Imaging Morphology with LAR *

Alberto Paoluzzi

March 17, 2014

Abstract

In this module we aim to implement the four operators of mathematical morphology, i.e. the *dilation*, *erosion*, *opening* and *closing* operators, by the way of matrix operations representing the linear operators—*boundary* and *coboundary*—over LAR. According to the multidimensional character of LAR, our implementation is dimension-independent. In few words, it works as follows: (a) the input is (the coordinate representation of) a d-chain γ; (b) compute its boundary $\partial_d(\gamma)$; (c) extract the maximal $(d-2)$-chain $\epsilon \subset \partial_d(\gamma)$; (d) consider the $(d-1)$-chain returned from its coboundary $\delta_{d-2}(\epsilon)$; (e) compute the d-chain $\eta := \delta_{d-1}(\delta_{d-2}(\epsilon)) \subset C_d$ *without* performing the mod 2 final transformation on the resulting coordinate vector, that would provide a zero result, according to the standard algebraic constraint $\delta \circ \delta = 0$. It is easy to show that $\eta \equiv (\oplus\gamma) - (\ominus\gamma)$ provides the *morphological gradient* operator. The four standard morphological operators are therefore consequently computable.

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. March 17, 2014

1 Test image generation

Various methods for the input or the generation of a test image are developed in the subsections of this section. The aim is to prepare a set of controlled test beds, used to check both the implementation and the working properties of our topological implementation of morphological operators.

1.1 Random binary multidimensional image

A multidimensional binary image is generated here by using a random approach, both for the bulk structure and the small artefacts of the image.

⟨ Generation of random image 2a ⟩ ≡

```
def randomImage(shape, structure, noiseFraction=0.0):
   """ Generation of random image of given shape and structure.
      Return scipy.ndarray(shape)
   """
   print "noiseFraction =",noiseFraction
   ⟨ Generation of bulk array structure 2b ⟩
   ⟨ Generation of random artifacts 3a ⟩
   return image_array
```

◇

Macro referenced in 15a.

Generation of the gross image First we generate a 2D grid of squares by Cartesian product, and produce the bulk of the random image then used to test our approach to morphological operators via topological ones.

⟨ Generation of bulk array structure 2b ⟩ ≡

```
ranges = [shape[k]/structure[k] for k in range(len(shape))]
random_array = randint(0, 255, size=structure)
image_array = numpy.zeros(shape)
for index in scipy.array(CART(AA(range)(shape))):
   block = index/ranges
   if random_array[tuple(block)] < 127:
      image_array[tuple(index)] = 0
   else:
      image_array[tuple(index)] = 255
```

◇

Macro referenced in 2a.

Generation of random artefacts upon the image Then random noise is added to the previously generated image, in order to produce artifacts at the pixel scale.

⟨ Generation of random artifacts 3a ⟩ ≡

```
noiseQuantity = PROD(list(shape))*noiseFraction
k = 0
while k < noiseQuantity:
   index = tuple(AA(randint)(list(shape)))
   if image_array[index] == 0: image_array[index] = 255
   else: image_array[index] = 0
   k += 1
if len(shape)==3:
   for k in range(shape[0]):
      scipy.misc.imsave('tmp/outfile'+str(k).zfill(3)+'.png', image_array[k])
else:
   scipy.misc.imsave('tmp/outfile'+str(k).zfill(3)+'.png', image_array)
```

◇

Macro referenced in 2a.

2 Selection of an image segment

In this section we implement several methods for image segmentation and segment selection.

2.1 Selection of a test chain

The first and simplest method is the selection of the portion of a binary image contained within a masking window. Here we select the (white) sub-image contained in a given window, and compute the coordinate representation of the (chain) sub-image.

Mask definition A *window* within a d-image is defined by $2 \times d$ integer numbers (2 multi-indices), corresponding to the window `minPoint` (minimum indices) and to the win-

dow `maxPoint` (maximum indices). A list of multi-index tuples, contained in the `window` variable, is generated by the function `setMaskWindow` below.

