Boundary operators on LAR [*]

Alberto Paoluzzi

February 26, 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 26, 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 `boundary2`. 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 3a ⟩ ≡
```
""" convex-cells boundary operator --- best implementation """
def boundary(cells,facets):
    lenV = max(max(CAT(cells)),max(CAT(facets)))+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 11.

2.2 Non-convex LAR cells

A more general `boundary2` 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}.

Boundary operator from 3-chains to 2-chains

⟨ path-connected-cells boundary operator 3b ⟩ ≡
```
""" path-connected-cells boundary operator """
def boundary2(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)
        lenV = max(max(CAT(CV)),max(CAT(FV)),max(CAT(EV)))+1
        FE = larcc.crossRelation0(lenV,FV,EV)
        out = csrBoundaryFilter2(unreliable,out,csrBBMat,CV,FE)
    return out

def boundary3(CV,FV,EV):
    out = boundary2(CV,FV,EV)
    lenV = max(max(CAT(CV)),max(CAT(FV)),max(CAT(EV)))+1
    VV = AA(LIST)(range(lenV))
    csrBBMat = scipy.sparse.csc_matrix(boundary(FV,EV) * boundary2(CV,FV,EV))
    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)]
    if unreliable != []:
        FE = larcc.crossRelation0(lenV,FV,EV)
        out = csrBoundaryFilter3(unreliable,out,csrBBMat,CV,FE)
    return out
◇
```

Macro defined by 3b, 4.
Macro referenced in 11.

Boundary operator from 2-chains to 1-chains 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.

⟨ path-connected-cells boundary operator 4 ⟩ ≡

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

def csrBoundaryFilter2(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 csrBoundaryFilter3(unreliable,out,csrBBMat,cells,FE):
    for col in unreliable:
        cooCE = csrBBMat.T[col].tocoo()
        flawedCells = [cooCE.col[k] for k,datum in enumerate(cooCE.data)
                    if datum>2]
        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
◇
```

Macro defined by 3b, 4.
Macro referenced in 11.

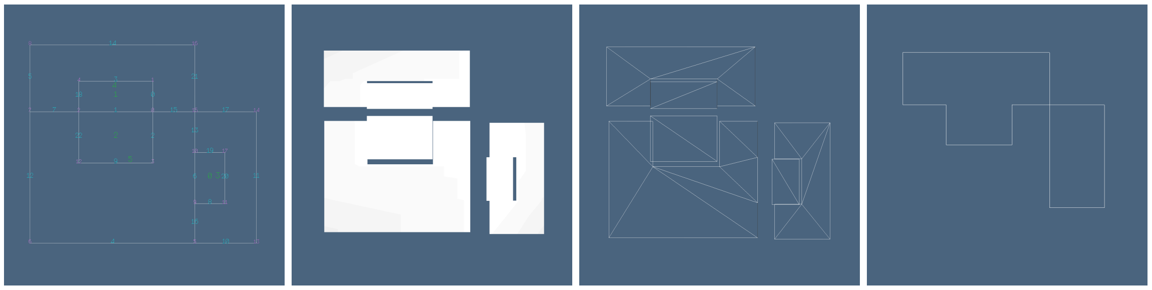

Figure 1: Non-convex LAR 2-complex with (two) 1-cells that are subsets of 2-cells without being their facets. Correctly disentangled by the `boundary2()` function: (a) Indexing of 0-, 1-, and 2-cells; (b) exploded 2-cells; (c) triangulated and exploded 2-cells; (d) boundary of the 2-chain `[1,1,1,1,1,0]`.

