Boolean chains *

Alberto Paoluzzi

April 4, 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

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. April 4, 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 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 $\texttt{V,FV}_i$ $(1 \leq i \leq n)$ of the varius arguments are retained here, and used by the following steps of the algorithm.

2.2 Bucketing

The bounding boxes of facets $\texttt{FV}_i$ 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 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

⟨ 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 19b.

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 19b.

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:
        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)
◇
```

Macro referenced in 19b.

Iterate the splitting until `splittingStack` is empty

⟨ Iterate the splitting until splittingStack is empty 4 ⟩ ≡

```
""" Iterate the splitting until \texttt{splittingStack} is empty """
def boxTest(boxes,h,k):
    B1,B2,B3,B4,B5,B6,_ = boxes[k]
```

```
        b1,b2,b3,b4,b5,b6,_ = boxes[h]
        return not (b4<B1 or B4<b1 or b5<B2 or B5<b2 or b6<B3 or B6<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)
    ◇
```

Macro referenced in 19b.

aaaaaa

⟨ aaaaaa 5a ⟩ ≡

```
    """ aaaaa """

    ◇
```

Macro never referenced.

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.

⟨ Computation of topological relation 5b ⟩ ≡

```
""" 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
```

◇

Macro referenced in 19b.

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).

⟨ Submanifold mapping computation 6a ⟩ ≡

```
""" 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
```

◇

Macro referenced in 19b.

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`.

⟨ Set of line segments partitioning a facet 6b ⟩ ≡

```
""" 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
```

◇

Macro referenced in 19b.

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 $d - 2$.

⟨ Computation of face transformations 7a ⟩ ≡

```
""" 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
    ◇
```

Macro referenced in 19b.

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 [Req80]) `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`.

⟨ Space partitioning via submanifold mapping 7b ⟩ ≡

```
    """ 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
    ◇
```

Macro referenced in 19b.

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.

⟨ Directional and orthogonal projection operators 8 ⟩ ≡

```
    """ 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
    ◇
```

Macro referenced in 19b.

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).

⟨ 3D boundary triangulation of the space partition 9 ⟩ ≡

```
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
```
◇

Macro referenced in 19b.

Computation of incidence between edges and 3D triangles

⟨ Computation of incidence between edges and 3D triangles 10 ⟩ ≡

```
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
```

◇

Macro referenced in 19b.

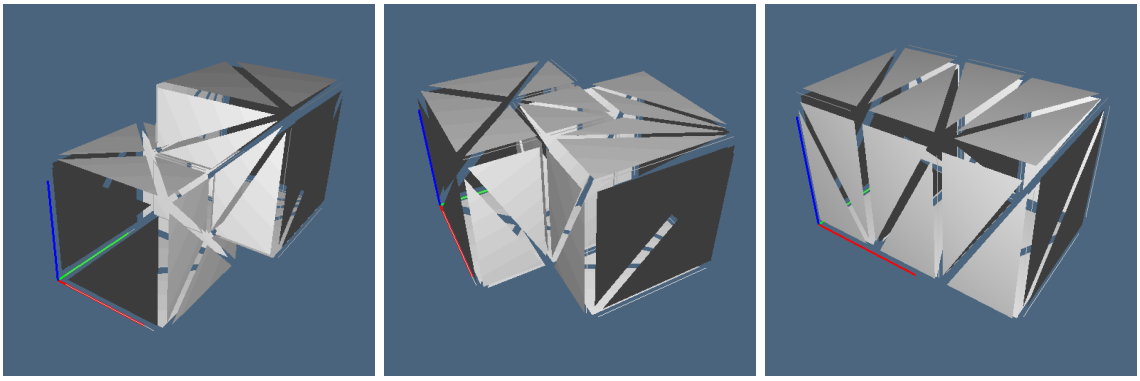

Figure 1: The triangulated boundaries of the space partition induced by two cubes (one is variously translated).

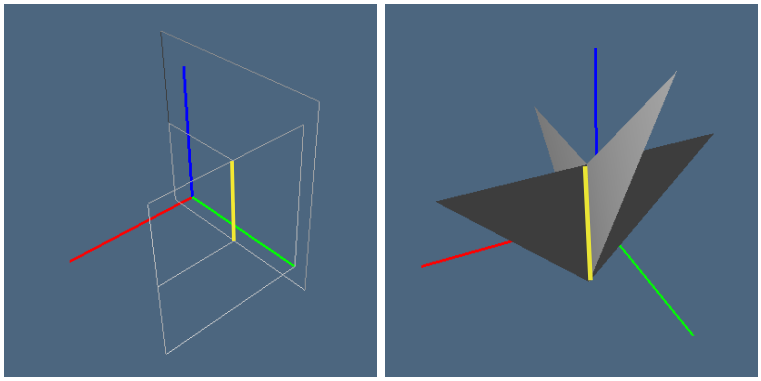

Figure 2: The triangles around an edge: `VIEW(STRUCT(MKPOLS((W,ET[35]))))`.

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 The `faceSlopeOrdering` function, given in the script below, return the list `EF_angle` of lists of faces incident to the model edges, counterclockwise ordered with respect to the orientation of the edge. Let us remember that the edges are naturally oriented from the vertex of lesser index to that of greater index.

⟨ Slope of edges 11 ⟩ ≡

