Boundary operators on LAR [*]

Alberto Paoluzzi

February 19, 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 19, 2016

1 Introduction

In the current `LarLib` implementation, we have to distinguish between between dimension-independent, dimension-dependent, oriented and non-oriented 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-oriented* or *oriented* 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-oriented operators

For several computations, the knowledge of the matrices of non-oriented 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-oriented 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 3⟩ ≡

```
""" 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 6a.

Non-convex LAR cells A more general `boundary1` operator is given in the following, aiming at compute the boundary matrix for general non-convex cellular decompositions, including *multiply connected* LAR models. Notice that in this case an input triple made by `CV`, `FV`, and `EV` is needed, where—more in general embedded in $\mathbf{E}^d$—they stand for the (binary compressed) characteristic matrices M_d, M_{d-1}, and M_{d-2}.

First the `boundary` operator for the convex case is computed within the `out` variable of `csr_matrix` type. Then every `out` row (i.e. every $(d-1)$-facet of the d-complex) is tested for *reliability*, since every $(d-1)$-face can be shared by *at most two* d-cells in a d-complex . When this condition is not satisfied, deeper tests are needed to understand what row elements must be forced to value 1, since the $(d-1)$-face itself is a subset, but not actually a facet, of the corresponding d-cell.

In presence of some "unreliable" facets, the matrix `csrBBMat` of the operator $\partial_{d-1} \circ \partial_d$ and the relation `FE` between faces of dimensions $d-1$ and $d-2$ are computed. Now, let us notice that the columns of `csrBBMat` report the number of incidences of the $d-2$ faces (as belonging to $(d-1)$-facets embedded on the boundary) and d-cells (that are associated to such matrix columns). Hence, in a regular (convex) d-complex, such numbers are always even, and in $\mathbb{Z}_2$ arithmetic are reduced to zero, in order to satisfy the fundaments equation $\partial\partial = 0$.

Conversely, with non-convex LAR cells, some incidence numbers may get odd values, due to the non-strict coincidence between cell facets and vertex subsets. Therefore, for "unreliable" h rows (facets) the `csrBBMat` columns tracked by ones in $[\partial_d]$ are checked, looking for elements of (h,k) indices with value greater that 2.

$\langle$ path-connected-cells boundary operator 4 $\rangle \equiv$

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

def csrBoundaryFilter1(unreliable,out,csrBBMat,cells,FE):
    for row in unreliable:
        for j in range(len(cells)):
            if out[row,j] == 1:
                cooCE = csrBBMat.T[j].tocoo()
                flawedCells = [cooCE.col[k] for k,datum in enumerate(cooCE.data)
                    if datum>2]
                if all([facet in flawedCells  for facet in FE[row]]):
                    out[row,j]=0
    return out

def csrBoundaryFilter2(unreliable,out,csrBBMat,cells,FE):
    for col in unreliable:
        print csrBBMat[:,col]
        cooCE = csrBBMat.T[col].tocoo()
        flawedCells = [cooCE.col[k] for k,datum in enumerate(cooCE.data)
                    if datum>2]
        print  flawedCells,
        for j in range(out.shape[0]):
            if out[j,col] == 1:
                if all([facet in flawedCells  for facet in FE[j]]):
                    out[j,col]=0
    return out

def boundary1(CV,FV,EV):
    out = boundary(CV,FV)
    def csrRowSum(h):
        return sum(out.data[out.indptr[h]:out.indptr[h+1]])
    unreliable = [h for h in range(len(FV)) if csrRowSum(h) > 2]
    if unreliable != []:
        csrBBMat = boundary(FV,EV) * boundary(CV,FV)
        print "\ncsrBBMat =",csrBBMat.todense(),"\n"
        lenV = max(CAT(CV))+1
        FE = larcc.crossRelation0(lenV,FV,EV)
        out = csrBoundaryFilter1(unreliable,out,csrBBMat,CV,FE)
    return out

def boundary2(CV,FV,EV):
    out = boundary1(CV,FV,EV)
    lenV = max(CAT(CV))+1
    VV = AA(LIST)(range(lenV))
    csrBBMat = scipy.sparse.csc_matrix(boundary(FV,EV) * boundary1(CV,FV,VV))
    print "\ncsrBBMat =",csrBBMat.todense(),"\n"
    def csrColCheck(h):
        return any([val for val in csrBBMat.data[csrBBMat.indptr[h]:csrBBMat.indptr[h+1]] if va
    unreliable = [h for h in range(len(CV)) if csrColCheck(h)]
    print "\nunreliable =",unreliable,"\n"
    if unreliable != []:
        FE = larcc.crossRelation0(lenV,FV,EV)
        out = csrBoundaryFilter2(unreliable,out,csrBBMat,CV,FE)
    return out
◇
```