⟨ From cells and facets to boundary cells 6 ⟩ ≡
```
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 boundary2Cells(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

def boundary3Cells(cells,facets,faces):
    csrBoundaryMat = boundary3(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 11.

2.3 Oriented operators

2.3.1 Oriented simplicial complexes

2.3.2 Oriented LAR complexes

3 From relations to operators

The LAR approach to topology, implemented in the `LarLib` modules, allows the user to consider the topological relations of incidence and adjacency between faces of a cellular complex as *linear operators* between chains of cells of various dimensions.

The previous approach was to consider the incidence and adjaceny as set-theoretical relations, to be solved by using typical database tools. Conversely, according to the novel IT evolution towards big data and cloud-based storage, even for geometrical data, `LarLib` takes advantage of a conceptual framework based on linear-algebra and sparse matrices.

3.1 Classification of operators

In the standard solid modeling approach, mainly based on boundary representations, the standard topological operations concern the answers to queries, by reporting the subsets of boundary elements which are incident (different dimension) or adjacent (equal dimension) to assigned boundary elements. Boundary elements stand there for three type of boundary cells: aka *faces* F, *edges* E, and *vertices* V. Nine binary relations may be considered, that are summarized in Table 1.

Table 1: Binary topological relations between boundary elements in boundary representations of solid models

	F	E	V
F	FF	FE	FV
E	EF	EE	EV
V	VF	VE	VV

Conversely, LAR models are normally based on cellular 3-complexes, so using four sets of cells, namely *3-cells* C, *2-cells* F, *1-cells* E, and *0-cells* V. The resulting tables of topological relations, and the associated linear operators, are shown in Tables 2a and 2b, respectively.

Table 2: Binary topological relations between cells of LAR decompositions of solid models and corresponding topological operators on $\partial_\circ$ chains.

	C	F	E	V
C	CC	$\boxed{\text{CF}}$	$\boxed{\text{CE}}$	CV
F	$\boxed{\text{FC}}$	FF	$\boxed{\text{FE}}$	FV
E	$\boxed{\text{EC}}$	$\boxed{\text{EF}}$	EE	EV
V	VC	VF	VE	VV

	C	F	E	V
C	$\mathbf{1}_C^\top \circ \mathbf{1}_C$	∂_3	$\partial_2 \circ \partial_3$	$\mathbf{1}_C$
F	δ_2	$\mathbf{1}_F^\top \circ \mathbf{1}_F$	∂_2	$\mathbf{1}_F$
E	$\delta_2 \circ \delta_1$	δ_1	$\mathbf{1}_E^\top \circ \mathbf{1}_E$	$\mathbf{1}_E$
V	$\mathbf{1}_C^\top$	$\mathbf{1}_F^\top$	$\mathbf{1}_E^\top$	$\mathbf{1}_V$

3.2 Topological relations

3.2.1 Adjacency relations

$\langle$ kfaces-to-kfaces relation 7 $\rangle \equiv$

```
""" kfaces-to-kfaces relation """
eeOp = larEdges2Edges(EV,VV)
EE = [eeOp([k]) for k in range(len(EV))]
```

```
ffOp = larFaces2Faces(FV,EV)
FF = [ffOp([k]) for k in range(len(FV))]

ccOp = larCells2Cells(CV,FV,EV)
CC = [ccOp([k]) for k in range(len(CV))]

◇
```

Macro referenced in 8a.

Adjacency relations examples

"test/py/boundary/test09.py" 8a ≡

```
""" Adjacency relations examples """
from larlib import *

sys.path.insert(0, 'test/py/boundary/')
from test07 import *

⟨ kfaces-to-kfaces relation 7 ⟩
print "\nCC =",CC
print "\nFF =",FF
print "\nEE =",EE,"\n"

V,BF,BE = larBoundary3(V,CV,FV,EV)([1,0])
VIEW(STRUCT(MKTRIANGLES((V,[FV[h] for h in FF[-1]],EV),color=True)))
VIEW(STRUCT(MKPOLS((V,[EV[h] for h in EE[-1]]))+[COLOR(RED)(MKPOLS((V,[EV[-1]]))[0])]))
◇
```

3.2.2 Incidence relations

⟨ mfaces-to-nfaces relations 8b ⟩ ≡

```
""" mfaces-to-nfaces relations """
fcOp = larCells2Faces(CV,FV,EV)
CF = [fcOp([k]) for k in range(len(CV))]
FC = invertRelation(CF)

ecOp = larCells2Edges(CV,FV,EV)
CE = [ecOp([k]) for k in range(len(CV))]
EC = invertRelation(CE)

efOp = larFaces2Edges(FV,EV)
FE = [efOp([k]) for k in range(len(FV))]
EF = invertRelation(FE)
◇
```

Macro referenced in 9a.

Incidence relations examples

"test/py/boundary/test10.py" 9a ≡