⟨ Generation of a masking window 3b ⟩ ≡

```
def setMaskWindow(window,image_array):
   minPoint, maxPoint = window
   imageShape = list(image_array.shape)
   ⟨ Generation of multi-index window 4 ⟩
   ⟨ Window-to-chain mapping 5b ⟩
   ⟨ Change chain color to grey 5c ⟩
   return segmentChain
```

◇

Macro referenced in 15a.

The set of tuples of indices contained in a (multidimensional) window is given below.

⟨ Generation of multi-index window 4 ⟩ ≡

```
indexRanges = zip(minPoint,maxPoint)
tuples = CART([range(min,max) for min,max in indexRanges])
```

◇

Macro referenced in 3b.

2.2 Mapping of integer tuples to integers

In order to produce the coordinate representation of a chain in a multidimensional image (or d-image) we need: (a) to choose a basis of image elements, i.e. of d-cells, and in particular to fix an ordering of them; (b) to map the multidimensional index, selecting a single d-cell of the image, to a single integer mapping the cell to its linear position within the chosen basis ordering.

Grid of hyper-cubes of unit size Let $S_i = (0, 1, ..., n_i - 1)$ be ordered integer sets with n_i elements, and

$$S = S_0 \times S_1 \times \cdots \times S_{d-1}$$

the set of indices of elements of a d-image.

Definition 1 (d-image shape)**.** *The* shape *of a d-image with $n_0 \times n_1 \times \cdots \times n_{d-1}$ elements (here called* voxels*) is the ordered set $(n_0, n_1, \ldots, n_{d-1})$.*

d-dimensional row-major order Given a d-image with shape $S = (n_0, n_1, ..., n_{d-1})$ and number of elements $n = \prod n_i$, the mapping

$$S_0 \times S_1 \times \cdots \times S_{d-1} \to \{0, 1, \ldots, n-1\}$$

is a linear combination with integer weights $(w_0, w_1, ..., w_{d-2}, 1)$, such that:

$$(i_0, i_1, ..., i_{d-1}) \mapsto i_0 w_0 + i_1 w_1 + \cdots + i_{d-1} w_{d-1},$$

where

$$w_k = n_{k+1} n_{k+2} \cdots n_{d-1}, \qquad 0 \leq k \leq d-2.$$

Implementation A functional implementation of the *Tuples to integers mapping* is given by the second-order `mapTupleToInt` function, that accepts in a first application the `shape` of the image (to compute the tuple space of indices of d-cells), and then takes a single tuple in the second application. Of course, the function returns the cell address in the linear address space associated to the given `shape`.

⟨ Tuples to integers mapping 5a ⟩ ≡

```
def mapTupleToInt(shape):
   d = len(shape)
   weights = [PROD(shape[(k+1):]) for k in range(d-1)]+[1]

   def mapTupleToInt0(tuple):
      return INNERPROD([tuple,weights])
   return mapTupleToInt0
```

◇

Macro referenced in 15a.

From tuples multi-indices to chain coordinates The set of address `tuples` of $d-cells$ (d-dimensional image elements) within the *mask* is here mapped to the corresponding set of (single) integers associated to the low-level image elements (pixels or voxels, depending on the image dimension and shape), denoted `windowChain`. Such total chain of the mask `window` is then filtered to contain the only coordinates of *white* image elements within the window, and returned as the set of integer cell indices `segmentChain`.

⟨ Window-to-chain mapping 5b ⟩ ≡

```
imageCochain = image_array.reshape(PROD(imageShape))
mapping = mapTupleToInt(imageShape)
windowChain = [mapping(tuple) for tuple in tuples]
segmentChain = [cell for cell in windowChain if imageCochain[cell]==255]
```

◇

Macro referenced in 3b.

