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 6a.

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.

⟨ Generation of a masking window 3a ⟩ ≡
```
def setMaskWindow(window,image_array):
   minPoint, maxPoint = window
   imageShape = list(image_array.shape)
   ⟨ Generation of multi-index window 3b ⟩
   ⟨ Window-to-chain mapping 4a ⟩
   ⟨ Change chain color to grey 4b ⟩
   return segmentChain
◇
```
Macro referenced in 6a.

⟨ Generation of multi-index window 3b ⟩ ≡

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

Macro referenced in 3a.

From tuples multi-indices to chain coordinates The set of `tuples` of all pixels (or 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 4a ⟩ ≡

```
d = len(imageShape)
weights = [PROD(imageShape[(k+1):]) for k in range(d-1)]+[1]
imageCochain = image_array.reshape(PROD(imageShape))
windowChain = [INNERPROD([index,weights]) for index in tuples]
segmentChain = [cell for cell in windowChain if imageCochain[cell]==255]
◇
```

Macro referenced in 3a.

2.2 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 4b ⟩ ≡

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

Macro referenced in 3a.

Test example The macros previously defined are here composed to generate a random black and white image, with a *image segment* (in a fixed position window within the image) extracted, colored in middle grey, and exported to an image file.

"test/py/morph/test01.py" 4c $\equiv$

```
⟨ Initial import of modules 6b ⟩
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)

if __name__== "__main__":
   model = visImageChain (shape,segmentChain)
   VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
   model = visImageChainBoundary (shape,segmentChain)
   VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS(model)))
◇
```

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.

A direct construction of cuboidal complexes is offered in `larcc` by the `largrid` module. The `visImageChain` function given by the macro *Visualisation of an image chain* below.

3.1 Visualisation of an image chain

d-Chain visualisation

⟨ Pyplasm visualisation of an image chain 5a ⟩ $\equiv$

```
def visImageChain (shape,chain):
   imageShape = list(shape)
   model = larCuboids(imageShape)
   imageVerts = model[0]
   imageLAR = model[1]
   chainLAR = [cell for k,cell in enumerate(imageLAR) if k in chain]
   return imageVerts,chainLAR
◇
```

Macro referenced in 6a.

Boundary visualisation of a d-chain

⟨Boundary visualisation of an image chain 5b⟩ ≡

```
def visImageChainBoundary (shape,chain):
   imageShape = list(shape)
   model = larCuboids(imageShape)
   imageVerts = model[0]
   skeletons = gridSkeletons(imageShape)
   facets = skeletons[-2]
   csrBoundaryMat = gridBoundaryMatrices(imageShape)[-1]
   csrChain = scipy.sparse.csr_matrix((PROD(imageShape),1))
   for k in chain: csrChain[k,0] = 1
   csrBoundaryChain = matrixProduct(csrBoundaryMat, csrChain)
   for k,value in enumerate(csrBoundaryChain.data):
      if MOD([value,2]) == 0: csrBoundaryChain.data[k] = 0
   cooBoundaryChain = csrBoundaryChain.tocoo()
   boundaryCells = [cooBoundaryChain.row[k]
      for k,val in enumerate(cooBoundaryChain.data) if val == 1]
   return imageVerts,[facets[k] for k in boundaryCells]
```

◇

Macro referenced in 6a.

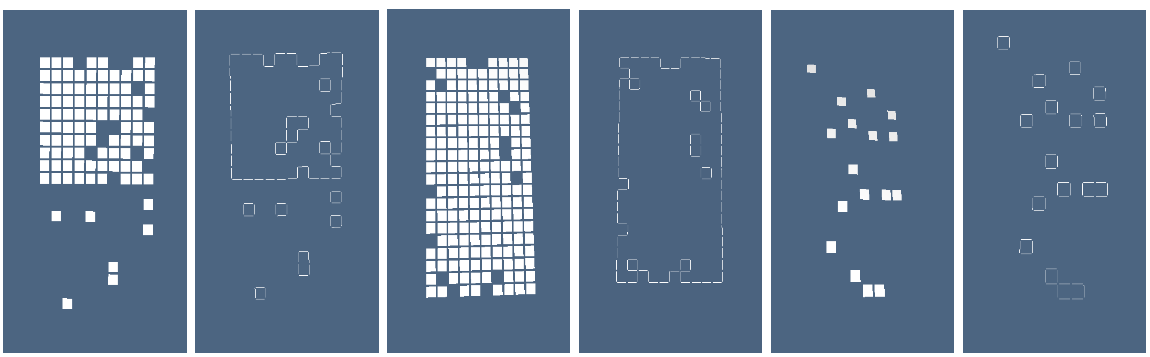

Figure 1: example caption

4 Exporting the `morph` module

Exporting the morph module

"lib/py/morph.py" 6a ≡

```
""" LAR implementation of morphological operators on multidimensional images."""
⟨ Initial import of modules 6b ⟩
⟨ Generation of random image 2a ⟩
⟨ Generation of a masking window 3a ⟩
⟨ Pyplasm visualisation of an image chain 5a ⟩
⟨ Boundary visualisation of an image chain 5b ⟩
◇
```

⟨ Initial import of modules 6b ⟩ ≡

```

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 (6c largrid ) 7a ⟩
⟨ Import the module (6d morph ) 7a ⟩
◇
```

Macro referenced in 4c, 6a.

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 7a ⟩ ≡

```
import @1
from @1 import *
◇
```

Macro referenced in 6b, 7b.

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 7b ⟩ ≡

```
def importModule(moduleName):
    ⟨ Import the module (7c moduleName ) 7a ⟩
◇
```

Macro never referenced.

References

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