Triangulation of the boundary of a 2-chain *

Alberto Paoluzzi

December 15, 2015

Abstract

In this module we perform the whole set of manipulations necessary to triangulate the boundary polygon of a 2-chain using a standard triangulation algorithm [] implemented by the `poly2tri` python package, largely diffuse in games applications. Notice that the input data may be of very general kind, corresponding to polygons that are non-connected, non-manifold w/o nested polygons of general kind. Conversely, the used library allows only for quite special polygons, with a simple exterior boundary and multiple (non-touching) internal holes, and without repeated vertices. The used package is based on the paper "Sweep-line algorithm for constrained Delaunay triangulation" by V. Domiter and and B. Zalik [D08].

Contents

*This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. December 15, 2015

1 Introduction

Let suppose that a 2-complex is given by a LAR triple `(V,FV,EV)` with the standard characteristics, and that a 2-chain `chain` $\in \mathbb{Z}_2^n$ is also given, with $\mathbb{Z}_2 = \{0,1\}$ and $n = |\texttt{FV}|$.

2 Implementation

2.1 Containments between non intersecting cycles

In this section we compute the containment relation between non-intersecting cycles generated on 2-faces by the incident faces. This step is preparatory to the representation of fragmented 2-faces — embedded in 2D — as LAR data structures, to be subsequently restored in the ambient 3D and sticked together to generate the the 2-skeleton of the Boolean complex.

For this purpose, the set of non-intersecting cycles, as lists of edges, are returned in the `EVs` array by the `biconnectedComponent` function. The pair `V,EVs` is therefore passed as input to the `latticeArray` function, that returns a dense matrix with elements in $\{-1,0,1\}$, where the element i,j either contains 0 if anyone (and hence all) of vertices of cycle j-th is *external* to cycle i-th, or contains 1 if anyone (and hence all) of vertices of cycle j-th is *internal* to cycle i-th, or finally contains -1 if anyone (and hence all) of vertices of cycle j-th is *on boundary* of cycle i-th. The last condition may hold only for diagonal elements i,j where $i = j$.

The returned `testArray` matrix of dimension $n \times n$, where n is the number of non-intersecting cycles on a fragmented 2D face, is the used by the `cellsFromCycles` function, that return a list of lists of cycles, each one defining the boundary of a single connected but possibly non path-connected 2-cell in the LAR representation of the fragmented 2-complex.

Classification of non intersecting cycles The function `latticeArray` takes as input the pair `V,EVs`, where V is the list of vertices and `EVs` is the list of cycles of a fragmented 2-cell in 2D. Each `EVs` element is given as a list of edges, given os pairs of integer vertex indices. The returned `testArray` matrix — characterizing the incidences between cycles — has dimension $n \times n$.

Remark on interior point classification

⟨Classification of non intersecting cycles 2⟩ ≡

```
""" Classification of non intersecting cycles """
def latticeArray(V,EVs):
    n = len(EVs)
    testArray = []
    for k,ev in enumerate(EVs):
        row = []
```

```
        classify = pointInPolygonClassification((V,ev))
        for h in range(0,n):
            i = EVs[h][0][0]
            point = V[i]
            test = classify(point)
            if test=="p_in": row += [1]
            elif test=="p_out": row += [0]
            elif test=="p_on": row += [-1]
            else: print "error: in cycle classification"
        testArray += [row]
    return testArray
```

◇

Macro referenced in 18.

Extraction of path-connected boundaries The function `cellsFromCycles`, given by the script below, takes as input the cycle-incidence matrix `testArray`, and return the list `out` of lists of cycles, providing the boundaries of a decomposition into connected (but possibly non path-connected) 2-cells of the fragmented 2-face whose non-intersecting cycles were computed as output of the `biconnectedComponent` function.

First the `sons` of each cycle are computed, i.e. the indices of cycles containd in it, as well as the `level` of every cycle, i.e. their position within the lattice of the containment relation (that is a partial order). The level of a cycle is computed as the sum of elements in its matrix column. The roots of the lattice, i.e. the more external cycles have level -1; the following levels have values $0, 1, 2, ...$, respectively.

Then the cycles are ordered in the encreasing value of their level (or *rank*), and finally the significant subsets of disjoint cycles are extracted within the `out` list, starticg from the root cycle(s). The important properties exploited for the extraction are the following: (a) the first element of each sublist is the external boundary cycle, whereas the following cycles, if any, are its internal boundaries; (b) the rank difference between each external and internal boundary must be less or equal to one. It will be 0 only whe the sublist contains only one element (the external boundary) wich is being compared with itself.

Finally the sublists are pruned, by eliminating those whose first element has benn previously used within some of the previous ones (of course: was already used as an internal cycle).

⟨Extraction of path-connected boundaries 3⟩ ≡

```
""" Extraction of path-connected boundaries """
def cellsFromCycles (testArray):
    n = len(testArray)
    sons = [[h]+[k for k in range(n) if row[k]==1] for h,row in enumerate(testArray)]
    level = [sum(col) for col in TRANS(testArray)]

    def rank(sons): return [level[x] for x in sons]
```

```
    preCells = sorted(sons,key=rank)

    def levelDifference(son,father): return level[son]-level[father]
    root = preCells[0][0]
    out = [[son for son in preCells[0] if (levelDifference(son,root)<=1) ]]
    for k in range(1,n):
        father = preCells[k][0]
        inout = [son for son in preCells[k] if levelDifference(son,father)<=1 ]
        if not (inout[0] in CAT(out)):
            out += [inout]
    return out
◇
```

Macro referenced in 18.

Testing containments between non intersecting cycles The test code for verifying the approch to computation of the containment lattice between non intersecting cycles is given below. Three test files, named respectively `lattice`, `lattice1` and `lattice2`, with the different situations shown in Figure **??**, may be imported from the directory `test/svg/inters/`. Other tests may be easily generated by inserting in this directory some `.svg` files generated with a drawing program.

