Module : mamba.arithmetic

Arithmetic and logical operators. This module provides arithmetic operators such as addition, subtraction, multiplication and division between images together with logical operators (and, or, not, xor ...) between these images.

Functions :

def add(imIn1, imIn2, imOut):
def addConst(imIn, v, imOut):
def ceilingAdd(imIn1, imIn2, imOut):
def ceilingAddConst(imIn, v, imOut):
def diff(imIn1, imIn2, imOut):
def div(imIn1, imIn2, imOut):
def divConst(imIn, v, imOut):
def floorSub(imIn1, imIn2, imOut):
def floorSubConst(imIn, v, imOut):
def logic(imIn1, imIn2, imOut, log):
def mul(imIn1, imIn2, imOut):
def mulConst(imIn, v, imOut):
def mulRealConst(imIn, v, imOut, nearest=False, precision=2):
def negate(imIn, imOut):
def sub(imIn1, imIn2, imOut):
def subConst(imIn, v, imOut):

Module : mamba.base

Image class definition. This is the base module of the Mamba Image library. It defines the imageMb class used to contain images.

Class :

imageMb

- __del__(self)
- __init__(self, *args, **kwargs)
- __str__(self)
- convert(self, depth)
- extractRaw(self)
- fastSetPixel(self, value, position)
- fill(self, v)
- freeze(self)
- getDepth(self)
- getName(self)
- getPixel(self, position)
- getSize(self)
- hide(self)
- load(self, path, rgbfilter=None)
- loadRaw(self, dataOrPath, preprocfunc=None)
- reset(self)
- save(self, path, palette=None)
- setName(self, name)
- setPixel(self, value, position)
- show(self, **options)
- unfreeze(self)
- update(self)

Functions :

def getImageCounter():
def getShowImages():
def setImageIndex(index):
def setShowImages(showThem):

Module : mamba.contrasts

Contrast operators. This module provides a set of functions to perform morphological contrast operators such as gradient, top-hat transform, ...

Functions :

def blackTopHat(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def gradient(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def halfGradient(imIn, imOut, type='intern', n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def regularisedGradient(imIn, imOut, n, grid=HEXAGONAL):
def supBlackTopHat(imIn, imOut, n, grid=HEXAGONAL):
def supWhiteTopHat(imIn, imOut, n, grid=HEXAGONAL):
def whiteTopHat(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):

Module : mamba.conversion

Depth conversion operators. This module regroups various functions/operators to perform conversions based on image depth. It allows to transfer data from an image depth to another.

Functions :

def convert(imIn, imOut):
def convertByMask(imIn, imOut, mFalse, mTrue):
def generateSupMask(imIn1, imIn2, imOut, strict):
def lookup(imIn, imOut, lutable):
def threshold(imIn, imOut, low, high):

Module : mamba.copies

Copy operators. This module regroups various complete or partial copy operators.

Functions :

def copy(imIn, imOut):
def copyBitPlane(imIn, plane, imOut):
def copyBytePlane(imIn, plane, imOut):
def copyLine(imIn, nIn, imOut, nOut):
def cropCopy(imIn, posIn, imOut, posOut, size):

Module : mamba.draw

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

Functions :

def drawBox(imOut, square, value):
def drawCircle(imOut, circle, value):
def drawFillCircle(imOut, circle, value):
def drawLine(imOut, line, value):
def drawSquare(imOut, square, value):
def getIntensityAlongLine(imOut, line):

Module : mamba.erodil

Erosion and dilation operators. This module provides a set of functions and class to perform erosions and dilations. The module contains basic and complex operators that are based on neighbor comparisons. In particular, it defines the structuring element class which serve as the base for these operators. The module also contains distance functions based on erosion.

Class :

structuringElement

- __eq__(self, otherSE)
- __init__(self, directions, grid)
- __ne__(self, otherSE)
- __repr__(self)
- getDirections(self, withoutZero=False)
- getEncodedDirections(self, withoutZero=False)
- getGrid(self)
- hasZero(self)
- rotate(self, step=1)
- setAs(self, se)
- transpose(self)

