Boolean combination of cellular complexes [*]

Alberto Paoluzzi

July 4, 2014

Contents

1 Introduction

2 Merging arguments

Building a covering of common convex hull

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. July 4, 2014

⟨ Building a covering of common convex hull 1a ⟩ ≡

```
def covering(model1,model2):
   V, CV1, CV2, n12 = vertexSieve(model1,model2)
   _,EEV1 = larFacets((V,CV1),dim=2,emptyCellNumber=1)
   _,EEV2 = larFacets((V,CV2),dim=2,emptyCellNumber=1)
   CV1 = CV1[:-1]
   CV2 = CV2[:-1]
   VV = AA(LIST)(range(len(V)))
   return V,[VV,EEV1,EEV2,CV1,CV2],n12
```
◇

Macro referenced in 6a.

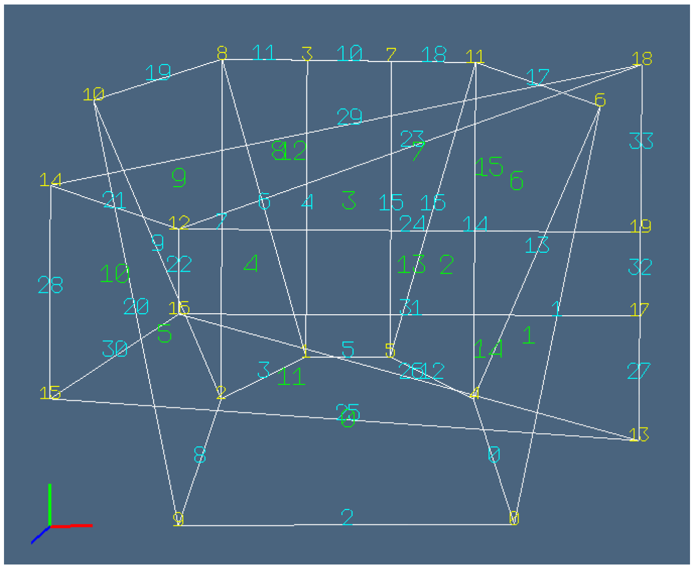

Figure 1: Set covering of the two Boolean arguments.

Building a partition of common convex hull

⟨ Building a partition of common convex hull of vertices 1b ⟩ ≡

```
def partition(V, CV1,CV2, EEV1,EEV2):
   CV = sorted(AA(sorted)(Delaunay(array(V)).vertices))
   BV1, BV2, BF1, BF2 = boundaryVertices( V, CV1,CV2, 'cuboid', EEV1,EEV2 )
   BV = BV1+BV2
   nE1 = len(EEV1)
   BF = BF1+[e+nE1 for e in BF2]
   return CV, BV1, BV2, BF1, BF2, BV, BF, nE1
```
◇

Macro referenced in 6a.

3 Selecting cells to split

Relational inversion (Characteristic matrix transposition)

⟨ Characteristic matrix transposition 2a ⟩ ≡

```
""" 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 6a.

⟨ Look for cells in Delaunay, with vertices in both operands 2b ⟩ ≡

```
""" Look for cells in Delaunay, with vertices in both operands """
def mixedCells(CV,n0,n1,m0,m1):
   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 6a.

⟨ Look for cells in cells12, with vertices on boundaries 3a ⟩ ≡

```
""" Look for cells in cells12, with vertices on boundaries """
def mixedCellsOnBoundaries(cells12,BV1,BV2):
   cells12BV1 = [cell for cell in cells12
                if len(list(set(cell).intersection(BV1))) != 0]
   cells12BV2 = [cell for cell in cells12
                if len(list(set(cell).intersection(BV2))) != 0]
   pivots = sorted(AA(sorted)(cells12BV1+cells12BV2))
   pivots = [cell for k,cell in enumerate(pivots[:-1]) if cell==pivots[k+1]]
   return pivots
◇
```

Macro referenced in 6a.

