Linear Algebraic Representation

Alberto Paoluzzi, Francesco Furiani, Giulio Martella

August 26, 2017

Contents

Chapter 1

Introduction

"lib/jl/LARLIB.jl" 1 ≡

```
module LARLIB
    module PlanarArrangement
        include("./planar_arrangement.jl")
    end

    function planar_arrangement(V, EV)
        PlanarArrangement.planar_arrangement(V, EV)
    end
end
◇
```

Chapter 2

Planar Arrangement

2.1 Overview

Here we present the planar arrangement algorithm. It takes the 1-skeleton of the σ face and returns complex made of 2-cells. It fragments every edge in `EV`. When the fragmentation is done, coincident vertices are merged into one and useless edges are deleted. At last, 2-cells are build and the result is returned.

```
"lib/jl/planar_arrangement.jl" 2 ≡
    ⟨ planar_arrangement imports 3, ... ⟩
    ⟨ planar_arrangement support functions 5, ... ⟩

    function planar_arrangement(V::Verts, EV::Cells)
        ⟨ planar_arrangement local variables 9 ⟩
        edgenum = size(EV, 1)
        for i in 1:edgenum
            ⟨ Fragment edge 8 ⟩
        end
        ⟨ Put fragmentation results together 10 ⟩
        ⟨ Merge coincident vertices 17 ⟩
        ⟨ Find maximal biconnected components 25 ⟩
        ⟨ Filter biconnected components 26 ⟩
        ⟨ Create faces 28 ⟩

        V, EV, FE
    end
    ◇
```

We include the utilities (ref. 5).

```
⟨ planar_arrangement imports 3 ⟩ ≡
    include("./utilities.jl")
    ◇
```

Macro defined by 3, 12, 29.
Macro referenced in 2.

2.1.1 Tests

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

"test/jl/planar_arrangement.jl" 4 ≡

```
using Base.Test
include("../../lib/jl/planar_arrangement.jl")

⟨ planar_arrangement support functions tests 11, ... ⟩
◇
```

General tests are defined in Appendix A (ref. A.1)

2.2 Edge fragmentation

2.2.1 Support function

The edge fragmentation is performed by using a function called `frag_edge`. It fragments the edge of index `edgenum` into `EV` computing the intersections of it with the other edges into `EV`. It returns the updated vertices list `V` and an `EV` matrix that contains the freshly computed edges. For every edge in `EV`, it needs to check if `edge`(the edge of index `edgenum` into `EV`) intersects with it. This is done through `intersect_edges` (ref. 2.2.2); this function takes two edges and returns a list of the intersections of the first edge with the second one; every entry of this list is a tuple made of the intersection point and a normalized intersection parameter. When there is an intersection, the new point is be pushed into the `V` matrix while the parameter is stored into the `alphas` dictionary as a key coupled to the new point index. When every possible intersection is found, the keys in `alphas` are sorted and, on the base of that, a new `EV` is computed.

⟨ planar_arrangement support functions 5 ⟩ ≡

```
function frag_edge(V::Verts, EV::Cells, edgenum::Int)
    alphas = Dict{Float64, Int}()
    edge = EV[edgenum, :]
    for i in 1:size(EV, 1)
        if i != edgenum
            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)))
    cells_num = length(alphas_keys)-1
    verts_num = size(V, 1)
    EV = spzeros(Int8, cells_num, verts_num)
```

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

    V, EV
end
◇
```

Macro defined by 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

2.2.2 Edge intersections

We used the method presented by Bourke [1] to calculate the intersection of two edges. 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 {`Verts, Float64`} which represent the intersection point and a parameter that is useful for sorting the fragmentation points of an edge.

⟨ planar_arrangement support functions 6 ⟩ ≡

```
function intersect_edges(V::Verts, edge1::Cell, edge2::Cell)
    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}()
    denom = (y4-y3)*(x2-x1) - (x4-x3)*(y2-y1)
    a = ((x4-x3)*(y1-y3) - (y4-y3)*(x1-x3)) / denom
    b = ((x2-x1)*(y1-y3) - (y2-y1)*(x1-x3)) / denom

    if 0 <= a <= 1 && 0 <= b <= 1
        p = [(x1 + a*(x2-x1))  (y1 + a*(y2-y1))]
        push!(ret, (p, a))
    elseif isnan(a) && isnan(b)
        ⟨Handle colinear edges 7⟩
    end
    return ret
end
◇
```

Macro defined by 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

If the ⟨ Handle colinear edges ⟩ macro is run, we are sure that the four vertices of `edge1` and `edge2` are colinear. So, 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:

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

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 7 ⟩ ≡

```
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 6.

