LARLIB.jl

Alberto Paoluzzi, Francesco Furiani, Giulio Martella

January 6, 2018

Book structure

This book has two parts: Introduction and Implementation.

The first chapter of the Introduction explain what LAR is and explores the finalities and goals of LARLIB.jl. The other chapters focus on explaining the algorithms behind LAR.

The Implementation part start with a brief overview of the LARLIB.jl module and the conventions used throughout the module. The chapters following contain the in-deep explanation of the code inside each source file of LARLIB.jl; most of the content is paired with test sets: these are embedded into the relative chapters. The last chapter contains the utilities function used everywhere inside LARLIB.jl.

Contents

Part I

Introduction

Chapter 1

LAR

LAR is a general representation scheme for geometric and topological modeling [6]. 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.

1.1 LAR fundamentals

LAR model A LAR model is a pair *geometry, topology*. The *geometry* is specified by the position vectors of *vertices* in a Euclidean space $\mathbb{E}^d$ of points

with d coordinates. The *topology* is specified by one or more bases of singleton k-chains (i.e. k-cells) for $0 \leq k \leq d$. The vertex sharing between cells implicitly provide the attachment maps between cells of various dimensions. Vertex positions are represented, by rows, by a 2-array of d real coordinates.

Chains as arrays Chain-based modeling and computing is based on representation of p-cell subsets as *chains*, elements of *linear* spaces C_p $(0 \leq p \leq d)$ generated by the space decomposition induced by a *cellular complex*, also said *CW- complex*. Chains can be simply represented as arrays of signed integers, one of simplest and more efficient data structure of most languages, particularly when oriented to scientific computing. Therefore, basic algebraic operations on chains as vectors (sum and product times a scalar) are implemented over arrays.

Characteristic matrices The LAR *representation scheme* [**?**], i.e. our mapping between mathematical models of solids and their computer representations, uses linear *chain spaces* C_p as models, and sparse *characteristic matrices* M_p of p-cells as symbolic representations, where the p-cell $\sigma^k \in \Lambda_p$ is represented as the k-th binary row of the sparse characteristic matrix $M_p : C_0 \to C_p$.

Boundary and coboundary matrices The boundary matrix $[\partial_p]$ is the matrix of the boundary operator $\partial_p : C_p \to C_{p-1}$ $(1 \leq p \leq d)$ that for each *chain* $c_p \in C_p$ returns the boundary $(p-1)$-cycle of its $(d-1)$-faces. A boundary operator is linear: $\partial_p c + d = \partial_p c + \partial_p d$, for each $c, d \in C_p$. A cycle is a chain without boundary. Hence, t*he* boundary of a boundary is the zero map: $partial_{p-1} \circ partial_p = 0$ $(2 \leq p \leq d)$.

Incidence matrices Incidence operators between chain spaces of different dimension are easy to compute by matrix products of characteristic matrices (see Section **??**), possibly transposed. Since both characteristic and operator matrices are very sparse, their products are computed with specialized algorithms for very sparse matrices, whose complexity is roughly linear in the size of the output sparse matrix, i.e., in the number of its stored non-zero elements. Incidence queries and other types of geometric or topological computations are not performed element-wise, that necessarily require iterative or recursive programming patterns, but only require matrix product times whole chains (sets of cells), so adapting naturally to parallel and/or dataflow computational patterns found in HPC and CNN architectures.

Validity test of a representation Data validity is easy to test by checking for satisfaction of basic equations $\partial\partial = \emptyset$ of a chain complex.

1.2 Data structures and transformations

In this section we introduce the basic data objects of the `LARLIB` module, i.e., incidence relations, chain bases, and chain transformations. The ground concept

of a *chain complex* is a cellular decomposition of a d-space into sets of connected p-cells ($0 \leq p \leq d$).

1.2.1 Examples

Example 1 (3D cubical mesh)**.** *A very simple example of 3D cubical mesh is given here. When considered as a solid model, a* boundary representation *would be normally applied to it. If used as the domain of a simulation, a* decompositive representation *would be usually chosen. Both can be derived by the LAR scheme, given below. A picture of the mesh with numbered p-cells in different colors ($0 \leq p \leq 3$) is shown in Figure 1.1.*

⟨ *LAR decompositive representation 1* ⟩ ≡

```
using LARVIEW

# Simple mesh of two 3-cuboids
V,cells = LARLIB.larCuboids([2,1,1],true)
VV,EV,FV,CV = cells
larmodel = V,(VV,EV,FV,CV)
LARVIEW.viewnumbered(larmodel)
```

◇

Macro never referenced.

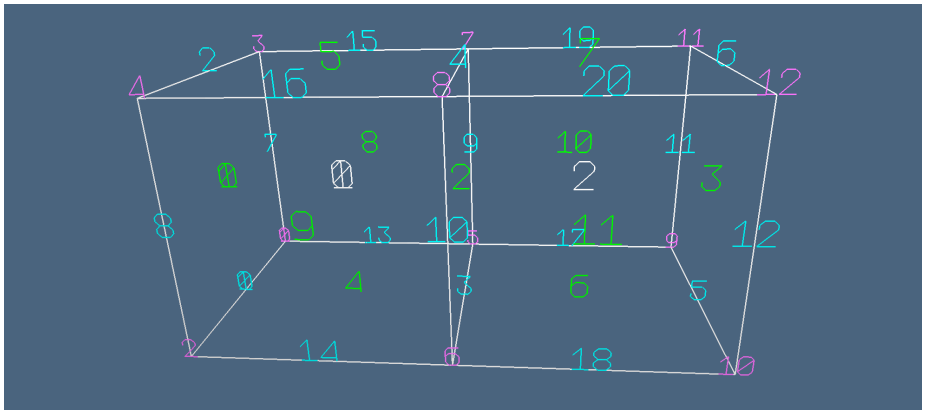


Figure 1.1: A cubical 3-mesh with two 3-cells. Notice that p-cells of index 1 are also indexed by 0 in the picture, due to (current) incompatibilities by the `pyplasm` library, used for mesh drawing, and Julia.

The geometric embedding of the grid, given the array V of vertices, and the indexing of p-cells as subsets of vertices are given below.

```
julia> V
312 Array{Int64,2}:
 0  0  0  0  1  1  1  1  2  2  2  2
```

```
 0  0  1  1  0  0  1  1  0  0  1  1
 0  1  0  1  0  1  0  1  0  1  0  1
julia> print(VV)
Array{Int64,1}[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]]
julia> print(EV)
Array{Int64,1}[[1,2],[3,4],[5,6],[7,8],[9,10],[11,12],[1,3],[2,4],[5,7],
[6,8],[9,11],[10,12],[1,5],[2,6],[3,7],[4,8],[5,9],[6,10],[7,11],[8,12]]
julia> print(FV)
Array{Int64,1}[[1,2,3,4],[5,6,7,8],[9,10,11,12],[1,2,5,6],[3,4,7,8],
[5,6,9,10],[7,8,11,12],[1,3,5,7],[2,4,6,8],[5,7,9,11],[6,8,10,12]]
julia> print(CV)
Array{Int64,1}[[1,2,3,4,5,6,7,8],[5,6,7,8,9,10,11,12]]
```

Example 2 (Input/output of space arrangement algorithm)**.**

1.2.2 Data structures

All data structures are either arrays, or sparse arrays, or arrays of arrays. We introduce here some stardard notations, largely used in the following. In particular, we use the symbols `V`, `E`, `F`, `C`, to denote the arrays of *vertices*, *edges*, *faces*, and *cells*, i.e., to denote the 0-, 1-, 2-, and 3-cells, respectively.

With some abuse of language, each *p-cell* or *elementary p-chain* is represented as the `Array{Int64,1}` of the integer indices of its $(p-1)$-faces, either signed or non-signed depending on the consideration of the orientations of the cells.

Incidence relations By definition of cellular complex, a q-cell τ and a r-cell σ, with $q \leq r$, either do not intersect or share a common *p-face* $(0 \leq p \leq q, r)$.

Hence in 3D we may consider up to $16 = 4 \times 4$ binary incidence relations between V, E, F, C, denoted respectively `VV`, `VE`, `VF`, `VC`, and `EV`, `EE`, `EF`, `EC`, etc, etc. Such incidence relations are represented in Julia as `Array{Array{Int64,1},1}` of integer indices. With reference to the cellular complex in Figure 1.1:

```
EV = [[1,2],[3,4],[5,6],[7,8],[9,10],[11,12],[1,3],[2,4],[5,7],[6,8],
      [9,11],[10,12],[1,5],[2,6],[3,7],[4,8],[5,9],[6,10],[7,11],[8,12]]
```

where `EV[4] = [7,8]` simply means the 1-cell (edge) indexed by 4 is incident to vertices indexed by 7 and 8. Analogous readings hold for all other binary relationships.

Chain bases A chain complex is a sequence of linear boundary transformations between a sequence of linear spaces of chains. The set of p-cells, considered as the set of singleton p-chains, provides the basis for generating the whole set of p-chains via linear combinations with the elements of an additive group, normally either $(+, \mathbb{Z}_2 = \{0, 1\})$ or $(+, \mathbb{Z}_3 \simeq \{-1, 0, 1\})$.

LAR describes the bases of p-chains $(0 \leq p \leq 3)$ using sparse characteristic matrices M_0, M_1, M_2, M_3 derived from the relations `VV`, `EV`, `FV`, `CV`, respectively (see [6]). Their compressed-sparse-column matrices are generated as follows:

```
cscEV = characteristicMatrix(EV)
cscFV = characteristicMatrix(FV)
cscCV = characteristicMatrix(CV)
```

The characteristic matrix `cscFV` for the incidence `FV` of the complex in Figure 1.1 is given in Figure 1.2

```
julia> full(characteristicMatrix(FV))
11x12 Array{Int8,2}:
 1  1  1  1  0  0  0  0  0  0  0  0
 0  0  0  0  1  1  1  1  0  0  0  0
 0  0  0  0  0  0  0  0  1  1  1  1
 1  1  0  0  1  1  0  0  0  0  0  0
 0  0  1  1  0  0  1  1  0  0  0  0
 0  0  0  0  1  1  0  0  1  1  0  0
 0  0  0  0  0  0  1  1  0  0  1  1
 1  0  1  0  1  0  1  0  0  0  0  0
 0  1  0  1  0  1  0  1  0  0  0  0
 0  0  0  0  1  0  1  0  1  0  1  0
 0  0  0  0  0  1  0  1  0  1  0  1
```

Figure 1.2: Dense characteristic matrix `cscFV` for the incidence `FV` of the small complex in Figure 1.1

1.2.3 Chain transformations

(Co)boundary transformations

Example 3 (Unsigned matrix of ∂_2)**.** *In this case we get the Compressed Sparse Column matrix representation* $[\partial_2]$ *of the unsigned operator, i.e. with values in* $\mathbb{Z}_2 = \{0, 1\}$*, shown* dense *in Figure 1.3, by computing*

$$\partial_2 = \texttt{uboundary2(FV,EV)}.$$

The generated sparse matrix is shown dense in Figure 1.3. Just notice that in Example 1 there are eleven 2-cells and twenty 1-cells, and that such ordered bases of 2-chains and 1-chains, respectively, are orderly associated to the rows and to the columns of $[\partial_2]$

Affine transformations

Geometric embedding The `V` array, of Julia type `Array{Float64,2}`, contains by columns the coordinates of position vectors of d-points associated to *vertices* (0-cells) of the cellular complex $X \subseteq \mathbb{R}^d$. Hence `V` specifies the geometric embedding of X.

```
julia> full(uboundary2(FV,EV))
1120 Array{Int8,2}:
 1  1  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  1  1  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  1  0  0  0  0  1  1  0  0  0  0  0  0  0  0
 1  0  1  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0
 0  1  0  1  0  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0
 0  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  1  1  0  0
 0  0  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  1  1
 0  0  0  0  0  0  1  0  1  0  0  0  1  0  1  0  0  0  0  0
 0  0  0  0  0  0  0  1  0  1  0  0  0  1  0  1  0  0  0  0
 0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  1  0  1  0
 0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  1  0  1
```

Figure 1.3: Dense unsigned $[\partial_2]$ matrix for the small complex in Figure 1.1

1.2.4 Data transformations

1.3 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.

