Boolean combinations of cellular complexes as chain operations *

Alberto Paoluzzi

September 17, 2014

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. September 17, 2014

1 Introduction

In this module a novel approach to Boolean operations of cellular complexes is defined and implemented. The novel algorithm may be summarised as follows.

First we compute the CDC (Common Delaunay Complex) of the input LAR complexes A and B, to get a LAR of the *simplicial* CDC.

Then, we split the cells intersecting the boundary faces of the input complexes, getting the final *polytopal* SCDC (Split Common Delaunay Complex), whose cells provide the basis for the linear coordinate representation of both input complexes, upon the same space decomposition.

Afterwards, every Boolean result is computed by bitwise operations, between the coordinate representations of the transformed A and B input.

Finally a greedy assembly of SCDC cells is executed, in order to return a polytopal complex with a reduced number of cells.

1.1 Preview of the Boolean algorithm

The goal is the computation of $A \diamond B$, with $\diamond \in \{\cup, \cap, -\}$, where a LAR representation of both A and B is given. The Boolean algorithm works as follows.

1. Embed both cellular complexes A and B in the same space (say, identify their common vertices) by $V_{ab} = V_a \cup V_b$.

2. Build their CDC (Common Delaunay Complex) as the LAR of *Delaunay triangulation* of the vertex set V_{ab}, and embedded ∂A and ∂B in it.

3. Split the (highest-dimensional) cells of CDC crossed by ∂A or ∂B. Their lower dimensional faces remain partitioned accordingly. We name the resulting complex SCDC (Split Common Delaunay Complex).

4. With respect to the SCDC basis of d-cells C_d, compute two coordinate chains $\alpha, \beta : C_d \to \{0, 1\}$, such that:

$$\alpha(cell) = 1 \quad \text{if } |cell| \subset A; \quad \text{else } \alpha(cell) = 0,$$
$$\beta(cell) = 1 \quad \text{if } |cell| \subset B; \quad \text{else } \beta(cell) = 0.$$

5. Extract accordingly the SCDC chain corresponding to $A \diamond B$, with $\diamond \in \{\cup, \cap, -\}$.

1.2 Remarks

You may make an analogy between the SCDC (*Split* CDC) and a CDT (Constrained Delaunay Triangulation). In part they coincide, but in general, the SCDC is a polytopal complex, and is not a simplicial complex as the CDC.

The more complex algorithmic step is the cell splitting. Every time, a single d-cell c is split by a single hyperplane (cutting its interior) giving either two splitted cells c_1 and c_2, or just one output cell (if the hyperplane is the affine hull of the CDC facet) whatever the input cell dimension d. After every splitting of the cell interior, the row c is substituted (within the `CV` matrix) by c_1, and c_2 is added to the end of the `CV` matrix, as a new row.

The splitting process is started by "splitting seeds" generated by $(d-1)$-faces of both operand boundaries. In fact, every such face, say f, has vertices on CDC and *may* split some incident CDC d-cell. In particular, starting from its vertices, f must split the CDC cells in whose interior it passes though.

So, a dynamic data structure is set-up, storing for each boundary face f the list of cells it must cut, and, for every CDC d-cell with interior traversed by some such f, the list of cutting faces. This data structure is continuously updated during the splitting process, using the adjacent cells of the split ones, who are to be split in turn. Every split cell may add some adjacent cell to be split, and after the split, the used pair (`cell,face`) is removed. The splitting process continues until the data structure becomes empty.

Every time a cell is split, it is characterized as either internal (1) or external (0) to the used (oriented) boundary facet f, so that the two resulting subcells c_1 and c_2 receive two opposite characterization (with respect to the considered boundary).

At the very end, every (polytopal) SCDC d-cell has two bits of information (one for argument A and one for argument B), telling whether it is internal (1) or external (0) or unknown (-1) with respect to every Boolean argument.

A final recursive traversal of the SCDC, based on cell adjacencies, transforms every -1 into either 0 or 1, providing the two final chains to be bitwise operated, depending on the Boolean operation to execute.

2 Step 1: merging discrete spaces

2.1 Requirements

The *join* of two sets $P, Q \subset \mathbb{E}^d$ is the set $PQ = \{\alpha\mathbf{x} + \beta\mathbf{y} \,|\mathbf{x} \in P,\ \mathbf{y} \in Q\}$, where $\alpha, \beta \in \mathbb{R}$, $\alpha, \beta \geq 0$, and $\alpha + \beta = 1$. The join operation is associative and commutative.

Input Two LAR models of two non-empty "solid" d-spaces A and B, denoted as `(V1,CV1)` and `(V2,CV2)`.

Output The LAR representation `(V,CV)` of Delaunay triangulation (simplicial d-complex) of the set $\operatorname{conv} AB \subset \mathbb{E}^d$, convex hull of the join of A and B, named Common Delaunay Complex (CDC) in the following.