```
    """ 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),[]
```

```
    for e,et in enumerate(ET):
        v1,v2 = EV[e]
        E = UNITVECT(VECTDIFF([V[v2],V[v1]]))
        refFace = ET[e][0]
        et_angle = []
        normals = []
        for triangle in et:
            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).I
        mappedNormals = (transform * mat(normals).T).T
        mappedNormals = planeProjection(mappedNormals)
        for k,t in enumerate(et):
            angle = math.atan2(mappedNormals[k,1],mappedNormals[k,0])
            et_angle += [angle]
        pairs = sorted(zip(et_angle,et))
        ET_angle += [[pair[1] for pair in pairs]]
    EF_angle = ET_to_EF_incidence(TV,FV, ET_angle)
    return EF_angle
```

◇

Macro referenced in 19b.

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`.

⟨ Edge-triangles to Edge-faces incidence 12 ⟩ ≡

```
"""" Edge-triangles to Edge-faces incidence """
def ET_to_EF_incidence(TW,FW, ET_angle):
    tableFT = [None for k in range(len(FW))]
    t = 0
    for f,trias in enumerate(TW):
        tableFT[f] = range(t,t+len(trias))
        t += len(trias)
    tableTF = invertRelation(tableFT)
    EF_angle = [[tableTF[t][0] for t in triangles] for triangles in ET_angle]
    #assert( len(EF_angle) == 2*len(FW) )
```

```
    return EF_angle
```
◇

Macro referenced in 19b.

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.

Oriented cycle of vertices from a 1-cycle of unoriented edges The below `edgeCycleOrientation` is used to transform a list of unoriented edges, know to correspond to a closed but unoriented 1-cycle, into a 0-cycle, to be easily transformed into an *oriented 1-cycle* by taking pairwise every two adjacent nodes, included the lat and the first to close the cycle.

⟨ Oriented cycle of vertices from a 1-cycle of unoriented edges 13 ⟩ ≡

```
""" Oriented cycle of vertices from a 1-cycle of unoriented edges """
def theNext(FE,EF_angle,EV,cb,previous_cb,previousOrientedEdges,cf):
    previous_cb = cb
    def theNext0(previous_edge,face):
        cbe = copy.copy(cb)
        edges = list(set(FE[face]).intersection(cbe)) #difference(cbe))
        if edges==[]:
            edges = list(cbe)
            face = list(set(EF_angle[edges[0]]).intersection(cf))[0]
        if type(previousOrientedEdges[0])!=list:
            signs,next = cycles2permutation([previousOrientedEdges])
        else: signs,next = cycles2permutation(previousOrientedEdges)
        edge = edges[0]
        edgeOrientation = signs[edge]
        edgeFaces = EF_angle[edge]
        n = len(edgeFaces)
        if edgeOrientation == 1:
            ind = (edgeFaces.index(face) + 1)%n
        elif edgeOrientation == -1:
            ind = (edgeFaces.index(face) - 1)%n
        nextFace = edgeFaces[ind]
        nextFaceBoundary = list(set(FE[nextFace]))
        orientedEdges = cyclesOrientation(previousOrientedEdges,nextFaceBoundary,EV)
        return orientedEdges,nextFace,edge
    return theNext0
```
◇

Macro referenced in 19b.

Check and store the orientation of faces

⟨ Check and store the orientation of faces 14 ⟩ ≡

