Linear Algebraic Representation

Alberto Paoluzzi, Francesco Furiani, Giulio Martella

November 11, 2017

Contents

I Introduction **1**

1 Introduction **3**
1.1 Boolean Operations . . . 3
1.2 LAR . . . 4
1.2.1 Historical notes . . . 4
1.3 Literate programming . . . 5
1.4 Julia . . . 5

2 The algorithm **7**
2.1 Overview . . . 7
2.2 The "1-cells in $\mathbb{E}^2$" base case . . . 8

II Implementation **11**

3 Module overview **13**
3.1 Standard types . . . 13
3.1.1 Floating point error . . . 14
3.2 Notes on variables names . . . 14

4 Spatial Arrangement **15**
4.1 Overview . . . 15
4.2 Coincident vertices merge . . . 16
4.3 3-cells creation . . . 19

5 Planar Arrangement **21**
5.1 Overview . . . 21
5.1.1 Tests . . . 22
5.2 Edge fragmentation . . . 22
5.2.1 Support function . . . 22
5.2.2 Edge intersections . . . 23
5.2.3 Implementation . . . 25
5.2.4 Tests . . . 26
5.3 Coincident vertices merge . . . 27

5.3.1 Implementation . . . 28
5.3.2 Tests . . . 28
5.4 Delete edges outside σ area . . . 29
5.5 Maximal biconnected components . . . 30
5.5.1 Support function . . . 30
5.5.2 Implementation . . . 33
5.5.3 Tests . . . 34
5.6 Faces creation . . . 35
5.6.1 Implementation . . . 35
5.6.2 Individuate the external cell . . . 35
5.6.3 Containment test . . . 36
5.6.4 Transitive reduction . . . 38
5.6.5 Cell merging . . . 38
5.6.6 Tests . . . 40

6 Dimension travel **47**
6.1 Overview . . . 47
6.1.1 Tests . . . 47
6.2 Submanifold mapping . . . 47
6.2.1 Tests . . . 48
6.3 Spatial index computation . . . 48
6.4 Face intersection with $x_3 = 0$ plane . . . 49

7 Minimal cycles computation **53**
7.1 Main function . . . 53
7.2 Dimensional wise implementations . . . 56
7.2.1 $d = 2$ . . . 56
7.2.2 $d = 3$ . . . 57

8 Utilities **61**
8.1 Overview . . . 61
8.1.1 Tests . . . 61
8.2 Bounding boxes . . . 61
8.2.1 Tests . . . 62
8.3 Face area calculation . . . 63
8.3.1 Tests . . . 64
8.4 Skeletal merge . . . 64
8.5 Edge deletion . . . 65
8.6 FV building . . . 65
8.7 Boundaries building . . . 66
8.8 Vertex equality utilities . . . 68
8.9 Full triangulation . . . 68
8.10 OBJ I/O . . . 69
8.11 Point in face area . . . 71

III Tests and conclusions **75**

9 Tests and examples **77**
9.1 Planar arrangement tests 77
9.2 Spatial arrangement tests 82

10 Conclusions **87**
10.1 Future developments 87
10.1.1 Parallelization 87
10.1.2 Boolean operations 87
10.1.3 Handling of 3-cells with non-intesecting shells 88
10.1.4 LAR 88

Part I

Introduction

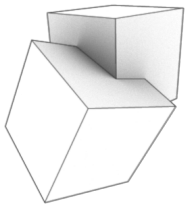
Chapter 1

Introduction

This thesis presents a Julia implementation of a novel algorithm to combine two cellular complexes, guaranteeing a minimal fragmentation of the resulting complex [12]. The algorithm has several applications, of which the most common and obvious one are the Boolean operations on solids. The whole system is based on LAR [6], a very general and versatile geometrical representation scheme.

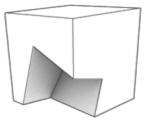
1.1 Boolean Operations

Boolean operations are a set of operations between solids. They took two solids and return one. The common three operations are[1]:



$A \cup B$

$A \cap B$



$A - B$

Boolean operations are very used through computer graphics, both in computer aided design and graphics for entertainment. For this reason, since the dawn of computer graphics, several algorithms for Boolean operations have been developed and implemented. But most of them have recurrent problems: the

[1] On the top of them many other operations can be built using De Morgan's Laws. For example, the exclusive disjunction (XOR) can be expressed as $(A \cup B) - (A \cap B)$

big ones are the excessive fragmentation of the cells in output and the huge conceptual complexity of the implementations. This happens partially due to the inadequacy of mainstream geometric representations in handling non-manifold solids, which are a common output of Boolean operations; that's why our algorithm takes heavily advantage from the LAR scheme.

1.2 LAR

LAR is a general representation scheme for geometric and topological modeling. The domain of the scheme is provided by *cellular complexes* while its codomain is a set of *sparse matrices*. The main advantages of the scheme are:

1. *It is extremely effective to easily represent general non-manifold solids.* For example, the memory representation of a $d = 3$ cellular complex using LAR consists in only two binary sparse matrices for the topology and a bi-dimensional array for the geometry.

2. *Computation and analysis of cellular complexes is done only through easy linear algebra operations.* The most common operation is the sparse matrix-vector multiplication.

In LAR we talk about cellular complexes which are made of cells, so we call d-cell the d-dimensional cell: 0-cells are vertices, 1-cells are edges, 2-cells are faces, and so on. Throughout this thesis, these names are completely interchangeable.

An important concept is represented by the *boundary* and *coboundary operators*. They express the relation between the cells of different dimension but of the same cellular complex. Even these operators are stored in memory as sparse matrices and they can be applied using just a matrix multiplication.

The relation $\partial_d = \delta_{d-1}^\top$ (where ∂_d is the d-boundary and δ_{d-1} is the $(d-1)$-coboundary) is particularly handy. So, for example a 2-boundary expresses the relation from the edges to the faces of the same complex and its transpose is the 1-coboundary that maps faces to edges.

An another concept of LAR used a lot in this thesis is the one of *skeleton*. A $(d-1)$-skeleton is the set of $(d-1)$-cells of a d-complex. For example, a 2-skeleton of a 3-complex is the set of all the faces of the complex.

1.2.1 Historical notes

LAR has been developed for several years, in a joint collaboration between Roma Tre University and the University of Wisconsin at Madison [5]. The development of a Python prototype start in 2012 by A. Paoluzzi but was interrupted in December 2016 for various reasons. The development of the current Julia implementation started few months later (March 2017) with G. Martella and F. Furiani as main developers. This thesis is the main core of the Julia implementation.

1.3 Literate programming

This thesis has been written using literate programming. Literate programming is a programming paradigm in which the program logic is explained in natural language and the code is embedded in macros. Quoting Donald E. Knuth, the creator of the paradigm: *"[Literate programming] allows a person to express programs in a stream of consciousness order. [...] [Code can] be explored in a psychologically correct order"* [10]. With this premise it is easy to understand why literate programming is widely used for academic works. When the goal is to learn and share knowledge, literate programming fits perfectly.

1.4 Julia

Julia is a relatively new high-level programming language targeted to numerical computing. The project was born back in 2009 and its first stable version was released in 2012. As stated in the first blog post on Julia's official website, the language has the goal to be *"Something that is dirt simple to learn, yet keeps the most serious hackers happy"*, with the speed of C, the dynamism of Ruby and the distributed power of Hadoop [2].

We choose Julia manly because of its elegance and simplicity: using a lower level programming language would have faded the underlying mathematical elegance of the algorithm.

Chapter 2

The algorithm

2.1 Overview

The algorithm is based on the concept of recursive problem simplification (a sort of *divide et impera* philosophy); if we have a d-complex, for every $(d-1)$-cell embedded into the $\mathbb{E}^d$ euclidean space, we bring the cell, and every other cell that could intersect it, down into $\mathbb{E}^{d-1}$. We do this until we reach the $d = 1$ in $\mathbb{E}^1$ case; in here, we fragment all the 1-cells. Then, we travel back to the original d-dimension, and, for each dimensional step, we build correct complexes from cells provided by the fragmentation of the lower dimension.

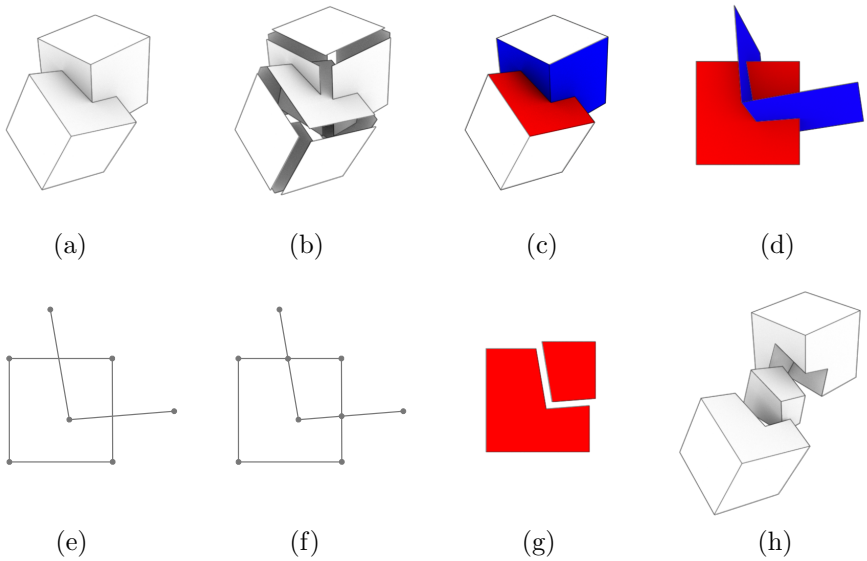


Figure 2.1: Algorithm overview

We have in input two cellular complexes [fig. 2.1, a], given as 2-skeletons, which are the sets of 2-cells [fig. 2.1, b, exploded]. Once we merged the skeletons [ref. 8.4], we individuate for each 2-cell (that we will call σ) all the other cells that could intersect it. We do this by computing the spatial index: it is a mapping $\mathcal{I}(\sigma)$ from a cell σ to every other cell τ of which $box(\sigma) \cap box(\tau) \neq \emptyset$, where the *box* function provides the axis aligned bounding box (AABB) of a cell [fig. 2.1, c, σ in red and $\mathcal{I}(\sigma)$ in blue]. The spatial arrangement calculation is speeded up by storing the AABBs as dimensional wise intervals into an interval tree [14]. Now for each cell σ we transform $\sigma \cup \mathcal{I}(\sigma)$ in a way that σ lays on the $x_3 = 0$ plane [fig. 2.1, d] and we find the intersections of the $\mathcal{I}(\sigma)$ cells with $x_3 = 0$ plane. So we have a "soup" of 1-cells in $\mathbb{E}^2$ [fig. 2.1, e], and we fragment each 1-cell with every other cell obtaining a valid 1-skeleton [fig. 2.1, f]. From this data it is possible to build the 2-cells using the ALGORITHM 1 presented and explored by Paoluzzi et al. [12] [fig. 2.1, g, exploded]. The procedure to fragment 1-cells on a plane and return a 2-complex is called *planar arrangement* and it is presented more in detail in the next section. When the planar arrangement is complete, fragmented σ can be transformed back to its original position in $\mathbb{E}^3$. With every 2-cell correctly fragmented, we can use the already cited ALGORITHM 1 again to build a full 3-complex[1] [fig. 2.1, h, exploded].

2.2 The "1-cells in $\mathbb{E}^2$" base case

This is our base case. We have called *planar arrangement* the procedure to handle this case since it literally arranges a bunch of edges laying on a plane. So, in input there are 1-cells in $\mathbb{E}^2$ and, optionally (but very likely), the boundary of the original 2-cell σ [fig. 2.2, a, σ in red]. We consider each edge and we fragment it with every other edge. This brings to the creation of several coincident vertices: these will be eliminated using a KD-Tree [fig. 2.2, b, exploded]. At this point we have a perfectly fragmented 1-complex but many edges are superfluous and must be eliminated; two kind of edges are to discard: the ones outside the area of σ and the ones which are not part of a maximal biconnected component [ref. 5.5.1]. The result of this edge pruning outputs a 1-skeleton [fig. 2.2, c, exploded].

After this, 2-cells must be computed: for each connected component[2] we build a containment tree, which indicates which component is spatially inside an other component. Computing these relations, let us launch the ALGORITHM 1 [12] on each component and then combine the results to create 2-cells with non-intersecting shells[3] [fig. 2.2, d, 2-cells numbered in green; please note that cell 2 has cell 1 as an hole].

[1] This is possible because ALGORITHM 1 is (almost) dimension independent [ref. 7].

[2] It is legit to talk about a 1-skeleton as a graph: 0-cells are nodes, 1-cells are edges and the boundary operator is a incidence matrix.

[3] A 2-cell with a non-intersecting shell can be trivially defined as a "face with holes"; the correct definition is that it cannot be shrunk to the dimension of a point.

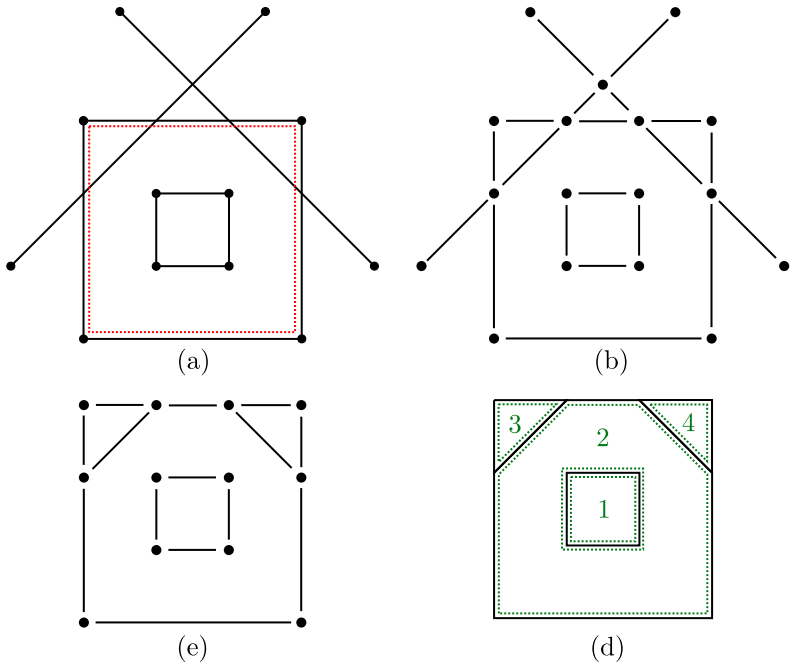


Figure 2.2: Planar arrangement overview

Part II

Implementation

Chapter 3

Module overview

We structured our code in a Julia module called `LARLIB`. The common user will mostly care about only three functions:

- `LARLIB.skel_merge`: Provides the skeletal merge between two 1-skeletons or 2-skeletons
- `LARLIB.spatial_arrangement`: Arranges one 2-skeleton in $\mathbb{E}^3$ passed as an array of vertices, and two boundary matrices.
- `LARLIB.planar_arrangement`: Arranges one 1-skeleton in $\mathbb{E}^2$ passed as an array of vertices and a boundary matrix.

```
"lib/jl/LARLIB.jl" 1 ≡
    module LARLIB

        ⟨ LAR imports 6, ... ⟩
        ⟨ LAR types 2 ⟩

        include("./utilities.jl")
        include("./minimal_cycles.jl")
        include("./dimension_travel.jl")
        include("./planar_arrangement.jl")
        include("./spatial_arrangement.jl")

    end
    ◇
```

3.1 Standard types

We define at the top of our module the standard types that will be used throughout LAR. As already explained in the introduction [ref. 1.2], LAR needs only one bi-dimensional array to store geometry and one or more sparse matrices for topology. Julia has already implemented CSC sparse matrices in its standard library so we are going to use them.