Auxiliary data structures This software module returns also:

1. a dictionary `vertDict` of `V` vertices, with *key* the symbolic representation of vertices `v` returned by expressions `vcode(v)`, $\texttt{v} \in \texttt{V}$, and with values the finite ordinal numbers of the vertices;

2. the numbers `n1`, `n12`, `n2` of the elements of `V1`, $\texttt{V1} \cap \texttt{V2}$, and `V2`, respectively. Notice that the following assertions must hold (see Figure 1):

$$\texttt{n1} - \texttt{n12} + \texttt{n2} = \texttt{n} \tag{1}$$

$$0 < \texttt{n} - \texttt{n2} \leq \texttt{n1} < \texttt{n} \tag{2}$$

3. the input boundary complex `(V,BC)`, with $\texttt{BC} = \texttt{BC1+BC2}$, i.e. the union of the two boundary $(d-1)$-complexes `(V,BC1)` and `(V,BC2)`, defined on the common vertices.

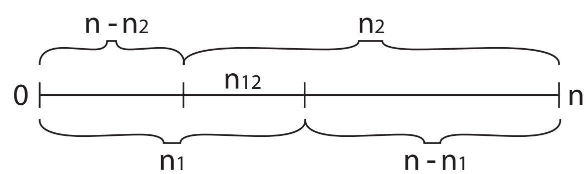

Figure 1: Relationships inside the orderings of CDC vertices

2.2 Implementation

⟨ Compute model boundaries of complex of convex cells 4a ⟩ ≡

```
""" Compute model boundaries of complex of convex cells """

def larFacetsOfPolytopalComplex(vertDict,cells,facets):
   (V1,CV1),(V2,CV2) = model1,model2
   for cell in CV1:
      Vcell = [V1[v] for v in cell]

```

◇

Macro never referenced.

⟨ Merge two dictionaries with keys the point locations 4b ⟩ ≡

```
""" Merge two dictionaries with keys the point locations """
def mergeVertices(model1, model2):

   (V1,CV1),(V2,CV2) = model1, model2

   n = len(V1); m = len(V2)
   def shift(CV, n):
```

```
      return [[v+n for v in cell] for cell in CV]
   CV2 = shift(CV2,n)

   vdict1 = defaultdict(list)
   for k,v in enumerate(V1): vdict1[vcode(v)].append(k)
   vdict2 = defaultdict(list)
   for k,v in enumerate(V2): vdict2[vcode(v)].append(k+n)
   vertDict = defaultdict(list)
   for point in vdict1.keys(): vertDict[point] += vdict1[point]
   for point in vdict2.keys(): vertDict[point] += vdict2[point]

   case1, case12, case2 = [],[],[]
   for item in vertDict.items():
      key,val = item
      if len(val)==2:  case12 += [item]
      elif val[0] < n: case1 += [item]
      else: case2 += [item]
   n1 = len(case1); n2 = len(case12); n3 = len(case2)

   invertedindex = list(0 for k in range(n+m))
   for k,item in enumerate(case1):
      invertedindex[item[1][0]] = k
   for k,item in enumerate(case12):
      invertedindex[item[1][0]] = k+n1
      invertedindex[item[1][1]] = k+n1
   for k,item in enumerate(case2):
      invertedindex[item[1][0]] = k+n1+n2

   V = [eval(p[0]) for p in case1] + [eval(p[0]) for p in case12] + [eval(
            p[0]) for p in case2]
   CV1 = [sorted([invertedindex[v] for v in cell]) for cell in CV1]
   CV2 = [sorted([invertedindex[v] for v in cell]) for cell in CV2]

   return V,CV1,CV2, n1+n2,n2,n2+n3
◇
```

Macro referenced in 15a.

⟨ Make Common Delaunay Complex 5 ⟩ ≡

```
""" Make Common Delaunay Complex """
def makeCDC(arg1,arg2):

   (V1,basis1), (V2,basis2) = arg1,arg2
   (facets1,cells1),(facets2,cells2) = basis1[-2:],basis2[-2:]
   model1, model2 = (V1,cells1),(V2,cells2)

   V, _,_, n1,n12,n2 = mergeVertices(model1, model2)
```

```
    n = len(V)
    assert n == n1 - n12 + n2

    CV = sorted(AA(sorted)(Delaunay(array(V)).simplices))

    vertDict = defaultdict(list)
    for k,v in enumerate(V): vertDict[vcode(v)] += [k]

    BC1 = signedCellularBoundaryCells(V1,basis1)
    BC2 = signedCellularBoundaryCells(V2,basis2)
    BC = [[ vertDict[vcode(V1[v])][0] for v in cell] for cell in BC1] + [
          [ vertDict[vcode(V2[v])][0] for v in cell] for cell in BC2] #+ qhullBoundary(V)

    return V,CV,vertDict,n1,n12,n2,BC
◇
```

Macro referenced in 15a.

