Triangulation of the boundary of a 2-chain [*]

Alberto Paoluzzi

January 1, 2016

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 B. Zalik [D08].

Contents

[*]This document is part of the *Linear Algebraic Representation with CoChains* (LAR-CC) framework [CL13]. January 1, 2016

1 From a finite set of 2D segments to a regular 2-complex

Let suppose that a 2-complex in the plane is given as a LAR triple `(V,FV,EV)`, and that a 2-chain `chain` $\in \mathbb{Z}_2^n$ is also given, with $\mathbb{Z}_2 = \{0, 1\}$ and $n = |\mathtt{V}|$. The aim of this module is to develop a methodology and an implementation to compute a triangulation of any 2-chain extracted from a 2-complex.

This first section will concentrate on (1) how to generate a regular 2-complex starting from a finite set of 2D segments, i.e. how to generate a LAR triple `(V,FV,EV)`, and (2) how to triangulate *any 2-chain*, and in particular a single 2-cell in 2D, using a standard CDT (Constrained Delaunay Triangulation) algorithm that does not allow for polygons with repeated vertices.

The first point is not trivial, since a general 2-chain may contain 2-cells that are not contractible to a point (since they may contain any finite number of internal holes), and furthermore may be non-manifold, i.e. with boundary vertices incident on more than 2 boundary edges. In addition, in the general case, holes of 2-cells may contain other 2-cells, possibly with holes, and so on.

In particular, the presence of non-manifold vertices is the common case in random generation of complexes of polygons in the plane, and/or for the typical 2-chain extracted from a general 2-complex. In order of preparing the input for the triangulation algorithm, the boundary 0-chain extracted from the general 2-chain must undergo to a circular ordering, producing—in the non-manifold case—any finite number of vertex repetitions. Repeated vertices will be disambiguated by applying small local perturbations towards the interior of the cycle. Anyway, boundary cycles will be perturbed only for drawing solidly the 2-chain as a single polygon with holes, whereas the exact LAR of the 2-complex will not be modified.

1.1 Definitions and methodology

First we provide some definitions useful in what follows, then we formalize the implemented methods.

Definition 1. *(Regular complex) A d-complex is* regular *when any k-cell ($0 \le k \le d$) is contained in a d-cell.*

Definition 2. *(Circular ordering) A* 0*-chain is* circularly ordered *when every consecutive pair of its* 0*-cells (module the lenght of the chain) maps to a* 1*-cell of the boundary 1-chain of a 2-chain.*

1.1.1 Input preprocessing

The input is given by m line segments in the plane, with $p \leq 2m$ extreme points.

First we compute all the k intersection points internal to the lines, using adeguate data structures and algorithms to accelerate the search. This problem is developed within the `inters` module, and implemented in the `larlib` sub-package named `inters.py`.

The arrangement of lines is correspondingly updated—by splitting the input lines on their *internal* intersection points—so that the splitted line segments may possibly intersect only on extreme points.

Let consider the undirected graph $G = (N, A)$, where the set of nodes N is given by the extreme points of splitted segments, and compute the set $\mathcal{G} = \{G_i\}_{i \in I}$ of maximal biconnected components of G. Suppose $\mathcal{G}$ nonempty.

Definition 3. *(Vertices) The union of node sets of biconnected components provides the* vertex *set* $\texttt{V} = \cup_{i \in I} N_i$*. This one supplies the geometric embedding of the 0-cells of the 2-complex.*

Definition 4. *(Edges) The disjoint union of arcs of biconnected components provides the* edge set $\texttt{EV} = \cup_{i \in I} A_i$*. It supplies an indexed array of pairs of node indices, i.e. a LAR representation of the 1-cells of the 2-complex.*

Definition 5. *(Cycles) The* boundaries *of 2-chains, corresponding one-to-one to the biconnected components* $G_i \in \mathcal{G}$ *considered as a plane partition, are* closed *1-chains called* cicles $\mathcal{C} = \{\partial G_i\}_{i \in I}$.

A cycle may be represented either as an unordered set of 1-cells (1-chain) or as an ordered set of 0-cells (0-chain), that we call *1-cycle* and *0-cycle*, respectively.

Definition 6. *(Manifold and non-manifold cycles) A boundary 1-chain with a number n of both 1-cells and 0-cells may be represented as a permutation of vertex indices. We call it a* manifold cycle*. Conversely, if the number of its vertices is smaller than the number of its edges, then the cycle is a* non-manifold*.*

A cycle may be decomposed into one or more loops of edges. In a cycle all loops are piecewise-linear curves; they share one or more common vertices in the case of non-manifold cycles. The set of loops, denoted in the following as $\mathcal{L}$, has cardinality $\ell = |\mathcal{L}|$ greater or equal to the cardinality $n = |\mathcal{G}|$, where the non-manifold vertices have even incidence number greater than 2. The cycles in $\mathcal{C}$ are boundaries of connected regions of the plane. The loops in $\mathcal{L}$ are boundaries of regions contractible to a point.

