Boolean combination of cellular complexes *

Alberto Paoluzzi

July 20, 2014

Contents

1

1 Introduction

1.1 Preview of the algorithm

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 ∂A or ∂B;

4. With respect to the (split) CDC basis, compute two coordinate chains $\alpha, \beta : CDC \to \{0, 1\}$, 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 CDC chain corresponding to $AopB$, with $op \in \{\cup, \cap, -\}$.

You may think to the split CDC as a CDT (constrained Delaunay Complex). In part they coincide, but in general, CDC is a polytopal complex (not a simplicial complex).

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.

⟨ Place the vertices of Boolean arguments in a common space 2a ⟩ ≡

```
""" First step of Boolean Algorithm """
```
⟨ Initial indexing of vertex positions 2b ⟩
⟨ Merge two dictionaries with keys the point locations 3 ⟩
⟨ Filter the common dictionary into three subsets 4a ⟩
⟨ Compute an inverted index to reorder the vertices of Boolean arguments 4b ⟩
⟨ Return the single reordered pointset and the two d-cell arrays 5a ⟩
◇

Macro referenced in 20.

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 V1 and V2 in a same array, and (explicitly) shifting the vertex indices in CV2 by the length of V1.

⟨ Initial indexing of vertex positions 2b ⟩ ≡

```
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 not (args[0] is None or callable(args[0])):
                raise TypeError('first argument must be callable or None')
            self.default_factory = args[0]
            args = args[1:]
        super(OrderedDefaultdict, self).__init__(*args, **kwargs)

    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 else tuple()
```

```
                return self.__class__, args, None, None, self.iteritems()


        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)
        ◇
```
Macro referenced in 2a.

Merge two dictionaries with point location as keys Since currently `CV1` and `CV2` point to a set of vertices larger than their initial sets `V1` and `V2`, we index the set $V1 \cup V2$ 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 provided by the `vcode` function given in the Appendix.

⟨ Merge two dictionaries with keys the point locations 3 ⟩ ≡

```
        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]
    ◇
```
Macro referenced in 2a.

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 `Vertdict`, 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.

⟨Filter the common dictionary into three subsets 4a⟩ ≡

```
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)
```
◇

Macro referenced in 2a.

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`.

⟨Compute an inverted index to reorder the vertices of Boolean arguments 4b⟩ ≡

```
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
```
◇

Macro referenced in 2a.

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 $V_1 \cap V_2$.

⟨Return the single reordered pointset and the two d-cell arrays 5a⟩ ≡

5

```
            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, len(case12)
        ◇
```

Macro referenced in 2a.

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`.

```
"test/py/bool/test02.py" 5b ≡
        ⟨Initial import of modules 25a⟩
        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)(SKEL_1(STRUCT(MKPOLS(model1)))),
            COLOR(RED)(SKEL_1(STRUCT(MKPOLS(model2)))) ]))
        ◇
```

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.

⟨Output from `test/py/boolean/test02.py` 6a⟩ ≡

```
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]]
◇
```

Macro never referenced.

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 $C_{12} = \{3,4,5\}$:

⟨ Reordering of vertex indexing of cells 6b ⟩ ≡

```
>>> 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]
◇
```

Macro never referenced.

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

⟨ Building a covering of common convex hull 6c ⟩ ≡
```
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)))
   return V,[VV,EEV1,EEV2,CV1,CV2],n12
◇
```
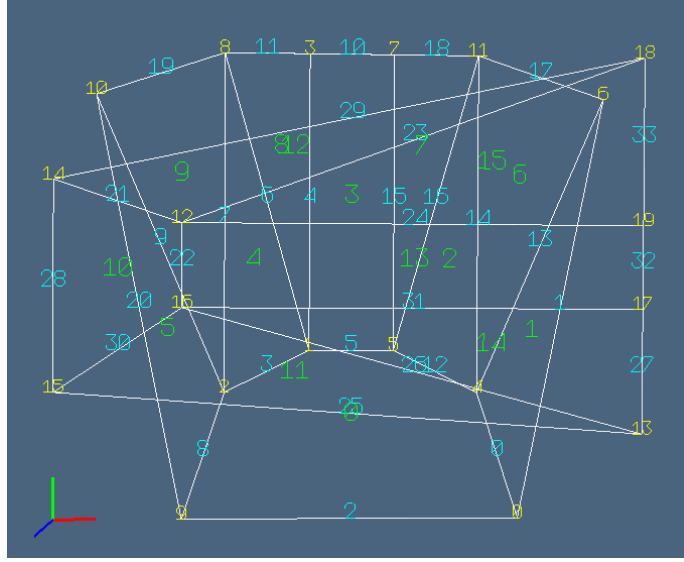
Macro referenced in 20.