3 Step 2: splitting cells

The goal of this section is to transform the CDC simplicial complex, into the polytopal Split Common Delaunay Complex (SCDC), by splitting the d-cells of CDC crossed in their interior by some cell of the input boundary complex.

3.1 Requirements

We call here for a sequential implementation, following every $(d-1)$-facet `lambda` in `BC` (for *Boundary Cells*). We start the splitting with `COVECTOR(lambda)` from `cell`, one of the CDC d-cells incident on a vertex of `lambda`, and continue the splitting on the d-cells $(d-1)$-adjacent to `cell`, where (a) `COVECTOR(lambda)` either crosses the `cell`'s interior or contains one of `cell`'s $(d-1)$-facets and (b) such that the intersection with `lambda` is not empty, until the queue (or stack) of d-cells to intersect with `covector` is not empty.

Best computational strategy First associate to each cutting facet the list of cells it may cut; then execute all the cuts. In this way we can compute the adjacency matrix just one time at the beginning of the procedure, and do not need to update it after every split.

Input The output of previous algorithm stage.

Output The LAR representation `(W,PW)` of the SCDC,

Auxiliary data structures This software module returns also a dictionary `splitFacets`, with keys the input boundary faces and values the list of pairs `(covector,fragmentedFaces)`.

3.2 Implementation

Computing the adjacent cells of a given cell To perform this task we make only use of the `CV` list. In a more efficient implementation we should make direct use of the sparse adjacency matrix, to be dynamically updated together with the `CV` list. The computation of the adjacent d-cells of a single d-cell is given here by extracting a column of the `CSR`$(M_d\, M_d^t)$. This can be done by multiplying `CSR`(M_d) by its transposed row corresponding to the query d-cell.

⟨ Computing the adjacent cells of a given cell 6 ⟩ ≡

```
""" Computing the adjacent cells of a given cell """
def adjacencyQuery (V,CV):
   dim = len(V[0])
   def adjacencyQuery0 (cell):
      nverts = len(CV[cell])
      csrCV =  csrCreate(CV)
      csrAdj = matrixProduct(csrCV,csrTranspose(csrCV))
      cellAdjacencies = csrAdj.indices[csrAdj.indptr[cell]:csrAdj.indptr[cell+1]]
      return [acell for acell in cellAdjacencies if dim <= csrAdj[cell,acell] < nverts]
   return adjacencyQuery0
◇
```

Macro referenced in 15a.

Relational inversion (characteristic matrix transposition) The operation could be executed by simple matrix transposition of the CSR (Compressed Sparse Row) representation of the sparse characteristic matrix $M_d \equiv$ `CV`. A simple relational inversion using Python lists is given here. The `invertRelation` function is given here, linear in the size of the `CV` list, where the complexity of each cell is constant and small in most cases.

⟨ Characteristic matrix transposition 7a ⟩ ≡

```
""" Characteristic matrix transposition """
def invertRelation(CV):
   def myMax(List):
      if List==[]: return -1
      else: return max(List)
   columnNumber = max(AA(myMax)(CV))+1
   VC = [[] for k in range(columnNumber)]
   for k,cell in enumerate(CV):
      for v in cell:
         VC[v] += [k]
   return VC
◇
```

Macro referenced in 15a.

Computation of splitting tests In order to compute, in the simplest and more general way, whether each of the two split d-cells is internal or external to the splitting boundary $d-1$-facet, it is necessary to consider the oriented covector ϕ (or one-form) canonically associated to the facet f by the covector representation theorem, i.e. the corresponding oriented hyperplane. In this case, the internal/external attribute of the split cell will be computed by evaluating the pairing $< v, \phi >$.

⟨ Splitting tests 7b ⟩ ≡

```
""" Splitting tests """
def testingSubspace(V,covector):
   def testingSubspace0(vcell):
      inout = SIGN(sum([INNERPROD([[1.]+V[v],covector]) for v in vcell]))
      return inout
   return testingSubspace0

def cuttingTest(covector,polytope,V):
   signs = [INNERPROD([covector, [1.]+V[v]]) for v in polytope]
   signs = eval(vcode(signs))
   return any([value<-0.001 for value in signs]) and any([value>0.001 for value in signs])

def tangentTest(covector,facet,adjCell,V):
   common = list(set(facet).intersection(adjCell))
   signs = [INNERPROD([covector, [1.]+V[v]]) for v in common]
   count = 0
   for value in signs:
      if -0.0001<value<0.0001: count +=1
   if count >= len(V[0]):
      return True
   else:
      return False
```
◇

Macro referenced in 8.

Elementary splitting test Let us remember that the adjacency matrix between d-cells is computed via SpMSpM multiplication by the double application

`adjacencyQuery(V,CV)(cell)`,

where the first application `adjacencyQuery(V,CV)` returns a partial function with bufferisation of the adjacency matrix, and the second application to `cell` returns the list of adjacent d-cells sharing with it a $(d-1)$-dimensional facet.