Functions :

def computeDistance(imIn, imOut, grid=HEXAGONAL, edge=EMPTY):
def conjugateHexagonalDilate(imIn, imOut, size, edge=EMPTY):
def conjugateHexagonalErode(imIn, imOut, size, edge=FILLED):
def diffNeighbor(imIn, imInout, nb, grid=HEXAGONAL, edge=EMPTY):
def dilate(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=EMPTY):
def dodecagonalDilate(imIn, imOut, size, edge=EMPTY):
def dodecagonalErode(imIn, imOut, size, edge=FILLED):
def doublePointDilate(imIn, imOut, d, n, grid=HEXAGONAL, edge=EMPTY):
def doublePointErode(imIn, imOut, d, n, grid=HEXAGONAL, edge=FILLED):
def erode(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED):
def infNeighbor(imIn, imInout, nb, grid=HEXAGONAL, edge=FILLED):
def infVector(imIn, imInout, vector, edge=FILLED):
def isotropicDistance(imIn, imOut, edge=FILLED):
def linearDilate(imIn, imOut, d, n=1, grid=HEXAGONAL, edge=EMPTY):
def linearErode(imIn, imOut, d, n=1, grid=HEXAGONAL, edge=FILLED):
def octogonalDilate(imIn, imOut, size, edge=EMPTY):
def octogonalErode(imIn, imOut, size, edge=FILLED):
def supNeighbor(imIn, imInout, nb, grid=HEXAGONAL, edge=EMPTY):
def supVector(imIn, imInout, vector, edge=EMPTY):

Module : mamba.erodilLarge

Large erosion and dilation operators. This module provides a set of functions performing erosions and dilations with large structuring elements. These operators are to be prefered to the standard ones when working with large structuring elements.

Functions :

def infFarNeighbor(imIn, imInout, nb, amp, grid=HEXAGONAL, edge=FILLED):
def largeDodecagonalDilate(imIn, imOut, size, edge=EMPTY):
def largeDodecagonalErode(imIn, imOut, size, edge=FILLED):
def largeHexagonalDilate(imIn, imOut, size, edge=EMPTY):
def largeHexagonalErode(imIn, imOut, size, edge=FILLED):
def largeLinearDilate(imIn, imOut, dir, size, grid=HEXAGONAL, edge=EMPTY):
def largeLinearErode(imIn, imOut, dir, size, grid=HEXAGONAL, edge=FILLED):
def largeOctogonalDilate(imIn, imOut, size, edge=EMPTY):
def largeOctogonalErode(imIn, imOut, size, edge=FILLED):
def largeSquareDilate(imIn, imOut, size, edge=EMPTY):
def largeSquareErode(imIn, imOut, size, edge=FILLED):
def supFarNeighbor(imIn, imInout, nb, amp, grid=HEXAGONAL, edge=EMPTY):

Module : mamba.extrema

Extrema (min/max) operators. This module provides a set of operators dealing with maxima and minima of a function. New operators linked to the notion of dynamics are also provided.

Functions :

def deepMinima(imIn, imOut, h, grid=HEXAGONAL):
def highMaxima(imIn, imOut, h, grid=HEXAGONAL):
def maxDynamics(imIn, imOut, h, grid=HEXAGONAL):
def maxPartialBuild(imIn, imMask, imOut, grid=HEXAGONAL):
def maxima(imIn, imOut, h=1, grid=HEXAGONAL):
def minDynamics(imIn, imOut, h, grid=HEXAGONAL):
def minPartialBuild(imIn, imMask, imOut, grid=HEXAGONAL):
def minima(imIn, imOut, h=1, grid=HEXAGONAL):

Module : mamba.filter

Filtering operators. This module provides a set of functions to perform morphological filtering operations such as alternate filters.

Functions :

def alternateFilter(imIn, imOut, n, openFirst, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def autoMedian(imIn, imOut, n, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def fullAlternateFilter(imIn, imOut, n, openFirst, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def largeDodecagonalAlternateFilter(imIn, imOut, start, end, step, openFirst):
def largeHexagonalAlternateFilter(imIn, imOut, start, end, step, openFirst):
def largeOctogonalAlternateFilter(imIn, imOut, start, end, step, openFirst):
def largeSquareAlternateFilter(imIn, imOut, start, end, step, openFirst):
def linearAlternateFilter(imIn, imOut, n, openFirst, grid=HEXAGONAL):
def simpleLevelling(imIn, imMask, imOut, grid=HEXAGONAL):
def strongLevelling(imIn, imOut, n, eroFirst, grid=HEXAGONAL):

Module : mamba.geodesy

Geodesic operators. This module provides a set of functions to perform geodesic computations. It includes build and dualbuild operations, geodesic erosion and dilation ...

Functions :

def build(imMask, imInout, grid=HEXAGONAL):
def buildNeighbor(imMask, imInout, d, grid=HEXAGONAL):
def closeHoles(imIn, imOut, grid=HEXAGONAL):
def dualBuild(imMask, imInout, grid=HEXAGONAL):
def dualbuildNeighbor(imMask, imInout, d, grid=HEXAGONAL):
def geodesicDilate(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def geodesicDistance(imIn, imMask, imOut, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def geodesicErode(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def hierarBuild(imMask, imInout, grid=HEXAGONAL):
def hierarDualBuild(imMask, imInout, grid=HEXAGONAL):
def lowerGeodesicDilate(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def lowerGeodesicErode(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def removeEdgeParticles(imIn, imOut, grid=HEXAGONAL):
def upperGeodesicDilate(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def upperGeodesicErode(imIn, imMask, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):

Module : mamba.grids

Grids handling and setting functions. This module defines various functions related to grid configurations and computations.

