

# Boolean combinations of cellular complexes as chain operations \*

Alberto Paoluzzi

August 21, 2014

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Preview of the algorithm . . . . .	1
<b>2</b>	<b>Merging arguments</b>	<b>2</b>
2.1	Reordering of vertex coordinates . . . . .	2
2.1.1	Re-indexing of vertices . . . . .	2
2.1.2	Re-indexing of d-cells . . . . .	3
2.1.3	Example of input with some coincident vertices . . . . .	4
2.1.4	Example . . . . .	5
<b>3</b>	<b>Selecting cells to split</b>	<b>5</b>
<b>4</b>	<b>Splitting cells traversing the boundaries</b>	<b>6</b>
4.1	Cell splitting . . . . .	7
4.2	Cross-building of two task dictionaries . . . . .	8
4.3	Updating the vertex set and dictionary . . . . .	9
4.4	Updating the split cell and the queues of seeds . . . . .	10
4.5	Updating the cells adjacent to the split cell . . . . .	12
<b>5</b>	<b>Reconstruction of results</b>	<b>13</b>
5.1	The Boolean algorithm flow . . . . .	13
5.2	Final traversal of the CDC . . . . .	13
<b>6</b>	<b>Exporting the library</b>	<b>15</b>

---

\*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [?].  
August 21, 2014

<b>7 Tests</b>	<b>15</b>
7.1 2D examples . . . . .	15
7.1.1 First examples . . . . .	15
7.1.2 Two squares . . . . .	16
<b>A Appendix: utility functions</b>	<b>18</b>
A.1 Numeric utilities . . . . .	18

## 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.

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

### 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  $V_{ab}$ .
3. Split the (highest-dimensional) cells of CDC crossed by  $\partial A$  or  $\partial B$ . 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  $\alpha, \beta : C_d \rightarrow \{0, 1\}$ , such that:

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

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

### 1.1.1 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 through.

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 Merging arguments

### 2.1 Reordering of vertex coordinates

A global reordering of vertex coordinates is executed as the first step of the Boolean algorithm, in order to eliminate the duplicate vertices, by substituting duplicate vertex copies (coming from two close points) with a single instance.

Two dictionaries are created, then merged in a single dictionary, and finally split into three subsets of (vertex,index) pairs, with the aim of rebuilding the input representations, by making use of a novel and more useful vertex indexing.

The union set of vertices is finally reordered using the three subsets of vertices belonging

(a) only to the first argument, (b) only to the second argument and (c) to both, respectively denoted as  $V_1, V_2, V_{12}$ . A top-down description of this initial computational step is provided by the set of macros discussed in this section.

@D Place the vertices of Boolean arguments in a common space @""" First step of Boolean Algorithm """ @i Initial indexing of vertex positions @i @j Merge two dictionaries with keys the point locations @i @j Filter the common dictionary into three subsets @i @j Compute an inverted index to reorder the vertices of Boolean arguments @i @j Return the single reordered pointset and the two  $d$ -cell arrays @i @

### 2.1.1 Re-indexing of vertices

**Initial indexing of vertex positions** The input LAR models are located in a common space by (implicitly) joining  $V_1$  and  $V_2$  in a same array, and (explicitly) shifting the vertex indices in  $CV_2$  by the length of  $V_1$ . @D Initial indexing of vertex positions @from collections import defaultdict, OrderedDict

""" TODO: change defaultdict to OrderedDefaultdict """

class OrderedDefaultdict(collections.OrderedDict): def *init*(self,\*args,\*\*kwargs): if not args: self.default\_factory=None else: if n

def *missing*(self,key): if self.default\_factory is None: raise KeyError(key) self[key]=default=self.default\_factory() return default

def *reduce*(self): optional, for pickle support args=(self.default\_factory,) if self.default\_factory is not None: return self.class, args, None, None, self.iter

def vertexSieve(model1, model2): from lar2psm import larModelBreak V1,CV1 = larModelBreak(model1) V2,CV2 = larModelBreak(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) @

**Merge two dictionaries with point location as keys** Since currently  $CV_1$  and  $CV_2$  point to a set of vertices larger than their initial sets  $V_1$  and  $V_2$ , we re-index the set  $V_1 \cup V_2$  using a Python `defaultdict` dictionary, in order to avoid errors of “missing key”. As dictionary keys, we use the string representation of the vertex position vector, with a given fixed floating-point approximation, as provided by the `vcode` function discussed in the in the Appendix of this document. @D Merge two dictionaries with keys the point locations @ 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] @

**Example of string coding of a vertex position** The position vector of a point of real coordinates is provided by the function `vcode`. An example of coding is given below. The *precision* of the string representation can be tuned at will.

```
>>> vcode([-0.011660381062724849, 0.297350056848685860])
'[-0.0116604, 0.2973501]'
```

**Filter the common dictionary into three subsets** `Verdict`, dictionary of vertices, uses as key the position vectors of vertices coded as string, and as values the list of integer indices of vertices on the given position. If the point position belongs either to the first or to second argument only, it is stored in `case1` or `case2` lists respectively. If the position (`item.key`) is shared between two vertices, it is stored in `case12`. The variables `n1`, `n2`, and `n12` remember the number of vertices respectively stored in each repository. @D Filter the common dictionary into three subsets @ `case1, case12, case2 = [], [], []` for `item` in `verdict.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)` @

**Compute an inverted index to reorder the vertices of Boolean arguments** The new indices of vertices are computed according with their position within the storage repositories `case1`, `case2`, and `case12`. Notice that every `item[1]` stored in `case1` or `case2` is a list with only one integer member. Two such values are conversely stored in each `item[1]` within `case12`. @D Compute an inverted index to reorder the vertices of Boolean arguments @ `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` @

### 2.1.2 Re-indexing of d-cells

**Return the single reordered pointset and the two *d*-cell arrays** We are now finally ready to return two reordered LAR models defined over the same set  $V$  of vertices, and where (a) the vertex array  $V$  can be written as the union of three disjoint sets of points  $C_1, C_{12}, C_2$ ; (b) the *d*-cell array  $CV1$  is indexed over  $C_1 \cup C_{12}$ ; (b) the *d*-cell array  $CV2$  is indexed over  $C_{12} \cup C_2$ .