```
    """ Incidence relations examples """
    from larlib import *

    sys.path.insert(0, 'test/py/boundary/')
    from test08 import *

    ⟨ mfaces-to-nfaces relations 8b ⟩
    print "\nFC =",FC
    print "\nEC =",EC
    print "\nEF =",EF,"\n"


    ◇
```

3.3 Querying

The more important topological operations in a geometric system concern the answer to queries of the type: "what is the h-chain whose cells are $(k-h)$-incident to a given h-chain"?

An efficient answer is given by the three higher-level functions in this section, respectively denoted as `larCells2Faces`, `larCells2Edges`, and `larFaces2Edges`. Their first application, over the two or three necessary (compressed) characteristic matrices, returns the `csr_matrix` of the topological operator, that can be so cached by the calling code. The second application, over the list of h-chain indices, returns the list of k-chain indices of the cells that share with them a $(k-h)$-face.

3.3.1 Topological incidences

Query from 3-chain to incident 2-chain

⟨ Query from 3-chain to incident 2-chain 9b ⟩ ≡

```
    """ Query from 3-chain to incident 2-chain """
    def larCells2Faces(CV,FV,EV):
        csrFC = boundary3(CV,FV,EV)
        def larCells2Faces0(chain):
            chainCoords = csc_matrix((csrFC.shape[1],1),dtype='b')
            for k in chain: chainCoords[k,0] = 1
            out = csrFC * chainCoords
            return out.tocoo().row.tolist()
        return larCells2Faces0
    ◇
```

Macro referenced in 11.

Query from 3-chain to incident 1-chain

⟨ Query from 3-chain to incident 1-chain 10a ⟩ ≡

```
""" Query from 3-chain to incident 1-chain """
def larCells2Edges(CV,FV,EV):
    lenV = max(CAT(CV))+1
    VV = AA(LIST)(range(lenV))
    csrEC = boundary2(FV,EV,VV) * boundary3(CV,FV,EV)
    def larCells2Faces0(chain):
        chainCoords = csc_matrix((csrEC.shape[1],1),dtype='b')
        for k in chain: chainCoords[k,0] = 1
        out = csrEC * chainCoords
        return out.tocoo().row.tolist()
    return larCells2Faces0
◇
```

Macro referenced in 11.

Query from 2-chain to incident 1-chain

⟨ Query from 2-chain to incident 1-chain 10b ⟩ ≡

```
""" Query from 2-chain to incident 1-chain """
def larFaces2Edges(FV,EV):
    lenV = max(CAT(FV)) + 1
    VV = AA(LIST)(range(lenV))
    csrEF = boundary2(FV,EV,VV)
    def larCells2Faces0(chain):
        chainCoords = csc_matrix((csrEF.shape[1],1),dtype='b')
        for k in chain: chainCoords[k,0] = 1
        out = csrEF * chainCoords
        return out.tocoo().row.tolist()
    return larCells2Faces0
◇
```

Macro referenced in 11.

3.3.2 Topological adjacencies

kfaces-to-kfaces relations

⟨ kfaces-to-kfaces relations 10c ⟩ ≡

```
""" kfaces-to-kfaces relations """

def larCells2Cells(CV,FV,EV):
    csrMat = boundary3(CV,FV,EV)
    csrCC = csrMat.T * csrMat
    def larCells2Cells0(chain):
        chainCoords = csc_matrix((csrCC.shape[1],1),dtype='b')
```

```
            for k in chain: chainCoords[k,0] = 1
            out = csrCC * chainCoords
            return out.tocoo().row.tolist()
        return larCells2Cells0

    def larFaces2Faces(FV,EV):
        lenV = max(CAT(FV)) + 1
        VV = AA(LIST)(range(lenV))
        csrMat = boundary2(FV,EV,VV)
        csrFF = csrMat.T * csrMat
        def larFaces2Faces0(chain):
            chainCoords = csc_matrix((csrFF.shape[1],1),dtype='b')
            for k in chain: chainCoords[k,0] = 1
            out = csrFF * chainCoords
            return out.tocoo().row.tolist()
        return larFaces2Faces0

    def larEdges2Edges(EV,VV):
        lenV = len(VV)
        csrMat = boundary(EV,VV)
        csrEE = csrMat.T * csrMat
        def larFaces2Faces0(chain):
            chainCoords = csc_matrix((csrEE.shape[1],1),dtype='b')
            for k in chain: chainCoords[k,0] = 1
            out = csrEE * chainCoords
            return out.tocoo().row.tolist()
        return larFaces2Faces0
    ◇
```

Macro referenced in 11.

3.4 Examples

4 Exporting

"larlib/larlib/boundary.py" 11 ≡