1.1.2 Algorithm: LAR generation

Every cycle in $\mathcal{C}$ subdivides the plane in 3 regions, of internal, external, and boundary points, respectively. The set union of $\mathcal{C}$ elements provides a *cover* of the plane. Let us consider the collection $\mathring{\mathcal{C}}$ of the open sets of points internal to the cycles.

Definition 7. *(Faces) A partition of the plane is given by the collection of closures of the set differences (*regularized differences*) between every element of $\mathring{\mathcal{C}}$ and all the others. We call this set* `F`*, and its elements either* faces *or 2-cells of the 2-complex.*

The triple `(V,E,F)`, as defined before, is a cellular 2-complex defined by the d-cells $(0 \leq d \leq 2)$ in `V,E,F`, that provide the 0-, 1-, and 2-cells, respectively.

Statement 1. *(LAR) Every cell of a 2-complex as a collection of point-sets is completely is represented discretely, in the sense that can be completely reconstructed, starting from the* `V` *subset of its boundary.*

The point now is how to construct the array `FV` starting from the arrays `V`, `EV`, and from the set $\mathcal{C}$ returned by the function `farFromLines()` implemented in the `larlib` module `inters.py`.

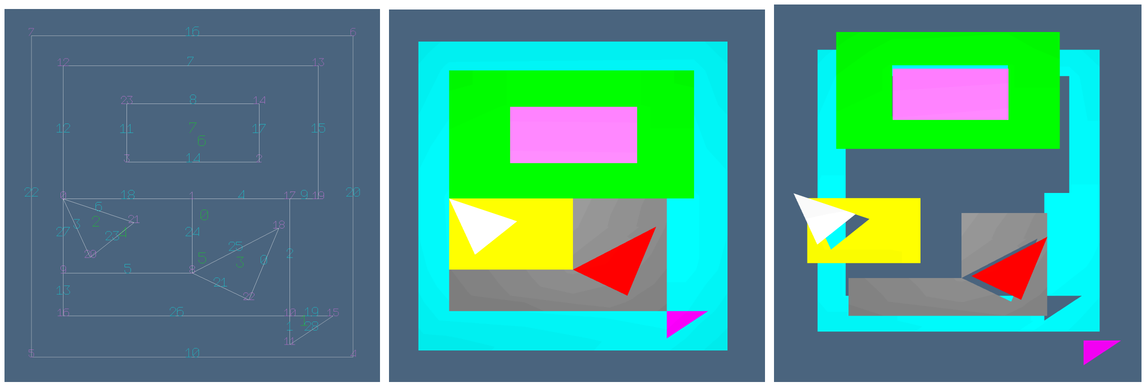

Figure 1: LAR cellular 2-complex generated by an arrangement of line segments: (a) drawing of numbered cells; (b) 2-complex with different colors; (c) exploded view of 2-cells. Notice that every cell is connected, but not necessarily path-connected.

Computing a LAR 2-complex from an arrangement of line segments The computational process discussed in Section 1.1.2 is implemented by the script below. First, the planar graph of edges is built by the `lines2lar` function, then the set of graph `cycles` corresponding to the partition of 2D plane induced by the graph is returned by the `facesFromComps` function. These are sorted in increasing order of the surrounded area, so getting at the end of the sorted list two copies of the exterior boundary cycle. The very last cycle is removed, so that every internal edge is used twice in some cycle, whereas the boundary edges are used only once. Finally the various cycles are checked pairwise for mutual containment, producing the `latticeArray` of their partial order. Finally, the reduced set of `cells` as lists of bounding cycles of vertices is produced by the `cellsFromCycles` function, finally removed by multiple vertices to produce the `FV` LAR representation.

⟨Computing a LAR 2-complex from an arrangement of line segments 4⟩ ≡

```
""" Computing a LAR 2-complex from an arrangement of line segments"""
def larPair2Triple(model):
    V,EV = model
    cycles,ecycles = makeCycles(model)
    areas = integr.surfIntegration((V,cycles,EV))
    orderedCycles = sorted([[area,cycles[f]] for f,area in enumerate(areas)])
    interiorCycles = [face for area,face in orderedCycles[:-1]]
    EdgeCyclesByVertices = [zip(cycle[:-1],cycle[1:])+[(cycle[-1],cycle[0])]
                        for cycle in interiorCycles]
    latticeArray = computeCycleLattice(V,EdgeCyclesByVertices)
    cells = cellsFromCycles(latticeArray)
    FV = [list(set(CAT([interiorCycles[k] for k in cell]))) for cell in cells]
    return V,FV,EV
◇
```

Macro referenced in 28.

LAR of a 2-complex of general type The LAR of the complex in Figure 1, and constructed starting from a set of line segmants in the plane, is given here. Notice that cells may have non connected boundaries (may contain holes) and/or non-manifold points.

