be used with the `HitOrMiss` function.

**getGrid(self)** Returns the grid on which the double structuring element is defined.

**getStructuringElement(self, ground)** Returns the structuring element of the foreground if 'ground' is set to 1 or the structuring element of the background otherwise.

**rotate(self, step=1)** Rotates the double structuring element 'step' times (default=1). When 'step' is positive, rotation is clockwise. When 'step' is negative, rotation is counterclockwise. No rotation occurs when 'step' equals zero.

## 17.2 Functions

### 17.2.1 `binaryHMT(imIn, imOut, dse, edge=EMPTY)`

This operator is similar to `mamba.hitOrMiss`, except that it takes into account the 'edge' configuration (in `mamba.hitOrMiss`, 'edge' is always empty) and that it uses the `doubleStructuringElement` class to define 'dse'. It also allows that 'imIn' and 'imOut' be identical (it is not allowed with `hitOrMiss`).

'imIn' and 'imOut' are binary images.

'edge' is `EMPTY` by default but it can be changed.

### 17.2.2 `blackClip(imIn, imOut, step=0, grid=DEFAULT_GRID)`

Performs a black skeleton clipping (clipping of a black skeleton image). If 'step' is not defined (or equal to 0), the clipping is performed until idempotence. If 'step' is defined, 'step' black points (if possible) will be removed from each branch of the black skeleton.

'edge' is always set to `FILLED`.

### 17.2.3 `computeSKIZ(imIn, imOut, grid=DEFAULT_GRID)`

Computes the influence zones of each connected component of 'imIn' and puts the result in 'imOut'. Inverting the result produces the skeleton by influence zones (SKIZ).

There exists a much faster way to compute the SKIZ operation (see `segment.py` module).

### 17.2.4 `endPoints(imIn, imOut, grid=DEFAULT_GRID, edge=FILLED)`

Extracts end points in 'imIn', supposed to be a "skeleton" image (connected components without thickness), and puts them in 'imOut'.

'edge' is `FILLED` by default and it can be modified to take into account extremities touching the edge.

### 17.2.5 fullGeodesicThick(imIn, imMask, imOut, dse)

Performs a complete geodesic thickening (until idempotence) of image 'imIn' inside mask 'imMask' with all the rotations of the double structuring element 'dse'. The result is put in 'imOut'.

### 17.2.6 fullGeodesicThin(imIn, imMask, imOut, dse)

Performs a complete geodesic thinning (until idempotence) of image 'imIn' inside mask 'imMask' with all the rotations of the double structuring element 'dse'. The result is put in 'imOut'.

### 17.2.7 fullThick(imIn, imOut, dse)

Performs a complete thickening of 'imIn' with the successive rotations of 'dse' (until idempotence) and puts the result in 'imOut'.

'imIn' and 'imOut' are binary images.

The edge is always set to EMPTY.

### 17.2.8 fullThin(imIn, imOut, dse, edge=EMPTY)

Performs a complete thinning of 'imIn' with the successive rotations of 'dse' (until idempotence) and puts the result in 'imOut'.

'imIn' and 'imOut' are binary images.

'edge' is set to EMPTY by default.

### 17.2.9 geodesicThick(imIn, imMask, imOut, dse)

Geodesic thickening of image 'imIn' inside 'imMask' by the double structuring element 'dse'. The result is stored in 'imOut'.

'imIn', 'imMask' and 'imOut' are binary images.

### 17.2.10 geodesicThin(imIn, imMask, imOut, dse)

Geodesic thinning of image 'imIn' inside 'imMask' by the double structuring element 'dse'. The result is stored in 'imOut'.

'imIn', 'imMask' and 'imOut' are binary images.

### 17.2.11 homotopicReduction(imIn, imOut, grid=DEFAULT\_GRID)

Reduces any simply connected component of 'imIn' (component without holes) to a single point. All other components are reduced to simpler ones, with same homotopy as the initial ones. This transformation is simply thinD on the hexagonal grid. It is a combination of thinnings with D and E structuring elements on square grid.

### 17.2.12 infThin(imIn, imOut, dse, edge=EMPTY)

Performs an inf of thinnings, each thinning being made with the successive rotations of 'dse'. The initial image 'imIn' is used at each step of thinning (intersection of thinnings).

'imIn' and 'imOut' are binary images.

'edge' is set to EMPTY by default.

### 17.2.13 multiplePoints(imIn, imOut, grid=DEFAULT\_GRID)

Extracts multiple points in 'imIn', supposed to be a "skeleton" image (connected components without thickness), and puts the result in 'imOut'.

Note that, on a square grid, the resulting skeleton is supposed to be defined on a 4-connectivity grid. if it is not the case, some multiple points are likely to be missed.