The `vertexSieve` function will return the new reordered vertex set  $V = (V_1 \cup V_2) \setminus (V_1 \cap V_2)$ , the two renumbered *s*-cell sets  $CV1$  and  $CV2$ , and the size `len(case12)` of  $V_1 \cap V_2$ . @D Return the single reordered pointset and the two *d*-cell arrays @  $V = [\text{eval}(p[0]) \text{ for } p \text{ in } case1] + [\text{eval}(p[0]) \text{ for } p \text{ in } case12] + [\text{eval}(p[0]) \text{ for } p \text{ in } case2]$   $CV1 = [\text{sorted}([\text{invertedindex}[v] \text{ for } v \text{ in } cell]) \text{ for } cell \text{ in } CV1]$   $CV2 = [\text{sorted}([\text{invertedindex}[v] \text{ for } v \text{ in } cell]) \text{ for } cell \text{ in } CV2]$  return  $V, CV1, CV2, \text{len}(case12)$  @

### 2.1.3 Example of input with some coincident vertices

In this example we give two very simple LAR representations of 2D cell complexes, with some coincident vertices, and go ahead to re-index the vertices, according to the method implemented by the function `vertexSieve`.

@o test/py/bool/test02.py @@i Initial import of modules @i from bool import \*  $V1 = [[1,1],[3,3],[3,1],[2,3],[2,1],[1,3]]$   $V2 = [[1,1],[1,3],[2,3],[2,2],[3,2],[0,1],[0,0],[2,0],[3,0]]$   $CV1$

```
= [[0,3,4,5],[1,2,3,4]] CV2 = [[3,4,7,8],[0,1,2,3,5,6,7]] model1 = V1,CV1; model2 = V2,CV2
VIEW(STRUCT([ COLOR(CYAN)(SKEL1(STRUCT(MKPOLS(model1)))), COLOR(RED)(SKEL1(ST
```

**Example discussion** The aim of the `vertexSieve` function is twofold: (a) eliminate vertex duplicates before entering the main part of the Boolean algorithm; (b) reorder the input representations so that it becomes less expensive to check whether a 0-cell can be shared by both the arguments of a Boolean expression, so that its coboundaries must be eventually split. Remind that for any set it is:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Let us notice that in the previous example

$$|V| = |V_1 \cup V_2| = 12 \leq |V_1| + |V_2| = 6 + 9 = 15,$$

and that

$$|V_1| + |V_2| - |V_1 \cup V_2| = 15 - 12 = 3 = |C_{12}| = |V_1 \cap V_2|,$$

where  $C_{12}$  is the subset of vertices with duplicated instances. @D Output from `test/py/boolean/test02.py`  
@V = [[3.0,1.0],[2.0,1.0],[3.0,3.0],[1.0,1.0],[1.0,3.0],[2.0,3.0], [3.0,2.0],[2.0,0.0],[2.0,2.0],[0.0,0.0],[3.0,0.0],[0.0,1.0]]  
CV1 = [[3,5,1,4],[2,0,5,1]] CV2 = [[8,6,7,10],[3,4,5,8,11,9,7]] @ Notice also that V has been  
reordered in three consecutive subsets  $C_1, C_{12}, C_2$  such that CV1 is indexed within  $C_1 \cup C_{12}$ ,  
whereas CV2 is indexed within  $C_{12} \cup C_2$ . In our example we have  $C_{12} = \{3, 4, 5\}$ : @D Re-  
ordering of vertex indexing of cells @ `sorted(CAT(CV1))` [0, 1, 1, 2, 3, 4, 5, 5] `sorted(CAT(CV2))` [3, 4, 5, 6, 7, 7, 8, 8, 9, 10, 11] @

**Cost analysis** Of course, this reordering after elimination of duplicate vertices will allow to perform a cheap  $O(n)$  discovering of (Delaunay) cells whose vertices belong both to V1 and to V2. Actually, the *same test* can be now used both when the vertices of the input arguments are all different, and when they have some coincident vertices. The total cost of such pre-processing, executed using dictionaries, is  $O(n \ln n)$ .

#### 2.1.4 Example

**Building a covering of common convex hull** @D Building a covering of common convex hull  
@def covering(model1,model2,dim=2,emptyCellNumber=1): V, CV1, CV2, n12 =  
vertexSieve(model1,model2) ,EEV1 = *larFacets*((V,CV1),dim,emptyCellNumber),EEV2 =  
*larFacets*((V,CV2),dim,emptyCellNumber)*if emptyCellNumber!* = 0 : CV1 = CV1[:  
-*emptyCellNumber*]*if emptyCellNumber!* = 0 : CV2 = CV2[: -*emptyCellNumber*]|VV =  
AA(LIST)(range(len(V)))returnV, [VV,EEV1,EEV2,CV1,CV2],n12@

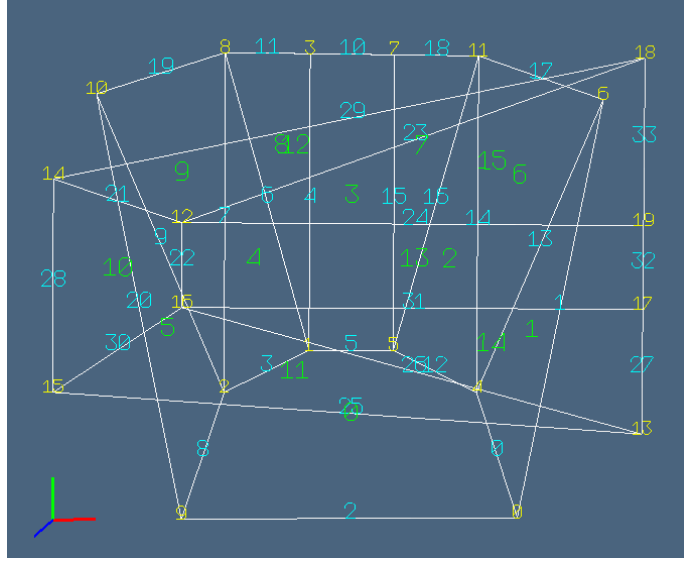


Figure 1: Set covering of the two Boolean arguments.

### 3 Selecting cells to split

The aim of this section is to provide some fast method to select a subset of CDC cells where to start the splitting of the CDC along the  $(d - 1)$  boundary facets of operand complexes. Of course, a lot of useful information is provided by the incidence relation  $VC$  between CDC vertices and  $d$ -cells.

Two dictionaries are used in order to split the CDC and compute the SCDC. The dictionary `dict_fc` is used with key a boundary  $(d - 1)$ -face and value the (dynamic) list of CDC  $d$ -cells crossed (and later split) by it. Conversely, the `dict_cf` dictionary is used with key a CDC  $d$ -cell and with value the list of boundary  $(d - 1)$ -faces crossing it.