```
""" Check and store the orientation of faces """
def checkOrientation(previousOrientedEdges,orientedEdges,orientedFaceEdges,faceOrientations,fa
    list2 = CAT(orientedFaceEdges)
    if orientedEdges != []:
        list1 = CAT(orientedEdges)
    else: list1 = CAT(previousOrientedEdges)
    theList = set(list1).intersection(set(list2).union((lambda args:[-arg for arg in args])(li
    if theList==set() or orientedEdges==[]:
        theList = set(CAT(orientedFaceEdges))
    edge = list(theList)[0]
    if theList.issubset(list1):  # equal signs
        if faceOrientations[face][0] == None:
            faceOrientations[face][0] = edge
        elif faceOrientations[face][1] == None:
            faceOrientations[face][1] = edge
        else: print "error: faceOrientations"
    elif not theList.issubset(list1): # different signs
        if faceOrientations[face][0] == None:
            faceOrientations[face][0] = -edge
        elif faceOrientations[face][1] == None:
            faceOrientations[face][1] = -edge
        else: print "error: faceOrientations"
    else: print "error: checkOrientation"
    return faceOrientations
```

◇

Macro referenced in 19b.

3.5 Progressive reconstruction of 3-cell boundaries

The input to this stage is a 2-complex embedded in 3D, with 2-cells non necessarily convex. The output is the 3-space partition defined by the cellular 3-complex, whose 2-skeleton is the inpiut complex. In other words, we mu reconstruct the 3-cells induced by the 2-cells of the input complex. This is done reconstructing the 3-cells stepwise. Each 3-cell reconstruction is done starting from one `face` two-dimensional previously taken into account no more than one single time, so that every 2-face is used at most exacly twice. An example of use of the functions implemented in this section is given in example `test12.py`

Edge cycles associated to a closed chain of edges The problem here is to conserve in the new `cycles` the same orientation of the previous ones, passed through the `orientedEdges` variable. We can formalize the problem as follows. Let call `pcycles` (for "previous cycles") and `fcycle` (for "face cycle") the algorithm input. the output is the

coherently oriented `outcycles`. First, an orientation is given to `fcycle`; then this one is compared with the `pcycles` orientation, and it is possibly reversed, in order to get them coherently oriented. Finally, the direct sum of `pcycles` and `fcycle` is executed, giving the `outcycles`.

⟨ Cycles orientation 15a ⟩ ≡

```
    """ Cycles orientation """
    def cyclesOrientation(pcycles,fcycle,EV):
        print "$$$$$ pcycles,fcycle =",pcycles,fcycle
        ofcycle = boundaryCycles(fcycle,EV)[0] # oriented
        if type(pcycles[0])==list: opcycle = CAT(pcycles)
        else: opcycle = pcycles
        int = set(opcycle).intersection(ofcycle)
        if int != set():
            ofcycle = CAT(reverseOrientation([ofcycle]))
        outChain = [e for e in ofcycle if not (-e in opcycle)]
        outChain += [e for e in opcycle if not (-e in ofcycle)]
        return outChain

    if __name__ == "__main__":
        pcycles = [[-19, 13, 22, 23]]
        fcycle = [30, 20, 18, 2, 26, 19]
        cyclesOrientation(pcycles,fcycle)
```

◇

Macro referenced in 19b.

⟨ Edge cycles associated to a closed chain of edges 15b ⟩ ≡