2.2.3 Implementation

When we need to fragment an `edge` we use the `frag_edge` function (ref. 2.2.1) and we simply update `V` and push the small `ev` matrix into a list of cells called `EVs`. We also keep the number of cells into `finalcells_num` to build `EV` with ease.

⟨ Fragment edge 8 ⟩ ≡

```
V, ev = frag_edge(V, EV, i)
finalcells_num += size(ev, 1)
push!(EVs, ev)
◇
```

Macro referenced in 2.

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

⟨ planar_arrangement local variables 9 ⟩ ≡

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

Macro referenced in 2.

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. The process is not immediate because every entry of the list has columns relative to the number of vertices in `V` at the moment of the computation.

⟨ Put fragmentation results together 10 ⟩ ≡

```
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 2.

2.2.4 Tests

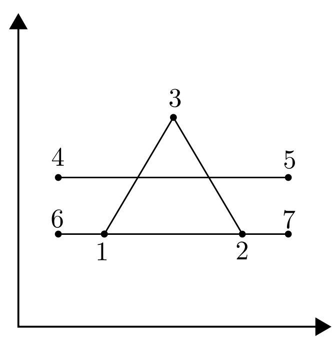


Figure 2.1: The bunch of edges used for the tests.

⟨ planar_arrangement support functions tests 11 ⟩ ≡

```
@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 = intersect_edges(V, EV[5, :], EV[1, :])
        inters2 = intersect_edges(V, EV[1, :], EV[4, :])
        inters3 = 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 = 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 11, 18, 27, 41.
Macro referenced in 4.

2.3 Coincident vertices merge

2.3.1 Support function

The merge of coincident is done in the `merge_vertices` function. This relies on the `NearestNeighbors.jl` package [2] which provides a reliable implementation of the `KDTree` data structure.

⟨ planar_arrangement imports 12 ⟩ ≡

```
using NearestNeighbors
◇
```

Macro defined by 3, 12, 29.
Macro referenced in 2.

⟨ planar_arrangement support functions 13 ⟩ ≡

```
function merge_vertices(V::Verts, EV::Cells, err=1e-4)
    kdtree = KDTree(V’)
    tocheck = collect(size(V,1):-1:1)
    todelete = Array{Int64, 1}()
    ⟨ Iterate over tocheck 14 ⟩
    ⟨ Delete vertices in todelete 15 ⟩
    ⟨ Delete superfluous cells 16 ⟩
    V,EV
end
◇
```

Macro defined by 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

We create two stacks: `tocheck` which contains the indices of the vertices to check and `todelete` that stores the indices of the vertices to delete later. Into `tocheck` we put all the vertices of the complex in reverse order (in this way we can pop from the stack the indices in crescent order). So, until `tocheck` is not empty, we pop a vertex `vi` from the stack and for each coincident vertex `vj`, we put it into the `todelete` stack and we sum the columns of `EV` relative to `vi` and `vj`

⟨ Iterate over tocheck 14 ⟩ ≡

```
while !isempty(tocheck)
    vi = pop!(tocheck)
    if !(vi in todelete)
        nearvs = inrange(kdtree, V[vi, :], err)
        for vj in nearvs
            if vj != vi
                push!(todelete, vj)
                EV[:,vi] = EV[:, vi] + EV[:, vj]
            end
        end
    end
end
◇
```

Macro referenced in 13.

We then calculate the vertices to keep and we filter out the data relative to the vertices into `todelete`.

⟨ Delete vertices in todelete 15 ⟩ ≡

```
tokeep = setdiff(collect(1:size(V,1)), todelete)
EV = EV[:, tokeep]
V = V[tokeep, :]
◇
```

Macro referenced in 13.

At last we delete duplicated, empty and broken edges.

⟨ Delete superfluous cells 16 ⟩ ≡

```
tokeep = Array{Int64, 1}()
cells = [Set(EV[i, :].nzind) for i in size(EV,1):-1:1]
i = 0
while !isempty(cells)
    i += 1
    c = pop!(cells)
    if !(length(c) != 2 || c in cells)
        push!(tokeep, i)
    end
end
EV = EV[tokeep, :]
◇
```

Macro referenced in 13.