```
    """ boundary operators """
    from larlib import *
    ⟨convex-cells boundary operator 3a⟩
    ⟨path-connected-cells boundary operator 3b, ... ⟩
    ⟨From cells and facets to boundary cells 6⟩
    ⟨Marshalling a structure to a LAR cellular model 20⟩
    ⟨Boundary of a 3-complex 21⟩
    ⟨Query from 3-chain to incident 2-chain 9b⟩
    ⟨Query from 3-chain to incident 1-chain 10a⟩
    ⟨Query from 2-chain to incident 1-chain 10b⟩
```

⟨ kfaces-to-kfaces relations 10c ⟩
◇

5 Testing

5.1 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" 12a ≡

```
""" 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 = boundary2(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 `boundary2` operator.

"test/py/boundary/test02.py" 12b ≡

```
""" 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 = boundary2(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)(MKTRIANGLES((V,FV,EV),color=True)))
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]))))
◇
```

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.

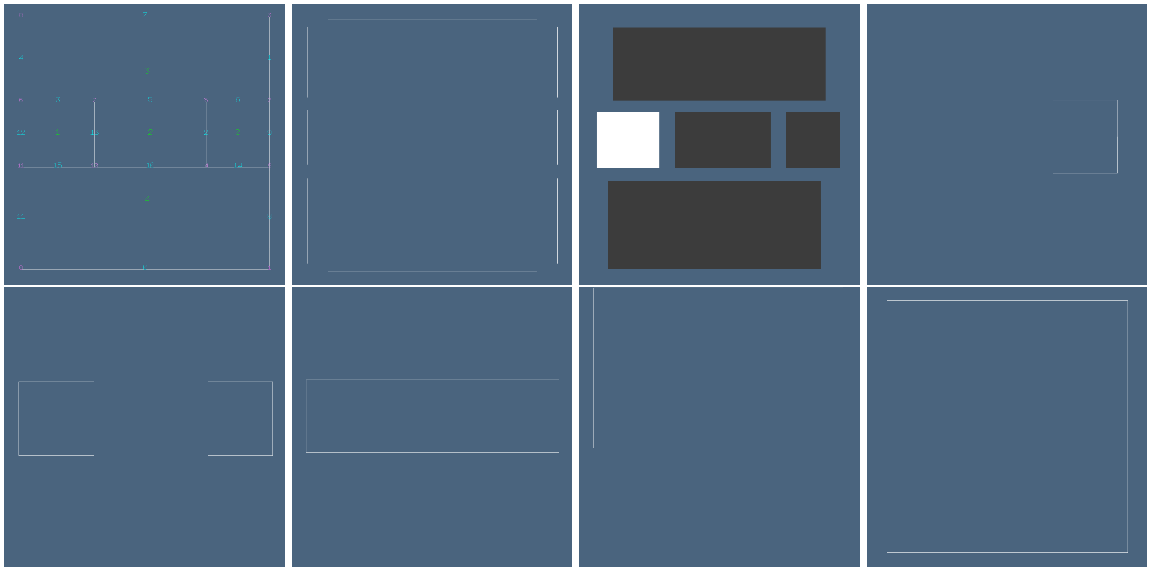

Figure 2: Convex-cell 2-complex. (a) Indexing of 0-,1-,and 2-cells; (b) exploded 2-boundary cells; (c) exploded 2-cells; (d) boundary of a singleton 2-chain; (e–h) boundaries of some 2-chains.

```
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]: boundary2(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]])
```

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" 14 ≡

```
""" 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],
```

