# Boolean chains *

Alberto Paoluzzi

January 28, 2016

**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]. January 28, 2016

# 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

#### Remove subsets from bucket list

⟨ Remove subsets from bucket list 2 ⟩ ≡

```
""" 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
```

◇

Macro referenced in 23.

**Iterate the splitting until `splittingStack` is empty**

⟨ Iterate the splitting until splittingStack is empty 3 ⟩ ≡

```
""" Iterate the splitting until \texttt{splittingStack} is empty """
def boxTest(boxes,h,k):
    if len(boxes[0])==4:
        B1,B2,B3,B4 = boxes[k]
        b1,b2,b3,b4 = boxes[h]
        return not (b3<B1 or B3<b1 or b4<B2 or B4<b2)
    else:
        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 boxBuckets3d(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 23.

## 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`$(i)$.

**Submanifold mapping computation** The $4\times4$ (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 4a ⟩ ≡

```
"""" 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 23.

### Helper functions for spacePartition

⟨ Helper functions for spacePartition 4b ⟩ ≡

```
""" Helper functions for spacePartition """
def submodel(V,FV,EV):
    FE = crossRelation(len(V),FV,EV)
    def submodel0(f,F):
        fE = list(set(FE[f] + CAT([FE[g] for g in F])))
        fF = [f]+F
        return fF,fE
    return submodel0

def meetZero( sW, (w1,w2) ):
    testValue = sW[w1][2] * sW[w2][2]
    if testValue > 10**-4:
        return False
    else: return True

def segmentIntersection(p1,p2):
    (x1,y1,z1),(x2,y2,z2) = p1,p2
    if abs(z1-z2) != 0.0:
        alpha = z1/(z1-z2)
```

```
            x = x1+alpha*(x2-x1)
            y = y1+alpha*(y2-y1)
            return x,y,0.0
        else: return None
```
◇

Macro referenced in 23.

**Computation of affine 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 $d - 2$.

A much better solution would be to compute a LU factorization of the the first columns of the U upper triangular or trapezoidal matrix ... see gauss-elimination-in-python

⟨ Computation of affine face transformations 5 ⟩ ≡

```
""" Computation of affine face transformations """
from numpy import array
from scipy.linalg import lu
from scipy.linalg.basic import det

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])
    m = mat( newFacet[1:d] + [covector[1:]] )
    if abs(det(m))<0.0001:
        for k in range(len(facet)-2):
            m = mat( newFacet[1+k+1:d+k+1] + [covector[1:]] )
            if abs(det(m))>0.0001: break
    transformMat = m.T.I
    # transformation in the subspace x_d = 0
    out = (transformMat * (mat(newFacet).T)).T.tolist()
    return transformMat
```
◇

Macro referenced in 23.

**Sorting of points along a line** Parametric sorting of aligned points along the array `line`.

⟨ Sorting of points along a line 6a ⟩ ≡

```
"""" Sorting of points along a line """"
def computeSegments(line):
    p0 = line[0]
    p1 = line[1]
    p1_p0 = VECTDIFF([p1,p0])
    h = sorted([(X_k,k) for k,X_k in enumerate(p1_p0) if X_k!=0.0], reverse=True)[0][1]
    params = [(p[h]-p0[h])/p1_p0[h] for p in line]
    sortedPoints = TRANS(sorted(zip(params,line)))[1]
    segments = TRANS([sortedPoints[:-1],sortedPoints[1:]])
    return segments
```

◇

Macro referenced in 23.

**Remove intersection points external to a submanifold face** When computing the LAR of each submanifold face embedded in $z = 0$ hyperplane, it is necessary to remove every vertex that was generated, outside the considered input face, by intersection of (incident) edges. This would produce spurious topology, i.e. cells that do not find adjacents cells incident to their boundaries.

⟨ Remove intersection points external to a submanifold face 6b ⟩ ≡

```
"""" Remove intersection points external to a submanifold face """"
def veryClose(p,q):
    out = False
    if VECTNORM(VECTDIFF([p,q])) <= 0.002:
        out = True
    return out

def removeExternals(M,V,EV,fe, z,fz,ez):
    (Z,FZ,EZ) = copy((z,fz,ez))
    inputFace = Struct([(V,[EV[e] for e in fe])])
    verts,ev = larApply(M)(struct2lar(inputFace))
    pol = [eval(vcode(vert[:-1])) for vert in verts],ev
    out = []
    classify = pointInPolygonClassification(pol)
    for k,point in enumerate(Z):
        if classify(point)=="p_out":  out += [k]

    # verify all v in out w.r.t. pol[0]
    trueOut = []
    for v in out:
        onBoundary = False
```

```
            for p in pol[0]:
                if veryClose(Z[v],p):
                    print "v,p,'close'", Z[v],p
                    onBoundary = True
                    Z[v] = p
            if not onBoundary: trueOut += [v]

        FW = [f for f in FZ if not any([v in trueOut for v in f])]  # trueOut
        EW = [e for e in EZ if not any([v in trueOut for v in e])]  # trueOut
        return Z,FW,EW
```

◇

Macro referenced in 23.

**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 7 ⟩ ≡

```
    """ Space partitioning via submanifold mapping """
    def spacePartition(V,FV,EV, parts):
        FE = crossRelation(len(V),FV,EV)
        submodel0 = submodel(V,FV,EV)
        out = []

        """ input: face index f; candidate incident faces F[f]; """
        for f,F in enumerate(parts):
            """ Selection of the LAR submodel S(f) := (V,FV,EV)(f) restricted to [f]+F[f] """
            fF,fE = submodel0(f,F)
            subModel = Struct([(V,[FV[g] for g in fF],[EV[h] for h in fE])])
            sV,sFV,sEV = struct2lar(subModel)
            #VIEW(STRUCT(MKPOLS((sV,sFV+sEV)) + [COLOR(RED)(STRUCT(MKPOLS((V,[FV[f]]))))]))

            """ Computation of submanifold map M moving f to z=0 """
            pivotFace = [V[v] for v in FV[f]]
            M = submanifoldMapping(pivotFace)  # submanifold transformation

            """ Transformation of S(f) by M, giving S = (sW,sEW) := M(S(f)) """
            sW,sFW,sEW = larApply(M)((sV,sFV,sEV))

            red = COLOR(RED)(STRUCT(MKPOLS(larApply(M)((V,[FV[f]])))))
            #VIEW(STRUCT(MKPOLS((sW,sFW+sEW)) + [red]))
```

```
            """ filtering of EW edges traversing z=0, giving EZ edges and incident faces FZEZ """
            sFE = crossRelation(len(V),sFW,sEW)
            edges = list(set([ e for k,face in enumerate(sFW)  for e in sFE[k]
                        if meetZero(sW, sEW[e]) ]))
            edgesPerFace = [ [e for e in sFE[k] if meetZero(sW, sEW[e])]
                        for k,face in enumerate(sFW) ]
            edges = list(set(CAT(edgesPerFace)))
            WW = AA(LIST)(range(len(sW)))
            wires = [sEW[e] for e in edges]
            #VIEW(STRUCT(MKPOLS((sW,wires)) + [red]))

            """ for each face in FZEZ, computation of the aligned set of points p(z=0) """
            points = collections.OrderedDict()
            lines = [[sW[w1],sW[w2]] for w1,w2 in wires]
            for k,(p,q) in enumerate(lines):
                point = segmentIntersection(p,q)
                if point != None: points[edges[k]] = point
            pointsPerFace = [set(face).intersection(points.keys()) for face in edgesPerFace]
            lines = [[points[e][:2] for e in face] for face in pointsPerFace]
            lines = [line for line in lines if line!=[]]
            vpoints = [[(vcode(point),k) for k,point in enumerate(line)] for line in lines]
            lines = [AA(eval)(dict(line).keys()) for line in vpoints]
            lines = [[line[0],line[-1]]  if len(line)>2 else line for line in lines]
            #VIEW(STRUCT(AA(POLYLINE)(lines) + [red]))

            """ Construction of the planar set FX,EX of faces and lines """
            u,fu,eu,polygons = larFromLines(lines)
            #VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((u,fu+eu))))

            """ Remove external vertices """
            w,fw,ew = removeExternals(M,V,EV,FE[f], u,fu,eu)
            #VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((w,fw+ew))))
            struct = Struct([(w,fw,ew)])
            z,fz,ez = struct2lar(struct)

            #VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((z,fz+ez))))

            w,fw,ew = larApply(M.I)(([v+[0.0] for v in z],fz,ez))
            out += [Struct([(w,fw,ew)])]
        return struct2lar(Struct(out))
```