Two different strategies may be used for boundary facets terminating by crossing the interior of some CDC cell, and for facets sharing the tangent space of the boundary of such cells. Alternatively to what initially implemented, all the boundary  $(d - 1)$ -faces must be considered as “splitting seeds”, and tracked against the current state of the SCDC.

**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.

```
@D Characteristic matrix transposition @""" Characteristic matrix transposition """
def invertRelation(V,CV): VC = [[] for k in range(len(V))] for k,cell in enumerate(CV): for
```

v in cell: VC[v] += [k] return VC @

### 3.1 Computing the boundary hyperplanes (BHs)

For each boundary  $(d-1)$ -face the affine hull is computed, producing a set of pairs (**face**, **covector**).

@D New implementation of splitting dictionaries @""" New implementation of splitting dictionaries """ VC = invertRelation(V,CV)

covectors = [] for faceVerts in BC: points = [V[v] for v in faceVerts] dim = len(points[0]) theMat = Matrix( [(dim+1)\*[1.] + [p+[1.] for p in points] ) covector = [(-1)\*\*(col)\*theMat.minor(0,col).determinant() for col in range(dim+1)] covectors += [covector]

@j Association of covectors to d-cells @i @j Initialization of splitting dictionaries @i @

### 3.2 Association of BHs to $d$ -cells of CDC

Every pair (**face**, **covector**) is associated uniquely to a single  $d$ -cell of CDC, producing a set of triples (**face**, **covector**, **cell**). Two cases are possible: (a) the face hyperplane crosses the interior of the cell; (b) the face hyperplane contains the face, so that the cell is left on the interior subspace of the (oriented) face covector.

For this purpose, it is checked that at least one of the face vertices, transformed into the common-vertex-based coordinate frame, have all positive coordinates. This fact guarantees the existence of a non trivial intersection between the  $(d-1)$ -face and the  $d$ -cell.

@D Association of covectors to d-cells @""" to compute a single d-cell associated to (face,covector) """ def covectorCell(face,faceVerts,covector,CV,VC): incidentCells = VC[faceVerts[0]] for cell in incidentCells: cellVerts = CV[cell] v0 = list(set(faceVerts).intersection(cellVerts))[0] v0 = common vertex transformMat = mat([DIFF([V[v],V[v0]]) for v in cellVerts if v != v0]).T.I vects = (transformMat \* (mat([DIFF([V[v],V[v0]]) for v in faceVerts if v != v0]).T)).T.tolist() if any([all([x\_i > -0.0001 for x in list(vect)]) for vect in vects]): return [face,cell,covector] print "error: found no face,cell,covector", "" @

### 3.3 Initialization of splitting dictionaries

The triples (**face**, **cell**, **covector**), computed by the **covectorCell** function, is suitably accommodated into two dictionaries denoted as **dict\_fc** (for *face*, *cell*) and **dict\_cf** (for *cell*, *face*), respectively.

@D Initialization of splitting dictionaries @""" Initialization of splitting dictionaries """ tasks = [] for face,covector in zip(range(len(BC)),covectors): tasks += [covectorCell(face,BC[face],covector,CV,VC)]

dict\_fc, dict\_cf = initTasks(tasks)print " > dict\_cf", dict\_cfprint " > dict\_fc", dict\_fc, "" @



## 4 Splitting cells traversing the boundaries

In the previous section we computed a set of "split seeds", each made by a boundary facet and by a Delaunay cell to be split by the facet's affine hull. Here we show how to partitionate each such cells into two cells, according to Figure 2, where the boundary facets of the two boolean arguments are shown in yellow color.

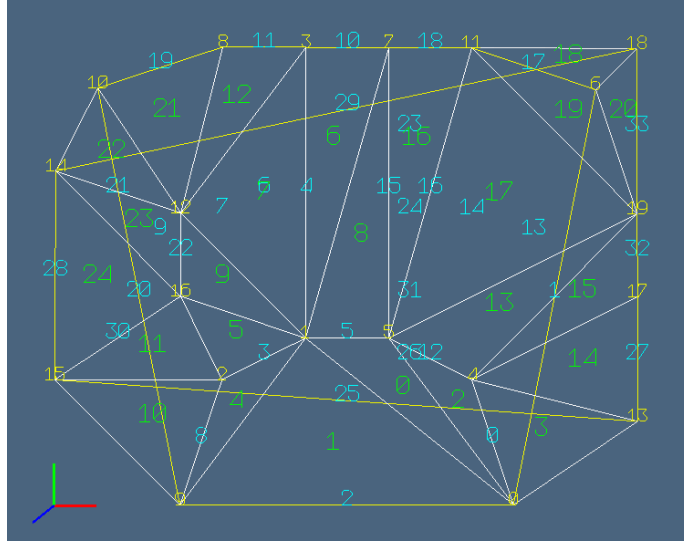


Figure 2: example caption

In the example in Figure 2, the set of pairs (**facet**,**cell**) to be used as split seeds are given below.

`[[25, 3], [1, 3], [29, 18], [20, 22], [1, 19], [25, 10], [20, 10], [29, 22]]`

### 4.1 Cell splitting

A cell will be split by pyplasm intersection with a suitable rotated and translated instance of a (large)  $d$ -cuboid with the superior face embedded in the hyperplane  $z = 0$ .

**Splitting a cell with an hyperplane** The macro below defines a function `cellSplitting`, with input the index of the **face**, the index of the **cell** to be bisected, the **covector** giving the coefficients of the splitting hyperplane, i.e. the affine hull of the splitting **face**, and the arrays **V**, **EEV**, **CV**, giving the coordinates of vertices, the (accumulated) facet to vertices relation (on the input models), and the cell to vertices relation (on the Delaunay model), respectively.