Macro referenced in 6a.

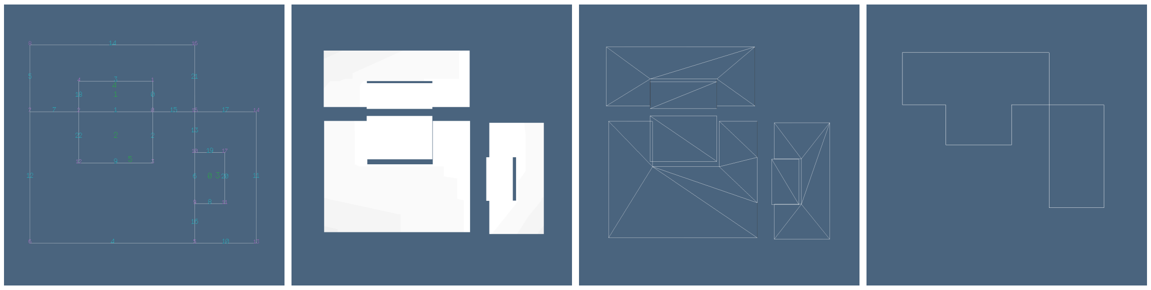

Figure 1: Non-convex LAR 2-complex with (two) 1-cells that are subsets of 2-cells without being their facets. Correctly disentangled by the `boundary1()` function: (a) Indexing of 0-, 1-, and 2-cells; (b) exploded 2-cells; (c) triangulated and exploded 2-cells; (d) boundary of the 2-chain `[1,1,1,1,1,0]`.

⟨ From cells and facets to boundary cells 5 ⟩ ≡

```
    def totalChain(cells):
        return csr_matrix(len(cells)*[[1]])

    def boundaryCells(cells,facets):
        csrBoundaryMat = boundary(cells,facets)
        csrChain = csr_matrix(totalChain(cells))
        csrBoundaryChain = csrBoundaryMat * csrChain
        out = [k for k,val in enumerate(csrBoundaryChain.data.tolist()) if val == 1]
        return out

    def boundaryCells1(cells,facets,faces):
        csrBoundaryMat = boundary1(cells,facets,faces)
        csrChain = csr_matrix(totalChain(cells))
        csrBoundaryChain = csrBoundaryMat * csrChain
        out = [k for k,val in enumerate(csrBoundaryChain.data.tolist()) if val == 1]
        return out

    def boundaryCells2(cells,facets,faces):
        csrBoundaryMat = boundary2(cells,facets,faces)
        csrChain = csr_matrix(totalChain(cells))
        csrBoundaryChain = csrBoundaryMat * csrChain
        out = [k for k,val in enumerate(csrBoundaryChain.data.tolist()) if val == 1]
        return out
    ◇
```

Macro referenced in 6a.