1.4 Literate programming

This package has been developed 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.

1.5 Julia

Ths implementation has been made in 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].

Chapter 2

The arrangement algorithm

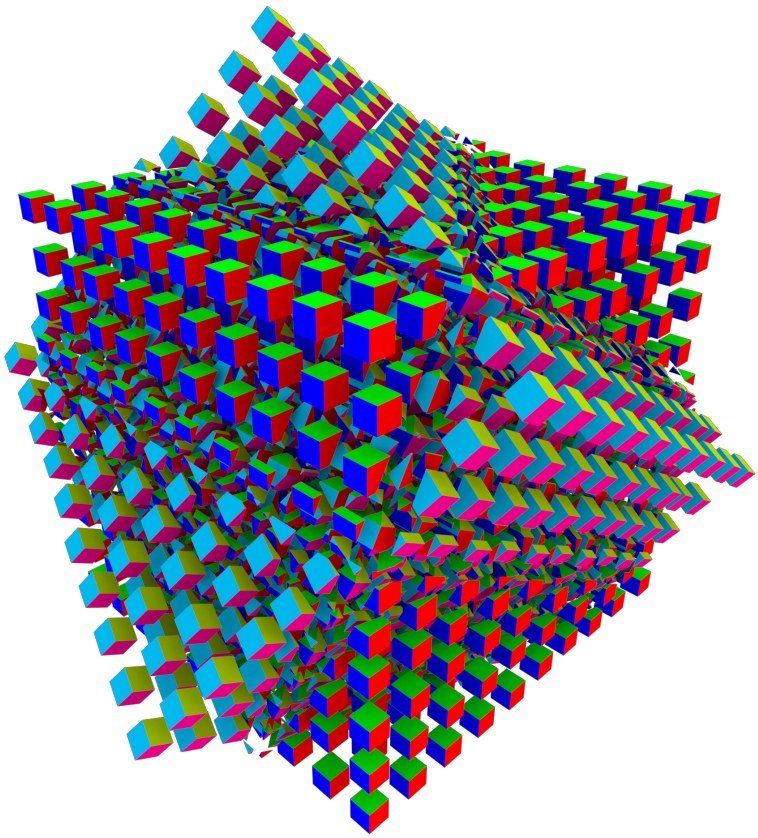


Figure 2.1: Arrangement of 2000($= 2 \times 10 \times 10 \times 10$) cubes

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.

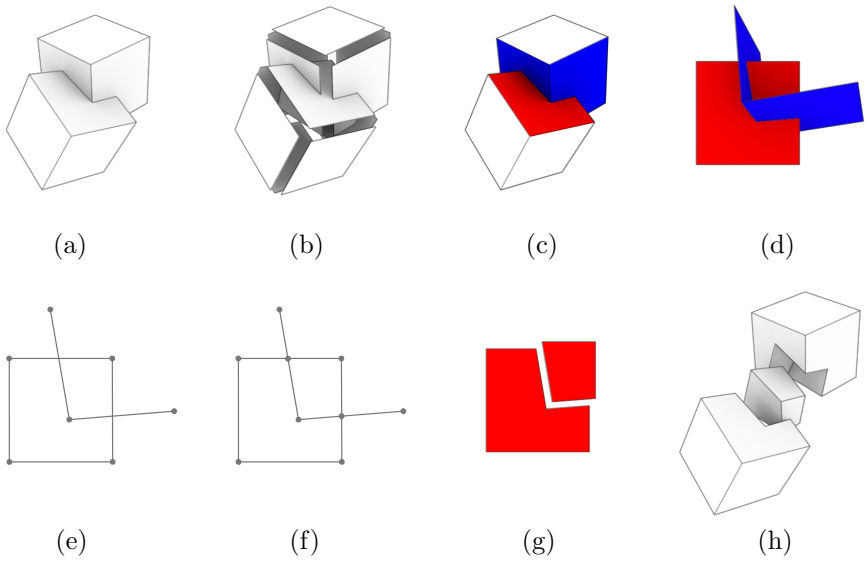


Figure 2.2: Algorithm overview

We have in input two cellular complexes [fig. 2.2, a], given as 2-skeletons, which are the sets of 2-cells [fig. 2.2, b, exploded]. Once we merged the skeletons [ref. 9.4], 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. 2.2, 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. 2.2, 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. 2.2, e], and we fragment each 1-cell with every other cell obtaining a valid 1-skeleton [fig. 2.2, 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. 2.2, 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. 2.2, h, exploded].

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

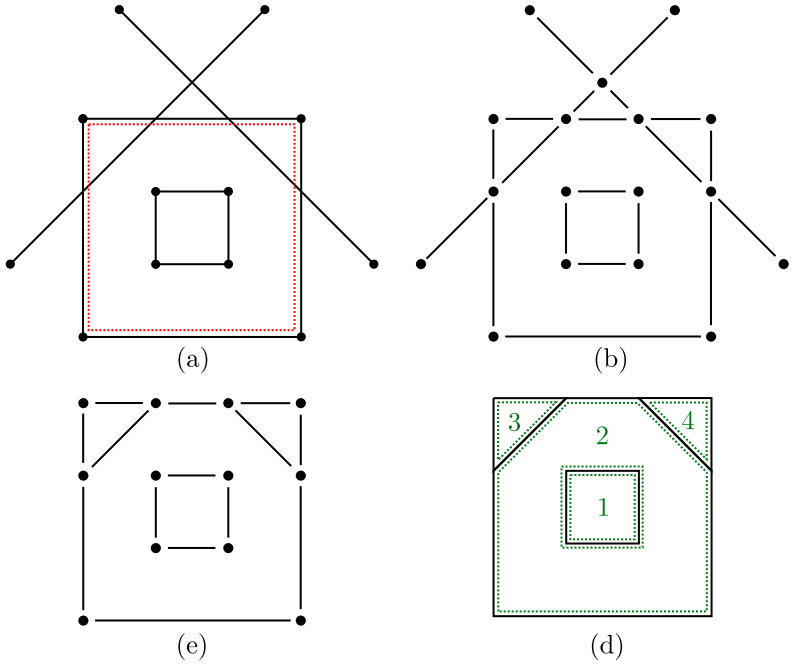


Figure 2.3: Planar arrangement overview

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. 2.3, 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. 2.3, 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. 5.6.1]. The result of this edge pruning outputs a 1-skeleton [fig. 2.3, c, exploded].

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

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. 2.3, d, 2-cells numbered in green; please note that cell 2 has cell 1 as an hole].

2.3 Implementation

Our implementation of this algorithm does not go over $d = 3$. It has been split in multiple chapters in this book:

- Spatial Arrangement [ch. 4]: It is the implementation of the "1-cells in $\mathbb{E}^2$" base case.
- Planar Arrangement [ch. 5]: This treats the $d = 3$ case.
- Dimension travel [ch. 6]: Here are contained the utilities to travel from a dimension to another.
- Minimal cycles computation [ch. 7]: This is dedicated to the implementation of the ALGORITHM 1 described by Paoluzzi et Al. [12].

[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.

Part II

Implementation

Chapter 3

Module overview

3.1 Data transforms and boundary operators

In this section we provide some Julia functions to compute (boundary) operator representations starting from other data representations, i.e., generally speaking, starting from one/two incidence relations.

Sparse characteristic matrices M_p The generic function `characteristicMatrix` produces a compressed sparse column representation of a single incidence relation, i.e. the associated characteristic matrix M_p In this code we use $p = 2$ and `FV` as parameter name, but it may refer to any other incidence relation as actual value. The function employs a COO (coordinate) method to build the sparse matrix, i.e. starts from a triple of arrays for `I,J,V` values.

⟨ Computation of sparse characteristic matrices M_p 2 ⟩ ≡

```
# Characteristic matrix $M_2$, i.e. M(FV)
function characteristicMatrix(FV)
   I,J,V = Int64[],Int64[],Int8[]
   for f=1:length(FV)
      for k in FV[f]
         push!(I,f)
         push!(J,k)
         push!(V,1)
      end
   end
   M_2 = sparse(I,J,V)
   return M_2
end
```

◇

Macro referenced in 9.

Hence we get the sparse matrix `cscEV`, shown in Figure 1.2, by

```
cscEV = characteristicMatrix(EV)
```

Boundary matrix $[\partial_1] = [\delta_0]^\top$ It is worth noting that the matrix $M_1 =$ `cscEV` previously computed is very similar to the operator matrix $[\delta_0]$, but with elements in $\{-1, 0, 1\}$ instead than in $\{0, 1\}$. The computation is than easily done, by using the fact that the 1-cell is written as oriented 0-chains $\eta = \nu_k - \nu_h$, with $k > h$, and hence in coordinates we have `spboundary1`$[k, e] = 1$ and `spboundary1`$[k, e] = -1$.

⟨ Computation of (signed) sparse boundary $C_1 \to C_0$ 3 ⟩ ≡

```
# Computation of sparse boundary $C_1 \to C_0$
function boundary1(EV)
   spboundary1 = characteristicMatrix(EV)'
   for e = 1:length(EV)
      spboundary1[EV[e][1],e] = -1
   end
   return spboundary1
end
```

◇

Macro referenced in 9.