```
    """ Edge cycles associated to a closed chain of edges """
    def boundaryCycles(edgeBoundary,EV):
        verts2edges = defaultdict(list)
        for e in edgeBoundary:
            verts2edges[EV[e][0]] += [e]
            verts2edges[EV[e][1]] += [e]
        cycles = []
        cbe = copy.copy(edgeBoundary)
        while cbe != []:
            e = cbe[0]
            v = EV[e][0]
            cycle = []
            while True:
                cycle += [(e,v)]
                e = list(set(verts2edges[v]).difference([e]))[0]
                cbe.remove(e)
                v = list(set(EV[e]).difference([v]))[0]
                if (e,v)==cycle[0]:
```

```
                break
            n = len(cycle)
            cycles += [[e if EV[e]==(cycle[(k-1)%n][1],cycle[k%n][1]) else -e
                for k,(e,v) in enumerate(cycle)]]
        return cycles
    ◇
```

Macro referenced in 19b.

Permutation of edges defined by edge cycles

⟨ Permutation of edges defined by edge cycles 16 ⟩ ≡

```
    """ Permutation of edges defined by edge cycles """
    def cycles2permutation(cycles):
        next = []
        for cycle in cycles:
            next += zip(AA(ABS)(cycle),AA(ABS)(cycle[1:]+[cycle[0]]))
        next = dict(next)
        sign = dict([[ABS(edge),SIGN(edge)] for cycle in cycles for edge in cycle])
        return sign,next
    ◇
```

Macro referenced in 19b.

The 3-cell traversal algorithm Initially, the list of counterclockwise ordered faces around the oriented edges are computed, and stored as indexed by edges in the `EF_angle` list of lists. This information is stored in the compressed sparse row matrix `csrEF`, whose element (e, f) provides the *next* face index incident on edge e, after f.

Also, a list of list of zeros is stored in the `visitedFE` variable, in order to memorize the visited pairs (f, e) by writing one in their corresponding positions. The `firstSearch` function will so retrieve the first non visited pair, in order to start the extraction of a new 3-cell. The `cv` variable accumulates the vertex indices of the current 3-cell. When the 3-cell is completely extracted (how-to test?), will be stored as a new row in the `CV` relation.

The test for completeness of the extraction is done by computing the current boundary of the cell as a set of edges of faces, by python `XOR` of the edges of every accumulated face-edge relation. When this set it becomes empty, the 3-cell extraction is completed.

⟨ Cells from $(d-1)$-dimensional LAR model 17 ⟩ ≡

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

    def facesFromComponents(model):
        debug = 1
        V,FV,EV = model
        CV,CF,CE = [],[],[]
        orientedEdges = []
        EF_angle = faceSlopeOrdering(model)
```

```
FE = crossRelation(FV,EV)
visitedCell = [[None,None] for k in range(len(FV))]+[0]
print "$$$$ count =",visitedCell[-1],0,0,0

face = 0
orientedEdges = CAT(boundaryCycles(FE[face],EV))
visitedCell[face][0] = orientedEdges[0]
edge = ABS(orientedEdges[0])

cv = set(FV[face])
cb = set(FE[face])
previous_cb = set(FE[face])
print "\norientedEdges,face,edge,visitedCell[face] =",orientedEdges,face,edge,visitedCell[
ce = set([edge])
cf = set([face])
while debug<35:
    print "\ncv,cb,ce, cf =",cv,cb,ce, cf
    debug += 1
    if (face,edge) == (-1,-1):
        print "BREAK"
        #break
    elif cb != set():
        previousOrientedEdges = orientedEdges
        orientedEdges,face,edge = theNext(FE,EF_angle,EV, cb,previous_cb, previousOriented
        print "\norientedEdges,face,edge,visitedCell[face] =",orientedEdges,face,edge,visi
        print ">>>> face =",face
        cv = cv.union(FV[face])
        edges = FE[face]
        cb_union = cb.union(edges)
        cb_intersection = cb.intersection(edges)
        previous_cb = cb
        cb = cb_union.difference(cb_intersection)
        edge = boundaryCycles(FE[face],EV)[0][0]
        if visitedCell[face][0]==None:
            visitedCell[face][0] = edge
            visitedCell[-1] += 1
            print "$$$$ count =",visitedCell[-1],face,edge,0
        elif visitedCell[face][0]!=None:
            visitedCell[face][1] = -edge
            visitedCell[-1] += 1
            print "$$$$ count =",visitedCell[-1],face,-edge,1
        print "\nvisitedCell =",visitedCell
        ce = ce.union(edges)
        cf = cf.union([face])
    else:
        CV += [cv]
```

```
                CF += [cf]
                CE += [ce]
                if orientedEdges==[]:
                    visitedCell,orientedEdges,face,edge = startCell(visitedCell,FE,EV)
                    print "\norientedEdges,face,edge,visitedCell[face] =",orientedEdges,face,edge,
                cv,cb,ce, cf = set(FV[face]),set(FE[face]),set([edge]),set([face])
        return V,CV,FV,EV,CF,CE
```