⟨ LAR types 2 ⟩ ≡
```
const Verts = Array{Float64, 2}
const Cells = SparseMatrixCSC{Int8, Int}
const Cell = SparseVector{Int8, Int}
```
◇
Macro referenced in 1.

We used the general name `Cells`, but we are going to use this type also for boundaries.

3.1.1 Floating point error

We stored geometry using 64-bit IEEE floats. As it is known, floating point arithmetic is not precise and introduces numerical errors. Usually this is not an issue[1], but when precision is a goal, floating point error must be handled very carefully. During the development we encountered several numerical problems and we tried various approaches (like normalizing the geometry inside the $[0, 1]$ interval for each dimension in order to maximize the significand of the floating-point numbers) but most of them turned out to be unstable. So we choose the less orthodox path we could possibly take: we set a fixed error and we performed every floating point comparison using this error. Examples of this "tweak" are to be found in 5.2.2, 6.4, 7.2.2 and 8.8.

3.2 Notes on variables names

Here a list of some often used variable names.

- `V:` Bi-dimensional array (`Array{Float64, 2}`) that keeps the geometry of a complex. Its dimensions are $n \times d$, where n is the number of vertices and d is the dimension of the euclidean space in which the complex is embedded.
- `EV:` 1-boundary. It is a $m \times n$ sparse matrix (`SparseMatrixCSC{Int8, Int}`) where m is the number of edges and n is the number of vertices. The possible values are $0, 1$ and -1.
- `FE:` 2-boundary. Same as `EV`, but faces on the rows and edges on the columns.
- `CF:` 3-boundary. Same as `EV`, but 3-cells on the rows and faces on the columns.

[1] The *machine epsilon*, which is the upper bound on the relative error in floating-point arithmetic, for double precision IEEE floating-point numbers is $2^5 3 \approx 1.11 \times 10^{-16}$.

Chapter 4

Spatial Arrangement

4.1 Overview

Here we present the spatial arrangement algorithm. It has been explained in the introduction [ref. 2.1].

"lib/jl/spatial_arrangement.jl" 3 ≡

⟨ spatial_arrangement support functions 5 ⟩

```
function spatial_arrangement(V::Verts, EV::Cells, FE::Cells)
    vs_num = size(V, 1)
    es_num = size(EV, 1)
    fs_num = size(FE, 1)

    sp_idx = spatial_index(V, EV, FE)

    rV = Verts(0,3)
    rEV = spzeros(Int8,0,0)
    rFE = spzeros(Int8,0,0)

    for sigma in 1:fs_num
        println(sigma, "/", fs_num)

        sigmavs = (abs(FE[sigma:sigma,:])*abs(EV))[1,:].nzind
        sV = V[sigmavs, :]
        sEV = EV[FE[sigma, :].nzind, sigmavs]
```

⟨ Sigma flattening 4 ⟩

```
        nV, nEV, nFE = planar_arrangement(sV, sEV, sparsevec(ones(Int8, length(sigmavs))))

        if nV == nothing
            continue
        end
```

```
            nvsize = size(nV, 1)
            nV = [nV zeros(nvsize) ones(nvsize)]*inv(M)[:, 1:3]

            rV, rEV, rFE = skel_merge(rV, rEV, rFE, nV, nEV, nFE)
        end

        rV, rEV, rFE = merge_vertices(rV, rEV, rFE)

        ⟨ Create 3-cells 10 ⟩

        return rV, rEV, rFE, rCF
    end

    ◇
```

To flatten the 2-cell σ on the $x_3 = 0$ plane, we build a linear transformation matrix with the `submanifold_mapping` utility [ref. 6.2], we transform the geometry and we intersect every cell in $\mathcal{I}(\sigma)$ [ref. 2.1] with the $x_3 = 0$ plane using `face_int` [ref. 6.4].

⟨ Sigma flattening 4 ⟩ ≡

```
    M = submanifold_mapping(sV)
    tV = ([V ones(vs_num)]*M)[:, 1:3]

    sV = tV[sigmavs, :]

    for i in sp_idx[sigma]
        tmpV, tmpEV = face_int(tV, EV, FE[i, :])

        sV, sEV = skel_merge(sV, sEV, tmpV, tmpEV)
    end

    sV = sV[:, 1:2]
    ◇
```

Macro referenced in 3.

4.2 Coincident vertices merge

The merge of coincident is done in the `merge_vertices` function.

⟨ spatial_arrangement support functions 5 ⟩ ≡

```
    function merge_vertices(V::Verts, EV::Cells, FE::Cells, err=1e-4)
        vertsnum = size(V, 1)
        edgenum = size(EV, 1)
        facenum = size(FE, 1)
        newverts = zeros(Int, vertsnum)
        kdtree = KDTree(V')
```

```
    ⟨ Find coincident vertices 7 ⟩
    ⟨ Merge edges 8 ⟩
    ⟨ Merge faces 9 ⟩

    return nV, nEV, nFE
end
```
◇

Macro referenced in 3.

First of all we need to find vertices which are near enough to be considered coincident. We perform this operation relying on the `NearestNeighbors.jl` package [4] which provides a rather good implementation of the `KDTree` data structure.

So, we identify the vertices to delete and we store a map from original vertices to new vertices. In the meanwhile we built a list of vertices to delete and we delete them as soon as possible.

⟨ LAR imports 6 ⟩ ≡

```
using NearestNeighbors
```
◇

Macro defined by 6, 22, 58, 73.
Macro referenced in 1.

⟨ Find coincident vertices 7 ⟩ ≡

```
todelete = []

i = 1
for vi in 1:vertsnum
    if !(vi in todelete)
        nearvs = inrange(kdtree, V[vi, :], err)

        newverts[nearvs] = i

        nearvs = setdiff(nearvs, vi)
        todelete = union(todelete, nearvs)

        i = i + 1
    end
end

nV = V[setdiff(collect(1:vertsnum), todelete), :]
```
◇

Macro referenced in 5, 23.

To delete the edges we write them as couples of vertex indices. We keep them in two versions: in `edges` we put the edges described with the indexes of the new vertices and in `oedges` we put the edges relative to the original vertex indices (we will use them when merging faces). Once we "translated" the edges, we delete the duplicates (using a set union) and the degenerated edges. Lastly we build a new EV matrix (called `nEV`). While we build the matrix, we also build

a dictionary which maps edges expressed as couples of vertex indices into edge indices relative to `nEV`; this data will be used in the $d = 2$ version of this function [ref. 5.3].

⟨ Merge edges 8 ⟩ ≡

```
edges = Array{Tuple{Int, Int}, 1}(edgenum)
oedges = Array{Tuple{Int, Int}, 1}(edgenum)

for ei in 1:edgenum
    v1, v2 = EV[ei, :].nzind

    edges[ei] = Tuple{Int, Int}(sort([newverts[v1], newverts[v2]]))
    oedges[ei] = Tuple{Int, Int}(sort([v1, v2]))

end
nedges = union(edges)
nedges = filter(t->t[1]!=t[2], nedges)

nedgenum = length(nedges)
nEV = spzeros(Int8, nedgenum, size(nV, 1))

etuple2idx = Dict{Tuple{Int, Int}, Int}()

for ei in 1:nedgenum
    nEV[ei, collect(nedges[ei])] = 1
    etuple2idx[nedges[ei]] = ei
end
```

◇

Macro referenced in 5, 23.

To merge the faces, we convert them into a lists of edges (represented as a couple of vertices). We then remove duplicated faces by checking which faces use the same vertices. At the end, we use the maps built during vertices and edges merge to rebuild the `FE` matrix correctly using the new vertex indices.

⟨ Merge faces 9 ⟩ ≡

```
faces = [[
    map(x->newverts[x], FE[fi, ei] > 0 ? oedges[ei] : reverse(oedges[ei]))
    for ei in FE[fi, :].nzind
] for fi in 1:facenum]


visited = []
function filter_fn(face)

    verts = []
    map(e->verts = union(verts, collect(e)), face)
    verts = Set(verts)

    if !(verts in visited)
```

```
            push!(visited, verts)
            return true
        end
        return false
    end

    nfaces = filter(filter_fn, faces)

    nfacenum = length(nfaces)
    nFE = spzeros(Int8, nfacenum, size(nEV, 1))

    for fi in 1:nfacenum
        for edge in nfaces[fi]
            ei = etuple2idx[Tuple{Int, Int}(sort(collect(edge)))]
            nFE[fi, ei] = sign(edge[2] - edge[1])
        end
    end
```

◇

Macro referenced in 5.

4.3 3-cells creation

⟨ Create 3-cells 10 ⟩ ≡

⟨ Compute connected components ? ⟩

⟨ Compute containment tree ? ⟩

⟨ Cell union ? ⟩

```
rCF = minimal_3cycles(rV, rEV, rFE)
```

◇

Macro referenced in 3.

Chapter 5

Planar Arrangement

5.1 Overview

The planar arrangement has been already explained in the introduction [ref. 2.2]. In the implementation we also build and return a map from the original edges to the new ones: this is necessary infrastructure to later implement boolean operations with ease.

"lib/jl/planar_arrangement.jl" 11 ≡
```
⟨ planar_arrangement support functions 14, ... ⟩

function planar_arrangement(V::Verts, EV::Cells, sigma::Cell=spzeros(Int8, 0))
    edgenum = size(EV, 1)
    ⟨ planar_arrangement local variables 12, ... ⟩

    for i in 1:edgenum
        ⟨ Fragment edge 18 ⟩
    end
    ⟨ Put fragmentation results together 20 ⟩

    ⟨ Merge coincident vertices 25 ⟩
    ⟨ Delete edges outside sigma area 27 ⟩
    ⟨ Find maximal biconnected components 35 ⟩
    ⟨ Filter biconnected components 36 ⟩
    ⟨ Create faces 38 ⟩

    V, EV, FE, edge_map
end
```
◇

The mapping from old edges to new ones is stored into `edge_map`.

⟨ planar_arrangement local variables 12 ⟩ ≡
```
edge_map = Array{Array{Int, 1}, 1}(edgenum)
```
◇

Macro defined by 12, 19.
Macro referenced in 11.

5.1.1 Tests

Every function responsible for the planar arrangement is coupled by some tests.

"test/jl/planar_arrangement.jl" 13 ≡

```
using Base.Test
using LARLIB

⟨ planar_arrangement support functions tests 21, ... ⟩
◇
```

General tests are defined in 9.1.

5.2 Edge fragmentation

5.2.1 Support function

The edge fragmentation is performed by using a function called `frag_edge`. It fragments the edge of index `edge_idx` computing the intersections of it with the other edges of the complex. It returns the updated vertices list and the freshly computed edges. For every edge, it needs to check if the edge to fragment intersects with it. The actual edge intersections are computed by `intersect_edges` function [ref. 5.2.2] The intersection points are then sorted along the edge to fragment, and correct fragments (which are edges themselves) are computed.

⟨ planar_arrangement support functions 14 ⟩ ≡

```
function frag_edge(V::Verts, EV::Cells, edge_idx::Int)
    alphas = Dict{Float64, Int}()
    edge = EV[edge_idx, :]
    for i in 1:size(EV, 1)
        if i != edge_idx
            intersection = intersect_edges(V, edge, EV[i, :])
            for (point, alpha) in intersection
                V = [V; point]
                alphas[alpha] = size(V, 1)
            end
        end
    end

    alphas[0.0], alphas[1.0] = edge.nzind

    alphas_keys = sort(collect(keys(alphas)))
    edge_num = length(alphas_keys)-1
    verts_num = size(V, 1)
    EV = spzeros(Int8, edge_num, verts_num)

    for i in 1:edge_num
        EV[i, alphas[alphas_keys[i]]] = 1
        EV[i, alphas[alphas_keys[i+1]]] = 1
    end
```

```
    V, EV
end
```
◇

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

5.2.2 Edge intersections

Three major cases are to be considered when intersecting two edges:

1. They are not parallel
2. They are colinear (they stand on the same line)
3. They are parallel but not colinear

In the third case there will be no intersections for sure so this case is skipped. When they are not parallel there will be no more than a single intersection; in this case we use the method presented by Bourke [3] to calculate it. Particular attention is needed on the case of colinear edges: it can happen that `edge2` is contained into the bounds of the colinear `edge1`; in this case, both points of `edge2` are to be considered intersection and hence must be returned. Because of this, the intersections are returned as a list than can contain from zero to two elements; each element is a couple containing the intersection point and a parameter useful for sorting the fragmentation points of an edge.

Here we are doing floating-point numbers comparisons so we use a fixed error to avoid numerical imprecisions [ref. 3.1.1].

⟨ planar_arrangement support functions 15 ⟩ ≡

```
function intersect_edges(V::Verts, edge1::Cell, edge2::Cell)
    err = 10e-8

    x1, y1, x2, y2 = vcat(map(c->V[c, :], edge1.nzind)...)
    x3, y3, x4, y4 = vcat(map(c->V[c, :], edge2.nzind)...)
    ret = Array{Tuple{Verts, Float64}, 1}()

    v1 = [x2-x1, y2-y1];
    v2 = [x4-x3, y4-y3];
    v3 = [x3-x1, y3-y1];

    ⟨ Check if colinear or parallel 16 ⟩

    if colinear
        ⟨ Handle colinear edges 17 ⟩
    elseif !parallel
        denom = (v2[2])*(v1[1]) - (v2[1])*(v1[2])
        a = ((v2[1])*(-v3[2]) - (v2[2])*(-v3[1])) / denom
        b = ((v1[1])*(-v3[2]) - (v1[2])*(-v3[1])) / denom

        if -err < a < 1+err && -err <= b <= 1+err
```

```
            p = [(x1 + a*(x2-x1))  (y1 + a*(y2-y1))]
            push!(ret, (p, a))
        end
    end

    return ret
end
```
◇

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

To check if edges are parallel, we check with the dot product the parallelism between the edges defining vectors. Edges are colinear if they are parallel and the points of the second edge stand on the line of the first edge or one of the points of the second edge is coincident to one point of the first one.

⟨ Check if colinear or parallel 16 ⟩ ≡
```
ang1 = dot(normalize(v1), normalize(v2))
ang2 = dot(normalize(v1), normalize(v3))

parallel = 1-err < abs(ang1) < 1+err
colinear = parallel && (1-err < abs(ang2) < 1+err || -err < norm(v3) < err)
```
◇

Macro referenced in 15.

In the case of colinearity, to find if `edge2` has one or both of its vertices inside `edge1` we follow this procedure:

1. We parametrize `edge1`:

$$p = p_1 + \alpha(p_2 - p_1), \quad \alpha \in [0, 1]$$

   Where p_1 and p_2 are the vertices of `edge1`

2. We solve for α:

$$\begin{gathered} o = p_1, \quad \vec{v} = p_2 - p_1 \\ p = o + \alpha\vec{v} \\ p - o = \alpha\vec{v} \\ \vec{v}^\top \cdot (p - o) = \alpha(\vec{v}^\top \cdot \vec{v}) \\ \alpha = \frac{\vec{v}^\top \cdot (p - o)}{\vec{v}^\top \cdot \vec{v}} \end{gathered}$$

3. We replace p of the last equation with both the vertices of `edge2`. If the result is $\in [0, 1]$ then an intersection is found.