"test/py/inters/test15.py" 4 ≡

```
""" Testing containments between non intersecting cycles """
from larlib import *

filename = "test/svg/inters/facade.svg"
lines = svg2lines(filename)
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,EV = lines2lar(lines)
V,EVs = biconnectedComponent((V,EV))
# candidate face
FVs = AA(COMP([list,set,CAT]))(EVs)

testArray = latticeArray(V,EVs)

for k in range(len(testArray)):
   print k,testArray[k]
print "\ncells = ", cellsFromCycles(testArray),"\n"

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FVs],submodel,0.15))
◇
```

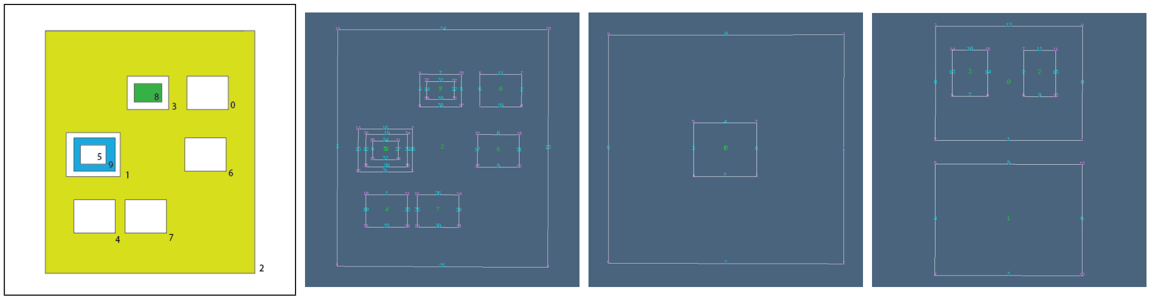

Figure 1: Some examples of nested non intersecting cycles. The corresponding solutions are given in the text.

2.2 Reduction of multiple cycles to a single polyline

The reduction of a lattice of non-intersecting 1-cycles on the boundary of a 2-cell into a single polyline is performed using a scan-line algorithm.

In order to filter the complications induced by edges aligned with the reference axes, first we perform a transformation of vertices from Cartesian to polar coordinates (see Figure 2).

Then a specialized scan-line algorithm is executed, producing a set of *bridge-edges* [YT85] that, added in double instance to the set `EV` of the LAR of the 2-cell, allow for a triangulation of its interior using the algorithm provided by the `poly2tria` module.

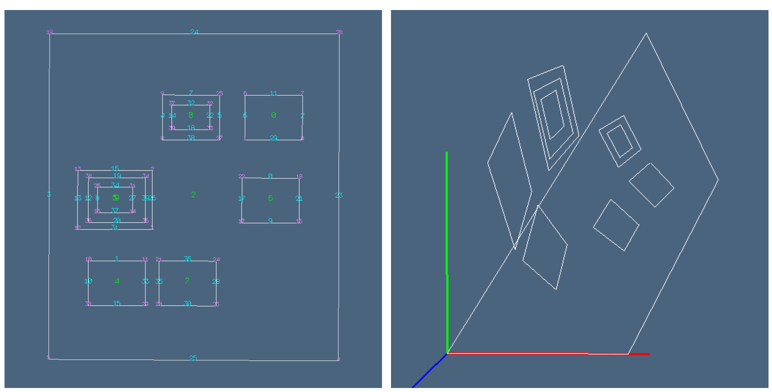

Figure 2: A set of non-intersecting boundary cycles in Cartesian and polar coordinates, using (improperly) an Euclidean metric in the transformed space.

Transforming to polar coordinates The transformation from Cartesian to polar coordinates of the vertices of a two-dimensional LAR model is given in the following script. An image of such transformation is shown in Figure 2. It is only used here to put the vertex coordinates in general position, so simplifying the scan-line algorithm.

⟨ Transforming to polar coordinates 5a ⟩ ≡

```
""" Transforming to polar coordinates """
def cartesian2polar(V):
    Z = [[sqrt(x*x + y*y),math.atan2(y,x)] for x,y in V]
    VIEW(STRUCT(MKPOLS((Z,EV))))
    return Z
```

◇

Macro referenced in 18.

Scan line algorithm

⟨ Scan line algorithm 5b ⟩ ≡

```
""" Scan line algorithm """
def scan(V,FVs, group,cycleGroup,cycleVerts):
    bridgeEdges = []
    scannedCycles = []
    for k,(point,cycle,v) in enumerate(cycleGroup[:-2]):

        nextCycle = cycleGroup[k+1][1]
        n = len(FVs[group][cycle])
        if nextCycle != cycle:
            if not ((nextCycle in scannedCycles) and (cycle in scannedCycles)):
                print "k =",k
                scannedCycles += [nextCycle]
                m = len(FVs[group][nextCycle])
                v1,v2 = v,cycleGroup[k+1][2]
                minDist = VECTNORM(VECTDIFF([V[v1],V[v2]]))
                for i in FVs[group][cycle]:
                    for j in FVs[group][nextCycle]:
                        dist = VECTNORM(VECTDIFF([V[i],V[j]]))
                        if  dist < minDist:
                            minDist = dist
                            v1,v2 = i,j
                bridgeEdges += [(v1,v2)]
    return bridgeEdges[:-1]
```

◇

Macro referenced in 18.

Scan line algorithm input/output The two-dimensional LAR `model` ≡ `V,EV` was previously created by a `lines2lar(lines)` expression.

⟨ Scan line algorithm input/output 6 ⟩ ≡

```
""" Scan line algorithm input/output """
def connectTheDots(model):
    V,EV = model
```

```
    V,EVs = biconnectedComponent((V,EV))
    FV = AA(COMP([sorted,set,CAT]))(EVs)
    testArray = latticeArray(V,EVs)
    cells = cellsFromCycles(testArray)
    FVs = [[FV[cycle] for cycle in cell] for cell in cells]

    indexedCycles = [zip(FVs[h],range(len(FVs[h])))   for h,cell in enumerate(cells)]
    indexedVerts = [CAT(AA(DISTR)(cell)) for cell in indexedCycles]
    sortedVerts = [sorted([(V[v],c,v) for v,c in cell]) for cell in indexedVerts]

    bridgeEdges = []
    cellIndices = range(len(cells))
    for (group,cycleGroup,cycleVerts) in zip(cellIndices,sortedVerts,indexedVerts):
        bridgeEdges += [scan(V,FVs, group,cycleGroup,cycleVerts)]
    return cells,bridgeEdges
```
◇

Macro referenced in 18.