⟨ Elementary splitting test 8 ⟩ ≡

⟨Splitting tests 7b⟩

```
""" Elementary splitting test """
def dividenda(V,CV, cell,facet,covector,unchosen):
   out = []
   adjCells = adjacencyQuery(V,CV)(cell)
   for adjCell in set(adjCells).difference(unchosen):
      if (cuttingTest(covector,CV[adjCell],V) and \
         cellFacetIntersecting(facet,adjCell,covector,V,CV)) or \
         tangentTest(covector,facet,CV[adjCell],V): out += [adjCell]
   return out
```

◇

Macro referenced in 15a.

CDC cell splitting with one or more facets When splitting a d-cell with some hyperplanes, we need to return not only either the two cut parts or the cell itself when the hyperplane is tangent to a $(d-1)$-face, but also the facet lying on the hyperplane. In the first cade it is directly computed by the `SPLITCELL` function, and returned as the `equal` set of points. In the second case, the cell is transformed by the map that sends the hyperplane in the $x_d = 0$ subspace ($z = 0$ in 3D), and the searched facet is returned as the (back-transformed) set of cell vertices on this subspace.

Actually, the process is strongly complicated by the fact that the input cell (and its facets) may be cut by several hyperplanes. By now, we resort to the simplex computation, even if more time-expensive: to compare each vertex of each cell fragment, against every hyperplanes. This approach will adapt well to the writing of a computational kernel on the GPU.

⟨CDC cell splitting with one or more cutting facets 9a⟩ ≡

```
""" CDC cell splitting with one or more cutting facets """
def fragment(cell,cellCuts,V,CV,BC):
   vcell = CV[cell]
   cellFragments = [[V[v] for v in vcell]]

   for f in cellCuts[cell]:
      facet = BC[f]
      plane = COVECTOR([V[v] for v in facet])
      for k,fragment in enumerate(cellFragments):

         #if not tangentTest(plane,facet,fragment,V):
         [below,equal,above] = SPLITCELL(plane,fragment,tolerance=1e-4,ntry=4)
         if below != above:
            cellFragments[k] = below
            cellFragments += [above]
```

```
        facets = facetsOnCuts(cellFragments,cellCuts,V,BC)
    return cellFragments
◇
```

Macro referenced in 15a.

SCDC splitting with every boundary facet The function `makeSCDC` is used to compute the LAR model `(W,CW)` of the SCDC. It takes as input the LAR model `(V,CV)` of the CDC, and the LAR model `(V,BC)` of the input Boolean Complex, and returns also the vertex-cell relation `VC`, i.e. the transposed of `CV`.

For every `k` ∈ `BC`, a list `cellsToSplit`

⟨SCDC splitting with every boundary facet 9b⟩ ≡

```
""" SCDC splitting with every boundary facet """
def makeSCDC(V,CV,BC):
    index,defaultValue = -1,-1
    VC = invertRelation(CV)
    CW,BCfrags = [],[]
    Wdict = dict()
    BCellcovering = boundaryCover(V,CV,BC,VC)
    #print "\nBCellcovering =",BCellcovering
    #BCellcovering = [cell if cell!=[] else [-1] for cell in BCellcovering ]

    cellCuts = invertRelation(BCellcovering)
    for k in range(len(CV) - len(cellCuts)): cellCuts += [[]]

    def verySmall(number): return abs(number) < 10**-5.5

    for k,frags in enumerate(cellCuts):
        if cellCuts[k] == []:
            cell = []
            for v in CV[k]:
                key = vcode(V[v])
                if Wdict.get(key,defaultValue) == defaultValue:
                    index += 1
                    Wdict[key] = index
                    cell += [index]
                else:
                    cell += [Wdict[key]]
            CW += [cell]
        else:
            cellFragments = fragment(k,cellCuts,V,CV,BC)
            for cellFragment in cellFragments:
                cellFrag = []
                for v in cellFragment:
                    key = vcode(v)
```

```
                if Wdict.get(key,defaultValue) == defaultValue:
                    index += 1
                    Wdict[key] = index
                    cellFrag += [index]
                else:
                    cellFrag += [Wdict[key]]
            CW += [cellFrag]

            BCfrags += [[Wdict[vcode(w)] for w in cellFragment if verySmall(
                        PROD([ COVECTOR( [V[v] for v in BC[h]] ), [1.]+w ])) ]
                      for h in cellCuts[k]]

    W = sorted(zip( Wdict.values(), Wdict.keys() ))
    W = AA(eval)(TRANS(W)[1])
    dim = len(W[0])
    BCfrags = [str(sorted(facet)) for facet in BCfrags if facet != [] and len(set(facet)) >= dim
    BCfrags = sorted(list(AA(eval)(set(BCfrags))))
    print "\nBCfrags =", BCfrags
    print "\nW =", W
    return W,CW,VC,BCellcovering,cellCuts,BCfrags
◇
```

Macro referenced in 15a.

