Boolean chains [*]

Alberto Paoluzzi

March 15, 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 to be used for local intersection, followed by elimination of duplicated data. Actually we reduce the intersection of boundaries in 3D to the independent intersections of the buckets of (transformed) faces with the 2D subspace $z = 0$, in order to reconstruct each splitted facet of boolean arguments, suitably transformed ther together with the bucket of indent facets. 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

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*.

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [**?**]. March 15, 2015

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 discrete set of points generated by intersection of cells of complementary dimension, i.e. whose dimensions add up to the dimension of the ambient space. Actually, only the (*oriented*) boundaries `V,FV`$_i$ $(1 \leq i \leq n)$ of the varius arguments are retained here, and used by the following steps of the algorithm.

2.2 Bucketing

The bounding boxes of facets `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 intersections of the boundary cells.

2.3 Intersection

For each facet f of one of Boolean arguments, the subset $F(f)$ of incident or intersecting facets of boundaries of the other arguments were computed in the previous *bucketing* step. So, each F is transformed by the affine map that sends f into the $z = 0$ subspace, and there is intersected with this subspace, generating a subset $E(f)$ of coplanar edges. This one is projected in 2D, and the *regularized* cellular 2-complex $G(f)$ induced by it is computed, and mapped back to the original space position and orientation of f (providing a partition of it induced by the other boundaries).

2.4 Reconstruction

Like for in the reconstruction of 2D solid cells using the angular ordering of edges around the vertices, the coincident edges are identified in 3D , and used to sort the incident faces sing vhe falues of solid angles given with one reference face. The 3D space partition induced by $\cup_f G(f)$ is finally reconstructed, possibly in parallel, by traversing the adjacent sets of facets on the boundary of each solid cell.

3 Implementation

3.1 Box-kd-tree

Split the boxes between the (below,above) subsets @D Split the boxes between the below,above subsets @""" 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 = coordb = 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 @

Test if bucket OK or append to splitting stack @D Test if bucket OK or append to splitting stack @""" 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) @

Remove subsets from bucket list @D Remove subsets from bucket list @ """ 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: for k in bucket: geomInters[k] = geomInters[k].union(bucket) for h,inters in enumerate(geomInters): geomInters[h] = geomInters[h].difference([h]) return AA(list)(geomInters) @

Iterate the splitting until `splittingStack` is empty @D Iterate the splitting until splittingStack is empty @""" Iterate the splitting until `splittingStack` is empty """ def boxTest(boxes,h,k): B1,B2,B3,B4,B5,B6,$_= boxes[k]b1,b2,b3,b4,b5,b6,_= boxes[h]returnnot(b4 < B1orB4 < b1orb5 < B2orB5 < b2orb6 < B3orB6 < b3)$

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) parts = [[h for h in part if boxTest(boxes,h,k)] for k,part in enumerate(parts)] return AA(sorted)(parts) @

aaaaaa @D aaaaaa @""" aaaaa """
@

3.2 Merging the boundaries

3.3 Elementary splitting

In this section we implement the splitting of $(d-1)$-faces, stored in `FV`, induced by the buckets of $(d-1)$-faces, stored in `parts`, and one-to-one associated to them. Of course, (a) both such arrays have the same number of elements, and (b) whereas `FV` contains the indices of incident vertices for each face, `parts` contains the indices of adjacent faces for each face, with the further constraint that $i \notin$ `parts`(i).

Computation of topological relations The function `crossRelation` is used here to compute a topological relation starting from two characteristic matrices `XV` and `YV`, that associate the sets of topological objects X and Y with their vertices, respectively. The technique using sparse binary matrices stored in `CSR` (Compressed Sparse Row) format is used.

@D Computation of topological relation @""" Computation of topological relation """ def crossRelation(XV,YV): csrXV = csrCreate(XV) csrYV = csrCreate(YV) csrXY = matrixProduct(csrXV, csrYV.T) XY = [None for k in range(len(XV))] for k,face in enumerate(XV): data = csrXY[k].data col = csrXY[k].indices XY[k] = [col[h] for h,val in enumerate(data) if val==2] NOTE: val depends on the relation under consideration ... return XY @

Submanifold mapping computation The 4×4 (affine) scipy matrix `transform` of type `mat` is computed by the function `submanifoldMapping`, using as input the array `pivotFace` that contains the vertices of the so-called *pivot* face, i.e. of the face to be mapped to the coordinate subspace $z = 0$ (in 3D).