Orientation of component cycles of unconnected boundaries This step of the algorithm needs some preliminary computation, starting from the list `EV` of edges by vertices. Just notice that, at this point of the implementation (as the edges of a set of non intersecting cycles) they all are boundary edges, and hence the `edgeBoundary` variable can be filled with consecutive integers. The `edgeCycles` returned by `boundaryCycles` are very well characterized, according to how discussed in the description of the `Edge cycles associated to a closed chain of edges` paragraph. The corresponding `vertexCycles` are first oriented accordingly to `boundaryCycles`, then rotated (as equivalent permutations) in order to put their element of minimum index in first position (accordingly to the numeration of cycles used in the variable `FVs`). Then, the `CVs` will be set to contain the circularly ordered vertices of the various boundary cycles of each LAR 2-cell, in a manner analogous to FVs, where the vertices are not circularly ordered. Finally, the first (i.e. the *external*) cycle of each cell is counterclockwise oriented, whereas the other cycles (i.e. the *internal* ones), are oriented clockwise.

⟨ Orientation of component cycles of unconnected boundaries 7 ⟩ ≡
```
""" Orientation of component cycles of unconnected boundaries """
def rotatePermutation(inputPermutation,transpositionNumber):
    n = transpositionNumber
    perm = inputPermutation
    permutation = range(n,len(perm))+range(n)
    return [perm[k] for k in permutation]

def canonicalRotation(permutation):
    n = permutation.index(min(permutation))
    return rotatePermutation(permutation,n)
```

```
def setCounterClockwise(h,k,cycle,areas,CVs):
    if areas[cycle] < 0.0:
        chain = copy.copy(CVs[h][k])
        CVs[h][k] = canonicalRotation(REVERSE(chain))

def setClockwise(h,k,cycle,areas,CVs):
    if areas[cycle] > 0.0:
        chain = copy.copy(CVs[h][k])
        CVs[h][k] = canonicalRotation(REVERSE(chain))

def orientBoundaryCycles(model,cells):
    print "\nmodel =",model
    print "\ncells =",cells
    V,EV = model
    edgeBoundary = range(len(EV))
    edgeCycles,_ = boundaryCycles(edgeBoundary,EV)
    vertexCycles = [[ EV[e][1] if e>0 else EV[-e][0] for e in cycle ] for cycle in edgeCycles]
    print "vertexCycles =",vertexCycles
    rotations = [cycle.index(min(cycle)) for cycle in vertexCycles]
    print "rotations =",rotations
    theCycles = sorted([rotatePermutation(perm,n) for perm,n in zip(vertexCycles,rotations)])
    print "theCycles =",theCycles
    CVs = [[theCycles[cycle] for cycle in cell] for cell in cells]
    print "CVs =",CVs
    areas = signedSurfIntegration((V,theCycles,EV),signed=True)
    print "areas =",areas,"\n"

    for h,cell in enumerate(cells):
        for k,cycle in enumerate(cell):
            if k == 0: setCounterClockwise(h,k,cycle,areas,CVs)
            else: setClockwise(h,k,cycle,areas,CVs)
    print "CVs =",CVs
    return CVs
◇
```

Macro referenced in 18.

From nested boundary cycles to triangulation

⟨From nested boundary cycles to triangulation 8⟩ ≡

```
""" From nested boundary cycles to triangulation """
def larTriangulation( (V,EV) ):
    model = V,EV
    cells,bridgeEdges = connectTheDots(model)
    CVs = orientBoundaryCycles(model,cells)
```

```
    polygons = [[[V[u] for u in cycle] for cycle in cell] for cell in CVs]
    triangleSet = []

    for polygon in polygons:
        triangledPolygon = []
        externalCycle = polygon[0]
        polyline = []
        for p in externalCycle:
            polyline.append(Point(p[0],p[1]))
        cdt = CDT(polyline)

        internalCycles = polygon[1:]
        for cycle in internalCycles:
            hole = []
            for p in cycle:
                hole.append(Point(p[0],p[1]))
            cdt.add_hole(hole)

        triangles = cdt.triangulate()
        trias = [ [[t.a.x,t.a.y,0],[t.c.x,t.c.y,0],[t.b.x,t.b.y,0]] for t in triangles ]
        triangleSet += [AA(REVERSE)(trias)]
    return triangleSet
◇
```

Macro referenced in 18.

2.3 Monocyclic polygons using bridge-edges

This subsection, even if correct, is not currently used, since the used triangulation algorithm, from the `poly2tri` package, cannot handle repeated vertices.

Generation of 1-boundaries as vertex permutation In algebraic topology a k-cycle is a k-chain whose boundary is empty. Also, an unconnected k-cycle is the direct sum of two or more k-cycles. A good formal representation of every simplicial k-cycle, where *each* component k-simplex has $k+1$ $(k-1)$-adjacent k-simplices is a $(k+1)$-array, indexed by k-simplices, i.e. the *Winged Representation* [PBCF93].

In the case of oriented 1-cycles, a good representation is given by considering the (ordering of) 0-faces (vertices) as a permutation of n integers, i.e. as a bijective function $\pi : [0, n] \to [0, n]$ that can be represented as an array `verts` of integers indexed on integers, and the 1-faces (edges) as a dictionary (mapping) `nextVert` $verts \to verts$.

