Boolean chains *

Alberto Paoluzzi

February 20, 2015

Abstract

A novel algorithm for computation of Boolean operations between cellular complexes is given in this module. It is based on bucketing of possibly interacting geometry using a box-extension of kd-trees, normally used for point proximity queries. Such kd-tree representation of containment boxes of cells, allow us to compute a number of independent buckets of data, with some elements possibly replicated, to be used for brute force intersection, followed by elimination of duplicated data. We use a fast algorithm [BP96] for splitting a cell with an hyperplane, and append the splitted sub-cells to their buckets, in order to apply all the splits induced by the boundary cells contained in the same bucket. A final tagging of cells as either belonging or not to each operand follows, allowing for fast extraction of Boolean results between any pair of chains (subsets of cells). This Boolean algorithm can be considered of a *Map-Reduce* kind, and hence suitable of a distributed implementation over big datasets. The actual engineered implementation will follow the present prototype, using some distributed NoSQL database, like MongoDB or Riak.

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. February 20, 2015

1 Introduction

2 Preview of the algorithm

The whole Boolean algorithm is composed by four stages in sequence, denoted in the following as *Unification*, *Bucketing*, *Intersection*, and *Reconstruction*. The algorithm described here is both multidimensional and variadic. Multidimensional means that the arguments are solid in Euclidean space of dimension d, with d small integer. The *arity* of a function or operation is the number of arguments or operands the function or operation accepts. In computer science, a function accepting a variable number of arguments is called *variadic*.

2.1 Unification

In this first step the boundaries of the n Boolean arguments are computed and merged together as a set of chains defined in the discrete set `V` made by the union of their vertices, and possibly by a set of random points.

Random points are only added when the L_1 norm (Manhattan or taxicab norm) of $(d-1)$-dimensional boundary cells is greater than a given fraction of the bounding box $BB(< args >)$ of the vertices of arguments.

The Delaunay triangulation `CV` is hence computed, as well as its $(d-1)$ skeleton `FV`, providing the `LAR` model `V,CV` of an initial *partition* of the space.

It is important to notice here that the cell decompositions of the n arguments, and in particula their boundaries, have vertices belonging to the same discrete space. The arguments of the boolean expression provide a *covering* of the same space.

Actually, only the (*oriented*) boundaries `V,FV`$_i$ $(1 \leq i \leq n)$ of the varius arguments are retained, togheter with the `V,CV` triangulation, for the successive steps of the algorithm.

2.2 Bucketing

The bounding boxes of `CV` and $\texttt{FV}_i$ are computed, and their *box-kd-tree* is worked-out, so providing a group of buckets of close cells, that can be elaborated independently, and possibly in parallel, to compute the intersection of the Delaunay simplices with the boundary cells.

2.3 Intersection

2.4 Reconstruction

3 Implementation

3.1 Box-kd-tree

Split the boxes between the below,above subsets

⟨ Split the boxes between the below,above subsets 2 ⟩ ≡

```
""" Split the boxes between the below,above subsets """
def splitOnThreshold(boxes,subset,coord):
    theBoxes = [boxes[k] for k in subset]
    threshold = centroid(theBoxes,coord)
    ncoords = len(boxes[0])/2
    a = coord%ncoords
    b = a+ncoords
    below,above = [],[]
    for k in subset:
        if boxes[k][a] <= threshold: below += [k]
    for k in subset:
        if boxes[k][b] >= threshold: above += [k]
    return below,above
```
◇

Macro referenced in 5a.

Test if bucket OK or append to splitting stack

⟨ Test if bucket OK or append to splitting stack 3a ⟩ ≡

```
""" Test if bucket OK or append to splitting stack """
def splitting(bucket,below,above, finalBuckets,splittingStack):
    if (len(below)<4 and len(above)<4) or len(set(bucket).difference(below))<7 \
        or len(set(bucket).difference(above))<7:
        finalBuckets.append(below)
        finalBuckets.append(above)
    else:
        splittingStack.append(below)
        splittingStack.append(above)
```
◇

Macro referenced in 5a.

Remove subsets from bucket list

⟨ Remove subsets from bucket list 3b ⟩ ≡

```
""" Remove subsets from bucket list """
def removeSubsets(buckets):
    n = len(buckets)
    A = zeros((n,n))
    for i,bucket in enumerate(buckets):
        for j,bucket1 in enumerate(buckets):
            if set(bucket).issubset(set(bucket1)):
                A[i,j] = 1
    B = AA(sum)(A.tolist())
    out = [bucket for i,bucket in enumerate(buckets) if B[i]==1]
    return out

def geomPartitionate(boxes,buckets):
    geomInters = [set() for h in range(len(boxes))]
    for bucket in buckets:
        print "\nbucket =",bucket
        for k in bucket:
            print "k =",k
            geomInters[k] = geomInters[k].union(bucket)
            print "geomInters[k] =",geomInters[k]

    """for h,inters in enumerate(geomInters):
        geomInters[h] = geomInters[h].difference([h])"""
    return AA(list)(geomInters)
◇
```

Macro referenced in 5a.