@D Submanifold mapping computation @""" Submanifold mapping computation """ def submanifoldMapping(pivotFace): tx,ty,tz = pivotFace[0] transl = mat([[1,0,0,-tx],[0,1,0,-ty],[0,0,1,-tz],[0,0,0,1]]) facet = [VECTDIFF([v,pivotFace[0]]) for v in pivotFace] m = faceTransformations(facet) mapping = mat([[m[0,0],m[0,1],m[0,2],0],[m[1,0],m[1,1],m[1,2],0],[m[2,0],m[2,1],m[2,2],0],[0,0,0,1]]) transform = mapping * transl return transform @

Set of line segments partitioning a facet The more important function of this section is the higher level `intersection` function, that accepts as input the `LAR` model `(V,FV,EV)` to be partitioned, and the pair `(k,bundledFaces)`, where `k` is the index of the pivot face (to be transformed to the $z = 0$ subspace) and where `bundledFaces` is an array of indices of faces that are guarantee to share points with face k. Such shared points may be either boundary edges of k or a segment that is internal both to face k and to some face in `bundledFaces`.

@D Set of line segments partitioning a facet @""" Set of line segments partitioning a facet """ def intersection(V,FV,EV): def intersection0(k,bundledFaces): FE = crossRelation(FV,EV) pivotFace = [V[v] for v in FV[k]] transform = submanifoldMapping(pivotFace) submanifold transformation transformedCells,edges,faces = [],[],[] for face in bundledFaces: edge = set(FE[k]).intersection(FE[face]) common edge index if edge == set(): candidateEdges = FE[face] facet = [] for e in candidateEdges: cell = [V[v]+[1.0] for v in EV[e]] verts of incident face transformedCell = (transform * (mat(cell).T)).T.tolist() vertices in local frame facet += [[point[:-1] for point in transformedCell]] faces += [facet] else: boundary edges of face k e, = edge vs = [V[v]+[1.0] for v in EV[e]] ws = (transform * (mat(vs).T)).T.tolist() edges += [[p[:-1] for p in ws]] return edges,faces,transform return intersection0 @

Computation of face transformations The faces in every `parts`(i) must be affinely transformed into the subspace $x_d = 0$, in order to compute the intersection of its elements with this subspace, that are submanifolds of dimension $d-2$.

@D Computation of face transformations @""" Computation of affine face transformations """ def COVECTOR(points): pointdim = len(points[0]) plane = Planef.bestFittingPlane(pointdim, [item for sublist in points for item in sublist]) return [plane.get(I) for I in range(0,pointdim+1)]

def faceTransformations(facet): covector = COVECTOR(facet) translVector = facet[0] translation newFacet = [VECTDIFF([v,translVector]) for v in facet] linear transformation: boundaryFacet -¿ standard (d-1)-simplex d = len(facet[0]) transformMat = mat(newFacet[1:d] + [covector[1:]]).T.I transformation in the subspace x$_d = 0 out = (transformMat * (mat(newFacet).T)).T.tolist() return transformMat$@

Space partitioning via submanifold mapping the function `spacePartition`, given in the below script, takes as input a *non-valid* (with the meaning used in solid modeling field — see [?]) `LAR` model of dimension $d-1$, i.e. a triple `(V,FV,EV)`, and an array `parts` indexed on faces, and containing the subset of faces with greatest probability of intersecting each indexing face, respectively. The `spacePartition` function returns the *valid* `LAR` boundary model `(W,FW,EW)` of the space partition induced by `FV`.

@D Space partitioning via submanifold mapping @""" Space partitioning via submanifold mapping """ def spacePartition(V,FV,EV, parts): transfFaces = [] for k,bundledFaces in enumerate(parts): edges,faces,transform = intersection(V,FV,EV)(k,bundledFaces) for face in faces: line = [] for edge in face: (x1,y1,z1),(x2,y2,z2) = edge if not verySmall(z2-z1): x = (x2-x1)/(z2-z1) + x1 y = (y2-y1)/(z2-z1) + y1 p = [x,y,0] line += [eval(vcode(p))] if line!=[]: edges += [line] v,fv,ev = larFromLines([[point[:-1] for point in edge] for edge in edges]) if len(fv)¿1: fv = fv[:-1] lar = [w+[0.0] for w in v],fv,ev transfFaces += [Struct([larApply(transform.I)(lar)])] W,FW,EW = struct2lar(Struct(transfFaces)) return W,FW,EW @