⟨ Handle colinear edges 17 ⟩ ≡

```
o = [x1 y1]
v = [x2 y2] - o
alpha = 1/dot(v,v')
ps = [x3 y3; x4 y4]
for i in 1:2
    a = alpha*dot(v',(reshape(ps[i, :], 1, 2)-o))
    if 0 < a < 1
        push!(ret, (ps[i:i, :], a))
    end
end
```
◇

Macro referenced in 15.

5.2.3 Implementation

When we need to fragment an edge we just use the `frag_edge` function [ref. 5.2.1] and we update data and store the changes. While we fragment the edges, we also build a temporary version of `edge_map`[ref. 5.1]. We do this using `i` (the index of the edge that must be fragmented) and the indices of the new edges inside `ev` offset by the `finalcells_num` counter, which is updated at every step adding the numbers of fragments created per edge; this counter will also be used later to build the complete 1-skeleton edge matrix with ease.

⟨ Fragment edge 18 ⟩ ≡

```
V, ev = frag_edge(V, EV, i)

newedges_nums = map(x->x+finalcells_num, collect(1:size(ev, 1)))

edge_map[i] = newedges_nums

finalcells_num += size(ev, 1)
push!(EVs, ev)
```
◇

Macro referenced in 11.

We declare `EVs` and `finalcells_num` as local variables of `planar_arrangement`.

⟨ planar_arrangement local variables 19 ⟩ ≡

```
EVs = Array{Cells, 1}()
finalcells_num = 0
```
◇

Macro defined by 12, 19.
Macro referenced in 11.

So now we have a `V` that contains the original points with the points computed with the fragmentation and `EVs`, a list of edges matrices. We must now put the entries of this list together to form an unique `EV` matrix.

⟨ Put fragmentation results together 20 ⟩ ≡

```
EV = spzeros(Int8, finalcells_num, size(V,1))
newcell_index = 1
for ev in EVs
    s = size(ev)
    EV[newcell_index:newcell_index+s[1]-1, 1:s[2]] = ev
    newcell_index += s[1]
end
```
◇

Macro referenced in 11.

5.2.4 Tests

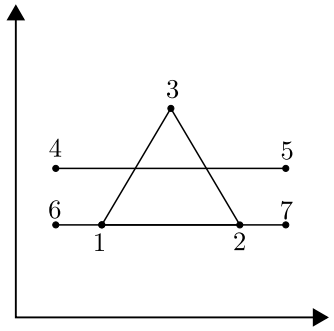


Figure 5.1: The bunch of edges used for the tests.

⟨ planar_arrangement support functions tests 21 ⟩ ≡

```
@testset "Edge fragmentation tests" begin
    V = [2 2; 4 2; 3 3.5; 1 3; 5 3; 1 2; 5 2]
    EV = sparse(Array{Int8, 2}([
        [1 1 0 0 0 0 0] #1->12
        [0 1 1 0 0 0 0] #2->23
        [1 0 1 0 0 0 0] #3->13
        [0 0 0 1 1 0 0] #4->45
        [0 0 0 0 0 1 1] #5->67
    ]))

    @testset "intersect_edges" begin
        inters1 = LARLIB.intersect_edges(V, EV[5, :], EV[1, :])
        inters2 = LARLIB.intersect_edges(V, EV[1, :], EV[4, :])
        inters3 = LARLIB.intersect_edges(V, EV[1, :], EV[2, :])
        @test inters1 == [([2. 2.], 1/4),([4. 2.], 3/4)]
        @test inters2 == []
        @test inters3 == [([4. 2.], 1)]
    end

    @testset "frag_edge" begin
        rV, rEV = LARLIB.frag_edge(V, EV, 5)
```

```
        @test rV == [2.0 2.0; 4.0 2.0; 3.0 3.5; 1.0 3.0;
                     5.0 3.0; 1.0 2.0; 5.0 2.0; 2.0 2.0;
                     4.0 2.0; 4.0 2.0; 2.0 2.0]
        @test full(rEV) == [0 0 0 0 0 1 0 0 0 0 1;
                            0 0 0 0 0 0 0 0 0 1 1;
                            0 0 0 0 0 0 1 0 0 1 0]
    end
end
◇
```

Macro defined by 21, 26, 37, 49.
Macro referenced in 13.

5.3 Coincident vertices merge

To merge vertices in $d = 2$ the procedure is obviously similar to the one used for $d = 3$ so we will reuse some macros already defined [ref. 4.2]

⟨ LAR imports 22 ⟩ ≡

```
using NearestNeighbors
◇
```

Macro defined by 6, 22, 58, 73.
Macro referenced in 1.

The function is marked with "`!`" in its signature because it has collateral effects on the `edge_map` argument; we will for sure modify both the geometry and the topology of the complex, so `edge_map` must be accordingly updated.

⟨ planar_arrangement support functions 23 ⟩ ≡

```
function merge_vertices!(V::Verts, EV::Cells, edge_map, err=1e-4)
    vertsnum = size(V, 1)
    edgenum = size(EV, 1)
    newverts = zeros(Int, vertsnum)
    kdtree = KDTree(V')

    ⟨ Find coincident vertices 7 ⟩
    ⟨ Merge edges 8 ⟩
    ⟨ Update edge_map after vertex merging 24 ⟩

    return nV, nEV
end
◇
```

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

The last step is to update `edge_map`. We update the indices using the data structures built in the ⟨ Merge edges ⟩ macro [ref. 4.2].

⟨ Update edge_map after vertex merging 24 ⟩ ≡

```
for i in 1:length(edge_map)
    row = edge_map[i]
    row = map(x->edges[x], row)
    row = filter(t->t[1]!=t[2], row)
    row = map(x->etuple2idx[x], row)
    edge_map[i] = row
end
```
◇

Macro referenced in 23.

5.3.1 Implementation

We simply call `merge_vertices`.

⟨ Merge coincident vertices 25 ⟩ ≡
```
V, EV = merge_vertices!(V, EV, edge_map)
```
◇

Macro referenced in 11.

5.3.2 Tests

Let's merge the vertices of a square built by numerous very similar edges.

⟨ planar_arrangement support functions tests 26 ⟩ ≡
```
@testset "merge_vertices test set" begin
    n0 = 1e-12
    n1l = 1-1e-12
    n1u = 1+1e-12
    V = [ n0  n0; -n0  n0;  n0 -n0; -n0 -n0;
          n0 n1u; -n0 n1u;  n0 n1l; -n0 n1l;
         n1u n1u; n1l n1u; n1u n1l; n1l n1l;
         n1u  n0; n1l  n0; n1u -n0; n1l -n0]
    EV = Int8[1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0;
              0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0;
              0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0;
              0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0;
              0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0;
              0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0;
              0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0;
              0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0;
              0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0;
              0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0;
              0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0;
              0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1;
              1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0;
              0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0;
              0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0;
              0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1]
    EV = sparse(EV)
    V, EV = LARLIB.merge_vertices!(V, EV, [])
```

```
    @test V == [n0 n0; n0 n1u; n1u n1u; n1u n0]
    @test full(EV) == [1 1 0 0;
                       0 1 1 0;
                       0 0 1 1;
                       1 0 0 1]
end
```
◇

Macro defined by 21, 26, 37, 49.
Macro referenced in 13.

5.4 Delete edges outside σ area

If a face σ is passed as input of the planar arrangement, we need to delete the edges outside the area of σ. First, we use `edge_map` to get the fragments of the edges of the original σ; then for every edge which is not a fragment of σ's edges, we check if its centroid is inside σ using the `point_in_face` utility [ref. 8.11]. Finally, once we have marked the edges to delete, we delete them [ref. 8.5] and update the `edge_map` (refer to next macro for this).

⟨ Delete edges outside sigma area 27 ⟩ ≡

```
if sigma.n > 0
    todel = []

    new_edges = []
    map(i->new_edges=union(new_edges, edge_map[i]), sigma.nzind)
    ev = EV[new_edges, :]

    for e in 1:EV.m
        if !(e in new_edges)

            vidxs = EV[e, :].nzind
            v1, v2 = map(i->V[vidxs[i], :], [1,2])
            centroid = .5*(v1 + v2)

            if !point_in_face(centroid, V, ev)
                push!(todel, e)
            end
        end
    end

    ⟨ Update edge_map 28 ⟩

    V, EV = delete_edges(todel, V, EV)
end
```
◇

Macro referenced in 11.

For every deleted edge the `edge_map` must be updated. So we delete the index of the edge from the mapping and subtract one to the indices greater than the index of the deleted edge.

⟨ Update edge_map 28 ⟩ ≡
```
for i in reverse(todel)
    for row in edge_map

        filter!(x->x!=i, row)

        for j in 1:length(row)
            if row[j] > i
                row[j] -= 1
            end
        end
    end
end
```
◇
Macro referenced in 27, 36.

5.5 Maximal biconnected components

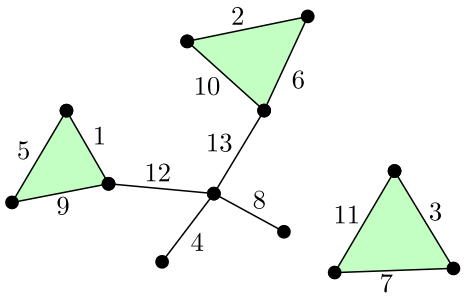


Figure 5.2: An example graph where the maximal biconnected components are highlighted in green and the edges are numbered. We have here three components formed by the sets of edges {1,5,9}, {2,6,10} and {3,11,7}

5.5.1 Support function

To individuate the maximal biconnected components of the fragmented and merged 1-skeleton we use the 1973 Hopcroft-Tarjan algorithm for biconnected components [8].

⟨ planar_arrangement support functions 29 ⟩ ≡
```
function biconnected_components(EV::Cells)
    ⟨ biconnected_components local variables 30 ⟩
    ⟨ DFS utilities 31 ⟩
```

```
    ⟨ Depth first visit 32 ⟩
    bicon_comps
end
◇
```

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

We will need a point stack (`ps`), an edge stack (`es`), a list of traversed edges (`todel`), a list of visited points (`visited`), a list of biconnected components (`bicon_comps`) and a index to avoid duplicate numbering of vertices (`hivtx`). `ps` is made of triples composed by the index of the vertex in `V`, the index assigned by the algorithm and the component identifier also assigned by the algorithm. `es` instead contains couples with the index of the edge inside `EV` and the assigned index of the tail node. The indexes in `todel` and `bicon_comps` are relative to `EV` while the ones of `visited` are relative to `V`

⟨ biconnected_components local variables 30 ⟩ ≡

```
ps = Array{Tuple{Int, Int, Int}, 1}()
es = Array{Tuple{Int, Int}, 1}()
todel = Array{Int, 1}()
visited = Array{Int, 1}()
bicon_comps = Array{Array{Int, 1}, 1}()
hivtx = 1
◇
```

Macro referenced in 29.

Here are implemented some functions helpful throughout the algorithm. `an_edge` returns the index relative to `EV` of the first edge out of `point` if exists or `false` otherwise. `get_head`, given an `edge` and a point (the `tail`), returns the index relative to `V` of the head (the point that is not `tail`) of the `edge`. `v_to_vi`, given the index relative to `V` of a vertex (`v`), returns its index using the algorithm numbering. This index can also not exists; in this case `false` is returned.

⟨ DFS utilities 31 ⟩ ≡

```
function an_edge(point)
    edges = setdiff(EV[:, point].nzind, todel)
    if length(edges) == 0
        edges = [false]
    end
    edges[1]
end

function get_head(edge, tail)
    setdiff(EV[edge, :].nzind, [tail])[1]
end

function v_to_vi(v)
    i = findfirst(t->t[1]==v, ps)
    if i == 0
        return false
```

```
        else
            return ps[i][2]
        end
    end
    ◇
```

Macro referenced in 29.

The DFS visit is mostly akin to the one proposed in the Hopcroft-Tarjan original algorithm. The starting point is the first one in `V`.

⟨ Depth first visit 32 ⟩ ≡

```
    push!(ps, (1,1,1))
    push!(visited, 1)
    exit = false
    while !exit
        edge = an_edge(ps[end][1])
        if edge != false
            tail = ps[end][2]
            head = get_head(edge, ps[end][1])
            hi = v_to_vi(head)
            if hi == false
                hivtx += 1
                push!(ps, (head, hivtx, ps[end][2]))
                push!(visited, head)
            else
                if hi < ps[end][3]
                    ps[end] = (ps[end][1], ps[end][2], hi)
                end
            end
            push!(es, (edge, tail))
            push!(todel, edge)
        else
            if length(ps) == 1
                ⟨ Handle disconnected graph 34 ⟩
            else
                if ps[end][3] == ps[end-1][2]
                    ⟨ Form biconnected component 33 ⟩
                else
                    if ps[end-1][3] > ps[end][3]
                        ps[end-1] = (ps[end-1][1], ps[end-1][2], ps[end][3])
                    end
                end
                pop!(ps)
            end
        end
    end
    ◇
```

Macro referenced in 29.

To form a biconnected component we pop edges out from the stack of edges (`es`) until we find the one of which the index of its tail is equal to the component

identifier (called `LOWPOINT` in the original algorithm) of the top point of the point stack (`ps`). We effectively put inside the `bicon_comps` only the components made of more than one edge because we are interested in building a 1-skeleton of valid 2-cells.

⟨ Form biconnected component 33 ⟩ ≡

```
edges = Array{Int, 1}()
while true
    edge, tail = pop!(es)
    push!(edges, edge)
    if tail == ps[end][3]
        if length(edges) > 1
            push!(bicon_comps, edges)
        end
        break
    end
end
```

◇

Macro referenced in 32.

When there are no more points to visit in the current connected component we search for a point in `V` which has not been visited yet (so a point not listed in the `visited` array) and we put it on the top of a new point stack and then let the algorithm iterate again. If there are no more new connected components to visit we break the algorithm iteration and exit.

⟨ Handle disconnected graph 34 ⟩ ≡

```
found = false
pop!(ps)
for i in 1:size(EV,2)
    if !(i in visited)
        hivtx = 1
        push!(ps, (i, hivtx, 1))
        push!(visited, i)
        found = true
        break
    end
end
if !found
    exit = true
end
```

◇

Macro referenced in 32.

5.5.2 Implementation

Like for the vertices merge we simply call the freshly implemented `biconnected_components` function [ref. 5.5.1]. If no biconnected components are found, the procedure will stop and return nothing.

⟨ Find maximal biconnected components 35 ⟩ ≡

```
bicon_comps = biconnected_components(EV)

if isempty(bicon_comps)
    println("No biconnected components found.")
    return (nothing, nothing, nothing)
end
```

◇

Macro referenced in 11.

We also need to delete edges that are not part of a maximal biconnected component and then to delete the isolated vertices from both `V` and `EV`. We also update the `edge_map` to adapt it to the deletions made (We use the macro defined in 5.4).

⟨ Filter biconnected components 36 ⟩ ≡

```
edges = sort(union(bicon_comps...))
todel = sort(setdiff(collect(1:size(EV,1)), edges))

⟨ Update edge_map 28 ⟩

V, EV = delete_edges(todel, V, EV)
```

◇

Macro referenced in 11.

5.5.3 Tests

The graph built here is the one of figure 5.2.

