Boolean Chains: set operations with topological chains [*]

Alberto Paoluzzi

May 1, 2014

Abstract

Boolean operations are a major addition to every geometric package. Union, intersection, difference and complementation of decomposed spaces are discussed and implemented in this module by making use of the Linear Algebraic Representation (LAR) introduced in [DPS14]. First, the two finite decompositions are merged, by merging their vertices (0-cells of support spaces); then a Delaunay complex based on the vertex set union is computed, and the shared d-chain is extracted and split, according to the cellular structure of the input d-chains. The results of a Boolean operation are finally computed by sum, product or difference of the (binary) coordinate representation of the (split) argument chains, by using the novel chain-basis resulted from the splitting step. Differently from the totality of algorithms known to the authors, neither search nor traversal of some (complicated) data structure is performed by this algorithm.

Contents

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. May 1, 2014

1 Introduction

In this section we introduce and shortly outline our novel algorithm for Boolean operations with chain of cells from different space decompositions implemented in this LAR-CC software module.

The input objects are denoted in the remainder as X_1 and X_2, and their finite cell decompositions as Λ^1 and Λ^1. Our goal is to compute $X = X_1\, op\, X_2$, where $op \in \{\cup, \cap, -, \ominus\}$ or $\complement X$, based on a common decomposition $\Lambda = \Lambda^1\, op\, \Lambda^2$, with Λ being a suitably fragmented decomposition of the X space.

Of course, we aim to compute a minimal (in some sense) decomposition, making the best use of the LAR framework, based on CSR representation of sparse binary matrices and standard matrix algebra operations. However, in this first implementation of the chain approach to Boolean operations, we are satisfied with a solution using simplicial triangulations of input spaces. Future revisions of our algorithm will be based on more general cellular complexes.

1.1 User interface

The API will contain the high-level binary functions `union`, `intersection`, `difference`, and `xor`. Each of them will call the same function `boolOps` and then suitably operates the two returned bit arrays, i.e. the coordinate representations of the input spaces in the merged cell decomposition. The input parameters `lar1` and `lar2` stand for two LAR models, each one constituted by a pair `(V,CV)`, i.e. by the matrix `V` of vertex coordinates and by an integer array `CV` giving the vertex indices of each d-cell.

⟨ High-level boolean operations 2 ⟩ ≡

```
def union(lar1,lar2):
   lar = boolOps(lar1,lar2)
def intersection(lar1,lar2):
   lar = boolOps(lar1,lar2)
def difference(lar1,lar2):
   lar = boolOps(lar1,lar2)
def xor(lar1,lar2):
   lar = boolOps(lar1,lar2)
```
◇

Macro referenced in 13a.

2 Merging 0-cells

The real work is performed by the function `boolOps`, that will procede step-by-step (see Section 4) to the computation of the minimally fragmented common cell complex where to compute the chain resulting from the requested Boolean operation.

2.1 Global 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 double vertex copies (coming from the 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 3a ⟩ ≡

```
""" First step of Boolen Algorithm """
```
⟨ Initial indexing of vertex positions 3b ⟩
⟨ Merge two dictionaries with keys the point locations 3c ⟩
⟨ 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 13a.

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 3b ⟩ ≡
```
    from collections import defaultdict, OrderedDict

    def vertexSieve(model1, model2):
       V1,CV1 = model1; V2,CV2 = 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 3a.

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` ∪ `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 3c ⟩ ≡
```

       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 3a.

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 stye 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 3a.

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 3a.