2.2 Oriented operators

3 Exporting

```
"larlib/larlib/boundary.py" 6a ≡
    """ boundary operators """
    from larlib import *
    ⟨ convex-cells boundary operator 3 ⟩
    ⟨ path-connected-cells boundary operator 4 ⟩
    ⟨ From cells and facets to boundary cells 5 ⟩
    ◇
```

4 Testing

4.1 Non-oriented operators

Correct boundary extraction example The `boundary()` operator is applied here to a cellular 2-complex of convex cells, producing correct result. It is worth noting that the operator is dimension-independent, and must be appliad to the *pair* of compressed characteristic matrices M_d and M_{d-1}, that — in list format — we call either `CV,FV` or `FV,EV`, depending on the dimension (either 3 or 2) of the embedding space.

```
"test/py/boundary/test01.py" 6b ≡
```

```
""" 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 = boundary1(FV,EV,VV)

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()` operator, applied to a cellular 2-complex wih some non-convex cells, produces 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. In order to get always a correct result, use the `boundary1` operator.

"test/py/boundary/test02.py" 7 ≡
```
""" testing boundary operators (wrong result) """
from larlib import *

filename = "test/svg/inters/boundarytest3.svg" # KO (MKTRIANGLES) with boundarytest3 !!!
#filename = "test/svg/inters/boundarytest4.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]))))
VIEW(EXPLODE(1.2,1.2,1.2)(MKFACES((V,FV,EV))))
VIEW(SKEL_1(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,FV,EV)))))

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]))))
```
◇

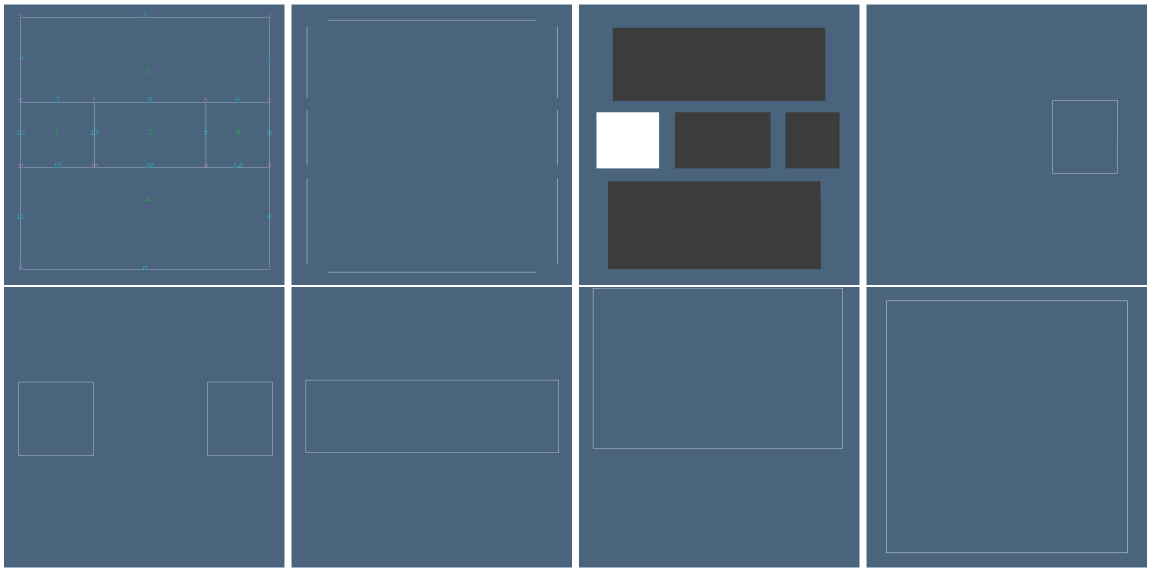

Figure 2: Convex-cell 2-complex. (a) Indexing of 0-,1-,and 2-cells; (b) exploded 2-boundary cells; (c) exploded 2-cells; (d) boundary of a singleton 2-chain; (e–h) boundaries of some 2-chains.

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 3d, 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]])
```