Iterate the splitting until `splittingStack` is empty

⟨ Iterate the splitting until splittingStack is empty 4a ⟩ ≡

```
""" Iterate the splitting until \texttt{splittingStack} is empty """
def boxBuckets(boxes):
    bucket = range(len(boxes))
    splittingStack = [bucket]
    finalBuckets = []
    while splittingStack != []:
        bucket = splittingStack.pop()
        below,above = splitOnThreshold(boxes,bucket,1)
        below1,above1 = splitOnThreshold(boxes,above,2)
```

```
            below2,above2 = splitOnThreshold(boxes,below,2)
            below11,above11 = splitOnThreshold(boxes,above1,3)
            below21,above21 = splitOnThreshold(boxes,below1,3)
            below12,above12 = splitOnThreshold(boxes,above2,3)
            below22,above22 = splitOnThreshold(boxes,below2,3)

            splitting(above1,below11,above11, finalBuckets,splittingStack)
            splitting(below1,below21,above21, finalBuckets,splittingStack)
            splitting(above2,below12,above12, finalBuckets,splittingStack)
            splitting(below2,below22,above22, finalBuckets,splittingStack)

            finalBuckets = list(set(AA(tuple)(finalBuckets)))
        parts = geomPartitionate(boxes,finalBuckets)
        return sorted(AA(sorted)(parts))
    ◇
```

Macro referenced in 5a.

aaaaaa

⟨ aaaaaa 4b ⟩ ≡

```
    """ aaaaa """

    ◇
```

Macro never referenced.

3.2 Merging the boundaries

3.3 Elementary splitting

3.4 Boolean chains

4 Esporting the Library

"lib/py/bool2.py" 5a ≡

```
    """ Module for Boolean computations between geometric objects """
    from pyplasm import *
    """ import modules from larcc/lib """
    import sys
    sys.path.insert(0, 'lib/py/')
    from inters import *
    DEBUG = True

    ⟨ Coding utilities 7a ⟩
    ⟨ Split the boxes between the below,above subsets 2 ⟩
    ⟨ Test if bucket OK or append to splitting stack 3a ⟩
```

⟨Remove subsets from bucket list 3b⟩
⟨Iterate the splitting until splittingStack is empty 4a⟩
◇

5 Test examples

5.1 Random triangles

Generation of random triangles and their boxes

"test/py/bool2/test01.py" 5b ≡
```
""" Generation of random triangles and their boxes """
import sys
sys.path.insert(0, 'lib/py/')
from bool2 import *
glass = MATERIAL([1,0,0,0.1,  0,1,0,0.1,  0,0,1,0.1, 0,0,0,0.1, 100])

randomTriaArray = randomTriangles(10,0.99)
VIEW(STRUCT(AA(MKPOL)([[verts, [[1,2,3]], None] for verts in randomTriaArray])))

boxes = containmentBoxes(randomTriaArray)
hexas = AA(box2exa)(boxes)
cyan = COLOR(CYAN)(STRUCT(AA(MKPOL)([[verts, [[1,2,3]], None] for verts in randomTriaArray])))
yellow = STRUCT(AA(glass)(AA(MKPOL)([hex for hex,qualifier in hexas])))
VIEW(STRUCT([cyan,yellow]))
```
◇

Generation of random quadrilaterals and their boxes

"test/py/bool2/test02.py" 6a ≡
```
""" Generation of random quadrilaterals and their boxes """
import sys
sys.path.insert(0, 'lib/py/')
from bool2 import *
glass = MATERIAL([1,0,0,0.1,  0,1,0,0.1,  0,0,1,0.1, 0,0,0,0.1, 100])

randomQuadArray = randomQuads(10,1)
VIEW(STRUCT(AA(MKPOL)([[verts, [[1,2,3,4]], None] for verts in randomQuadArray])))

boxes = containmentBoxes(randomQuadArray)
hexas = AA(box2exa)(boxes)
cyan = COLOR(CYAN)(STRUCT(AA(MKPOL)([[verts, [[1,2,3,4]], None] for verts in randomQuadArray])
yellow = STRUCT(AA(glass)(AA(MKPOL)([hex for hex,qualifier in hexas])))
VIEW(STRUCT([cyan,yellow]))
```
◇