◇

Macro referenced in 19b.

Start a new 3-cell The function `startCell` below is used to begin the extraction of a new 3-cell (after the first one was already extracted). Therefore its aim is to choose as first face one already previously extracted, in order to begin the current boundary with one cycle coherently oriented. This will is implemented by looking for a "`face`" position stored in `visitedCell` with just one `None` value in its row.

⟨ Start a new 3-cell 18 ⟩ ≡

```
    """ Start a new 3-cell """
    def startCell(visitedCell,FE,EV):
        for face in range(len(visitedCell)):
            if len([term for term in visitedCell[face] if term==None])==1:
                break
        # visitedCell[face] now contains only one "None"
        previousOrientation = visitedCell[face][0]
        orientedEdges = CAT(boundaryCycles(FE[face],EV))
        if previousOrientation in orientedEdges:
            orientedEdges = reverseOrientation([orientedEdges])[0]
        edge = -previousOrientation
        visitedCell[face][1] = edge
        visitedCell[-1] += 1
        print "$$$$ count =",visitedCell[-1],face,edge,1
        return visitedCell,orientedEdges, face, ABS(edge)
```

◇

Macro referenced in 19b.

Face orientations storage In order to correctly accomplish the extraction of 3-cells from the 2-complex partition of the arguments' space, it is necessary to use twice every 2-face, belonging with opposite orientations to the boundaries of two adjacent 3-cells. The array `faceOrientations`, initializated to $n \times 2$ zeros, with n equal to the number of 2-cells, is so used to store the orientations of faces considered as 2-cycles of edges.

In particular, the orientation of the 2-face is equivalent to the embedded orientation of one of its edges, corresponding either to the intrinsic orientation of this one, or to its opposite orientation. Hence, every time a face is used during the extraction of a 3-cell,

(the elementary 1-chain of) one of its oriented edges is stored in `faceOrientations`, to remember its orientation, and eventually reverse the orientation of the face the next time it is used again. At the very end of the extraction algorithm, all the faces must be used twice, with opposite orientations.

⟨ Face orientations storage 19a ⟩ ≡

```
""" Face orientations storage """
def reverseOrientation(chain):
    return [REVERSE([-cell for cell in cycle]) for cycle in chain]
```

◇

Macro referenced in 19b.

