Boundary operators on LAR *

Alberto Paoluzzi

February 14, 2016

Abstract

The various versions of boundary operators on Linear Algebraic Representation of cellular complexes are developed in this module, in order to maintain under focus their proper development, including the possible special cases.

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. February 14, 2016

1 Introduction

In the current `LarLib` implementation, we have to distinguish between between dimension-independent, dimension-dependent, signed and non-signed operators. Therefore a code refactoring of `LarLib`—related to boundary/coboundary operators—started here, with the aim of both providing a precise mathematical definition within the LAR framework, and to simplify and generalise the implemented algorithms.

2 Implementation

We start this section by making a distinction between the (matrices of) boundary operators for the linear spaces C_k of chains over the field $\mathbb{Z}_2 = \{0, 1\}$ and over the field $\mathbb{Z}$ of integer numbers. We call either *non-signed* or *signed* the corresponding boundary operators, respectively, since the matrix elements take values within the sets $\{-1, 0, +1\}$ or $\{0, 1\}$, correspondingly. Of course, the associated matrices of *coboundary* operators are their transpose matrices.

2.1 Non-signed operators

For several computations, the knowledge of the matrices of non-signed boundary operators is sufficient. Therefore we will use such tool wherever possible, since its computation is much faster in term of computing time.

In the following we provide be binary operator matrices provided by two implementations, respectively named `boundary` and `boundary1`. The first one works correctly only with convex cells; the second one works also with non-convex but path-connected cells.

2.1.1 Dimension-independence

As we show in the following, in order to compute the non-signed boundary operator ∂_d, it it sufficient to have knowledge of the M_d and M_{d-1} characteristic matrices of d-cells and their $(d-1)$-facets, at least in the case of cellular complexes with convex cells. Conversely, for more general non-convex but simply-connected cells, also the M_{d-2} matrix is needed.

Convex-cells The algorithm used is pretty easy to present. The compressed characteristic matrices of d-cells and $(d-1)$-cells, denoted as `cells` and `facets`, respectively, are first put in `csr` format as `csrCV` and `csrFV`. Then the incidence matrix `csrFC` in compressed sparse row format is computed by matrix product of the compressed characteristic matrices.

The element (i, j) of this matrix provides the number of vertices in the intersection of *facet* i and *cell* j, whereas the number of non-zero elements in each `csrFV` *row* gives the number of vertices of the facet represented by the row, and is stored in `facetLengths`.

The `boundary` function—to be used only with dimension-independent LAR convex cells—is written efficiently in the following script, by using only the standard functions and attributes of the `scipy.sparse` module.

The variable `facetCoboundary` stores in a list, for every facet (`for h in range(m)`) the list of cells in its *coboundary*, to be stored in the output `csr_matrix` boundary matrix as column indices of elements with non-zero (i.e. 1) value.

Notice that both the computation of `facetCoboundary` contents, and the output of the compressed boundary matrix, are performed in the most efficient way—according to the internal design of the scipy's `csr` sparse data structure.

⟨ convex-cells boundary operator 3a ⟩ ≡

```
""" convex-cells boundary operator --- best implementation """
def boundary(cells,facets):
    lenV = max(CAT(cells))+1
    csrCV = csrCreate(cells,lenV)
    csrFV = csrCreate(facets,lenV)
    csrFC = csrFV * csrCV.T
    facetLengths = [csrFacet.getnnz() for csrFacet in csrFV]
    m,n = csrFC.shape
    facetCoboundary = [[csrFC.indices[csrFC.indptr[h]+k]
        for k,v in enumerate(csrFC.data[csrFC.indptr[h]:csrFC.indptr[h+1]])
            if v==facetLengths[h]] for h in range(m)]
    indptr = [0]+list(cumsum(AA(len)(facetCoboundary)))
    indices = CAT(facetCoboundary)
    data = [1]*len(indices)
    return csr_matrix((data,indices,indptr),shape=(m,n),dtype='b')
```

◇

Macro referenced in 4a.

Path-connected cells

⟨ path-connected-cells boundary operator 3b ⟩ ≡

```
""" path-connected-cells boundary operator """
import larlib
import larcc
from larcc import *

def csrBoundaryFilter1(csrBoundaryBoundaryMat,cells,facets,faces,lenV,FE):
    out = boundary(cells,facets)

    def csrRowSum(h): return sum(out.data[out.indptr[h]:out.indptr[h+1]])

    unreliable = [h for h in range(out.shape[0]) if csrRowSum(h) > 2]
    if unreliable != []:
        print "\n>>>>> unreliable =",unreliable
```

```
            for row in unreliable:
                for j in range(len(cells)):
                    if out[row,j] == 1:
                        csrCFE = csrBoundaryBoundaryMat[:,j]
                        cooCFE = csrCFE.tocoo()
                        flawedCells = [cooCFE.row[k] for k,datum in enumerate(cooCFE.data)
                            if datum>2]
                        if all([facet in flawedCells  for facet in FE[row]]):
                            out[row,j]=0
        return out

    def boundary1(CV,FV,EV):
        lenV = max(CAT(CV))+1
        csrBBMat = boundary(FV,EV) * boundary(CV,FV)
        print "\ncsrBBMat =",csrBBMat,"\n"
        FE = larcc.crossRelation(lenV,FV,EV)
        return csrBoundaryFilter1(csrBBMat,CV,FV,EV,lenV,FE)
    ◇
```

Macro referenced in 4a.