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 ⟩ ≡

```
        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 3a.

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/boolean2/test02.py" 5b ≡

⟨ Initial import of modules 18a ⟩

⟨ Import the module (5c `boolean2`) 18h ⟩

```
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)))) ]))
V, n1,n2,n12, B1,B2 = boolOps(model1,model2)
# VIEW(SKEL_1(STRUCT(MKPOLS((V, CV_un[:n1]+CV_int )))))
# VIEW(SKEL_1(STRUCT(MKPOLS((V, CV_un[n1-n12:]+CV_int )))))
◇
```

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/boolean2/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 $O(n \ln n)$.

3 Extracting divisor d-cells

It is well-known that, in order to compute any Boolean operation with cell-decomposed arguments, either using a decompositive or a boundary scheme, the intersection of boundaries must be computed [PRS89]. In this section we develop the preparatory work for such a task, aiming to compute in the end the coboundary of boundary of each Boolean argument. We call the d-cells in such two sets, namely

$$\Delta^1 = (\delta_{d-1} \circ \partial_d)(\Lambda^1_d) \quad \text{and} \quad \Delta^2 = (\delta_{d-1} \circ \partial_d)(\Lambda^2_d),$$

as the *divisors* of Λ^2 and Λ^1, respectively.

Coboundary of boundary The function `coboundaryOfBoundaryCells` returns the d-cells of the coboundary of boundary, providing as input the `BRC` representation of the d-cells and the $(d-1)$-cells. In other words, it computes the set of d-cells

$$\Delta = (\delta_{d-1} \circ \partial_d)(\Lambda_d)$$

where `cells` $:=$ `BRC`(Λ_d), and `facets` $:=$ `BRC`(Λ_{d-1}).

⟨ Compute the d-cells in the coboundary of boundary 7a ⟩ ≡
```
def coboundaryOfBoundaryCells(cells,facets):
    csrBoundaryMat = boundary(cells,facets)
    csrChain = totalChain(cells)
    csrBoundaryChain = matrixProduct(csrBoundaryMat, csrChain)
    for k,value in enumerate(csrBoundaryChain.data):
        if value % 2 == 0: csrBoundaryChain.data[k] = 0
    boundaryCells = [k for k,val in enumerate(csrBoundaryChain.data.tolist())
                            if val == 1]
    csrCoboundaryBoundaryChain = matrixProduct(csrBoundaryMat.T, csrBoundaryChain)
    coboundaryBoundaryCells = [k for k in csrCoboundaryBoundaryChain.indices.tolist()]
    return coboundaryBoundaryCells
```
◇
Macro referenced in 13a.

Boundary divisors The actual computation of the two sets of boundary divisors Δ^1 and Δ^2 is performed by the `boundarySuperCells` function below.

⟨ Compute the boundary divisors of both arguments 7b ⟩ ≡
```
""" Second stage of Boolean operations """
def boundarySuperCells( V, CV ):
   FV = larSimplexFacets(CV)
   BSupCells = coboundaryOfBoundaryCells(CV,FV)
   return BSupCells
```
◇
Macro referenced in 13a.

By definition, a *divisor* is a single d-cell and, in the present implementation, it is a d-simplex. `BSupCells` is the array of d-cells in the coboundary of boundary. They become *divisors* of the (other argument's boundary) iff NONE d-star of their vertices does contain the d-cell itself.

⟨ Minimal covering of divisors 8a ⟩ ≡
```
""" Minimal covering chains of divisors """
def minimalCovers(V,divisors,CV1,CV,setCV):
   covers = [selectIncidentChain( V, CV )(v) for cell in divisors for v in CV1[cell] ]
   print "\n covers =",covers
   return covers
```
◇
Macro referenced in 10a.

3.1 Boundary computation

The matrices of the boundary operators of the boolean arguments Λ^1 and Λ^2 are computed here as supported by the novel vertex set $V := V_1 \cup V_2$. Both the characteristic matrices M_d and M_{d-1} are needed to compute a $[\partial_d]$ matrix (see Reference [DPS14]). Hence we start this section by computing the new basis of $(d-1)$-faces FV $:=$ CSR(M_{d-1}), and then compute the two subsets $B^1, B^2 \subset V$ of boundary vertices (upon the joint Delaunay complex V), where

$$B^1 = [\mathcal{VF}^1]\,[\partial_d^1]\,\mathbf{1}, \quad \text{and} \quad B^2 = [\mathcal{VF}^2]\,[\partial_d^2]\,\mathbf{1}.$$

where $[\mathcal{VF}^1]^\top$ = CSR(FV1) and $[\mathcal{VF}^2]^\top$ = CSR(FV2), and where FV1 and FV2 are the relations *face-vertices* computed from the relation CV supported by the *joint* Delaunay vertex set.

Compute the boundary vertices of both arguments The two bases of d-cells, given as input to the boundaryVertices function below, were already renumbered. In other words, their vertices currently belong to the common Delaunay complex. Therefore, the subsequent calls to larSimplexFacets also return two sets of boundary facets, denoted as BF1 and BF2, are supported by the Delaunay complex. BV1 and BV2 contain the boundary vertex indices of the input cells CV1 and CV2.

⟨ Compute boundary vertices of both arguments 8b ⟩ ≡