◇

Macro referenced in 23.

### 3.4 Circular ordering of faces around edges

**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 ⟩ ≡

```
""" 3D boundary triangulation of the space partition """

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

from copy import copy

def boundaryTriangulation(V,FV,EV,FE):
    triangleSet = []

    def mapVerts(inverseMap):
        def mapVerts0(mappedVerts):
            return (inverseMap * (mat(mappedVerts).T)).T.tolist()
        return mapVerts0

    for f,face in enumerate(FV):
        print "f,face =",f,face
        triangledFace = []
        EW = [EV[e] for e in FE[f]]
        pivotFace = [V[v] for v in face]
        vertdict = dict([(w,v) for v,w in enumerate(face)])
        EW = [[vertdict[w] for w in edge] for edge in EW]
        transform = submanifoldMapping(pivotFace)
        mappedVerts = (transform * (mat([p+[1.0] for p in pivotFace]).T)).T.tolist()
        verts2D = [point[:-2] for point in mappedVerts]

        # Construction of CDT (Constrained Delaunay Triangulation) for LAR face
        model = (verts2D,[range(len(verts2D))],EW)
        struct = Struct([model])
```

```
            U,EU = structBoundaryModel(struct)
            W,FW,EW,polygons = larFromLines([[U[u],U[v]] for u,v in EU])
            setsOfTriangles = polygons2TriangleSet(W,polygons)
            trias = [[p+[1],q+[1],r+[1]] for p,q,r in setsOfTriangles[0]]

            inverseMap = transform.I
            trias = AA(mapVerts(inverseMap))(trias)
            triangledFace += [[v[:-1] for v in triangle] for triangle in trias]
            triangleSet += [triangledFace]
        return triangleSet

    def triangleIndices(triangleSet,W):
        tree = spatial.cKDTree(W)
        TV,FT,t = [],[],-1
        for face in triangleSet:
            ft = []
            for triangle in face:
                vertices = tree.query(triangle,1)[1].tolist()
                t += 1
                TV += [vertices]
                ft += [t]
            FT += [ft]
        VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,TV))))
        return TV,FT
```
◇

Macro referenced in 23.

### 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 23.

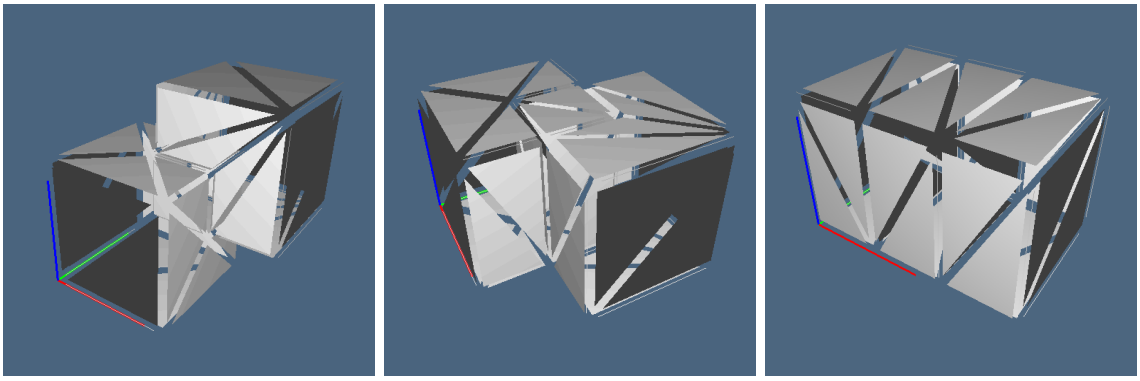

Figure 1: The triangulated boundaries of the space partition induced by two cubes (one is variously translated).

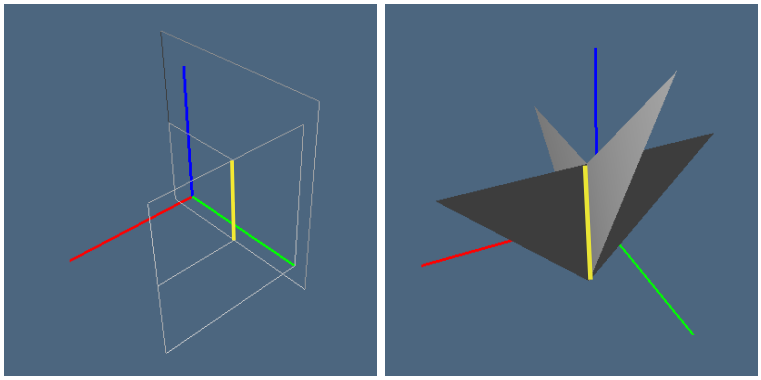

Figure 2: The triangles around an edge: `VIEW(STRUCT(MKPOLS((W,ET[35]))))`.

