

# Boolean combinations of cellular complexes as chain operations \*

Alberto Paoluzzi

September 10, 2014

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Preview of the Boolean algorithm . . . . .	1
1.2	Remarks . . . . .	2
<b>2</b>	<b>Merging discrete spaces</b>	<b>3</b>
<b>3</b>	<b>Splitting cells</b>	<b>3</b>
<b>4</b>	<b>Cell labeling</b>	<b>3</b>
<b>5</b>	<b>Greedy cell gathering</b>	<b>3</b>
<b>6</b>	<b>Exporting the library</b>	<b>3</b>
<b>7</b>	<b>Tests and examples</b>	<b>3</b>
<b>A</b>	<b>Appendix: utility functions</b>	<b>5</b>
A.1	Numeric utilities . . . . .	5

## 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 [?].  
September 10, 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  $\partial A$  and  $\partial B$  in it.
3. Split the (highest-dimensional) cells of CDC crossed by  $\partial A$  or  $\partial B$ . Their lower dimensional faces remain partitioned accordingly. We name the resulting complex SCDC (Split Common Delaunay Complex).
4. With respect to the SCDC basis of  $d$ -cells  $C_d$ , compute two coordinate chains  $\alpha, \beta : C_d \rightarrow \{0, 1\}$ , such that:

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

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  $\mathbf{CV}$  matrix) by  $c_1$ , and  $c_2$  is added to the end of the  $\mathbf{CV}$  matrix, as a new row.

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

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

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

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

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

## 2 Merging discrete spaces

**Input** Two LAR models of “solid”  $d$ -spaces  $A$  and  $B$ , denoted as  $(\mathbf{V1}, \mathbf{CV1})$  and  $(\mathbf{V2}, \mathbf{CV2})$ .

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.

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

## 3 Splitting cells

**Input**

**Output**

## 4 Cell labeling

**Input**

## Output

## 5 Greedy cell gathering

### Input

### Output

## 6 Exporting the library

```
@O lib/py/bool1.py @""" Module for Boolean ops with LAR """ @i Initial import of
modules @i DEBUG = True @
```

## 7 Tests and examples

```
@O test/py/bool1/test1.py @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]]
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)))
VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @
@O test/py/bool1/test2.py @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]]
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)))
VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @
@O test/py/bool1/test3.py @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)))
```

```

VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @
@O test/py/bool1/test4.py @ 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]
VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @
@O test/py/bool1/test5.py @ 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)))
VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @
@O test/py/bool1/test6.py @ 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)))
VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @
@O test/py/bool1/test7.py @ 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],[14,8],[14,13],[8,13]]
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],[13,14],[14,15],[12,15]] VV2 = AA(LIST)(range(len(V2)))
VIEW(STRUCT(MKPOLS((V1,EV1)) + MKPOLS((V2,EV2)))) @

```

## A Appendix: utility functions

```

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

```

## A.1 Numeric utilities

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

```
@D Symbolic utility to represent points as strings @""" TODO: use package Decimal
(http://docs.python.org/2/library/decimal.html) """ global PRECISION PRECISION =
4.95
```

```
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)) @
```