```
""" Second stage of Boolean operations """
def boundaryVertices( V, CV1,CV2 ):
   FV1 = larSimplexFacets(CV1)
   FV2 = larSimplexFacets(CV2)
   BF1 = boundaryCells(CV1,FV1)
   BF2 = boundaryCells(CV2,FV2)
   BV1 = list(set([ v for f in BF1 for v in FV1[f] ]))
   BV2 = list(set([ v for f in BF2 for v in FV2[f] ]))
   VIEW(STRUCT([
      COLOR(GREEN)(STRUCT(AA(MK)([V[v] for v in BV1]))),
      COLOR(MAGENTA)(STRUCT(AA(MK)([V[v] for v in BV2]))) ]))
   return BV1, BV2
◇
```

Macro referenced in 13a.

4 Splitting argument chains

The boolOps function is the main procedure of the Boolean algorithm. Its steps give an outline of the computations to be performed in sequence. The input LAR representations are first decomposed in their geometric and topological components, and embedded in the

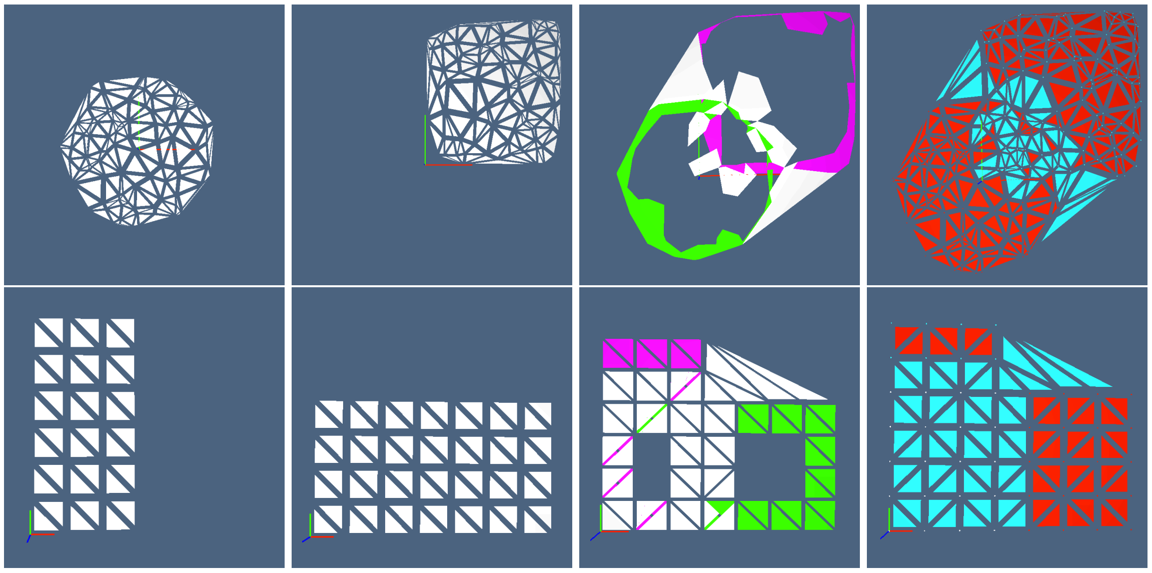

Figure 1: The selection of chains candidate to splitting, in both unstructured and structured meshes.

same space, so obtaining a global renumbering as `V`, `CV1`, `CV2`. Then the global Delaunay's triangulation `CV` and its set representation `setCV` are computed. In due order, the following steps are:

1. the extraction of $\gamma^1, \gamma^2 \subseteq \Lambda_d$, the coboundary of the boundaries of the Boolean arguments:
$$\gamma^1 = (\delta_{d-1} \circ \partial_d)(\Lambda_d^1), \qquad \gamma^2 = (\delta_{d-1} \circ \partial_d)(\Lambda_d^2);$$
2. the selection of the two invariant sub-chains, i.e. of the subsets of Λ_d^1, Λ_d^2 that were not changed by the re-triangulation, and hence continue to contain some facets of the original boundaries:
$$\gamma^{1'} \subseteq \gamma^1, \qquad \gamma^{2'} \subseteq \gamma^2$$

The top-level algorithm

⟨Boolean subdivided complex 10a⟩ ≡