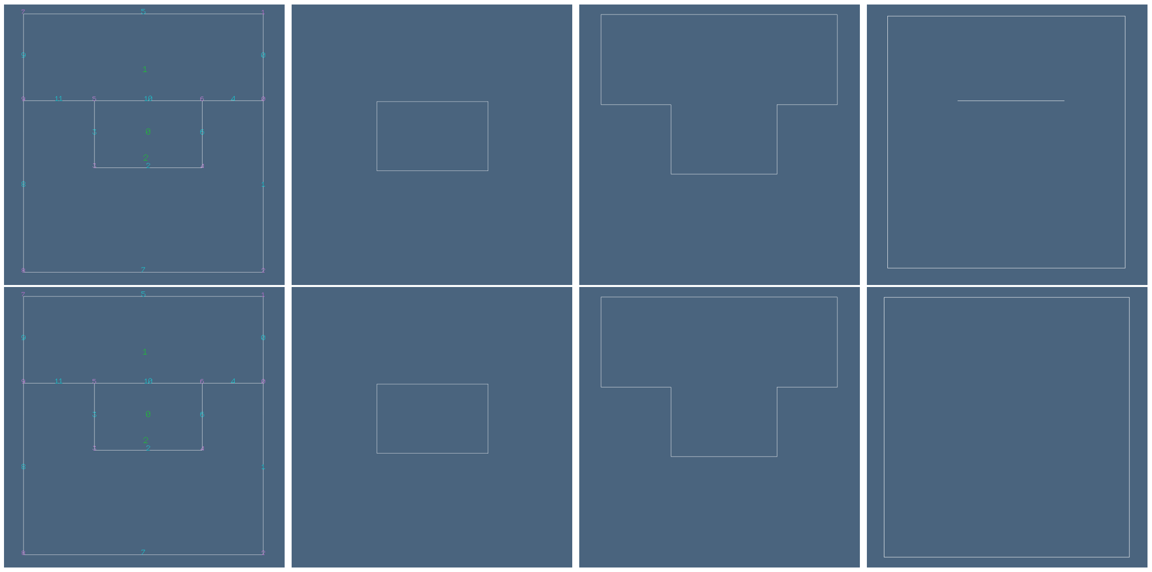

Figure 3: Non-working (i.e. *wrong*) example with `boundary`. (a) Indexing of 0-,1-,and 2-cells; (b) boundary of a singleton 2-chain; (c) exploded 2-cells; (d) boundary of a singleton 2-chain. Working (i.e. *exact*) example using `boundary2`: (e–h) as above.

```
[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 = boundary3Cells(CV,FV,EV)
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,[FV[f] for f in BF],EV),color=True)))
```