Functions :

def getDirections(grid=HEXAGONAL, withoutZero=False):
def gridNeighbors(grid=HEXAGONAL):
def rotateDirection(d, step=1, grid=HEXAGONAL):
def setDefaultGrid(grid):
def transposeDirection(d, grid=HEXAGONAL):

Module : mamba.hierarchies

Hierarchical segmentation operators. This module provides a set of functions to perform hierarchical segmentation operations. This module contains the waterfalls algorithm and various hierarchical operators (enhanced waterfalls, standard hierarchy and P algorithm).

Functions :

def enhancedWaterfalls(imIn, imOut, grid=HEXAGONAL):
def extendedSegment(imIn, imTest, imOut, offset=255, grid=HEXAGONAL):
def generalSegment(imIn, imOut, gain=2.0, offset=1, grid=HEXAGONAL):
def hierarchicalLevel(imIn, imOut, grid=HEXAGONAL):
def hierarchy(imIn, imMask, imOut, grid=HEXAGONAL):
def segmentByP(imIn, imOut, gain=2.0, grid=HEXAGONAL):
def standardSegment(imIn, imOut, gain=2.0, grid=HEXAGONAL):
def waterfalls(imIn, imOut, grid=HEXAGONAL):

Module : mamba.labellings

This module contains various labelling procedures for sets and for partitions. They use the label operator now available for sets and for grey images.

Functions :

def areaLabelling(imIn, imOut):
def diameterLabelling(imIn, imOut, dir, grid=HEXAGONAL):
def feretDiameterLabelling(imIn, imOut, direc):
def measureLabelling(imIn, imMeasure, imOut):
def partitionLabel(imIn, imOut):
def volumeLabelling(imIn1, imIn2, imOut):

Module : mamba.measure

Measure operators. This module provides a set of functions which perform measure operations on an image. Measures include volume, range, area ...

Functions :

def computeArea(imIn, scale=(1.0, 1.0)):
def computeComponentsNumber(imIn, grid=HEXAGONAL):
def computeConnectivityNumber(imIn, grid=HEXAGONAL):
def computeDiameter(imIn, dir, scale=(1.0, 1.0), grid=HEXAGONAL):
def computeFeretDiameters(imIn, scale=(1.0, 1.0)):
def computeMaxRange(imIn):
def computePerimeter(imIn, scale=(1.0, 1.0), grid=HEXAGONAL):
def computeRange(imIn):
def computeVolume(imIn):

Module : mamba.miscellaneous

Various unclassed operators. 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.

Functions :

def Mamba2PIL(imIn):
def PIL2Mamba(pilim, imOut):
def checkEmptiness(imIn):
def compare(imIn1, imIn2, imOut):
def drawEdge(imOut, thick=1):
def extractFrame(imIn, threshold):
def mix(imInR, imInG, imInB):
def multiSuperpose(imInout, *imIns):
def shift(imIn, imOut, d, amp, fill, grid=HEXAGONAL):
def shiftVector(imIn, imOut, vector, fill):
def split(pilimIn, imOutR, imOutG, imOutB):

Module : mamba.openclose

Opening and closing operators. This module provides a set of functions to perform opening and closing operations. 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).

Functions :

def buildClose(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def buildOpen(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def closing(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED):
def infClose(imIn, imOut, n, grid=HEXAGONAL):
def linearClose(imIn, imOut, dir, n, grid=HEXAGONAL, edge=FILLED):
def linearOpen(imIn, imOut, dir, n, grid=HEXAGONAL, edge=FILLED):
def opening(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED):
def supOpen(imIn, imOut, n, grid=HEXAGONAL):

Module : mamba.partitions

Partitioning operators. This module contains operators acting on partitions. Two types of operators are defined: operators acting on each cell of the partition independently, or operators considering each cell of the partition as a node of a weighted graph. In the first case, each cell of the partition is considered as a binary set and the defined operation is applied on each cell to produce a new transformation (which is not necessarily a partition). In the second case, the partition is considered as a graph and morphological operators are defined on this graph. These operators provide morphological transformations on graphs, without the need to explicitly define this graph structure from the partition.