7

Figure 1: Set covering of the two Boolean arguments.

3 Selecting cells to split

Relational inversion (Characteristic matrix transposition)

⟨ Characteristic matrix transposition 7a ⟩ ≡

```python
""" 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
```
◇

Macro referenced in 20.

⟨ Look for cells in Delaunay, with vertices in both operands 7b ⟩ ≡

```python
""" Look for cells in Delaunay, with vertices in both operands """
def mixedCells(CV,CV1,CV2,n12):
    n0,n1 = 0, max(AA(max)(CV1))        # vertices in CV1 (extremes included)
    m0,m1 = n1+1-n12, max(AA(max)(CV2))    # vertices in CV2 (extremes included)
    return [list(cell) for cell in CV if any([ n0<=v<=n1 for v in cell])
        and any([ m0<=v<=m1 for v in cell])]
```
◇

Macro referenced in 20.

⟨ Look for cells in cells12, with vertices on boundaries 8a ⟩ ≡

```
""" Look for cells in cells12, with vertices on boundaries """
def mixedCellsOnBoundaries(cells12,BV):
    cells12BV = [cell for cell in cells12
                if len(list(set(cell).intersection(BV))) != 0]
    return cells12BV
```
◇

Macro referenced in 20.

⟨ Build intersection tasks 8b ⟩ ≡

```
""" Build intersection tasks """
def cuttingTest(cuttingHyperplane,polytope,V):
    signs = [INNERPROD([cuttingHyperplane, 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 splittingTasks(V,pivots,BV,BC,VBC,CV,VC):
    tasks = []
    for pivotCell in pivots:
        cutVerts = [v for v in pivotCell if v in BV]
        for v in cutVerts:
            cutFacets = VBC[v]
            cells2cut = VC[v]
            for face,cell in CART([cutFacets,cells2cut]):
                polytope = CV[cell]
                points = [V[w] for w in BC[face]]
                dim = len(points[0])
                theMat = Matrix( [(dim+1)*[1.]] + [p+[1.] for p in points] )
                cuttingHyperplane = [(-1)**(col)*theMat.minor(0,col).determinant()
                            for col in range(dim+1)]
                if cuttingTest(cuttingHyperplane,polytope,V):
                    tasks += [[face,cell,cuttingHyperplane]]
    tasks = AA(eval)(set(AA(str)(tasks)))
    tasks = TrivialIntersection(tasks,V,BC,CV)
    return tasks
```
◇

Macro referenced in 20.

facet-cell trivial intersection filtering A final filtering is applied to the pairs (cutting-Hyperplane,polytope in the tasks array, in order to remove those facets (pairs in 2D) whose intersection reduces to a single point, i.e. to the comman vertex between the boundary $(d-1)$-face, having cuttingHyperplane as affine hull, and the polytope d-cell.

For this purpose, it is checked that at least one of the facet 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.

$\langle$ Trivial intersection filtering 9 $\rangle \equiv$

```
    """ Trivial intersection filtering """
    def TrivialIntersection(tasks,V,EEV,CV):
        out = []
        for face,cell,affineHull in tasks:
            faceVerts, cellVerts = EEV[face], CV[cell]
            v0 = list(set(faceVerts).intersection(cellVerts))[0] # v0 = common vertex
            transformMat = mat([VECTDIFF([V[v],V[v0]]) for v in cellVerts if v != v0]).T.I
            vects = (transformMat * (mat([VECTDIFF([V[v],V[v0]]) for v in faceVerts
                        if v != v0]).T)).T.tolist()
            if any([all([x>0 for x in list(vect)]) for vect in vects]):
                out += [[face,cell,affineHull]]
        return out
    ◇
```

Macro referenced in 20.

