Boolean combinations of cellular complexes as chain operations *

Alberto Paoluzzi

September 11, 2014

Contents

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.

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. September 11, 2014

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

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

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

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

1.1 Preview of the Boolean algorithm

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

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

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

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

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

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

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

1.2 Remarks

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

The more complex algorithmic step is the cell splitting. Every time, a single d-cell c is split by a single hyperplane (cutting its interior) giving either two splitted cells c_1 and c_2, or just one output cell (if the hyperplane is the affine hull of the CDC facet) whatever the

input cell dimension d. After every splitting of the cell interior, the row c is substituted (within the `CV` matrix) by c_1, and c_2 is added to the end of the `CV` matrix, as a new row.

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

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

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

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

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

2 Step 1: merging discrete spaces

2.1 Requirements

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

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

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

Auxiliary data structures This software module returns also:

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

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

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

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

3. the input boundary complex `(V,BV1+BV2)`, i.e. the two $(d-1)$-complexes `(V,BV1)` and `(V,BV2)`, defined on the common vertices.

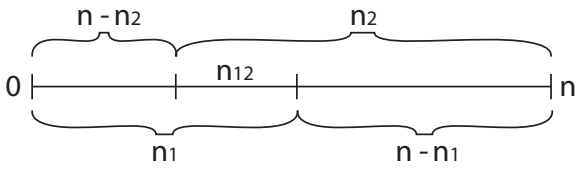

Figure 1: Relationships inside the orderings of CDC vertices

2.2 Implementation

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

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

   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)

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

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

```
        else: case2 += [item]
    n1 = len(case1); n2 = len(case12); n3 = len(case2)

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

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

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

◇

Macro referenced in 6a.

⟨ Make Common Delaunay Complex 5 ⟩ ≡

```
"""  Make Common Delaunay Complex """
def makeCDC(model1, model2):
    V, _,_, n1,n12,n2 = mergeVertices(model1, model2)
    n = len(V)
    assert n == n1 - n12 + n2

    CV = sorted(AA(sorted)(Delaunay(array(V)).simplices))
    vertDict = defaultdict(list)
    for k,v in enumerate(V): vertDict[vcode(v)] += [k]

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

◇

Macro referenced in 6a.

3 Step 2: splitting cells

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

Input The output of previous algorithm stage.

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

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

4 Step 3: cell labeling

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

Input The output of previous algorithm stage.

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

5 Step 1: greedy cell gathering

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

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

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

6 Exporting the library

"lib/py/bool1.py" 6a ≡

```
""" Module for Boolean ops with LAR """
⟨ Initial import of modules 10b ⟩
DEBUG = False
⟨ Merge two dictionaries with keys the point locations 4 ⟩
⟨ Make Common Delaunay Complex 5 ⟩
◇
```

7 Tests and examples

⟨ Debug input and vertex merging 6b ⟩ ≡

```
    if DEBUG: VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2))))

    model1,model2 = (V1,FV1),(V2,FV2)
    V, CV1,CV2, n1,n12,n2 = mergeVertices(model1,model2)  #<<<<<<<<<<<<<<<<

    print "\nV =", V
    print "\nCV1,CV2 =", CV1,CV2
    print "\nn1,n12,n2 =", n1,n12,n2

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

    V,CV,vertDict,n1,n12,n2 = makeCDC(model1, model2)     #<<<<<<<<<<<<<<<<

    print "\nCV =", CV
    print "\nvertDict =", vertDict

    submodel = SKEL_1(STRUCT(MKPOLS((V,CV))))
    VIEW(STRUCT([ submodel,larModelNumbering(V,[VV,_,CV],submodel,4)]))
    ◇
```

Macro referenced in 7ab, 8abc, 9ab, 10a.

"test/py/bool1/test1.py" 7a ≡

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

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

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

    ⟨ Debug input and vertex merging 6b ⟩
    ◇
```

"test/py/bool1/test2.py" 7b ≡

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

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

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

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

⟨ Debug input and vertex merging 6b ⟩
◇
```

"test/py/bool1/test3.py" 8a ≡

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

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

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

⟨ Debug input and vertex merging 6b ⟩
◇
```

"test/py/bool1/test3a.py" 8b ≡

```
import sys
""" import modules from larcc/lib """
```

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

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

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

⟨ Debug input and vertex merging 6b ⟩
◇
```

"test/py/bool1/test4.py" 8c ≡

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

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

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

⟨ Debug input and vertex merging 6b ⟩
◇
```

"test/py/bool1/test5.py" 9a ≡

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


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

V2 = [[2.5,2.5],[7.5,2.5],[7.5,7.5],[2.5,7.5]]
```

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

⟨ Debug input and vertex merging 6b ⟩

◇

"test/py/bool1/test6.py" 9b ≡

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


n = 24
V1 = [[5*cos(angle*2*PI/n)+2.5, 5*sin(angle*2*PI/n)+2.5] for angle in range(n)]
FV1 = [range(n)]
EV1 = TRANS([range(n),range(1,n+1)]); EV1[-1] = [0,n-1]
VV1 = AA(LIST)(range(len(V1)))

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

⟨ Debug input and vertex merging 6b ⟩

◇

"test/py/bool1/test7.py" 10a ≡

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

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

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

⟨ Debug input and vertex merging 6b ⟩

◇

A Appendix: utility functions

⟨ Initial import of modules 10b ⟩ ≡

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

◇

Macro referenced in 6a.

A.1 Numeric utilities

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

⟨ Symbolic utility to represent points as strings 11 ⟩ ≡

```
""" TODO: use package Decimal (http://docs.python.org/2/library/decimal.html) """
global PRECISION
PRECISION = 4.95

def verySmall(number): return abs(number) < 10**-(PRECISION/1.15)

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

def fixedPrec(value):
   out = round(value*10**(PRECISION*1.1))/10**(PRECISION*1.1)
   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 never referenced.

References

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