Functions :

def cellsBuild(imIn, imInOut, grid=HEXAGONAL):
def cellsComputeDistance(imIn, imOut, grid=HEXAGONAL, edge=EMPTY):
def cellsErode(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED):
def cellsExtract(imIn, imMarkers, imOut, grid=HEXAGONAL):
def cellsFullThin(imIn, imOut, dse, edge=EMPTY):
def cellsHMT(imIn, imOut, dse, edge=EMPTY):
def cellsOpen(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL), edge=FILLED):
def cellsOpenByBuild(imIn, imOut, n=1, se=structuringElement([0, 1, 2, 3, 4, 5, 6], mamba.HEXAGONAL)):
def cellsThin(imIn, imOut, dse, edge=EMPTY):
def equalNeighbors(imIn, imOut, nb, grid=HEXAGONAL, edge=FILLED):
def nonEqualNeighbors(imIn, imOut, nb, grid=HEXAGONAL, edge=FILLED):
def partitionDilate(imIn, imOut, n=1, grid=HEXAGONAL):
def partitionErode(imIn, imOut, n=1, grid=HEXAGONAL):

Module : mamba.residues

Residual operators. This module provides a set of functions to perform residual operations. 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.

Functions :

def binarySkeletonByOpening(imIn, imOut1, imOut2, grid=HEXAGONAL, edge=FILLED):
def binaryUltimateErosion(imIn, imOut1, imOut2, grid=HEXAGONAL, edge=FILLED):
def fullRegularisedGradient(imIn, imOut1, imOut2, grid=HEXAGONAL, maxSize=16):
def quasiDistance(imIn, imOut1, imOut2, grid=HEXAGONAL):
def skeletonByOpening(imIn, imOut1, imOut2, grid=HEXAGONAL):
def ultimateBuildOpening(imIn, imOut1, imOut2, grid=HEXAGONAL):
def ultimateErosion(imIn, imOut1, imOut2, grid=HEXAGONAL):
def ultimateIsotropicOpening(imIn, imOut1, imOut2, step=1, grid=HEXAGONAL):
def ultimateOpening(imIn, imOut1, imOut2, grid=HEXAGONAL):

Module : mamba.segment

Segmentation operators. This module provides a set of functions to perform segmentation operations (such as watershed and basin). The module also contains the labelling operator.

Functions :

def basinSegment(imIn, imMarker, grid=HEXAGONAL, max_level=0):
def fastSKIZ(imIn, imOut, grid=HEXAGONAL):
def geodesicSKIZ(imIn, imMask, imOut, grid=HEXAGONAL):
def label(imIn, imOut, lblow=0, lbhigh=256, grid=HEXAGONAL):
def markerControlledWatershed(imIn, imMarkers, imOut, grid=HEXAGONAL):
def mosaic(imIn, imOut, imWts, grid=HEXAGONAL):
def mosaicGradient(imIn, imOut, grid=HEXAGONAL):
def valuedWatershed(imIn, imOut, grid=HEXAGONAL):
def watershedSegment(imIn, imMarker, grid=HEXAGONAL, max_level=0):

Module : mamba.statistic

Statistical operators. This module provides a set of functions to compute statistical values such as mean and median inside an image.

Functions :

def getHistogram(imIn):
def getMean(imIn):
def getMedian(imIn):
def getVariance(imIn):

Module : mamba.thinthick

Thinning and thickening operators. This module contains morphological thinning and thickening operators based on the Hit-or-Miss transformation, together with various homotopic and geodesic functions derived from these operators. The module also defines the double structuring element class which serve as a base for these operators.

Class :

doubleStructuringElement

- __init__(self, *args)
- __repr__(self)
- flip(self)
- getCSE(self)
- getGrid(self)
- getStructuringElement(self, ground)
- rotate(self, step=1)

Functions :

def blackClip(imIn, imOut, step=0, grid=HEXAGONAL):
def computeSKIZ(imIn, imOut, grid=HEXAGONAL):
def endPoints(imIn, imOut, grid=HEXAGONAL, edge=FILLED):
def fullGeodesicThick(imIn, imMask, imOut, dse):
def fullGeodesicThin(imIn, imMask, imOut, dse):
def fullThick(imIn, imOut, dse):
def fullThin(imIn, imOut, dse, edge=EMPTY):
def geodesicThick(imIn, imMask, imOut, dse):
def geodesicThin(imIn, imMask, imOut, dse):
def hitOrMiss(imIn, imOut, dse, edge=EMPTY):
def homotopicReduction(imIn, imOut, grid=HEXAGONAL):
def infThin(imIn, imOut, dse, edge=EMPTY):
def multiplePoints(imIn, imOut, grid=HEXAGONAL):
def rotatingGeodesicThick(imIn, imMask, imOut, dse):
def rotatingGeodesicThin(imIn, imMask, imOut, dse):
def rotatingThick(imIn, imOut, dse):
def rotatingThin(imIn, imOut, dse, edge=FILLED):
def supThick(imIn, imOut, dse):
def thick(imIn, imOut, dse):
def thickD(imIn, imOut, grid=HEXAGONAL):
def thickL(imIn, imOut, grid=HEXAGONAL):
def thickM(imIn, imOut, grid=HEXAGONAL):
def thin(imIn, imOut, dse, edge=EMPTY):
def thinD(imIn, imOut, grid=HEXAGONAL, edge=EMPTY):
def thinL(imIn, imOut, grid=HEXAGONAL, edge=EMPTY):
def thinM(imIn, imOut, grid=HEXAGONAL, edge=EMPTY):
def whiteClip(imIn, imOut, step=0, grid=HEXAGONAL, edge=FILLED):