In order to join two component cycles using one of `bridgeEdges`, say (u, v), computed by the function `connectTheDots` previously given, we must save $\pi(u)$ and $\pi(v)$, say, within x and y, respectively

⟨ Generation of 1-boundaries as vertex permutation 9 ⟩ ≡

```
""" Generation of 1-boundaries as vertex permutation """
def boundaryCycles2vertexPermutation( model ):
    V,EV = model
    cells,bridgeEdges = connectTheDots(model)
    CVs = orientBoundaryCycles(model,cells)

    verts = CAT(CAT( CVs ))
    n = len(verts)
    W = copy.copy(V)
    assert len(verts) == sorted(verts)[n-1]-sorted(verts)[0]+1
    nextVert = dict([(v,cycle[(k+1)%(len(cycle))]) for cell in CVs for cycle in cell
                    for k,v in enumerate(cycle)])
    for k,(u,v) in enumerate(CAT(bridgeEdges)):
        x,y = nextVert[u],nextVert[v]
        nextVert[u] = n+2*k+1
        nextVert[v] = n+2*k
        nextVert[n+2*k] = x
        nextVert[n+2*k+1] = y
        W += [W[u]]
        W += [W[v]]
        EW = nextVert.items()
    return W,EW
◇
```

Macro referenced in 18.

Wire-frame LAR to boundary polygons The 2-dimensional LAR model (`W,EW`) returned by the function `boundaryCycles2vertexPermutation` is pretty special, since it is at the same time both a standard LAR model, i.e. a pair (vertices, edges_by_vertices), and a permutation of vertex indices providing implicitly the ordered cycles on the boundary of a 2-complex. In other words, `EW` is both a (possibly unconnected) 1-cycle and an 1-boundary.

In the following script we extract the list of connected boundaries (including bridge-edges) from the `EW` permutation of the first m integers.

⟨ Wire-frame LAR to boundary polygons 10 ⟩ ≡

```
""" lar2boundaryPolygons """
def lar2boundaryPolygons(model):
    W,EW = boundaryCycles2vertexPermutation( model )
    EW = AA(list)(EW)
    polygons = []
    for k,edge in enumerate(EW):
        polygon = []
        if edge[0]>=0:
            first = edge[0]
            done = False
        while (not done) and edge[0] >= 0:
```

```
                polygon += [edge[0]]
                edge[0] = -edge[0]
                edge = EW[edge[1]]
                if len(polygon)>1 and polygon[-1] == first:
                    EW[first][0] = -float(first)
                    break
            if polygon != []:
                if polygon[0]==polygon[-1]: polygon=polygon[:-1]
                polygons += [polygon]
        return W,polygons
```
◇

Macro referenced in 18.

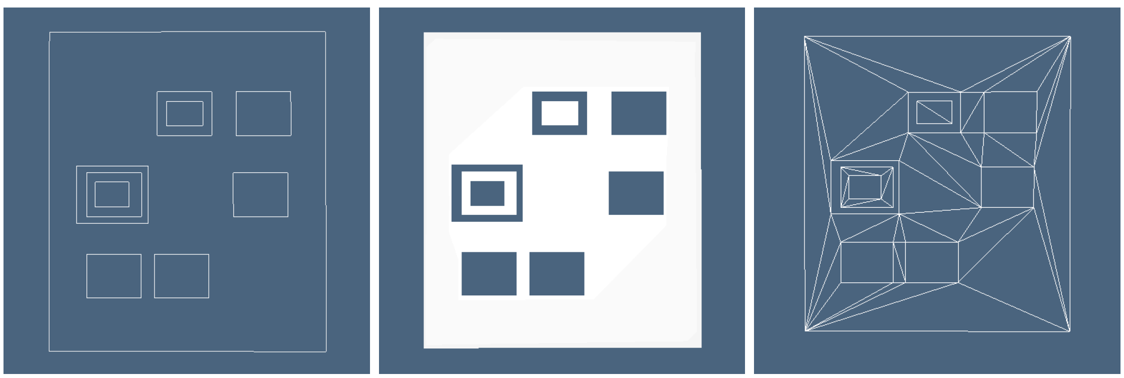

Figure 3: (a) Closed 1-chain c, i.e. such that $\partial\, c = 0$; c is both cycle and boundary at the same time; (b) 2-chain h such that $\partial\, h = c$; (c) 1-skeleton of a triangulation of h.

Edge cycles associated to a closed chain of edges The output `cycles` are returned as lists of oriented edges, given in consecutive sequence in each list. Each cycle is given in the opposite ordering of its first edge. The first edge of each cycle is the one of minimum index in the cycle. The list of output cycles is returned ordered for increasing (or better, in decreasing order, since negative) order of its first element. The first output cycle starts from index 0, that cannot oriented directly, since -0 is not allowed for integer indices. It must be considered as negative, i.e. as oriented as the opposite of its canonical orientation.

⟨ Edge cycles associated to a closed chain of edges 11 ⟩ ≡
```
""" Edge cycles associated to a closed chain of edges """

from collections import defaultdict

def detachManifolds(polygonVerts):
```

```
    vertCycles = []
    for vertexList in polygonVerts:
        vertCount,counts = defaultdict(list),list
        for v in vertexList:
            vertCount[v] += [1]
        counts = [sum(vertCount[v]) for v in vertexList]
        vertCycles += [counts]
    return vertCycles

def splitManifolds(cycles,vertices,manifolds):
    out = []
    for cycle,verts,manifold in zip(cycles,vertices,manifolds):
        if sum(manifold) == len(manifold):
            out += [cycle]
        else:
            transpositionNumbers = [n for n,k in enumerate(manifold) if k>1]
            n = transpositionNumbers[0]
            cycle = rotatePermutation(cycle,n)
            verts = rotatePermutation(verts,n)
            manifold = rotatePermutation(manifold,n)
            starts = AA(C(sum)(-n))(transpositionNumbers)+[len(manifold)]
            pairs = [(start,starts[k+1]) for k,start in enumerate(starts[:-1])]
            splitCycles = [[cycle[k] for k in range(*interval)] for interval in pairs]
            splitVerts = [[verts[k] for k in range(*interval)] for interval in pairs]
            out += splitCycles
    return out

def boundaryCycles(edgeBoundary,EV):
    cycles,cycle = [],[]
    vertices = []

    def singleBoundaryCycle(edgeBoundary):
        verts2edges = defaultdict(list)
        for e in edgeBoundary:
            verts2edges[EV[e][0]] += [e]
            verts2edges[EV[e][1]] += [e]
        cycle,verts = [],[]

        if edgeBoundary == []: return cycle,verts
        e = edgeBoundary[0]
        v,w = EV[e]
        verts = [v,w]
        while edgeBoundary != []:
            cycle += [e]
            edgeBoundary.remove(e)
            v,w = EV[e]
```

```
            verts2edges[v].remove(e)
            verts2edges[w].remove(e)
            w = list(set(EV[e]).difference([verts[-1]]))[0]
            if verts2edges[w] == []: break
            e = verts2edges[w][0]
            verts += [w]
        verts = verts[1:]
        return cycle,verts

    while edgeBoundary != []:
        edgeBoundary = list(set(edgeBoundary).difference(cycle))
        cycle,verts = singleBoundaryCycle(edgeBoundary)
        if cycle!= []:
            cycle = [e if verts[k]==EV[e][0] else -e for k,e in enumerate(cycle)]
            cycles += [cycle]
            vertices += [verts]
    manifolds = detachManifolds(vertices)
    cycles = splitManifolds(cycles,vertices,manifolds)
    return cycles,vertices
◇
```