3.6 Boolean chains

4 Esporting the Library

"lib/py/bool2.py" 19b ≡

```
""" 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 28a ⟩
⟨ 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 4 ⟩
⟨ Computation of face transformations 7a ⟩
⟨ Computation of affine face transformations ? ⟩
⟨ Computation of topological relation 5b ⟩
⟨ Submanifold mapping computation 6a ⟩
⟨ Set of line segments partitioning a facet 6b ⟩
⟨ Space partitioning via submanifold mapping 7b ⟩
⟨ 3D boundary triangulation of the space partition 9 ⟩
⟨ Computation of incidence between edges and 3D triangles 10 ⟩
⟨ Directional and orthogonal projection operators 8 ⟩
⟨ Slope of edges 11 ⟩
⟨ Oriented cycle of vertices from a 1-cycle of unoriented edges 13 ⟩
⟨ Edge-triangles to Edge-faces incidence 12 ⟩
⟨ Cells from $(d-1)$-dimensional LAR model 17 ⟩
⟨ Edge cycles associated to a closed chain of edges 15b ⟩
⟨ Permutation of edges defined by edge cycles 16 ⟩

⟨ Cycles orientation 15a ⟩
⟨ Start a new 3-cell 18 ⟩
⟨ Face orientations storage 19a ⟩
⟨ Check and store the orientation of faces 14 ⟩
◇

5 Test examples

5.1 Random triangles

Generation of random triangles and their boxes

"test/py/bool2/test01.py" 20a ≡

```
""" 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" 20b ≡

```
""" 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]))
◇
```

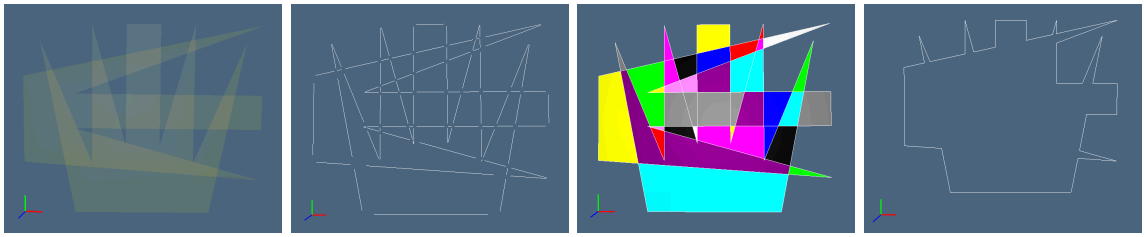

Figure 3: LAR complex from two polygons. (a) the input polygons; (b) the intersection of boundary lines; (c) the extracted *regularized* 2-complex; (d) the boundary LAR.

"test/py/bool2/test03.py" 21 ≡

```
""" 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],[4,10],[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((V,EV))))

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV[:-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[k%12])(FAN(pol)) for k,pol in enumerate(polylines)]
VIEW(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

"test/py/bool2/test04.py" 22 ≡

```
""" 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

⟨ Two unit cubes 23a ⟩ ≡

```
""" 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)
◇
```

Macro referenced in 23b, 24a.

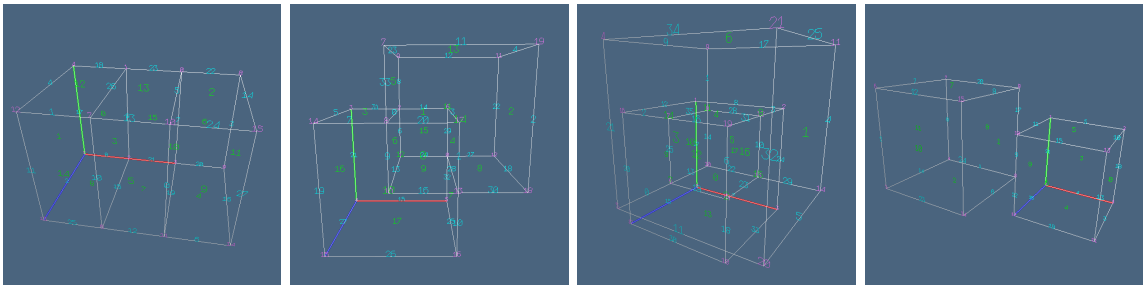

Figure 4: LAR complex of the space decomposition generated by two cubes in special positions. (a) translation on one coordinate; (b) translation on two coordinates; (c) translation on three coordinates; (d) non-manifold position along an edge.