⟨ Build intersection tasks 3b ⟩ ≡

```
""" Build intersection tasks """
def cuttingTest(cuttingHyperplane,polytope,V):
   signs = [INNERPROD([cuttingHyperplane, V[v]+[1.]]) for v in polytope]
   return any([value<0 for value in signs]) and any([value>0 for value in signs])

def splittingTasks(V,pivots,BV,BF,VC,CV,EEV,VE):
   tasks = []
   for pivotCell in pivots:
      cutVerts = [v for v in pivotCell if v in BV]
      for v in cutVerts:
         cutFacets = [e for e in VE[v] if e in BF]
         cells2cut = VC[v]
         for facet,cell2cut in CART([cutFacets,cells2cut]):
            polytope = CV[cell2cut]
            points = [V[w] for w in EEV[facet]]
            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 += [[facet,cell2cut,cuttingHyperplane]]
   tasks = AA(eval)(set(AA(str)(tasks)))
   tasks = TrivialIntersection(tasks,V,EEV,CV)
   print "\ntasks =",tasks
   return tasks
◇
```

Macro referenced in 6a.

facet-cell trivial intersection filtering A final filtering is applied to the pairs (`cuttingHyperplane,polyt`
in the `tasks` array, in order to remove those pairs 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 leaf 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.

⟨ Trivial intersection filtering 4 ⟩ ≡

```
""" 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]).I
      vects = (transformMat * mat([VECTDIFF([V[v],V[v0]]) for v in faceVerts
               if v != v0]).T).tolist()
      if any([all([x>0 for x in list(vect)]) for vect in vects]): out += [[face,cell,affineHul
   return out
◇
```

Macro referenced in 6a.

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.

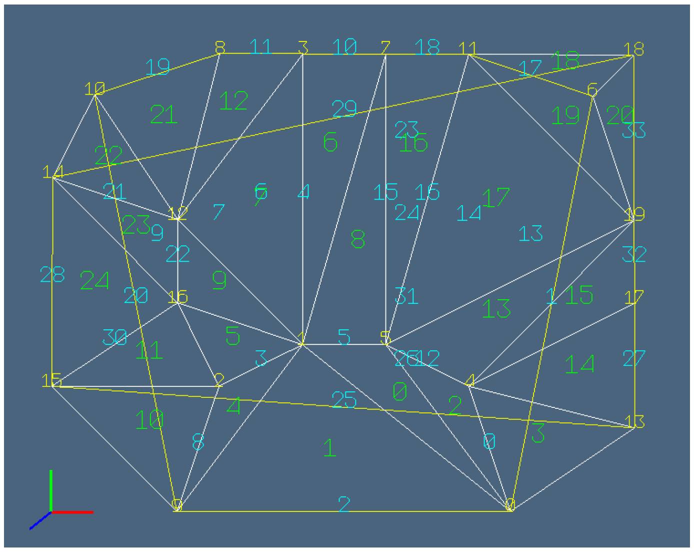

Figure 2: example caption

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 5 ⟩ ≡

```
""" 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])
   print "\ncell =",UKPOL(cellHpc)[0]
   # cell1 = INTERSECTION([cellHpc,rototranslSubspace])

   tolerance=0.0001
   use_octree=False
   cell1 = Plasm.boolop(BOOL_CODE_AND,
      [cellHpc,rototranslSubspace],tolerance,plasm_config.maxnumtry(),use_octree)

   # cell2 = DIFFERENCE([cellHpc,rototranslSubspace])

   cell2 = Plasm.boolop(BOOL_CODE_DIFF,
      [cellHpc,rototranslSubspace],tolerance,plasm_config.maxnumtry(),use_octree)

   return cell1,cell2
```
◇

Macro referenced in 6a.

5 Reconstruction of results

6 Exporting the library

"lib/py/bool.py" 6a ≡

```
""" Module for Boolean ops with LAR """
from matrix import *
```

```
⟨Initial import of modules 9a⟩
⟨Symbolic utility to represent points as strings 9b⟩
⟨Building a covering of common convex hull 1a⟩
⟨Building a partition of common convex hull of vertices 1b⟩
⟨Characteristic matrix transposition 2a⟩
⟨Look for cells in Delaunay, with vertices in both operands 2b⟩
⟨Look for cells in cells12, with vertices on boundaries 3a⟩
⟨Build intersection tasks 3b⟩
⟨Trivial intersection filtering 4⟩
⟨Cell splitting 5⟩
◇
```

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 6b⟩ ≡

```
""" 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], range(8)]
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], range(6)]
◇
```

Macro referenced in 8a.