2.2 Signed operators

3 Exporting

"larlib/larlib/boundary.py" 4a ≡

```
    """ boundary operators """
    from larlib import *
    ⟨ convex-cells boundary operator 3a ⟩
    ⟨ path-connected-cells boundary operator 3b ⟩
    ◇
```

4 Testing

5 Non-signed operators

Correct boundary extraction example The `boundary.boundary()` operator is applied here to a cellular 2-complex of convex cells, producing correct result.

"test/py/boundary/test01.py" 4b ≡

```
    """ testing boundary operators (correct result) """
    from larlib import *

    filename = "test/svg/inters/boundarytest0.svg"
    lines = svg2lines(filename)
```

```
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,FV,EV,polygons = larFromLines(lines)
VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,0.2))
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,[EV[e] for e in boundaryCells(FV,EV)],))))
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,FV,EV))))

boundaryOp = boundary(FV,EV)

for k in range(1,len(FV)+1):
    faceChain = k*[1]
    BF = chain2BoundaryChain(boundaryOp)(faceChain)
    VIEW(STRUCT(MKPOLS((V,[EV[e] for e in BF]))))
```
◇

Wrong boundary extraction example The `boundary.boundary()` operator is applied here to a cellular 2-complex wih some non-convex cells, producing incorrect results. In such cases a correct result may be produced only by chance (sometimes this happens). So, be careful to use it only when the precondition (of cell convexity) is everywhere verified.

"test/py/boundary/test02.py" 5 ≡
```
""" testing boundary operators (wrong result) """
from larlib import *

filename = "test/svg/inters/boundarytest1.svg"
lines = svg2lines(filename)
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,FV,EV,polygons = larFromLines(lines)
VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,0.2))

boundaryOp = boundary1(FV,EV,VV)  # <<======  NB
#boundaryOp = boundary(FV,EV)  # <<======  NB
BF = chain2BoundaryChain(boundaryOp)([1]*len(FV))

VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,[EV[e] for e in BF])))) # ERROR !!!
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,FV,EV)))) # ERROR !!!

for k in range(1,len(FV)+1):
    faceChain = k*[1]
    boundaryChain = chain2BoundaryChain(boundaryOp)(faceChain)
    VIEW(STRUCT(MKPOLS((V,[EV[e] for e in boundaryChain]))))
```