In this case we get the compressed sparse column matrix representation ∂_1, shown dense in Figure 3.1, by computing

$$\partial_1 = \texttt{boundary1(EV)}$$

```
julia> julia> full(boundary1(EV))
1220 Array{Int8,2}:
 -1   0   0   0   0   0  -1   0   0   0   0   0  -1   0   0   0   0   0   0   0
  1   0   0   0   0   0   0  -1   0   0   0   0   0  -1   0   0   0   0   0   0
  0  -1   0   0   0   0   1   0   0   0   0   0   0   0  -1   0   0   0   0   0
  0   1   0   0   0   0   0   1   0   0   0   0   0   0   0  -1   0   0   0   0
  0   0  -1   0   0   0   0   0  -1   0   0   0   1   0   0   0  -1   0   0   0
  0   0   1   0   0   0   0   0   0  -1   0   0   0   1   0   0   0  -1   0   0
  0   0   0  -1   0   0   0   0   1   0   0   0   0   0   1   0   0   0  -1   0
  0   0   0   1   0   0   0   0   0   1   0   0   0   0   0   1   0   0   0  -1
  0   0   0   0  -1   0   0   0   0   0  -1   0   0   0   0   0   1   0   0   0
  0   0   0   0   1   0   0   0   0   0   0  -1   0   0   0   0   0   1   0   0
  0   0   0   0   0  -1   0   0   0   0   1   0   0   0   0   0   0   0   1   0
  0   0   0   0   0   1   0   0   0   0   0   1   0   0   0   0   0   0   0   1
```

Figure 3.1: Dense signed $[\partial_1]$ matrix ∂_1 for the small complex in Figure 1.1

Boundary matrix $[\partial_2] = [\delta_1]^\top$ The matrix of the unsigned operator ∂_2 is computed here, according to the method introduced in []. In particular, the two sparse incidence matrices `cscFV` and `cscEV` are first computed, and the product matrix in `temp` codifies, for any pair (i, j) of indices the number of vertices shared between the i-th 2-cell and the j-th 1-cell; when this number is equal to 2, the j-th 1-cell belongs to the boundary of the i-th 2-cell, so providing a triple

$(i, j, 1)$ to be inserted in the COO representation of `sp_uboundary2`, and finally in its CSC representation.

⟨ Computation of (unsigned) sparse boundary $C_2 \to C_1$ 4 ⟩ ≡

```
# Computation of sparse uboundary2
function uboundary2(FV,EV)
   cscFV = characteristicMatrix(FV)
   cscEV = characteristicMatrix(EV)
   temp = cscFV * cscEV'
   I,J,V = Int64[],Int64[],Int8[]
   for j=1:size(temp,2)
      for i=1:size(temp,1)
         if temp[i,j] == 2
            push!(I,i)
            push!(J,j)
            push!(V,1)
         end
      end
   end
   sp_uboundary2 = sparse(I,J,V)
   return sp_uboundary2
end
```

◇

Macro referenced in 9.

Signed boundary matrix $[\partial_2] = [\delta_1]^\top$ Here we build a possible value for $[\partial_2] \in \{-1, 0, 1\}_n^m$. By rows, it describes each elementary 2-chain (2-cell) as a signed 1-cycle (1-chain without boundary). Since no standard orientation exists for p-cycles in d-space ($p < d$), *by convention* we compute each 1-cycle oriented as its first 1-cell, i.e. going in the direction from the second to the first 0-cell of it.

The construction algorithm may be split into two phases: the initialization and the loop indexed on faces. In turn, the faces loop calls the function `columninfo(col)`, deputed to store a quadruple of data providing a data structure to set properly (in sign) each 1-cell of a 1-cycle.

Initialization

⟨ Initialization of the (signed) sparse boundary 5 ⟩ ≡

```
# Initialization
function boundary2(FV,EV)
    sp_u_boundary2 = uboundary2(FV,EV)
    larEV = characteristicMatrix(EV)
    # unsigned incidence relation
    FE = [findn(sp_u_boundary2[f,:]) for f=1:size(sp_u_boundary2,1) ]
    I,J,V = Int64[],Int64[],Int8[]
    vedges = [findn(larEV[:,v]) for v=1:size(larEV,2)]
```

◇

Macro referenced in 8.

Local storage A local function `columninfo` is used to store the needed data structure into a temporary array `infos`

⟨ Local temporary storage function 6 ⟩ ≡

```
# Local storage
function columninfo(col)
    infos[1,col] = 1
    infos[2,col] = next
    infos[3,col] = EV[next][1]
    infos[4,col] = EV[next][2]
    vpivot = infos[4,col]
end
```

◇

Macro referenced in 8.

Loop on faces

⟨ Loop on faces to construct the (signed) sparse boundary2 7 ⟩ ≡

```
# Loop on faces
for f=1:length(FE)
    fedges = Set(FE[f])
    next = pop!(fedges)
    col = 1
    vpivot = infos[4,col]
    infos = zeros(Int64,(4,length(FE[f])))
    while fedges != Set()
        nextedge = intersect(fedges, Set(vedges[vpivot]))
        fedges = setdiff(fedges,nextedge)
        next = pop!(nextedge)
        col += 1
        vpivot = columninfo(col)
        if vpivot == infos[4,col-1]
            infos[3,col],infos[4,col] = infos[4,col],infos[3,col]
            infos[1,col] = -1
            vpivot = infos[4,col]
        end
    end
    for j=1:size(infos,2)
        push!(I, f)
        push!(J, infos[2,j])
        push!(V, infos[1,j])
    end
end
```

◇

Macro referenced in 8.

The assembly of the whole function `boundary2(FV,EV)` is given below.

Assembly signed boundary construction

⟨ Computation of (signed) sparse boundary $C_2 \to C_1$ 8 ⟩ ≡

```
⟨ Local temporary storage function 6 ⟩
⟨ Initialization of the (signed) sparse boundary 5 ⟩
    ⟨ Loop on faces to construct the (signed) sparse boundary2 7 ⟩
    spboundary2 = sparse(I,J,V)
    return spboundary2
end
```

◇

Macro referenced in 9.

"lib/jl/LARLIB.jl" 9 ≡

```
module LARLIB

    ⟨ LAR imports 20, ... ⟩
    ⟨ LAR types 10 ⟩
    ⟨ Computation of sparse characteristic matrices $M_p$ 2 ⟩
    ⟨ Computation of (signed) sparse boundary $C_1 \to C_0$ 3 ⟩
    ⟨ Computation of (unsigned) sparse boundary $C_2 \to C_1$ 4 ⟩
    ⟨ Computation of (signed) sparse boundary $C_2 \to C_1$ 8 ⟩

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

end
```

◇

3.2 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], 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 10 ⟩ ≡

```
const Verts = Array{Float64, 2}
const Cells = SparseMatrixCSC{Int8, Int}
const Cell = SparseVector{Int8, Int}
const LarCells = Array{Array{Int, 1}, 1}
```

◇

Macro referenced in 9.

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

3.2.1 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 5.3.2, 6.4, 7.2.2 and 9.8.

3.3 Notes on variables names

Here a list of some often used variable names.

`V`: Bi-dimensional array (`Verts`) 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 (`Cells`) 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.

3.4 Tests and examples

There are several unit tests throughout the implementation. They are inside the `test` directory and can be run at once by executing `test/runtests.jl`

```
"test/jl/runtests.jl" 11 ≡
    using LARLIB

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

Also general examples of some main functionalities are provided. They can be found into `examples/general_examples.jl`

[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 $2^5 3 \approx 1.11 \times 10^{-16}$.

"examples/jl/general_examples.jl" 12 ≡

```
using LARLIB

⟨ planar_arrangement general examples 72 ⟩
⟨ spatial_arrangement general examples 25 ⟩
◇
```

Chapter 4

Spatial Arrangement

4.1 Overview

Refer to the introduction for algorithm explanation [ref. 2].

"lib/jl/spatial_arrangement.jl" 13 ≡

⟨ spatial_arrangement support functions 16, ... ⟩

```
function spatial_arrangement(V::Verts, EV::Cells, FE::Cells; multiproc=false)
    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)

    if (multiproc == true)
        ⟨ Run parallel spatial_arrangement 15 ⟩
    else
        ⟨ Run sequential spatial_arrangement 14 ⟩
    end

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

    ⟨ Create 3-cells 24 ⟩

    return rV, rEV, rFE, rCF
end
```

◇

4.1.1 Tests and examples

Unit tests has yet to be written for this chapter, but general examples are available at the end of the chapter [ref. 4.6]

4.2 Parallel approach

If the user does not require parallelization (using the `multiproc` argument) the sequential version of the implementation starts. This fragments each face sequentially.

⟨ Run sequential spatial_arrangement 14 ⟩ ≡

```
for sigma in 1:fs_num
    print(sigma, "/", fs_num, "\r")
    nV, nEV, nFE = frag_face(V, EV, FE, sp_idx, sigma)
    rV, rEV, rFE = skel_merge(rV, rEV, rFE, nV, nEV, nFE)
end
```

◇

Macro referenced in 13.

If instead, parallelization is required, we build the in and out channels, we feed the data to the input channel, we dispatch tasks to the workers and then read the data from the output channel. We also defined a function called `frag_face_channel` to call the `frag_face` function from every worker with the data inside the output channel.

⟨ Run parallel spatial_arrangement 15 ⟩ ≡

```
in_chan = RemoteChannel(()->Channel{Int64}(0))
out_chan = RemoteChannel(()->Channel{Tuple}(0))

@schedule begin
    for sigma in 1:fs_num
        put!(in_chan, sigma)
    end
    for p in workers()
        put!(in_chan, -1)
    end
end

for p in workers()
    @async Base.remote_do(
        frag_face_channel, p, in_chan, out_chan, V, EV, FE, sp_idx)
end

for sigma in 1:fs_num
    rV, rEV, rFE = skel_merge(rV, rEV, rFE, take!(out_chan)...)
end
```

◇

Macro referenced in 13.

⟨ spatial_arrangement support functions 16 ⟩ ≡

```
function frag_face_channel(in_chan, out_chan, V, EV, FE, sp_idx)
    run_loop = true
    while run_loop
        sigma = take!(in_chan)
        if sigma != -1
            put!(out_chan, frag_face(V, EV, FE, sp_idx, sigma))
        else
            run_loop = false
        end
    end
end
```

◇

Macro defined by 16, 17, 19.
Macro referenced in 13.

4.3 Face fragmentation

To fragment a σ 2-cell, we flat it out on the $x_3 = 0$ plane, we launch the planar arrangement and then bring the fragmented σ to its original place.

⟨ spatial_arrangement support functions 17 ⟩ ≡

```
function frag_face(V, EV, FE, sp_idx, sigma)
    vs_num = size(V, 1)

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

    ⟨ Sigma flattening 18 ⟩

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

    if nV == nothing
        return [], spzeros(Int8, 0,0), spzeros(Int8, 0,0)
    end

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

◇

Macro defined by 16, 17, 19.
Macro referenced in 13.