3.4 Circular ordering of faces around edges

Directional and orthogonal projection operators In order to sort circularly the faces incident on each edge, we need of course to compute the relation EF, and for each face f incident on $e = (v_1, v_2)$, to project a vector $w_f = (v_1, v_f)$, non parallel to (v_1, v_2), on the subspace ortogonal to e. This may be done by mapping w_f with the tensor $I - e \otimes e$. Finally, the angles between vectors a, b in this orthogonal space to e may be computed by using the `atan2` function, that combines both the *sin* and the *cos* of the angle:

$$angle = atan2(norm(cross(a, b)), dot(a, b)).$$

Let us just remember that, by definition, $(e \otimes e)v = (e \cdot v)e$, where e, v are vectors. @D Directional and orthogonal projection operators @""" Directional and orthogonal projection operators """ def dirProject (e): def dirProject0 (v): return SCALARVECTPROD([INNERPROD([UNITVECT(e), v]), UNITVECT(e)]) return dirProject0

def orthoProject (e): def orthoProject0 (v): return VECTDIFF([v, dirProject(UNITVECT(e))(v)]) return orthoProject0 @

3D boundary triangulation of the space partition The function `boundaryTriangulation` given below is used to guarantee that there is a unique (simple) facet incident to an edge and contained in one LAR facet. More clearly, the Boolean decompositions generated by LAR allow for non convex cells, and in particular for nonconvex boundary facets of d-cells. This fact may induce errors in the computation of circularly sorted faces around edges. Conversely, by decomposing the faces into triangles, such ordering problems cannot appear. We also note that whereas every $(d-1)$-facet is made by coherently oriented triangles, it is not possible to give—a priori—a coherently orientation to all the facets, since the object interior and exterior are not defined (for now).

@D 3D boundary triangulation of the space partition @from support import PolygonTessellator,vertex

def orientTriangle(pointTriple): v1 = array(pointTriple[1])-pointTriple[0] v2 = array(pointTriple[2])-pointTriple[0] if cross(v1,v2)[2] ¡ 0: return REVERSE(pointTriple) else: return pointTriple

def boundaryTriangulation(W,FW): triangleSet = [] for face in FW: pivotFace = [W[v] for v in face+(face[0],)] transform = submanifoldMapping(pivotFace) mappedVerts = (transform * (mat([p+[1.0] for p in pivotFace]).T)).T.tolist() facet = [point[:-2] for point in mappedVerts] pol = PolygonTessellator() vertices = [vertex.Vertex((x,y,0)) for (x,y) in facet] verts = pol.tessellate(vertices) ps = [list(v.point) for v in verts] trias = [[ps[k],ps[k+1],ps[k+2],ps[k]] for k in range(0,len(ps),3)] mappedVerts = (transform.I * (mat([p+[1.0] for p in ps]).T)).T.tolist() points = [p[:-1] for p in mappedVerts] trias = [[points[k],points[k+1],points[k+2],points[k]] for k in range(0,len(points),3) if scipy.linalg.norm(cross(array(points[k+1])-points[k], array(points[k+2])-points[k])) != 0] triangleSet += [AA(orientTriangle)(trias)] return triangleSet

def triangleIndices(triangleSet,W): vertDict,out = defaultdict(),[] for k,vertex in enumerate(W): vertDict[vcode(vertex)] = k for h,faceSetOfTriangles in enumerate(triangleSet):

out += [[[vertDict[vcode(p)] for p in triangle[:-1]] for triangle in faceSetOfTriangles]] return out @

Computation of incidence between edges and 3D triangles @D Computation of incidence between edges and 3D triangles @ def edgesTriangles(EF, FW, TW, EW): ET = [None for k in range(len(EF))] for e,edgeFaces in enumerate(EF): ET[e] = [] for f in edgeFaces: for t in TW[f]: if set(EW[e]).intersection(t)==set(EW[e]): ET[e] += [t] return ET @