2.3.2 Implementation

We simply call `merge_vertices` (ref. 2.3.1).

⟨ Merge coincident vertices 17 ⟩ ≡

```
V, EV = merge_vertices(V, EV)
◇
```

Macro referenced in 2.

2.3.3 Tests

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

⟨ planar_arrangement support functions tests 18 ⟩ ≡

```
@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 = 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 11, 18, 27, 41.
Macro referenced in 4.

2.4 Maximal biconnected components

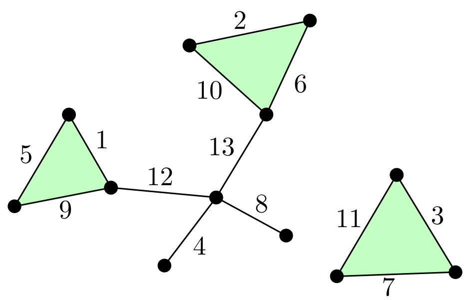


Figure 2.2: An example graph where the maximal biconnected components are highlighted in green and the edges are numbered. We have here three components formed by the sets of edges 1,5,9, 2,6,10 and 3,11,7

2.4.1 Support function

To individuate the maximal biconnected components of the fragmented and merged 1-skeleton we use the well know 1973 Hopcroft-Tarjan algorithm for biconnected components [3].

⟨ planar_arrangement support functions 19 ⟩ ≡

```
function biconnected_components(EV::Cells)
    ⟨ biconnected_components local variables 20 ⟩
    ⟨ DFS utilities 21 ⟩
    ⟨ Depth first visit 22 ⟩
    bicon_comps
end
◇
```

Macro defined by 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

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 20 ⟩ ≡

```
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 19.

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 21 ⟩ ≡

```
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 19.

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 22 ⟩ ≡

```
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 24 ⟩
        else
            if ps[end][3] == ps[end-1][2]
                ⟨ Form biconnected component 23 ⟩
            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 19.

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 23 ⟩ ≡

```
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 22.

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 24 ⟩ ≡

```
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 22.

2.4.2 Implementation

Like for the vertices merge we simply call the freshly implemented `biconnected_components` function (ref. 2.4.1). If no biconnected components are found, the procedure will stop and return nothing.

⟨ Find maximal biconnected components 25 ⟩ ≡

```
bicon_comps = biconnected_components(EV)

if isempty(bicon_comps)
    println("No biconnected components found.")
    return
end
◇
```

Macro referenced in 2.

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`.

⟨ Filter biconnected components 26 ⟩ ≡

```
todel = setdiff(collect(1:size(EV,1)), union(bicon_comps...))
EV = EV[union(bicon_comps...), :]

vertinds = 1:size(EV, 2)
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, :]
    ◇
```

Macro referenced in 2.

2.4.3 Tests

The graph built here is the one of figure 2.2.

⟨ planar_arrangement support functions tests 27 ⟩ ≡

```
@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 = 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 11, 18, 27, 41.
Macro referenced in 4.

2.5 Faces creation

2.5.1 Implementation

⟨ Create faces 28 ⟩ ≡

```
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[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 31 ⟩
⟨ Transitive reduction 35 ⟩
⟨ Cell merging 37 ⟩

◇

Macro referenced in 2.

⟨ planar_arrangement imports 29 ⟩ ≡

```
include("./minimal_cycles.jl")
```

◇

Macro defined by 3, 12, 29.
Macro referenced in 2.

2.5.2 Individuate the external cell

Once we computed the minimal 2-cycles (ref. 4) 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 Chapter 5 (ref. 5.3)

⟨ planar_arrangement support functions 30 ⟩ ≡

```
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 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

2.5.3 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 is also visually explained in the tests: ref. 2.5.6)

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 of chapter 5 (ref. 5.2)

⟨ Containment test 31 ⟩ ≡

```
    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 28.