The flattening of σ on the $x_3 = 0$ plane is performed by building a linear transformation matrix with the `submanifold_mapping` utility [ref. 6.2], transforming the geometry and intersecting every cell in $\mathcal{I}(\sigma)$ [ref. 2.1] with the $x_3 = 0$ plane using `face_int` [ref. 6.4].

⟨ Sigma flattening 18 ⟩ ≡
```
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 17.

4.4 Coincident vertices merge

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

⟨ spatial_arrangement support functions 19 ⟩ ≡
```
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 21 ⟩
    ⟨ Merge edges 22 ⟩
    ⟨ Merge faces 23 ⟩

    return nV, nEV, nFE
end
```
◇
Macro defined by 16, 17, 19.
Macro referenced in 13.

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 20 ⟩ ≡
```
using NearestNeighbors
```
◇

Macro defined by 20, 30, 40, 77, 92.
Macro referenced in 9.

⟨ Find coincident vertices 21 ⟩ ≡

```
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 19, 41.

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. 5.4].

⟨ Merge edges 22 ⟩ ≡

```
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 19, 41.

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 23 ⟩ ≡

```
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 19.

4.5 3-cells creation

⟨ Create 3-cells 24 ⟩ ≡
⟨ Compute connected components ? ⟩
⟨ Compute containment tree ? ⟩
⟨ Cell union ? ⟩

```
rCF = minimal_3cycles(rV, rEV, rFE)
```
◇
Macro referenced in 13.

4.6 General examples

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 general examples 25 ⟩ ≡

```
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 12.

On the next pages the results are visualized.

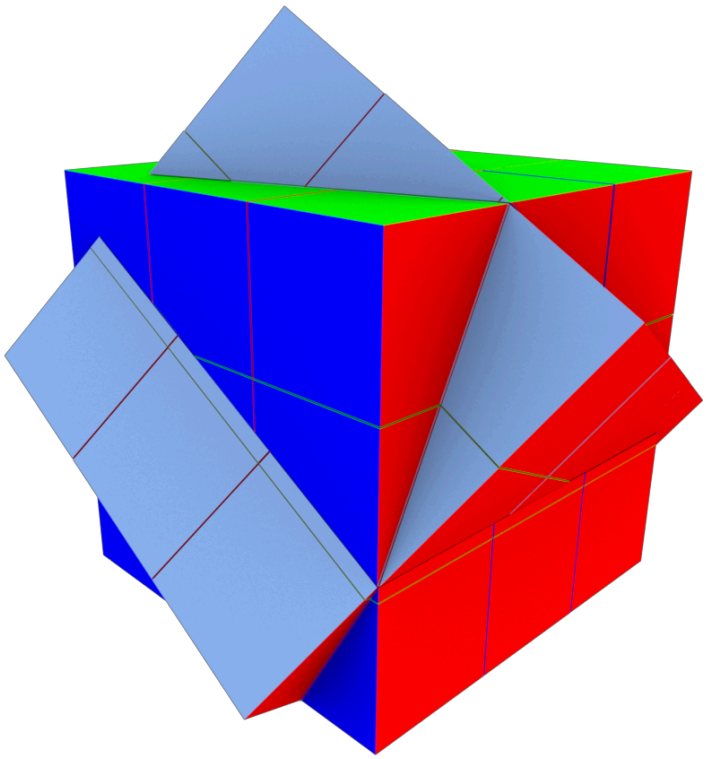


Figure 4.1: Input

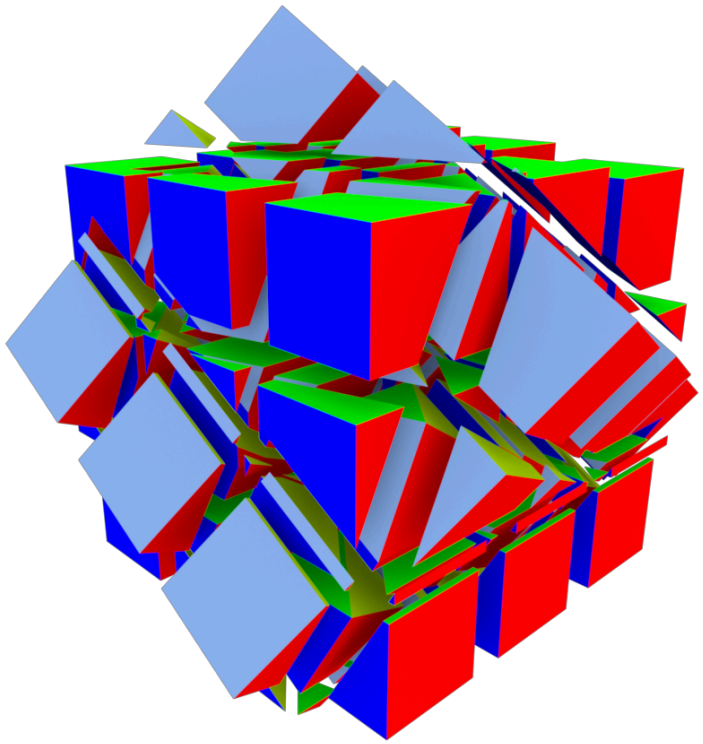


Figure 4.2: Output (Exploded)

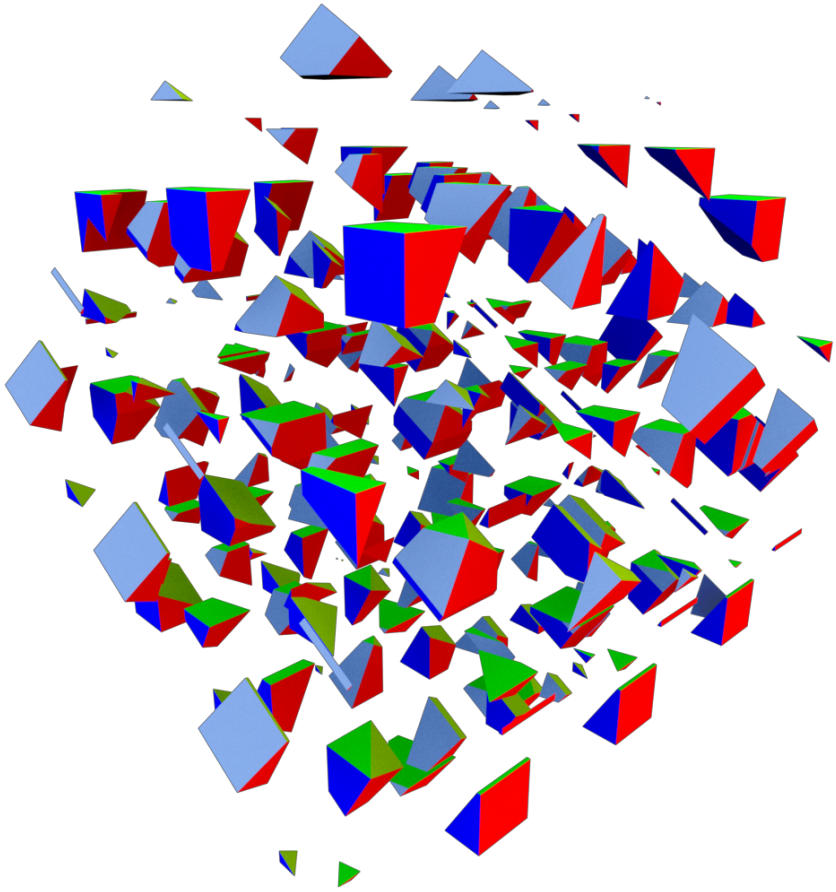


Figure 4.3: Output (More exploded)

Chapter 5

Planar Arrangement

5.1 Overview

Refer to the introduction for algorithm explanation [ref. 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" 26 ≡
    ⟨ planar_arrangement support functions 31, ... ⟩

    function planar_arrangement(V::Verts, EV::Cells, sigma::Cell=spzeros(Int8, 0); multiproc=false)
        edgenum = size(EV, 1)
        ⟨ planar_arrangement local variables 27, ... ⟩

        if (multiproc == true)
            ⟨ Run parallel planar_arrangement 32 ⟩
        else
            ⟨ Run sequential planar_arrangement 29 ⟩
        end

        V, EV = rV, rEV

        ⟨ Merge coincident vertices 43 ⟩
        ⟨ Delete edges outside sigma area 45 ⟩
        ⟨ Find maximal biconnected components 53 ⟩
        ⟨ Filter biconnected components 54 ⟩
        ⟨ Create faces 56 ⟩

        V, EV, FE, edge_map
    end
    ◇
```

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

⟨ planar_arrangement local variables 27 ⟩ ≡

```
edge_map = Array{Array{Int, 1}, 1}(edgenum)
```
◇
Macro defined by 27, 38.
Macro referenced in 26.

5.1.1 Tests and examples

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

"test/jl/planar_arrangement.jl" 28 ≡
```
using Base.Test
using LARLIB

⟨ planar_arrangement support functions tests 39, ... ⟩
```
◇

Examples to show the functionalities implemented in this chapter are defined at the end of it [ref. 5.8].

5.2 Parallel approach

The sequential implementation of the algorithm is straightforward: fragment each edge in order and update the `edge_map` at each step.

⟨ Run sequential planar_arrangement 29 ⟩ ≡
```
for i in 1:edgenum
    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)
    rV, rEV = skel_merge(rV, rEV, v, ev)
end
```
◇
Macro referenced in 26.

The parallel version is very similar to the one presented for the Spatial Arrangement [ref. 4]. The only difference here is that we need to recompose the data in order, so we use a `SortedDict` to store the data from of the out channel. The `SortedDict` data structure is provided by the `DataStructures` Julia package.

⟨ LAR imports 30 ⟩ ≡
```
using DataStructures
```
◇
Macro defined by 20, 30, 40, 77, 92.
Macro referenced in 9.

The `frag_edge_channel` function is the job distributed to the workers that fetch data from the in channel and feeds the results of `frag_edge` to the out channel.

⟨ planar_arrangement support functions 31 ⟩ ≡

```
function frag_edge_channel(in_chan, out_chan, V, EV)
    run_loop = true
    while run_loop
        edgenum = take!(in_chan)
        if edgenum != -1
            put!(out_chan, (edgenum, frag_edge(V, EV, edgenum)))
        else
            run_loop = false
        end
    end
end
```

◇

Macro defined by 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

The definition of the channels, the set up of the workers, the dispatch, the output data fetch and the reconstruction of the fragmented `V` and `EV` (with relative `edge_map`) is implemented here:

⟨ Run parallel planar_arrangement 32 ⟩ ≡

```
in_chan = RemoteChannel(()->Channel{Int64}(0))
out_chan = RemoteChannel(()->Channel{Tuple}(0))

ordered_dict = SortedDict{Int64,Tuple}()

@schedule begin
    for i in 1:edgenum
        put!(in_chan,i)
    end
    for p in workers()
        put!(in_chan,-1)
    end
end

for p in workers()
    @async Base.remote_do(frag_edge_channel, p, in_chan, out_chan, V, EV)
end

for i in 1:edgenum
    frag_done_job = take!(out_chan)
    ordered_dict[frag_done_job[1]] = frag_done_job[2]
end

for (dkey, dval) in ordered_dict
    i = dkey
    v, ev = dval
    newedges_nums = map(x->x+finalcells_num, collect(1:size(ev, 1)))

    edge_map[i] = newedges_nums
```

```
        finalcells_num += size(ev, 1)

        rV, rEV = skel_merge(rV, rEV, v, ev)
    end
    ◇
```

Macro referenced in 26.