Computation of boundary facets covering with CDC cells

⟨Computation of boundary facets covering with CDC cells 10⟩ ≡

```
""" Computation of boundary facets covering with CDC cells """
def boundaryCover(V,CV,BC,VC):
    cellsToSplit = list()
    boundaryCellCovering = []
    for k,facet in enumerate(BC):
        covector = COVECTOR([V[v] for v in facet])
        seedsOnFacet = VC[facet[0]]
        cellsToSplit = [dividenda(V,CV, cell,facet,covector,[]) for cell in seedsOnFacet ]
        cellsToSplit = set(CAT(cellsToSplit))
        while True:
            newCells = [dividenda(V,CV, cell,facet,covector,cellsToSplit) for cell in cellsToSplit
            if newCells != []: newCells = CAT(newCells)
            covering = cellsToSplit.union(newCells)
            if covering == cellsToSplit:
                break
            cellsToSplit = covering
        boundaryCellCovering += [list(covering)]
    return boundaryCellCovering
◇
```

Macro referenced in 15a.

Cell-facet intersection test

⟨ Cell-facet intersection test 11 ⟩ ≡

```
""" Cell-facet intersection test """
def cellFacetIntersecting(boundaryFacet,cell,covector,V,CV):
   points = [V[v] for v in CV[cell]]
   vcell1,newFacet,vcell2 = SPLITCELL(covector,points,tolerance=1e-4,ntry=4)
   boundaryFacet = [V[v] for v in boundaryFacet]
   translVector = boundaryFacet[0]

   # translation
   newFacet = [ VECTDIFF([v,translVector]) for v in newFacet ]
   boundaryFacet = [ VECTDIFF([v,translVector]) for v in boundaryFacet ]

   # linear transformation: boundaryFacet -> standard (d-1)-simplex
   d = len(V[0])
   transformMat = mat( boundaryFacet[1:d] + [covector[1:]] ).T.I

   # transformation in the subspace x_d = 0
   newFacet = (transformMat * (mat(newFacet).T)).T.tolist()
   boundaryFacet = (transformMat * (mat(boundaryFacet).T)).T.tolist()

   # projection in E^{d-1} space and Boolean test
   newFacet = MKPOL([ AA(lambda v: v[:-1])(newFacet), [range(1,len(newFacet)+1)], None ])
   boundaryFacet = MKPOL([ AA(lambda v: v[:-1])(boundaryFacet), [range(1,len(boundaryFacet)+1)]
   verts,cells,pols = UKPOL(INTERSECTION([newFacet,boundaryFacet]))

   if verts == []: return False
   else: return True
```

◇

Macro referenced in 15a.

4 Step 3: cell labeling

The goal of this stage is to label every cell of the SCDC with two bits, corresponding to the input spaces A and B, and telling whether the cell is either internal (1) or external (0) to either spaces.

4.1 Requirements

Input The output of previous algorithm stage.

Output The array `cellLabels` with *shape* `len(PW)` $\times 2$, and values in $\{0, 1\}$.

4.2 Implementation

The labelling of LAR of the SCDC may be decomposed in five consecutive steps. The first step was actually executed during the splitting stage, by accumulating a single facet of every split cells embedded on the affine hull (the covector hyperplane) of the splitting boundary `facet`. The second step provides the computation of the sparse matrix of the linear coboundary operator $\delta_{d-1} : C_{d-1} \to C_d$. The third step operates upon the previous two pieces of information, in order to compute the coboundary chain of the boundary chain of both input Boolean arguments. The fourth step attaches a IN/OUT label to each d-cell of the previously computed d-chain. Finally, the fifth step spreads around the labels to cover all the d-cells of SCDC. This knowledge allows for the computation of every interesting Boolean expressions between the input complexes.

Computation of boundary cells embedded in SCDC

⟨ Computation of embedded boundary cells 12 ⟩ ≡

```
""" Computation of embedded boundary cells """
def facetsOnCuts(cellFragments,cellCuts,V,BC):


    pass
    return #facets
```

◇

Macro referenced in 15a.

Coboundary operator on SCDC space decomposition In this section we develop a stronger characterisation of the boundaries, by fully tagging in SCDC the internal coboundary of boundaries of A and B Boolean arguments. This novel strategy should allow the recursive tagging extension to work correctly in all cases.

As we know, the coboundary operators $\delta_{k-1} : C_{k-1} \to C_k$ are the transpose of the boundary operators $\partial_k : C_k \to C_{k-1}$ $(1 \le k \le d)$. We therefore proceed to the construction of the operator δ_{d-1}, according to the procedure illustrated in []. For this purpose we need to use both the C_d and the C_{d-1} bases of SCDC. The first basis is generated as `CV` array during the splitting. The second basis will be built from C_d using the proper d-adjacency algorithm from [].