4   Splitting cells traversing the boundaries

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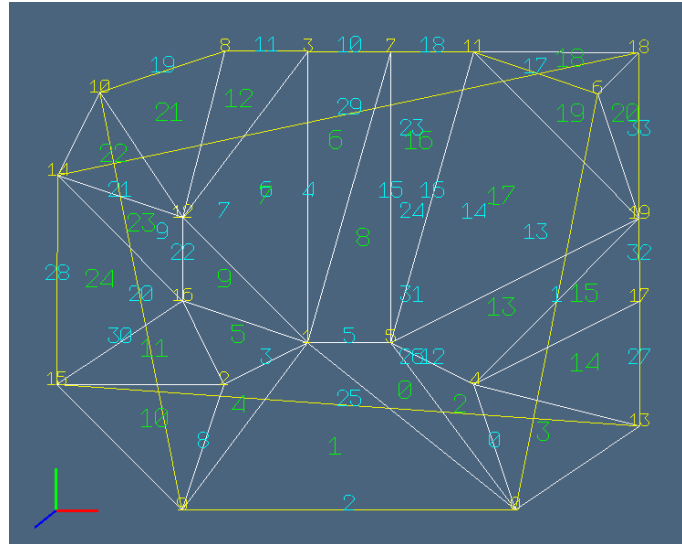

Figure 2: example caption

In the example in Figure 2, the set of pairs (facet,cell) to be used as splitting 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 extracted using the `UKPOL` primitive.

⟨Cell splitting 10⟩ ≡

```
""" 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])))
    normal = 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_octree=False
    cell1 = Plasm.boolop(BOOL_CODE_AND,
        [cellHpc,rototranslSubspace],tolerance,plasm_config.maxnumtry(),use_octree)
    verts,cells,pols = UKPOL(cell1)
    cell1 = AA(vcode)(verts)

    # cell2 = DIFFERENCE([cellHpc,rototranslSubspace])
```

```
cell2 = Plasm.boolop(BOOL_CODE_DIFF,
    [cellHpc,rototranslSubspace],tolerance,plasm_config.maxnumtry(),use_octree)
verts,cells,pols = UKPOL(cell2)
cell2 = AA(vcode)(verts)

return cell1,cell2
```
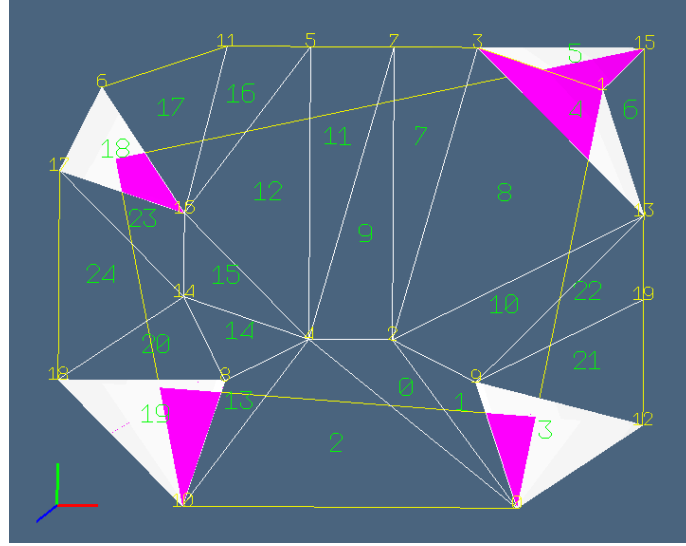◇

Macro referenced in .

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.

12

Init face-cell and cell-face dictionaries

⟨ Init face-cell and cell-face dictionaries 12a ⟩ ≡

```
""" 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
```
◇

Macro referenced in 20.

Example of face-cell and cell-face dictionaries

⟨ Example of face-cell and cell-face dictionaries 12b ⟩ ≡

```
""" 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])],
 22: [(3, [-1.0, -14.0, 42.0]), (19, [-1.0, -14.0, 42.0])],
 31: [(5, [3.0, -14.0, 112.0]), (18, [3.0, -14.0, 112.0])]  })

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]), (31, [3.0, -14.0, 112.0])],
  3: [(22, [-1.0, -14.0, 42.0]), (0, [-10.0, 2.0, 110.0])],
  4: [(0, [-10.0, 2.0, 110.0])],
  5: [(31, [3.0, -14.0, 112.0])]  })
```
◇

Macro never referenced.

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.

TODO: update dict_fc ...

⟨ Managing the splitting dictionaries 13 ⟩ ≡

```
""" Managing the splitting dictionaries """
def splittingControl(face,cell,covector,vcell1,vcell2,dict_fc,dict_cf,V,BC,CV,VC,CVbits,lenBC1

    # 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))  # remove the split cell
```

```
        dict_cf[cell].remove((face,covector))  # remove the splitting face

        # 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] = alist1
        dict_cf[len(CV)-1] = alist2
        return V,CV,CVbits, dict_cf, dict_fc,[c1,c2]
    ◇
```

Macro referenced in [20].