5.3 Edge fragmentation

5.3.1 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. 5.3.2] The intersection points are then sorted along the edge to fragment, and correct fragments (which are edges themselves) are computed.

⟨ planar_arrangement support functions 33 ⟩ ≡

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

        alphas[0.0], alphas[1.0] = [1, 2]

        alphas_keys = sort(collect(keys(alphas)))
        edge_num = length(alphas_keys)-1
        verts_num = size(verts, 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

        verts, ev
    end
```

◇

Macro defined by 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

5.3.2 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. 3.2.1].

⟨ planar_arrangement support functions 34 ⟩ ≡

```
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 35 ⟩

    if colinear
        ⟨ Handle colinear edges 36 ⟩
    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 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

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 35 ⟩ ≡

```
    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 34.

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 36 ⟩ ≡

```
    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 34.

5.3.3 Implementation

When we need to fragment an edge we just use the `frag_edge` function [ref. 5.3.1] 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 37 ⟩ ≡

```
    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)

    rV, rEV = skel_merge(rV, rEV, v, ev)
    ◇
```

Macro never referenced.

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

⟨ planar_arrangement local variables 38 ⟩ ≡

```
    rV = zeros(0, 2)
    rEV = spzeros(Int8, 0, 0)
    finalcells_num = 0
    ◇
```

Macro defined by 27, 38.
Macro referenced in 26.

5.3.4 Tests

⟨ planar_arrangement support functions tests 39 ⟩ ≡

```
    @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
```

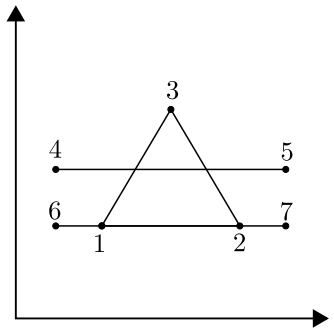


Figure 5.1: The bunch of edges used for the tests.

```
    ]))

    @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 == [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) == [1 0 0 0 0 1;
                            0 0 0 0 1 1;
                            0 1 0 0 1 0]
    end
end
```
◇

Macro defined by 39, 44, 55, 67.
Macro referenced in 28.

5.4 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. 4.4]

⟨ LAR imports 40 ⟩ ≡
```
using NearestNeighbors
```
◇

Macro defined by 20, 30, 40, 77, 92.
Macro referenced in 9.

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 41 ⟩ ≡
```
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 21 ⟩
    ⟨ Merge edges 22 ⟩
    ⟨ Update edge_map after vertex merging 42 ⟩

    return nV, nEV
end
```
◇

Macro defined by 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

The last step is to update `edge_map`. We update the indices using the data structures built in the ⟨ Merge edges ⟩ macro [ref. 4.4].

⟨ Update edge_map after vertex merging 42 ⟩ ≡
```
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 41.

5.4.1 Implementation

We simply call `merge_vertices`.

⟨ Merge coincident vertices 43 ⟩ ≡
```
V, EV = merge_vertices!(V, EV, edge_map)
```
◇

Macro referenced in 26.

5.4.2 Tests

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

⟨ planar_arrangement support functions tests 44 ⟩ ≡

```
@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 39, 44, 55, 67.
Macro referenced in 28.

5.5 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. 9.11]. Finally, once we have marked the edges to delete, we delete them [ref. 9.5] and update the `edge_map` (refer to next macro for this).

⟨ Delete edges outside sigma area 45 ⟩ ≡

```
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 46 ⟩

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

Macro referenced in 26.

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 46 ⟩ ≡

```
    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 45, 54.

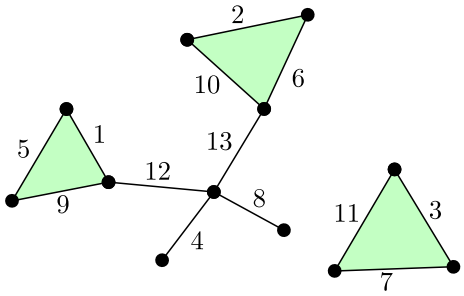


Figure 5.2: An example graph where the maximal biconnected components are highlighted in green and the edges are numbered. We have here three components formed by the sets of edges {1,5,9}, {2,6,10} and {3,11,7}

5.6 Maximal biconnected components

5.6.1 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 47 ⟩ ≡
```
function biconnected_components(EV::Cells)
    ⟨ biconnected_components local variables 48 ⟩
    ⟨ DFS utilities 49 ⟩
    ⟨ Depth first visit 50 ⟩
    bicon_comps
end
```
◇
Macro defined by 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

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 48 ⟩ ≡
```
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 47.

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 49 ⟩ ≡

```
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 47.

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 50 ⟩ ≡

```
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 52 ⟩
            else
                if ps[end][3] == ps[end-1][2]
                    ⟨ Form biconnected component 51 ⟩
                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 47.

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 51 ⟩ ≡

```
    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 50.

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 52 ⟩ ≡
```
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 50.

5.6.2 Implementation

Like for the vertices merge we simply call the freshly implemented `biconnected_components` function [ref. 5.6.1]. If no biconnected components are found, the procedure will stop and return nothing.

⟨ Find maximal biconnected components 53 ⟩ ≡
```
bicon_comps = biconnected_components(EV)

if isempty(bicon_comps)
    println("No biconnected components found.")
    return (nothing, nothing, nothing)
end
```
◇
Macro referenced in 26.

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 5.5).

⟨ Filter biconnected components 54 ⟩ ≡
```
edges = sort(union(bicon_comps...))
todel = sort(setdiff(collect(1:size(EV,1)), edges))

⟨ Update edge_map 46 ⟩

V, EV = delete_edges(todel, V, EV)
```
◇
Macro referenced in 26.

5.6.3 Tests

The graph built here is the one of figure 5.2.

⟨ planar_arrangement support functions tests 55 ⟩ ≡

```
@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 39, 44, 55, 67.
Macro referenced in 28.

5.7 Faces creation

5.7.1 Implementation

⟨ Create faces 56 ⟩ ≡

```
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 58 ⟩
⟨ Transitive reduction 61 ⟩
⟨ Cell merging 63 ⟩

◇

Macro referenced in 26.

5.7.2 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 9.3,

⟨ planar_arrangement support functions 57 ⟩ ≡

```
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 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

5.7.3 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. 5.7.6].

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. 9.2].

⟨ Containment test 58 ⟩ ≡

```
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 56.

⟨ planar_arrangement support functions 59 ⟩ ≡

```
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 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

To check if a point is really inside a face we use the `point_in_face` utility [ref. 9.11]

⟨ planar_arrangement support functions 60 ⟩ ≡
```
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 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

5.7.4 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 61 ⟩ ≡
```
transitive_reduction!(containment_graph)
```
◇

Macro referenced in 56.

⟨ planar_arrangement support functions 62 ⟩ ≡

```
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 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

5.7.5 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 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 63 ⟩ ≡

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

Macro referenced in 56.

⟨ planar_arrangement support functions 64 ⟩ ≡

```
function cell_merging(n, containment_graph, V, EVs, boundaries, shells, shell_bboxes)
    ⟨ Cell merging support functions 65 ⟩

    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
```

[1] Please note that that the `bboxes` is not part of the bounding box utilities [ref. 9.2] but it is defined in the next paragraph

```
                    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 66 ⟩
        return EV, FE
    end
    ◇
```

Macro defined by 31, 33, 34, 41, 47, 57, 59, 60, 62, 64.
Macro referenced in 26.

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

⟨ Cell merging support functions 65 ⟩ ≡

```
    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 64.

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 66 ⟩ ≡

```
    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 64.

5.7.6 Tests

⟨ planar_arrangement support functions tests 67 ⟩ ≡

```
    @testset "Face creation" begin
        ⟨ Face creation tests 68, ... ⟩
    end
    ◇
```

Macro defined by 39, 44, 55, 67.
Macro referenced in 28.

External cell individuation

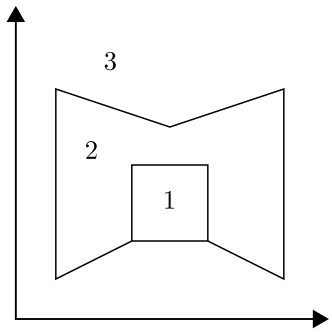


Figure 5.3: This biconnected component has three faces. The external one is the number 3. This is a particularly difficult case because the most "external" vertices of face 2 are in common with the external cell.

⟨ Face creation tests 68 ⟩ ≡

```
    @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 68, 69, 70, 71.
Macro referenced in 67.

Containment test

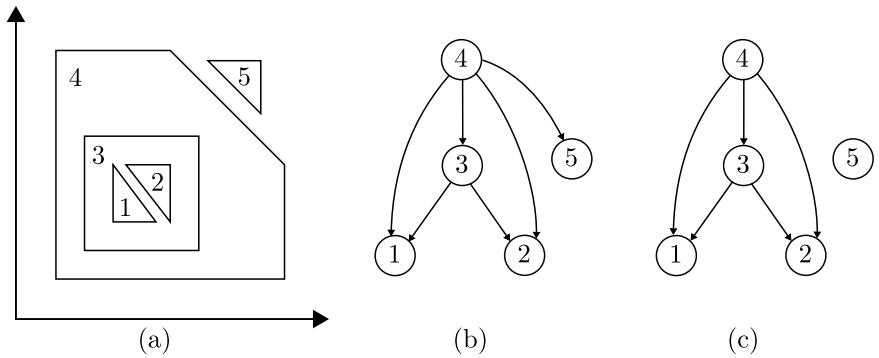


Figure 5.4: (a) is our test case. The numbers identify the connected components. (b) is the containment graph built using only the `pre_containment_test` function. The arc (4,5) is in there because the bounding box of the component no. 5 is completely contained in the bounding box of no. 4. (c) shows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

⟨ Face creation tests 69 ⟩ ≡

```
    @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 -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]
    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 68, 69, 70, 71.
Macro referenced in 67.