◇

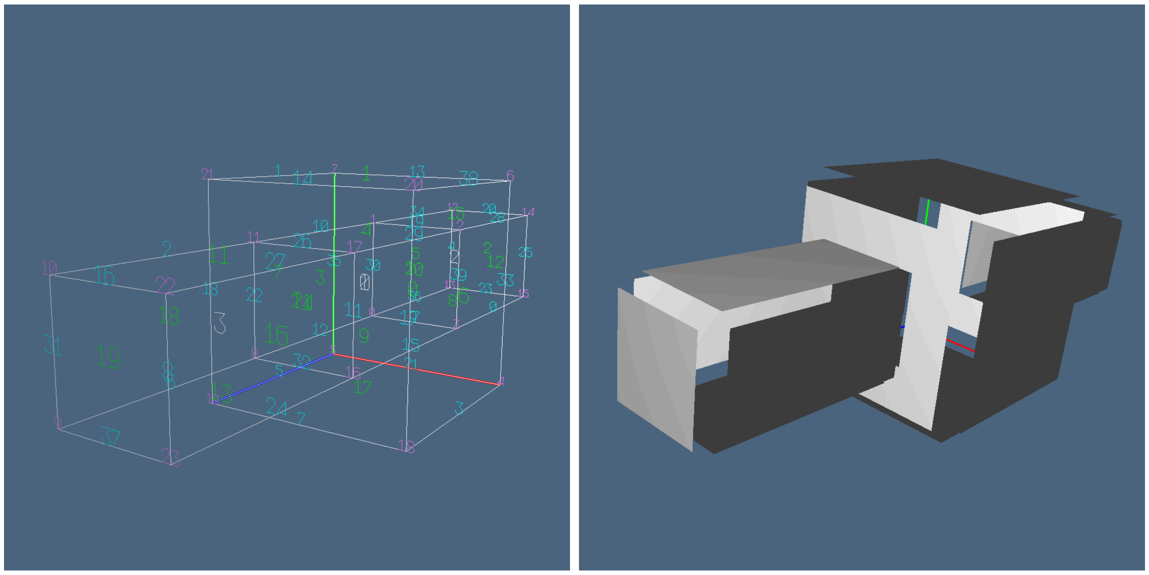

Figure 4: Non-convex 3-complex. (a) Indexing of 0-,1-,2- and 3-cells; (b) exploded 2-boundary cells. Notice that two faces are multiply-connected.

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"` 16a ≡

```
""" 3D non-convex LAR cells """
from larlib import *
```
⟨Input of a cellular 3-complex 16b⟩
⟨Visualization of a 2-chain of a 3-complex 17⟩
⟨Visualization of a 3-chain of a 3-complex 18a⟩
◇

Input of a cellular 3-complex

⟨Input of a cellular 3-complex 16b⟩ ≡

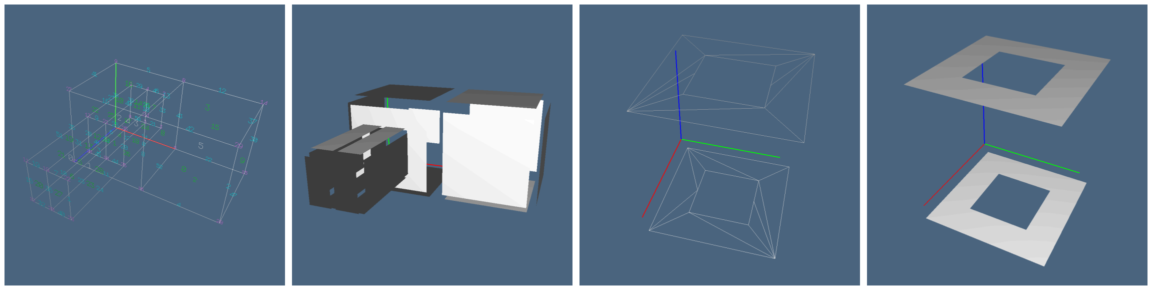
Figure 5: Non-convex 3-complex. (a) Indexing of 0-,1-,2- and 3-cells; (b) exploded 2-boundary cells —notice a drawing error on the back of the model—conversely, the data structures involved are correct, as shown by the two following pictures; (c) solid drawing of the 2-chain [FV[29],FV[30]]; (d) triangulation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

```
""" Input of a cellular 3-complex """
V,[VV,EV,FV,CV] = larCuboids([2,1,1],True)
struct = Struct([(V,FV,EV),t(.25,.25,0),s(.25,.5,2),(V,FV,EV)])

V,FV,EV = struct2Marshal(struct)
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))
◇
```

Macro referenced in 16a.

Visualization of a 2-chain of a 3-complex

⟨Visualization of a 2-chain of a 3-complex 17⟩ ≡

```
""" Visualization of the boundary 2-chain of a 3-complex """

V,BF,BE = larBoundary3(V,CV,FV,EV)(len(CV)*[1])
VIEW(STRUCT(MKTRIANGLES((V,BF,EV),color=True)))
VIEW(SKEL_1(STRUCT(MKTRIANGLES((V,BF,EV)) )))

boundaryEdges = chain2BoundaryChain(boundary2(FV,EV,VV))
edgeChain = boundaryEdges(29*[0]+[1]+[1])
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,FV[29:31],[EV[e] for e in edgeChain]),color=True)))
VIEW(SKEL_1(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,FV[29:31],[EV[e] for e in edgeChain])))))
◇
```

Macro referenced in 16a.