2.3 Show segment chain on binary image

Now we need to show visually the selected `segmentChain`, by change the color of its cells from white (255) to middle grey (127). Just remember that `imageCochain` is the linear representation of the image, with number of cells equal to `PROD(imageShape)`. Then the

modified image is restored within `image_array`, and is finally exported to a `.png` image file.

⟨ Change chain color to grey 5c ⟩ ≡

```
for cell in segmentChain: imageCochain[cell] = 127
image_array = imageCochain.reshape(imageShape)
#for k in range(shape[0]):
#  scipy.misc.imsave('tmp/outfile'+str(k).zfill(3)+'.png', image_array[k])
```

◇

Macro referenced in 3b.

3 Construction of (co)boundary operators

A d-image is a *cellular d-complex* where cells are k-cuboids ($0 \leq k \leq d$), i.e. Cartesian products of a number k of 1D intervals, embedded in d-dimensional Euclidean space.

3.1 Reading and writing LAR of image from disk

Since the construction of the *chain complex* supported by a cellular complex with $O(n^d)$ d-cells—and $n = O(10^3)$—may be really time-consuming, it is unquestionably useful to store on disk, once and for all, the topological model of a multidimensional image, i.e. the LAR of its chain complex, and just restore it (or the needed part of it — TODO) when necessary, since *it only depends on the* `shape` *of a d-image*, i.e. from the array arrangement of its d-cells ("hypervoxels").

⟨ Save and restore data object from file 6 ⟩ ≡

```
def dump(object,filename):
   with open(filename, 'wb') as f:
       pickle.dump(object, f)

def load(filename):
   with open(filename, 'rb') as f:
       object = pickle.load(f)
   return object
```

◇

Macro referenced in 15a.

⟨ Save or restore the chain complex of multidimensional image 7a ⟩ ≡

```
def imageChainComplex (shape):
   tokens = str(shape)[1:-1].split(',')
   tokens = [token.strip() for token in tokens]
   filename = "tmp/larimage-" + "-".join(tokens) + ".pickle"

   if os.path.isfile(filename):
      shape, skeletons, operators = loadImageLAR(filename)
   else:
      skeletons = gridSkeletons(list(shape))
      operators = boundaryOps(skeletons)
      imageLAR = (shape, skeletons, operators)
      dump(imageLAR,filename)
      print "filename =",filename
   return shape, skeletons, operators

def loadImageLAR(filename):
   object = load(filename)
   shape, skeletons, operators = object
   return shape, skeletons, operators
◇
```

Macro referenced in 15a.

3.2 LAR chain complex construction

In our first multidimensional implementation of morphological operators through algebraic topology of the image seen as a cellular complex, we compute the whole sequence of characteristic matrices M_k $(0 \leq k \leq d)$ in `BRC` form, and the whole sequence of matrices $[\partial_k]$ $(0 \leq k \leq d)$ in `CSR` form.

Array of characteristic matrices A direct construction of cuboidal complexes is offered, within the `larcc` package, by the `largrid.larCuboids` function.

⟨ Characteristic matrices of multidimensional image 7b ⟩ ≡

```
def larImage(shape):
   """ Compute vertices and skeletons of an image of given shape """
   imageVerts = larImageVerts(shape)
   skeletons = gridSkeletons(list(shape))
   return imageVerts, skeletons
◇
```

Macro referenced in 15a.

Example Consider a (very!) small 3D image of `shape=(2,2,2)`. The data structures returned by the `larImage` function are shown below, where `imageVerts` gives the integer