**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 12 ⟩ ≡

```
    """ Circular ordering of faces around edges """

    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,FE):
        V,FV,EV = model
        VIEW(COLOR(YELLOW)(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,EV)))))
        triangleSet = boundaryTriangulation(V,FV,EV,FE) # corrected with non-contractible faces
        VIEW(EXPLODE(1.2,1.2,1.2)(AA(JOIN)( AA(POLYLINE)(CAT(triangleSet)) )))
        VIEW(EXPLODE(1.2,1.2,1.2)( AA(POLYLINE)(AA(lambda tri: tri+[tri[0]])(CAT(triangleSet))) ))
        TV,FT = triangleIndices(triangleSet,V)
        TE = crossRelation(len(V),TV,EV)
        ET,ET_angle = invertRelation(TE),[]
        #import pdb; pdb.set_trace()
        for e,et in enumerate(ET):
            v1,v2 = EV[e]
            v1v2 = set([v1,v2])
            et_angle = []
            t0 = et[0]
            tverts = [v1,v2] + list(set(TV[t0]).difference(v1v2))
            e3 = UNITVECT(VECTDIFF([ V[tverts[1]], V[tverts[0]] ]))
```

```
            e1 = UNITVECT(VECTDIFF([ V[tverts[2]], V[tverts[0]] ]))
            e2 = cross(array(e1),e3).tolist()
            basis = mat([e1,e2,e3]).T
            transform = basis.I
            normals = []
            Tvs = []
            for triangle in et:
                verts = TV[triangle]
                vertSet = set(verts).difference(v1v2)
                tvs = [v1,v2] + list(vertSet)
                Tvs += [tvs]
                w1 = UNITVECT(VECTDIFF([ V[tvs[2]], V[tvs[0]] ]))
                w2 = (transform * mat([w1]).T).T
                w3 = cross(array([0,0,1]),w2).tolist()[0]
                normals += [w3]
            normals = mat(normals)
            for k,t in enumerate(et):
                angle = math.atan2(normals[k,1],normals[k,0])
                et_angle += [angle]
            pairs = sorted(zip(et_angle,et,Tvs))
            sortedTrias = [pair[1] for pair in pairs]
            triasVerts = [pair[2] for pair in pairs]
            ET_angle += [sortedTrias]
        EF_angle = ET_to_EF_incidence(TV,FV,FT, ET_angle)
        return EF_angle, ET,TV,FT
```

◇

Macro referenced in 23.

**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 13 ⟩ ≡

```
    """ Edge-triangles to Edge-faces incidence """
    def ET_to_EF_incidence(TW,FW,FT, ET_angle):
        tableFT = FT
        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 23.

**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.

### 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 14 ⟩ ≡

```
""" Cycles orientation """
def cyclesOrient(pcycles,fcycle,EV):
    if set(AA(ABS)(pcycles)).difference(fcycle)==set(): return []
    ofcycle,_ = boundaryCycles(fcycle,EV) # oriented
    ofcycle = ofcycle[0] # oriented
    if type(pcycles[0])==list: opcycle = CAT(pcycles)
    else: opcycle = pcycles
    int = set(opcycle).intersection(ofcycle)
    if int != set():
        ofcycle = 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 23.

**3-cell traversal** The following script contains a completely reviewed and rewritten version of the extraction of 3-cells from its 2-skeleton embedded in 3D. The number of boundary edges of general LAR 2-faces is given by `len(EF_angle)` or by `len(ET)`. The array `len(EV)` has length `len(EF_angle)` + the number of added edges due to the triangulation of original faces. The added edges are not returned inside `FE` and `EF_angle`.

⟨ 3-cell traversal 15a ⟩ ≡

◇

Macro referenced in 23.

**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 15b ⟩ ≡

```
    """ Cells from $(d-1)$-dimensional LAR model """
    def facesFromComponents(model,FE,EF_angle):

        accumulated = []
        def viewStep (CF,CV,CE,COE,accumulated):
            VV = AA(LIST)(range(len(V)))
            edges = list(set(CE[-1]).difference(accumulated))
            accumulated = CE[-1]
            submodel = STRUCT(MKPOLS((V,[EV[k] for k in edges])))
            VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,1))

        print "\nECCOMI\n"
        # initialization
```

```
    V,FV,EV = model
    visitedCell = [[ None, None ] for k in range(len(FV)) ]
    face = 0
    boundaryLoop,_ = boundaryCycles(FE[face],EV)
    boundaryLoop = boundaryLoop[0]
    firstEdge = boundaryLoop[0]
    #import pdb; pdb.set_trace()
    cf,coe = getSolidCell(FE,face,visitedCell,boundaryLoop,EV,EF_angle,V,FV)
    for face,edge in zip(cf,coe):
        if visitedCell[face][0]==None: visitedCell[face][0] = edge
        else: visitedCell[face][1] = edge
    cv,ce = set(),set()
    cv = cv.union(CAT([FV[f] for f in cf]))
    ce = ce.union(CAT([FE[f] for f in cf]))
    CF,CV,CE,COE = [cf],[list(cv)],[list(ce)],[coe]
    viewStep (CF,CV,CE,COE,accumulated)

    # main loop
    #import pdb; pdb.set_trace()
    while True:
        face, edge = startCell(visitedCell,FE,EV)
        print "face, edge =",face, edge
        if face == -1: break
        boundaryLoop,_ = boundaryCycles(FE[face],EV)
        boundaryLoop = boundaryLoop[0]
        if edge not in boundaryLoop:
            boundaryLoop = reverseOrientation(boundaryLoop)
        cf,coe = getSolidCell(FE,face,visitedCell,boundaryLoop,EV,EF_angle,V,FV)
        CF += [cf]
        COE += [coe]
        for face,edge in zip(cf,coe):
            if visitedCell[face][0]==None: visitedCell[face][0] = edge
            else: visitedCell[face][1] = edge

        cv,ce = set(),set()
        cv = cv.union(CAT([FV[f] for f in cf]))
        ce = ce.union(CAT([FE[f] for f in cf]))
        CV += [list(cv)]
        CE += [list(ce)]
        viewStep (CF,CV,CE,COE,accumulated)
    return V,CV,FV,EV,CF,CE,COE
```

◇

Macro referenced in 23.

**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 16 ⟩ ≡

```
""" Start a new 3-cell """
def startCell(visitedCell,FE,EV):
    if len([term for cell in visitedCell for term in cell if term==None])==1: return -1,-1
    for face in range(len(visitedCell)):
        if len([term for term in visitedCell[face] if term==None])==1:
            edge = visitedCell[face][0]
            break
        else: pass  #TODO: implement search for isolated shells
        face,edge = -1,-1
    return face,edge
```

◇

Macro referenced in 23.