Macro referenced in 18.

From Struct object to LAR boundary model

⟨From Struct object to LAR boundary model 13a⟩ ≡

```
""" From Struct object to LAR boundary model """
def structFilter(obj):
    if isinstance(obj,list):
        if (len(obj) > 1):
            return [structFilter(obj[0])] + structFilter(obj[1:])
        return [structFilter(obj[0])]
    if isinstance(obj,Struct):
        if obj.category in ["external_wall", "internal_wall", "corridor_wall"]:
            return
        return Struct(structFilter(obj.body),obj.name,obj.category)
    return obj

def structBoundaryModel(struct):
    filteredStruct = structFilter(struct)
    #import pdb; pdb.set_trace()
    V,FV,EV = struct2lar(filteredStruct)
    edgeBoundary = boundaryCells(FV,EV)
    cycles,_ = boundaryCycles(edgeBoundary,EV)
    edges = [signedEdge for cycle in cycles for signedEdge in cycle]
    orientedBoundary = [ AA(SIGN)(edges), AA(ABS)(edges)]
    cells = [EV[e] if sign==1 else REVERSE(EV[e]) for (sign,e) in zip(*orientedBoundary)]
```

```
        if cells[0][0]==cells[1][0]: # bug badly patched! ... TODO better
            temp0 = cells[0][0]
            temp1 = cells[0][1]
            cells[0] = [temp1, temp0]
        return V,cells
    ◇
```

Macro referenced in 18.

From structures to boundary polylines Notice that the `if` predicate `len(V) < len(boundaryEdges)` is used to discriminate between manifold (`False`) and non-manifold (`True`) boundary cases.

⟨From structures to boundary polylines 13b⟩ ≡

```
    """ From structures to boundary polylines """
    def boundaryPolylines(struct):
        V,boundaryEdges = structBoundaryModel(struct)
        print "\nV =",V
        print "\nboundaryEdges =",boundaryEdges
        if len(V) < len(boundaryEdges):
            EV = AA(sorted)(boundaryEdges)
            boundaryEdges = range(len(boundaryEdges))
            cycles,vertices = boundaryCycles(boundaryEdges,EV)
            print "\ncycles =",cycles
            print "\nvertices =",vertices
            polylines = [[V[v] for v in verts]+[V[verts[0]]] for verts in REVERSE(vertices)]
            print "\npolylines =",polylines,"\n"
        else:
            polylines = boundaryModel2polylines((V,boundaryEdges))
        print "\npolylines =",polylines,"\n"
        return polylines
    ◇
```

Macro referenced in 18.

From LAR oriented boundary model to polylines

⟨From LAR boundary model to polylines 14⟩ ≡

```
    """ From LAR oriented boundary model to polylines """
    def boundaryModel2polylines(model):
        if len(model)==2: V,EV = model
        elif len(model)==3: V,FV,EV = model
        polylines = []
        succDict = dict(EV)
        visited = [False for k in range(len(V))]
        nonVisited = [k for k in succDict.keys() if not visited[k]]
        while nonVisited != []:
```

```
        first = nonVisited[0]; v = first; polyline = []
        while visited[v] == False:
            visited[v] = True;
            polyline += V[v],
            v = succDict[v]
        polyline += [V[first]]
        polylines += [polyline]
        nonVisited = [k for k in succDict.keys() if not visited[k]]
    return polylines

def boundaryModel2polylines(model):
    if len(model)==2: V,EV = model
    elif len(model)==3: V,FV,EV = model
    polylines = []
    succDict = dict(EV)
    visited = [False for k in range(len(V))]
    nonVisited = [k for k in succDict.keys() if not visited[k]]
    while nonVisited != []:
        first = nonVisited[0]; v = first; polyline = []
        while visited[v] == False:
            visited[v] = True;
            polyline += V[v],
            v = succDict[v]
        polyline += [V[first]]
        polylines += [polyline]
        nonVisited = [k for k in succDict.keys() if not visited[k]]
    return polylines
◇
```

Macro referenced in 18.

2.4 Visualization of a 2D complex and 2-chain

⟨ Visualization of a 2D complex and 2-chain 15 ⟩ ≡

```
""" Visualization of a 2D complex and 2-chain """
def larComplexChain(model):
    V,FV,EV = model
    VV = AA(LIST)(range(len(V)))
    csrBoundaryMat = boundary1(FV,EV,VV)
    def larComplexChain0(chain):
        boundaryChain = chain2BoundaryChain(csrBoundaryMat)(chain)
        outModel = V,[EV[e] for e in boundaryChain]
        triangleSet = larTriangulation(outModel)
        return boundaryChain,triangleSet
    return larComplexChain0

def viewLarComplexChain(model):
```

```
    V,FV,EV = model
    operator = larComplexChain(model)
    def viewLarComplexChain0(chain):
        boundaryChain,triangleSet = operator(chain)
        hpcChain = AA(JOIN)(AA(AA(MK))(CAT(triangleSet)))
        hpcChainBoundary = AA(COLOR(RED))(MKPOLS((V,[EV[e] for e in boundaryChain])))
        VIEW(STRUCT( hpcChain + hpcChainBoundary ))
        VIEW(EXPLODE(1.2,1.2,1.2)( hpcChain + hpcChainBoundary ))
    return viewLarComplexChain0
```
◇

Macro referenced in 18.