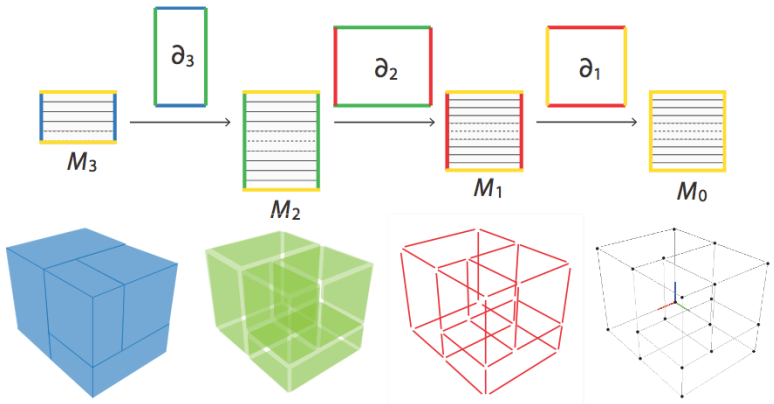

Figure 1: The LAR definition of a chain complex: a sequence of characteristic matrices *and* a sequence of boundary operators.

coordinates of vertices of the 3D (image) complex, and `skeletons` is the list of characteristic matrices M_k $(0 \leq k \leq d)$ in `BRC` form.

⟨ Example of characteristic matrices (and vertices) of multidimensional image 8 ⟩ ≡

```
imageVerts, skeletons = larImage((2,2,2))

print imageVerts,
>>> [[0,0,0],[0,0,1],[0,0,2],[0,1,0],[0,1,1],[0,1,2],[0,2,0],[0,2,1],[0,2,2],
[1,0,0],[1,0,1],[1,0,2],[1,1,0],[1,1,1],[1,1,2],[1,2,0],[1,2,1],[1,2,2],[2,0,
0],[2,0,1],[2,0,2],[2,1,0],[2,1,1],[2,1,2],[2,2,0],[2,2,1],[2,2,2]]

print skeletons[1:],
>>> [
[[0,1],[1,2],[3,4],[4,5],[6,7],[7,8],[9,10],[10,11],[12,13],[13,14],[15,
16],[16,17],[18,19],[19,20],[21,22],[22,23],[24,25],[25,26],[0,3],[1,4],[2,
5],[3,6],[4,7],[5,8],[9,12],[10,13],[11,14],[12,15],[13,16],[14,17],[18,21],
[19,22],[20,23],[21,24],[22,25],[23,26],[0,9],[1,10],[2,11],[3,12],[4,13],[5,
14],[6,15],[7,16],[8,17],[9,18],[10,19],[11,20],[12,21],[13,22],[14,23],[15,
24],[16,25],[17,26]],
[[0,1,3,4],[1,2,4,5],[3,4,6,7],[4,5,7,8],[9,10,12,13],[10,11,13,14],[12,13,15,
16],[13,14,16,17],[18,19,21,22],[19,20,22,23],[21,22,24,25],[22,23,25,26],[0,
1,9,10],[1,2,10,11],[3,4,12,13],[4,5,13,14],[6,7,15,16],[7,8,16,17],[9,10,18,
19],[10,11,19,20],[12,13,21,22],[13,14,22,23],[15,16,24,25],[16,17,25,26],[0,
3,9,12],[1,4,10,13],[2,5,11,14],[3,6,12,15],[4,7,13,16],[5,8,14,17],[9,12,18,
21],[10,13,19,22],[11,14,20,23],[12,15,21,24],[13,16,22,25],[14,17,23,26]],
[[0,1,3,4,9,10,12,13],[1,2,4,5,10,11,13,14],[3,4,6,7,12,13,15,16],[4,5,7,8,13,
14,16,17],[9,10,12,13,18,19,21,22],[10,11,13,14,19,20,22,23],[12,13,15,16,21,
22,24,25],[13,14,16,17,22,23,25,26]]
```

```
]
```
◇

Macro never referenced.

Array of matrices of boundary operators The function `boundaryOps` takes the array of `BRC` reprs of characteristic matrices, and returns the array of `CSR` matrix reprs of boundary operators ∂_k $(1 \leq k \leq d)$.