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).

⟨ Updating the vertex set of split cells 14 ⟩ ≡
```
    """ Updating the vertex set of split cells """
    def testingSubspace(V,covector):
        def testingSubspace0(vcell):
            inout = SIGN(sum([INNERPROD([V[v]+[1.],covector]) for v in vcell]))
            return inout
        return testingSubspace0

    def splitCellsCreateVertices(vertdict,dict_fc,dict_cf,V,BC,CV,VC,lenBC1):
        DEBUG = False; CVbits = [[-1,-1] for k in range(len(CV))]
        nverts = len(V); cellPairs = []; twoCellIndices = []; cuttingFaces = [];
        while any([tasks != [] for face,tasks in dict_fc.items()]) :
            for face,tasks in dict_fc.items():
                for task in tasks:
```

```
                cell,covector = task
                if cuttingTest(covector,CV[cell],V):
                    cell1,cell2 = cellSplitting(face,cell,covector,V,BC,CV)
                    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,vcell1,vcell2, dict_fc,dict_cf,V,BC,CV,VC,CVbits,lenBC

                        for adjCell in adjCells:
                            if cuttingTest(covector,CV[adjCell],V) and not ((face,covector) in dict_c
                                dict_fc[face] += [(adjCell,covector)]
                                dict_cf[adjCell] += [(face,covector)]
                        cellPairs += [[vcell1, vcell2]]
                        twoCellIndices += [[twoCells]]
                        cuttingFaces += [face]
                    if DEBUG: showSplitting(V,cellPairs,BC,CV)
                else:
                    dict_fc[face].remove((cell,covector))   # remove the split cell
                    dict_cf[cell].remove((face,covector))   # remove the splitting face
                showSplitting(V,cellPairs,BC,CV)
        #for k in range(len(CV)):  print "k,CVbits[k],CV[k] =",k,CVbits[k],CV[k]
        return CVbits,cellPairs,twoCellIndices,cuttingFaces
    ◇
```

Macro referenced in

4.4 Updating the split cell and the stack 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 split cell 16a ⟩ ≡
```
""" Updating the split cell """
def splitCellUpdate(cell,vcell1,vcell2,CV):
   newVerts = list(set(vcell1).difference(CV[cell]))
   return newVerts
```
◇
Macro referenced in 20.

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

⟨ Computing the adjacent cells of a given cell 16b ⟩ ≡
```
""" 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 20.

17

Updating the adjacency matrix At every step of the CDC splitting, generating two output cells `cell1` and `cell2` from the input `cell`, the element of such index in the list `CV` is restored with the `cell1` vertices, and a new (last) element is created in `CV`, to store the `cell2` vertices. Therefore the row of index `cell` of the symmetric adjacency matrix must be recomputed, being the `cell` column updated consequently. Also, a new last row (and column) must be added to the matrix.

⟨ Updating the adjacency matrix 17a ⟩ ≡

```
    """ Updating the adjacency matrix """
    pass
```
◇

Macro never referenced.

5 Reconstruction of results

5.1 The Boolean algorithm flow

Show the process of CDC splitting

⟨ Show the process of CDC splitting 17b ⟩ ≡

```
    """ Show the process of CDC splitting """
    def showSplitting(V,cellPairs,BC,CV):
       VV = AA(LIST)(range(len(V)))
       boundaries = COLOR(RED)(SKEL_1(STRUCT(MKPOLS((V,BC)))))
       submodel = COLOR(CYAN)(STRUCT([ SKEL_1(STRUCT(MKPOLS((V,CV)))), boundaries ]))
       if cellPairs != []:
          cells1,cells2 = TRANS(cellPairs)
          out = [COLOR(WHITE)(MKPOL([V,[[v+1 for v in cell] for cell in cells1],None])),
                 COLOR(MAGENTA)(MKPOL([V,[[v+1 for v in cell] for cell in cells2],None]))]
          VIEW(STRUCT([ STRUCT(out),larModelNumbering(V,[VV,BC,CV],submodel,2) ]))
       else:
          VIEW(STRUCT([ larModelNumbering(V,[VV,BC,CV],submodel,2) ]))
```
◇

Macro referenced in 20.

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 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 $< v, \phi >$.

5.2 Final traversal of the CDC

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 $k \in$ `range(len(`C_d`))` and $h \in$ `range(2)`, has the following meanings:

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

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

$$\texttt{adjacencyQuery(V,CV)(cell)},$$

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

⟨Traversing a Boolean argument within the CDC 18⟩ ≡