**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 17 ⟩ ≡

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

def faceOrientation(boundaryLoop,face,FE,EV,cf):
    theBoundary = set(AA(ABS)(boundaryLoop))
    if theBoundary.intersection(FE[face])==set() and theBoundary.difference(FE[face])!=set(): #
        coboundaryFaces = [f for f in cf if set(FE[f]).intersection(theBoundary)!=set()]
        face = coboundaryFaces[0]
    faceLoop,_ = boundaryCycles(FE[face],EV)
    faceLoop = faceLoop[0]
    commonEdges = set(faceLoop).intersection(boundaryLoop)
```

```
        if commonEdges == set() or commonEdges == {0}:
            faceLoop = reverseOrientation(faceLoop)
            commonEdges = set(faceLoop).intersection(boundaryLoop)
        theEdge = list(commonEdges)[0]
        #if theEdge==0: theEdge = list(commonEdges)[1]
        return -theEdge,face
```
◇

Macro referenced in 23.

### Get single solid cell

⟨ Get single solid cell 18a ⟩ ≡

```
"""" Get single solid cell """
def getSolidCell(FE,face,visitedCell,boundaryLoop,EV,EF_angle,V,FV):

    def orientFace(face,boundaryLoop):
        for e in boundaryLoop:
            if ABS(e) in FE[face]: return -e

    coe = [orientFace(face,boundaryLoop)]
    cf = [face]
    #import pdb; pdb.set_trace()
    while boundaryLoop != []:
        edge,face = faceOrientation(boundaryLoop,face,FE,EV,cf)
        if edge > 0: edgeFaces = EF_angle[edge]
        elif edge < 0: edgeFaces = REVERSE(EF_angle[-edge])
        e = ABS(edge)
        n = len(edgeFaces)
        ind = (edgeFaces.index(face)+1)%n
        nextFace = edgeFaces[ind]
        coe += [-orientFace(nextFace,boundaryLoop)]
        boundaryLoop = cyclesOrient(boundaryLoop,FE[nextFace],EV)
        cf += [nextFace]
        face = nextFace
        VIEW(STRUCT( MKPOLS((V,[EV[h] for f in cf for h in FE[f]])) )) #add EV!
    return cf,coe
```
◇

Macro referenced in 23.

**Double check the faces boundaries made of edges** Let us notice that a sistematic use of the `FE` relation to compute the edges on the boundary of a face is *not* reliable when the faces are non-convex (WHY ??). A better solution is to double-check the result `FE[f]` when `len(FE[f]) > len(FV[f])`, in order to filter out the spurious edges ... The function below works with the precondition that vertices in `FV[f]` are spatially ordered along the face boundary.

⟨ Double check the faces boundaries made of edges 18b ⟩ ≡

```
""" Double check the faces boundaries made of edges """
def doubleCheckFaceBoundaries(FE,V,FV,EV):
    FEout = []
    for f,face in enumerate(FE):
        n = len(FV[f])
        if len(FE[f]) > n:
            # contains both edges coded 0 and 1 ... (how to solve?)
            FEout += [list(set(face).difference([0]))]
        else:
            FEout += [face]
    return FEout
```

◇

Macro referenced in 19.

### 3.5.1 Main procedure of arrangement partitioning

⟨ Main procedure of arrangement partitioning 19 ⟩ ≡

```
""" Main procedure of arrangement partitioning """

⟨ Double check the faces boundaries made of edges 18b ⟩

def thePartition(W,FW,EW):
    quadArray = [[W[v] for v in face] for face in FW]
    parts = boxBuckets3d(containmentBoxes(quadArray))
    Z,FZ,EZ = spacePartition(W,FW,EW, parts)
    Z,FZ,EZ = larSimplify((Z,FZ,EZ),radius=0.0001)
    model = Z,FZ,EZ

    ZZ = AA(LIST)(range(len(Z)))
    submodel = STRUCT(MKPOLS((Z,EZ)))
    VIEW(larModelNumbering(1,1,1)(Z,[ZZ,EZ,FZ],submodel,0.6))

    FE = crossRelation(len(Z),FZ,EZ) ## to be double checked !!
    EF_angle, ET,TV,FT = faceSlopeOrdering(model,FE)

    V,CV,FV,EV,CF,CE,COE = cellsFromComponents((Z,FZ,EZ),FE,EF_angle, ET,TV,FT)
    V,CV,FV,EV,CF,CE,COE = facesFromComponents((Z,FZ,EZ),FE,EF_angle)
    return V,CV,FV,EV,CF,CE,COE,FE
```

◇

Macro referenced in 23.

## 3.6 Signed (co)boundary operator with non-contractible 2-cells

In this section we construct the matrix of the *signed* (i.e. oriented) boundary operator—with elements in $\{-1, 0, 1\}$—in order to traverse efficiently the partial boundary of the

(coherently oriented) 3-cells to be reconstructed from a 2-skeleton embedded in 3D.

We start computing the 2-boundary / 2-coboundary using a triangulation of general LAR faces. The original LAR edges are uncoupled from those added by the triangulation algorithm. For the generation of the boundary triangulation matrix we do not not use any geometric embedding (i.e. sign of determinants induced by vertex positions) but the standard sign rules of abstract simplicial complexes.

In particular, the 2-boundary matrix in the more general LAR cell setting is a contraction of a block-decomposition of the 2-boundary matrix built for the triangulated space. We cannot test for the 3-case because I do not have (for now :-) a tetrahedral CDT to apply to LAR 3-cells, but the method should extend naturally to higher dimensions.

**Extend LAR edges with a given (2D) triangulation** The function `extendEV` below extends the LAR edges, contained within the `EV` array, with the new arcs introduced by a given (2D) triangulation. The `TV` array contains, for each triangle, the list of its three vertex indices. The `ET` array, gives, for each triangle in the original edge array, the indices of incident triangles, already computed by the triangulation algorithm, invoked (please see the `test/py/bool/test16.py` example) by the `faceSlopeOrdering` function. The function return the new array of edges, as lists of vertex indices. The first $n$ pairs are the original edges; the following ones are those added bu the triangulation. The dictionary `EVdict` is used to construct the returned output.

⟨ Extend LAR edges with a given (2D) triangulation 20 ⟩ ≡

```
""" Extend LAR edges with a given (2D) triangulation """
from collections import OrderedDict

def extendEV(EV,ET,TV):
    EVdict = OrderedDict([(edge,k) for k,edge in enumerate(EV)])
    n = len(EV)-1
    for e,(u,w) in enumerate(EV):
        for t in ET[e]:
            v1,v2,v3 = TV[t]
            v = list({v1,v2,v3}.difference([u,w]))[0]
            if u<v: newEdge = (u,v)

            else: newEdge = (v,u)
            if not newEdge in EVdict:
                n += 1
                EVdict[newEdge]=n

            if w<v: newEdge = (w,v)
            else: newEdge = (v,w)
            if not newEdge in EVdict:
                n += 1
```

```
                    EVdict[newEdge]=n
        return EVdict.keys()
```
◇

Macro referenced in 23.

**Signed boundary operator for a general LAR 2-complex** The function `larSignedBoundary` is our current implementation for the construction of the $p \times q$ matrix of the signed (i.e. oriented) boundary operator of the general LAR 2-complex in `larModel`, starting from the $n \times m$ signed matrix of the LAR triangulation in `triaModel`. The `FT` array provides, for each original LAR face, the list of triangle indices (to be used, for example, in `TV`) of its triangulation.

The `output` matrix is actually computed by first constructing the *unsigned* `boundary` operator, and then by updating its values by those provided by the contraction of the `input` columns corresponding to the subset of triangles associated to each original face.

No actual sum is necessary for the contraction of a set of `input` column to a single `output` column, because the nonzero elements corresponding to original edges are always located on different `input` rows. Conversely, the `input` rows corresponding to the edges added by the triangulation, located in the lower block of the `input` matrix, contain both one -1 term and one +1 term, that mutually cancel, so that their whole block can be implicitly removed from the `output` matrix.

⟨ Signed boundary operator for a general LAR 2-complex 21a ⟩ ≡