Module : mamba3D.arithmetic3D

Arithmetic and logical 3D operators. This module provides arithmetic operators such as addition, subtraction, multiplication, division and logical operators between 3D images. (and, or, not, xor ...).

Functions :

def add3D(imIn1, imIn2, imOut):
def addConst3D(imIn, v, imOut):
def ceilingAdd3D(imIn1, imIn2, imOut):
def ceilingAddConst3D(imIn, v, imOut):
def diff3D(imIn1, imIn2, imOut):
def div3D(imIn1, imIn2, imOut):
def divConst3D(imIn, v, imOut):
def floorSub3D(imIn1, imIn2, imOut):
def floorSubConst3D(imIn, v, imOut):
def logic3D(imIn1, imIn2, imOut, log):
def mul3D(imIn1, imIn2, imOut):
def mulConst3D(imIn, v, imOut):
def mulRealConst3D(imIn, v, imOut, nearest=False, precision=2):
def negate3D(imIn, imOut):
def sub3D(imIn1, imIn2, imOut):
def subConst3D(imIn, v, imOut):

Module : mamba3D.base3D

3D Image class definition. This is the base module of the Mamba 3D Image library. It defines the image3DMb class used to contain images. A 3D image can be considered as a stack or sequence of 2D images. The module also defines an alias to image3DMb named sequenceMb.

Class :

image3DMb

- __del__(self)
- __getitem__(self, key)
- __init__(self, *args, **kwargs)
- __iter__(self)
- __len__(self)
- __next__(self)
- __str__(self)
- convert(self, depth)
- extractRaw(self)
- fill(self, v)
- freeze(self)
- getDepth(self)
- getName(self)
- getPixel(self, position)
- getSize(self)
- hide(self)
- load(self, path, rgbfilter=None)
- loadRaw(self, dataOrPath, preprocfunc=None)
- next(self)
- reset(self)
- save(self, path, extension='.png', palette=None)
- setName(self, name)
- setPixel(self, value, position)
- show(self, **options)
- unfreeze(self)
- update(self)

sequenceMb(image3DMb)

Module : mamba3D.contrasts3D

Contrast 3D operators. This module provides a set of functions to perform morphological contrast operators such as gradient, top-hat transform, ...

Functions :

def blackTopHat3D(imIn, imOut, n, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def gradient3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def halfGradient3D(imIn, imOut, type='intern', n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def regularisedGradient3D(imIn, imOut, n, grid=mamba3D.FACE_CENTER_CUBIC):
def supBlackTopHat3D(imIn, imOut, n, grid=mamba3D.FACE_CENTER_CUBIC):
def supWhiteTopHat3D(imIn, imOut, n, grid=mamba3D.FACE_CENTER_CUBIC):
def whiteTopHat3D(imIn, imOut, n, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):

Module : mamba3D.conversion3D

Depth conversion 3D operators. This module regroups various functions/operators to perform conversions based on image depth. It allows to transfer data from a 3D image depth to another.

Functions :

def convert3D(imIn, imOut):
def convertByMask3D(imIn, imOut, mFalse, mTrue):
def generateSupMask3D(imIn1, imIn2, imOut, strict):
def lookup3D(imIn, imOut, lutable):
def threshold3D(imIn, imOut, low, high):

Module : mamba3D.copies3D

Copy 3D operators. This module regroups various complete or partial copy operators for 3D images.

Functions :

def copy3D(imIn, imOut, firstPlaneIn=0, firstPlaneOut=0):
def copyBitPlane3D(imIn, plane, imOut):
def copyBytePlane3D(imIn, plane, imOut):
def cropCopy3D(imIn, posin, imOut, posout, size):

Module : mamba3D.draw3D

Drawing 3D operators. This module defines functions to draw inside Mamba 3D images. Drawing functions include lines, cubes, ... The module also provides functions to extract pixel information.

Functions :

def drawCube(imOut, cube, value):
def drawLine3D(imOut, line, value):
def getIntensityAlongLine3D(imOut, line):

Module : mamba3D.erodil3D

Erosion and dilation 3D operators. This module provides a set of functions and class to perform erosions and dilations. The module contains basic and complex operators that are based on neighbor comparisons. In particular it defines the 3D structuring element class which serves as the base for these operators. The module also contains distance functions based on erosion.