```
""" High level Boolean Application Programming Interface """
⟨Minimal covering chains of divisors ?⟩
def boolOps(lar1,lar2):
    (V1,CV1),(V2,CV2) = lar1,lar2
    n1,n2 = len(V1),len(V2)
    V, CV1, CV2, n12 = vertexSieve(lar1, lar2)
    CV = Delaunay(array(V)).vertices
```

```
        setCV = set([tuple(sorted(cell)) for cell in CV])
        ⟨ Extraction of coboundary of boundary chains 10b ⟩
        ⟨ Computation of invariant supercells 11a ⟩
        ⟨ Invariant subsets of supercells 11b ⟩
        ⟨ Minimal covering of divisors 8a ⟩
        ⟨ Covers of divisors computation 11c ⟩
        return V,n1,n2,n12, B1,B2
    ◇
```

Macro referenced in 13a.

Extraction of coboundary of boundary chains After the embedding of the two topologies in the same coordinate space and the Delaunay's re-triangulation, the two γ^1, γ^2 d-chains are computed by the following macro, and stored in the `BSupCells1` and `BSupCells2` variables.

⟨ Extraction of coboundary of boundary chains 10b ⟩ ≡

```
    # Second stage of Boolean algorithm
    B1,B2 = boundaryVertices( V, CV1, CV2 )
    # Extraction of coboundary of boundary chains
    BSupCells1 = boundarySuperCells( V, CV1 )
    BSupCells2 = boundarySuperCells( V, CV2 )

    VIEW(STRUCT([
        COLOR(GREEN)(STRUCT(MKPOLS(((V,[CV1[c] for c in BSupCells1]))))),
        COLOR(MAGENTA)(STRUCT(MKPOLS(((V,[CV2[c] for c in BSupCells2])))))
    ]))
    ◇
```

Macro referenced in 10a.

Invariant subsets of supercells Then the sub-chains $\gamma^{1'}, \gamma^{2'}$ (those not changed by the Boolean merging) are computed by the `invariantSuperCells` function, and stored in `invSupCells1` and `invSupCells2` variables, in order to compute, by chain difference, the `divisor1` and `divisor2` d-chains.

⟨ Computation of invariant supercells 11a ⟩ ≡

```
    def invariantSuperCells(V,BSupCells1,CV1,setCV):
        out = []
        for cell in BSupCells1:
            if tuple(CV1[cell]) in setCV: out += [cell]
        return out
    ◇
```

Macro referenced in 10a.

⟨ Invariant subsets of supercells 11b ⟩ ≡