```
    """ 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
    ◇
```

Macro referenced in 20.

Boolean fragmentation and classification of CDC

⟨ Boolean fragmentation and classification of CDC 19 ⟩ ≡

```
""" 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,v in enumerate(V): vertdict[vcode(v)] += [k]

   BC1 = signedCellularBoundaryCells(V1,basis1)
   BC2 = signedCellularBoundaryCells(V2,basis2)
   submodel1 = mkSignedEdges((V1,BC1))
   submodel2 = mkSignedEdges((V2,BC2))
   VIEW(STRUCT([submodel1,submodel2]))

   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)))
   submodel = SKEL_1(STRUCT(MKPOLS((V,CV))))
   VIEW(larModelNumbering(V,[VV,BC,CV],submodel,4))
   submodel = STRUCT([SKEL_1(STRUCT(MKPOLS((V,CV)))), COLOR(RED)(STRUCT(MKPOLS((V,BC))))])
   VIEW(larModelNumbering(V,[VV,BC,CV],submodel,4))

   cells12 = mixedCells(CV,CV1,CV2,n12)
   pivots = mixedCellsOnBoundaries(cells12,BV)
   VBC = invertRelation(V,BC)
   VC = invertRelation(V,CV)
   tasks = splittingTasks(V,pivots,BV,BC,VBC,CV,VC)
   dict_fc,dict_cf = initTasks(tasks)

   CVbits,cellPairs,twoCellIndices,cuttingFaces = splitCellsCreateVertices(
      vertdict,dict_fc,dict_cf,V,BC,CV,VC,len(BC1))
   showSplitting(V,cellPairs,BC,CV)

   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)

        for k in range(len(CV)):  print "k,CVbits[k],CV[k] =",k,CVbits[k],CV[k]
        chain1,chain2 = TRANS(CVbits)
        return V,CV,chain1,chain2
    ◇
```

Macro referenced in 20.

6   Exporting the library

`"lib/py/bool.py" 20 ≡`
```
    """ Module for Boolean ops with LAR """
    from matrix import *
```
⟨Initial import of modules 25a⟩
⟨Symbolic utility to represent points as strings 25b⟩
⟨Place the vertices of Boolean arguments in a common space 2a⟩
⟨Building a covering of common convex hull 6c⟩
⟨Building a partition of common convex hull of vertices ?⟩
⟨Characteristic matrix transposition 7a⟩
⟨Look for cells in Delaunay, with vertices in both operands 7b⟩
⟨Look for cells in cells12, with vertices on boundaries 8a⟩
⟨Build intersection tasks 8b⟩
⟨Trivial intersection filtering 9⟩
⟨Cell splitting 10⟩
⟨Init face-cell and cell-face dictionaries 12a⟩
⟨Updating the split cell 16a⟩
⟨Updating the vertex set of split cells 14⟩
⟨Managing the splitting dictionaries 13⟩
⟨Computing the adjacent cells of a given cell 16b⟩
⟨Show the process of CDC splitting 17b⟩
⟨Traversing a Boolean argument within the CDC 18⟩
⟨Boolean fragmentation and classification of CDC 19⟩
    ◇

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 $O(n^2)$.

⟨First set of 2D data: Fork-0 input 21a⟩ ≡

```
""" 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)))
    ◇
```

Macro never referenced.

⟨First set of 2D data: Fork-1 input 21b⟩ ≡
```
""" 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],[5,9],[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)))
    ◇
```

Macro referenced in 22b.

Input and visualisation of Boolean arguments

⟨Computation of lower-dimensional cells 21c⟩ ≡
```
""" 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))+MKPOLS((V2,basis2[1])))
VIEW(larModelNumbering(V1,basis1,submodel12,4))
VIEW(larModelNumbering(V2,basis2,submodel12,4))
    ◇
```

Macro referenced in 22a.

Exporting test file

⟨ Bulk of Boolean task computation 22a ⟩ ≡

```
    """ Bulk of Boolean task computation """
```
⟨ Computation of lower-dimensional cells 21c ⟩