Face (and incident faces) transformation

"test/py/bool2/test05.py" 23b ≡

```
""" non-valid -> valid solid representation of a space partition """

⟨ Two unit cubes 23a ⟩

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

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

WW = AA(LIST)(range(len(W)))
submodel = STRUCT(MKPOLS((W,EW)))
VIEW(larModelNumbering(1,1,1)(W,[WW,EW,FW],submodel,0.5))
◇
```

3-cell reconstruction from LAR space partition

"test/py/bool2/test06.py" 24a ≡

```
""" 3-cell reconstruction from LAR space partition """
⟨ Two unit cubes 23a ⟩
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.

"test/py/bool2/test07.py" 24b ≡

```
""" 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 = larFromLines(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

"test/py/bool2/test08.py" 25a ≡

```
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

"test/py/bool2/test09.py" 25b ≡

```
""" 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

"test/py/bool2/test10.py" 26a ≡

```
""" 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(larModelNumbering(1,1,1)(W,[WW,EW,CAT(TW)],submodel,0.6))
```
◇

Visualization after sorted edge-faces incidence computation

"test/py/bool2/test11.py" 26b ≡

```
""" 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 *

global count
count = 0

model = W,FW,EW
EF_angle = faceSlopeOrdering(model)

V,CV,FV,EV,CF,CE = facesFromComponents(model)
CF = AA(list)(CF)
CE = AA(list)(CE)
VIEW(EXPLODE(2,2,2)(MKPOLS((V,[CAT([FV[c] for c in cell]) for cell in CF]))))

VIEW(EXPLODE(2,2,2) (AA(STRUCT)(AA(MKPOLS)( DISTL([V,[[EV[c] for c in cell] for cell in [CE[-1]
```

```
VIEW(EXPLODE(2,2,2) (AA(STRUCT)(AA(MKPOLS)( DISTL([V,[[FV[c] for c in cell] for cell in CF]])))

WW = AA(LIST)(range(len(W)))
submodel = SKEL_1(STRUCT(MKPOLS((W,EW))))
VIEW(larModelNumbering(1,1,1)(W,[WW,EW,FW],submodel,0.6))
◇
```

Generation of the edge permutation associated to the 1-boundary of a 2-chain

"test/py/bool2/test12.py" 27 ≡

```
""" Generation of the edge permutation associated to the 1-boundary of a 2-chain """
import sys;sys.path.insert(0, 'lib/py/')
from bool2 import *
sys.path.insert(0, 'test/py/larcc/')
from test11 import *

C2 = csr_matrix((len(FV),1))
for i in [21,16,23,22, 2,3,4, 9,28,5]: C2[i,0] = 1
BD = boundary(FV,EV)
C1 = BD * C2
C_1 = [i for i in range(len(EV)) if ABS(C1[i,0]) == 1 ]
C_2 = [i for i in range(len(FV)) if C2[i,0] == 1 ]

VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,[EV[k] for k in C_1] + [FV[k] for k in C_2]))))

sign,next = cycles2permutation(boundaryCicles(C_1, EV))
print "\nsign =",sign
print "\nnext =",next,"\n"
◇
```

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 28a ⟩ ≡

```
""" Coding utilities """
global count
⟨ Generation of a random 3D point 29a ⟩
⟨ Generation of random 3D triangles 28b ⟩
⟨ Generation of random 3D quadrilaterals 28c ⟩
⟨ Generation of a single random triangle 29b ⟩
⟨ Containment boxes 29c ⟩
⟨ Transformation of a 3D box into an hexahedron 30 ⟩
```

⟨ Computation of the 1D centroid of a list of 3D boxes 31 ⟩
◇

Macro referenced in 19b.

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 28b ⟩ ≡

```
""" 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 28a.

Generation of random 3D quadrilaterals

⟨ Generation of random 3D quadrilaterals 28c ⟩ ≡

```
""" 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 28a.

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 29a ⟩ ≡

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

Macro referenced in 28a.

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 29b ⟩ ≡

```
    """ 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 28a.

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 29c ⟩ ≡

```
    """ 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 28a.

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 30 ⟩ ≡

```
    """ 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 28a.

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 31 ⟩ ≡

```
    """ 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 28a.

References

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

[Req80] Aristides G. Requicha, *Representations for rigid solids: Theory, methods, and systems*, ACM Comput. Surv. **12** (1980), no. 4, 437–464.