⟨ CSR matrices of boundary operators 9a ⟩ ≡

```
def boundaryOps(skeletons):
   """ CSR matrices of boundary operators from list of skeletons """
   return [boundary(skeletons[k+1],faces)
      for k,faces in enumerate(skeletons[:-1])]
```

◇

Macro referenced in 15a.

Boundary chain of a k-chain of a d-image

⟨ Boundary of image chain computation 9b ⟩ ≡

```
def imageChainBoundary(shape, operators):
   imageVerts, skeletons = larImage(shape)
   # operators = boundaryOps(skeletons)
   cellNumber = PROD(list(shape))

   def imageChainBoundary0(k):
      csrBoundaryMat = operators[-1]
      facets = skeletons[k-1]

      def imageChainBoundary1(chain):
         ⟨ Boundary*chain product and interpretation 10a ⟩
         boundaryChainModel = imageVerts, [facets[h] for h in boundaryChain]
         return boundaryChainModel,boundaryChain

      return imageChainBoundary1
   return imageChainBoundary0
```

◇

Macro referenced in 15a.

Low-level SpMSpV matrix-chain product A low-level implementation of the product of boundary matrix times the coordinate representation of the image chain under consideration is given below. It was enveloped and protected in this macro because it is strongly dependent on the CSR structures and tools provided by the sparse matrix module of the `scipy` library.

⟨ Boundary*chain product and interpretation 10a ⟩ ≡

```
csrChain = scipy.sparse.csr_matrix((cellNumber,1))
for h in chain: csrChain[h,0] = 1
csrBoundaryChain = matrixProduct(csrBoundaryMat, csrChain)
for h,value in enumerate(csrBoundaryChain.data):
   if MOD([value,2]) == 0: csrBoundaryChain.data[h] = 0
cooBoundaryChain = csrBoundaryChain.tocoo()
boundaryChain = [cooBoundaryChain.row[h]
   for h,val in enumerate(cooBoundaryChain.data) if val == 1]
```

◇

Macro referenced in 9b.

3.3 Visualisation of an image chain and its boundary

***d*-Chain visualisation** The `visImageChain` function given by the macro *Visualisation of an image chain* below.

⟨ Pyplasm visualisation of an image chain 10b ⟩ ≡

```
def visImageChain (shape,chain, imageVerts, skeletons):
   # imageVerts, skeletons = larImage(shape)
   chainLAR = [cell for k,cell in enumerate(skeletons[-1]) if k in chain]
   return imageVerts,chainLAR
```

◇

Macro referenced in 15a.

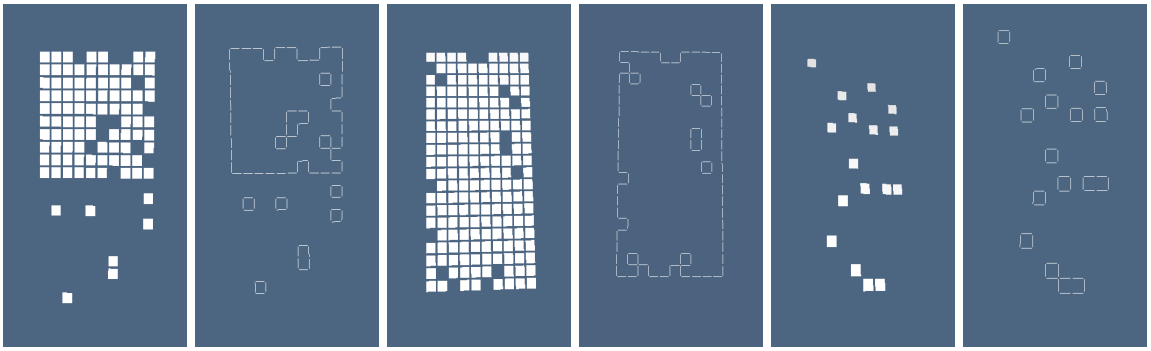

Figure 2: example caption

4 The bulk of morphological imaging

The main part of the implementation of our algebraic construction of morphological operators on *d*-images is given in this section. First we introduce two generalized incidence

operators, respectively named $\mathcal{U}_d$, which stands for *Upper*, and $\mathcal{L}_d$, which stands for *Lower*. Then we implement the standard morphological operators of Dilation ($\mathcal{D}_d$), Erosion ($\mathcal{E}_d$), Opening ($\mathcal{O}_d$), and Closing ($\mathcal{C}_d$).