Transitive reduction

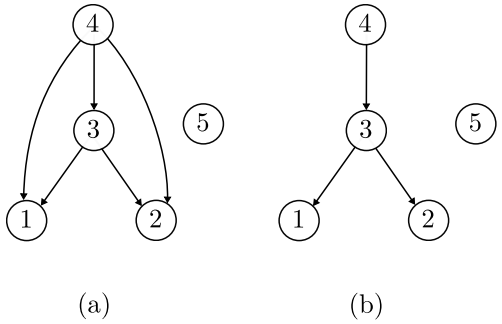


Figure 5.5: Before (a) and after (b) transitive reduction performed on the graph of the previous test set.

⟨ Face creation tests 70 ⟩ ≡

```
@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 68, 69, 70, 71.
Macro referenced in 67.

Cell merging

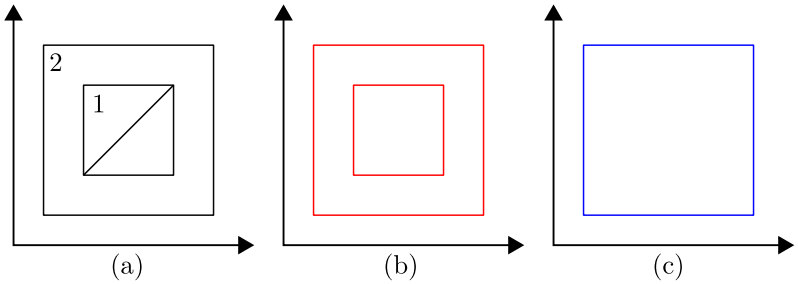


Figure 5.6: Here we have two biconnected components, one inside the other (a). If we don't perform cell merging, the boundary of the arranged set will be the red one (b), which is incorrect. The correct boundary is the blue one (c).

⟨ Face creation tests 71 ⟩ ≡

```
@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 68, 69, 70, 71.
Macro referenced in 67.

5.8 General examples

⟨ planar_arrangement general examples 72 ⟩ ≡

```
    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 12.

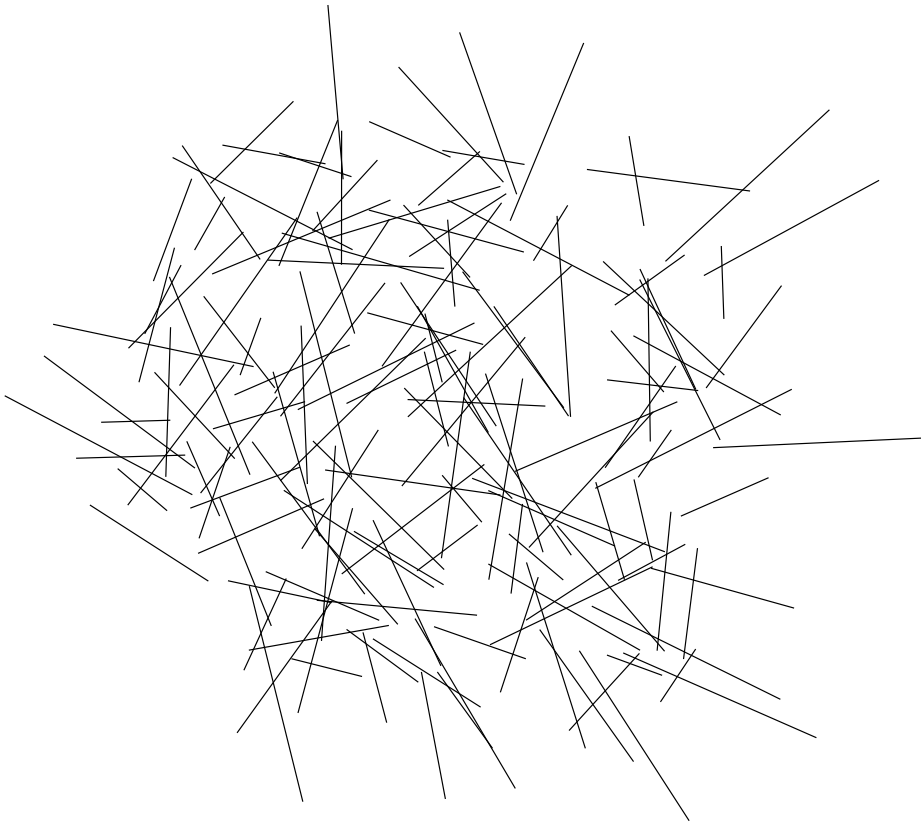


Figure 5.7: Input

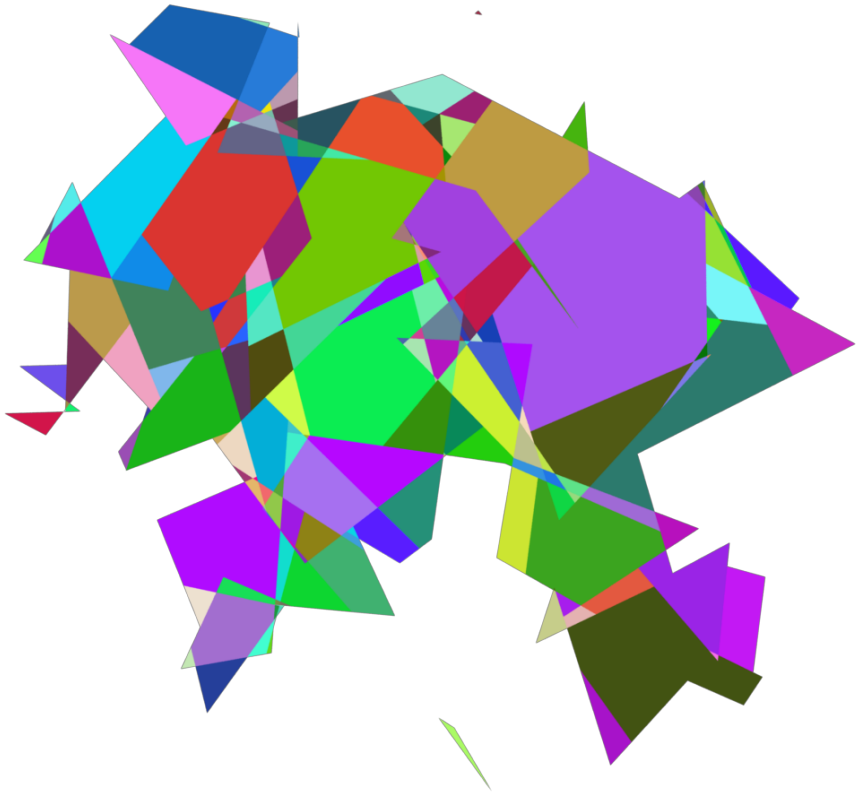


Figure 5.8: Output

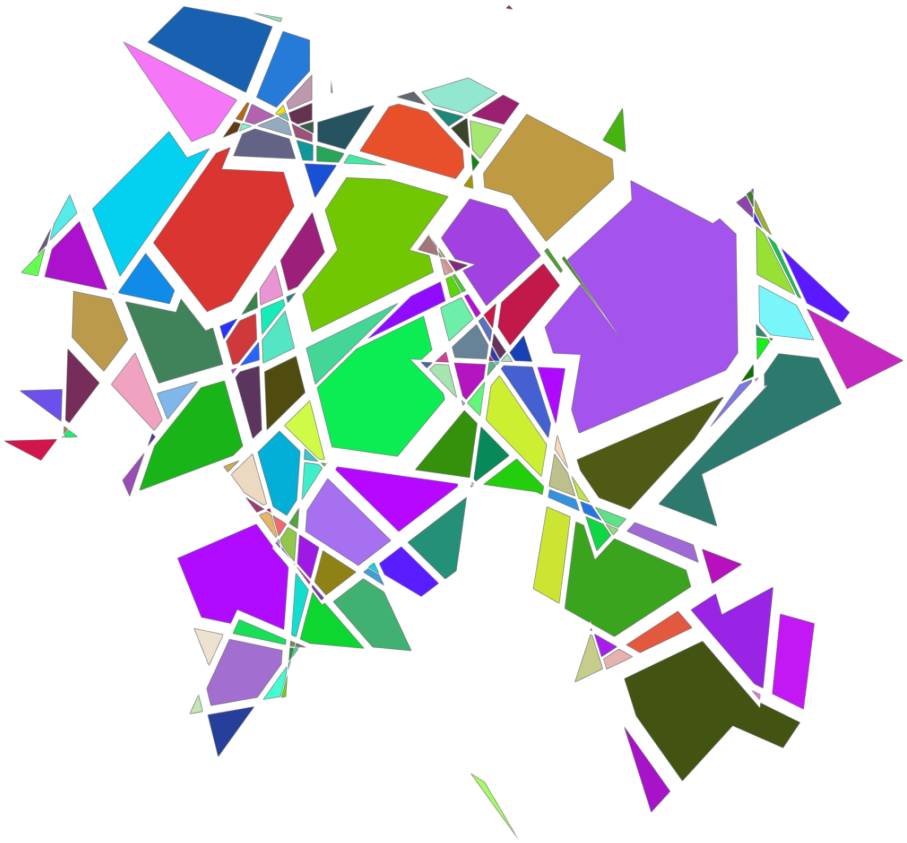


Figure 5.9: Output (Exploded)

Chapter 6

Dimension travel

6.1 Overview

This chapter is dedicated to the utilities designed to travel from $\mathbb{E}^3$ to $\mathbb{E}^2$. Refer to the introduction for algorithm explanation [ref. 2].

"lib/jl/dimension_travel.jl" 73 ≡
```
⟨ Dimension travel functions 75, ... ⟩
```
◇

6.1.1 Tests

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

"test/jl/dimension_travel.jl" 74 ≡
```
using Base.Test
using LARLIB

⟨ Tests 76 ⟩
```
◇

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 75 ⟩ ≡
```
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 75, 78, 81.
Macro referenced in 73.

6.2.1 Tests

⟨ Tests 76 ⟩ ≡
```
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 74.

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 77 ⟩ ≡
```
using IntervalTrees
```
◇

Macro defined by 20, 30, 40, 77, 92.
Macro referenced in 9.

⟨ Dimension travel functions 78 ⟩ ≡
```
function spatial_index(V::Verts, EV::Cells, FE::Cells)
    d = 3
    faces_num = size(FE, 1)
    ⟨ Build the d-IntervalTrees 79 ⟩
    ⟨ Create the mapping 80 ⟩
    mapping
end
```
◇

Macro defined by 75, 78, 81.
Macro referenced in 73.

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. 9.2].

⟨ Build the d-IntervalTrees 79 ⟩ ≡
```
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 78.

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 80 ⟩ ≡
```
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 78.

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. 3.2.1].

⟨ Dimension travel functions 81 ⟩ ≡

```
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 75, 78, 81.
Macro referenced in 73.

