Imaging Morphology with LAR *

Alberto Paoluzzi

March 1, 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 1, 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=0.1):
   """ Generation of random image of given shape and structure.
      Return a scipy.ndarray(shape)
   """
   rows, columns = shape
   rowSize, columnSize = structure
   random_array = randint(0, 255, size=(rowSize, columnSize))
   image_array = numpy.zeros((rows, columns))
   ⟨Generation of bulk array structure 2b⟩
   ⟨Generation of random artifacts 2c⟩
   return image_array
```
◇

Macro referenced in 9a.

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 ⟩ ≡

```
for i in range(rowSize):
   for j in range(columnSize):
      for h in range(i*rowSize,i*rowSize+rowSize):
         for k in range(j*columnSize,j*columnSize+columnSize):
            if random_array[i,j] < 127:
               image_array[h,k] = 0
            else:
               image_array[h,k] = 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 2c ⟩ ≡

```
noiseQuantity = rows*columns*noiseFraction
k = 0
while k < noiseQuantity:
   i,j = randint(rows),randint(columns)
   if image_array[i,j] == 0: image_array[i,j] = 255
   else: image_array[i,j] = 0
   k += 1
scipy.misc.imsave('./outfile.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 window `maxPoint` (maximum indices). A list of multi-index tuples, contained in the `window` variable, is generated by the function `setMaskWindow` below.

$\langle$ Generation of a masking window 3a $\rangle \equiv$

```
def setMaskWindow(window,image_array):
   minPoint, maxPoint = window
   imageShape = list(image_array.shape)
   ⟨ Generation of multi-index window 3b ⟩
   ⟨ Window-to-chain mapping 5a ⟩
   ⟨ Change chain color to grey 5b ⟩
   return segmentChain
◇
```

Macro referenced in 9a.

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

$\langle$ Generation of multi-index window 3b $\rangle \equiv$

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

Macro referenced in 3a.

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 4 ⟩ ≡

```
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 9a.

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 5a ⟩ ≡

```
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 3a.

2.3 Show segment chain from 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 5b⟩ ≡

```
for cell in segmentChain: imageCochain[cell] = 127
image_array = imageCochain.reshape(imageShape)
scipy.misc.imsave('./outfile.png', image_array)
◇
```

Macro referenced in 3a.

3 Construction of (co)boundary operators

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

3.1 LAR chain complex construction

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

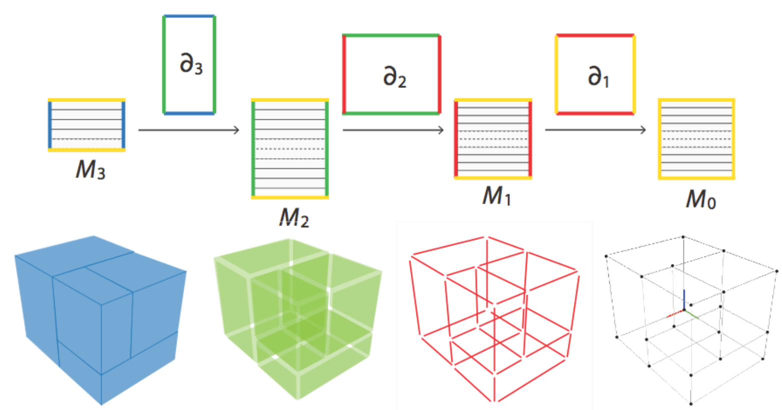

Figure 1: The LAR definition of a chain complex: a sequence of characteristic matrices *and* a sequence of boundary operators.

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 6 ⟩ ≡

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

Macro referenced in 9a.

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 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 7a ⟩ ≡

```
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 $(1 \leq k \leq d)$.

$\langle$ CSR matrices of boundary operators 7b $\rangle \equiv$

```
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])]
```

$\diamond$

Macro referenced in 9a.