"test/py/triangulation/test08.py" 5a ≡

```
""" Test example of LAR of a 2-complex with non-contractible and non-manifold cells"""
from larlib import *

V = [[0.0989,0.492],[0.5,0.492],[0.708,0.6068],[0.2966,0.6068],[1.0,0.0],
[0.0,0.0],[1.0,1.0],[0.0,1.0],[0.5,0.2614],[0.0989,0.2614],[0.8034,
0.1273],[0.8034,0.0386],[0.0989,0.9068],[0.892,0.9068],[0.708,0.7886],
[0.9375,0.1273],[0.0989,0.1273],[0.8034,0.492],[0.7693,0.4009],[0.892,
0.492],[0.183,0.3097],[0.3193,0.4182],[0.6761,0.1773],[0.2966,0.7886]]

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

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

```
VV = AA(LIST)(range(len(V)))
submodel = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],submodel,0.3))
◇
```

Generation of LAR of a 2-complex from line segments The LAR model `(V,FV,EV)` displayed in Figure fig:linesToLAR and listed in the Script above, i.e. in the `test08.py` file, can be generated as below.

"test/py/triangulation/test09.py" 5b ≡

```
""" Test example of LAR of a 2-complex with non-contractible and non-manifold cells"""
from larlib import *

filename = "test/svg/inters/graph.svg"
lines = inters.svg2lines(filename)
V,FV,EV = larFromLines(lines)

VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],hpc,0.2))
◇
```

"test/py/triangulation/test11.py" 5c ≡

```
""" Test example of LAR of a 2-complex with non-contractible and non-manifold cells"""
from larlib import *

filename = "test/svg/triangulation/facade.svg"
lines = inters.svg2lines(filename)
V,FV,EV = larFromLines(lines)

VV = AA(LIST)(range(len(V)))
hpc = STRUCT(MKPOLS((V,EV)))
VIEW(larModelNumbering(1,1,1)(V,[VV,EV,FV],hpc,0.2))
◇
```

Solid PyPLaSM visualization of a 2-complex A PyPLaSM visualization of a 2-complex of very general kind may be obtained with the `MKFACES` function, implemented by the `SOLIDIFY` of geometric values of *HPC* type, starting from the LAR model of the 2-complex to be displayed. Of course, a `boundaryOperator` is built using the LAR representation of 2-cells and 1-cells, i.e. `FV` and `EV`, respectively.

⟨ Solid PyPLaSM visualization of a 2-complex 6a ⟩ ≡

```
""" Solid PyPLaSM visualization of a 2-complex with non-contractible
    and non-manifold cells"""
def MKFACES(model):
    V,FV,EV = model
    VV = AA(LIST)(range(len(V)))
    bmatrix = boundary1(FV,EV,VV)
    boundaryOperator = chain2BoundaryChain(bmatrix)
    chain = [0]*len(FV)
    boundingEdges = []
    for k,face in enumerate(FV):
      unitChain = copy.copy(chain)
      unitChain[k] = 1
      boundingEdges += [boundaryOperator(unitChain)]
    faces = [SOLIDIFY(STRUCT([POLYLINE([V[v] for v in EV[e]]) for e in edges]))
        for edges in boundingEdges]
    return faces
◇
```

Macro referenced in 28.

Color visualization A color visualization of the 2-cells of the previous cellular complex is produced by the `test10.py` file.

"test/py/triangulation/test10.py" 6b ≡

```
""" Test example of LAR of a 2-complex with non-contractible and non-manifold cells"""
from larlib import *
sys.path.insert(0, 'test/py/triangulation/')
from test09 import *

model = V,FV,EV

faces = MKFACES(model)
colors = [CYAN,MAGENTA,WHITE,RED,YELLOW,GRAY,GREEN,ORANGE,BLUE,PURPLE,BROWN,BLACK]
components = [COLOR(colors[k%12])(faces[k]) for k in range(len(FV))]
VIEW(STRUCT(components))
◇
```

"test/py/triangulation/test12.py" 7 ≡

```
""" Test example of LAR of a 2-complex with non-contractible and non-manifold cells"""
from larlib import *
sys.path.insert(0, 'test/py/triangulation/')
from test11 import *

model = V,FV,EV
```

```
faces = MKFACES(model)
colors = [CYAN,MAGENTA,WHITE,RED,YELLOW,GRAY,GREEN,ORANGE,BLUE,PURPLE,BROWN,BLACK]
components = [COLOR(colors[k%12])(faces[k]) for k in range(len(FV))]
VIEW(STRUCT(components))