```
# Invariant subsets of supercells
invSupCells1 = invariantSuperCells(V,BSupCells1,CV1,setCV)
invSupCells2 = invariantSuperCells(V,BSupCells2,CV2,setCV)

divisor1 = set(BSupCells1).difference(invSupCells1)
divisor2 = set(BSupCells2).difference(invSupCells2)
VIEW(STRUCT([
   COLOR(GREEN)(STRUCT(MKPOLS(((V,[CV1[c] for c in divisor1]))))),
   COLOR(MAGENTA)(STRUCT(MKPOLS(((V,[CV2[c] for c in divisor2])))))
]))
◇
```

Macro referenced in 10a.

Covers of divisors computation

⟨ Covers of divisors computation 11c ⟩ ≡

```
# Covers of divisors computation
covers1 = minimalCovers(V,divisor1,CV1,CV,setCV)
covers2 = minimalCovers(V,divisor2,CV2,CV,setCV)
VIEW(STRUCT([
   EXPLODE(1.2,1.2,1)(MKPOLS(((V,CV)))),
   COLOR(GREEN)(EXPLODE(1.2,1.2,1)(MKPOLS(((V,[CV[c] for c in CAT(covers1)]))))),
   COLOR(MAGENTA)(EXPLODE(1.2,1.2,1)(MKPOLS(((V,[CV[c] for c in CAT(covers2)])))))
    ]))
◇
```

Macro referenced in 10a.

4.1 Matching cells in $\Sigma_\cap$ with spanning chains in Λ^1, Λ^2

The next step of the *Boolean Chains* algorithm retrieves and compare the chains of $d$-cells incident on $\mathtt{V}(\partial_d(\Lambda^1)), \mathtt{V}(\partial_d(\Lambda^2))$ and on $\mathtt{V}(\partial_d(\Sigma^\cap))$. A filtering step working in $O(n)$ is performed in regard to this.

Filtering cells incident on a subset of vertices The selection of the $d$-chain incident on a 0-chain (subset of vertices) can be executed either in time roughly proportional to the output, via a SpMV multiplication of the $\mathtt{CV} := \mathtt{CSR}(M_d)$ matrix times the (binary) coordinate representation of the 0-chain, or in time proportional to the input, via a more traditional select filtering of cells in a CV table, with respect to the subset of vertices.

⟨ Select cells incident on vertices 12 ⟩ ≡

```
""" Select the $d$-chain incident on a $0$-chain """
def selectIncidentChain( V, CV ):
   def selectIncidentChain0( v ):
```

```
            return [k for k in range(len(CV)) if v in CV[k] ]
        return selectIncidentChain0
```
◇

Macro referenced in 13a.

4.2 Splitting cells

4.3 Keeping cell dictionaries updated

5 Boolean outputs computations

6 Export the boolean module

The `boolean.py` module is exported to the library `lar-cc/lib`. Therefore many of the macros developed in this module are expanded and written to an external file.

`"lib/py/boolean2.py"` 13a ≡

```
""" Module with Boolean operators using chains and CSR matrices """
```
⟨ Initial import of modules 18a ⟩
⟨ Symbolic utility to represent points as strings 19 ⟩
⟨ Affine transformations of d-points 18i ⟩
⟨ Generation of n random points in the unit d-disk 13b ⟩
⟨ Generation of n random points in the standard d-cuboid 14a ⟩
⟨ Triangulation of random points 14b ⟩
⟨ Boolean subdivided complex 10a ⟩
⟨ High-level boolean operations 2 ⟩
⟨ Place the vertices of Boolean arguments in a common space 3a ⟩
⟨ Show vertices of arguments ? ⟩
⟨ Compute boundary vertices of both arguments 8b ⟩
⟨ Select cells incident on vertices 12 ⟩
⟨ Compute the d-cells in the coboundary of boundary 7a ⟩
⟨ Compute the boundary divisors of both arguments 7b ⟩
⟨ Visualization of a subset of cells 17c ⟩
◇

7 Tests

7.1 Generation of random data

We found useful to drive the development of new modules using randomly generated data, so that every upcoming execution of the developed algorithms is naturally driven to be challenged by different data.

7.1.1 Testing the main algorithm

Write the test executable file

7.1.2 Lowest-level space generation procedures

Random points in unit disk First we generate a set of n random points in the unit D^d disk centred on the origin, to be subsequently used to generate a random Delaunay complex of variable granularity.

⟨ Generation of n random points in the unit d-disk 13b ⟩ ≡