⟨ planar_arrangement support functions 32 ⟩ ≡

```
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 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

The ray logic is explained just below the next macro.

⟨ planar_arrangement support functions 33 ⟩ ≡

```
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
                    contains = false
                    ⟨ Shoot ray 34 ⟩
                    if !contains
                        graph[i, j] = 0
                    end
                end
             end
         end

     end
     return graph
end
◇
```

Macro defined by 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

We shoot a ray from the vertex o with the same direction of the positive x_1 semi-axis. When we want to find the intersection of the ray with an edge, we parametrize it:

$$p = p_1 + \alpha(p_2 - p_1), \quad \alpha \in [0, 1] \tag{2.1}$$

Then we find the value that α assumes when the edge intersects the line parallel to the x_1 axis that passes through o:

$$\begin{bmatrix} o_x \\ o_y \end{bmatrix} = \begin{bmatrix} p_{1x} \\ p_{1y} \end{bmatrix} + \alpha \begin{bmatrix} p_{2x} - p_{1x} \\ p_{2y} - p_{1y} \end{bmatrix}$$
$$o_y = p_{1y} + \alpha(p_{2y} - p_{1y})$$
$$\alpha = \frac{o_y - p_{1y}}{p_{2y} - p_{1y}}$$

If $\alpha \in [0, 1]$ then the edge intersect the line, and we find the point of intersection by putting α into equation 2.1. The intersection must be count only if the freshly computed point is on the right of the vertex o.

The case when the ray encounters a vertex requires additional care: we are testing the intersections of the ray with edges, and every vertex is shared by two or more edges. So we cannot simply increase the `hits` counter every time we encounter a vertex because this will lead to miscalculations when an even number of edges share the same vertex. We resolve this by storing the already visited vertices into the `visited_verts` list.

⟨Shoot ray 34⟩ ≡

```
hits = 0
visited_verts = []
shell_edge_indexes = shells[j].nzind
ev = EVs[j][shell_edge_indexes, :]

for edge in 1:ev.m
    a_id, b_id = ev[edge, :].nzind
    a = V[a_id, :]
    b = V[b_id, :]
    v = b - a
    alpha = (origin[2] - a[2]) / v[2]
    if 0 <= alpha <= 1
        x_int = a[1] + v[1]*alpha
        if x_int > origin[1]
            if 0 <= alpha <= 1
                hits += 1
            else
                p = (alpha == 0) ? a : b
                if !(p in visited_verts)
```

```
                    hits += 1
                    push!(visited_verts, p)
                end
            end
        end
    end
end

contains = hits % 2 == 1
◇
```

Macro referenced in 33.

2.5.4 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 [5] we have:

⟨ Transitive reduction 35 ⟩ ≡

```
transitive_reduction!(containment_graph)
◇
```

Macro referenced in 28.

⟨ planar_arrangement support functions 36 ⟩ ≡

```
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 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

2.5.5 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. The bounding box logic is implemented in the utilities of chapter 5 (ref. 5.2, please note that that the `bboxes` is not part of the utilities but it is defined in the next paragraph). The `sums` array contains the indexes of the rows of the various boundary 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 37 ⟩ ≡

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

Macro referenced in 28.

⟨ planar_arrangement support functions 38 ⟩ ≡

```
function cell_merging(n, containment_graph, V, EVs, boundaries, shells, shell_bboxes)
    ⟨ Cell merging support functions 39 ⟩

    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 40 ⟩
    return EV, FE
end
◇
```

Macro defined by 5, 6, 13, 19, 30, 32, 33, 36, 38.
Macro referenced in 2.

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

⟨ Cell merging support functions 39 ⟩ ≡

```
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 38.

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 40 ⟩ ≡

```
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 38.

2.5.6 Tests

⟨ planar_arrangement support functions tests 41 ⟩ ≡

```
@testset "Face creation" begin
    ⟨ Face creation tests 42, ... ⟩
end
◇
```

Macro defined by 11, 18, 27, 41.
Macro referenced in 4.

External cell individuation

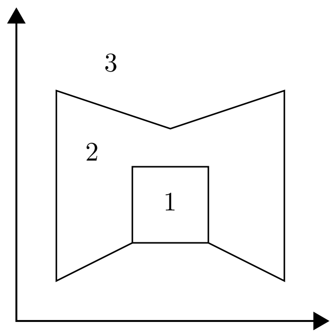


Figure 2.3: This biconnected component has three faces. The external one is the number 3. This is a particularly difficult case because the most "external" vertices of face 2 are in common with the external cell.