Visualization of a 3-chain of a 3-complex

⟨ Visualization of a 3-chain of a 3-complex 18a ⟩ ≡

```
""" Visualization of a 3-chain of a 3-complex """

V,BF,BE = larBoundary3(V,CV,FV,EV)([0,0,0,0,1,1])
VIEW(STRUCT(MKTRIANGLES((V,BF,BE))))
VIEW(SKEL_1(STRUCT(MKTRIANGLES((V,BF,BE)) )))
◇
```

Macro referenced in 16a.

"test/py/boundary/test05.py" 18b ≡

```
""" Boundary of a 3-complex """
from larlib import *

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True)
cube = Struct([ (V,FV,EV) ])
assembly = Struct([ cube, Struct([t(0,.5,0), r(PI/4,0,0), s(.5,.5,.5),cube]) ])

V,FV,EV = struct2Marshal(assembly)
VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],hpc,0.6))

CF = [[1,2,3,4,6,7],[0,1,2,3,4,5,6,7,8,9,10,11]]
CV = [list(set(CAT([FV[f]  for f in faces]))) for faces in CF]

V,BF,BE = larBoundary3(V,CV,FV,EV)([0,1])
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,BF,BE),color=True)))
◇
```

"test/py/boundary/test06.py" 18c ≡

```
""" Boundary of a 3-complex """
from larlib import *

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True)
cube = Struct([ (V,FV,EV) ])
hole = Struct([t(0,.5,0), r(PI/4,0,0), s(.5,.5,.5),cube])
assembly = Struct([ cube, hole, t(0,0,SQRT(0.5)), hole ])

V,FV,EV = struct2Marshal(assembly) # WRONG:  TODO: check ...
VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[[],[],FV],hpc,0.6))

CF = [[4,5,7,16,17,19,20],[3,8,6,12,11,13],[0,1,10,20],[]]
```

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

V,BF,BE = larBoundary3(V,CV,FV,EV)([0,1,1,0])
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,BF,BE)))) # ERROR in MKTRIANGLES with non-manifold fa
VIEW(EXPLODE(1.2,1.2,1.2)(MKFACES((V,BF,EV))))
VIEW(SKEL_1(EXPLODE(1.2,1.2,1.2)(MKFACES((V,BF,EV)))))
◇
```

"test/py/boundary/test07.py" 19a ≡

```
""" Boundary of a 3-complex """
from larlib import *

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True)
cube = Struct([ (V,FV,EV) ])
hole = Struct([t(0,.5,0), r(PI/4,0,0), s(1,.5/SQRT(2),.5/SQRT(2)),cube])
assembly = Struct([ cube, hole ])

V,FV,EV = struct2Marshal(assembly) # WRONG:  TODO: check ...
VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],hpc,0.6))

CF = [[1,3,6,7,12,11],[0,2,4,5,9,8]]
CV = [list(set(CAT([FV[f]  for f in faces]))) for faces in CF]

V,BF,BE = larBoundary3(V,CV,FV,EV)([1,0])
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,BF,EV),color=True)))
◇
```

"test/py/boundary/test08.py" 19b ≡

```
""" Boundary of a 3-complex """
from larlib import *

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True)
cube = Struct([ (V,FV,EV) ])
hole = Struct([t(0,.5,0), r(PI/4,0,0), s(1,.5/SQRT(2),.5/SQRT(2)),cube])
assembly = Struct([ cube, hole ])
assembly2 = Struct([ assembly, t(0,0,.5), s(0.5,1,1), hole ])

V,FV,EV = struct2Marshal(assembly2) # WRONG:  TODO: check ...
VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[[],[],FV],hpc,0.7))

CF = [[1,3,6,14,17,18,19, 0,4,9,11,15,16, 2,5,7,8,10,13],[0,4,9,11,15,16],[2,5,7,8,10,13]]
CV = [list(set(CAT([FV[f]  for f in faces]))) for faces in CF]
```