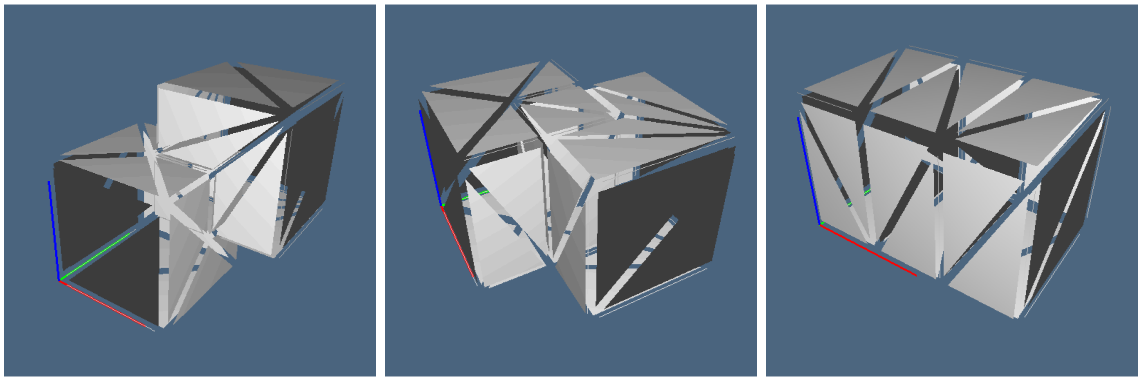

Figure 1: The triangulated boundaries of the space partition induced by two cubes (one is variously translated).

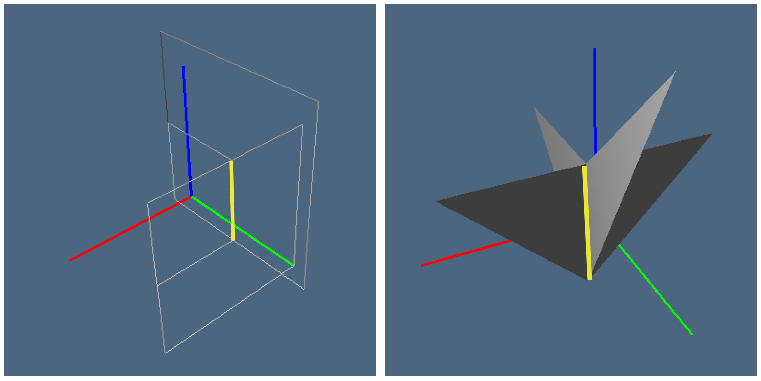

Figure 2: The triangles around an edge: `VIEW(STRUCT(MKPOLS((W,ET[35]))))`.

Example

```
In [2]: ET[35]
```

```
Out[2]: [[19, 7, 8], [6, 8, 7], [8, 7, 16], [4, 7, 8]]

In [3]: EF[35]
Out[3]: [4, 10, 11, 14]

In [4]: [FW[f] for f in  EF[35]]
Out[4]: [(19, 7, 8, 12), (6, 10, 8, 7), (12, 8, 7, 16, 1, 2), (4, 5, 6, 7, 8, 9)]

In [5]: EW[35]
Out[5]: (7, 8)
```

Slope of edges @D Slope of edges @""" Circular ordering of faces around edges """ from bool1 import invertRelation

def orientFace(v1,v2): def orientFace0(faceVerts): facet = list(faceVerts) + [list(faceVerts)[0]] pairs = [[facet[k],facet[k+1]] for k in range(len(facet[:-1]))] OK = False for pair in pairs: if [v1,v2] == pair: OK = True if OK: return faceVerts else: return REVERSE(faceVerts) return orientFace0

def planeProjection(normals): V = mat(normals) if all(V[:,0]==0): V = np.delete(V, 0, 1) elif all(V[:,1]==0): V = np.delete(V, 1, 1) elif all(V[:,2]==0): V = np.delete(V, 2, 1) return V

def faceSlopeOrdering(model): V,FV,EV = model triangleSet = boundaryTriangulation(V,FV) TV = triangleIndices(triangleSet,V) TE = crossRelation(CAT(TV),EV) ET,ET$_angle =$ $invertRelation(TE), []fore, etinenumerate(ET) : v1, v2 = EV[e]E = UNITVECT(VECTDIFF([V[v2], V[$ $ET[e][0]et_angle = []normals = []fortriangleinet : theFace = orientFace(v1, v2)(CAT(TV)[triangle])vect1 =$ $array(V[theFace[1]])-V[theFace[0]]vect2 = array(V[theFace[2]])-V[theFace[0]]normals+ =$ $[cross(vect1, vect2).tolist()]w1, w2 = E, normals[0]w3 = cross(array(w1), w2).tolist()basis =$ $[w1, w2, w3]transform = mat(basis).ImappedNormals = (transform*mat(normals).T).TmappedNormals$ $planeProjection(mappedNormals)fork, tinenumerate(et) : angle = math.atan2(mappedNormals[k, 1], map$ $[angle]pairs = sorted(zip(et_angle, et))ET_angle+ = [[pair[1]forpairinpairs]]EF_angle =$ $ET_{t}o_EF_incidence(TV, FV, ET_angle)returnEF_angle$@

Edge-triangles to Edge-faces incidence In the function `ET_to_EF_incidence` below, we convert the Edge-triangles incidence table `ET_angle` to a Edge-faces incidence table `EF_angle`. The input data to the algoritm are the relations `TW,FW`, and, of course, the incidence `ET_angle`. It works by computing two translationa tables `tableFT` and `tableTF` from face indices to triangle indices and viceversa. Of course, `assert( len(EF_angle) == 2*len(FW) )` must be `True`.