The actual subdivision of the input `cell` onto the two output cells `cell1` and `cell2` is performed by using the `pyplasm` Boolean operations of intersection and difference of the input with a solid simulation of the needed hyperspace, provided by the `rototranslSubspace` variable. Of course, such `pyplasm` operators return two `Hpc` values, whose vertices will then be extracted using the `UKPOL` primitive.

```
@D Cell splitting @""" Cell splitting in two cells """ def cellSplitting(face,cell,covector,V,EEV,CV):
    dim = len(V[0]) subspace = (T(range(1,dim+1))(dim*[-50])(CUBOID(dim*[100]))) nor-
    mal = covector[:-1] if len(normal) == 2: 2D complex rotatedSubspace = R([1,2])(ATAN2(normal)-
    PI/2)(T(2)(-50)(subspace)) elif len(normal) == 3: 3D complex rotatedSubspace = R()(subspace)
    else: print "rotation error" t = V[EEV[face][0]] rototranslSubspace = T(range(1,dim+1))(t)(rotatedSubspace)
    cellHpc = MKPOL([V,[v+1 for v in CV[cell]]],None)
    cell1 = INTERSECTION([cellHpc,rototranslSubspace]) tolerance=0.0001 use_ootree =
    False cell1 = Plasm.boolop(BOOL_CODE_AND, [cellHpc,rototranslSubspace],tolerance,plasm_config.maxn
    UKPOL(cell1)cell1 = AA(vcode)(verts)
    cell2 = DIFFERENCE([cellHpc,rototranslSubspace]) cell2 = Plasm.boolop(BOOL_CODE_DIFF, [cellHpc,r
    UKPOL(cell2)cell2 = AA(vcode)(verts)
    return cell1,cell2 @
```

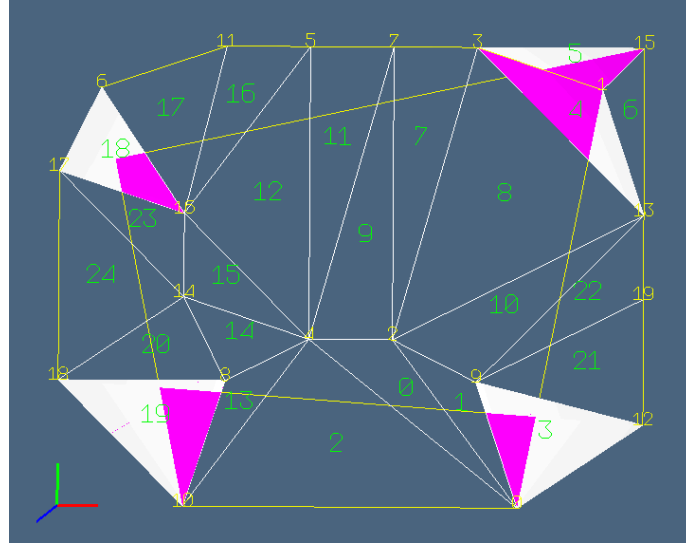


Figure 3: example caption

## 4.2 Cross-building of two task dictionaries

The correct and efficient splitting of the combined Delaunay complex (CDC) with the (closed and orientable) boundaries of two Boolean arguments, requires the use of two special dictionaries, respectively named `dict_fc` (for *face-cell*), and `dict_cf` (for *cell-face*).

On one side, for each splitting facet  $((d-1)$ -face), used as key, we store in `dict_fc` the list of traversed  $d$ -cells of CDC, starting in 2D with the two cells containing the two extreme vertices of the cutting edge, and in higher dimensions, with all the  $d$ -cells containing one of vertices of the splitting  $(d-1)$ -face.

On the other side, for each  $d$ -cell to be split, used as key, we store in `dict_cf` the list of cutting  $(d-1)$ -cells, since a single  $d$ -cell may be traversed and split by more than one facet.

**Init face-cell and cell-face dictionaries** @D Init face-cell and cell-face dictionaries

```
@""" Init face-cell and cell-face dictionaries """
def initTasks(tasks): dict_fc = defaultdict(list) dict_cf =
defaultdict(list) for task in tasks : face, cell, covector = task dict_fc[face] += [(cell, covector)] dict_cf[cell] +=
[(face, covector)] return dict_fc, dict_cf @
```

**Example of face-cell and cell-face dictionaries** @D Example of face-cell and cell-face dictionaries @""" Example of face-cell and cell-face dictionaries """

```
tasks (face, cell)
= [ [0, 4, [-10.0, 2.0, 110.0]], [31, 5, [3.0, -14.0, 112.0]], [17, 18, [10.0, 2.0, -30.0]], [22, 3,
[-1.0, -14.0, 42.0]], [17, 19, [10.0, 2.0, -30.0]], [31, 18, [3.0, -14.0, 112.0]], [22, 19, [-1.0, -14.0,
42.0]], [0, 3, [-10.0, 2.0, 110.0]]
```

```
tasks (dict_fc) = defaultdict(< type'list' >, 0 : [(4, [-10.0, 2.0, 110.0]), (3, [-10.0, 2.0, 110.0])], 17 : [(18, [10.0, 2.0, -30.0]), (19, [10.0, 2.0, -30.0])], 31 : [(18, [3.0, -14.0, 112.0]), (19, [3.0, -14.0, 112.0])], 22 : [(19, [-1.0, -14.0, 42.0]), (18, [-1.0, -14.0, 42.0])], 0 : [(0, [-10.0, 2.0, 110.0]), (3, [-10.0, 2.0, 110.0])])
tasks (dict_cf) = defaultdict(< type'list' >, 19 : [(17, [10.0, 2.0, -30.0]), (22, [-1.0, -14.0, 42.0])], 18 : [(17, [10.0, 2.0, -30.0]), (22, [-1.0, -14.0, 42.0])], 4 : [(0, [-10.0, 2.0, 110.0]), (3, [-10.0, 2.0, 110.0])], 3 : [(0, [-10.0, 2.0, 110.0]), (3, [-10.0, 2.0, 110.0])], 17 : [(18, [10.0, 2.0, -30.0]), (19, [10.0, 2.0, -30.0])], 31 : [(18, [3.0, -14.0, 112.0]), (19, [3.0, -14.0, 112.0])], 22 : [(19, [-1.0, -14.0, 42.0]), (18, [-1.0, -14.0, 42.0])], 0 : [(0, [-10.0, 2.0, 110.0]), (3, [-10.0, 2.0, 110.0])])
```

### 4.3 Updating the vertex set and dictionary

In any dimension, the split of a  $d$ -cell with an hyperplane (crossing its interior) produces two  $d$ -cells and some new vertices living upon the splitting hyperplane.

When the  $d$ -cell  $c$  is contained in only one seed of the CDC decomposition, i.e. when `dict_cf[c]` has cardinality one (in other words: it is crossed only by one boundary facet), the two generated cells `vcell1`, `vcell2` can be safely output, and accommodated in two slots of the CV list.

Conversely, when more than one facet crosses  $c$ , much more care must be taken to guarantee the correct fragmentation of this cell.

**Managing the splitting dictionaries** The function `splittingControl` takes care of cells that must be split several times, as crossed by several boundary faces.

If the dictionary item `dict_cf[cell]` has *length* one (i.e. is crossed *only* by one face) the CV list is updated and the function returns, in order to update the `dict_fc` dictionary.