VIEW(STRUCT(MKFACES((V,[FV[4]],EV))))
```
◇

1.2 Construction of LAR of 2-faces

The contruction of faces in FV begins by computing the containment relation between all pairs of cycles in $\mathcal{C}$.

Since they are boundary 1-chains of 2-chains corresponding to maximal biconnected components of (V, EV) graph, the cycles in $\mathcal{C}$ cannot have intersection, except possibly on common vertices, corresponding to *separation nodes* of the graph. In order to test if c_h is either *internal* or *external* to c_k, for $c_h, c_k \in \mathcal{C}$, it is sufficient to test the containment in c_k of a single point *internal* to c_h.

We construct the binary matrix of the containment relation, which is anti-symmetric, reflexive and transitive. Therefore the containment relation is a partial order, that can be represented as the DAG (Directed Acyclic Graph) of its transitive reduction, often called *Hasse diagram*.

The Hasse diagram of this relation is either a single tree or a forest, i.e. a set of trees without common nodes, depending on the connection of the domain. By numerating the nodes with their level number, defined as the integer distance of the node from the tree roots, and by deleting the arcs with starting node odd, we get a forest of trees with two levels, whose root(s) correspond to the exterior cycle(s) and whose leaves correspond to the internal holes of the faces.

Every such small tree ("sapling") of cycles, possibly reduced to the only root, in case of a face without holes, will correspond to a 2-cell of the 2-complex.

1.3 Solid display of a 2-complex

Let we start with a triple LAR model (V,FV,EV).

First we compute the signed boundary operator. Then for each cycle c in $\mathcal{C}$ we have a list of signed boundary edges. The loops of each cycle are extracted and classified as external or internal loops, and returned as circularly ordered sequences of vertex indices. Then the kind as manifold or non-manifold of c is computed, by comparing the number of edges with the number of vertices.

The non-manifold vertices are possibly signed, and the transformation from a list of vertex indices to a list of vertices (i.e. list of lists of real coordinates) is applied.

Finally, every non-manifold vertex is moved in general position by applying to it a small translation perturbation towards the *interior* of the cycle. The translation vector is

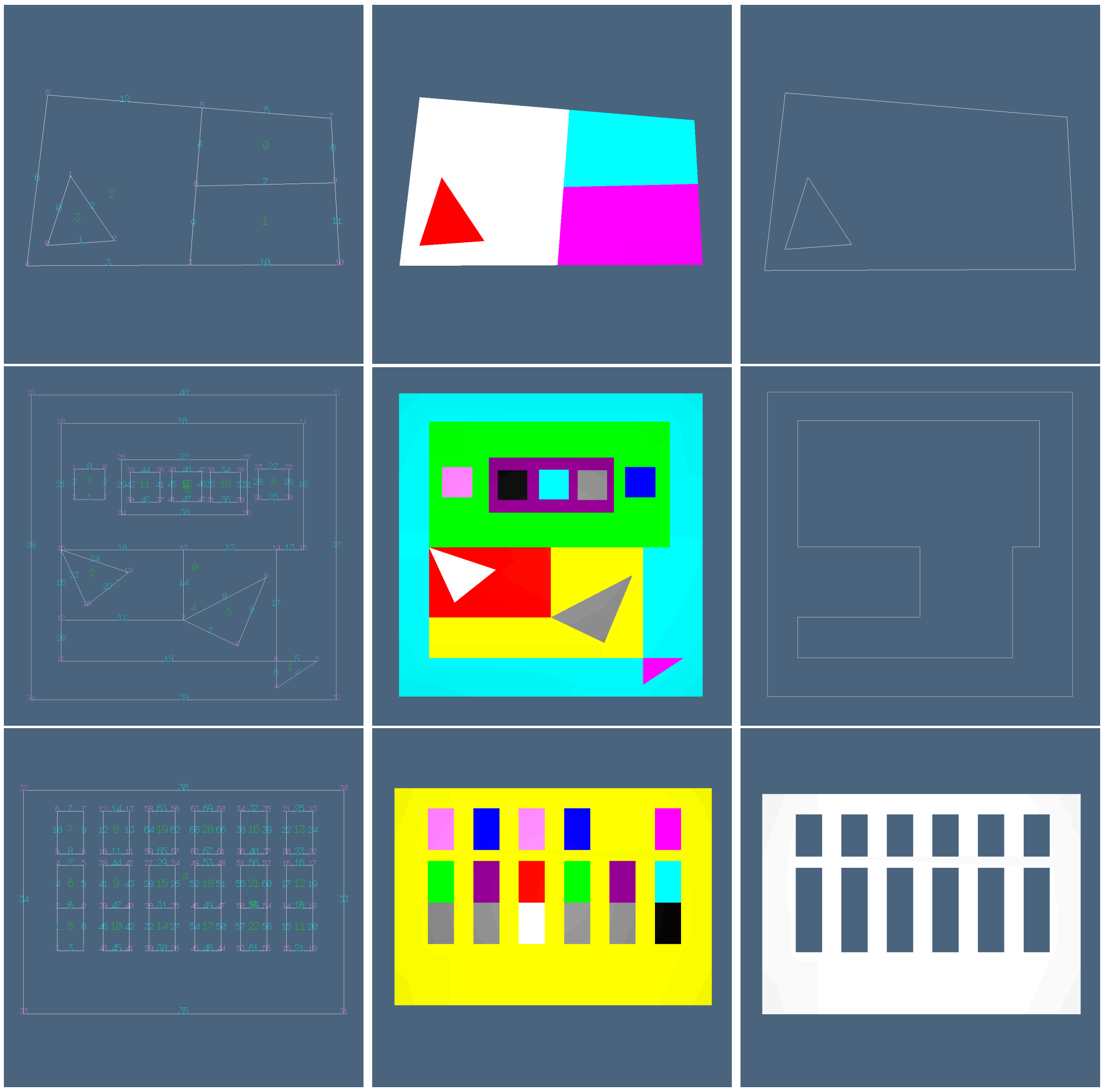

Figure 2: From top to bottom, left to right: (a) numbering of cycles of a 2-complex with a hole in a 2-cell; (b) color display of 2-cells; (c) boundary of the 2-chain $\{f_0, f_1, f_2\}$, codified in coordinates as `[1,1,1,0]`. (d) cycles of a 2-complex with nested 2-cells and non-manifold points; (e) color display of 2-cells; (f) boundary of the 2-chain $\{f_0, f_1, f_2, f_3\}$. (g) The 2-complex modeling a building front; (h) color display of 2-cells; (i) solid display of elementary 2-chain $\{f_4\} \subset$ `FV`, of high topological genus, codified in coordinates by the array $[0, 0, 0, 0, 1, 0, \ldots, 0]$ of length `len(FV)`.

computed by vector summation of the two adjacent tangent vectors incident to the vertex. Each incident vector is computed by properly orienting the vector difference between two consecutive vertices.

For each set of disentangled loops, given as arrays of points without repetition, the appropriate functions of the python package *Poly2tri*, named `p2t`, are called, returning a list of triples of 2D points corresponding to the generated triangulation of the face, possibly holding holes corresponding to the interior loops, if any.

2 Implementation

2.1 Winged-inspired algoritm for computation of all elementary cycles

In this section we introduce a novel implementation of the algorithm for computing all the cycles from a two-dimensional LAR model. It is mostly based on the very simple idea that each edge (i.e. every element of the `EV` list) must be used twice, in the opposite orientations. Therefore, a list `edgeCounts`, indexed by edge indices, stores dinamically the number of times that each edge was already used, of course starting from zero.

Extract all cycles from a LAR pair model The `makeCycles` function takes as input a LAR pair model `(V,EV)`, and returns as output the list of 1-chains `cycles`, implemented as lists of vertex indices. The function starts with initializing the used variables, using as first `edge` of the first cycle the 0 edge, and as first `vertex` its corresponding first vertex. The existence of edges that have not completed their usage is codified by the truth value of the `unusedEdges` predicate. Just notice that the list `VE` of edges inciding on vertices and indexed by vertices is that returned by the `edgeSlopeOrdering` function, that provides a sorting on angles formed with the horizontal line (i.e. on relative slope) of the incident edges.

⟨Extract all cycles from a LAR pair model 10a⟩ ≡
```
""" Extract all cycles from a LAR pair model """
import inters
def makeCycles(theModel):
    #theModel = copy.copy(model)
    V,EV = theModel
    VE = inters.edgeSlopeOrdering(theModel)
    unusedEdges = True
    cycles,ecycles = [],[]
    edgeCounts = [0 for edge in EV]
    edge,vertex = 0,EV[0][0]
    edgeCounts[0] = 1
    while unusedEdges:
        theCycles = extractCycle(edgeCounts,EV,VE,edge,vertex,[vertex],[edge])
        cycles += [theCycles[0]]
```