⟨ Face creation tests 42 ⟩ ≡

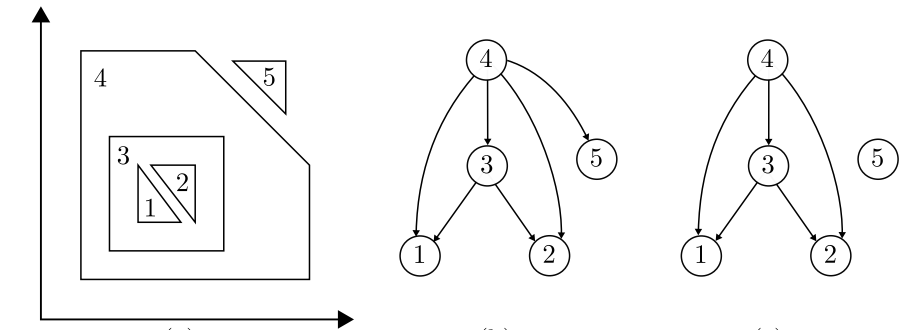
```
@testset "External cell individuation" begin
    V = [ .5 .5;  1.5   1;  1.5  2;
         2.5  2;  2.5   1;  3.5 .5;
         3.5  3;    2 2.5;   .5  3]

    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 get_external_cycle(V, EV, FE) == 3
end
```

◇

Macro defined by 42, 43, 44, 45.
Macro referenced in 41.

Containment test

⟨ Face creation tests 43 ⟩ ≡



(a) (b) (c)

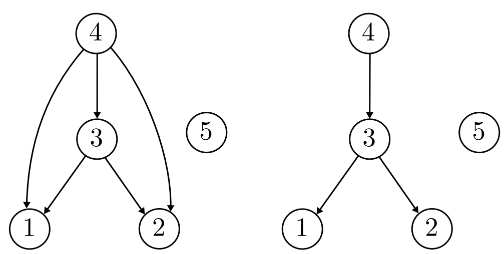
Figure 2.4: (a) is our test case. The numbers identify the connected components. (b) is the containment graph built using only the `pre_containment_test` function. The arc (4,5) is in there because the bounding box of the component no. 5 is completely contained in the bounding box of no. 4. (c) shows 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 -1  1  0  0  0  0  0;
                0  0  0  0  0  0  0  0  0 -1  0  1  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 -1  1  0  0;
                0  0  0  0  0  0  0  0  0  0  0  0 -1  0  1  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 -1  1  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  1  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 -1  1  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  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]
    EV5 = Int8[ 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  1;
                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, bbox(V[vs_indexes, :]))
    end

    graph = 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 = 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 42, 43, 44, 45.
Macro referenced in 41.

Transitive reduction



(a) (b)

Figure 2.5: Before (a) and after (b) transitive reduction performed on the graph of the previous test set.

⟨ Face creation tests 44 ⟩ ≡

```
@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]
    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 42, 43, 44, 45.
Macro referenced in 41.

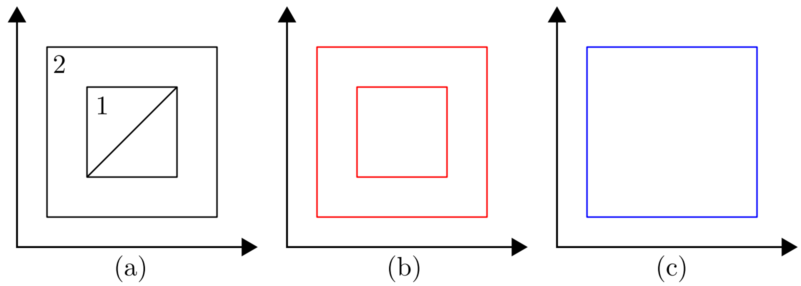


Figure 2.6: Here we have two biconnected components, one inside the other (a). If we don't perform cell merging, the boundary of the arranged set will be the red one (b), which is incorrect. The correct boundary is the blue one (c).

Cell merging

⟨ Face creation tests 45 ⟩ ≡

```
@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, bbox(V[vs_indexes, :]))
    end

    EV, FE = 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 42, 43, 44, 45.
Macro referenced in 41.

Chapter 3

Dimension travel

3.1 Overview

"lib/jl/dimension_travel.jl" 46 ≡
    ⟨ Imports and aliases 47, ... ⟩
    ⟨ Dimension travel functions 49, ... ⟩
    ◇

We define some aliases to standardize data formats.