```
V,BF,BE = larBoundary3(V,CV,FV,EV)([1,0,0])
VIEW(STRUCT(MKTRIANGLES((V,BF,BE),color=True)))
VIEW(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,BF,BE),color=True)))
VIEW(SKEL_1(EXPLODE(1.2,1.2,1.2)(MKTRIANGLES((V,BF,BE)))))
◇
```

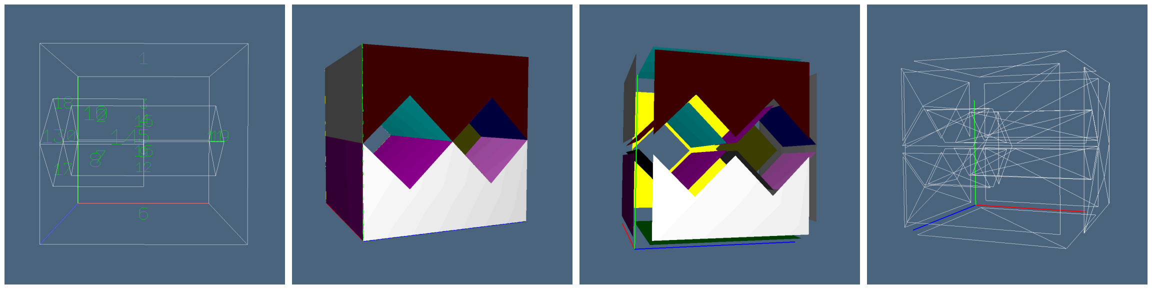

Figure 6: Decomposition of the unit 3-cube in a cellular 3-complex with three 3-cells. Two 3-cells are homeomorphic to the 3-ball, while the remaining one is homeomorphic to the 3-torus. Notice that one of 2-cells (as well one of 3-cells) are non-contractible and non-manifold, as well as non-convex: (a) Indexing of 2-faces of the 3-complex; (b) drawing of the (boundary of) non-convex 3-cell; (c) exploded drawing of the (boundary of) non-convex 3-cell; (d) triangulation of its 2-faces. The triangulation of LAR 2-faces is needed in order to draw them solidly.

5.2 Oriented operators

A Utilities

Marshalling a structure to a LAR cellular model The function `struct2Marshal` transforms a `Struct` object, often used to define some assembly of simpler models, to a correctly defined LAR cellular model, i.e. to a cellular partition of the space, in other words a quasi-disjoint partition of the object into well-glued cells of suitable dimensions.

⟨Marshalling a structure to a LAR cellular model 20⟩ ≡

```
""" Marshalling a structure to a LAR cellular model """
import boolean,inters

def struct2Marshal(struct):
    W,FW,EW = struct2lar(struct)
    quadArray = [[W[v] for v in face] for face in FW]
    parts = boolean.boxBuckets3d(boolean.containmentBoxes(quadArray))
    Z,FZ,EZ = boolean.spacePartition(W,FW,EW, parts)
    V,FV,EV = inters.larSimplify((Z,FZ,EZ),radius=0.0001)
```

```
        return V,FV,EV
    ◇
```

Macro referenced in 11.

Boundary of a 3-complex

⟨ Boundary of a 3-complex 21 ⟩ ≡

```
    """ Boundary of a 3-complex """
    import larcc
    """  WHY wrong ????  TOCHECK !!
    def larBoundary3(V,CV,FV,EV):
        VV = AA(LIST)(range(len(V)))
        operator3 = larcc.chain2BoundaryChain(boundary3(CV,FV,EV))
        operator2 = larcc.chain2BoundaryChain(boundary2(FV,EV,VV))
        def larBoundary30(chain):
            BF = operator3(chain)
            faceCoords = len(FV)*[0]
            for f in BF: faceCoords[f] = 1
            BE = operator2(faceCoords)
            return V,[FV[f] for f in BF],[EV[e] for e in BE]
        return larBoundary30
    """
    def larBoundary3(V,CV,FV,EV):
        VV = AA(LIST)(range(len(V)))
        operator3 = larcc.chain2BoundaryChain(boundary3(CV,FV,EV))
        operator2 = larcc.chain2BoundaryChain(boundary2(FV,EV,VV))
        def larBoundary30(chain):
            BF = operator3(chain)
            BE = set()
            for f in BF:
                faceCoords = len(FV)*[0]
                faceCoords[f] = 1
                BE = BE.union(operator2(faceCoords))
            return V,[FV[f] for f in BF],[EV[e] for e in BE]
        return larBoundary30
    ◇
```

Macro referenced in 11.

References

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