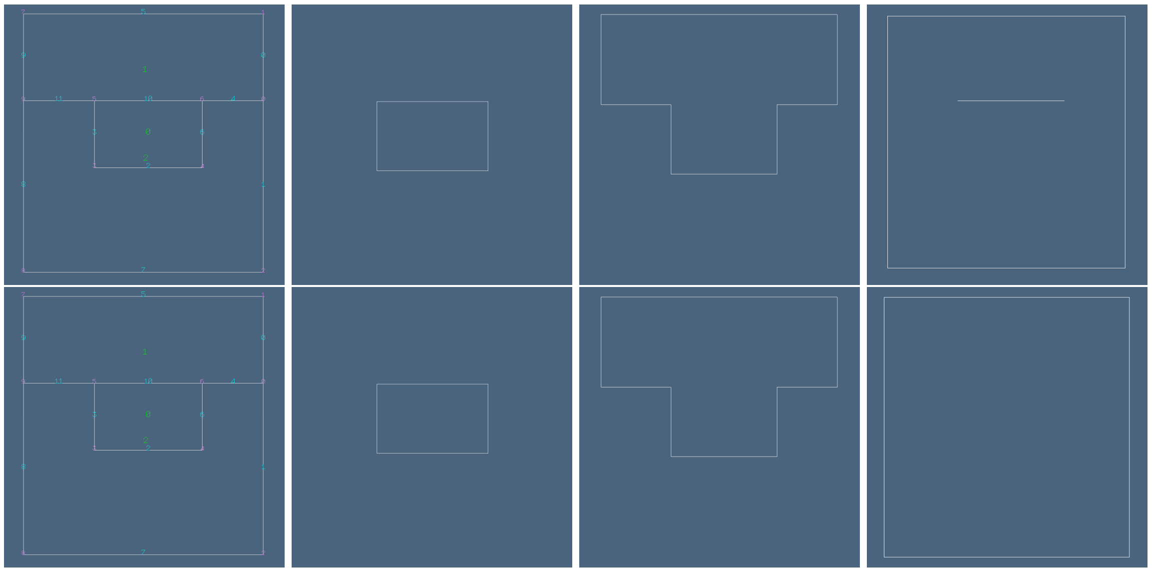

Figure 3: Non-working (i.e. *wrong*) example with `boundary`. (a) Indexing of 0-,1-,and 2-cells; (b) boundary of a singleton 2-chain; (c) exploded 2-cells; (d) boundary of a singleton 2-chain. Working (i.e. *exact*) example using `boundary1`: (e–h) as above.

3D non-convex LAR cells In this example and in the next one we show the boundary computation of LAR models with non-contractible 3- and 2-cells.

"test/py/boundary/test03.py" 9 ≡

```
""" 3D non-convex LAR cells """
from larlib import *

V = [[0.25,0.25,0.0],[0.25,0.75,0.0],[0.75,0.75,0.0],[0.75,0.25,0.0],[1.0, 0.0,0.0],
[0.0,0.0,0.0],[1.0,1.0,0.0],[0.0,1.0,0.0],[0.25,0.25,1.0],[0.25, 0.25,2.0],[0.25,0.75,
2.0],[0.25,0.75,1.0],[0.25,0.75,-1.0],[0.25,0.25, -1.0],[0.75,0.75,-1.0],[0.75,0.25,
-1.0],[0.75,0.25,1.0],[0.75,0.75,1.0], [1.0,0.0,1.0],[0.0,0.0,1.0],[1.0,1.0,1.0],
[0.0,1.0,1.0],[0.75,0.75,2.0],[0.75,0.25,2.0]]

CV = [(0,1,2,3,4,5,6,7,8,11,16,17,18,19,20,21), (0,1,2,3,8,11,16,17),
(0,1,2,3,12,13,14,15), (8,9,10,11,16,17,22,23)]

FV = [(2,3,16,17),(6,7,20,21),(12,13,14,15),(0,1,8,11),(1,2,11,17),(0,1,12,13),
(4,6,18,20),(5,7,19,21),(0,3,13,15),(0,3,8,16),(0,1,2,3),
(10,11,17,22),(2,3,14,15),(8,9,16,23),(8,11,16,17),
(1,2,12,14),(16,17,22,23),(4,5,18,19),(8,9,10,11),(
9,10,22,23),(0,1,2,3,4,5,6,7),(8, 11,16,17,18,19,20,21)]

EV =[(3,15),(7,21),(10,11),(4,18),(12,13),(5,19),(8,9),(18,19),(22,23),(0,3),(1,11),
(16,17),(0,8),(6,7),(20,21),(3,16),(10,22),(18,20),(19,21),(1,2),(12,14),(4,5),(
8,11),(13,15),(16,23),(14,15),(11,17),(17,22),(2,14),(2,17),(0,1),(9,10),(8,16),
(4,6),(1,12),(5,7),(0,13),( 9,23),(6,20),(2,3)]

VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV,CV],hpc,0.6))

BF = boundaryCells2(CV,FV,EV)
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,[FV[f] for f in BF],EV))))
◇
```