⟨ Imports and aliases 47 ⟩ ≡

```
typealias Verts Array{Float64, 2}
typealias Cells SparseMatrixCSC{Int8, Int}
typealias Cell SparseVector{Int8, Int}
```

◇

Macro defined by 47, 52.
Macro referenced in 46.

3.1.1 Tests

"test/jl/dimension_travel.jl" 48 ≡

```
using Base.Test
include("../../lib/jl/dimension_travel.jl")
```

    ⟨ Tests 51 ⟩
    ◇

3.2 Normalization

This function returns the direct and inverse transformation that normalizes every point in `V` to mek them fit into the unitary d-dimensional hyper-cube. First of all, the function computes the bounding box of the input points. Then it combines a translation and a scaling matrix. Lastly, it inverts the matrix and returns both the direct and the inverse matrix.

⟨ Dimension travel functions 49 ⟩ ≡

```
function normalizer(V::Verts)
    d = size(V, 2)
    upper = mapslices(x->max(x...), V, 1)
    lower = mapslices(x->min(x...), V, 1)
    diff = upper-lower

    T = eye(d+1)
    T[d+1, 1:d] = -lower

    S = eye(d+1)
    S[1:d, 1:d] = diagm(vec(map(inv, diff)))

    mat = T*S
    mat, inv(mat)
end
◇
```

Macro defined by 49, 50, 53.
Macro referenced in 46.

3.3 Submanifold mapping

This function, given a list of vertices `V` (in $\mathbb{E}^3$) and a `face`, returns a 4×4 transformation matrix that "flattens" `face` on the $x_3 = 0$ plane. The matrix is made of the composition of a translation matrix and a coordinate reference change. The translation makes the first vertex of `face` coincident to the origin and the coordinate reference change is computed by building a reference system of the plane on which `face` lays.

⟨ Dimension travel functions 50 ⟩ ≡

```
function submanifold_mapping(V, face)
    p1, p2, p3 = map(i->V[face.nzind[i], :], 1:3)
    u1 = p2-p1
    u2 = p3-p1
    u3 = cross(u1, u2)
    T = eye(4)
    T[4, 1:3] = -p1
    M = eye(4)
    M[1:3, 1:3] = [u1 u2 u3]
    return T*M
end
◇
```

Macro defined by 49, 50, 53.
Macro referenced in 46.

3.3.1 Tests

⟨ Tests 51 ⟩ ≡

```
V = rand(3, 3)
face = sparsevec(Int8[1 1 1])
m = submanifold_mapping(V, face)
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 48.

3.4 Spatial index computation

The aim of this function is to compute a *spatial index* that maps each cell to a set of cells which it may collide with. This is achieved by profuse use of bounding boxes and interval trees. These last ones are implemented with the `IntervalTrees.jl` package (https://github.com/BioJulia/IntervalTrees.jl)

⟨ Imports and aliases 52 ⟩ ≡

```
using IntervalTrees
◇
```

Macro defined by 47, 52.
Macro referenced in 46.

⟨ Dimension travel functions 53 ⟩ ≡

```
function spatial_index(V::Verts, CV::Cells)
    d = size(V,2)
    cell_num = size(CV, 1)
    ⟨ Build the d-IntervalTrees 54 ⟩
    ⟨ Create the mapping 55 ⟩
    mapping
end
◇
```

Macro defined by 49, 50, 53.
Macro referenced in 46.

The basic idea is to "unfold" every d-dimensional bounding box into d one-dimensional boxes. 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.

⟨ Build the d-IntervalTrees 54 ⟩ ≡

```
IntervalsType = IntervalValue{Float64, Int64}
boxes1D = Array{IntervalsType, 2}(0, d)
for ci in 1:cell_num
    intervals = map((l,u)->IntervalsType(l,u,ci), bbox(V,CV[ci, :])...)
    boxes1D = vcat(boxes1D, intervals)
end
trees = mapslices(IntervalTree{Float64, IntervalsType}, sort(boxes1D, 1), 1)
◇
```

Macro referenced in 53.

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 55 ⟩ ≡