⟨ planar_arrangement support functions tests 37 ⟩ ≡

```
@testset "biconnected_components test set" begin
    EV = Int8[0 0 0 1 0 0 0 0 0 0 1 0; #1
              0 0 1 0 0 1 0 0 0 0 0 0; #2
              0 0 0 0 0 0 1 0 0 1 0 0; #3
              1 0 0 0 1 0 0 0 0 0 0 0; #4
              0 0 0 1 0 0 0 1 0 0 0 0; #5
              0 0 1 0 0 0 0 0 1 0 0 0; #6
              0 1 0 0 0 0 0 0 0 1 0 0; #7
              0 0 0 0 1 0 0 0 0 0 0 1; #8
              0 0 0 0 0 0 0 1 0 0 1 0; #9
              0 0 0 0 0 1 0 0 1 0 0 0; #10
              0 1 0 0 0 0 1 0 0 0 0 0; #11
              0 0 0 0 1 0 0 0 0 0 1 0; #12
              0 0 0 0 1 0 0 0 1 0 0 0] #13
    EV = sparse(EV)

    bc = LARLIB.biconnected_components(EV)
    bc = Set(map(Set, bc))
```

```
    @test bc == Set([Set([1,5,9]), Set([2,6,10]), Set([3,7,11])])
end
```

◇

Macro defined by 21, 26, 37, 49.
Macro referenced in 13.

5.6 Faces creation

5.6.1 Implementation

⟨ Create faces 38 ⟩ ≡

```
bicon_comps = biconnected_components(EV)

n = size(bicon_comps, 1)
shells = Array{Cell, 1}(n)
boundaries = Array{Cells, 1}(n)
EVs = Array{Cells, 1}(n)
for p in 1:n
    ev = EV[sort(bicon_comps[p]), :]
    fe = minimal_2cycles(V, ev)
    shell_num = get_external_cycle(V, ev, fe)

    EVs[p] = ev
    tokeep = setdiff(1:fe.m, shell_num)
    boundaries[p] = fe[tokeep, :]
    shells[p] = fe[shell_num, :]
end

⟨ Containment test 40 ⟩
⟨ Transitive reduction 43 ⟩
⟨ Cell merging 45 ⟩

◇
```

Macro referenced in 11.

5.6.2 Individuate the external cell

Once we computed the minimal 2-cycles [ref. 7] we need to individuate the external cycle. To do this we iterate over the vertices of the passed `EV` to find four vertices: the two with biggest x_1 and x_2 coordinates (`maxv_x1` and `maxv_x2`) and the two with the smallest one (`minv_x1` and `minv_x2`). Then we check which face the two vertices have in common.

It can happen that the two vertices have more than one face in common (for example when a biconnected component is made up only by one face); in this case we simply pick the cell with negative area. The area computation routines are located into section 8.3,

⟨ planar_arrangement support functions 39 ⟩ ≡

```
function get_external_cycle(V::Verts, EV::Cells, FE::Cells)
    FV = abs(FE)*EV
    vs = sparsevec(mapslices(sum, abs(EV), 1)).nzind
    minv_x1 = maxv_x1 = minv_x2 = maxv_x2 = pop!(vs)
    for i in vs
        if V[i, 1] > V[maxv_x1, 1]
            maxv_x1 = i
        elseif V[i, 1] < V[minv_x1, 1]
            minv_x1 = i
        end
        if V[i, 2] > V[maxv_x2, 2]
            maxv_x2 = i
        elseif V[i, 2] < V[minv_x2, 2]
            minv_x2 = i
        end
    end
    cells = intersect(
        FV[:, minv_x1].nzind,
        FV[:, maxv_x1].nzind,
        FV[:, minv_x2].nzind,
        FV[:, maxv_x2].nzind
    )
    if length(cells) == 1
        return cells[1]
    else
        for c in cells
            if face_area(V, EV, FE[c, :]) < 0
                return c
            end
        end
    end
end
◇
```

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

5.6.3 Containment test

For each shell we must compute if it is contained in another shell. So, for every couple of shells we must check if one is contained into the other. This check must be performed by shooting a ray from a vertex of the first cell and then count the intersections of it with the edges of the second cell; if the number of the intersections is odd then the first cell is contained in the second one. This computation is rather heavy but can be speeded up by pre-computing an approximate containment graph using a bounding box containment test. Then the graph must be pruned shooting a ray for every arc of it. In this way we reduce considerably the amount of rays we shoot. This will be also visually explained in the tests [ref. 5.6.6].

Before building the containment graph, we compute the bounding boxes of

the shells and we store them into the `shell_bboxes` list (we are going to use this also later). The bounding box logic is implemented in the utilities [ref. 8.2].

⟨ Containment test 40 ⟩ ≡
```
shell_bboxes = []
for i in 1:n
    vs_indexes = (abs(EVs[i]')*abs(shells[i])).nzind
    push!(shell_bboxes, bbox(V[vs_indexes, :]))
end

containment_graph = pre_containment_test(shell_bboxes)
containment_graph = prune_containment_graph(n, V, EVs, shells, containment_graph)
```
◇
Macro referenced in 38.

⟨ planar_arrangement support functions 41 ⟩ ≡
```
function pre_containment_test(bboxes)
    n = length(bboxes)
    containment_graph = spzeros(Int8, n, n)

    for i in 1:n
        for j in 1:n
            if i != j && bbox_contains(bboxes[j], bboxes[i])
                containment_graph[i, j] = 1
            end
        end
    end

    return containment_graph
end
```
◇
Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

To check if a point is really inside a face we use the `point_in_face` utility [ref. 8.11]

⟨ planar_arrangement support functions 42 ⟩ ≡
```
function prune_containment_graph(n, V, EVs, shells, graph)

    for i in 1:n
        an_edge = shells[i].nzind[1]
        origin_index = EVs[i][an_edge, :].nzind[1]
        origin = V[origin_index, :]

        for j in 1:n
            if i != j
                if graph[i, j] == 1
                    shell_edge_indexes = shells[j].nzind
                    ev = EVs[j][shell_edge_indexes, :]
```

```
                        if !point_in_face(origin, V, ev)
                            graph[i, j] = 0
                        end
                    end
                end
            end

        end
        return graph
    end
```
◇

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

5.6.4 Transitive reduction

We have an adjacency matrix and we must perform a transitive reduction. As explained by A. V. Aho, M. R. Garey, and J. D. Ullman [1] we have:

⟨ Transitive reduction 43 ⟩ ≡
```
    transitive_reduction!(containment_graph)
```
◇

Macro referenced in 38.

⟨ planar_arrangement support functions 44 ⟩ ≡
```
    function transitive_reduction!(graph)
        n = size(graph, 1)
        for j in 1:n
            for i in 1:n
                if graph[i, j] > 0
                    for k in 1:n
                        if graph[j, k] > 0
                            graph[i, k] = 0
                        end
                    end
                end
            end
        end
    end
```
◇

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

5.6.5 Cell merging

For every arc of the containment tree we have a father component and a child component and we must find the cycle of the father that contains the child. This happens if the bounding box of the child is fully contained in the box of the cycle[1]. The `sums` array contains the indexes of the rows of the various boundary

[1] Please note that that the `bboxes` is not part of the bounding box utilities [ref. 8.2] but it is defined in the next paragraph

matrices to sum after the containment graph has been traversed. Every element is a triple made of: the father index, the father's container cell index and the child index. Once we individuated the rows to sum, we actually need to perform the sum. This is non trivial because we must build the final boundary matrix. These computations are delegated to the ⟨ Create EV and FE ⟩ macro.

⟨ Cell merging 45 ⟩ ≡

```
EV, FE = cell_merging(n, containment_graph, V, EVs, boundaries, shells, shell_bboxes)
```

◇

Macro referenced in 38.

⟨ planar_arrangement support functions 46 ⟩ ≡

```
function cell_merging(n, containment_graph, V, EVs, boundaries, shells, shell_bboxes)
    ⟨ Cell merging support functions 47 ⟩

    sums = Array{Tuple{Int, Int, Int}}(0);

    for father in 1:n
        if sum(containment_graph[:, father]) > 0
            father_bboxes = bboxes(V, abs(EVs[father]')*abs(boundaries[father]'))
            for child in 1:n
                if containment_graph[child, father] > 0
                    child_bbox = shell_bboxes[child]
                    for b in 1:length(father_bboxes)
                        if bbox_contains(father_bboxes[b], child_bbox)
                            push!(sums, (father, b, child))
                            break
                        end
                    end
                end
            end
        end
    end

    ⟨ Create EV and FE 48 ⟩
    return EV, FE
end
```

◇

Macro defined by 14, 15, 23, 29, 39, 41, 42, 44, 46.
Macro referenced in 11.

The `bboxes` computes the bounding boxes of each cycle described in the `indexes` matrix.

⟨ Cell merging support functions 47 ⟩ ≡

```
function bboxes(V::Verts, indexes::Cells)
    boxes = Array{Tuple{Any, Any}}(indexes.n)
    for i in 1:indexes.n
```

```
            v_inds = indexes[:, i].nzind
            boxes[i] = bbox(V[v_inds, :])
        end
        boxes
    end
```
◇

Macro referenced in 46.

To actually build the complete and correct boundary matrix `FE`, we compute the final dimensions of it, then we initialize it filled with zeros and then we fill it with the correct data in the correct position. While doing this we store into `c_offsets` the column offset of each biconnected component; we will use this information to quickly find the columns to sum from the `sums` array of triples.

⟨ Create EV and FE 48 ⟩ ≡

```
    EV = vcat(EVs...)
    edgenum = size(EV, 1)
    facenum = sum(map(x->size(x,1), boundaries))
    FE = spzeros(Int8, facenum, edgenum)
    shells2 = spzeros(Int8, length(shells), edgenum)
    r_offsets = [1]
    c_offset = 1
    for i in 1:n
        min_row = r_offsets[end]
        max_row = r_offsets[end] + size(boundaries[i], 1) - 1
        min_col = c_offset
        max_col = c_offset + size(boundaries[i], 2) - 1
        FE[min_row:max_row, min_col:max_col] = boundaries[i]
        shells2[i, min_col:max_col] = shells[i]
        push!(r_offsets, max_row + 1)
        c_offset = max_col + 1
    end

    for (f, r, c) in sums
        FE[r_offsets[f]+r-1, :] += shells2[c, :]
    end
```
◇

Macro referenced in 46.

5.6.6 Tests

⟨ planar_arrangement support functions tests 49 ⟩ ≡

```
    @testset "Face creation" begin
        ⟨ Face creation tests 50, ... ⟩
    end
```
◇

Macro defined by 21, 26, 37, 49.
Macro referenced in 13.

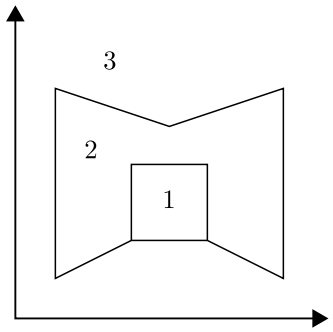


Figure 5.3: This biconnected component has three faces. The external one is the number 3. This is a particularly difficult case because the most "external" vertices of face 2 are in common with the external cell.

External cell individuation

⟨ Face creation tests 50 ⟩ ≡

```
@testset "External cell individuation" begin
    V = [ .5 .5;  1.5   1;  1.5  2;
         2.5  2;  2.5   1;  3.5 .5;
         3.5  3;    2 2.5;   .5  3]

    EV = Int8[-1  1  0  0  0  0  0  0  0;
               0 -1  1  0  0  0  0  0  0;
               0  0 -1  1  0  0  0  0  0;
               0  0  0 -1  1  0  0  0  0;
               0  0  0  0 -1  1  0  0  0;
               0  0  0  0  0 -1  1  0  0;
               0  0  0  0  0  0 -1  1  0;
               0  0  0  0  0  0  0 -1  1;
              -1  0  0  0  0  0  0  0  1;
               0 -1  0  0  1  0  0  0  0]
    EV = sparse(EV)

    FE = Int8[ 0 -1 -1 -1  0  0  0  0  0  1;
               1  1  1  1  1  1  1  1 -1  0;
              -1  0  0  0 -1 -1 -1 -1  1 -1]
    FE = sparse(FE)

    @test LARLIB.get_external_cycle(V, EV, FE) == 3
end
```

◇

Macro defined by 50, 51, 52, 53.
Macro referenced in 49.

Containment test

⟨ Face creation tests 51 ⟩ ≡

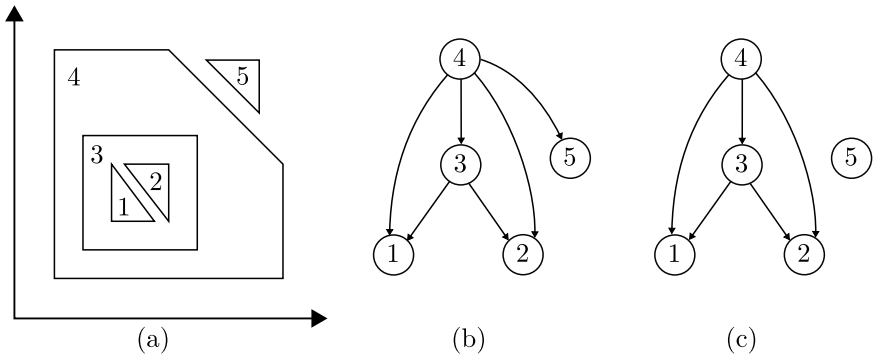


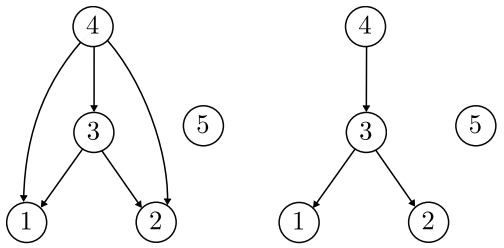
Figure 5.4: (a) is our test case. The numbers identify the connected components. (b) is the containment graph built using only the `pre_containment_test` function. The arc (4,5) is in there because the bounding box of the component no. 5 is completely contained in the bounding box of no. 4. (c) shows the graph after the `prune_containment_graph` function.

```
@testset "Containment test" begin
    V = [  0   0;    4   0;    4   2;   2   4;  0 4;
          .5  .5;  2.5  .5;  2.5 2.5;  .5 2.5;
           1   1;  1.5   1;    1   2;
           2   1;    2   2;  1.5   2;
         3.5 3.5;    3 3.5;  3.5   3]
    EV1 = Int8[ 0  0  0  0  0  0  0  0  0 -1  1  0  0  0  0  0  0  0;
                0  0  0  0  0  0  0  0  0  0 -1  1  0  0  0  0  0  0;
                0  0  0  0  0  0  0  0  0 -1  0  1  0  0  0  0  0  0]
    EV2 = Int8[ 0  0  0  0  0  0  0  0  0  0  0  0 -1  1  0  0  0  0;
                0  0  0  0  0  0  0  0  0  0  0  0  0 -1  1  0  0  0;
                0  0  0  0  0  0  0  0  0  0  0  0 -1  0  1  0  0  0]
    EV3 = Int8[ 0  0  0  0  0 -1  1  0  0  0  0  0  0  0  0  0  0  0;
                0  0  0  0  0  0 -1  1  0  0  0  0  0  0  0  0  0  0;
                0  0  0  0  0  0  0 -1  1  0  0  0  0  0  0  0  0  0;
                0  0  0  0  0 -1  0  0  1  0  0  0  0  0  0  0  0  0]
    EV4 = Int8[-1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0;
                0 -1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0;
                0  0 -1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0;
                0  0  0 -1  1  0  0  0  0  0  0  0  0  0  0  0  0  0;
               -1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
    EV5 = Int8[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  1  0;
                0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  1;
                0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  1]
    EVs = map(sparse, [EV1, EV2, EV3, EV4, EV5])

    shell1 = Int8[-1 -1  1];
    shell2 = Int8[-1 -1  1];
    shell3 = Int8[-1 -1 -1  1];
    shell4 = Int8[-1 -1 -1 -1  1];
```

```
    shell5 = Int8[-1 -1  1];
    shells = map(sparsevec, [shell1, shell2, shell3, shell4, shell5])

    shell_bboxes = []
    n = 5
    for i in 1:n
        vs_indexes = (abs(EVs[i]')*abs(shells[i])).nzind
        push!(shell_bboxes, LARLIB.bbox(V[vs_indexes, :]))
    end

    graph = LARLIB.pre_containment_test(shell_bboxes)
    @test graph == [0 0 1 1 0; 0 0 1 1 0; 0 0 0 1 0; 0 0 0 0 0; 0 0 0 1 0]

    graph = LARLIB.prune_containment_graph(n, V, EVs, shells, graph)
    @test graph == [0 0 1 1 0; 0 0 1 1 0; 0 0 0 1 0; 0 0 0 0 0; 0 0 0 0 0]
end
```
◇

Macro defined by 50, 51, 52, 53.
Macro referenced in 49.