4.1 "Down" and "Up" operators on Chains

The Up and Down incidence operators are defined as mapping d-cells to $(d+1)$- and $(d+1)$-cells that, respectively, share vertices with them:

$$\mathcal{U}_d : C_d \to C_{d+1}, \qquad \mathcal{D}_d : C_d \to C_{d-1}.$$

Remembering that the characteristic matrix M_d corresponds to the mapping from vertices to d-cells, we have

$$[\mathcal{U}_d] = M_{d+1}\, M_d^t, \qquad [\mathcal{D}_d] = M_{d-1}\, M_d^t.$$

Down and Up operators Just consider that the characteristic matrices M_{d-1}, M_d, M_{d+1}, stored in position $d-1$, d, and $d+1$ of the `skeletons` array, respectively provide the subsets of vertices of proper dimensions incising on each cell. They are used to compute the topological incidence operators, according to the paper [DPS14].

⟨ Down and Up operators 11 ⟩ ≡

```
def larDown(skeletons,d):
   """ Down operator, to multiply a d-chain and return the incident (d-1)-chain """
   csrMd = csrCreate(skeletons[d])
   csrMinus = csrCreate(skeletons[d-1])
   csrDown = matrixProduct(csrMinus,csrTranspose(csrMd))
   return csrDown

def larUp(skeletons,d):
   """ Up operator, to multiply a d-chain and return the incident (d+1)-chain """
   csrMd = csrCreate(skeletons[d])
   csrPlus = csrCreate(skeletons[d+1])
   csrUp = matrixProduct(csrPlus,csrTranspose(csrMd))
   return csrUp
```

◇

Macro referenced in 15a.

The $\mathcal{UUD}$ operator A generic $\mathcal{UUD} : C_{d-1} \to C_d$ operator is given to be associated with the boundary ∂_2, in order to return the $\mathcal{ERO} \cup \mathcal{DIL}$ (TODO:correct) of a d-chain.

⟨ UUD operator: maps Down-UP-UP its input chains 12a ⟩ ≡

```
def UUD(skeletons,d):
   """ Compute the morphological operator UUP.
      Return a CSR matrix to be applied to the coordinate representation of a chain
   """
   D = larDown(skeletons,d)
   U1 = larUp(skeletons,d-1)
   U2 = larUp(skeletons,d)
   UUDout = matrixProduct(U2,matrixProduct(U1,D))
   return D,U1,U2,UUDout
```

◇

Macro referenced in 15a.

A pair of similar function `testAlgebraicMorphology` and `testAlgebraicMorphologyStepByStep` are given here, both accepting the same (redundant) inputs, in order to test our algebraic approach to mathematical morphology. In particular:

⟨ Testing the algebraic morphology 12b ⟩ ≡

```
def testAlgebraicMorphology (solid, b_rep, chain, imageVerts, skeletons):
   d = len(skeletons)-1
   D,U1,U2,csrMorphOp = UUD(skeletons,d-1)
   outputChain = chainTransform(skeletons,1,chain,csrMorphOp,d)
   chainLAR = [cell for k,cell in enumerate(skeletons[d]) if k in outputChain]
   model2 = (imageVerts,chainLAR)
   return imageVerts,chainLAR
```

◇

Macro referenced in 15a.