Let us remember that a (co)boundary operator may be applied to *any* chain from the linear space of chains defined upon a cellular complex. In our case we have already generated the $(d-1)$-chains ∂A and ∂B while building the SCDC, by accumulating, in the course of the splitting phase, the $(d-1)$-facets discovered while tracking the boundaries of A and B. We just need now to tag (a subset of) $\delta_{d-1}\partial_d A$ and $\delta_{d-1}\partial_d B$.

⟨ Coboundary operator on the convex decomposition of common space 13 ⟩ ≡

```
""" Coboundary operator on the convex decomposition of common space """
from scipy.spatial import ConvexHull
def qhullBoundary(V):
   points = array(V)
   hull = ConvexHull(points)
   out = hull.simplices.tolist()
   return sorted(out)

""" Extracting a $(d-1)$-basis of SCDC """
def larConvexFacets (V,CV):
   dim = len(V[0])
   model = V,CV
   V,FV = larFacets(model,dim)
   FV = sorted(FV + convexBoundary(V,CV))
   return FV
◇
```

Macro referenced in 15a.

Computation of boundary operator The computation of the boundary operator ∂_d on the SCDC d-basis `(W,CW)` requires the knowledge of the $(d-1)$-basis `(W,FW)`. The goal of this section is hence the—partially incremental—computation of `FW`. This set can be partitioned into *internal* cells, that have 2 cofaces, and *boundary* cells, that have only 1 coface. The first subset is easily computed by the `larFacets` function; the computation of the second subset requires some more work, specified in the following.

First, we compute the 0-chain of boundary vertices of the SCDC, using *qHull*, and take advantage of the `CV` matrix to extract the chain of d-cells sharing with the boundary a $(d-1)$facet. Second, using the *partial* boundary operator generated by using only the interior $(d-1)$-facets, and the associated $(d-2)$-boundary operator, we select the sub-chain made by the non-closed d-cells of this subset. Third, the boundary facet of each of them is finally selected, added to the $(d-1)$-basis of SCDC, and the corresponding row is added at the bottom line of the matrix of ∂_{d-1}.

⟨Computation of boundary operator of a convex LAR model 14a⟩ ≡

```
""" Computation of boundary operator of a convex LAR model"""
def convexBoundary(W,CW):
   points = array(W)
   hull = ConvexHull(points,qhull_options="Qc")
   coplanarVerts = hull.coplanar.tolist()
   if coplanarVerts != []:  coplanarVerts = CAT(coplanarVerts)
   BWchain = set( CAT(qhullBoundary(W)) + coplanarVerts )
   dim = len(W[0])
   bfacets = [list(BWchain.intersection(cell))
               for cell in CW if len(BWchain.intersection(cell)) >= dim]
   return bfacets
```

◇

Macro referenced in 15a.

Coboundary of boundary chains

⟨ Coboundary of boundary chain 14b ⟩ ≡

```
""" Coboundary of boundary chain """
```

◇

Macro never referenced.

Labeling seeds

⟨ Writing labelling seeds on SCDC 14c ⟩ ≡

```
""" Writing labelling seeds on SCDC """
```

◇

Macro never referenced.

Recursive diffusion of labels

⟨ Recursive diffusion of labels on SCDC 14d ⟩ ≡

```
""" Recursive diffusion of labels on SCDC """
```

◇

Macro never referenced.

5 Step 4: greedy cell gathering

The goal of this stage is to make as lower as possible the number of cells in the output LAR of the space AB, partitioned into convex cells.

Input The LAR model `(W,PW)` of the SCDC and the array `cellLabels`.

Output The LAR representation `(W,RW)` of the final fragmented and labeled space AB.

6 Exporting the library

`"lib/py/bool1.py"` 15a ≡

```
""" Module for Boolean ops with LAR """
⟨Initial import of modules 21a⟩
from splitcell import *
DEBUG = False
⟨Symbolic utility to represent points as strings 21b⟩
⟨Merge two dictionaries with keys the point locations 4b⟩
⟨Make Common Delaunay Complex 5⟩
⟨Cell-facet intersection test 11⟩
⟨Elementary splitting test 8⟩
⟨Computing the adjacent cells of a given cell 6⟩
⟨Computation of boundary facets covering with CDC cells 10⟩
⟨CDC cell splitting with one or more cutting facets 9a⟩
⟨SCDC splitting with every boundary facet 9b⟩
⟨Characteristic matrix transposition 7a⟩
⟨Computation of embedded boundary cells 12⟩
⟨Coboundary operator on the convex decomposition of common space 13⟩
⟨Computation of boundary operator of a convex LAR model 14a⟩
◇
```