```
            ecycles += [theCycles[1]]
            unusedEdges,edge,vertex = takeEdgeVertex(edgeCounts,EV)
            edgeCounts[edge] += 1
        return cycles,ecycles
    ◇
```

Macro referenced in 28.

Return a feasible pair edge/vertex The function `takeEdgeVertex` is used to return a new triple of values for the variables `unusedEdges,edge,vertex` used by is calling function `makeCycles`.

⟨ Return a feasible pair edge/vertex 10b ⟩ ≡

```
    """ Return a feasible pair edge/vertex """
    def takeEdgeVertex(edgeCounts,EV):
        e = edgeCounts.index(min(edgeCounts))
        if edgeCounts[e] < 2:
            v = EV[e][0]
            return True,e,v
        else: return False,0,0
    ◇
```

Macro referenced in 28.

Extract a single edge/vertex cycle The `extractCycle` recursive function given below is used to return a new cycle extracted from the `EV` list, and starting with the pair `e,v`. While the abstract formulation of a cycle is a closed 1-chain, its actual implementation here is as an ordered sequence (list) of 0-cells (vertices). Notice that every time an edge `e` is used in a cycle, its orientation is reversed by exchanging the storage positions of its first and second vertex. The base case of the recursion checks if the `nextVertex` to be included in the list is different from the first element of the `cycle`. If equal, the recursion is stopped and the constructed cycle returned.

⟨ Extract a single edge/vertex cycle 11 ⟩ ≡