```
""" Signed boundary operator for a general LAR 2-complex """
def larSignedBoundary(larModel,triaModel,FT):
    input = signedSimplicialBoundary(*triaModel)
    output = boundary(*larModel)
    (n,m),(p,q) = input.shape, output.shape

    for h in range(p):
        for k,triangles in enumerate(FT):
            lo,hi = triangles[0], triangles[-1]
            val = [input[h,t] for t in range(lo,hi+1)  if input[h,t]!=0]
            if val!=[]: output[h,k] = val[0]
    return output
```
◇

Macro referenced in 23.

**Generation of signed boundary operator of a general LAR complex** A complete example of computation of the signed boundary operator of a general LAR complex is given in the `test/py/bool/test16.py`. Notice that the last two faces in `FV` are not contractible to a point, and contain two boundary cycles (island with a lake). The LAR model is the one generated by the `test/py/bool/test15.py` example (see).

"test/py/bool/test16.py" 21b ≡

```
""" Generation of signed boundary operator of a general LAR complex """
from larlib import *

V = [[0.25,0.25,-5e-05],[0.25,0.75,-5e-05],[0.75,0.75,-5e-05],[0.75,0.25,-5e-05],[1.
0,0.0,0.0],[0.0,0.0,0.0],[1.0,1.0,0.0],[0.0,1.0,0.0],[0.25,0.25,1.0],[0.25,0.25,
2.0],[0.25,0.75,2.0],[0.25,0.75,1.0],[0.25,0.75,-1.0],[0.25,0.25,-1.0],[0.75,0.
75,-1.0],[0.75,0.25,-1.0],[0.75,0.25,1.0],[0.75,0.75,1.0],[1.0,0.0,1.0],[0.0,0.0
,1.0],[1.0,1.0,1.0],[0.0,1.0,1.0],[0.75,0.75,2.0],[0.75,0.25,2.0]]

FV = [(2,3,16,17),(6,7,20,21),(12,13,14,15),(0,1,8,11),(1,2,11,17),(0,1,12,13),
(4,6,18,20),(5,7,19,21),(0,3,13,15),(0,3,8,16),(0,1,2,3),(10,11,17,22),(2,3,14,
15),(8,9,16,23),(8,11,16,17),(1,2,12,14),(16,17,22,23),(4,5,18,19),(8,9,10,11),(
9,10,22,23),(0,1,2,3,4,5,6,7),(8, 11,16,17,18,19,20,21)]

EV =[(3,15),(7,21),(10,11),(4,18),(12,13),(5,19),(8,9),(18,19),(22,23),(0,3),(1,11),
(16,17),(0,8),(6,7),(20,21),(3,16),(10,22),(18,20),(19,21),(1,2),(12,14),(4,5),(
8,11),(13,15),(16,23),(14,15),(11,17),(17,22),(2,14),(2,17),(0,1),(9,10),(8,16),
(4,6),(1,12),(5,7),(0,13),( 9,23),(6,20),(2,3)]

VV = AA(LIST)(range(len(V)))
submodel = SKEL_1(STRUCT(MKPOLS((V,EV))))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,0.6))

FE = crossRelation(len(V),FV,EV)
EF_angle, ET,TV,FT = faceSlopeOrdering((V,FV,EV),FE)
EW = extendEV(EV,ET,TV)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,TV))))

submodel = SKEL_1(STRUCT(MKPOLS((V,EW))))
VIEW(larModelNumbering(1,1,1)(V,[VV,EW,TV],submodel,0.6))

triaModel, larModel = (TV,EW), (FV,EV)
op = larSignedBoundary(larModel,triaModel,FT)
print op.todense()
```

◇

**The output matrix** The output matrix of the signed boundary operator follows.

⟨ Matrix of the signed boundary operator for a general LAR 2-complex 22 ⟩ ≡

```
""" (see test/py/bool/test16) """
[[ 0  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0]
 [ 0  1  0  0  0  0  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  1  0  0  0  0  0  0  1  0  0  0]
 [ 0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0  0 -1  0  0  0]
 [ 0  0  1  0  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 -1  0  0  0  0  0  0  0  0  0  1  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0  0  1]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  1  1 -1  0  0  0  0  0  0  0  0  0  1  0]
[ 0  0  0  1 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 1  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  1  0  0  0  0  1]
[ 0  0  0 -1  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0]
[ 0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1]
[ 1  0  0  0  0  0  0  0  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 -1  0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1]
[ 0  0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0  1]
[ 0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0 -1  0]
[ 0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0]
[ 0  0  0 -1  0  0  0  0  0  0  0  0  0  0  1  0  0  0 -1  0  0 -1]
[ 0  0  1  0  0  0  0  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  0 -1  0  0 -1  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  0  0  0  0  0  0  1  0  0  1  0  0  0  0  0  0 -1]
[ 0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0  1  0  0  0  0  0  0]
[-1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0  0 -1  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  0  0]
[ 0  0  0  0  0  0  0  0  0 -1  0  0  0 -1 -1  0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0]
[ 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0 -1  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  1  0]
[ 0  0  0  0  0 -1  0  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  0  1  0  0  0  0  0 -1  0  0]
[ 0 -1  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 1  0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  0  0 -1  0]]
◇
```

Macro never referenced.

### 3.7 Boolean chains

# 4 Esporting the Library

"larlib/larlib/bool.py" 23 ≡

```
""" Module for Boolean computations between geometric objects """
from larlib import *
```

```
from copy import copy
DEBUG = False
```

⟨ Coding utilities 35d ⟩
⟨ Remove subsets from bucket list 2 ⟩
⟨ Iterate the splitting until splittingStack is empty 3 ⟩
⟨ Computation of affine face transformations 5 ⟩
⟨ Submanifold mapping computation 4a ⟩
⟨ Helper functions for spacePartition 4b ⟩
⟨ Sorting of points along a line 6a ⟩
⟨ Remove intersection points external to a submanifold face 6b ⟩
⟨ Space partitioning via submanifold mapping 7 ⟩
⟨ 3D boundary triangulation of the space partition 9 ⟩
⟨ Computation of incidence between edges and 3D triangles 10 ⟩
⟨ Slope of edges 12 ⟩
⟨ Edge-triangles to Edge-faces incidence 13 ⟩
⟨ 3-cell traversal 15a ⟩
⟨ Cells from $(d-1)$-dimensional LAR model 15b ⟩
⟨ Cycles orientation 14 ⟩
⟨ Start a new 3-cell 16 ⟩
⟨ Face orientations storage 17 ⟩
⟨ Extend LAR edges with a given (2D) triangulation 20 ⟩
⟨ Signed boundary operator for a general LAR 2-complex 21a ⟩
⟨ Get single solid cell 18a ⟩
⟨ Main procedure of arrangement partitioning 19 ⟩
◇

# 5 Test examples

## 5.1 Random triangles

### Generation of random triangles and their boxes

"test/py/bool/test01.py" 24 ≡