#### 17.2.14 `rotatingGeodesicThick(imIn, imMask, imOut, dse)`

Performs successive geodesic thickenings of 'imIn' inside 'imMask' with clockwise rotations of the double structuring element 'dse'. The number of rotations is either 6 or 8 according to the grid where 'dse' is defined. All the thickenings are concatenated.

'imIn', 'imMask' and 'imOut' are binary images.

#### 17.2.15 `rotatingGeodesicThin(imIn, imMask, imOut, dse)`

Performs successive geodesic thinnings of 'imIn' inside 'imMask' with clockwise rotations of the double structuring element 'dse'. The number of rotations is either 6 or 8 according to the grid where 'dse' is defined. All the thinnings are concatenated.

'imIn', 'imMask' and 'imOut' are binary images.

#### 17.2.16 `rotatingThick(imIn, imOut, dse)`

Performs a complete rotation of thickenings, the initial 'dse' double structuring element being turned one step clockwise after each thickening. At each rotation step, the previous result is used as input for the next thickening (chained thickenings). Depending on the grid where 'dse' is defined, 6 or 8 rotations are performed.

'imIn' and 'imOut' are binary images.

The edge is always set to EMPTY.

#### 17.2.17 `rotatingThin(imIn, imOut, dse, edge=FILLED)`

Performs a complete rotation of thinnings, the initial 'dse' double structuring element being turned one step clockwise after each thinning. At each rotation step, the previous result is used as input for the next thinning (chained thinnings). Depending on the grid where 'dse' is defined, 6 or 8 rotations are performed.

'imIn' and 'imOut' are binary images.

'edge' is set to FILLED by default (default value is EMPTY in simple thin).

#### 17.2.18 `supThick(imIn, imOut, dse)`

Performs a sup of thickenings, each thickening being made with the successive rotations of 'dse'. The initial image 'imIn' is used at each step of thickening (union of thickenings).

'imIn' and 'imOut' are binary images.

The edge is always set to EMPTY.

#### 17.2.19 `thick(imIn, imOut, dse)`

Elementary thickening operator with 'dse' double structuring element. The

'imIn' and 'imOut' are binary images.

The edge is always EMPTY (as for `mamba.hitOrMiss`).

#### 17.2.20 `thickD(imIn, imOut, grid=DEFAULT_GRID)`

Complete thickening with D structuring element. Depending on the grid in use, `hexaM` or `squareM` will be used. M structuring element is the flipping of D for the thickening.

This operator must be used with binary images.

#### 17.2.21 `thickL(imIn, imOut, grid=DEFAULT_GRID)`

Complete thickening with L structuring element. Depending on the grid in use, `hexagonalL` or `squareL` will be used. Note that L is equal to its flipping (same structuring element is used in thinning and thickening).

This operator must be used with binary images.

The edge is always EMPTY.

#### 17.2.22 `thickM(imIn, imOut, grid=DEFAULT_GRID)`

Complete thickening with M structuring element. Depending on the grid in use, `hexagonalD` or `squareD` will be used. D structuring element is the flipping of M structuring element for the thickening.

This operator must be used with binary images.

### 17.2.23 `thin(imIn, imOut, dse, edge=EMPTY)`

Elementary thinning operator with 'dse' double structuring element.

'imIn' and 'imOut' are binary images.

'edge' is set to EMPTY by default.

### 17.2.24 `thinD(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY)`

Complete thinning with D structuring element. Depending on the grid in use, hexagonalD or squareD will be used. This operator is mainly used to simplify each connected component to the simplest homotopic equivalent set (see `homotopicReduction` in this module).

This operator must be used with binary images.

The edge is set to EMPTY by default.

### 17.2.25 `thinL(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY)`

Complete thinning with L structuring element. Depending on the grid in use, hexagonalL or squareL will be used. This operator is also called skeleton, as it produces a result which looks like a connected skeleton of each connected component of 'imIn'.

This operator must be used with binary images.

The edge is set to EMPTY by default.

### 17.2.26 `thinM(imIn, imOut, grid=DEFAULT_GRID, edge=EMPTY)`

Complete thinning with M structuring element. Depending on the grid in use, hexagonalM or squareM will be used. This operator produces skeletons with lots of fishbones.

This operator must be used with binary images.

The edge is set to EMPTY by default.

### 17.2.27 `whiteClip(imIn, imOut, step=0, grid=DEFAULT_GRID, edge=FILLED)`

Performs a skeleton clipping of 'imIn' (supposed to contain a skeleton image) and puts the result in 'imOut'. If 'step' is not defined (or equal to 0), the clipping is performed until idempotence. If 'step' is defined, 'step' points (if possible) will be removed from each branch of the skeleton.

'edge' is set to FILLED by default.