2.5 Point in polygon classification

⟨ Point in polygon testing 16a ⟩ ≡

```
""" Point-in-polygon classification algorithm """
```
⟨ Half-line crossing test 16c ⟩
⟨ Tile codes computation 16b ⟩
⟨ Point-in-polygon classification algorithm 17 ⟩
◇

Macro referenced in 18.

Tile codes computation

⟨ Tile codes computation 16b ⟩ ≡

```
""" Tile codes computation """
def setTile(box):
    tiles = [[9,1,5],[8,0,4],[10,2,6]]
    b1,b2,b3,b4 = box
    def tileCode(point):
        x,y = point
        code = 0
        if y>b1: code=code|1
        if y<b2: code=code|2
        if x>b3: code=code|4
        if x<b4: code=code|8
        return code
    return tileCode
```
◇

Macro referenced in 16a.

Half-line crossing test

⟨ Half-line crossing test 16c ⟩ ≡

```
""" Half-line crossing test """
def crossingTest(new,old,count,status):
    if status == 0:
        status = new
        count += 0.5
    else:
        if status == old: count += 0.5
        else: count -= 0.5
        status = 0
◇
```

Macro referenced in 16a.

Point in polygon testing

⟨ Point-in-polygon classification algorithm 17 ⟩ ≡

```
""" Point in polygon classification """
def pointInPolygonClassification(pol):

    V,EV = pol
    # edge orientation
    FV = [sorted(set(CAT(EV)))]
    orientedCycles = boundaryPolylines(Struct([(V,FV,EV)]))
    EV = []
    for cycle in orientedCycles:
        EV += zip(cycle[:-1],cycle[1:])

    def pointInPolygonClassification0(p):
        x,y = p
        xmin,xmax,ymin,ymax = x,x,y,y
        tilecode = setTile([ymax,ymin,xmax,xmin])
        count,status = 0,0

        for k,edge in enumerate(EV):
            p1,p2 = edge[0],edge[1]
            (x1,y1),(x2,y2) = p1,p2
            c1,c2 = tilecode(p1),tilecode(p2)
            c_edge, c_un, c_int = c1^c2, c1|c2, c1&c2

            if c_edge == 0 and c_un == 0: return "p_on"
            elif c_edge == 12 and c_un == c_edge: return "p_on"
            elif c_edge == 3:
                if c_int == 0: return "p_on"
                elif c_int == 4: count += 1
            elif c_edge == 15:
                x_int = ((y-y2)*(x1-x2)/(y1-y2))+x2
                if x_int > x: count += 1
```

```
            elif x_int == x: return "p_on"
        elif c_edge == 13 and ((c1==4) or (c2==4)):
                crossingTest(1,2,status,count)
        elif c_edge == 14 and (c1==4) or (c2==4):
                crossingTest(2,1,status,count)
        elif c_edge == 7: count += 1
        elif c_edge == 11: count = count
        elif c_edge == 1:
            if c_int == 0: return "p_on"
            elif c_int == 4: crossingTest(1,2,status,count)
        elif c_edge == 2:
            if c_int == 0: return "p_on"
            elif c_int == 4: crossingTest(2,1,status,count)
        elif c_edge == 4 and c_un == c_edge: return "p_on"
        elif c_edge == 8 and c_un == c_edge: return "p_on"
        elif c_edge == 5:
            if (c1==0) or (c2==0): return "p_on"
            else: crossingTest(1,2,status,count)
        elif c_edge == 6:
            if (c1==0) or (c2==0): return "p_on"
            else: crossingTest(2,1,status,count)
        elif c_edge == 9 and ((c1==0) or (c2==0)): return "p_on"
        elif c_edge == 10 and ((c1==0) or (c2==0)): return "p_on"
    if ((round(count)%2)==1): return "p_in"
    else: return "p_out"
return pointInPolygonClassification0
```
◇

Macro referenced in 16a.

3 Exporting the library

"larlib/larlib/triangulation.py" 18 ≡

```
""" Module for pipelined intersection of geometric objects """
from larlib import *

⟨Edge cycles associated to a closed chain of edges 11⟩
⟨Point in polygon testing 16a⟩
⟨Classification of non intersecting cycles 2⟩
⟨Extraction of path-connected boundaries 3⟩
⟨Transforming to polar coordinates 5a⟩
⟨Scan line algorithm 5b⟩
⟨Scan line algorithm input/output 6⟩
⟨Orientation of component cycles of unconnected boundaries 7⟩
⟨From nested boundary cycles to triangulation 8⟩
⟨Generation of 1-boundaries as vertex permutation 9⟩
```

⟨ Wire-frame LAR to boundary polygons 10 ⟩
⟨ From Struct object to LAR boundary model 13a ⟩
⟨ From structures to boundary polylines 13b ⟩
⟨ From LAR boundary model to polylines 14 ⟩
⟨ Visualization of a 2D complex and 2-chain 15 ⟩
⋄

4 Testing

This section in subdivided in two subsections, testing respectively (a) the construction and visualization of the containment lattice of non-intersecting polygonal cycles, and (b) the triangulation of the boundary polygon of random polygonal domains, including non connected parts, nested cycles and non-manifold cycles, both internal and external to the given 2D domain.

4.1 Drawing multiply-connected boundary polylines

A multiply-connected boundary polyline gives the boundary edges of a 2-cell. In LAR, 2-cells must be connected, but not necessarily simply-connected (or path-connected, or homotopic to a point). The "solid" drawing of such a generally non-convex 2-cells is not easy. In particular, they must be decomposed into a coherently-oriented simplicial 2-complex, i.e. into a set of equioriented triangles. The library function used at this purpose (), contained in the `poly2tria` module, only accepts a single boundary polyline. Hence the purpose of this section is to transform a list of cycles, corresponding orderly to the exterior and the interior boundaries of a 2-cell, into a single polyline, using the so-called *bridge-edges* [YT85] mechanism. The transformation from a set of non-intersecting boundary cycles to a single connected polyline is given in Section 2.2.