Otherwise, the function subdivides the facets cutting `cell` between those to be associated to `vcell1` and to `vcell2`. For each pair `aface, covector` in `dict_cf[cell]` and in position following `face` in the list of pairs, check if either `vcell1` or `vcell2` or both, have intersection with the subset of vertices shared between `cell` and `aface`, and respectively

put in `alist1`, in `alist2`, or in both. Finally, store `vcell1` and `vcell2` in `CV`, and `alist1`, `alist2` in `dict_cf`.

```

@D Managing the splitting dictionaries @""" Managing the splitting dictionaries """
def splittingControl(face,cell,covector,vcell,vcell1,vcell2, dict_fc, dict_cf, V, BC, CV, VC, CVbits, lenBC1) :
    boundaryFacet = BC[face] translVector = V[boundaryFacet[0]] tcovector = [cv+tv*covector[-
1] for (cv,tv) in zip( covector[:-1],translVector) ]+[0.0]
    c1,c2 = cell,cell if not haltingSplitTest(cell,vcell,vcell1,vcell2,boundaryFacet, translVec-
tor,tcovector,V) :
        only one facet covector crossing the cell cellVerts = CV[cell] CV[cell] = vcell1 CV +=
[vcell2] CVbits += [copy(CVbits[cell])] c1,c2 = cell,len(CV)-1
        firstCell,secondCell = AA(testingSubspace(V,covector))([vcell1,vcell2]) if face != lenBC1
and firstCell==-1: face in boundary(op1) CVbits[c1][0] = 0 CVbits[c2][0] = 1 elif face
!= lenBC1 and firstCell==-1: face in boundary(op2) CVbits[c1][1] = 0 CVbits[c2][1] =
1 else: print "error splitting face,c1,c2 =",face,c1,c2
        dict_fc[face].remove((cell,covector))removethesplitcelldict_cf[cell].remove((face,covector))removethespl
more than one facet covectors crossing the cell alist1,alist2 = list(),list() for aface,covector
in dict_cf[cell] :
    for each facet crossing the cell compute the intersection between the facet and the cell
faceVerts = BC[aface] commonVerts = list(set(faceVerts).intersection(cellVerts))
    and attribute the intersection to the split subcells if set(vcell1).intersection(commonVerts)
!= set(): alist1.append((aface,covector)) else: dict_fc[aface].remove((cell,covector))
    if set(vcell2).intersection(commonVerts) != set(): alist2.append((aface,covector)) dict_fc[aface] +=
[(len(CV) - 1, covector)]
    dict_cf[cell] = alist1dict_cf[len(CV) - 1] = alist2
    else: dict_fc[face].remove((cell,covector))removethesplitcelldict_cf[cell].remove((face,covector))removet
return V, CV, CVbits, dict_cf, dict_fc, [c1, c2]@

```

#### 4.4 Updating the split cell and the queues of seeds

When a  $d$ -cell of the combined Delaunay complex (CDC) is split into two  $d$ -cells, the first task to perform is to update its representation as vertex list, and to update the list of  $d$ -cells. In particular, as `cell`, and `cell1`, `cell2` are the input  $d$ -cell and the two output  $d$ -cells, respectively, we go to substitute `cell` with `cell1`, and to add the `cell2` as a new row of the  $\text{CSR}(M_d)$  matrix, i.e. as the new terminal element of the `CV` array. Of course, the reverse relation `VC` must be updated too.

**Updating the split cell** First of all notice that, whereas `cell` is given as an integer index to a `CV` row, `cell1`, `cell2` are returned by the `cellSplitting` function as lists of lists of coordinates (of vertices). Therefore such vectors must be suitably transformed into dictionary keys, in order to return the corresponding vertex indices. When transformed

into two lists of vector indices, `cell1`, `cell2` will be in the form needed to update the `CV` and `VC` relations.

**Updating the vertex set of split cells** The code in the macro below provides the splitting of the CDC along the boundaries of the two Boolean arguments. This function, and the ones called by its, provide the dynamic update of the two main data structures, i.e. of the LAR model (`V`,`CV`).

```
@D Updating the vertex set of split cells @""" Updating the vertex set of split cells """
def testingSubspace(V,covector): def testingSubspace0(vcell): inout = SIGN(sum([INNERPROD([V[v]+[1.],co
for v in vcell])) return inout return testingSubspace0
def cuttingTest(covector,polytope,V): signs = [INNERPROD([covector, V[v]+[1.]]) 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(face,polytope,V,BC): faceVerts = BC[face] cellVerts = polytope print
"faceVerts,cellVerts =",faceVerts,cellVerts commonVerts = list(set(faceVerts).intersection(cellVerts))
if commonVerts != []: v0 = commonVerts[0] v0 = common vertex (TODO more gen-
eral) transformMat = mat([DIFF([V[v],V[v0]]) for v in cellVerts if v != v0]).T.I vects =
(transformMat * (mat([DIFF([V[v],V[v0]]) for v in faceVerts if v != v0]).T)).T.tolist() if
all([all([x<=-0.0001 for x in list(vect)]) for vect in vects]): print "vects =",vects return True
else: return False
def splitCellsCreateVertices(vertdict,dict_fc,dict_cf,V,BC,CV,VC,lenBC1) : CVbits =
[[-1,-1]forkinrange(len(CV))]nverts = len(V); cellPairs = []; twoCellIndices = []; whileany([tasks! =
[]forface,tasksindict_fc.items()]) : forface,tasksindict_fc.items() : fortaskintasks : cell,covector =
taskvcell = CV[cell]
cell1,cell2 = cellSplitting(face,cell,covector,V,BC,CV) if cuttingTest(covector,vcell,V):
print "cell1,cell2 =",cell1,cell2 if cell1 == [] or cell2 == []: print "cell1,cell2 =",cell1,cell2
else: adjCells = adjacencyQuery(V,CV)(cell)
vcell1 = [] for k in cell1: if vertdict[k]==[]: vertdict[k] += [nverts] V += [eval(k)]
nverts += 1 vcell1 += [vertdict[k]]
vcell1 = CAT(vcell1) vcell2 = CAT([vertdict[k] for k in cell2])
V,CV,CVbits, dict_cf,dict_fc,twoCells = splittingControl(face,cell,covector,vcell,vcell1,vcell2,dict_fc,d
twoCells[1] :
for adjCell in adjCells: dict_fc[face] += [(adjCell,covector)]dict_cf[adjCell] += [(face,covector)]cellPairs
[[vcell1,vcell2]]twoCellIndices += [[twoCells]]
DEBUG = False if DEBUG: showSplitting(V,cellPairs,BC,CV)
elif tangentTest(face,vcell,V,BC): print "facet tangent to cell"
def inOutTest(face,cell,vertdict,covector,V,BC): vcell = CAT([vertdict[k] for k in cell])
for v in list(set(vcell).difference(BC[face])): inOut = INNERPROD([covector, V[v]+[1.]])
if abs(inOut) < 10**-PRECISION: return sign(inOut)
if cell1 != []: theSign = inOutTest(face,cell1,vertdict,covector,V,BC) if cell2 != []: the-
Sign = inOutTest(face,cell2,vertdict,covector,V,BC) print "theSign =",theSign if theSign
```

```

== 1.0 and face i lenBC1: CVbits[cell][0] = 1 elif theSign == 1.0 and face i= lenBC1:
CVbits[cell][1] = 1 elif theSign == -1.0 and face i lenBC1: CVbits[cell][0] = 0 elif theSign
== -1.0 and face i= lenBC1: CVbits[cell][1] = 0 else: print "error with InOut test" print
"face,cell,covector =",face,cell,covector,"" dict_fc[face].remove((cell,covector))removethesplitcelldict_cf[cell].
else: print "facet out to cell" print "face,cell,covector =",face,cell,covector,"" dict_fc[face].remove((cell,cov