```
""" Generation of random triangles and their boxes """
from larlib 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/bool/test02.py" 25a ≡

```
""" Generation of random quadrilaterals and their boxes """
from larlib 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]))
◇
```

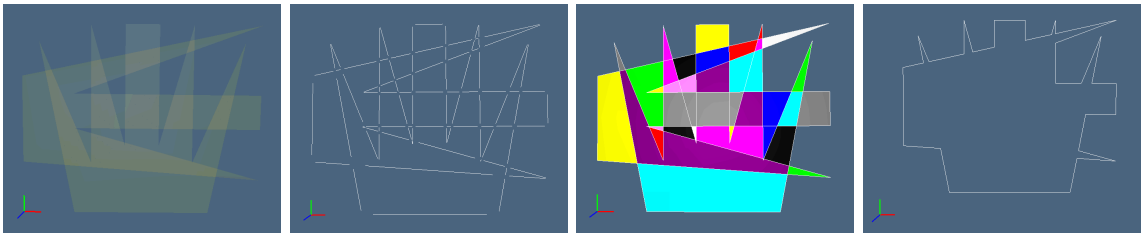

Figure 3: LAR complex from two polygons. (a) the input polygons; (b) the intersection of boundary lines; (c) the extracted *regularized* 2-complex; (d) the boundary LAR.

"test/py/bool/test03.py" 25b ≡

```
""" Boolean complex generated by boundaries of two complexes """
from larlib 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,polygons = 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/bool/test04.py" 26 ≡

```
""" Visualizing with different colors the buckets of box-kd-tree """
from larlib import *

randomQuadArray = randomQuads(30,0.8)
VIEW(STRUCT(AA(MKPOL)([[verts, [[1,2,3,4]], None] for verts in randomQuadArray])))

V,[VV,EV,FV,CV] = larCuboids([2,2,1],True)
cubeGrid = Struct([(V,FV,EV)],"cubeGrid")
cubeGrids = Struct(2*[cubeGrid,s(.5,.5,.5)])
V,FV,EV = struct2lar(cubeGrids)
boxes = containmentBoxes([[V[v] for v in f] for f in FV])
VV = AA(LIST)(range(len(V)))
```

```
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,0.6))
parts = boxBuckets3d(boxes)
V,FV,EV = spacePartition(V,FV,EV, parts)
VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,0.6))


#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 = boxBuckets3d(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]))]
    if k==30: VIEW(STRUCT(bunch))
```

◇

## 5.3 Intersection of geometry subsets

### Two unit cubes

⟨ Two unit cubes 27 ⟩ ≡

```
""" Two unit cubes """
from larlib import *
V,[VV,EV,FV,CV] = larCuboids([2,2,2],True)
cube1 = Struct([(V,FV,EV)],"cube1")
twoCubes = Struct([cube1,t(.5,.5,.5),cube1])

glass = MATERIAL([1,0,0,0.1,  0,1,0,0.1,  0,0,1,0.1, 0,0,0,0.1, 100])

#twoCubes = Struct([cube1,t(-1,.5,1),cube1])     # other test example
#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 = boxBuckets3d(boxes)
◇
```

Macro referenced in 28ab.

def POLYGONS((V,FV)):

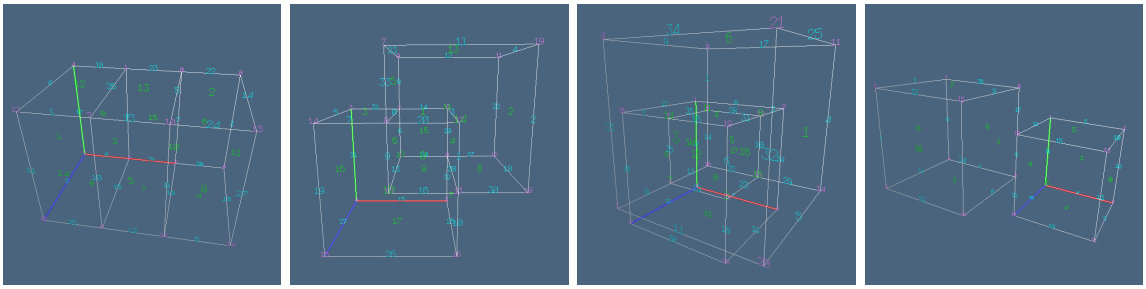

Figure 4: LAR complex of the space decomposition generated by two cubes in special positions. (a) translation on one coordinate; (b) translation on two coordinates; (c) translation on three coordinates; (d) non-manifold position along an edge.

### Face (and incident faces) transformation

"test/py/bool/test05.py" 28a ≡

```
""" non-valid -> valid solid representation of a space partition """
from larlib import *

⟨ Two unit cubes 27 ⟩

W,FW,EW = spacePartition(V,FV,EV, parts)
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/bool/test06.py" 28b ≡

```
""" 3-cell reconstruction from LAR space partition """
from larlib import *
⟨ Two unit cubes 27 ⟩
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.

"test/py/bool/test07.py" 29a ≡

```
""" 2D polygon triangulation """
from larlib import *
#from support import PolygonTessellator,vertex

filename = "test/svg/bool/interior.svg"
lines = svg2lines(filename)
V,FV,EV,polygons = larFromLines(lines)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,FV[:-1]+EV)) + AA(MK)(V)))

VIEW(EXPLODE(1.2,1.2,1)(MKTRIANGLES((V,FV,EV)) ))
◇
```

### From triples of points to LAR model of boundary triangulation

"test/py/bool/test08.py" 29b ≡

```
import sys
from larlib import *

sys.path.insert(0, 'test/py/bool/')
from test06 import *

""" From triples of points to LAR model """
FE = crossRelation(len(W),FW,EW)
triangleSet = boundaryTriangulation(W,FW,EW,FE)
TW,FT = 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/bool/test09.py" 29c ≡

```
""" Visualization of of incidence between edges and 3D triangles """
import sys
from larlib import *

sys.path.insert(0, 'test/py/bool/')
```

```
from test08 import *

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

triangleSet = boundaryTriangulation(W,FW,EW,FE)
TW,FT = 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 larlib.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/bool/test10.py" 30a ≡

```
""" Visualization of indices of the boundary triangulation """
from larlib import *

sys.path.insert(0, 'test/py/bool/')
from test09 import *

model = W,FW,EW
FE = crossRelation(len(W),FW,EW)
EF_angle, ET,TV = faceSlopeOrdering(model,FE)