7 Tests and examples

⟨Debug input and vertex merging 15b⟩ ≡

```
V1,basis1 = arg1
V2,basis2 = arg2
cells1 = basis1[-1]
cells2 = basis2[-1]

if DEBUG: VIEW(STRUCT(MKPOLS((V1,basis1[1])) + MKPOLS((V2,basis2[1]))))

model1,model2 = (V1,cells1),(V2,cells2)
V, CV1,CV2, n1,n12,n2 = mergeVertices(model1,model2)  #<<<<<<<<<<<<<<<<<

submodel = SKEL_1(STRUCT(MKPOLS((V,CV1+CV2))))
VV = AA(LIST)(range(len(V)))
if DEBUG: VIEW(STRUCT([ submodel,larModelNumbering(V,[VV,_,CV1+CV2],submodel,3)]))

V,CV,vertDict,n1,n12,n2,BC = makeCDC(arg1,arg2)    #<<<<<<<<<<<<<<<<<

W,CW,VC,BCellCovering,cellCuts,BCfrags = makeSCDC(V,CV,BC)

assert len(VC) == len(V)
assert len(BCellCovering) == len(BC)

submodel = STRUCT([ SKEL_1(STRUCT(MKPOLS((V,CV)))), COLOR(RED)(STRUCT(MKPOLS((V,BC)))) ])
dim = len(V[0])
```

```
VIEW(STRUCT([ submodel,larModelNumbering(V,[VV,BC,CV],submodel,3)]))
VIEW(EXPLODE(2,2,2)(MKPOLS((W,CW))))
"""
for k in range(1,len(CW)+1):
   VIEW(STRUCT([ STRUCT(MKPOLS((W,CW[:k]))), submodel,larModelNumbering(V,[VV,BC,CV],submodel,3
"""


WW = AA(LIST)(range(len(W)))
FW = larConvexFacets (W,CW)
#submodel = SKEL_1(STRUCT(MKPOLS((W,CW))))
#VIEW(larModelNumbering(W,[WW,FW,CW],submodel,3))

VIEW(EXPLODE(1.5,1.5,1)(MKPOLS((W,FW))))
```

◇

Macro referenced in 16, 17ab, 18ab, 19ab, 20ab.

"test/py/bool1/test1.py" 16 ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

""" Definition of Boolean arguments """
V1 = [[3,0],[11,0], [13,10], [10,11], [8,11], [6,11], [4,11], [1,10], [4,3], [6,4],
   [8,4], [10,3]]
FV1 = [[0,1,8,9,10,11],[1,2,11], [3,10,11], [4,5,9,10], [6,8,9], [0,7,8], [2,3,11],
   [3,4,10], [5,6,9], [6,7,8]]
EV1 = [[0,1],[0,7],[0,8],[1,2],[1,11],[2,3],[2,11],[3,4],[3,10],[3,11],[4,5],[4,10],[5,6],[5,9
VV1 = AA(LIST)(range(len(V1)))