@D Edge-triangles to Edge-faces incidence @""" Edge-triangles to Edge-faces incidence """ def ET$_{t}o_EF_incidence(TW, FW, ET_angle) : tableFT = [Noneforkinrange(len(FW))]t =$ $0forf, triasinenumerate(TW) : tableFT[f] = range(t, t+len(trias))t+ = len(trias)tableTF =$ $invertRelation(tableFT)EF_angle = [[tableTF[t][0]fortintriangles]fortrianglesinET_angle]print"ET_angle =$ $", ET_angleassert(len(EF_angle) == 2*len(FW))print"EF_angle = ", EF_anglereturnEF_angle$@

Ordered incidence relationship of edges and faces The `EF_angle` list of lists reports, for every edge in `EW`, the list of incident boundary faces, *counterclockwise ordered* around the positive edge direction (going from vertex of smaller index to vertex of grater index). Therefore the `ordered_csrEF` function, given below, returns the “compressed sparse row” matrix, row-indexed by edges (of the standard LAR model `W,FW,EW` returned after the faces splitting) and column-indexed by faces of the boundary, and such that in position (e, f) contains the index ℓ of the next face (after f, say) in the counterclockwise ordering of faces around e.

@D Ordered incidence relationship of edges and faces @””” Ordered incidence relationship of edges and faces ””” def ordered$_{c}srEF(EF_{a}ngle) : triples = []fore, efinenumerate(EF_{a}ngle) :$ $n = len(ef)fork, faceinenumerate(ef) : triples+ = [[e, ef[k], ef[(k+1)csrEF = triples2mat(triples, shape =$ $”csr”)returncsrEF@$

Example

```
[[ 0, 0,14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0,14, 0, 0, 0, 0, 0, 8, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0,11, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,15, 0, 0, 0,11, 0, 0, 0],
 [ 0, 8, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,16, 0, 0, 0,12, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,17, 0, 0, 0,13],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,17, 0, 0, 0, 0, 0,11],
 [ 0, 0, 0, 0, 0, 0,14, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0],
 [ 0, 0, 0, 0, 0,15, 0, 0,16, 0, 0, 0, 0, 0, 8, 5, 0],
 [ 0, 0, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0,11, 0, 9, 0, 0, 0, 0, 0, 0],
 [10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0,10, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0,13, 0, 0, 0, 9, 0, 0, 0, 0],
 [ 0, 0, 0, 0,12, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 8, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0,15, 0, 0, 0, 0, 0, 0, 7, 0, 0],
 [12, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 9, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0,14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0],
 [ 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [15, 0,16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,17, 0,15],
 [ 0, 6, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0,14, 0, 0, 7, 0,10, 0,12, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0,17, 0, 0, 0, 0, 0, 0, 0, 9],
 [ 0, 0, 0, 0, 5,11, 0, 0, 4, 0, 0, 8, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 8, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0,13, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0],
 [ 0, 0, 0, 0,16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0],
 [ 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,15, 0,13, 0, 0],
 [ 0, 0, 0, 6, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [ 0, 0, 0, 0,10, 0, 0, 0, 0, 0,11,14, 0, 0, 4, 0, 0, 0]]
```

Cells from $(d-1)$-dimensional LAR model Since faces in the space partition induced by overlaping 3-coverings are $(d-1)$-cells, they are located on the boundary of *two* d-cells of the partition. Hence, the traversal algorithm of the data structure storing the relevant information may be driven by signing the two cofaces of each face as being either already visited or not.