```

**Test for split halting along a boundary facet** The cell splitting is operated by the facet's hyperplane  $H(f)$ , that we call *covector*, and the splitting with it may continues outside  $f$  ... !!

This fact may induce some local errors in the decision procedure (attributing either 0 or 1 to each split cell pair). So, when splitting a pair (**cell,face**) — better: (**cell,covector**) — already stored in the data structure, and then computing its adjacent pairs, we should check if the common facet  $f_{12}$  between  $c_1$  and  $c_2$  is (or is not) at least partially internal to  $f$ .

If this fact is not true, and hence  $f_{12}$  is *out(f)* in the induced topology of the  $H(f)$  hyperplane, the split process on that pair must be halted:  $c_1$  and  $c_2$  are not stored, and their adjacent cells not split.

```

@D Test for split halting along a boundary facet @"" Test for split halting along a
boundary facet "" def haltingSplitTest(cell,vcell,vcell1,vcell2,boundaryFacet,translVector,tcovector,V):
newFacet = list(set(vcell1).intersection(vcell2))

```

```

translation newFacet = [ eval(vcode(VECTDIFF([V[v],translVector]))) for v in new-
Facet ] boundaryFacet = [ eval(vcode(VECTDIFF([V[v],translVector]))) for v in bound-
aryFacet ]

```

```

linear transformation: newFacet -i standard (d-1)-simplex transformMat = mat( bound-
aryFacet[1:] + [tcovector[:-1]] ).T.I

```

```

transformation in the subspace  $x_d = 0$  newFacet = AA(COMP([eval, vcode]))((transformMat*
(mat(newFacet).T)).T.tolist()) boundaryFacet = AA(COMP([eval, vcode]))((transformMat*
(mat(boundaryFacet).T)).T.tolist())

```

```

projection in  $E^{d-1}$  space and Booleantest newFacet = MKPOL([AA(lambdav : v[: -1])(newFacet), [range
1]), None]) boundaryFacet = MKPOL([AA(lambdav : v[: -1])(boundaryFacet), [range(1, len(boundaryFace
1]), None)])
verts, cells, polys = UKPOL(INTERSECTION([newFacet, boundaryFacet])) if verts ==
[] : print "*****cell = ", cell return True else : return False

```

```

cell1 = INTERSECTION([cellHpc, rototranslSubspace]) tolerance=0.0001 use_octree =
False cell1 = Plasm.boolop(BOOLCODE_AND, [cellHpc, rototranslSubspace], tolerance, plasm_config.maxn
UKPOL(cell1) cell1 = AA(vcode)(verts) if @

```

## 4.5 Updating the cells adjacent to the split cell

Once the list of  $d$ -cells has been updated with respect to the results of a split operation, it is necessary to consider the possible update of all the cells that are adjacent to the split one. In particular we need to update their lists of vertices, by introducing the new vertices

produced by the split, and by updating the dictionaries of tasks, by introducing the new (adjacent) splitting seeds.

**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  $\text{CSR}(M_d M_d^t)$ . This can be done by multiplying  $\text{CSR}(M_d)$  by its transposed row corresponding to the query  $d$ -cell.

```
@D Computing the adjacent cells of a given cell @""" 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 @
```

**Updating the adjacency matrix** At every step of the CDC splitting, generating two output cells **cell11** and **cell12** from the input **cell1**, the element of such index in the list **CV** is restored with the **cell11** vertices, and a new (last) element is created in **CV**, to store the **cell12** vertices. Therefore the row of index **cell1** of the symmetric adjacency matrix must be recomputed, being the **cell1** column updated consequently. Also, a new last row (and column) must be added to the matrix.

```
@D Updating the adjacency matrix @""" Updating the adjacency matrix """ pass @
```

**Computation of bits of split cells** 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  $\phi$  (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  $\langle v, \phi \rangle$ .

## 4.6 The Boolean algorithm flow

**Show the process of CDC splitting** @D Show the process of CDC splitting @""" Show
the process of CDC splitting """ def showSplitting(V,cellPairs,BC,CV): VV = AA(LIST)(range(len(V)))
boundaries = COLOR(RED)(SKEL<sub>1</sub>(STRUCT(MKPOL<sub>S</sub>((V,BC))))) submodel = COLOR(CYAN)(STRUCT
[] : cells1,cells2 = TRANS(cellPairs)out = [COLOR(WHITE)(MKPOL([V, [[v +
1forvincell]forcellincells1],None))),COLOR(MAGENTA)(MKPOL([V, [[v+1forvincell]forcellincells2],
VIEW(STRUCT([larModelNumbering(V,[VV,BC,CV],submodel,2)))]@

## 5 Reconstruction of results

Once the SCDC has been constructed, some of its  $d$ -cells are fully characterized, using two bits of information, as either internal (1) or external (0) to one or both the cellular spaces of Boolean arguments  $A$  and  $B$ .

In particular, when a CDC cell was split, the two resulting subcells were both labeled: one as internal, and the other as external to the splitting oriented facet hyperplane. Conversely, when such hyperplane was the support (i.e. the affine hull) of one  $(d - 1)$ -face of the CDC cell, just this cell was characterised as either internal or external to such support hyperplane.

A third value (-1) was used for the initial characterisation of all the SCDC cells, so that at the end of the SCDC construction, every  $d$ -cell is tagged with two values from the set  $\{-1, 0, 1\}$ .