⟨ Testing the algebraic morphology method step-by-step 13a ⟩ ≡

```
def testAlgebraicMorphologyStepByStep (solid, b_rep, chain, imageVerts, skeletons):
   d = len(skeletons)-1
   D,U1,U2,csrMorphOp = UUD(skeletons,d-1)

   VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(solid)))
   VIEW(COLOR(MAGENTA)(STRUCT(MKPOLS(b_rep))))

   outputChain = chainTransform(skeletons,d-1,chain,D,d-1)
   chainLAR = [cell for k,cell in enumerate(skeletons[0]) if k in outputChain]
   model0 = (imageVerts,chainLAR)
   VIEW(COLOR(RED)(STRUCT(MKPOLS(model0))))

   M2 = csrCreate(skeletons[d])

   outputChain = chainTransform(skeletons,0,outputChain,M2,d)
   chainLAR = [cell for k,cell in enumerate(skeletons[d]) if k in outputChain]
   model2 = (imageVerts,chainLAR)
   VIEW(COLOR(YELLOW)(STRUCT(MKPOLS(model2))))

   return imageVerts,chainLAR
```

◇

Macro referenced in 15a.

Boundary chain mapping The function `chainTransform` takes as input a `chain` of dimension `d` (actually the boundary of a chain of dimension $d+1$), the `skeletons` array of `BRC` representations of matrices M_k $(0 \leq k \leq d+1)$, and the number n that $(d+1)$-cells resulting by multiplication with the operator's matrix `csrMatrix` must ahare with they incident d-cells in order to be included in the output of the considered algebraic morphology operator.

⟨ csrMatrix*chain product and interpretation 13b ⟩ ≡

```
def chainTransform(skeletons,d,chain,csrMatrix,n):
   # n: number of vertices shared in the incidence relation
   cellNumber = len(skeletons[d])
   csrChain = scipy.sparse.csr_matrix((cellNumber,1))
   for h in chain: csrChain[h,0] = 1
   cooOutChain = matrixProduct(csrMatrix, csrChain).tocoo()
   outChain = [cooOutChain.row[h]
      for h,val in enumerate(cooOutChain.data) if int(val) >= n]
   return outChain
```

◇

Macro referenced in 15a.

Example

⟨ Algebraic dilation minus erosion 14a ⟩ ≡

```
D1 = larDown(skeletons,1)
U0 = larUp(skeletons,0)
U1 = larUp(skeletons,1)
UUD1 = UUD(skeletons,1)
```

◇

Macro never referenced.

4.2 Maximal $(d-2)$-chain extraction

⟨ Extract the maximal $(d-2)$-chain from a $(d-1)$-chain 14b ⟩ ≡

◇

Macro never referenced.

4.3 $(d-1)$-Star of a $(d-2)$-chain computation

⟨ Compute the $(d-1)$-star of a $(d-2)$-chain 14c ⟩ ≡

◇

Macro never referenced.

4.4 d-Star of a $(d-1)$-chain computation

⟨ Compute the d-star of a $(d-1)$-chain 14d ⟩ ≡

◇

Macro never referenced.

4.5 Dilation and erosion computation

4.6 Opening and closing computation

5 Exporting the `morph` module

Exporting the morph module

"lib/py/morph.py" 15a ≡

```
""" LAR implementation of morphological operators on multidimensional images."""
⟨ Initial import of modules 15b ⟩
⟨ Generation of random image 2a ⟩
⟨ Save and restore data object from file 6 ⟩
⟨ Save or restore the chain complex of multidimensional image 7a ⟩
⟨ Tuples to integers mapping 5a ⟩
⟨ Generation of a masking window 3b ⟩
⟨ Characteristic matrices of multidimensional image 7b ⟩
⟨ CSR matrices of boundary operators 9a ⟩
⟨ csrMatrix*chain product and interpretation 13b ⟩
⟨ Pyplasm visualisation of an image chain 10b ⟩
⟨ Boundary of image chain computation 9b ⟩
⟨ Down and Up operators 11 ⟩
⟨ UUD operator: maps Down-UP-UP its input chains 12a ⟩
⟨ Testing the algebraic morphology method step-by-step 13a ⟩
⟨ Testing the algebraic morphology 12b ⟩
◇
```

The set of importing commends needed by test files in this module is given in the macro below.