@D Cells from $(d-1)$-dimensional LAR model @""" Cells from $(d-1)$-dimensional LAR model """

def firstSearch(visited): for f,edges in enumerate(visited): for e,edge in enumerate(edges): if visited[f,e] == 0.0: visited[f,e] = 1.0 return f,e return -1,-1

def facesFromComponents(model): V,FV,EV = model CV = [] EF$_angle = faceSlopeOrdering(model)csrF$
$ordered_csrEF(EF_angle).TFE = crossRelation(FV,EV)visitedFE = [[0foredgeinface]forfaceinFE]face,$
$firstSearch(visited)edge = FV[face][e]ce = []whileTrue : if(ce == [])or(ce[0]! = edge) :$

ce += [edge] nextFace = csrFE[face,edge] edges = FE[nextFace]

try: edges = set(edges).difference([edge]) except ValueError: print 'ValueError: too many values to unpack' break

if e0==edge: pos=0 elif e1==edge: pos=1 elif e2==edge: pos=2

if visited[nextFace, pos] == 0: visited[nextFace, pos] = 1 face = nextface else: CV += [ce] ce = [] edge,v = firstSearch(visited) vertex = EV[edge][v] FV = [face for face in FV if face != None] return V,CV,FV,EV @

3.5 Boolean chains

4 Esporting the Library

@O lib/py/bool2.py @""" 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 @¿ @¡ Split the boxes between the below,above subsets @¿ @¡ Test if bucket OK or append to splitting stack @¿ @¡ Remove subsets from bucket list @¿ @¡ Iterate the splitting until splittingStack is empty @¿ @¡ Computation of face transformations @¿ @¡ Computation of affine face transformations @¿ @¡ Computation of topological relation @¿ @¡ Submanifold mapping computation @¿ @¡ Set of line segments partitioning a facet @¿ @¡ Space partitioning via submanifold mapping @¿ @¡ 3D boundary triangulation of the space partition @¿ @¡ Computation of incidence between edges and 3D triangles @¿

@¡ Directional and orthogonal projection operators @¿ @¡ Slope of edges @¿ @¡ Ordered incidence relationship of edges and triangles @¿ @¡ Edge-triangles to Edge-faces incidence @¿ @

5 Test examples

5.1 Random triangles

Generation of random triangles and their boxes @O test/py/bool2/test01.py @””” 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 @O test/py/bool2/test02.py @””” 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,y
@

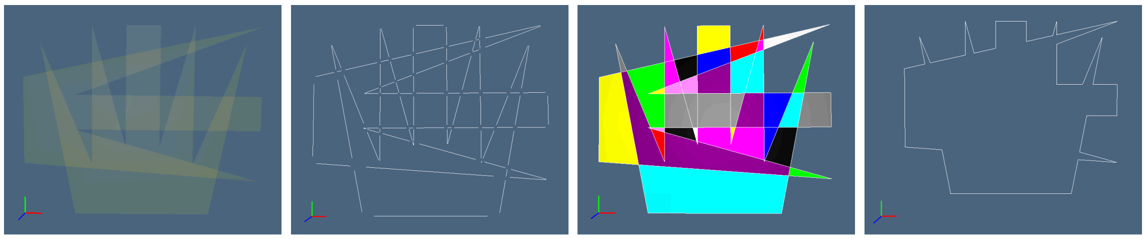

Figure 3: `LAR` complex from two polygons. (a) the input polygons; (b) the intersection of boundary lines; (c) the extracted *regularized* 2-complex; (d) the boundary `LAR`.

@O test/py/bool2/test03.py @””” Boolean complex generated by boundaries of two complexes """ import sys sys.path.insert(0, 'lib/py/') from inters import * glass = MATERIAL([1,0,0,0.1, 0,1,0,0.1, 0,0,1,0.1, 0,0,0,0.1, 100])

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 9],[6,8],[6,9],[7,8],[8,9],[9,10],[10,11]] BE1 = boundaryCells(FV1,EV1) lines1 = [[V1[v] for v in EV1[edge]] for edge in BE1]

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]] EV2 = [[0,1],[0,5],[0,7],[1,7],[2,3],[2,7],[3,6],[4,5],[4,6],[5,6],[6,7]] BE2 = boundaryCells(FV2,EV2) lines2 = [[V2[v] for v in EV2[edge]] for edge in BE2]

VIEW(STRUCT([glass(STRUCT(MKPOLS((V1,FV1)))), glass(STRUCT(MKPOLS((V2,FV2))))])) lines = lines1 + lines2 VIEW(STRUCT(AA(POLYLINE)(lines)))

global precision PRECISION += 2 V,FV,EV = larFromLines(lines) VIEW(EXPLODE(1.2,1.2,1)(MKPOLS