3D non-convex LAR cells In this example the 3D model is constructed partly in automated way, partly by hand. In particular, first we generate a structure of cuboidal complexes, then we transform it is a single complex using part of the computational pipeline being developed for the Boolean arrangments of complexes, so that all the included cells are mutually fragmented. Then the 3-cells are assembed as sets of 2-faces, giving the `CF` (cells-by-faces) variable. Finally this one is transformed automatically into `CV` (cells-by-vertices).

"test/py/boundary/test04.py" 10 ≡
```
""" 3D non-convex LAR cells """
from larlib import *

V,[VV,EV,FV,CV] = larCuboids([2,1,1],True)
mod1 = Struct([(V,FV,EV),t(.25,.25,0),s(.25,.5,2),(V,FV,EV)])
W,FW,EW = struct2lar(mod1)

quadArray = [[W[v] for v in face] for face in FW]
parts = boxBuckets3d(containmentBoxes(quadArray))
Z,FZ,EZ = spacePartition(W,FW,EW, parts)
```

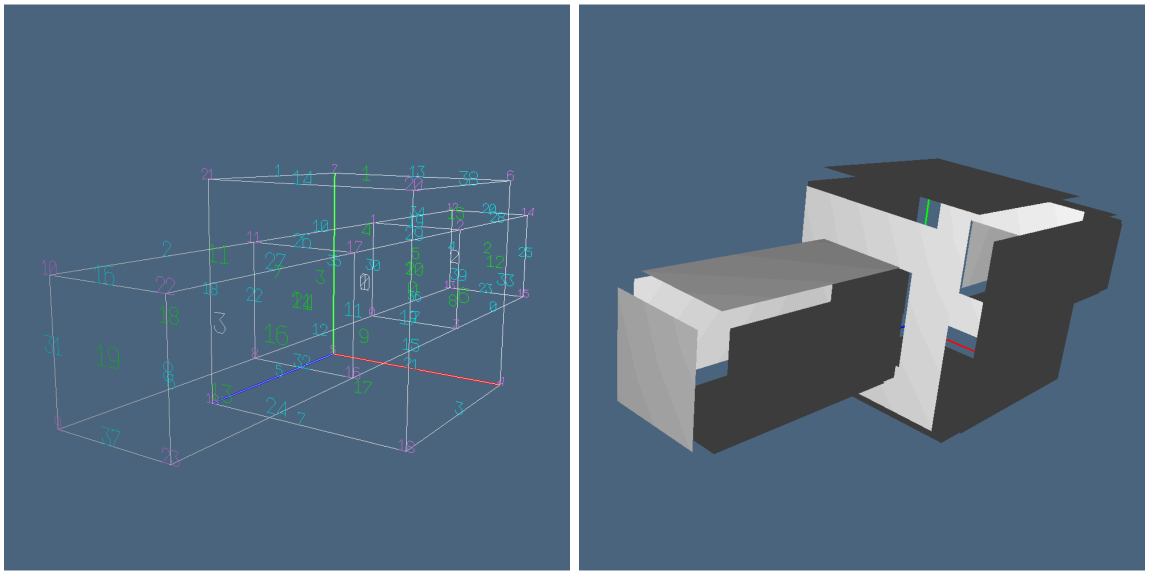

Figure 4: Non-convex 3-complex. (a) Indexing of 0-,1-,2- and 3-cells; (b) exploded 2-boundary cells. Notice that two faces are multiply-connected.

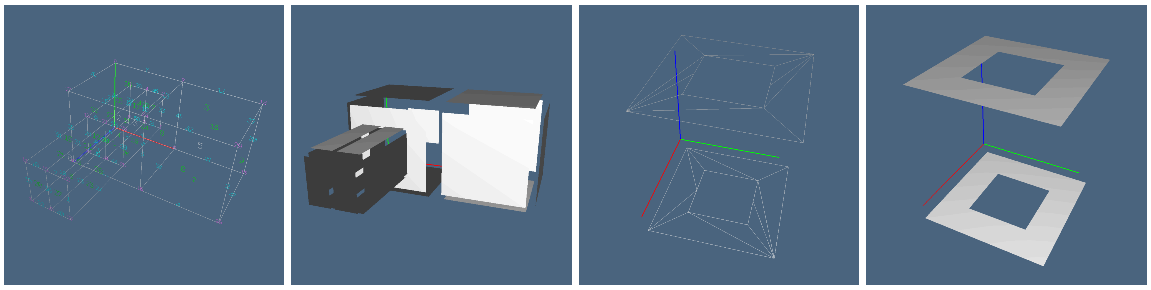

Figure 5: Non-convex 3-complex. (a) Indexing of 0-,1-,2- and 3-cells; (b) exploded 2-boundary cells —notice a drawing error on the back of the model—conversely, the data structures involved are correct, as shown by the two following pictures; (c) solid drawing of the 2-chain `[FV[29],FV[30]]`; (d) triangulation of the same 2-chain.