Generating the LAR of a set of non-intersecting cycle We use the `test/svg/lattice.svg` file to test the exporting of different boundary chains. The example `test15.py` aims to prepare the computational environment for writing down the LAR of the 2-complex generated by any set of non-intersecting 1-cles in 2D.

Let us remember that any d-cell in the domain of the LAR scheme must be connected, but not necessarily contractible to a point, i.e. may contain internal holes. The goal af this test and of the previous one, is hence to generalize the creation of 2-complexes (sets of polygons) starting from several non-intersecting boundary cycles, instead than starting from just one closed polyline.

The motivation arises from situations created by the Boolean algorithm, as well as from the imput of a 2-complex from general wire-frame drawings.

"`test/py/triangulation/test01.py`" 19 ≡

```
""" Testing containments between non intersecting cycles """
from larlib import *

filename = "test/svg/inters/facade.svg"
lines = svg2lines(filename)
VIEW(STRUCT(AA(POLYLINE)(lines)))

V,EV = lines2lar(lines)
V,EVs = biconnectedComponent((V,EV))
# candidate face
FVs = AA(COMP([list,set,CAT]))(EVs)

testArray = latticeArray(V,EVs)

for k in range(len(testArray)):
   print k,testArray[k]
print "\ncells = ", cellsFromCycles(testArray),"\n"

VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FVs],submodel,0.15))
```
◇

"test/py/triangulation/test02.py" 20 ≡

```
""" Generating the LAR of a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/triangulation/')
from test01 import *

cells = cellsFromCycles(testArray)
CV = AA(COMP([list,set,CAT]))(EVs)
EVdict = dict(zip(EV,range(len(EV))))
FE = [[EVdict[edge] for edge in cycle] for cycle in EVs]
edges = [CAT([FE[cycle] for cycle in cell]) for cell in cells]
FVs = [[CV[cycle] for cycle in cell] for cell in cells]
FV = AA(CAT)(FVs)

n = len(cells)
chains = allBinarySubsetsOfLenght(n)

cycles = STRUCT(MKPOLS((V,EV)))
csrBoundaryMat = boundary(FV,EV)
for chain in chains:
    chainBoundary = COLOR(RED)(STRUCT(MKPOLS((V,[EV[e]
                        for e in chain2BoundaryChain(csrBoundaryMat)(chain)]))))
    VIEW(STRUCT([cycles, chainBoundary]))
```

◇

LAR of the 2-complex generated by non-intersecting cycles Therefore, the LAR of the 2-complex generated from the input get from `test15.py` is `(V,FV,EV)` with the components given by the the following:

```
V = [[0.7808,0.6751],[0.6044,0.6751],[0.319,0.5804],[0.319,0.3994],[0.8936,0.0],
[0.0,0.0],[0.6044,0.8123],[0.7808,0.8123],[0.3495,0.6751],[0.3495,0.8123],
[0.1218,0.3036],[0.2983,0.3036],[0.0886,0.3994],[0.0886,0.5804],[0.2581,0.4505],
[0.1495,0.4505],[0.7717,0.4213],[0.5952,0.4213],[0.7717,0.5585],[0.0,1.0],
[0.3403,0.1664],[0.3403,0.3036],[0.5952,0.5585],[0.5168,0.1664],[0.5168,0.3036],
[0.5259,0.8123],[0.1495,0.5293],[0.5259,0.6751],[0.8936,1.0],[0.2983,0.1664],
[0.1218,0.1664],[0.2581,0.5293],[0.4965,0.7815],[0.4965,0.7059],[0.2983,
0.5585],[0.2983,0.4213],[0.1218,0.4213],[0.3789,0.7815],[0.1218,0.5585],
[0.3789,0.7059]]

FV = [[0,1,2,3,4,5,6,7,8,9,10,11,12,13,16,17,18,19,20,21,22,23,24,25,27,
28,29,30],[32,33,37,39],[34,35,36,38,26,15,14,31]]

EV = [(0,1),(0,7),(1,6),(2,3),(2,13),(3,12),(4,5),(4,28),(5,19),(6,7),(8,9),
(8,27),(9,25),(10,11),(10,30),(11,29),(12,13),(14,15),(14,31),(15,26),
(16,17),(16,18),(17,22),(18,22),(19,28),(20,21),(20,23),(21,24),(23,24),
(25,27),(26,31),(29,30),(32,33),(32,37),(33,39),(34,35),(34,38),(35,36),
(36,38),(37,39)]
```

Of course, this LAR representation gives full control of the complex topology, including k-(co)chains and (co)boundary operators. For example, in Figure 4 we show a solid image of the three 2-cells in `FV`, and, drawn in red the boundary 1-cells in the complex, corresponding to the 2-chains of coordinates $[1, 0, 0]$, $[0, 1, 0]$, $[0, 0, 1]$, and $[1, 1, 0]$, respectively.

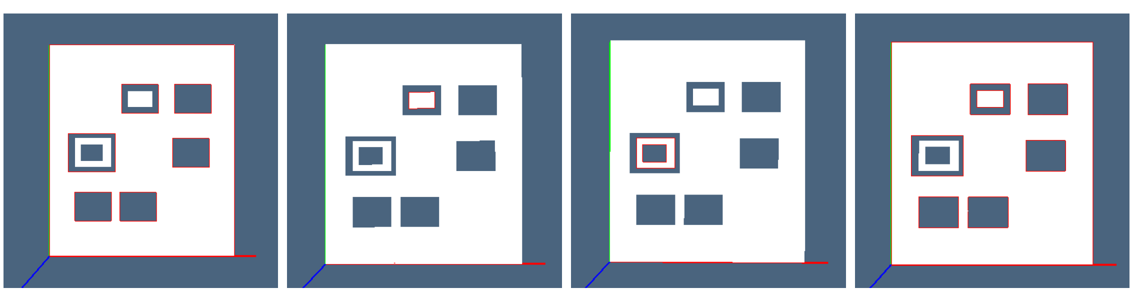

Figure 4: Some examples of boundaries (in red) of 2-chains from nested non intersecting cycles.