Class :

structuringElement3D

- __eq__(self, otherSE)
- __init__(self, directions, grid)
- __ne__(self, otherSE)
- __repr__(self)
- getDirections(self, withoutZero=False)
- getGrid(self)
- hasZero(self)
- transpose(self)

Functions :

def computeDistance3D(imIn, imOut, grid=mamba3D.FACE_CENTER_CUBIC, edge=EMPTY):
def dilate3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC), edge=EMPTY):
def dilateByCylinder3D(imInOut, height, section):
def erode3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC), edge=FILLED):
def erodeByCylinder3D(imInOut, height, section):
def linearDilate3D(imIn, imOut, d, n=1, grid=mamba3D.FACE_CENTER_CUBIC, edge=EMPTY):
def linearErode3D(imIn, imOut, d, n=1, grid=mamba3D.FACE_CENTER_CUBIC, edge=FILLED):

Module : mamba3D.erodilLarge3D

Erosion and dilation 3D operators for large structuring elements. This module provides a set of functions and class to perform erosions and dilations with large structuring elements on 3D images.

Functions :

def infFarNeighbor3D(imIn, imInOut, nb, amp, grid=mamba3D.FACE_CENTER_CUBIC, edge=FILLED):
def largeCubeDilate(imIn, imOut, size, edge=EMPTY):
def largeCubeErode(imIn, imOut, size, edge=FILLED):
def largeLinearDilate3D(imIn, imOut, dir, size, grid=mamba3D.FACE_CENTER_CUBIC, edge=EMPTY):
def largeLinearErode3D(imIn, imOut, dir, size, grid=mamba3D.FACE_CENTER_CUBIC, edge=FILLED):
def supFarNeighbor3D(imIn, imInOut, nb, amp, grid=mamba3D.FACE_CENTER_CUBIC, edge=EMPTY):

Module : mamba3D.extrema3D

Extrema (min/max) 3D operators. This module provides a set of operators dealing with maxima and minima of a function. New operators linked to the notion of dynamics are also provided.

Functions :

def deepMinima3D(imIn, imOut, h, grid=mamba3D.FACE_CENTER_CUBIC):
def highMaxima3D(imIn, imOut, h, grid=mamba3D.FACE_CENTER_CUBIC):
def maxDynamics3D(imIn, imOut, h, grid=mamba3D.FACE_CENTER_CUBIC):
def maxPartialBuild3D(imIn, imMask, imOut, grid=mamba3D.FACE_CENTER_CUBIC):
def maxima3D(imIn, imOut, h=1, grid=mamba3D.FACE_CENTER_CUBIC):
def minDynamics3D(imIn, imOut, h, grid=mamba3D.FACE_CENTER_CUBIC):
def minPartialBuild3D(imIn, imMask, imOut, grid=mamba3D.FACE_CENTER_CUBIC):
def minima3D(imIn, imOut, h=1, grid=mamba3D.FACE_CENTER_CUBIC):

Module : mamba3D.filter3D

Filtering operators. This module provides a set of functions to perform morphological filtering operations such as alternate filters.

Functions :

def alternateFilter3D(imIn, imOut, n, openFirst, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def autoMedian3D(imIn, imOut, n, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def fullAlternateFilter3D(imIn, imOut, n, openFirst, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def linearAlternateFilter3D(imIn, imOut, n, openFirst, grid=mamba3D.FACE_CENTER_CUBIC):
def simpleLevelling3D(imIn, imMask, imOut, grid=mamba3D.FACE_CENTER_CUBIC):
def strongLevelling3D(imIn, imOut, n, eroFirst, grid=mamba3D.FACE_CENTER_CUBIC):

Module : mamba3D.geodesy3D

Geodesic 3D operators. This module provides a set of functions to perform geodesic computations. It includes build and dualbuild operations, geodesic erosion and dilation ...

Functions :

def build3D(imMask, imInout, grid=mamba3D.FACE_CENTER_CUBIC):
def buildNeighbor3D(imMask, imInOut, d, grid=mamba3D.FACE_CENTER_CUBIC):
def closeHoles3D(imIn, imOut, grid=mamba3D.FACE_CENTER_CUBIC):
def dualBuild3D(imMask, imInout, grid=mamba3D.FACE_CENTER_CUBIC):
def dualbuildNeighbor3D(imMask, imInOut, d, grid=mamba3D.FACE_CENTER_CUBIC):
def geodesicDilate3D(imIn, imMask, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def geodesicErode3D(imIn, imMask, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def lowerGeodesicDilate3D(imIn, imMask, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def lowerGeodesicErode3D(imIn, imMask, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def removeEdgeParticles3D(imIn, imOut, grid=mamba3D.FACE_CENTER_CUBIC):
def upperGeodesicDilate3D(imIn, imMask, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def upperGeodesicErode3D(imIn, imMask, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):

Module : mamba3D.grids3D

3D Grids handling and setting functions. This module defines various functions related to grid configurations and computations. It also defines the 3D grids (cubic, face-centered cubic ...) used with the 3D operators.