```
    """ Extract a single edge/vertex cycle """
    def succ(e,seq):
        return seq[(seq.index(e)+1) % len(seq)]

    def extractCycle(edgeCounts,EV,VE,e,v,cycle,ecycle):
        nextEdge = succ(e,VE[v])
        nextVertex, = set(EV[nextEdge]).difference({v})
        if nextVertex != cycle[0]:
            cycle.append(nextVertex)
            ecycle.append(nextEdge)
            edgeCounts[nextEdge] += 1
            v1,v2 = EV[e]
            extractCycle(edgeCounts,EV,VE,nextEdge,nextVertex,cycle,ecycle)
            EV[e] = v2,v1
        return cycle,ecycle
    ◇
```

Macro referenced in 28.

2.2 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 a 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 `computeCycleLattice` 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 `latticeArray` 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 `computeCycleLattice` 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 `latticeArray` matrix — characterizing the incidences between cycles — has dimension $n \times n$.

Remark on interior point vs cycle's vertex classification When reconstructing the containment lattice of component cycles of a boundary chain, the strategy of testing for containment a single vertex of a cycle against another cycle does not work, since the tested vertex might stay in non-manifold position, so that—while internal points stay either internal or external to the target cycle, the result of the containment test would be of type `.on.`, corresponding to test point on the boundary of the target cycle, so producing a wrong containment lattice.

So, in computing the (k, h) term of the containment matrix, stored witin the `latticeArray` below, we cannot in general use the simpler testing `point = V[i]`, but another `point` guaranteed to belong to a close neighborhood of it, in the interior of `EVs[h]`.

For this reason the `internalTo` function is introduced, that returns a `point` that is internal to the `ev` cycle, and in particular internal to the plane sector interior to the angle formed by the first vertex, named `v`, of `ve` (edges incident on vertices) array, that is incident only to two edges (and hence is not in non-manifold position). The function will work with both acute and ottuse interior angles incident on `v`.

⟨ Classification of non intersecting cycles 12 ⟩ ≡
```
    """ Classification of non intersecting cycles """
    def internalTo(V,ev):
        classify = pointInPolygonClassification((V,ev))
        ve = invertRelation(ev)
        for v,edgeIndices in enumerate(ve):
            if len(edgeIndices) == 2: break
        v1,v2 = set(CAT([list(ev[e]) for e in edgeIndices])).symmetric_difference([v])
        vect1 = VECTDIFF([V[v1],V[v]])
        vect2 = VECTDIFF([V[v2],V[v]])
        point = VECTSUM([ V[v], SCALARVECTPROD([ 0.05, VECTSUM([vect1,vect2]) ])])
        if classify(point) == "p_out":
            point = VECTSUM([ V[v], SCALARVECTPROD([ -0.05, VECTSUM([vect1,vect2]) ])])
        return point

    def computeCycleLattice(V,EVs):
        n = len(EVs)
        latticeArray = []
        interiorPoints = [internalTo(V,ev) for k,ev in enumerate(EVs)]
        for k,ev in enumerate(EVs):
            row = []
            classify = pointInPolygonClassification((V,ev))
            for h in range(n):
                i = EVs[h][0][0]
                #point = V[i]
                point = interiorPoints[h]
                test = classify(point)
                if h==k: row += [-1]
                elif test=="p_in": row += [1]
                elif test=="p_out": row += [0]
                elif test=="p_on": row += [-1]
                else: print "error: in cycle classification"
            latticeArray += [row]
        for k in range(n):
            for h in range(k+1,n):
                if latticeArray[k][h] == latticeArray[h][k]:
                    latticeArray[k][h] = 0
        return latticeArray
```
◇

Macro referenced in 28.

Extraction of path-connected boundaries The function `cellsFromCycles`, given by the script below, takes as input the cycle-incidence matrix `latticeArray`, 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 14a ⟩ ≡
```
""" Extraction of path-connected boundaries """
def cellsFromCycles (latticeArray):
    n = len(latticeArray)
    sons = [[h]+[k for k in range(n) if row[k]==1] for h,row in enumerate(latticeArray)]
    level = [sum(col) for col in TRANS(latticeArray)]

    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 28.

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" 14b ≡
     """ 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)

     latticeArray = computeCycleLattice(V,EVs)

     for k in range(len(latticeArray)):
        print k,latticeArray[k]

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

2.3 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 4).

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.

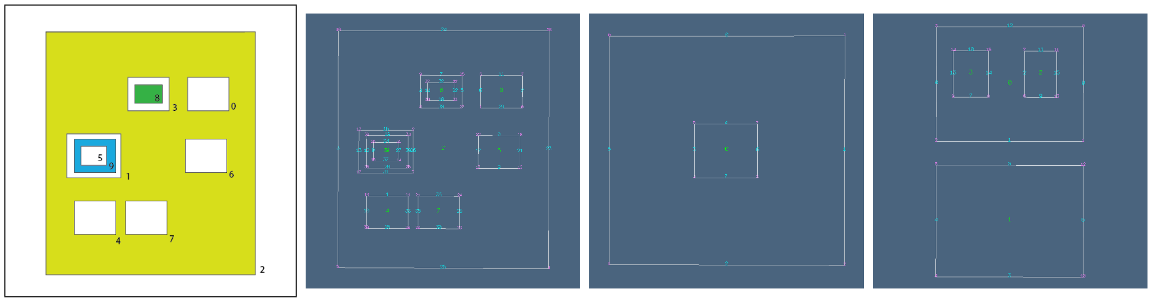

Figure 3: Some examples of nested non intersecting cycles. The corresponding solutions are given in the text.