⟨ Initial import of modules 15b ⟩ ≡

```
import sys,os
import scipy.misc, numpy, pickle
from numpy.random import randint
from pyplasm import *

""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')

⟨ Import the module (15c largrid ) 17a ⟩
◇
```

Macro referenced in 15a, 16a.

6 Morphological operations examples

6.1 2D image masking and boundary computation

Test example The `larcc.morph` API is used here to generate a random black and white image, with an *image segment* selected and extracted by masking, then colored in middle grey, and exported to an image file. The `shape` and `structure` variables must contain two tuples, of equal `len`, of powers of 2, used toy define the size of random blocks in the generated image. The `window` variable is used to define the portion of the image where the *segmentChain* of white pixels (or voxels, for 3-images) is computed.

"test/py/morph/test01.py" 16a ≡

```
⟨ Initial import of modules 15b ⟩
⟨ Import the module (16b morph ) 17a ⟩
shape = 64,64
structure = 8,8
assert len(shape) == len(structure)
imageVerts = larImageVerts(shape)
_, skeletons, operators = imageChainComplex (shape)
image_array = randomImage(shape, structure, 0.05)
minPoint, maxPoint = (0,0), (64,64)
window = minPoint, maxPoint
segmentChain = setMaskWindow(window,image_array)

solid = visImageChain (shape,segmentChain, imageVerts, skeletons)
b_rep,boundaryChain = imageChainBoundary(shape, operators)(2)(segmentChain)

stepwiseTest = True
if stepwiseTest:
   model = testAlgebraicMorphologyStepByStep(solid, b_rep,
            boundaryChain, imageVerts, skeletons)
else:
   model = testAlgebraicMorphology(solid, b_rep,
            boundaryChain, imageVerts, skeletons)
VIEW(STRUCT(MKPOLS(model)))

V = model[0]
M = AA(tuple)(model[1])
S = AA(tuple)(solid[1])
B = AA(tuple)(b_rep[1])
D = list(set(S).union(M))
E = list(set(S).difference(M))
M,S,B,D,E = (AA(AA(list)))([M,S,B,D,E])
M2 = STRUCT(MKPOLS((V,M)))
S2 = STRUCT(MKPOLS((V,S)))

S2 = COLOR(CYAN)(STRUCT(MKPOLS((V,S))))
B1 = COLOR(MAGENTA)(STRUCT(MKPOLS((V,B))))
D2 = COLOR(YELLOW)(STRUCT(MKPOLS((V,D))))
E2 = COLOR(WHITE)(STRUCT(MKPOLS((V,E))))
VIEW(STRUCT([D2,S2,E2,B1]))

VIEW(STRUCT([D2,S2,B1]))
VIEW(STRUCT([S2,E2,B1]))

◇
```

A Utilities

A.1 Importing a generic module

First we define a parametric macro to allow the importing of `larcc` modules from the project repository `lib/py/`. When the user needs to import some project's module, she may call this macro as done in Section **??**.

⟨ Import the module 17a ⟩ ≡

```
import @1
from @1 import *
```

◇

Macro referenced in 15b, 16a, 17b.

Importing a module A function used to import a generic `lacccc` module within the current environment is also useful.

⟨ Function to import a generic module 17b ⟩ ≡

```
def importModule(moduleName):
    ⟨ Import the module (17c moduleName ) 17a ⟩
```

◇

Macro never referenced.

References

[CL13] CVD-Lab, *Linear algebraic representation*, Tech. Report 13-00, Roma Tre University, October 2013.

[DPS14] Antonio Dicarlo, Alberto Paoluzzi, and Vadim Shapiro, *Linear algebraic representation for topological structures*, Comput. Aided Des. **46** (2014), 269–274.