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]. Refer also to the introduction [ref. 2]. 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" 82 ≡
    ⟨ Minimal cycles implementations 89, ... ⟩

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

        function _minimal_cycles(V::Verts, ld_bounds::Cells)
            ⟨ Function body 83 ⟩
        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 83 ⟩ ≡

```
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 86 ⟩
⟨ minimal_cycles utilities 84, ... ⟩

while (sigma = get_seed_cell()) > 0

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

    ⟨ Compute a cycle 85 ⟩
end

return d_bounds
◇
```

Macro referenced in 82.

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 84 ⟩ ≡

```
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 84, 87, 88.
Macro referenced in 83.

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 85 ⟩ ≡

```
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 83.

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 86 ⟩ ≡

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

Macro referenced in 83.

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 87 ⟩ ≡

```
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 84, 87, 88.
Macro referenced in 83.

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 88 ⟩ ≡

```
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 84, 87, 88.
Macro referenced in 83.

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 89 ⟩ ≡

```
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 89, 90.
Macro referenced in 82.

7.2.2 $d = 3$

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

⟨ Minimal cycles implementations 90 ⟩ ≡

```
    function minimal_3cycles(V::Verts, EV::Cells, FE::Cells)

        ⟨ Face angle function 91 ⟩

        EF = FE'

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

        return -FC'
    end
    ◇
```

Macro defined by 89, 90.
Macro referenced in 82.

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 91 ⟩ ≡

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

    function face_angle(e::Int, f::Int)
        if !isassigned(triangulated_faces, f)
            ⟨ Triangulate face 93 ⟩
        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 90.

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 92 ⟩ ≡

```
    using TRIANGLE
    ◇
```

Macro defined by 20, 30, 40, 77, 92.
Macro referenced in 9.

⟨ Triangulate face 93 ⟩ ≡

[3] We check the length of the cross product against a fixed error [ref. 3.2.1].

```
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 91.

Chapter 8

Cubical grids and topological products

Here we introduce a fast implementation of the general Cartesian product of cellular complexes. Both kind of operators, depending on the dimension of their input, may generate either full-dimensional (i.e. solid) output complexes or cellular complexes of dimension d embedded in Euclidean space of dimension n, with $d \leq n$.

8.1 Overview

This chapter aims to discuss the design and the implementation of the `largrid` module of the `LARLIB.jl` library, including also the Cartesian product of general cellular complexes. In particular, we show that both n-dimensional grids of (hyper)-cuboidal cells and their d-dimensional skeletons ($0 \leq d \leq n$), embedded in $\mathbb{E}^n$, may be properly and efficiently generated by assembling the cells produced by a number n of either 0- or 1-dimensional cell complexes, that in such lowest dimensions coincide with simplicial complexes.

In Section 8.3 we give the simple implementation of generation of lower-dimensional (say, either 0- or 1-dimensional) regular cellular complexes with integer coordinates. In Section 8.4 a functional decomposition of the generation of either full-dimensional cuboidal complexes in $\mathbb{E}^n$ and of their d-skeletons ($0 \leq d \leq n$) is given, showing in particular that every skeleton can be efficiently generated as a partition in cell subsets produced by the Cartesian product of a proper disposition of 0-1 complexes, according to the binary representation of a subset of the integer interval $[0, 2^n]$.

In Section 8.5 we provide a very simple and general implementation of the topological product of *two* cellular complexes of any topology. When applied to embedded linear cellular complexes (i.e. when the coordinates of 0-cells of arguments are fixed and given) the algorithm produces a Cartesian product of its two arguments. In Section 8.2.1 the exporting of the module to different

languages is provided.

The Section **??** contains the unit tests associated to the various algorithms, that are exported by the used literate environment in the proper `test` subdirectory—depending on the implementation language. In Section **??** the indexing structure of the macro sources and variables is exposed by the sake of the reader. The Appendix **??** contains some programming utilities possibly needed by the developers.

8.2 Implementation

8.2.1 Largrid exporting

We assemble top-down the `largrid.jl` file, by listing the macros it is composed of. As might be expected, the present one is the module version corresponding to the current state of the system, i.e. to a very initial state.

`"lib/jl/largrid.jl"` 94 ≡

```
#= Functions for grid generation and Cartesian product =#

using IterTools
using DataStructures

⟨ Generation of uniform 0D cellular complex 95 ⟩
⟨ Generation of uniform 1D cellular complex 96 ⟩
⟨ Generation of cellular complex of dimension d = 0|1 97 ⟩
⟨ Interface to Cartesian Product of arrays, strings, unitRanges, etc. 98 ⟩
⟨ Generation of grid vertices 100 ⟩
⟨ Transformation from multindex to address in a linear array storage 102 ⟩
⟨ Generation of grid cells 104 ⟩
⟨ Enumeration of binary ranges of given order 106 ⟩
⟨ Filtering binary ranges by order 107 ⟩
⟨ Assembling grid skeletons 109 ⟩
⟨ Multidimensional grid generation 110 ⟩
⟨ Cartesian product of two lar models 112 ⟩
◇
```

8.3 0D- and 1D-complexes

We are going to use 0- and 1-dimensional cell complexes as the basic material for several operations, including generation of simplicial and cellular grids and topological and Cartesian product of cell complexes.

8.3.1 Generation of cells

Uniform 0D complex The `grid0` second-order function generates a 0-dimensional uniform complex embedding $n + 1$ equally-spaced (at unit intervals) 0-cells within the 1D interval. It returns the cells of this 0-complex.

⟨ Generation of uniform 0D cellular complex 95 ⟩ ≡
```
function grid_0(n)
    return hcat([[i] for i in range(0,n+1)]...)
end
```
◇
Macro referenced in 94.

Uniform 1D complex A similar `grid1` function returns a uniform 1D cellular complex with n 1D `cells`.

⟨ Generation of uniform 1D cellular complex 96 ⟩ ≡
```
function grid_1(n)
    return hcat([[i,i+1] for i in range(0,n)]...)
end
```
◇
Macro referenced in 94.

Uniform 0D or 1D complex A `larGrid` function is finally given to generate the LAR representation of the cells of either a 0- or a 1-dimensional complex, depending on the value of the `d` parameter, to take values in the set $\{0, 1\}$, and providing the *order* of the output complex.

⟨ Generation of cellular complex of dimension $d = 0|1$ 97 ⟩ ≡
```
function larGrid(n)
    function larGrid1(d)
        if d==0
         return grid_0(n)
        elseif d==1
         return grid_1(n)
        end
    end
    return larGrid1
end
```
◇
Macro referenced in 94.

Cartesian Product Unary interface to Cartesian product of collections (arrays, tuples, strings, ranges, etc.). Enter the input as an array. The output is a sorted array of tuples.

⟨ Interface to Cartesian Product of arrays, strings, unitRanges, etc. 98 ⟩ ≡
```
# Cartesian product of collections in its unary argument
function cart(args)
   return sort(collect(IterTools.product(args...)))
end
```
◇
Macro referenced in 94.

8.3.2 Unit tests

`"test/jl/largrid.jl"` 99 ≡

```
using LARLIB
using Base.Test

@testset "LarGrid-0-1 Tests" begin
   @testset "grid_0" begin
      @test LARLIB.grid_0(1) == [0  1]
      @test LARLIB.grid_0(2) == [0  1  2]
      @test LARLIB.grid_0(3) == [0  1  2  3]
   end

   @testset "Grid_1 Tests" begin
      @test repr(LARLIB.grid_1(1)) == "[0; 1]"
      @test LARLIB.grid_1(2) == [0 1; 1 2]
      @test LARLIB.grid_1(3) == [0 1 2; 1 2 3]
   end

   @testset "LarGrid Tests" begin
      @test repr(LARLIB.larGrid(1)(0)) == "[0 1]"
      @test repr(LARLIB.larGrid(1)(1)) == "[0; 1]"
      @test LARLIB.larGrid(2)(0) == [0  1  2]
      @test LARLIB.larGrid(2)(1) == [0 1; 1 2]
      @test LARLIB.larGrid(3)(0) == [0  1  2  3]
      @test LARLIB.larGrid(3)(1) == [0  1  2; 1  2  3]
   end
end;
◇
```

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

8.4 Cuboidal grids

More interesting is the generation of *hyper-cubical grids* of intrinsic dimension d embedded in n-dimensional space, via the Cartesian product of d 1-complexes and $(n-d)$ 0-complexes. When $d = n$ the resulting grid is said *solid*; when $d = 0$ the output grid is 0-dimensional, and corresponds to a grid-arrangement of a discrete set of points in $\mathbb{E}^n$.

8.4.1 Full-dimensional grids

Vertex generation First the grid vertices are produced by the `larVertProd` function, via Cartesian product of vertices of the n 1-dimensional arguments (vertex lists in `vertLists`), orderly corresponding to x_0, x_1, ..., x_{n-1} in the output points $(x_0, x_1, \ldots, x_{n-1})$.

⟨ Generation of grid vertices 100 ⟩ ≡

```
function larVertProd(vertLists)
   coords = [[x[1] for x in v] for v in cart(vertLists)]
   return sortcols(hcat(coords...))
end
```

◇

Macro referenced in 94.

8.4.2 Unit tests

"test/jl/largrid.jl" 101 ≡

```
@testset "LarVertProd Tests" begin
   @testset "LarVertProd 1D" begin
      shape = [3]
      vertLists = [LARLIB.vertexDomain(k+1) for k in shape]
      @test typeof(LARLIB.larVertProd(vertLists))==Array{Int64,2}
      @test size(LARLIB.larVertProd(vertLists))==(1, 4)
      @test LARLIB.larVertProd(vertLists)[:,1]==[0]
      @test LARLIB.larVertProd(vertLists)[:,4]==[3]
   end

   @testset "LarVertProd 2D" begin
      shape = [3,2]
      vertLists = [LARLIB.vertexDomain(k+1) for k in shape]
      @test typeof(LARLIB.larVertProd(vertLists))==Array{Int64,2}
      @test size(LARLIB.larVertProd(vertLists))==(2, 12)
      @test LARLIB.larVertProd(vertLists)[:,1]==[0;0]
      @test LARLIB.larVertProd(vertLists)[:,12]==[3;2]
   end

   @testset "LarVertProd 3D" begin
      shape = [3,2,1]
      vertLists = [LARLIB.vertexDomain(k+1) for k in shape]
      @test typeof(LARLIB.larVertProd(vertLists))==Array{Int64,2}
      @test size(LARLIB.larVertProd(vertLists))==(3, 24)
      @test LARLIB.larVertProd(vertLists)[:,1]==[0;0;0]
      @test LARLIB.larVertProd(vertLists)[:,24]==[3;2;1]
   end
end
```

◇

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

8.4.3 Mapping of indices to storage

Multi-index to address transformation The second-order utility `index2addr` function transforms a `shape` list for a multidimensional array into a function that, when applied to a multindex array, i.e. to a list of integers within the `shape`'s bounds, returns the integer address of the array component within the linear storage of the multidimensional array.