Input and visualisation of Boolean arguments

⟨Computation of lower-dimensional cells 7a⟩ ≡

```
""" Computation of edges an input visualisation """
model1 = V1,FV1
model2 = V2,FV2
submodel = SKEL_1(STRUCT(MKPOLS(model1)+MKPOLS(model2)))
VV1 = AA(LIST)(range(len(V1)))
_,EV1 = larFacets((V1,FV1),dim=2,emptyCellNumber=1)
VV2 = AA(LIST)(range(len(V2)))
_,EV2 = larFacets((V2,FV2),dim=2,emptyCellNumber=1)
VIEW(larModelNumbering(V1,[VV1,EV1,FV1],submodel,4))
VIEW(larModelNumbering(V2,[VV2,EV2,FV2],submodel,4))
◇
```

Macro referenced in 7b.

Exporting test file

⟨ Bulk of Boolean task computation 7b ⟩ ≡

```
""" Bulk of Boolean task computation """
⟨ Computation of lower-dimensional cells 7a ⟩
V,[VV,EEV1,EEV2,CV1,CV2],n12 = covering(model1,model2)
CCV = CV1+CV2
EEV = EEV1+EEV2
VIEW(larModelNumbering(V,[VV,EEV,CCV],submodel,4))

CV, BV1, BV2, BF1, BF2, BV, BF, nE1 = partition(V, CV1,CV2, EEV1,EEV2)
boundaries = COLOR(YELLOW)(SKEL_1(STRUCT(MKPOLS((V,[EEV[e] for e in BF])))))
submodel = STRUCT([ SKEL_1(STRUCT(MKPOLS((V,CV)))), boundaries ])
VIEW(larModelNumbering(V,[VV,EEV,CV],submodel,4))
⟨ Inversion of incidences 8b ⟩

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)
VE = [VEE1[v]+VEE2[v] for v in range(len(V))]
cells12 = mixedCells(CV,n0,n1,m0,m1)
pivots = mixedCellsOnBoundaries(cells12,BV1,BV2)
tasks = splittingTasks(V,pivots,BV,BF,VC,CV,EEV,VE)
print "\ntasks (facet,cell2cut) =",tasks

out = []
for task in tasks:
   face,cell,covector = task
   cell1,cell2 = cellSplitting(face,cell,covector,V,EEV,CV)
   verts,cells,pols = UKPOL(cell1)
   print "\n cell1 =",AA(vcode)(verts), cells, pols
   verts,cells,pols = UKPOL(cell2)
   print " cell2 =",AA(vcode)(verts), cells, pols
   out += [ COLOR(GREEN)(cell2), COLOR(CYAN)(cell1) ]

VIEW(STRUCT([ STRUCT(out), submodel ]))
◇
```

Macro referenced in 8a.

"test/py/bool/test01.py" 8a ≡

```
import sys
""" import modules from larcc/lib """
sys.path.insert(0, 'lib/py/')
from bool import *
⟨ First set of 2D data: Fork-0 input 6b ⟩
⟨ Bulk of Boolean task computation 7b ⟩
◇
```

association of cells and boundaries

⟨Inversion of incidences 8b⟩ ≡

```
""" Inversion of incidences """
VC = invertRelation(V,CV)
VC1 = invertRelation(V,CV1)
VC2 = invertRelation(V,CV2)
VEE1 = invertRelation(V,EEV1)
VEE2 = [[e+nE1  for e in vE] for vE in invertRelation(V,EEV2)]
submodel = SKEL_1(STRUCT(MKPOLS((V,CV1+CV2))))
VE = [VEE1[v]+VEE2[v] for v in range(len(V))]
◇
```

Macro referenced in 7b.

A Appendix: utility functions

⟨Initial import of modules 9a⟩ ≡

```
from pyplasm import *
from scipy import *
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 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 9b⟩ ≡

```

""" TODO: use package Decimal (http://docs.python.org/2/library/decimal.html) """
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 6a.

References

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