◇

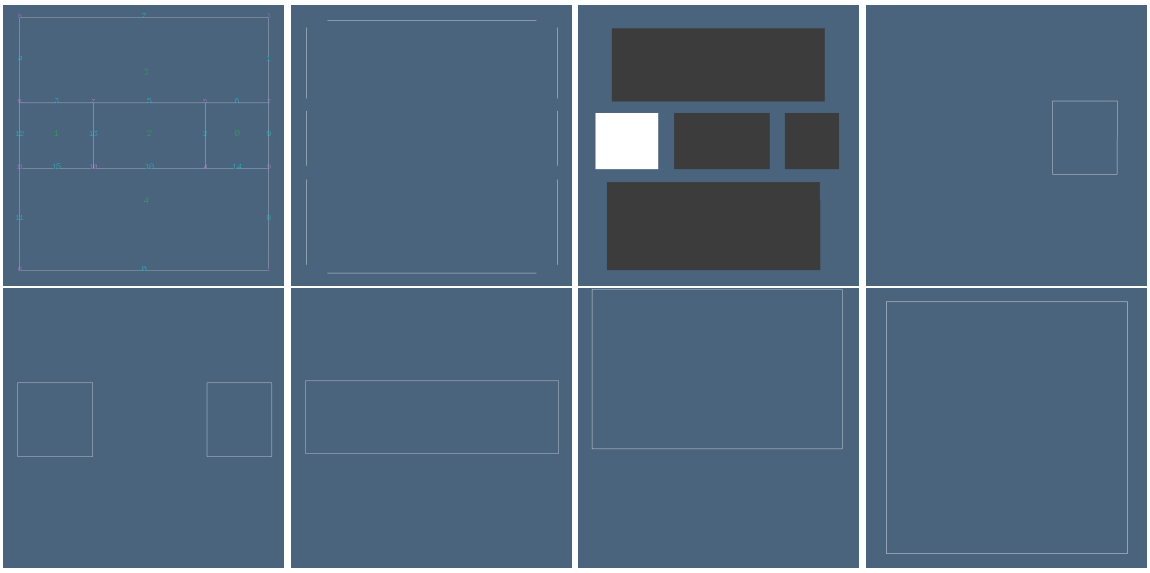

Figure 1: Convex-cell 2-complex. (a) Indexing of 0-,1-,and 2-cells; (b) exploded 2-boundary cells; (c) exploded 2-cells; (d) boundary of a singleton 2-chain; (e–h) boundaries of some 2-chains.

Example Comparison of two implementations of the ∂ operator. Notice the difference between the penultimate rows. In particular, the penultimate row of the matrix generated by `boundary(FV,EV)` is plain wrong. It means that the edge e_{10} is shared by all the (three) 2-cells of the complex. Conversely, it is well known that, for a solid complex, i.e. a d-complex embedded in $\mathbb{E}^d$, every $(d-1)$-facet may be shared by no more than 2 d-cells. The resulting boundary of the total chain $[f_0, f_1, f_2]$ codified in coordinates as $[1, 1, 1]$, and shown in Figure 2d, is sonsequently incorrect.

```
In [1]: boundary(FV,EV).todense()
Out[1]:
matrix([[0, 1, 0],
        [0, 0, 1],
        [1, 0, 1],
        [1, 0, 1],
        [0, 1, 1],
        [0, 1, 0],
        [1, 0, 1],
        [0, 0, 1],
        [0, 0, 1],
        [0, 1, 0],
        [1, 1, 1],
        [0, 1, 1]])
```

```
In [2]: boundary1(FV,EV,VV).todense()
Out[2]:
matrix([[0, 1, 0],
        [0, 0, 1],
        [1, 0, 1],
        [1, 0, 1],
        [0, 1, 1],
        [0, 1, 0],
        [1, 0, 1],
        [0, 0, 1],
        [0, 0, 1],
        [0, 1, 0],
        [1, 1, 0],
        [0, 1, 1]])
```

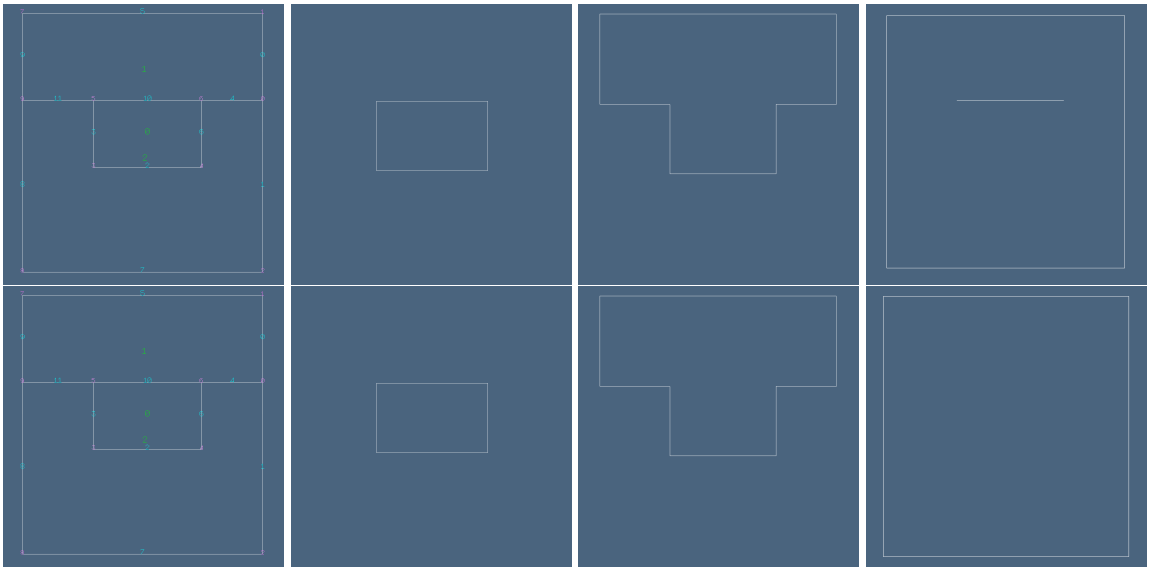

Figure 2: Non-working (i.e. *wrong*) example with `boundary`. (a) Indexing of 0-,1-,and 2-cells; (b) boundary of a singleton 2-chain; (c) exploded 2-cells; (d) boundary of a singleton 2-chain. Working (i.e. *exact*) example using `boundary1`: (e–h) as above.

References

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