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/bool/test11a.py" 30b ≡

```
from larlib import *
⟨ testing example (30c t(.5,.5,.5),r(0,0,PI/6) ) 31m ⟩
◇
```

"test/py/bool/test11b.py" 30d ≡

```
from larlib import *
```
⟨ testing example (30e `t(.5,.5,0),r(0,0,PI/6)` ) 31m ⟩
◇

"test/py/bool/test11c.py" 31a ≡

```
from larlib import *
```
⟨ testing example (31b `t(.5,0,0),r(0,0,PI/6)` ) 31m ⟩
◇

"test/py/bool/test11d.py" 31c ≡

```
from larlib import *
```
⟨ testing example (31d `t(0,0,0),r(0,0,PI/6)` ) 31m ⟩
◇

"test/py/bool/test11e.py" 31e ≡

```
from larlib import *
```
⟨ testing example (31f `s(.5,.5,.5),r(0,0,PI/6)` ) 31m ⟩
◇

"test/py/bool/test11f.py" 31g ≡

```
from larlib import *
```
⟨ testing example (31h `t(.25,.25,.25),s(.5,.5,.5),r(0,0,PI/6)` ) 31m ⟩
◇

"test/py/bool/test11g.py" 31i ≡

```
from larlib import *
```
⟨ testing example (31j `t(.25,.25,.75),s(.5,.5,.5),r(0,0,PI/6)` ) 31m ⟩
◇

"test/py/bool/test11h.py" 31k ≡

```
from larlib import *
```
⟨ testing example (31l `t(1.5,1.5,0),r(0,0,PI/6)` ) 31m ⟩
◇

⟨ testing example 31m ⟩ ≡

```
""" Visualization of indices of the boundary triangulation """

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True)
cubeGrid = Struct([(V,FV,EV)],"cubeGrid")
```

```
cubeGrids = Struct(2*[cubeGrid,@1])

V,FV,EV = struct2lar(cubeGrids)
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((V,FV))))
V,CV,FV,EV,CF,CE,COE,FE = thePartition(V,FV,EV)

cellLengths = AA(len)(CF)
boundaryPosition = cellLengths.index(max(cellLengths))
BF = CF[boundaryPosition]; del CF[boundaryPosition]; del CE[boundaryPosition]
BE = list({e for f in BF for e in FE[f]})

#Volume((V,[FV[f] for f in CF[0]]))

VIEW(EXPLODE(1.2,1.2,1.2)( MKTRIANGLES((V,[FV[f] for f in BF],[EV[e] for e in BE])) ))
VIEW(EXPLODE(1.2,1.2,1.2)([ MKSOLID(V,[FV[f] for f in cell],[EV[e] for e in set(CAT([FE[f] for
VIEW(EXPLODE(1.2,1.2,1.2)([STRUCT(MKPOLS((V,[EV[e] for e in cell]))) for cell in CE]))
◇
```

Macro referenced in 30bd, 31acegik.

## 5.4 Polygon triangulation

"test/py/bool/test13.py" 32a ≡

```
""" Polygon triangulation and importing to LAR """

from larlib import *
from p2t import *

⟨ load points 32b ⟩
⟨ input polyline visualization 34a ⟩
⟨ CDT triangulation with poly2tri 34b ⟩
⟨ conversion of triangulation to LAR 34c ⟩
⟨ visualization of LAR model of triangulation 34d ⟩
⟨ reconstruction of boundary polyline for LAR model 35a ⟩
⟨ visualization of LAR generated boundary polyline 35b ⟩
◇
```

### load points

⟨ load points 32b ⟩ ≡

```
""" load points """
def load_points(file_name):
    infile = open(file_name, "r")
    points = []
    while infile:
        line = infile.readline()
```