Transitive reduction



(a) (b)

Figure 5.5: Before (a) and after (b) transitive reduction performed on the graph of the previous test set.

⟨ Face creation tests 52 ⟩ ≡

```
@testset "Transitive reduction" begin
    graph = [0 0 1 1 0; 0 0 1 1 0; 0 0 0 1 0; 0 0 0 0 0; 0 0 0 0 0]
    LARLIB.transitive_reduction!(graph)
    @test graph == [0 0 1 0 0; 0 0 1 0 0; 0 0 0 1 0; 0 0 0 0 0; 0 0 0 0 0]
end
```
◇

Macro defined by 50, 51, 52, 53.
Macro referenced in 49.

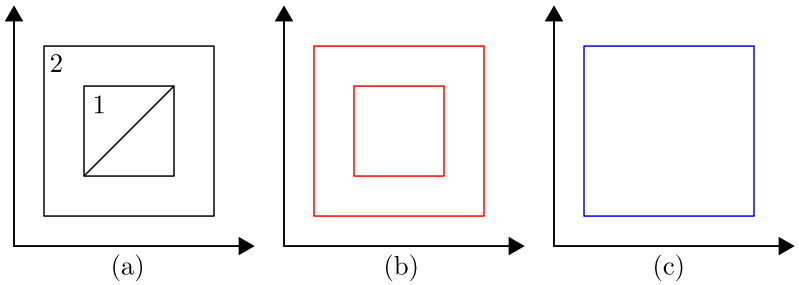


Figure 5.6: Here we have two biconnected components, one inside the other (a). If we don't perform cell merging, the boundary of the arranged set will be the red one (b), which is incorrect. The correct boundary is the blue one (c).

Cell merging

⟨ Face creation tests 53 ⟩ ≡

```
@testset "Cell merging" begin
    graph = [0 1; 0 0]
    V = [.25 .25; .75 .25; .75 .75; .25 .75;
           0   0;   1   0;   1   1;   0   1]
    EV1 = Int8[-1  1  0  0  0  0  0  0;
                0 -1  1  0  0  0  0  0;
                0  0 -1  1  0  0  0  0;
               -1  0  0  1  0  0  0  0;
               -1  0  1  0  0  0  0  0]
    EV2 = Int8[ 0  0  0  0 -1  1  0  0;
                0  0  0  0  0 -1  1  0;
                0  0  0  0  0  0 -1  1;
                0  0  0  0 -1  0  0  1]
    EVs = map(sparse, [EV1, EV2])

    shell1 = Int8[-1 -1 -1  1  0]
    shell2 = Int8[-1 -1 -1  1]
    shells = map(sparsevec, [shell1, shell2])

    boundary1 = Int8[ 1  1  0  0 -1;
                      0  0  1 -1  1]
    boundary2 = Int8[ 1  1  1 -1]
    boundaries = map(sparse, [boundary1, boundary2])

    shell_bboxes = []
    n = 2
    for i in 1:n
        vs_indexes = (abs(EVs[i]')*abs(shells[i])).nzind
        push!(shell_bboxes, LARLIB.bbox(V[vs_indexes, :]))
    end

    EV, FE = LARLIB.cell_merging(2, graph, V, EVs, boundaries, shells, shell_bboxes)
```

```
    selector = sparse(ones(Int8, 1, 3))

    @test selector*FE == [0  0  0  0  0  1  1  1 -1]
end
```

◇

Macro defined by 50, 51, 52, 53.
Macro referenced in 49.

Chapter 6

Dimension travel

6.1 Overview

This chapter is dedicated to the utilities designed to travel from $\mathbb{E}^3$ to $\mathbb{E}^2$

"lib/jl/dimension_travel.jl" 54 ≡
```
⟨ Dimension travel functions 56, ... ⟩
```
◇

6.1.1 Tests

Some unit tests has been written through development and they are collected here.

"test/jl/dimension_travel.jl" 55 ≡
```
using Base.Test
using LARLIB

⟨ Tests 57 ⟩
```
◇

6.2 Submanifold mapping

This function, given three points (in $\mathbb{E}^3$), returns a 4×4 transformation matrix that "flattens" the plane defined by the three points onto the $x_3 = 0$ plane.

⟨ Dimension travel functions 56 ⟩ ≡
```
function submanifold_mapping(vs)
    u1 = vs[2,:] - vs[1,:]
    u2 = vs[3,:] - vs[1,:]
    u3 = cross(u1, u2)
    T = eye(4)
    T[4, 1:3] = - vs[1,:]
    M = eye(4)
```

```
        M[1:3, 1:3] = [u1 u2 u3]
        return T*M
    end
```
◇

Macro defined by 56, 59, 62.
Macro referenced in 54.

6.2.1 Tests

⟨ Tests 57 ⟩ ≡
```
    V = rand(3, 3)
    m = LARLIB.submanifold_mapping(V)
    err = 1e-10
    @testset "submanifold_mapping test" begin
        @test any(map((x,y)->-err<x-y<err, m*inv(m), eye(4)))
        @test any(x->-err<x<err, ([V [1; 1; 1]]*m)[:, 3])
    end
```
◇

Macro referenced in 55.

6.3 Spatial index computation

The aim of this function is to compute a *spatial index* that maps each face to a set of faces which it may collide with. This is achieved by profuse use of bounding boxes and interval trees. We use the interval trees implementation of the `IntervalTrees.jl` package [9].

⟨ LAR imports 58 ⟩ ≡
```
    using IntervalTrees
```
◇

Macro defined by 6, 22, 58, 73.
Macro referenced in 1.

⟨ Dimension travel functions 59 ⟩ ≡
```
    function spatial_index(V::Verts, EV::Cells, FE::Cells)
        d = 3
        faces_num = size(FE, 1)
        ⟨ Build the d-IntervalTrees 60 ⟩
        ⟨ Create the mapping 61 ⟩
        mapping
    end
```
◇

Macro defined by 56, 59, 62.
Macro referenced in 54.

The basic idea is to "unfold" every d-dimensional bounding box into d one-dimensional boxes (which are intervals). To do so, one interval tree per dimension must be created. We build the d-trees by firstly building the intervals for each box and then the trees. In this way we keep in memory the `boxes1D` array (which contains the intervals) for later use. Bounding box calculation is performed by the `bbox` utility [ref. 8.2].

⟨ Build the d-IntervalTrees 60 ⟩ ≡

```
IntervalsType = IntervalValue{Float64, Int64}
boxes1D = Array{IntervalsType, 2}(0, d)
for fi in 1:faces_num
    vidxs = (abs(FE[fi:fi,:])*abs(EV))[1,:].nzind
    intervals = map((l,u)->IntervalsType(l,u,fi), bbox(V[vidxs, :])...)
    boxes1D = vcat(boxes1D, intervals)
end
trees = mapslices(IntervalTree{Float64, IntervalsType}, sort(boxes1D, 1), 1)
```

◇

Macro referenced in 59.

The *spatial index* is returned as an array of `Int64` arrays. The `intersect_intervals` function returns every cell of which its bounding box collides with the d-intervals passed as argument. This function then is called for the d-intervals (stored in the `boxes1D` array) of every cell. Obviously every cell collides with itself, so a set difference is performed for every cell to exclude itself from the mapping.

⟨ Create the mapping 61 ⟩ ≡

```
function intersect_intervals(intervals)
    cells = Array{Int64,1}[]
    for axis in 1:d
        vs = map(i->i.value, intersect(trees[axis], intervals[axis]))
        push!(cells, vs)
    end
    mapreduce(x->x, intersect, cells)
end

mapping = Array{Int64,1}[]
for fi in 1:faces_num
    cell_indexes = setdiff(intersect_intervals(boxes1D[fi, :]), [fi])
    push!(mapping, cell_indexes)
end
```

◇

Macro referenced in 59.

6.4 Face intersection with $x_3 = 0$ plane

The intersection of a polygonal face with the $x_3 = 0$ plane computes zero, one or more edges. To perform the intersection we find the intersection point of every edge with the $x_3 = 0$ plane and then we connect the points. It is safe to completely ignore edges parallel to the $x_3 = 0$ plane. This is an another procedure where floating-point numbers comparison is involved and the fixed error rounding is adopted [ref. 3.1.1].

⟨ Dimension travel functions 62 ⟩ ≡

```
function face_int(V::Verts, EV::Cells, face::Cell)

    vs = buildFV(EV, face)
    retV = Verts(0, 3)

    visited_verts = []
    for i in 1:length(vs)
        o = V[vs[i],:]
        j = i < length(vs) ? i+1 : 1
        d = V[vs[j],:] - o

        err = 10e-8
        if !(-err < d[3] < err)

            alpha = -o[3] / d[3]

            if -err <= alpha <= 1+err
                p = o + alpha*d

                if -err < alpha < err || 1-err < alpha < 1+err
                    if !(vin(p, visited_verts))
                        push!(visited_verts, p)
                        retV = [retV; reshape(p, 1, 3)]
                    end
                else
                    retV = [retV; reshape(p, 1, 3)]
                end
            end
        end

    end

    vnum = size(retV, 1)


    if vnum == 1
        vnum = 0
        retV = Verts(0, 3)
    end
    enum = Int(vnum / 2)
    retEV = spzeros(Int8, enum, vnum)

    for i in 1:enum
        retEV[i, 2*i-1:2*i] = [-1, 1]
    end

    retV, retEV
end
```

◇

Macro defined by 56, 59, 62.

Macro referenced in 54.

Chapter 7

Minimal cycles computation

7.1 Main function

Computing the minimal cycles means to compute the d-boundary matrix from the $(d-1)$-boundary. The method has been profusely illustrated by A. Paoluzzi et al. in *Arrangements of cellular complexes* [12]. The method is dimension-independent, so works for both $d = 2$ and $d = 3$; the only difference between the two cases lays in the `angles_fn` function [ref. 7.2]. To support this multidimensional behavior, the algorithm has been implemented as an high-order function[1]:

```
"lib/jl/minimal_cycles.jl" 63 ≡
    ⟨ Minimal cycles implementations 70, ... ⟩

    function minimal_cycles(angles_fn::Function, verbose=false)

        function _minimal_cycles(V::Verts, ld_bounds::Cells)
            ⟨ Function body 64 ⟩
        end

        return _minimal_cycles
    end
    ◇
```

In the internal function we store an array of integers called `count_marks` that increments every time a cells is visited. We do that because to build a complete d-boundary, we must visit every $(d-1)$-cell exactly twice; Said so, it appears clear that the algorithm must iterate until a $(d-1)$-cell marked with 0 or 1 can be found. Near to `count_marks` is stored another array called `dir_marks` that memorizes the direction in which each $(d-1)$-cell has been visited the last time

[1] **Notes on variables names:** `ld` stands for *lower dimension* $(d-1)$ and `lld` for *lower lower dimension* $(d-2)$. So, `ld_cellsnum` is the short form of *lower dimension cell number*. For example, if $d = 2$, `ld_cellsnum` stands for the number of $1-$cells, aka the edges.

(this is useful to determine the direction in which the cell must be visited next). We also print a progression counter for user feedback if the `verbose` flag has been set.

⟨ Function body 64 ⟩ ≡
```
lld_cellsnum, ld_cellsnum = size(ld_bounds)
count_marks = zeros(Int8, ld_cellsnum)
dir_marks = zeros(Int8, ld_cellsnum)
d_bounds = spzeros(Int8, ld_cellsnum, 0)

⟨ minimal_cycles local variables 67 ⟩
⟨ minimal_cycles utilities 65, ... ⟩

while (sigma = get_seed_cell()) > 0

    if verbose
        println(Int(floor(50 * sum(count_marks) / ld_cellsnum)), "%")
    end

    ⟨ Compute a cycle 66 ⟩
end

return d_bounds
```
◇

Macro referenced in 63.

The `get_seed_cell` function returns the first $d-1$ cell marked with zero. If there are no cells marked with zero, the first cell marked with one will be returned. If every cell is marked with 2 then -1 will be returned.

⟨ minimal_cycles utilities 65 ⟩ ≡
```
function get_seed_cell()
    s = -1
    for i in 1:ld_cellsnum
        if count_marks[i] == 0
            return i
        elseif count_marks[i] == 1 && s < 0
            s = i
        end
    end
    return s
end
```
◇

Macro defined by 65, 68, 69.
Macro referenced in 64.

The bigger part of the algorithm is the computation of a single cycle. It is mostly equivalent to the ALGORITHM 1 by A. Paoluzzi et al. [12]

⟨ Compute a cycle 66 ⟩ ≡

```
c_ld = spzeros(Int8, ld_cellsnum)
if count_marks[sigma] == 0
    c_ld[sigma] = 1
else
    c_ld[sigma] = -dir_marks[sigma]
end
c_lld = ld_bounds*c_ld
while c_lld.nzind != []
    corolla = spzeros(Int8, ld_cellsnum)
    for tau in c_lld.nzind
        b_ld = ld_bounds[tau, :]
        pivot = intersect(c_ld.nzind, b_ld.nzind)[1]
        adj = nextprev(tau, pivot, sign(-c_lld[tau]))
        corolla[adj] = c_ld[pivot]
        if b_ld[adj] == b_ld[pivot]
            corolla[adj] *= -1
        end
    end
    c_ld += corolla
    c_lld = ld_bounds*c_ld
end
map(s->count_marks[s] += 1, c_ld.nzind)
map(s->dir_marks[s] = c_ld[s], c_ld.nzind)
d_bounds = [d_bounds c_ld]
◇
```

Macro referenced in 64.

This algorithm revolves around the *next* and *prev* functions. To speed up their computation, before the cycles iteration starts, we calculate and store for each $(d-2)$-cell the angles that its incident $(d-1)$-cells form with it.

⟨ minimal_cycles local variables 67 ⟩ ≡

```
angles = Array{Array{Int64, 1}, 1}(lld_cellsnum)
◇
```

Macro referenced in 64.

Here we use the parameter `angles_fn::Function`. As explained earlier, this function is the only difference between the $d = 3$ and $d = 2$ version of `minimal_cycles`.

⟨ minimal_cycles utilities 68 ⟩ ≡

```
for lld in 1:lld_cellsnum
    as = []
    for ld in ld_bounds[lld, :].nzind
        push!(as, (ld, angles_fn(lld, ld)))
    end
    sort!(as, lt=(a,b)->a[2]<b[2])
    as = map(a->a[1], as)
    angles[lld] = as
end
◇
```

Macro defined by 65, 68, 69.
Macro referenced in 64.

Once computed the `angles`, the `nextprev` function is easy to implement. The `norp` parameter is a short form for *next or prev*. It determines if the function should choose the first available $(d-1)$-cell rotating clockwise or counterclockwise around the $(d-2)$-cell.