5.2 Testing the box-kd-trees

Visualizing with different colors the buckets of box-kd-tree

"test/py/bool2/test04.py" 6b ≡

```
""" Visualizing with different colors the buckets of box-kd-tree """
from pyplasm import *
""" import modules from larcc/lib """
import sys
sys.path.insert(0, 'lib/py/')
from bool2 import *
glass = MATERIAL([1,0,0,0.1,  0,1,0,0.1,  0,0,1,0.1, 0,0,0,0.1, 100])

randomQuadArray = randomQuads(30,0.2)
VIEW(STRUCT(AA(MKPOL)([[verts, [[1,2,3,4]], None] for verts in randomQuadArray])))

boxes = containmentBoxes(randomQuadArray)
hexas = AA(box2exa)(boxes)
glass = MATERIAL([1,0,0,0.1,  0,1,0,0.1,  0,0,1,0.1, 0,0,0,0.1, 100])
yellow = STRUCT(AA(glass)(AA(MKPOL)([hex for hex,data in hexas])))
VIEW(STRUCT([#cyan,
    yellow]))

parts = boxBuckets(boxes)
for k,part in enumerate(parts):
    bunch = [glass(STRUCT( [MKPOL(hexas[h][0]) for h in part]))]
    bunch += [COLOR(RED)(MKPOL(hexas[k][0]))]
    VIEW(STRUCT(bunch))
◇
```

A Code utilities

Coding utilities Some utility fuctions used by the module are collected in this appendix. Their macro names can be seen in the below script.

⟨Coding utilities 7a⟩ ≡

```
""" Coding utilities """
⟨Generation of a random 3D point 8b⟩
⟨Generation of random 3D triangles 7b⟩
⟨Generation of random 3D quadrilaterals 8a⟩
⟨Generation of a single random triangle 8c⟩
⟨Containment boxes 9a⟩
⟨Transformation of a 3D box into an hexahedron 9b⟩
⟨Computation of the 1D centroid of a list of 3D boxes 10⟩
◇
```

Macro referenced in 5a.

Generation of random triangles The function `randomTriangles` returns the array `randomTriaArray` with a given number of triangles generated within the unit 3D interval. The `scaling` parameter is used to scale every such triangle, generated by three randow points, that could be possibly located to far from each other, even at the distance of the diagonal of the unit cube.

The arrays `xs`, `ys` and `zs`, that contain the x, y, z coordinates of triangle points, are used to compute the minimal translation `v` needed to transport the entire set of data within the positive octant of the 3D space.

⟨ Generation of random 3D triangles 7b ⟩ ≡

```
""" Generation of random triangles """
def randomTriangles(numberOfTriangles=400,scaling=0.3):
    randomTriaArray = [rtriangle(scaling) for k in range(numberOfTriangles)]
    [xs,ys,zs] = TRANS(CAT(randomTriaArray))
    xmin, ymin, zmin = min(xs), min(ys), min(zs)
    v = array([-xmin,-ymin, -zmin])
    randomTriaArray = [[list(v1+v), list(v2+v), list(v3+v)] for v1,v2,v3 in randomTriaArray]
    return randomTriaArray
```
◇

Macro referenced in 7a.

Generation of random 3D quadrilaterals

⟨ Generation of random 3D quadrilaterals 8a ⟩ ≡

```
""" Generation of random 3D quadrilaterals """
def randomQuads(numberOfQuads=400,scaling=0.3):
    randomTriaArray = [rtriangle(scaling) for k in range(numberOfQuads)]
    [xs,ys,zs] = TRANS(CAT(randomTriaArray))
    xmin, ymin, zmin = min(xs), min(ys), min(zs)
    v = array([-xmin,-ymin, -zmin])
    randomQuadArray = [AA(list)([ v1+v, v2+v, v3+v, v+v2-v1+v3 ]) for v1,v2,v3 in randomTriaAr
    return randomQuadArray
```
◇

Macro referenced in 7a.

Generation of a random 3D point A single random point, codified in floating point format, and with a fixed (quite small) number of digits, is returned by the `rpoint()` function, with no input parameters.

⟨ Generation of a random 3D point 8b ⟩ ≡

```
""" Generation of a random 3D point """
def rpoint():
    return eval( vcode([ random.random(), random.random(), random.random() ]) )
```
◇

Macro referenced in 7a.

Generation of a single random triangle A single random triangle, scaled about its centroid by the `scaling` parameter, is returned by the `rtriangle()` function, as a tuple ot two random points in the unit square.