Functions :

def getDirections3D(grid=mamba3D.FACE_CENTER_CUBIC, withoutZero=False):
def gridNeighbors3D(grid=mamba3D.FACE_CENTER_CUBIC):
def setDefaultGrid3D(grid):
def transposeDirection3D(d, grid=mamba3D.FACE_CENTER_CUBIC):

Module : mamba3D.measure3D

Measure 3D operators. This module provides a set of functions which perform measure operations on a 3D image. Measures include volume, range ...

Functions :

def computeMaxRange3D(imIn):
def computeRange3D(imIn):
def computeVolume3D(imIn):

Module : mamba3D.miscellaneous3D

Various unclassed 3D operators. 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.

Functions :

```
def checkEmptiness3D(imIn):
def compare3D(imIn1, imIn2, imOut):
def drawEdge3D(imOut, thick=1):
def shift3D(imIn, imOut, d, amp, fill,
grid=mamba3D.FACE_CENTER_CUBIC):
```

Module : mamba3D.openclose3D

Opening and closing operators. This module provides a set of functions to perform opening and closing operations. 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 erodil3D and geodesy3D).

Functions :

```
def buildClose3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2,
3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def buildOpen3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2,
3, 4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC)):
def closeByCylinder3D(imInOut, height, section):
def closing3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2, 3, 4,
5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC),
edge=FILLED):
def infClose3D(imIn, imOut, n,
grid=mamba3D.FACE_CENTER_CUBIC):
def linearClose3D(imIn, imOut, dir, n,
grid=mamba3D.FACE_CENTER_CUBIC, edge=FILLED):
def linearOpen3D(imIn, imOut, dir, n,
grid=mamba3D.FACE_CENTER_CUBIC, edge=FILLED):
def openByCylinder3D(imInOut, height, section):
def opening3D(imIn, imOut, n=1, se=structuringElement3D([0, 1, 2, 3,
4, 5, 6, 7, 8, 9, 10, 11, 12], mamba3D.FACE_CENTER_CUBIC),
edge=FILLED):
def supOpen3D(imIn, imOut, n,
grid=mamba3D.FACE_CENTER_CUBIC):
```

Module : mamba3D.segment3D

Segmentation 3D operators. This module provides a set of functions to perform 3D segmentation operations (such as watershed and basin). The module also contains the labelling operator.

Functions :

```
def basinSegment3D(imIn, imMarker,
grid=mamba3D.FACE_CENTER_CUBIC, max_level=0):
def fastSKIZ3D(imIn, imOut, grid=mamba3D.FACE_CENTER_CUBIC):
def label3D(imIn, imOut, lblow=0, lbhigh=256,
grid=mamba3D.FACE_CENTER_CUBIC):
def markerControlledWatershed3D(imIn, imMarkers, imOut,
grid=mamba3D.FACE_CENTER_CUBIC):
def mosaic3D(imIn, imOut, imWts,
grid=mamba3D.FACE_CENTER_CUBIC):
def mosaicGradient3D(imIn, imOut,
grid=mamba3D.FACE_CENTER_CUBIC):
def valuedWatershed3D(imIn, imOut,
grid=mamba3D.FACE_CENTER_CUBIC):
def watershedSegment3D(imIn, imMarker,
grid=mamba3D.FACE_CENTER_CUBIC, max_level=0):
```

Module : mamba3D.statistic3D

Statistical 3D operators. This module provides a set of functions to compute statistical values such as mean and median inside a 3D image.

Functions :

```
def getHistogram3D(imIn):
def getMean3D(imIn):
def getMedian3D(imIn):
def getVariance3D(imIn):
```

Module : mamba3D.thinthick3D

Thinning and thickening 3D operators. This module contains morphological thinning and thickening operators based on the Hit-or-Miss transformation. The module also defines the double 3D structuring element class which serves as a base for these operators.