⟨ minimal_cycles utilities 69 ⟩ ≡

```
function nextprev(lld::Int64, ld::Int64, norp)
    as = angles[lld]
    ne = findfirst(as, ld)
    while true
        ne += norp
        if ne > length(as)
            ne = 1
        elseif ne < 1
            ne = length(as)
        end

        if count_marks[as[ne]] < 2
            break
        end
    end
    as[ne]
end
```

◇

Macro defined by 65, 68, 69.
Macro referenced in 64.

7.2 Dimensional wise implementations

7.2.1 $d = 2$

When in $d = 2$, $(d-2)$-cells are vertices and $(d-1)$-cells are edges. The `edge_angle` function uses the Julia's `atan2` built-in function to calculate the angle of the edge from the vertex point of view.

⟨ Minimal cycles implementations 70 ⟩ ≡

```
function minimal_2cycles(V::Verts, EV::Cells)

    function edge_angle(v::Int, e::Int)
        edge = EV[e, :]
        v2 = setdiff(edge.nzind, [v])[1]
        x, y = V[v2, :] - V[v, :]
        return atan2(y, x)
    end

    for i in 1:EV.m
```

```
            j = min(EV[i,:].nzind...)
            EV[i, j] = -1
        end
        VE = EV'

        EF = minimal_cycles(edge_angle)(V, VE)

        return EF'
    end
```
◇

Macro defined by 70, 71.
Macro referenced in 63.

7.2.2 $d = 3$

Here we have edges for $(d-2)$-cells and faces for $(d-1)$-cells.

⟨ Minimal cycles implementations 71 ⟩ ≡
```
    function minimal_3cycles(V::Verts, EV::Cells, FE::Cells)

        ⟨ Face angle function 72 ⟩

        EF = FE'

        FC = minimal_cycles(face_angle, true)(V, EF)

        return -FC'
    end
```
◇

Macro defined by 70, 71.
Macro referenced in 63.

This time we need to sort faces around an hinge edge. To compute the angle of a face, we transform it in a way that the hinge lays on the x_1 positive axis[2]. In this way, we can compute the angle of a face by using a classic `atan2` call.

Due to the fact that faces can be non-convex, we triangulate them to be sure to compute their angle correctly; in the case of a non-convex face, it can happen that is picked erroneously the opposite angle of the right one. The triangulation is performed only when the face of index `f` is visited for the first time.

⟨ Face angle function 72 ⟩ ≡
```
    triangulated_faces = Array{Any, 1}(FE.m)

    function face_angle(e::Int, f::Int)
        if !isassigned(triangulated_faces, f)
            ⟨ Triangulate face 74 ⟩
        end
```

[2]The method to compute an univocal reference frame from a single vector comes from *Physically Based Rendering* by Pharr and Humphreys [13]

```
        edge_vs = EV[e, :].nzind

        t = findfirst(x->edge_vs[1] in x && edge_vs[2] in x, triangulated_faces[f])

        v1 = normalize(V[edge_vs[2], :] - V[edge_vs[1], :])
        if abs(v1[1]) > abs(v1[2])
            invlen = 1. / sqrt(v1[1]*v1[1] + v1[3]*v1[3])
            v2 = [-v1[3]*invlen, 0, v1[1]*invlen]
        else
            invlen = 1. / sqrt(v1[2]*v1[2] + v1[3]*v1[3])
            v2 = [0, -v1[3]*invlen, v1[2]*invlen]
        end
        v3 = cross(v1, v2)

        M = reshape([v1; v2; v3], 3, 3)

        triangle = triangulated_faces[f][t]
        third_v = setdiff(triangle, edge_vs)[1]
        vs = V[[edge_vs..., third_v], :]*M

        v = vs[3, :] - vs[1, :]
        angle = atan2(v[2], v[3])

        return angle
    end
    ◇
```

Macro referenced in 71.

To perform triangulation we use the Julia porting by F. Furiani of Triangle, a well known C library for constrained Delaunay triangulations [7] [15]. Due to the fact that Delaunay triangulation works only in $\mathbb{E}^2$, we need to transform the face to triangulate on the $x_3 = 0$ plane. To compute a reference frame on the face plane, we use the classic method of doing two differences of three non-colinear vertices of the face and then cross multiply the vectors resulting from the differences two to get a third one. To make sure that the three chosen vertices are not colinear, we check if the cross of the two difference vectors has non-zero length and we choose new set of vertices until this condition is satisfied[3].

⟨ LAR imports 73 ⟩ ≡

```
    using TRIANGLE
    ◇
```

Macro defined by 6, 22, 58, 73.
Macro referenced in 1.

⟨ Triangulate face 74 ⟩ ≡

[3] We check the length of the cross product against a fixed error [ref. 3.1.1].

```
vs_idxs = Array{Int64, 1}()
edges_idxs = FE[f, :].nzind
edge_num = length(edges_idxs)
edges = zeros(Int64, edge_num, 2)

for (i, ee) in enumerate(edges_idxs)
    edge = EV[ee, :].nzind
    edges[i, :] = edge
    vs_idxs = union(vs_idxs, edge)
end

vs = V[vs_idxs, :]

v1 = normalize(vs[2, :] - vs[1, :])
v3 = [0 0 0]
err = 1e-8
i = 3
while -err < norm(v3) < err
    v2 = normalize(vs[i, :] - vs[1, :])
    v3 = cross(v1, v2)
    i = i + 1
end

M = reshape([v1; v2; v3], 3, 3)

vs = vs*M

triangulated_faces[f] = TRIANGLE.constrained_triangulation(
    vs, vs_idxs, edges, fill(true, edge_num))
◇
```

Macro referenced in 72.

Chapter 8

Utilities

8.1 Overview

The functionalities shared between all the components of LAR are defined in here.

"lib/jl/utilities.jl" 75 ≡
⟨ Utilities 77, ... ⟩
◇

8.1.1 Tests

As usual every function has some unit tests.

"test/jl/utilities.jl" 76 ≡
```
using Base.Test
using LARLIB

```
⟨ Utilities tests 78, ... ⟩
◇

8.2 Bounding boxes

Bounding boxes are essential in many steps of many algorithms in LAR. Here we present a method for building and performing containment tests on n-dimensional axis aligned bounding boxes.

⟨ Utilities 77 ⟩ ≡
```
function bbox(vertices::Verts)
    minimum = mapslices(x->min(x...), vertices, 1)
    maximum = mapslices(x->max(x...), vertices, 1)
    minimum, maximum
end
```

```
function bbox_contains(container, contained)
    b1_min, b1_max = container
    b2_min, b2_max = contained
    all(map((i,j,k,l)->i<=j<=k<=l, b1_min, b2_min, b2_max, b1_max))
end
```
◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.2.1 Tests

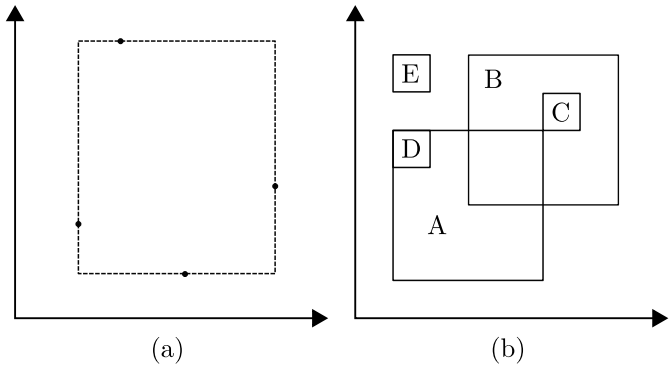


Figure 8.1: (a) is a visualization of the test for bboxes building, (b) for bbox containment.

⟨ Utilities tests 78 ⟩ ≡

```
@testset "Bounding boxes building test" begin
    V = [.56 .28; .84 .57; .35  1.0; .22  .43]
    @test LARLIB.bbox(V) == ([.22 .28], [.84 1.0])
end

@testset "Bounding boxes containment test" begin
    bboxA = ([0. 0.], [1. 1.])
    bboxB = ([.5 .5], [1.5 1.5])
    bboxC = ([1. 1.], [1.25 1.25])
    bboxD = ([0 .75], [.25 1])
    bboxE = ([0 1.25], [.25 1.5])

    @test LARLIB.bbox_contains(bboxA, bboxD)
    @test LARLIB.bbox_contains(bboxB, bboxC)
    @test !LARLIB.bbox_contains(bboxA, bboxB)
    @test !LARLIB.bbox_contains(bboxA, bboxE)
end
```
◇

Macro defined by 78, 80.
Macro referenced in 76.

8.3 Face area calculation

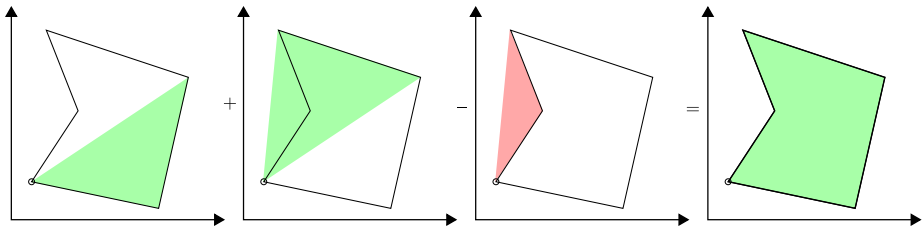


Figure 8.2: A visual representation of the face area calculation algorithm. The area of the face is the sum of the areas of each triangle which can be build using the pivot vertex and the other vertices of the face

To compute the area of a generic (convex or concave) face, we pick a pivot vertex of the face and then we iterate over every edge of the face calculating the area of the triangle made by the pivot vertex and the ordered extremes of the current edge. The area of the full face is the sum of the areas of the single triangles. This works because of the single triangles we compute the signed area with this formula:

$$A = \frac{1}{2}\begin{vmatrix} p_{1x} & p_{1y} & 1 \\ p_{2x} & p_{2y} & 1 \\ p_{3x} & p_{3y} & 1 \end{vmatrix}$$

Where p_1, p_2 and p_3 are the vertices of the triangle (p_1 is the pivot vertex). Please notice that the result of this formula will be negative only if these vertices are arranged in clockwise order.

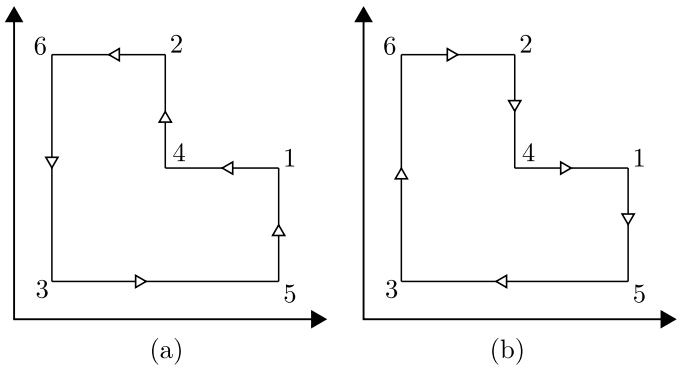
⟨ Utilities 79 ⟩ ≡

```
function face_area(V::Verts, EV::Cells, face::Cell)
    function triangle_area(triangle_points::Verts)
        ret = ones(3,3)
        ret[:, 1:2] = triangle_points
        return .5*det(ret)
    end

    area = 0

    fv = buildFV(EV, face)

    verts_num = length(fv)
    v1 = fv[1]

    for i in 2:(verts_num-1)

        v2 = fv[i]
        v3 = fv[i+1]
```

```
            area += triangle_area(V[[v1, v2, v3], :])
        end

        return area
    end
```
◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.3.1 Tests



The two faces drawn above they must have complimentary area.

⟨ Utilities tests 80 ⟩ ≡

```
    @testset "Face area calculation test" begin
        V = Float64[2 1; 1 2; 0 0; 1 1; 2 0; 0 2]
        EV = spzeros(Int8, 6, 6)
        EV[1, [1, 4]] = [-1, 1]; EV[2, [2, 4]] = [-1, 1]
        EV[3, [2, 6]] = [-1, 1]; EV[4, [3, 6]] = [-1, 1]
        EV[5, [3, 5]] = [-1, 1]; EV[6, [1, 5]] = [-1, 1]
        FE = spzeros(Int8, 2, 6)
        FE[1, :] = [ 1 -1  1 -1  1 -1]
        FE[2, :] = [-1  1 -1  1 -1  1]

        @test LARLIB.face_area(V, EV, FE[1,:]) == -LARLIB.face_area(V, EV, FE[2,:])
    end
```
◇

Macro defined by 78, 80.
Macro referenced in 76.

8.4 Skeletal merge

The first step of the arrangement algorithm is ever the skeletal merge [ref. 2.1].

⟨ Utilities 81 ⟩ ≡

```
function skel_merge(V1::Verts, EV1::Cells, V2::Verts, EV2::Cells)
    V = [V1; V2]
    EV = spzeros(Int8, EV1.m + EV2.m, EV1.n + EV2.n)
    EV[1:EV1.m, 1:EV1.n] = EV1
    EV[EV1.m+1:end, EV1.n+1:end] = EV2
    V, EV
end

function skel_merge(V1::Verts, EV1::Cells, FE1::Cells, V2::Verts, EV2::Cells, FE2::Cells)
    FE = spzeros(Int8, FE1.m + FE2.m, FE1.n + FE2.n)
    FE[1:FE1.m, 1:FE1.n] = FE1
    FE[FE1.m+1:end, FE1.n+1:end] = FE2
    V, EV = skel_merge(V1, EV1, V2, EV2)
    V, EV, FE
end
```

◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.5 Edge deletion

Deleting edges ia a common operation in planar arrangement. When edges are deleted, some vertices can remain unconnected; these must be deleted too.

⟨ Utilities 82 ⟩ ≡

```
function delete_edges(todel, V::Verts, EV::Cells)
    tokeep = setdiff(collect(1:EV.m), todel)
    EV = EV[tokeep, :]

    vertinds = 1:EV.n
    todel = Array{Int64, 1}()
    for i in vertinds
        if length(EV[:, i].nzind) == 0
            push!(todel, i)
        end
    end

    tokeep = setdiff(vertinds, todel)
    EV = EV[:, tokeep]
    V = V[tokeep, :]

    return V, EV
end
```

◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.6 FV building

Sometimes is useful to represent a face like a sequence of vertices.