⟨ Generation of a single random triangle 8c ⟩ ≡

```
""" Generation of a single random triangle """
def rtriangle(scaling):
    v1,v2,v3 = array(rpoint()), array(rpoint()), array(rpoint())
    c = (v1+v2+v3)/3
    pos = rpoint()
    v1 = (v1-c)*scaling + pos
    v2 = (v2-c)*scaling + pos
    v3 = (v3-c)*scaling + pos
    return tuple(eval(vcode(v1))), tuple(eval(vcode(v2))), tuple(eval(vcode(v3)))
◇
```

Macro referenced in 7a.

Containment boxes Given as input a list `randomTriaArray` of pairs of 2D points, the function `containmentBoxes` returns, in the same order, the list of *containment boxes* of the input lines. A *containment box* of a geometric object of dimension d is defined as the minimal d-cuboid, equioriented with the reference frame, that contains the object. For a 2D line it is given by the tuple $(x1, y1, x2, y2)$, where $(x1, y1)$ is the point of minimal coordinates, and $(x2, y2)$ is the point of maximal coordinates.

⟨ Containment boxes 9a ⟩ ≡

```
""" Containment boxes """
def containmentBoxes(randomPointArray,qualifier=0):
    if len(randomPointArray[0])==2:
        boxes = [eval(vcode([min(x1,x2), min(y1,y2), min(z1,z2),
                             max(x1,x2), max(y1,y2), max(z1,z2)]))+[qualifier]
                for ((x1,y1,z1),(x2,y2,z2)) in randomPointArray]
    elif len(randomPointArray[0])==3:
        boxes = [eval(vcode([min(x1,x2,x3), min(y1,y2,y3), min(z1,z2,z3),
                             max(x1,x2,x3), max(y1,y2,y3), max(z1,z2,z3)]))+[qualifier]
                for ((x1,y1,z1),(x2,y2,z2),(x3,y3,z3)) in randomPointArray]
    elif len(randomPointArray[0])==4:
        boxes = [eval(vcode([min(x1,x2,x3,x4), min(y1,y2,y3,y4), min(z1,z2,z3,z4),
                             max(x1,x2,x3,x4), max(y1,y2,y3,y4), max(z1,z2,z3,z4)]))+[qualifie
                for ((x1,y1,z1),(x2,y2,z2),(x3,y3,z3),(x4,y4,z4)) in randomPointArray]
    return boxes
◇
```

Macro referenced in 7a.

Transformation of a 3D box into an hexahedron The transformation of a 2D box into a closed rectangular polyline, given as an ordered sequwncw of 2D points, is produced by the function `box2exa`

⟨Transformation of a 3D box into an hexahedron 9b⟩ ≡

```
""" Transformation of a 3D box into an hexahedron """
def box2exa(box):
    x1,y1,z1,x2,y2,z2,type = box
    verts = [[x1,y1,z1], [x1,y1,z2], [x1,y2,z1], [x1,y2,z2], [x2,y1,z1], [x2,y1,z2], [x2,y2,z1
    cell = [range(1,len(verts)+1)]
    return [verts,cell,None],type

def lar2boxes(model,qualifier=0):
    V,CV = model
    boxes = []
    for k,cell in enumerate(CV):
        verts = [V[v] for v in cell]
        x1,y1,z1 = [min(coord) for coord in TRANS(verts)]
        x2,y2,z2 = [max(coord) for coord in TRANS(verts)]
        boxes += [eval(vcode([min(x1,x2),min(y1,y2),min(z1,z2),max(x1,x2),max(y1,y2),max(z1,z2
    return boxes
```
◇

Macro referenced in 7a.

Computation of the 1D centroid of a list of 3D boxes The 1D `centroid` of a list of 3D boxes is computed by the function given below. The direction of computation (either x, y or z) is chosen depending on the value of the `coord` parameter.

⟨Computation of the 1D centroid of a list of 3D boxes 10⟩ ≡

```
""" Computation of the 1D centroid of a list of 3D boxes """
def centroid(boxes,coord):
    delta,n = 0,len(boxes)
    ncoords = len(boxes[0])/2
    a = coord%ncoords
    b = a+ncoords
    for box in boxes:
        delta += (box[a] + box[b])/2
    return delta/n
```
◇

Macro referenced in 7a.

References

[BP96] Chandrajit L. Bajaj and Valerio Pascucci, *Splitting a complex of convex polytopes in any dimension*, Proceedings of the Twelfth Annual Symposium on Computational Geometry (New York, NY, USA), SCG '96, Acm, 1996, pp. 88–97.

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