```
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 ci in 1:cell_num
    cell_indexes = setdiff(intersect_intervals(boxes1D[ci, :]), [ci])
    push!(mapping, cell_indexes)
end
◇
```

Macro referenced in 53.

Chapter 4

Minimal cycles computation

4.1 Main function

Computing the minimal cycles means to compute the d-boundary matrix from the $(d-1)$-boundary. This function works for both $d = 2$ and $d = 3$; the only difference between the two cases lays in the `angles_fn` function (ref. 4.2). To support this multidimensional behavior, the algorithm has been implemented as an high-order function[1]:

"lib/jl/minimal_cycles.jl" 56 ≡
```
include("./utilities.jl")

⟨ Minimal cycles implementations 63 ⟩

function minimal_cycles(angles_fn::Function)

    function _minimal_cycles(V::Verts, ld_bounds::Cells)
        ⟨ Function body 57 ⟩
    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 (this is useful to determine the direction in which the cell must be visited next)

[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.

⟨ Function body 57 ⟩ ≡
```
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 60 ⟩
⟨ minimal_cycles utilities 58, ... ⟩

while (sigma = get_seed_cell()) > 0
    ⟨ Compute a cycle 59 ⟩
end

return d_bounds
◇
```
Macro referenced in 56.

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 58 ⟩ ≡
```
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 58, 61, 62.
Macro referenced in 57.

The bigger part of the algorithm is the computation of a single cycle. It is mostly equivalent to the **ALGORITHM 1** presented by A. Paoluzzi et al. in *Arrangements of cellular complexes* [4]

⟨ Compute a cycle 59 ⟩ ≡
```
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 57.

As profusely explained by A. Paoluzzi et al. [4], 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 60 ⟩ ≡

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

Macro referenced in 57.

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 61 ⟩ ≡

```

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

Macro defined by 58, 61, 62.
Macro referenced in 57.

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 62 ⟩ ≡

```
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 58, 61, 62.
Macro referenced in 57.

4.2 Dimensional wise implementations

4.2.1 $d = 2$

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 63 ⟩ ≡

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

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

    for i in 1:ev.m
        j = ev[i,:].nzind[1]
        ev[i, j] = -1
    end
    VE = ev'

    EF = minimal_cycles(edge_angle)(V, VE)

    return EF'
end
◇
```

Macro referenced in 56.

4.2.2 $d = 3$

TODO

Chapter 5

Utilities

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

"lib/jl/utilities.jl" 64 ≡
⟨ Aliases 66 ⟩
⟨ Utilities 67, … ⟩
◇

5.0.1 Tests

As usual every function has some unit tests.

"test/jl/utilities.jl" 65 ≡

```
using Base.Test
include("../../lib/jl/utilities.jl")
```

⟨ Utilities tests 68, … ⟩
◇

5.1 Types

To store vertices and cells boundary matrices, we use types already built in the standard Julia. We use these aliases to standardize the types used throughout LAR.

⟨ Aliases 66 ⟩ ≡

```
const Verts = Array{Float64, 2}
const Cells = SparseMatrixCSC{Int8, Int}
const Cell = SparseVector{Int8, Int}
```

◇

Macro referenced in 64.

5.2 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 bounding boxes.

⟨ Utilities 67 ⟩ ≡
```
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 67, 69, 71.
Macro referenced in 64.

5.2.1 Tests

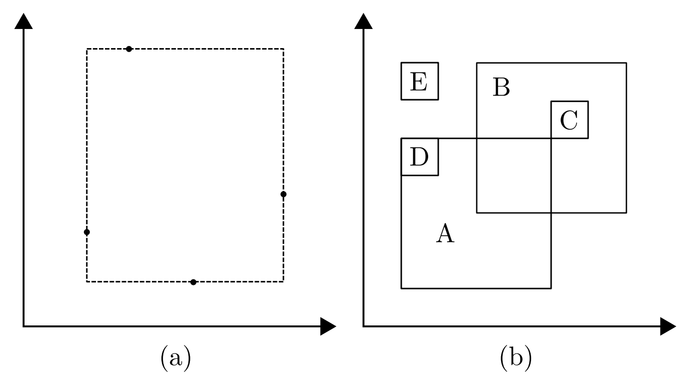


Figure 5.1: (a) is a visualization of the test for bboxes building, (b) for bbox containment.