VV = AA(LIST)(range(len(V))) submodel = STRUCT(MKPOLS((V,EV))) VIEW(larModelNumbering(1, 1]],submodel,1))

polylines = [[V[v] for v in face+[face[0]]] for face in FV[:-1]] colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GREEN, GRAY, ORANGE, BLACK, BLUE, PURPLE, BROWN] sets = [COLOR(colors[kVIEW(STRUCT([T(3)(0.02)(STRUCT(AA(POLYLINE)(lines))), STRUCT(sets)]))

VIEW(EXPLODE(1.2,1.2,1)((AA(POLYLINE)(polylines)))) polylines = [[V[v] for v in FV[-1]+[FV[-1][0]]]] VIEW(EXPLODE(1.2,1.2,1)((AA(POLYLINE)(polylines)))) @

5.2 Testing the box-kd-trees

Visualizing with different colors the buckets of box-kd-tree @O test/py/bool2/test04.py @ """ 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 *

randomQuadArray = randomQuads(30,0.8) 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)) @

5.3 Intersection of geometry subsets

Two unit cubes @D Two unit cubes @""" Two unit cubes """ 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])

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True) cube1 = Struct([(V,FV,EV)],"cube1") twoCubes = Struct([cube1,t(-1,.5,1),cube1]) other test example twoCubes = Struct([cube1,t(.5,.5,.5),cube1]) twoCubes = Struct([cube1,t(.5,.5,0),cube1]) other test example twoCubes = Struct([cube1,t(.5,0,0),cube1]) other test example V,FV,EV = struct2lar(twoCubes) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,FV))))

quadArray = [[V[v] for v in face] for face in FV] boxes = containmentBoxes(quadArray) hexas = AA(box2exa)(boxes) parts = boxBuckets(boxes) @

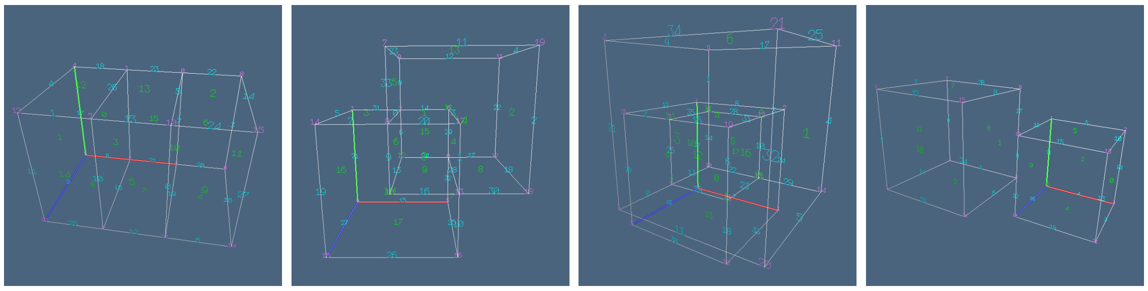

Figure 4: `LAR` complex of the space decomposition generated by two cubes in special positions. (a) translation on one coordinate; (b) translation on two coordinates; (c) translation on three coordinates; (d) non-manifold position along an edge.

Face (and incident faces) transformation @O test/py/bool2/test05.py @""" non-valid -¿ valid solid representation of a space partition """

@¡ Two unit cubes @¿

W,FW,EW = spacePartition(V,FV,EV, parts)

from architectural import * polylines = lar2polylines((W,FW)) VIEW(EXPLODE(1.2,1.2,1.2)(AA(POLYL

WW = AA(LIST)(range(len(W))) submodel = STRUCT(MKPOLS((W,EW))) VIEW(larModelNumbering

@

3-cell reconstruction from LAR space partition @O test/py/bool2/test06.py @""" 3-cell reconstruction from LAR space partition """ @¡ Two unit cubes @¿ W,FW,EW = spacePartition(V,FV,EV, parts) WW = AA(LIST)(range(len(W))) submodel = STRUCT(MKPOLS((W,EW) VIEW(larModelNumbering(1,1,1)(W,[WW,EW,FW],submodel,0.6)) @

2D polygon triangulation Here a 2D polygon is imported from an SVG file made of boundary lines, and the `V,FV,EV` LAR model is generated. Then the unique polygonal face in `FV` is embedded in 3D ($z = 0$), and triangulated using the tassellation algorithm extracted from pyOpenGL and pyGLContext, stored in the `lib/py/support.py` file. The generated triangles are finally coherently oriented, by testing the z-component of their normal vector.