### 5.1 Computing the coboundary of SCDC space

### 5.2 Final traversal of the SCDC

Several cells of the split CDC are characterised as either internal or external to the Boolean arguments  $A$  and  $B$  according to the splitting process. Such characterisation is stored within the `CVbits` array of pairs of values in  $\{-1, 0, 1\}$ , where `CVbits[k][h]`, with  $k \in \text{range}(\text{len}(C_d))$  and  $h \in \text{range}(2)$ , has the following meanings:

$$\text{CVbits}[k][h] = \begin{array}{ll} -1, & \text{if position of } c_k \in C_d \text{ is } \textit{unknown} \text{ w.r.t. complex } K_h \\ 0, & \text{if cell } c_k \in C_d \text{ is } \textit{external} \text{ w.r.t. complex } K_h \\ 1, & \text{if cell } c_k \in C_d \text{ is } \textit{internal} \text{ w.r.t. complex } K_h \end{array}$$

Therefore, a double  $d$ -cell visit of CDC must be executed, starting from some  $d$ -cell interior to either  $A$  or  $B$ , and traversing from a cell to its untraversed adjacent cells, but without crossing the complex boundary, until all cells have been visited.

**The initial computation of chains of Boolean arguments** The initial setting of `CVbits[k][h]` values is done within the splitting process by the `splitCellsCreateVertices` function, and mainly by the `splittingControl` function.

**The traversal of Boolean arguments** Let us remember that the adjacency matrix between  $d$ -cells is computed via SpMSpM multiplication by the double application

$$\text{adjacencyQuery}(\mathbf{V}, \mathbf{CV})(\text{cell}),$$

where the first application `adjacencyQuery(V,CV)` returns a partial function with bufferization 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.



**Traversing a Boolean argument within the CDC** A recursive function `booleanChainTraverse` is given in the script below, where

```
@D Traversing a Boolean argument within the CDC @""" Traversing a Boolean argument within the CDC """
def booleanChainTraverse(h,cell,V,CV,CVbits,value): adjCells = adjacencyQuery(V,CV)(cell) for adjCell in adjCells: if CVbits[adjCell][h] == -1: CVbits[adjCell][h] = value CVbits = booleanChainTraverse(h,adjCell,V,CV,CVbits,value)
return CVbits @
```

**Input and CDC visualisation** @D Input and CDC visualisation @""" Input and CDC visualisation """ submodel1 = mkSignedEdges((V1,BC1)) submodel2 = mkSignedEdges((V2,BC2)) VIEW(STRUCT([submodel1,submodel2])) submodel = SKEL<sub>1</sub>(STRUCT(MKPOLS((V,CV),STRUCT([SKEL<sub>1</sub>(STRUCT(MKPOLS((V,BC))),COLOR(RED)(STRUCT(MKPOLS((V,BC)))))))))

**Boolean fragmentation and classification of CDC** @D Boolean fragmentation and classification of CDC @""" Boolean fragmentation and classification of CDC """

```
def booleanChains(arg1,arg2): (V1,basis1), (V2,basis2) = arg1,arg2 model1, model2 = (V1,basis1[-1]), (V2,basis2[-1]) V,[VV,,CV1,CV2],n12 = covering(model1,model2,2,0) CV = sorted(AA(sorted)(Delaunay(array(V)).vertices)) vertdict = defaultdict(list) for k,vin enumerate(V): vertdict[vcode(v)] += [k]
BC1 = signedCellularBoundaryCells(V1,basis1) BC2 = signedCellularBoundaryCells(V2,basis2)
BC = sorted([[ 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]) BV = list(set(CAT([v for v in BC]))) VV = AA(LIST)(range(len(V)))
print "BC =",BC,"
if DEBUG: @i Input and CDC visualisation @i @i New implementation of splitting dictionaries @i
CVbits,cellPairs,twoCellIndices = splitCellsCreateVertices( vertdict,dictfc,dictcf, V, BC, CV, VC, len(BC1))
print "" for k in range(len(CV)): print "k,CVbits[k],CV[k] =",k,CVbits[k],CV[k]
for cell in range(len(CV)): if CVbits[cell][0] == 1: CVbits = booleanChainTraverse(0,cell,V,CV,CVbits,1)
if CVbits[cell][0] == 0: CVbits = booleanChainTraverse(0,cell,V,CV,CVbits,0) if CVbits[cell][1] == 1: CVbits = booleanChainTraverse(1,cell,V,CV,CVbits,1) if CVbits[cell][1] == 0: CVbits = booleanChainTraverse(1,cell,V,CV,CVbits,0)
print "" for k in range(len(CV)): print "k,CVbits[k],CV[k] =",k,CVbits[k],CV[k]
chain1,chain2 = TRANS(CVbits) print "cf",dictcf print "fc",dictfc, "" return V, CV, chain1, chain2, CVbits
```

## 6 Exporting the library

@O lib/py/bool.py @""" Module for Boolean ops with LAR """ DEBUG = True from matrix import \* @i Initial import of modules @i @i Symbolic utility to represent points as strings @i @i Place the vertices of Boolean arguments in a common space @i @i Building a covering of common convex hull @i @i Building a partition of common convex hull of

vertices @i @j Characteristic matrix transposition @i @j Look for cells in Delaunay, with vertices in both operands @i @j Look for cells in cells12, with vertices on boundaries @i @j Build intersection tasks @i @j Trivial intersection filtering @i @j Cell splitting @i @j Init face-cell and cell-face dictionaries @i @j Updating the split cell @i @j Updating the vertex set of split cells @i @j Managing the splitting dictionaries @i @j Test for split halting along a boundary facet @i @j Computing the adjacent cells of a given cell @i @j Show the process of CDC splitting @i @j Traversing a Boolean argument within the CDC @i @j Boolean fragmentation and classification of CDC @i @

## 7 Tests

### 7.1 2D examples

#### 7.1.1 First examples

Three sets of input 2D data are prepared here, ranging from very simple to a small instance of the hardest kind of dataset, known to produce an output of size  $O(n^2)$ .

```
@D First set of 2D data: Fork-0 input @"" 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],[6,7],[6,8],[6,9],[7,8],[8,9],[9,10]]
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))) @
@D First set of 2D data: Fork-1 input @"" 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]] EV1 = [[0,1],[0,7],[0,8],[1,2],[1,11],[2,11],[3,10],[3,11],[4,5],[4,10],[5,6],[5,9],[6,7],[6,8],[6,9],[7,8],[8,9],[9,10]]
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))) @
```