⟨ Utilities tests 68 ⟩ ≡
```
@testset "Bounding boxes building test" begin
    V = [.56 .28; .84 .57; .35  1.0; .22  .43]
    @test 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 bbox_contains(bboxA, bboxD)
    @test bbox_contains(bboxB, bboxC)
    @test !bbox_contains(bboxA, bboxB)
    @test !bbox_contains(bboxA, bboxE)
end
◇
```

Macro defined by 68, 70.
Macro referenced in 65.

5.3 Face area calculation

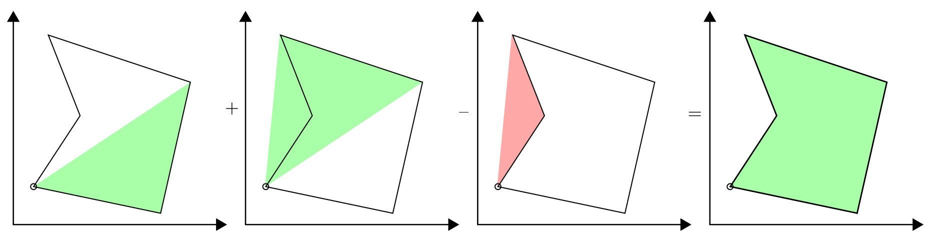


Figure 5.2: A visual representation of the face area calculation algorithm. The area of the face is the sum of the areas of each triangle which can be build using the pivot vertex and the other vertices of the face

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 69 ⟩ ≡

```
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
    ps = [0, 0, 0]

    for i in face.nzind
        edge = face[i]*EV[i, :]
        skip = false

        for e in edge.nzind
            if e != ps[1]
                if edge[e] < 0
                    if ps[1] == 0
                        ps[1] = e
                        skip = true
                    else
                        ps[2] = e
                    end
                else
                    ps[3] = e
                end
            else
                skip = true
                break
            end
        end

        if !skip
            area += triangle_area(V[ps, :])
        end
    end

    return area
end
◇
```

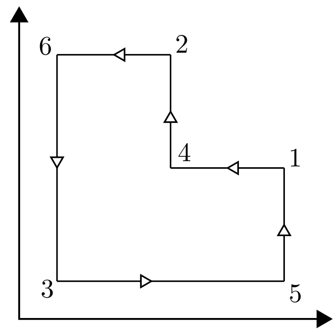
Macro defined by 67, 69, 71.
Macro referenced in 64.

5.3.1 Tests

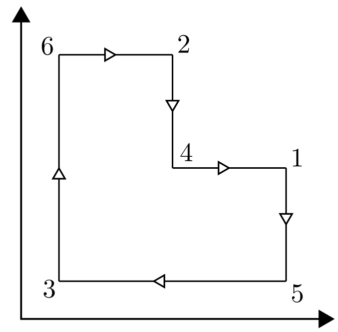
The two faces drawn above they must have complimentary area.

⟨ Utilities tests 70 ⟩ ≡

```
@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]
```



(a)



(b)

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

Macro defined by 68, 70.
Macro referenced in 65.

5.4 Skeletal merge

It is generally useful to have a utility to merge the skeletons of two $d = 2$ cellular complexes.

⟨ Utilities 71 ⟩ ≡

```
function skel_merge(V1::Verts, V2::Verts, EV1::Cells, 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
◇
```

Macro defined by 67, 69, 71.
Macro referenced in 64.

Appendix A

Tests

"test/jl/runtests.jl" 72 ≡
```
include("./planar_arrangement.jl")
include("./dimension_travel.jl")
include("./utilities.jl")
◇
```

"test/jl/general_tests.jl" 73 ≡
```
using Base.Test

⟨ planar_arrangement tests 74 ⟩
◇
```

A.1 Planar arrangement tests

Here we present some general tests for the `planar_arrangement` function (ref. 2)

⟨ planar_arrangement tests 74 ⟩ ≡
```
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 73.

Bibliography

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

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

[3] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. http://doi.acm.org/10.1145/362248.362272, June 1973.

[4] A. Paoluzzi V. Shapiro and A. Dicarlo. Arrangements of cellular complexes. https://arxiv.org/abs/1704.00142, March 2017.

[5] D. Bosnacki W. Ligtenberg M. Odenbrett A. Wijs and P. Hilbers. Parallel algorithm for transitive reduction for weighted graphs. https://www.win.tue.nl/~awijs/articles/TransitiveReductionMM.pdf, 2010.