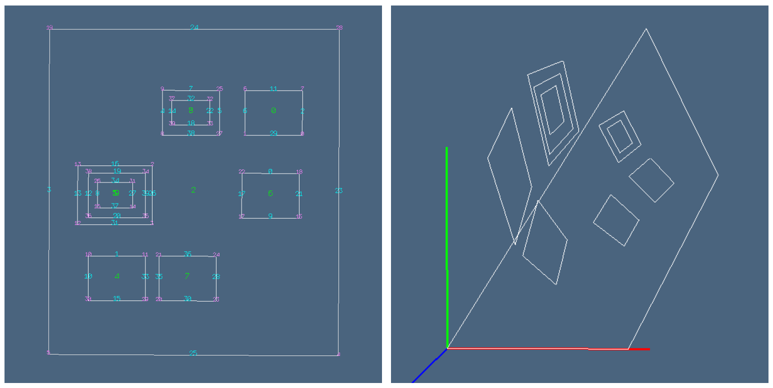

Figure 4: A set of non-intersecting boundary cycles in Cartesian and polar coordinates, using (improperly) an Euclidean metric in the transformed space.

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 4. It is only used here to put the vertex coordinates in general position, so simplifying the scan-line algorithm.

⟨ Transforming to polar coordinates 15 ⟩ ≡

```
""" 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 28.

Scan line algorithm

⟨ Scan line algorithm 16 ⟩ ≡

```
""" 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)):
                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 28.

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 17 ⟩ ≡

```
""" Scan line algorithm input/output """
def connectTheDots(model):
    V,EV = model
    V,EVs = biconnectedComponent((V,EV))
    FV = AA(COMP([sorted,set,CAT]))(EVs)
    latticeArray = computeCycleLattice(V,EVs)
    cells = cellsFromCycles(latticeArray)
    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 28.

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 18a ⟩ ≡

```
""" 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):
    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]
    rotations = [cycle.index(min(cycle)) for cycle in vertexCycles]
    theCycles = sorted([rotatePermutation(perm,n) for perm,n in zip(vertexCycles,rotations)])
    CVs = [[theCycles[cycle] for cycle in cell] for cell in cells]
    areas = signedSurfIntegration((V,theCycles,EV),signed=True)

    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)
    return CVs
◇
```

Macro referenced in 28.

From nested boundary cycles to triangulation

⟨From nested boundary cycles to triangulation 18b⟩ ≡

```
""" 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 28.

2.4 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 20a ⟩ ≡

```
    """ 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 28.

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 20b ⟩ ≡

```
""" 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 28.

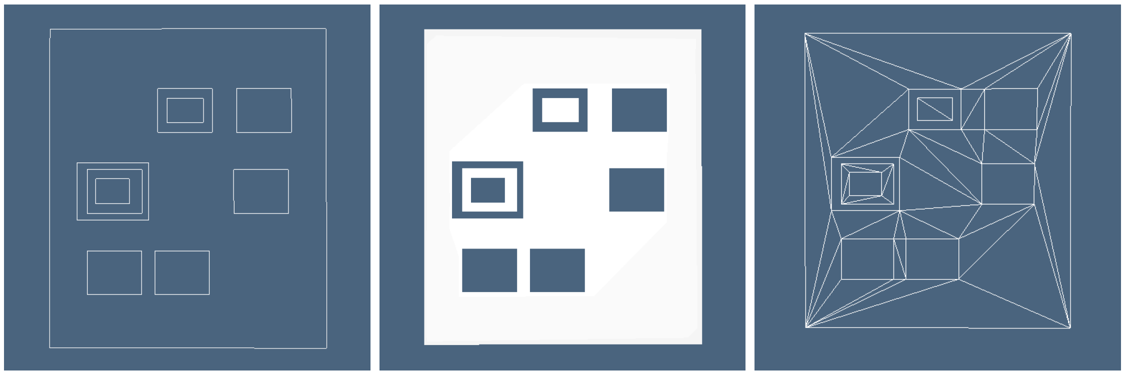

Figure 5: (a) Closed 1-chain c, i.e. such that $\partial\, c = 0$; c is both cycle and boundary at the same time; (b) 2-chain h such that $\partial\, h = c$; (c) 1-skeleton of a triangulation of h.

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 21 ⟩ ≡

```
    """ 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 28.

From Struct object to LAR boundary model

⟨From Struct object to LAR boundary model 23⟩ ≡

```
""" 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 28.

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 24a⟩ ≡

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

Macro referenced in 28.

From LAR oriented boundary model to polylines

⟨From LAR boundary model to polylines 24b⟩ ≡

```
    """ 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 28.

2.5 Visualization of a 2D complex and 2-chain

⟨Visualization of a 2D complex and 2-chain 25⟩ ≡

```
""" 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 28.

2.6 Point in polygon classification

⟨ Point in polygon testing 26a ⟩ ≡

```
    """ Point-in-polygon classification algorithm """
    ⟨ Half-line crossing test 26c ⟩
    ⟨ Tile codes computation 26b ⟩
    ⟨ Point-in-polygon classification algorithm 27 ⟩
    ◇
```

Macro referenced in 28.

Tile codes computation

⟨ Tile codes computation 26b ⟩ ≡

```
    """ 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 26a.

Half-line crossing test

⟨ Half-line crossing test 26c ⟩ ≡

```
    """ 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 26a.