4.2 Boundary polylines

Generating the LAR of a set of non-intersecting cycles The example provided by `test/py/inters/test17.py` is completed here by showing the solid drawing of the generated LAR data structure, and superimposing to it the boundary 1-chains generated by several 2-chains.

"test/py/triangulation/test03.py" 21 ≡
```
""" Generating the LAR of a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/triangulation/')
from test02 import *

lar = (V,FV,EV)

bcycles,_ = boundaryCycles(range(len(EV)),EV)
polylines = [[V[EV[e][1]] if e>0 else V[EV[-e][0]] for e in cycle ] for cycle in bcycles]
polygons = [polyline + [polyline[0]] for polyline in polylines]

complex = SOLIDIFY(STRUCT(AA(POLYLINE)(polygons)))
csrBoundaryMat = boundary(FV,EV)
for chain in chains:
    chainBoundary = COLOR(RED)(STRUCT(MKPOLS((V,[EV[e]
                        for e in chain2BoundaryChain(csrBoundaryMat)(chain)]))))
    VIEW(STRUCT([complex, chainBoundary]))
◇
```

"test/py/triangulation/test04.py" 22a ≡
```
""" Orienting a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/triangulation/')
from test03 import *

cells,bridgeEdges = connectTheDots((V,EV))
CVs = orientBoundaryCycles((V,EV),cells)

print "\nCVs =",CVs
◇
```

"test/py/triangulation/test05.py" 22b ≡
```
""" Generating the LAR of a set of non-intersecting cycles """
from larlib import *

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

```
from test03 import *

W,EW = boundaryCycles2vertexPermutation( (V,EV) )
VIEW(EXPLODE(1.2,1.2,1.2)(MKPOLS((W,EW))))
◇
```

"test/py/triangulation/test06.py" 22c ≡

```
""" Generating the Triangulation of a set of non-intersecting cycles """
from larlib import *

sys.path.insert(0, 'test/py/triangulation/')
from test03 import *

triangleSet = larTriangulation( (V,EV) )

VIEW(STRUCT(AA(JOIN)(AA(AA(MK))(CAT(triangleSet)))))
VIEW(SKEL_1(STRUCT(AA(JOIN)(AA(AA(MK))(CAT(triangleSet))))))

"""
model = V,EV
W,FW = lar2boundaryPolygons(model)
polygons = [[W[u] for u in poly] for poly in FW]
VIEW(STRUCT(AA(POLYLINE)(polygons)))

triangleSet,triangledFace = [],[]
for polygon in polygons:
    triangledPolygon = []
    polyline = []
    for p in polygon:
        polyline.append(Point(p[0],p[1]))
    cdt = CDT(polyline)

    triangles = cdt.triangulate()
    trias = [ [[t.a.x,t.a.y,0],[t.c.x,t.c.y,0],[t.b.x,t.b.y,0]] for t in triangles ]
    triangleSet += [AA(REVERSE)(trias)]
"""
◇
```

4.3 Testing with random polygons

In order to test the algorithms implemented in this library, we make use of random triangulations of $C_2 = \{x \in \mathbb{R}^2 : \|x\| \leq 1\}$.

"test/py/triangulation/test07.py" 24a ≡

```
from larlib import *
```

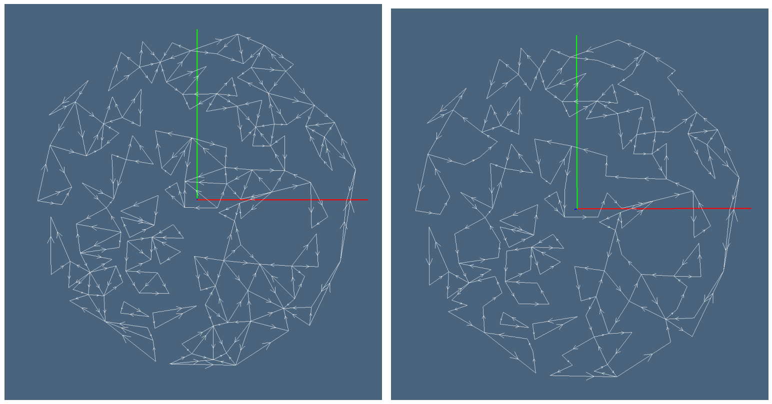

Figure 5: example caption

⟨ random 1-boundary generation 24b ⟩
◇

random 1-boundary generation

⟨ random 1-boundary generation 24b ⟩ ≡

```
""" random 1-boundary generation """
import sys
sys.path.insert(0, '/Users/paoluzzi/Documents/dev/lar-cc/test/py/larcc/')
from test16 import *

EV = AA(list)(cells)
V,EVs = biconnectedComponent((V,EV))
FV = AA(COMP([sorted,list,set,CAT]))(EVs)
FV = sorted( FV,key=len,reverse=True )
EVs = sorted( EVs,key=len,reverse=True )
W = [eval(vcode(v)) for v in V]
testArray = latticeArray(W,EVs)

"""
bcycles,bverts = boundaryCycles(range(len(EW)),EW)
VIEW(STRUCT(AA(POLYLINE)([[V[v] for v in verts] for verts in bverts])))
"""

colors = [CYAN, MAGENTA, WHITE, RED, YELLOW, GRAY, GREEN, ORANGE, BLACK, BLUE,
         PURPLE, BROWN]
components = [COLOR(colors[k%12])(STRUCT(MKPOLS((V,ev)))) for k,ev in enumerate(EVs)]
VIEW(STRUCT(components))
```
◇

Macro referenced in 24a.

References

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

[D08] V. Domiter and B. alik, *Sweepline algorithm for constrained delaunay triangulation*, International Journal of Geographical Information Science **22** (2008), no. 4, 449–462.

[PBCF93] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci, *Dimension-independent modeling with simplicial complexes*, ACM Trans. Graph. **12** (1993), no. 1, 56–102.

[YT85] F. Yamaguchi and T. Tokieda, *Bridge edge and triangulation approach in solid modeling*, Frontiers in Computer Graphics (Berlin) (T.L. Kunii, ed.), Springer Verlag, 1985.