```
def randomPointsInUnitCircle(n=200,d=2, r=1):
   points = random.random((n,d)) * ([2*math.pi]+[1]*(d-1))
   return [[SQRT(p[1])*COS(p[0]),SQRT(p[1])*SIN(p[0])] for p in points]
   ## TODO: correct for $d$-sphere

if __name__=="__main__":
   VIEW(STRUCT(AA(MK)(randomPointsInUnitCircle())))
◇
```

Macro referenced in 13a.

Random points in the standard d-cuboid A set of n random d-points is then generated within the standard d-cuboid, i.e. withing the d-dimensional interval with a vertex on the origin.

⟨ Generation of n random points in the standard d-cuboid 14a ⟩ ≡

```
def randomPointsInUnitCuboid(n=200,d=2):
   return random.random((n,d)).tolist()

if __name__=="__main__":
   VIEW(STRUCT(AA(MK)(randomPointsInUnitCuboid())))
◇
```

Macro referenced in 13a.

Triangulation of random points The Delaunay triangulation of `randomPointsInUnitCircle` is generated by the following macro.

⟨ Triangulation of random points 14b ⟩ ≡

```
from scipy.spatial import Delaunay
def randomTriangulation(n=200,d=2,out='disk'):
   if out == 'disk':
      V = randomPointsInUnitCircle(n,d)
   elif out == 'cuboid':
      V = randomPointsInUnitCuboid(n,d)
   CV = Delaunay(array(V)).vertices
   model = V,CV
   return model
```

```
if __name__=="__main__":
   from lar2psm import *
   VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(model)))
◇
```

Macro referenced in 13a.

7.2 Disk saving of test data

7.3 Algorithm execution

7.4 Unit tests

7.4.1 First Boolean stage

Some unit tests of the first Boolean stage are discussed in the following. They are mainly aimed to check a correct execution of the filtering of common vertices with renumbering of the union set of vertices, and to the consequential redefinition of the *d*-cell basis.

Union of 2D non-structured grids

"test/py/boolean2/test01.py" 15a ≡

```
""" Union of 2D non-structured grids """
⟨ Initial import of modules 18a ⟩
⟨ Import the module (15b boolean2 ) 18h ⟩
⟨ Import the module (15c lar2psm ) 18h ⟩
⟨ Import the module (15d myfont ) 18h ⟩
model1 = randomTriangulation(1000,2,'disk')
V1,CV1 = model1
VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(model1)+cellNames(model1,CV1,MAGENTA)))
model2 = randomTriangulation(1000,2,'cuboid')
V2,CV2 = model2
V2 = larScale( [2,2])(V2)
model2 = V2,CV2
VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(model2)+cellNames(model2,CV2,RED)))
V, n1,n2,n12, B1,B2 = boolOps(model1,model2)
⟨ Visualization of first Boolean step 17d ⟩
◇
```

Union of 3D non-structured grids