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

$\langle$ Boundary of image chain computation 8a $\rangle \equiv$

```
def imageChainBoundary(shape):
    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 8b 〉
            boundaryChainModel = imageVerts, [facets[h] for h in boundaryCells]
            return boundaryChainModel

        return imageChainBoundary1
    return imageChainBoundary0
```

$\diamond$

Macro referenced in 9a.

$\langle$ Boundary*chain product and interpretation 8b $\rangle \equiv$

```
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()
boundaryCells = [cooBoundaryChain.row[h]
    for h,val in enumerate(cooBoundaryChain.data) if val == 1]
```

$\diamond$

Macro referenced in 8a.

3.2 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 8c ⟩ ≡

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

Macro referenced in 9a.

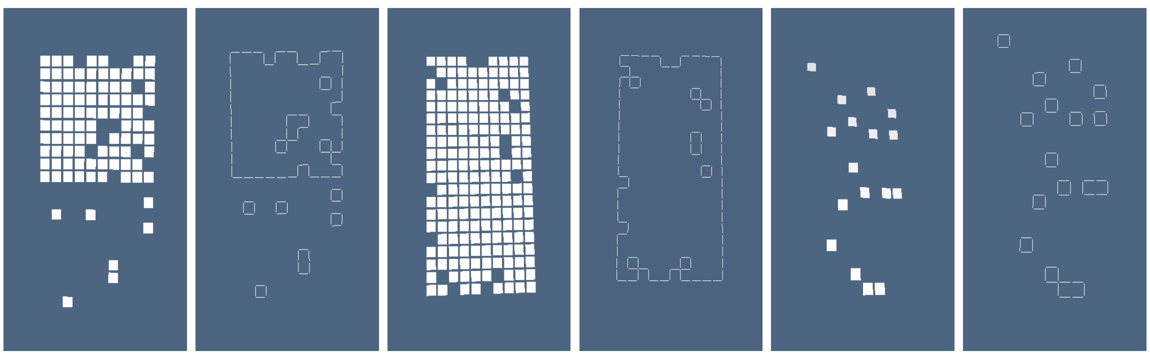

Figure 2: example caption

4 Exporting the `morph` module

Exporting the morph module

"lib/py/morph.py" 9a ≡

```
""" LAR implementation of morphological operators on multidimensional images."""
⟨ Initial import of modules 9b ⟩
⟨ Generation of random image 2a ⟩
⟨ Tuples to integers mapping 4 ⟩
⟨ Generation of a masking window 3a ⟩
⟨ Characteristic matrices of multidimensional image 6 ⟩
⟨ CSR matrices of boundary operators 7b ⟩
⟨ Pyplasm visualisation of an image chain 8c ⟩
⟨ Boundary of image chain computation 8a ⟩
◇
```

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

⟨ Initial import of modules 9b ⟩ ≡

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

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

⟨ Import the module (9c largrid ) 10c ⟩
◇
```

Macro referenced in 9a, 10a.

5 Morphological operations examples

5.1 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.

"test/py/morph/test01.py" 10a ≡

```
⟨ Initial import of modules 9b ⟩
⟨ Import the module (10b morph ) 10c ⟩
rows, columns = 100,100
rowSize, columnSize = 10,10
shape = (rows, columns)
structure = (rowSize, columnSize)
image_array = randomImage(shape, structure, 0.3)
minPoint, maxPoint = (20,20), (40,30)
window = minPoint, maxPoint
segmentChain = setMaskWindow(window,image_array)

solid = visImageChain (shape,segmentChain)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(solid)))
b_rep = imageChainBoundary(shape)(2)(segmentChain)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(b_rep)))
◇
```

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 10c ⟩ ≡

```
import @1
from @1 import *
◇
```

Macro referenced in 9b, 10a, 11a.

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 11a ⟩ ≡

```
def importModule(moduleName):
   ⟨ Import the module (11b moduleName ) 10c ⟩
◇
```

Macro never referenced.

References

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