Point in polygon testing

⟨ Point-in-polygon classification algorithm 27 ⟩ ≡

```
""" 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 26a.

3 Exporting the library

"larlib/larlib/triangulation.py" 28 ≡

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

⟨Return a feasible pair edge/vertex 10b⟩
⟨Extract a single edge/vertex cycle 11⟩
⟨Extract all cycles from a LAR pair model 10a⟩
⟨Edge cycles associated to a closed chain of edges 21⟩
⟨Point in polygon testing 26a⟩
⟨Classification of non intersecting cycles 12⟩
⟨Extraction of path-connected boundaries 14a⟩
⟨Transforming to polar coordinates 15⟩
⟨Scan line algorithm 16⟩
⟨Scan line algorithm input/output 17⟩
⟨Orientation of component cycles of unconnected boundaries 18a⟩
⟨From nested boundary cycles to triangulation 18b⟩
⟨Generation of 1-boundaries as vertex permutation 20a⟩
⟨Wire-frame LAR to boundary polygons 20b⟩
⟨From Struct object to LAR boundary model 23⟩
⟨From structures to boundary polylines 24a⟩
⟨From LAR boundary model to polylines 24b⟩
⟨Computing a LAR 2-complex from an arrangement of line segments 4⟩
⟨Visualization of a 2D complex and 2-chain 25⟩
⟨Solid PyPLaSM visualization of a 2-complex 6a⟩
◇
```

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 2.3.

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" 30a ≡
```
""" 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)

      latticeArray = computeCycleLattice(V,EVs)

      for k in range(len(latticeArray)):
         print k,latticeArray[k]

      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" 30b ≡

```
      """ 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(latticeArray)
      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)
      cycles = STRUCT(MKPOLS((V,EV)))
      csrBoundaryMat = boundary(FV,EV)

      n = len(cells)
      chains = allBinarySubsetsOfLenght(n)
      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 6 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.

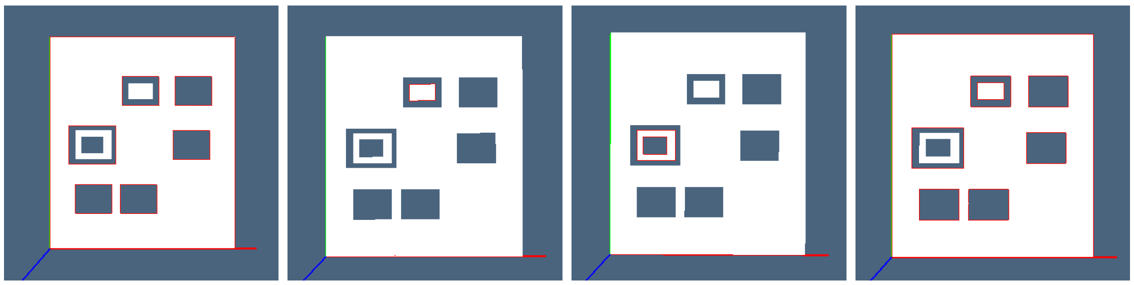
Figure 6: Some examples of boundaries (in red) of 2-chains from nested non intersecting cycles.

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" 32a $\equiv$

```
""" 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" 32b ≡

```
""" 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)
◇
```

"test/py/triangulation/test05.py" 32c ≡

```
""" 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" 33a ≡

```
""" 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" 33b ≡

```
from larlib import *

⟨random 1-boundary generation 34⟩
◇
```

random 1-boundary generation

⟨random 1-boundary generation 34⟩ ≡

```
""" random 1-boundary generation """
from larlib import *
```

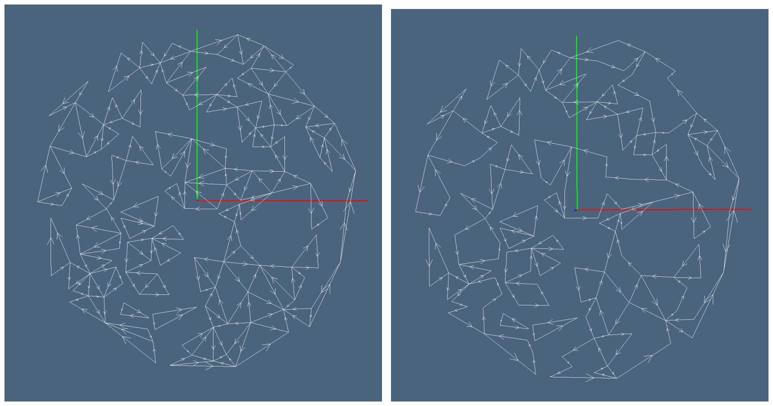

Figure 7: example caption

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

EV = AA(list)(cells)
V,FV,EV = larPair2Triple((V,EV))

bcycles,bverts = boundaryCycles(range(len(EV)),EV)
VIEW(STRUCT(AA(POLYLINE)([[V[v] for v in verts+[verts[0]]] for verts in bverts])))

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

Macro referenced in 33b.

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.