```
"test/py/boolean2/test05.py" 15e ≡
      """ Union of 3D non-structured grids """
      ⟨ Initial import of modules 18a ⟩
      ⟨ Import the module (15f boolean2 ) 18h ⟩
      ⟨ Import the module (15g lar2psm ) 18h ⟩
      ⟨ Import the module (15h myfont ) 18h ⟩
      model1 = randomTriangulation(100,3,'cuboid')
      V1,CV1 = model1
      V1 = larScale( [2,2,2])(V1)
      V1 = larTranslate( [-1,-1,-1])(V1)
      model1 = V1,CV1
      VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model1)+cellNames(model1,CV1,MAGENTA)))
      model2 = randomTriangulation(100,3,'cuboid')
      V2,CV2 = model2
      V2 = larScale( [2,2,2])(V2)
      model2 = V2,CV2
      VIEW(EXPLODE(1.5,1.5,1.5)(MKPOLS(model2)+cellNames(model2,CV2,RED)))
      V, n1,n2,n12, B1,B2 = boolOps(model1,model2)
      ⟨ Visualization of first Boolean step 17d ⟩
      ◇
```

Union of structured grids

```
"test/py/boolean2/test04.py" 16a ≡
      """ test program for the boolean module """
      ⟨ Initial import of modules 18a ⟩
      ⟨ Import the module (16b boolean2 ) 18h ⟩
      blue = larSimplexGrid([30,60])
      V2,CV2 = larSimplexGrid([70,40])
      V2 = larTranslate( [.5,.5])(V2)
      red = V2,CV2
      VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(blue) ))
      VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(red) ))
      V, CV1, CV2, n12 = vertexSieve(red,blue)
      V, n1,n2,n12, B1,B2 = boolOps(red,blue)
      CV = Delaunay(array(V)).vertices
      ⟨ Visualization of first Boolean step 17d ⟩
      ◇
```

model = checkModel(larHollowCyl(0.8,1,1,angle=PI/4)([12,2,2])) VIEW(STRUCT(MKPOLS(model))) model = checkModel(larHollowSphere(0.8,1,PI/6,PI/4)([6,12,2]))

Union of structured grids

"test/py/boolean2/test06.py" 16c ≡
```
""" test program for the boolean module """
⟨Initial import of modules 18a⟩
from boolean2 import *
from mapper import *
blue = larHollowCyl(0.8,1,1,angle=PI/4)([6,2,5])
red = larHollowSphere(0.8,1,PI/6,PI/4)([6,12,2])
VIEW(EXPLODE(1.5,1.5,1)(MKPOLS(blue) + MKPOLS(red) ))
V, CV1, CV2, n12 = vertexSieve(red,blue)
V, n1,n2,n12, B1,B2 = boolOps(red,blue)
CV = Delaunay(array(V)).vertices
⟨Visualization of first Boolean step 17d⟩
◇
```

Boolean operations with general LAR cells

"test/py/boolean2/test03.py" 17a ≡
```
""" test example with general LAR cells for the boolean module """
⟨Initial import of modules 18a⟩
⟨Import the module (17b boolean2 ) 18h⟩
V1 = [[0,5],[6,5],[0,2],[3,2],[6,2],[0,0],[3,0],[3,-2],[6,-2]]
CV1 = [[2,3,5,6],[0,1,2,3,4],[3,4,6,7,8]]
blue = V1,CV1
V2 = [[3,6],[7,6],[0,5],[3,5],[3,4],[7,4],[3,2],[7,2],[0,0],[3,0],[6,0],[6,2]]
CV2 = [[0,1,3,4,5],[2,3,4,6,8,9],[6,9,10,11],[4,5,6,7,11]]
red = V2,CV2
V, n1,n2,n12, B1,B2 = boolOps(red,blue)
CV = Delaunay(array(V)).vertices
⟨Visualization of first Boolean step 17d⟩
◇
```

7.5 Visualization of the Boolean algorithm

Display of colored cell numbers

⟨Visualization of a subset of cells 17c⟩ ≡
```
def cellNames(model,cells, color=BLACK):
   V,CV= model
   print "\n CV =",CV
   print "\n cells =",cells
   texts = []
   for k,cell in enumerate(cells):
      centroid = CCOMB([V[v] for v in cell])
      print "centroid =",centroid
      d = len(centroid)
      texts += [T(range(1,d+1))(centroid)(S(range(1,d+1))([0.005 for k in range(d)])(TEXT(str(
   return AA(COLOR(color))(texts)
◇
```

Macro referenced in 13a.

First step of visualization

⟨ Visualization of first Boolean step 17d ⟩ ≡