⟨ Utilities 83 ⟩ ≡

```
function buildFV(EV::Cells, face::Cell)
    startv = -1
    nextv = 0
    edge = 0

    vs = Array{Int64, 1}()

    while startv != nextv
        if startv < 0
            edge = face.nzind[1]
            startv = EV[edge,:].nzind[face[edge] < 0 ? 2 : 1]
            push!(vs, startv)
        else
            edge = setdiff(intersect(face.nzind, EV[:, nextv].nzind), edge)[1]
        end
        nextv = EV[edge,:].nzind[face[edge] < 0 ? 1 : 2]
        push!(vs, nextv)

    end

    return vs[1:end-1]
end
```

◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.7 Boundaries building

⟨ Utilities 84 ⟩ ≡

```
function buildFE(FV, edges)
    faces = []

    for face in FV
        f = []
        for (i,v) in enumerate(face)
            edge = [v, face[i==length(face)?1:i+1]]
            ord_edge = sort(edge)

            edge_idx = findfirst(e->e==ord_edge, edges)

            push!(f, (edge_idx, sign(edge[2]-edge[1])))
        end

        push!(faces, f)
    end

    FE = spzeros(Int8, length(faces), length(edges))
```

```
    for (i,f) in enumerate(faces)
        for e in f
            FE[i, e[1]] = e[2]
        end
    end

    return FE
end

function buildEV(edges)
    maxv = max(map(x->max(x...), edges)...)
    EV = spzeros(Int8, length(edges), maxv)

    for (i,e) in enumerate(edges)
        e = sort(collect(e))
        EV[i, e] = [-1, 1]
    end

    return EV
end


function buildFV(EV::Cells, face)
    startv = face[1]
    nextv = startv

    vs = []
    visited_edges = []

    while true
        curv = nextv
        push!(vs, curv)

        edge = 0
        for edge in EV[:, curv].nzind
            nextv = setdiff(EV[edge, :].nzind, curv)[1]
            if nextv in face && (nextv == startv || !(nextv in vs)) && !(edge in visited_edges)
                break
            end
        end

        push!(visited_edges, edge)

        if nextv == startv
            break
        end
    end

    return vs
end
```

```
function build_bounds(edges, faces)
    EV = buildEV(edges)
    FV = map(x->buildFV(EV,x), faces)
    FE = buildFE(FV, edges)

    return EV, FE
end
```
◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.8 Vertex equality utilities

Vertex comparison must be performed using floating-point fixed error [ref. 3.1.1].

⟨ Utilities 85 ⟩ ≡

```
function vin(vertex, vertices_set)
    for v in vertices_set
        if vequals(vertex, v)
            return true
        end
    end
    return false
end

function vequals(v1, v2)
    err = 10e-8
    return length(v1) == length(v2) && all(map((x1, x2)->-err < x1-x2 < err, v1, v2))
end
```
◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.9 Full triangulation

⟨ Utilities 86 ⟩ ≡

```
function triangulate(V::Verts, EV::Cells, FE::Cells)

    triangulated_faces = Array{Any, 1}(FE.m)

    for f in 1:FE.m
        if f % 10 == 0
            print(".")
        end

        edges_idxs = FE[f, :].nzind
        edge_num = length(edges_idxs)
```

```
            edges = zeros(Int64, edge_num, 2)


            fv = buildFV(EV, FE[f, :])

            vs = V[fv, :]

            v1 = normalize(vs[2, :] - vs[1, :])
            v2 = [0 0 0]
            v3 = [0 0 0]
            err = 1e-8
            i = 3
            while -err < norm(v3) < err
                v2 = normalize(vs[i, :] - vs[1, :])
                v3 = cross(v1, v2)
                i = i + 1
            end
            M = reshape([v1; v2; v3], 3, 3)

            vs = (vs*M)[:, 1:2]

            for i in 1:length(fv)
                edges[i, 1] = fv[i]
                edges[i, 2] = i == length(fv) ? fv[1] : fv[i+1]
            end

            triangulated_faces[f] = TRIANGLE.constrained_triangulation(vs, fv, edges, fill(true, edge_num)

            tV = (V*M)[:, 1:2]

            area = face_area(tV, EV, FE[f, :])
            if area < 0
                for i in 1:length(triangulated_faces[f])
                    triangulated_faces[f][i] = triangulated_faces[f][i][end:-1:1]
                end
            end
        end

        return triangulated_faces
    end
    ◇
```

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.10 OBJ I/O

OBj is a common format for 3D models exchange. Here an exporter of LAR model to OBJ. It returns a string.

⟨ Utilities 87 ⟩ ≡

```
function lar2obj(V::Verts, EV::Cells, FE::Cells, CF::Cells)
    obj = ""
    for v in 1:size(V, 1)
        obj = string(obj, "v ", round(V[v, 1], 6), " ", round(V[v, 2], 6), " ", round(V[v,
    end

    print("Triangulating")
    triangulated_faces = triangulate(V, EV, FE)
    println("DONE")

    for c in 1:CF.m
    obj = string(obj, "\ng cell", c, "\n")
    for f in CF[c, :].nzind
        triangles = triangulated_faces[f]
        for tri in triangles
            t = CF[c, f] > 0 ? tri : tri[end:-1:1]
            obj = string(obj, "f ", t[1], " ", t[2], " ", t[3], "\n")
        end
    end
end

    return obj
end
◇
```

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

And here an importer. It wants a path to the obj file expressed as a string. It returns the classic tuple `V, EV, FE`.

⟨ Utilities 88 ⟩ ≡

```
function obj2lar(path)
    fd = open(path, "r")
    vs = Array{Float64, 2}(0, 3)
    edges = Array{Array{Int, 1}, 1}()
    faces = Array{Array{Int, 1}, 1}()

    while (line = readline(fd)) != ""
        elems = split(line)
        if length(elems) > 0
            if elems[1] == "v"

                x = parse(Float64, elems[2])
                y = parse(Float64, elems[3])
                z = parse(Float64, elems[4])
                vs = [vs; x y z]

            elseif elems[1] == "f"
                v1 = parse(Int, elems[2])
                v2 = parse(Int, elems[3])
```

```
                v3 = parse(Int, elems[4])

                e1 = sort([v1, v2])
                e2 = sort([v2, v3])
                e3 = sort([v1, v3])

                if !(e1 in edges)
                    push!(edges, e1)
                end
                if !(e2 in edges)
                    push!(edges, e2)
                end
                if !(e3 in edges)
                    push!(edges, e3)
                end

                push!(faces, sort([v1, v2, v3]))
            end
        end
    end

    close(fd)
    vs, build_bounds(edges, faces)...
end
```

◇

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

8.11 Point in face area

Point in face inclusion is performed using the algorithm presented by A. Paoluzzi in 1986 [11]. It is based on the ray shooting and it analyzes more than thirty possible ray-edge intersection cases.

⟨ Utilities 89 ⟩ ≡

```
function point_in_face(origin, V::Verts, ev::Cells)

    function pointInPolygonClassification(V,EV)

        function crossingTest(new, old, status, count)
        if status == 0
            status = new
            return status, (count + 0.5)
        else
            if status == old
                return 0, (count + 0.5)
            else
                return 0, (count - 0.5)
            end
```

```
end
end

function setTile(box)
tiles = [[9,1,5],[8,0,4],[10,2,6]]
b1,b2,b3,b4 = box
function tileCode(point)
    x,y = point
    code = 0
    if y>b1 code=code|1 end
    if y<b2 code=code|2 end
    if x>b3 code=code|4 end
    if x<b4 code=code|8 end
    return code
end
return tileCode
end

function pointInPolygonClassification0(pnt)
    x,y = pnt
    xmin,xmax,ymin,ymax = x,x,y,y
    tilecode = setTile([ymax,ymin,xmax,xmin])
    count,status = 0,0

    for k in 1:EV.m
        edge = EV[k,:]
        p1, p2 = V[edge.nzind[1], :], V[edge.nzind[2], :]
        (x1,y1),(x2,y2) = p1,p2
        c1,c2 = tilecode(p1),tilecode(p2)
        c_edge, c_un, c_int = xor(c1, c2), c1|c2, c1&c2

        if (c_edge == 0) & (c_un == 0) return "p_on"
        elseif (c_edge == 12) & (c_un == c_edge) return "p_on"
        elseif c_edge == 3
            if c_int == 0 return "p_on"
            elseif c_int == 4 count += 1 end
        elseif c_edge == 15
            x_int = ((y-y2)*(x1-x2)/(y1-y2))+x2
            if x_int > x count += 1
            elseif x_int == x return "p_on" end
        elseif (c_edge == 13) & ((c1==4) | (c2==4))
                status, count = crossingTest(1,2,status,count)
        elseif (c_edge == 14) & ((c1==4) | (c2==4))
                status, count = crossingTest(2,1,status,count)
        elseif c_edge == 7 count += 1
        elseif c_edge == 11 count = count
        elseif c_edge == 1
            if c_int == 0 return "p_on"
            elseif c_int == 4
                status, count = crossingTest(1,2,status,count)
```

```
                    end
                elseif c_edge == 2
                    if c_int == 0 return "p_on"
                    elseif c_int == 4
                        status, count = crossingTest(2,1,status,count)
                    end
                elseif (c_edge == 4) & (c_un == c_edge) return "p_on"
                elseif (c_edge == 8) & (c_un == c_edge) return "p_on"
                elseif c_edge == 5
                    if (c1==0) | (c2==0) return "p_on"
                    else
                        status, count = crossingTest(1,2,status,count)
                    end
                elseif c_edge == 6
                    if (c1==0) | (c2==0) return "p_on"
                    else
                        status, count = crossingTest(2,1,status,count)
                    end
                elseif (c_edge == 9) & ((c1==0) | (c2==0)) return "p_on"
                elseif (c_edge == 10) & ((c1==0) | (c2==0)) return "p_on"
                end
            end

            if (round(count)%2)==1
                return "p_in"
            else
                return "p_out"
            end
        end
        return pointInPolygonClassification0
    end

    return pointInPolygonClassification(V, ev)(origin) == "p_in"
end
◇
```

Macro defined by 77, 79, 81, 82, 83, 84, 85, 86, 87, 88, 89.
Macro referenced in 75.

Part III

Tests and conclusions

Chapter 9

Tests and examples

"test/jl/runtests.jl" 90 ≡
```
using LARLIB

include("./planar_arrangement.jl")
include("./dimension_travel.jl")
include("./utilities.jl")
```
◇

"examples/jl/general_tests.jl" 91 ≡
```
using LARLIB

⟨ planar_arrangement tests 92 ⟩
⟨ spatial_arrangement tests 93 ⟩
```
◇

9.1 Planar arrangement tests

Here we present some general tests for the `planar_arrangement` function [ref. 5]

⟨ planar_arrangement tests 92 ⟩ ≡
```
function generate_perpendicular_lines(steps::Int, minlen, maxlen)
    V = zeros(0,2)

    function rec(o, d, s)
        if s == 0 return end

        a = (maxlen-minlen)*rand() + minlen
        p = o + a*d
        V = [V; o; p]

        b = (a-minlen)*rand() + minlen
        p = o + b*d
        rec(p, d, s-1)
```

```
            b = (a-minlen)*rand() + minlen
            p = o + b*d
            rec(p, perpendicular(d), s-1)
        end

        function perpendicular(vec)
            v = zeros(size(vec))
            v[1] = vec[2]
            v[2] = vec[1]
            return v
        end

        rec([0 0], [1 0], steps)
        rec([0 0], [0 1], steps)
        vnum = size(V, 1)
        enum = vnum >> 1
        EV = spzeros(Int8, enum, vnum)
        for i in 1:enum
            EV[i, i*2-1:i*2] = 1
        end
        V, EV
    end


    function generate_random_lines(n, points_range, alphas_range)
        origins = points_range[1] + (points_range[2]-points_range[1])*rand(n, 2)
        directions = mapslices(normalize, rand(n, 2) - .5*ones(n, 2), 2)
        alphas = alphas_range[1] + (alphas_range[2]-alphas_range[1])*rand(n)
        new_points = Array{Float64, 2}(n, 2)
        for i in 1:n
            new_points[i, :] = origins[i, :] + alphas[i]*directions[i, :]
        end
        V = [origins; new_points]
        EV = spzeros(Int8, n, n*2)
        for i in 1:n
            EV[i, i] = 1
            EV[i, n+i] = 1
        end
        V, EV
    end
    ◇
```

Macro referenced in 91.