V2 = [[0,3],[14,2], [14,5], [14,7], [14,11], [0,8], [3,7], [3,5]]
FV2 =[[0,5,6,7], [0,1,7], [4,5,6], [2,3,6,7], [1,2,7], [3,4,6]]
EV2 = [[0,1],[0,5],[0,7],[1,2],[1,7],[2,3],[2,7],[3,4],[3,6],[4,5],[4,6],[5,6],[6,7]]
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
```

⟨Debug input and vertex merging 15b⟩

◇

"test/py/bool1/test2.py" 17a ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

V1 = [[3,0],[11,0], [13,10], [10,11], [8,11], [6,11], [4,11], [1,10], [4,3], [6,4],
   [8,4], [10,3]]

FV1 = [[0,1,8,9,10,11],[1,2,11], [3,10,11], [4,5,9,10], [6,8,9], [0,7,8]]
EV1 = [[0,1],[0,7],[0,8],[1,2],[1,11],[2,11],[3,10],[3,11],[4,5],[4,10],[5,9],[6,8],[6,9],[7,8
VV1 = AA(LIST)(range(len(V1)))

V2 = [[0,3],[14,2], [14,5], [14,7], [14,11], [0,8], [3,7], [3,5]]
FV2 =[[0,5,6,7], [0,1,7], [4,5,6], [2,3,6,7], [1,2,7], [3,4,6]]
EV2 = [[0,1],[0,5],[0,7],[1,2],[1,7],[2,3],[2,7],[3,4],[3,6],[4,5],[4,6],[5,6],[6,7]]
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test3.py" 17b ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

V1 = [[3,0],[11,0], [13,10], [10,11], [8,11], [6,11], [4,11], [1,10], [4,3], [6,4],
   [8,4], [10,3]]

FV1 = [[0,1,8,9,10,11],[1,2,11], [3,10,11], [4,5,9,10], [6,8,9], [0,7,8]]
EV1 = [[0,1],[0,7],[0,8],[1,2],[1,11],[2,11],[3,10],[3,11],[4,5],[4,10],[5,9],[6,8],[6,9],[7,8
VV1 = AA(LIST)(range(len(V1)))

V2 = [[0,3],[14,2], [14,5], [14,7], [14,11], [0,8], [3,7], [3,5]]
FV2 =[[0,5,6,7], [0,1,7], [4,5,6], [2,3,6,7]]
EV2 = [[0,1],[0,5],[0,7],[1,7],[2,3],[2,7],[3,6],[4,5],[4,6],[5,6],[6,7]]
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test4.py" 18a ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

V1 = [[0,0],[10,0],[10,10],[0,10]]
FV1 = [range(4)]
EV1 = [[0,1],[1,2],[2,3],[0,3]]
VV1 = AA(LIST)(range(len(V1)))

V2 = [[2.5,2.5],[12.5,2.5],[12.5,12.5],[2.5,12.5]]
FV2 = [range(4)]
EV2 = [[0,1],[1,2],[2,3],[0,3]]
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test5.py" 18b ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

V1 = [[0,0],[5,0],[5,5],[0,5]]
FV1 = [range(4)]
EV1 = [[0,1],[1,2],[2,3],[0,3]]
VV1 = AA(LIST)(range(len(V1)))

V2 = [[5,0],[10,0],[10,5],[5,5]]
FV2 = [range(4)]
EV2 = [[0,1],[1,2],[2,3],[0,3]]
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test6.py" 19a ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

V1 = [[0,0,0],[10,0,0],[10,10,0],[0,10,0],[0,0,10],[10,0,10],[10,10,10],[0,10,10]]
V1,[VV1,EV1,FV1,CV1] = larCuboids((1,1,1),True)
V1 = [SCALARVECTPROD([5,v]) for v in V1]

V2 = [SUM([v,[2.5,2.5,2.5]]) for v in V1]
[VV2,EV2,FV2,CV2] = [VV1,EV1,FV1,CV1]

arg1 = V1,(VV1,EV1,FV1,CV1)
arg2 = V2,(VV2,EV2,FV2,CV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test7.py" 19b ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *


V1 = [[0,0],[10,0],[10,10],[0,10]]
FV1 = [range(4)]
EV1 = [[0,1],[1,2],[2,3],[0,3]]
VV1 = AA(LIST)(range(len(V1)))

V2 = [[2.5,2.5],[7.5,2.5],[7.5,7.5],[2.5,7.5]]
FV2 = [range(4)]
EV2 = [[0,1],[1,2],[2,3],[0,3]]
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test8.py" 20a ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *
```

```
n = 48
V1 = [[5*cos(angle*2*PI/n)+2.5, 5*sin(angle*2*PI/n)+2.5] for angle in range(n)]
FV1 = [range(n)]
EV1 = TRANS([range(n),range(1,n+1)]); EV1[-1] = [0,n-1]
VV1 = AA(LIST)(range(len(V1)))

V2 = [[4*cos(angle*2*PI/n), 4*sin(angle*2*PI/n)] for angle in range(n)]
FV2 = [range(n)]
EV2 = EV1
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

"test/py/bool1/test9.py" 20b ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool1 import *

V1 = [[0,0],[15,0],[15,14],[0,14]]
FV1 = [range(4)]
EV1 = [[0,1],[1,2],[2,3],[0,3]]
VV1 = AA(LIST)(range(len(V1)))

V2 = [[1,1],[7,1],[7,6],[1,6], [8,1],[14,1],[14,7],[8,7], [1,7],[7,7],[7,13],[1,13], [8,8],[14
FV2 = [range(4),range(4,8),range(8,12),range(12,16)]
EV2 = [[0,1],[1,2],[2,3],[0,3], [4,5],[5,6],[6,7],[4,7], [8,9],[9,10],[10,11],[8,11], [12,13],
VV2 = AA(LIST)(range(len(V2)))

arg1 = V1,(VV1,EV1,FV1)
arg2 = V2,(VV2,EV2,FV2)
⟨Debug input and vertex merging 15b⟩
◇
```

A Appendix: utility functions

⟨Initial import of modules 21a⟩ ≡

```
from pyplasm import *
from scipy import *
import sys
""" import modules from larcc/lib """
```

```
sys.path.insert(0, 'lib/py/')
from lar2psm import *
from simplexn import *
from larcc import *
from largrid import *
from myfont import *
from mapper import *
◇
```

Macro referenced in 15a.

A.1 Numeric utilities

A small set of utility functions is used to transform a *point* representation, given as array of coordinates, into a string of fixed format to be used as point key into python dictionaries.

⟨Symbolic utility to represent points as strings 21b⟩ ≡

```
""" TODO: use package Decimal (http://docs.python.org/2/library/decimal.html) """
global PRECISION
PRECISION = 5.

def verySmall(number): return abs(number) < 10**-(PRECISION)

def prepKey (args): return "["+", ".join(args)+"]"

def fixedPrec(value):
   out = round(value*10**(PRECISION))/10**(PRECISION)
   if out == -0.0: out = 0.0
   return str(out)

def vcode (vect):
   """
   To generate a string representation of a number array.
   Used to generate the vertex keys in PointSet dictionary, and other similar operations.
   """
   return prepKey(AA(fixedPrec)(vect))
◇
```

Macro referenced in 15a.

References

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