```
""" Visualization of first Boolean step  """
if n12==0:
   hpc0 = STRUCT([ COLOR(RED)(EXPLODE(1.5,1.5,1)(AA(MK)(V[:n1-n12]) )),
            COLOR(CYAN)(EXPLODE(1.5,1.5,1)(AA(MK)(V[n1:]) )) ])
else:
   hpc0 = STRUCT([ COLOR(RED)(EXPLODE(1.5,1.5,1)(AA(MK)(V[:n1-n12]) )),
            COLOR(CYAN)(EXPLODE(1.5,1.5,1)(AA(MK)(V[n1:]) )),
            COLOR(WHITE)(EXPLODE(1.5,1.5,1)(AA(MK)(V[n1-n12:n1]) )) ])

# hpc1 = COLOR(RED)(EXPLODE(1.5,1.5,1)(MKPOLS((V,CV_un)) ))
# hpc2 = COLOR(CYAN)(EXPLODE(1.5,1.5,1)(MKPOLS((V,CV_int)) ))
# VIEW(STRUCT([hpc0, hpc1, hpc2]))
◇
```

Macro referenced in 15ae, 16ac, 17a.

A Utility functions

⟨ Initial import of modules 18a ⟩ ≡

```
from pyplasm import *
from scipy import *
import os,sys

""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
⟨ Import the module (18b lar2psm ) 18h ⟩
⟨ Import the module (18c simplexn ) 18h ⟩
⟨ Import the module (18d larcc ) 18h ⟩
⟨ Import the module (18e largrid ) 18h ⟩
⟨ Import the module (18f myfont ) 18h ⟩
⟨ Import the module (18g mapper ) 18h ⟩
◇
```

Macro referenced in 5b, 13a, 15ae, 16ac, 17a.

⟨ Import the module 18h ⟩ ≡

```
import @1
from @1 import *
◇
```

Macro referenced in 5b, 15ae, 16a, 17a, 18a.

Affine transformations of points Some primitive maps of points to points are given in the following, including translation, rotation and scaling of array of points via direct transformation of their coordinates.

⟨ Affine transformations of *d*-points 18i ⟩ ≡

```
def translatePoints (points, tvect):
   return [VECTSUM([p,tvect]) for p in points]

def rotatePoints (points, angle):      # 2-dimensional !! TODO: n-dim
   a = angle
   return [[x*COS(a)-y*SIN(a), x*SIN(a)+y*COS(a)] for x,y in points]

def scalePoints (points, svect):
   return [AA(PROD)(TRANS([p,svect])) for p in points]
```
◇

Macro referenced in 13a.

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 19 ⟩ ≡

```
""" TODO: use package Decimal (http://docs.python.org/2/library/decimal.html) """
ROUND_ZERO = 1E-07
def round_or_zero (x,prec=7):
   """
   Decision procedure to approximate a small number to zero.
   Return either the input number or zero.
   """
   def myround(x):
      return eval(('%.'+str(prec)+'f') % round(x,prec))
   xx = myround(x)
   if abs(xx) < ROUND_ZERO: return 0.0
   else: return xx

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

def fixedPrec(value):
   if abs(value - int(value))<ROUND_ZERO: value = int(value)
   out = ('%0.7f'% value).rstrip('0')
   if out == '-0.': out = '0.'
   return 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 13a.

References

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

[DPS14] Antonio Dicarlo, Alberto Paoluzzi, and Vadim Shapiro, *Linear algebraic representation for topological structures*, Comput. Aided Des. **46** (2014), 269–274.

[PRS89] A. Paoluzzi, M. Ramella, and A Santarelli, *Boolean algebra over linear polyhedra*, Comput. Aided Des. **21** (1989), no. 10, 474–484.