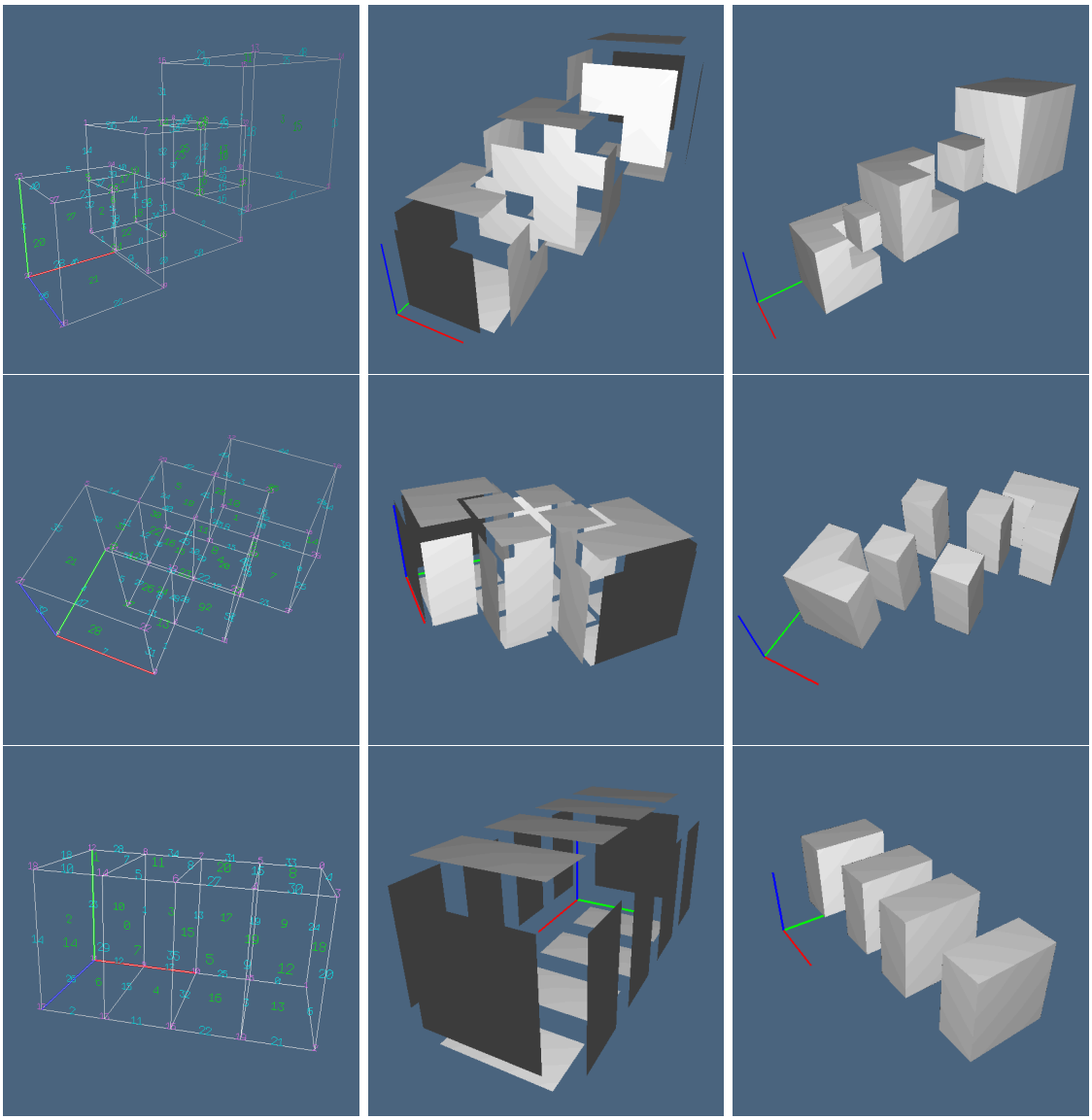

Figure 5: Examples of 3-cell extraction of two simple Boolean 2-complex, and their boundaries. Notice the numbers of (solid) 3-cells.

```
            s = line.split()
            if len(s) == 0:
                break
            points.append([float(s[0]), float(s[1])])
        return points
```
◇

Macro referenced in 32a.

### input polyline visualization

⟨ input polyline visualization 34a ⟩ ≡
```
""" input polyline visualization """
points = load_points("test/data/nazca_monkey.dat")
VIEW(POLYLINE(points))
```
◇

Macro referenced in 32a.

### CDT triangulation with poly2tri

⟨ CDT triangulation with poly2tri 34b ⟩ ≡
```
""" CDT triangulation with poly2tri """
polyline = [Point(p[0],p[1]) for p in points]
cdt = CDT(polyline)
triangles = cdt.triangulate()
```
◇

Macro referenced in 32a.

### conversion of triangulation to LAR

⟨ conversion of triangulation to LAR 34c ⟩ ≡
```
""" conversion of triangulation to LAR """
trias = [ [[t.a.x,t.a.y],[t.b.x,t.b.y],[t.c.x,t.c.y]] for t in triangles ]
vdict = defaultdict(list)
for k,point in enumerate(CAT(trias)):
    vdict[vcode(point)] += [k]
vdict = OrderedDict(zip(vdict.keys(),range(len(vdict.keys()))))
FV = [(vdict[vcode(a)], vdict[vcode(b)], vdict[vcode(c)]) for [a,b,c] in trias]
repeatedEdges = CAT([[[v1,v2],[v2,v3],[v3,v1]] for [v1,v2,v3] in FV])
EV = list(set(AA(tuple)(AA(sorted)(repeatedEdges))))
V = [eval(vect) for vect in vdict]
```
◇

Macro referenced in 32a.

### visualization of LAR model of triangulation

⟨ visualization of LAR model of triangulation 34d ⟩ ≡

```
""" visualization of LAR model of triangulation """
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,FV))))
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,EV))))
◇
```

Macro referenced in 32a.

### reconstruction of boundary polyline for LAR model

⟨ reconstruction of boundary polyline for LAR model 35a ⟩ ≡

```
""" reconstruction of boundary polyline for LAR model """
EW = boundaryCells(FV,EV)
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,[EV[e] for e in EW]))))
model = (V,FV,[EV[e] for e in EW])
struct = Struct([model])
◇
```

Macro referenced in 32a.

### visualization of LAR generated boundary polyline

⟨ visualization of LAR generated boundary polyline 35b ⟩ ≡

```
""" visualization of LAR generated boundary polyline """
poly = boundaryModel2polylines(structBoundaryModel(struct))
VIEW(POLYLINE(poly[0][:-1]))
◇
```

Macro referenced in 32a.

### Arrangements with non-contractible cells

"test/py/bool/test15.py" 35c ≡

```
""" Arrangements with non-contractible cells """
from larlib import *

V,[VV,EV,FV,CV] = larCuboids([1,1,1],True)
cube1 = Struct([(V,FV,EV)],"cube1")
cube2 = Struct([t(.25,.25,-1),s(.5,.5,3),(V,FV,EV)],"cube2")
V,FV,EV = struct2lar(Struct([cube1,cube2]))
VIEW(STRUCT(MKPOLS((V,FV,EV))))
"""
V,CV,FV,EV,CF,CE,COE,FE = thePartition(V,FV,EV)
"""
◇
```

# A Code utilities

## A.1 Generation of random data

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

⟨ Coding utilities 35d ⟩ ≡

```
""" Coding utilities """
global count
⟨ Generation of a random 3D point 37c ⟩
⟨ Generation of random 3D triangles 37a ⟩
⟨ Generation of random 3D quadrilaterals 37b ⟩
⟨ Generation of a single random triangle 37d ⟩
⟨ Containment boxes 38a ⟩
⟨ Transformation of a 3D box into an hexahedron 38b ⟩
⟨ Generation of a list of HPCs from a LAR model with non-convex faces 36 ⟩
◇
```

Macro referenced in 23.

### Generation of a list of HPCs from a LAR model with non-convex faces

⟨ Generation of a list of HPCs from a LAR model with non-convex faces 36 ⟩ ≡

```
""" Generation of a list of HPCs from a LAR model with non-convex faces """
def MKTRIANGLES(model):
    V,FV,EV = model
    FE = crossRelation(len(V),FV,EV)
    if len(V[0]) == 2: V=[v+[0] for v in V]
    triangleSets = boundaryTriangulation(V,FV,EV,FE)
    return [ STRUCT([MKPOL([verts,[[3,2,1]],None]) for verts in triangledFace])
        for triangledFace in triangleSets ]

def MKSOLID(*model):
    V,FV,EV = model
    FE = crossRelation(len(V),FV,EV)
    pivot = V[0]
    #VF = invertRelation(FV)
    #faces = [face for face in FV if face not in VF[0]]
    faces = [face for face in FV]
    triangleSets = boundaryTriangulation(V,faces,EV,FE)
    return XOR([ MKPOL([face+[pivot], [range(1,len(face)+2)],None])
        for face in CAT(triangleSets) ])
◇
```

Macro referenced in 35d.

**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 37a ⟩ ≡

```
""" 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 35d.

### Generation of random 3D quadrilaterals

⟨ Generation of random 3D quadrilaterals 37b ⟩ ≡

```
""" 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 35d.

**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 `rpoint3d()` function, with no input parameters.

⟨ Generation of a random 3D point 37c ⟩ ≡

```
""" Generation of a random 3D point """
def rpoint3d():
    return eval( vcode([ random.random(), random.random(), random.random() ]) )
```

◇

Macro referenced in 35d.

**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 37d ⟩ ≡

```
""" Generation of a single random triangle """
def rtriangle(scaling):
    v1,v2,v3 = array(rpoint3d()), array(rpoint3d()), array(rpoint3d())
    c = (v1+v2+v3)/3
    pos = rpoint3d()
    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 35d.

**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 38a ⟩ ≡

```
""" 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 35d.

**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 38b ⟩ ≡

```
""" 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 35d.

## 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.