```
Z,FZ,EZ = larSimplify((Z,FZ,EZ),radius=0.0001)
V,FV,EV = Z,FZ,EZ

CF = AA(sorted)([[20,12,21,5,19,6],[27,1,5,28,13,23],[12,14,25,17,10,4],
[1,7,17,24,11,18],[30,29,26,16,8,22,10,11,4,18,24,25],[2,3,8,9,0,15]])

CV = [list(set(CAT([FV[f]  for f in faces]))) for faces in CF]

VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV,CV],hpc,0.6))

BF = boundaryCells2(CV,FV,EV)
VIEW(SKEL_1(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,[FV[f] for f in BF],EV)))))
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,[FV[f] for f in BF],EV))))

boundaryChain = chain2BoundaryChain(boundary1(FV,EV,VV))
faceChain = boundaryChain(29*[0]+[1]) + boundaryChain(30*[0]+[1])
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES(
    (V,[FV[29],FV[30]],[EV[e] for e in faceChain]))))
VIEW(SKEL_1(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES(
    (V,[FV[29],FV[30]],[EV[e] for e in faceChain])))))

cellBoundary = chain2BoundaryChain(boundary2(CV,FV,EV))([0,0,0,0,1,0])
faceChain = set()
for f in cellBoundary:
    faceCoords = len(FV)*[0]
    faceCoords[f] = 1
    faceChain = faceChain.union(boundaryChain(faceCoords))
VIEW(STRUCT(MKTRIANGLES(
    (V,[FV[f] for f in cellBoundary],[EV[e] for e in faceChain]))))
VIEW(SKEL_1(STRUCT(MKTRIANGLES(
    (V,[FV[f] for f in cellBoundary],[EV[e] for e in faceChain])))))
◇
```

4.2 Oriented operators

References

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