```
    V,CV,chain1,chain2 = booleanChains((V1,basis1), (V2,basis2))

    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>=1] )))
```

⋄

Macro referenced in 22b, 23ab, 24.

"test/py/bool/test01.py" 22b ≡

```
    import sys
    """ import modules from larcc/lib """
    sys.path.insert(0, 'lib/py/')
    from bool import *
```
⟨ First set of 2D data: Fork-1 input 21b ⟩
⟨ Bulk of Boolean task computation 22a ⟩

⋄

7.1.2   Two squares

"test/py/bool/test03.py" 23a ≡

```
    """ 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],[3,0]]
    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],[3,0]]
    VV2 = AA(LIST)(range(len(V2)))
```
⟨ Bulk of Boolean task computation 22a ⟩

⋄

```
"test/py/bool/test04.py" 23b ≡
    """ 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],[3,0]]
    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],[3,0]]
    VV2 = AA(LIST)(range(len(V2)))
    ⟨Bulk of Boolean task computation 22a⟩
    ⋄

"test/py/bool/test05.py" 24 ≡
    """ 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],[3,0]]
    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],[3,0]]
    VV2 = AA(LIST)(range(len(V2)))
    ⟨Bulk of Boolean task computation 22a⟩
    ⋄
```

A Appendix: utility functions

⟨Initial import of modules 25a⟩ ≡
```
    from pyplasm import *
    from scipy import *
    import sys
    """ import modules from larcc/lib """
    sys.path.insert(0, 'lib/py/')
    from lar2psm import *
```
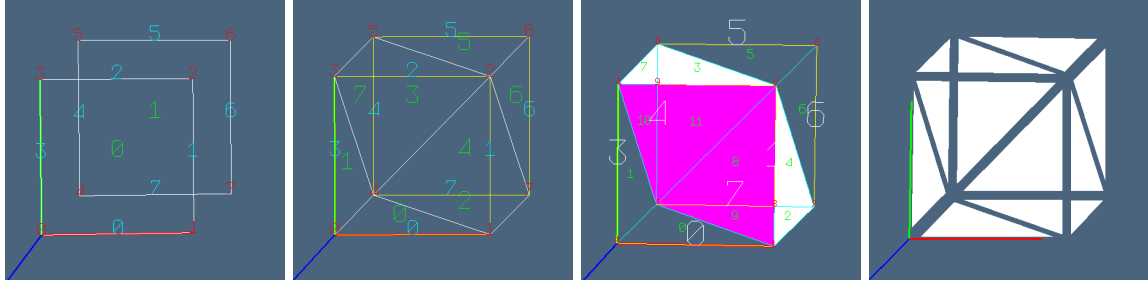
Figure 4: Partitioning of the CDC (Common Delaunay Complex): (a) the two Boolean arguments merged in a single covering; (b) the CDC together with the two (yellow) boundaries; (c) the split CDC cells; (d) the exploded CDC partition.
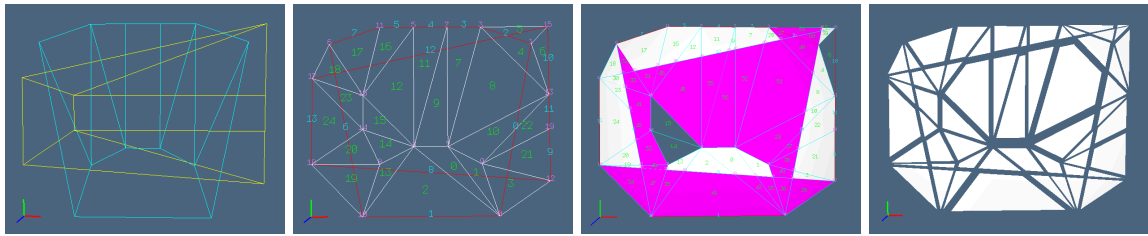

Figure 5: Partitioning of the CDC (Common Delaunay Complex): (a) the two Boolean arguments merged in a single covering; (b) the CDC together with the two (yellow) boundaries; (c) the split CDC cells; (d) the exploded CDC partition.

```
from simplexn import *
from larcc import *
from largrid import *
from myfont import *
from mapper import *
```
◇

Macro referenced in 5b, 20.

A.1   Numeric utilities

A small set of utilityy functions is used to transform a point representation 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 25b⟩ ≡

```
""" 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))
```
◇

Macro referenced in 20.

References

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