**Input and visualisation of Boolean arguments** @D Computation of lower-dimensional cells @"" Computation of edges an input visualisation "" model1 = V1,FV1 model2 = V2,FV2 basis1 = [VV1,EV1,FV1] basis2 = [VV2,EV2,FV2] submodel12 = STRUCT(MKPOLS((V1,EV1))+M VIEW(larModelNumbering(V1,basis1,submodel12,4)) VIEW(larModelNumbering(V2,basis2,submodel12,4)) @

**Exporting test file** @D Bulk of Boolean task computation @"" Bulk of Boolean task computation "" @j Computation of lower-dimensional cells @i

```

V,CV,chain1,chain2,CVbits = booleanChains((V1,basis1), (V2,basis2)) for k in range(len(CV)):
print ",CVbits[k],CV[k] =",k,CVbits[k],CV[k] if DEBUG: VIEW(EXPLODE(1.2,1.2,1)(MKPOLs((V,[cell
for cell,c in zip(CV,chain1) if c==1] )))) VIEW(EXPLODE(1.2,1.2,1)(MKPOLs((V,[cell
for cell,c in zip(CV,chain2) if c==1] )))) VIEW(EXPLODE(1.2,1.2,1)(MKPOLs((V,[cell
for cell,c1,c2 in zip(CV,chain1,chain2) if c1+c2==2] )))) VIEW(EXPLODE(1.2,1.2,1)(MKPOLs((V,[cell
for cell,c1,c2 in zip(CV,chain1,chain2) if c1+c2==1] )))) VIEW(EXPLODE(1.2,1.2,1)(MKPOLs((V,[cell
for cell,c1,c2 in zip(CV,chain1,chain2) if c1+c2_i=1] )))) @
@O test/py/bool/test01.py @import sys """ import modules from larcc/lib """ sys.path.insert(0,
'lib/py/') from bool import * @i First set of 2D data: Fork-0 input @i @i Bulk of Boolean
task computation @i @

```

### 7.1.2 Two squares

```

@o test/py/bool/test03.py @""" import modules from larcc/lib """ import sys sys.path.insert(0,
'lib/py/') from bool 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))) @i Bulk of Boolean task computation @i @
@o test/py/bool/test04.py @""" import modules from larcc/lib """ import sys sys.path.insert(0,
'lib/py/') from bool 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))) @i Bulk of Boolean task computation @i @
@o test/py/bool/test05.py @""" import modules from larcc/lib """ import sys sys.path.insert(0,
'lib/py/') from bool import *
V1 = [[2.5,2.5],[7.5,2.5],[7.5,7.5],[2.5,7.5]] 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))) @i Bulk of Boolean task computation @i @

```

## A Appendix: utility functions

```

@D Initial import of modules @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 * @

```

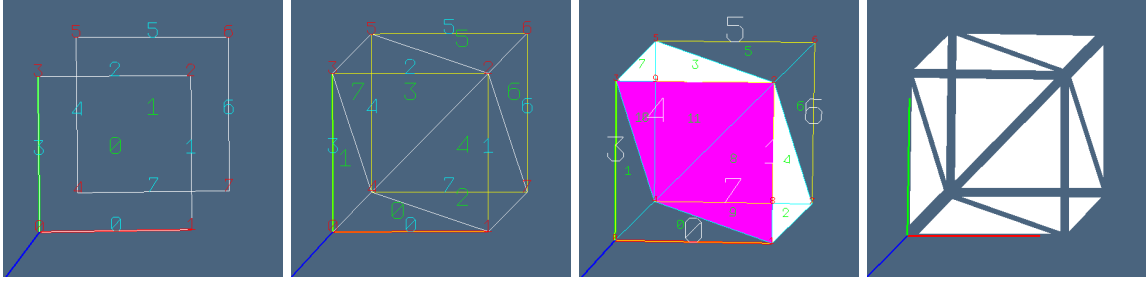


Figure 4: Partitioning of the CDC (Common Delaunay Complex): (a) the two Boolean arguments merged in a single covering; (b) the CDC together with the two (yellow) boundaries; (c) the split CDC cells; (d) the exploded CDC partition.

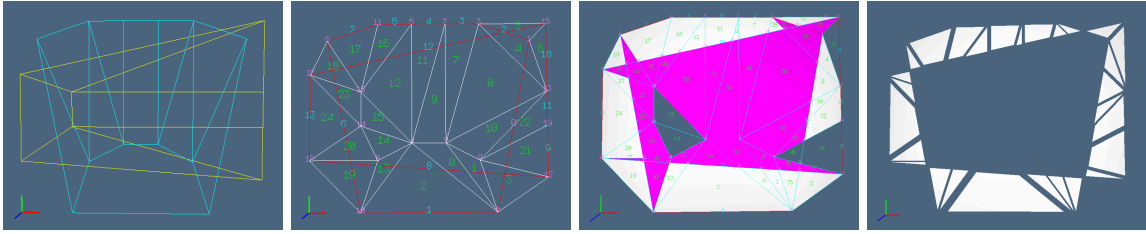


Figure 5: Partitioning of the CDC (Common Delaunay Complex): (a) the two Boolean arguments merged in a single covering; (b) the CDC together with the two (yellow) boundaries; (c) the split CDC cells; (d) the XOR of Boolean arguments.

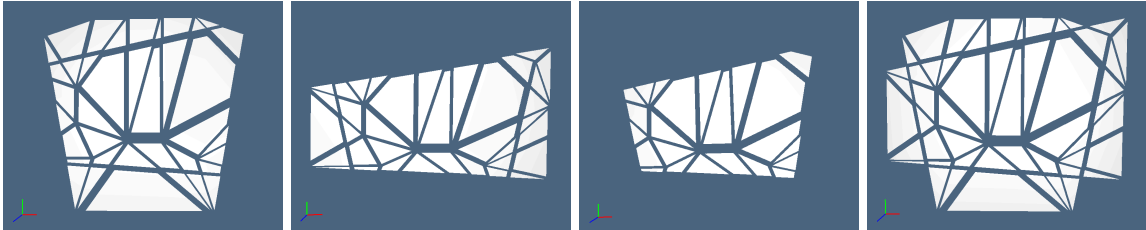


Figure 6: Some chains defined on the CDC (Common Delaunay Complex): (a) the first Boolean argument; (b) the second Boolean argument; (c) the intersection chain; (d) the union chain.

## 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.

```
@D Symbolic utility to represent points as strings @""" TODO: use package Decimal
(http://docs.python.org/2/library/decimal.html) """ global PRECISION PRECISION =
4
```

```
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)) @
```