@O test/py/bool2/test07.py @””” 2D polygon triangulation ””” import sys sys.path.insert(0, ’lib/py/’) from bool2 import *

filename = ”test/py/bool2/interior.svg” lines = svg2lines(filename) V,FV,EV = lar-FromLines(lines) VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,FV[:-1]+EV)) + AA(MK)(V)))

pivotFace = [V[v] for v in FV[0]+[FV[0][0]]] pol = PolygonTessellator() vertices = [vertex.Vertex((x,y,0)) for (x,y) in pivotFace] verts = pol.tessellate(vertices) ps = [list(v.point) for v in verts] trias = [[ps[k],ps[k+1],ps[k+2],ps[k]] for k in range(0,len(ps),3)] VIEW(STRUCT(AA(POLYLINE)(trias)))

triangles = DISTR([AA(orientTriangle)(trias),[[0,1,2]]]) VIEW(STRUCT(CAT(AA(MKPOLS)(triangles))
@

From triples of points to LAR model of boundary triangulation @O test/py/bool2/test08.py @ import sys sys.path.insert(0, ’lib/py/’) from bool2 import * sys.path.insert(0, ’test/py/bool2/’) from test06 import *

””” From triples of points to LAR model ”””

triangleSet = boundaryTriangulation(W,FW) TW = triangleIndices(triangleSet,W) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,CAT(TW))))) @

Visualization of of incidence between edges and 3D triangles @O test/py/bool2/test09.py @ ””” Visualization of of incidence between edges and 3D triangles ””” import sys sys.path.insert(0, ’lib/py/’) from bool2 import * sys.path.insert(0, ’test/py/bool2/’) from test08 import *

model = W,FW,EW FE = crossRelation(FW,EW) EF = invertRelation(FE)

triangleSet = boundaryTriangulation(W,FW) TW = triangleIndices(triangleSet,W) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,CAT(TW)))))

ET = edgesTriangles(EF,FW,TW,EW) VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,CAT(ET))))) VIEW(STRUCT(MKPOLS((W,ET[35]))))

from iot3d import polyline2lar V,FV,EV = polyline2lar([[W[v] for v in FW[f]] for f in EF[35]]) VIEW(STRUCT(MKPOLS((V,EV)))) @

Visualization of indices of the boundary triangulation @O test/py/bool2/test10.py @ ””” Visualization of indices of the boundary triangulation ””” import sys sys.path.insert(0, ’lib/py/’) from bool2 import * sys.path.insert(0, ’test/py/bool2/’) from test09 import *

model = W,FW,EW EF$_angle = faceSlopeOrdering(model)$

WW = AA(LIST)(range(len(W))) submodel = SKEL$_1(STRUCT(MKPOLS((W,CAT(TW)))))VIEW(l$

Visualization after sorted edge-faces incidence computation @O test/py/bool2/test11.py @ ””” Visualization of indices of the boundary triangulation ””” import sys sys.path.insert(0, ’lib/py/’) from bool2 import * sys.path.insert(0, ’test/py/bool2/’) from test06 import *

model = W,FW,EW EF$_angle = faceSlopeOrdering(model)$

WW = AA(LIST)(range(len(W))) submodel = SKEL$_1(STRUCT(MKPOLS((W,EW))))VIEW(larMod$

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.

@D Coding utilities @""" Coding utilities """ @¡ Generation of a random 3D point @¿ @¡ Generation of random 3D triangles @¿ @¡ Generation of random 3D quadrilaterals @¿ @¡ Generation of a single random triangle @¿ @¡ Containment boxes @¿ @¡ Transformation of a 3D box into an hexahedron @¿ @¡ Computation of the 1D centroid of a list of 3D boxes @¿ @

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.

@D Generation of random 3D triangles @""" 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 @

Generation of random 3D quadrilaterals @D Generation of random 3D quadrilaterals @""" 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(randomTria
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 randomTriaArray] return randomQuadArray @

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

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. @D Generation of a single random triangle @””” 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))) @

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.

@D Containment boxes @””” 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)]))+[qualifier] for ((x1,y1,z1),(x2,y2,z2),(x3,y3,z3),(x4,y4,z4)) in randomPointArray] return boxes @

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` @D Transformation of a 3D box into an hexahedron @””” 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], [x2,y2,z2]] 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),m
return boxes @

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. @D Computation of the 1D centroid of a list of 3D boxes @””” 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 = coordb = a+ncoords for box in boxes: delta += (box[a] + box[b])/2 return delta/n @