Class :

doubleStructuringElement3D

- __init__(self, *args)
- __repr__(self)
- flip(self)
- getGrid(self)
- getStructuringElement3D(self, ground)

Functions :

```
def hitOrMiss3D(imIn, imOut, dse, edge=EMPTY):
def thick3D(imIn, imOut, dse):
def thin3D(imIn, imOut, dse, edge=EMPTY):
```

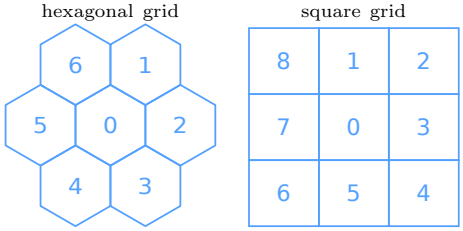
General

Grid and Edge:

Mamba can work with two grids : **HEXAGONAL** and **SQUARE**. The **DEFAULT_GRID** is used by the function when no other grid is specified. Its value, HEXAGONAL at start, can be changed with the appropriate function.
Mamba3D can work with three grids : **CUBIC**, **FACE_CENTER_CUBIC** and **CENTER_CUBIC**. Only the first two are supported by the C operators.
The **DEFAULT_GRID3D** is used by the function when no other grid is specified. Its value, FACE_CENTER_CUBIC at start, can be changed with the appropriate function.
Two edge behaviors are defined : **EMPTY** and **FILLED**.

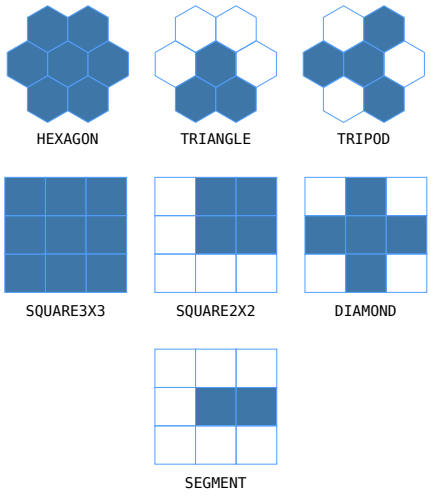
Directions/Neighbors:

Here is the encoding used for Mamba grids:



Structuring elements:

By default Mamba defines seven structuring elements:



By default Mamba3D defines four structuring elements: **CUBOCTAHEDRON** which is based on the FACE_CENTER_CUBIC grid, **CUBE2X2X2** and **CUBE3X3X3** both based on the CUBIC grid, and finally **CUBOCTAHEDRON_BIS** which is based on the CENTER_CUBIC grid.

Package : mambaDisplay

Display and palette functions. mambaDisplay contains all the elements allowing to display mamba images (2D and 3D). The documentation for this package is intended for advanced users who wish to define their own display. Palette definitions. Basic users can use it to list and use existing palettes and build new ones through the provided functions.

Class :

Displayer

- addWindow(self, im)
- controlWindow(self, wKey, ctrl)
- destroyWindow(self, wKey)
- hideWindow(self, wKey)
- showWindow(self, wKey, **options)
- tidyWindows(self)
- updateWindow(self, wKey)

Functions :

```
def getDisplayer():
def setDisplayer(displayer):
def setMaxDisplay(size):
def setMinDisplay(size):
def tidyDisplays():
def addPalette(name, palette):
def getPalette(name):
def listPalettes():
def tagOneColorPalette(value, color):
```

Module : mambaDisplay.extra

Extra displays. This module defines specific extra displays that are meant to be used interactively with the user. This module is not loaded by default with mambaDisplay.

Functions :

```
def dynamicThreshold(imIn):
def hitormissPatternSelector(grid=HEXAGONAL):
def interactiveSegment(imIn, imOut):
def superpose(imIn1, imIn2):
```

Displays shortcuts

Keyboard:

- **P**: will circle through all the available palettes.
- **Z** and **A**: will zoom in/out in 2D displays (basic, projection, player ...)
- **B** and **N**: With 32-bit images, these keys will allow you to move through byte planes or to show the complete image downscaled.
- **Control-F**: will freeze/unfreeze the display. See the freeze() method.
- **Control-R**: will reset the display to its original size and zoom value.
- **Control-V**: will copy any image stored inside the clipboard in your image (only works on 2D images on Windows platforms).
- **F1**: on a 3D image display, will switch to mode PROJECTION.
- **F2**: on a 3D image display, will switch to mode VOLUME RENDERING if you have VTK python bindings available on your computer.
- **F3**: on a 3D image display, will switch to mode PLAYER.
- **F5**: on a 3D image display, will update the display with the image content. Similar to the method update.
- **Space** : will start/stop the player in the 3D image display PLAYER mode.
- **Page-Up** : will display the next image in the sequence in the 3D image display PLAYER mode.
- **Page-Down** : will display the previous image in the sequence in the 3D image display PLAYER mode.

Mouse:

- **scrolling** : will zoom in/out in 2D displays (basic, projection, player ...)
- **motion+Control** : will allow to move through the image in the 3D image display PROJECTION mode.