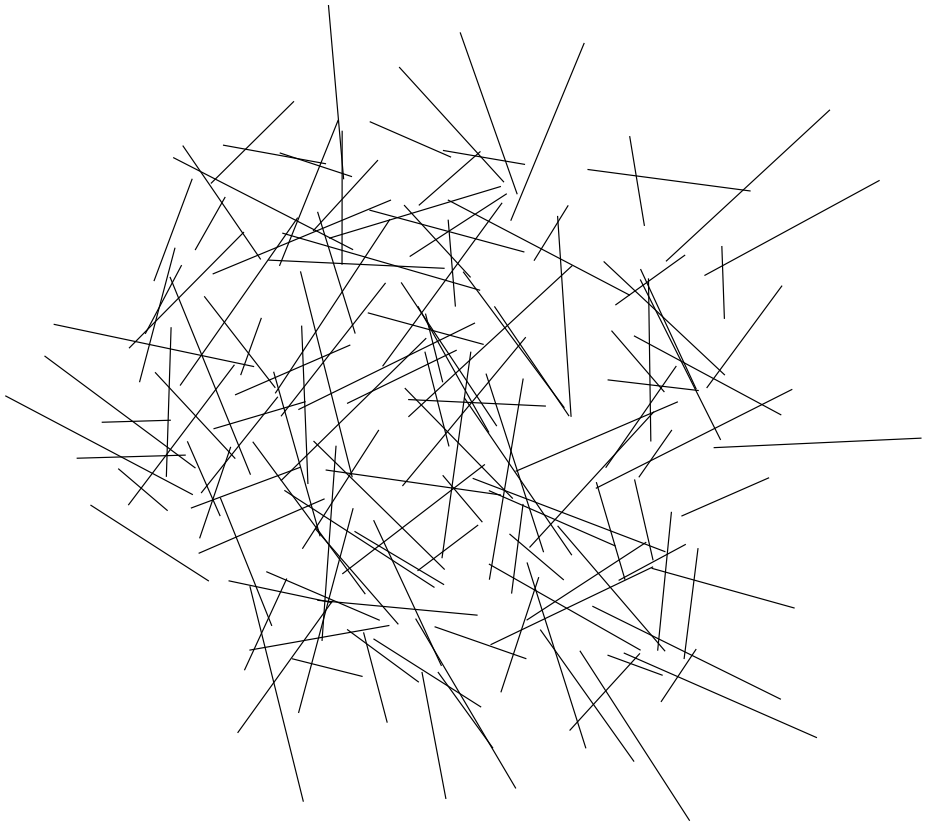


Figure 9.1: Input

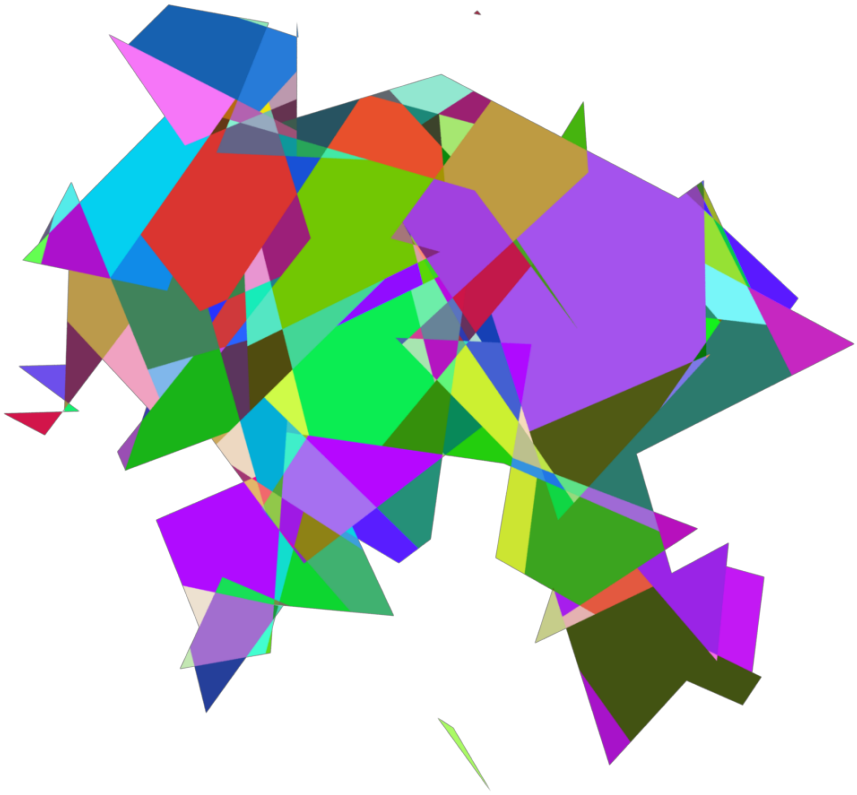


Figure 9.2: Output

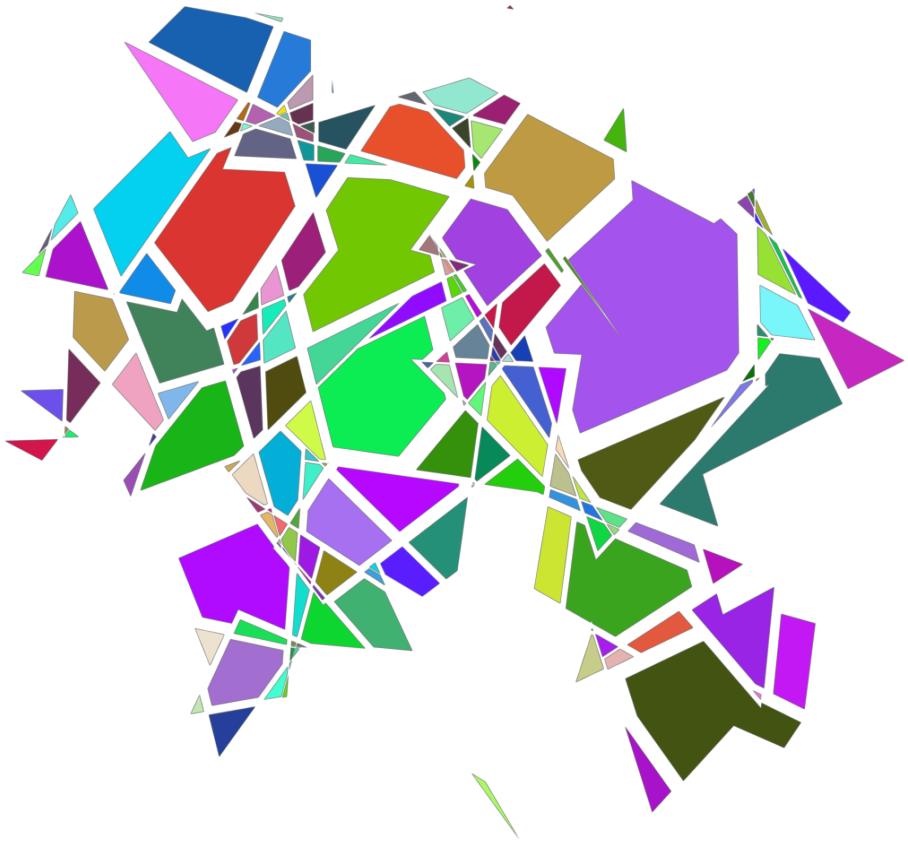


Figure 9.3: Output (Exploded)

9.2 Spatial arrangement tests

We used this test a lot during development. It builds a cube made of $3 \times 3 \times 3$ cubes. The it arranges the cubes, building a sort of Rubik's cube. Then it duplicates it and rotates a copy by $\pi/6$ on the x_1-axis and then on the x_3-axis.

⟨ spatial_arrangement tests 93 ⟩ ≡

```
function rubiks_example(ncubes = 3)
    V = Float64[
        0 0 0; 0 1 0;
        1 1 0; 1 0 0;
        0 0 1; 0 1 1;
        1 1 1; 1 0 1
    ]

    EV = sparse(Int8[
        -1  1  0  0  0  0  0  0;
        0 -1  1  0  0  0  0  0;
        0  0 -1  1  0  0  0  0;
        -1  0  0  1  0  0  0  0;
        -1  0  0  0  1  0  0  0;
        0 -1  0  0  0  1  0  0;
        0  0 -1  0  0  0  1  0;
        0  0  0 -1  0  0  0  1;
        0  0  0  0 -1  1  0  0;
        0  0  0  0  0 -1  1  0;
        0  0  0  0  0  0 -1  1;
        0  0  0  0 -1  0  0  1;
    ])

    FE = sparse(Int8[
        1  1  1 -1  0  0  0  0  0  0  0  0;
        0  0  0  0  0  0  0  0 -1 -1 -1  1;
        -1  0  0  0  1 -1  0  0  1  0  0  0;
        0 -1  0  0  0  1 -1  0  0  1  0  0;
        0  0 -1  0  0  0  1 -1  0  0  1  0;
        0  0  0  1 -1  0  0  1  0  0  0 -1;
    ])

    cube = [V, EV, FE]
    cubesRow = (zeros(0,3),spzeros(Int8,0,0),spzeros(Int8,0,0))

    for i in 1:ncubes
        cubesRow = LARLIB.skel_merge(cubesRow..., cube...)
        cube[1] = cube[1] + [zeros(8) zeros(8) ones(8)]
    end

    cubesRow = collect(cubesRow)
    cubesPlane = cubesRow
    num = size(cubesRow[1], 1)
```

```
    for i in 1:ncubes
        cubesPlane = LARLIB.skel_merge(cubesPlane..., cubesRow...)
        cubesRow[1] = cubesRow[1] + [zeros(num) ones(num) zeros(num)]
    end

    cubesPlane = collect(cubesPlane)
    cubesCube = cubesPlane
    num = size(cubesPlane[1], 1)
    for i in 1:ncubes
        cubesCube = LARLIB.skel_merge(cubesCube..., cubesPlane...)
        cubesPlane[1] = cubesPlane[1] + [ones(num) zeros(num) zeros(num)]
    end

    println("Arranging a cube of ", ncubes^3," cubes...")
    rubik = LARLIB.spatial_arrangement(cubesCube...)
    println("DONE")

    rubik = rubik[1] - (.5*ncubes), rubik[2:3]...
    c = cos(pi/6); s = sin(pi/6)
    M1 = [1  0 0; 0 c -s; 0 s c]
    M2 = [c -s 0; s c  0; 0 0 1]
    rot_rubik = rubik[1]*M1*M2, rubik[2:3]...

    println("Arranging two rubik cubes...")
    two_rubiks = LARLIB.skel_merge(rubik..., rot_rubik...)
    println("DONE")

    arranged_rubiks = LARLIB.spatial_arrangement(two_rubiks...)
end
◇
```

Macro referenced in 91.

On the next pages the results are visualized.

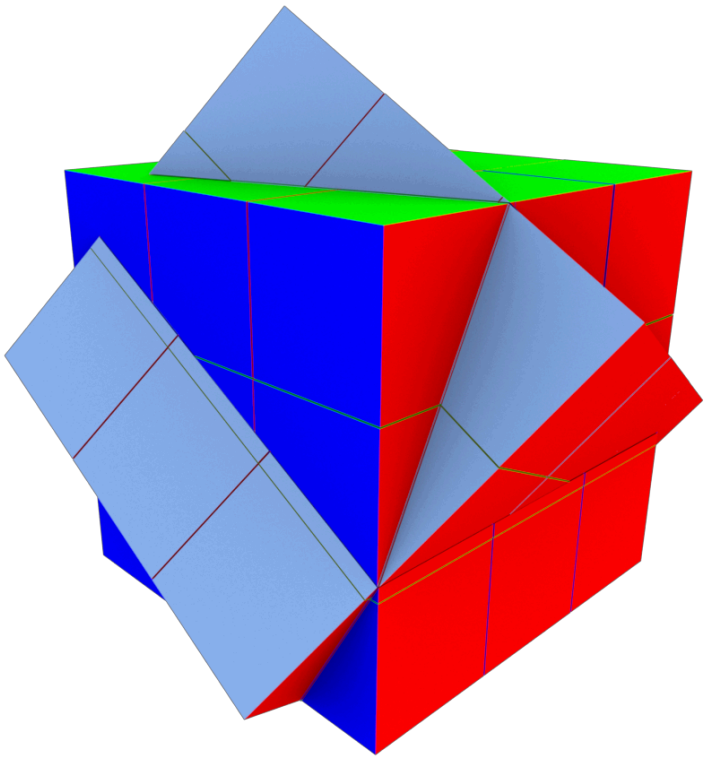


Figure 9.4: Input

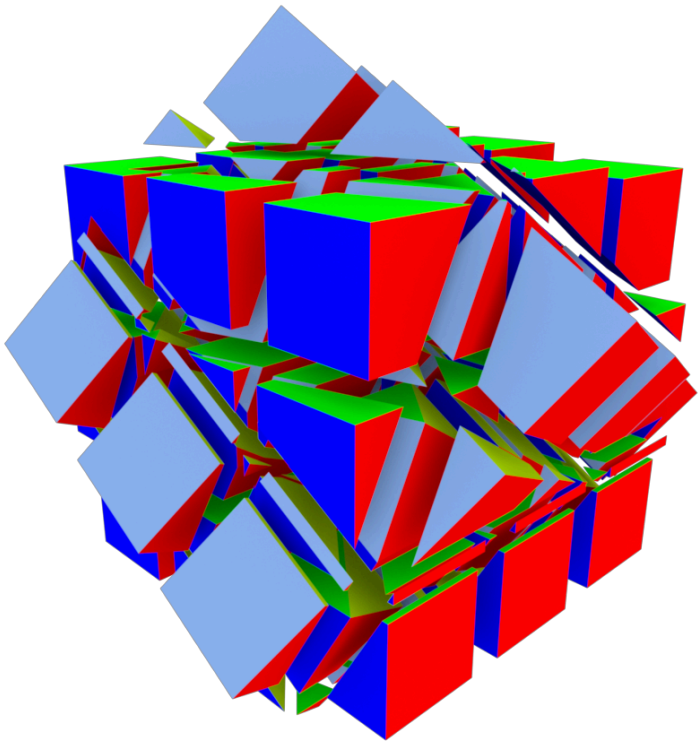


Figure 9.5: Output (Exploded)

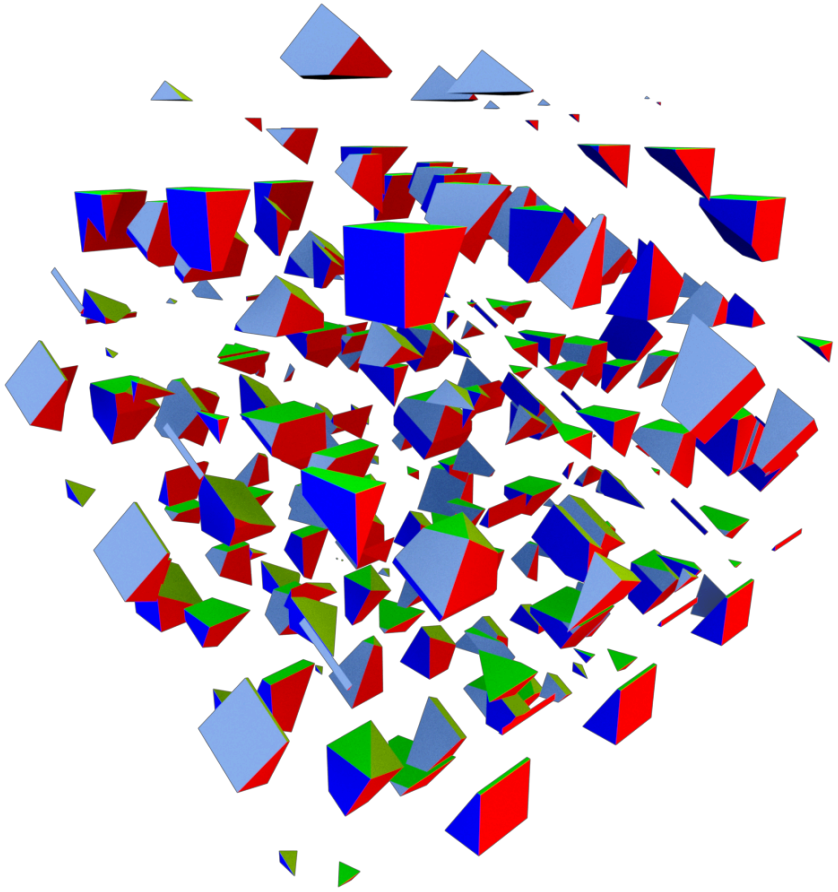


Figure 9.6: Output (More exploded)

Chapter 10

Conclusions

We described in depth the implementation of the merge algorithm as formulated by A. Paoluzzi et al. [12]. We introduced the thesis using a very brief theoretical overview of the algorithm and its applications, and then we explored the implementation starting from the $d = 3$ case down to the $d = 1$ one. At the end we presented two very representative examples to show the capabilities of this algorithm.

10.1 Future developments

10.1.1 Parallelization

In the introduction, we said the algorithm follows the *divide et impera* philosophy; this happens to be a very good thing to do while doing parallel programming. The best way to parallelize the system is to launch a separate job for every face of the 2-skeleton during the fragmentation of the complex, which is the heavier part of the whole algorithm. For this reason, the implementation has been developed to be "parallel ready", in this way, it will be possible to easily make this implementation parallel for real using the Julia parallelization capabilities. This work will be done in the next few weeks.

10.1.2 Boolean operations

As you may have noticed, the arrangement algorithm is not enough to perform Boolean operations by itself. But you may also have noticed we developed this implementation being constantly conscious about the finalities of the algorithm and so we laid the foundations to a future easy implementation of real Boolean operations.

10.1.3 Handling of 3-cells with non-intesecting shells

In this implementation we handled only 2-cells with non-intersecting shells [ref. 2.2] but also 3-cells with non-intersecting shells (which trivially are 3-cells with holes) must be handled. This lack will be fixed as soon as possible following the directions of the ALGORITHM 2 as described by A. Paoluzzi et al. [12].

10.1.4 LAR

This thesis is only the beginning of the Julia implementation of LAR. There are many modules that has been written during the development of the Python version of LAR [ref. 1.2.1] that are ready to be ported on the foundations that the module presented in this thesis laid. The porting of these other modules is planned for the next months.

Bibliography

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. August 1971.

[2] J. Bezanson, S. Karpinski, V. Shah, and A. Edelman. Why we created julia. `https://julialang.org/blog/2012/02/why-we-created-julia`, February 2012.

[3] P. Bourke. Points, lines, and planes. `http://paulbourke.net/geometry/pointlineplane/`, October 1988.

[4] K. Carlsson. `NearestNeighbors.jl`. `https://github.com/KristofferC/NearestNeighbors.jl`, November 2015.

[5] A. DiCarlo, F. Milicchio, A. Paoluzzi, and V. Shapiro. Chain-based representations for solid and physical modeling. *Automation Science and Engineering, IEEE Transactions on*, 6(3):454 –467, July 2009.

[6] Antonio Dicarlo, Alberto Paoluzzi, and Vadim Shapiro. Linear algebraic representation for topological structures. *Comput. Aided Des.*, 46:269–274, January 2014.

[7] F. Furiani. `Triangle.jl`. `https://github.com/furio/TRIANGLE.jl`, May 2017.

[8] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[9] D. C. Jones. `IntervalTrees.jl`. `https://github.com/BioJulia/IntervalTrees.jl`, April 2014.

[10] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, May 1984.

[11] A. Paoluzzi. A robust tile-based algorithm for point/polygon classification. Technical Report 03-86, Dip. di Informatica e Sistemistica, Università 'La Sapienza', June 1986.

[12] A. Paoluzzi, V. Shapiro, and A. DiCarlo. Regularized arrangements of cellular complexes, April 2017.

[13] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[14] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., 1985.

[15] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.