The transformation formula for a d-dimensional array with `shape` $(n_0, n_1, ..., n_{d-1})$ is a linear combination of the 0-based[1] multi-index $(i_0, i_1, ..., i_{d-1})$ with `weights` equal to $(w_0, w_1, ..., w_{d-2}, 1)$:

$$addr = i_0 \times w_0 + i_1 \times w_1 + \cdots + i_{d-1} \times w_{d-1}$$

where

$$w_k = n_{k+1} \times n_{k+2} \times \cdots \times n_{d-1}, \qquad 0 \le k \le d-2.$$

Therefore, we get `index2addr([4,3,6])([2,2,0])` $= 48 = 2 \times (3 \times 6) + 2 \times (6 \times 1) + 0$, where `[2,2,0]` represent the numbers of (pages, rows, columns) indexing an element in the three-dimensional array of shape `[4,3,6]`.

⟨ Transformation from multindex to address in a linear array storage 102 ⟩ ≡

```
function index2addr(shape::Array{Int64,1})
   index2addr(hcat(shape...))
end

function index2addr(shape::Array{Int64,2})
    n = length(shape)
    theShape = append!(shape[2:end],1)
    weights = [prod(theShape[k:end]) for k in range(1,n)]
    function index2addr0(multiIndex)
        return dot(collect(multiIndex), weights) + 1
    end
    return index2addr0
end
```

◇

Macro referenced in 94.

8.4.4 Unit tests

"test/jl/largrid.jl" 103 ≡

```
@testset "Index2addr Tests" begin
   @testset "Shape 1D Tests" begin
      @test LARLIB.index2addr([10])([0])==1
      @test LARLIB.index2addr([10])([9])==10
      @test [LARLIB.index2addr([10])([index]) for index in collect(0:9)]==collect(1:10)
   end

   @testset "Shape 2d Tests" begin
      aa = LARLIB.cart([[0;1;2],[0;1]])
      bb = LARLIB.cart([collect(0:9),collect(0:1)])
      cc = LARLIB.cart([collect(0:2),collect(0:2)])
      dd = "Tuple{Int64,Int64}[(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)]"
      ee = "Tuple{Int64,Int64}[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]"
      ff = repr( LARLIB.cart([collect(0:2),collect(0:1)]) )
      @test [ LARLIB.index2addr([3,2])(index) for index in aa ]==collect(1:6)
```

[1] 0-based array, like in C, java and python, as opposed to 1-based, like in fortran or matlab.

```
      @test [ LARLIB.index2addr([10,2])(index) for index in bb ] == collect(1:20)
      @test [ LARLIB.index2addr([3,3])(index) for index in cc] == collect(1:9)
      @test (ff==dd) | (ff==ee)
   end

   @testset "Shape 3d Tests" begin
      aaa = LARLIB.cart([collect(0:3),collect(0:2),collect(0:1)])
      bbb = LARLIB.cart([[0;1;2],[0;1],[0;1]])
      ccc = LARLIB.cart([[0;1],[0;1],[0;1;2]])
      @test LARLIB.index2addr([3,2,1])([0,0,0]) == 1
      @test [ LARLIB.index2addr([4,3,2])(index) for index in aaa ] == collect(1:24)
      @test [LARLIB.index2addr([3,2,2])(index) for index in bbb ] == collect(1:12)
      @test [LARLIB.index2addr([2,2,3])(index) for index in ccc ] == collect(1:12)
   end
end

@testset "BinaryRange Tests" begin
   @test LARLIB.binaryRange(1)==["0";"1"]
   @test LARLIB.binaryRange(2)==["00","01","10","11"]
   @test LARLIB.binaryRange(3)==["000","001","010","011","100","101","110","111"]
end
◇
```

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

8.4.5 Cartesian product of 0/1-complexes

Here, the input is given by the array `cellLists` of lists of cells of the argument complexes. Hence, the `shapes` variable contains the (list of) numbers $m_0, m_1, ...$ of cells in each argument complex, and the `indices` variable (generated by Cartesian product) collects the whole set $M_0 \times M_1 \times \cdots$ of 0-based multi-indices corresponding to the cells of the output complex, with $M_k = \{0, 1, ..., m_k - 1\}$.

The `jointCells` variable is used to contain the list of outputs of Cartesian products of `cells` corresponding to every `index` in `indices`.

⟨ Generation of grid cells 104 ⟩ ≡

```
function larCellProd(cellLists)
   shapes = [length(item) for item in cellLists]
   subscripts = cart([collect(range(0,shape)) for shape in shapes])
   indices = hcat([collect(tuple) for tuple in subscripts]...)
   jointCells = Any[]
   for h in 1:size(indices,2)
      index = indices[:,h]
      cell = hcat(cart([cells[k+1] for (k,cells) in zip(index,cellLists)])...)
      append!(jointCells,[cell])
   end
   convertIt = index2addr([ (length(cellLists[k][1]) > 1)? shape+1 : shape
      for (k,shape) in enumerate(shapes) ])
   [vcat(map(convertIt, jointCells[j])...) for j in 1:size(jointCells,1)]
end
◇
```

Macro referenced in 94.

8.4.6 Unit tests

"test/jl/largrid.jl" 105 ≡

```
@testset "LarCellProd Tests" begin
   @testset "$shape" for shape in [[3,2,1],[3,2],[3],[1,1,1]]
      @testset "$d" for d in 0:length(shape)
         @test typeof(LARLIB.larGridSkeleton(shape)(d)) == Array{Array{Int64,1},1}
      end
   end
end
```
◇

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

8.4.7 Lower-dimensional grid skeletons

In order to compute the d-skeletons of a n-dimensional cuboidal "grid" complex, with $0 \leq d \leq n$, let us start by remarking a similarity with the generation of the boolean representation of numbers between 0 and $2^n - 1$, generated as a list of strings by the `binaryRange` function, given in Section 8.4.7.

The binary representations of such numbers are in fact filtered according to the number of their ones in Section 8.4.7, and used to generate the distinct components of different order skeletons of the assembled grid complexes in Section 8.4.8.

Generation of skeleton components The `binaryRange` function, applied to an integer n, returns the string representation of all binary numerals between 0 and $2^n - 1$. All the strings have the same length n. The bits in each strings will be used to select between either a 0- or a 1-dimensional complex as generator (via a Cartesian product of complexes) of a component of an embedded grid skeleton of proper intrinsic dimension.

⟨ Enumeration of binary ranges of given order 106 ⟩ ≡

```
function binaryRange(n)
   return [bin(k,n) for k in 0:2^n-1]
end
```
◇

Macro referenced in 94.

Filtering grid skeleton components The function `filterByOrder` is used to partition the previous binary strings into $n+1$ subsets, such that the bits into each string sum to the same number, ranging from 0 to n included, respectively.

⟨ Filtering binary ranges by order 107 ⟩ ≡

```
function filterByOrder(n)
   terms = [[parse(Int8,bit) for bit in convert(Array{Char,1},term)] for term in binaryRange(n)]
   return [[term for term in terms if sum(term) == k] for k in 0:n]
end
```
◇

Macro referenced in 94.

8.4.8 Unit tests

"test/jl/largrid.jl" 108 ≡

```
@testset "FilterByOrder Tests" begin
   term = "000"
   bit = '0'
   theTerm = convert(Array{Char,1},term)
   @test typeof(theTerm) == Array{Char,1}
   @test parse(Int8,bit) == 0
   @test [parse(Int8,bit) for bit in theTerm] == zeros(3)
   out = hcat([[parse(Int8,bit) for bit in term] for term in LARLIB.binaryRange(3)]...)
   @test typeof(out) == Array{Int8,2}
   @test size(out) == (3,8)
   @test repr(out) == "Int8[0 0 0 0 1 1 1 1; 0 0 1 1 0 0 1 1; 0 1 0 1 0 1 0 1]"

   @testset  "$n" for n in 1:4
      data = [LARLIB.filterByOrder(n)[k] for (k,el) in enumerate(LARLIB.filterByOrder(n))]
      @test sum(map(length,data)) == 2^n
   end

   @testset "$n,$k" for n in 1:4, k in 0:n
      @test length(LARLIB.filterByOrder(n)[k+1]) == binomial(n,k)
   end
end
```
◇

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

Assembling grid skeleton components We are now finally able to generate the various subsets of cells of a d-dimensional cuboidal grid skeleton, produced respectively by the expression `larCellProd(cellLists)` for every permutation of 0- and 1-complexes, according to the partition classes of permtation of n bits previously produced. To understand why this assembling step of cells is necessary, the reader should look at Figure 8.1, where three subsets of 2-cells of the 2-skeleton, respectively generated by the bit dispositions `[[0,1,1], [1,0,1], [1,1,0]]`, are separately displayed. Notice also that, whereas the dimension n of the embedding space is implicittly provided by the `length` of the `shape` parameter, the intrinsic dimension d of the skeleton to be produced must be given explicitly.

⟨ Assembling grid skeletons 109 ⟩ ≡

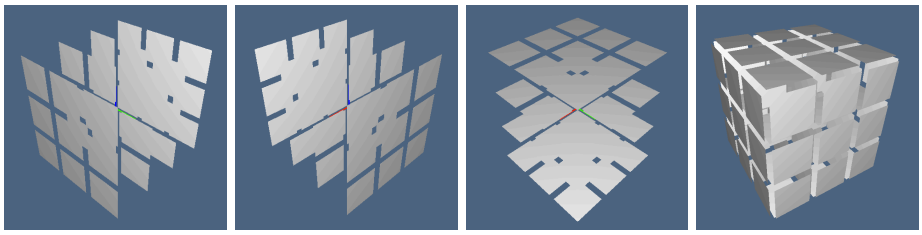


Figure 8.1: (a,b,c) Exploded views of subsets (orthogonal to coordinate axes) of 2-cells of a 2-skeleton grid; (d) their assembled set.

```
function larGridSkeleton(shape)
    n = length(shape)
    function larGridSkeleton0(d)
        components = filterByOrder(n)[d+1]
        mymap(arr) = [arr[:,k]  for k in 1:size(arr,2)]
        componentCellLists = [ [ mymap(f(x)) for (f,x) in zip( [larGrid(dim)
         for dim in shape],component ) ]
               for component in components ]
        out = [ larCellProd(cellLists)  for cellLists in componentCellLists ]
        return vcat(out...)
    end
    return larGridSkeleton0
end
◇
```

Macro referenced in 94.

8.4.9 Highest-level grid interface

The highest-level user interface for (hyper)-cuboidal grid generation is given by the function `larCuboids` applied to the `shape` parameter. For the sake of storage efficiency, the generated vertex coordinates are integer and 0-based in the lowest corner. The model may be properly scaled and/or translated *a posteriori* when needed.

Generation of (hyper)-cuboidal grids The generated complex is always full-dimension, i.e. *solid*, and possibly includes the cells of all dimensions, depending on the Boolean value of the `full` parameter. The grid's intrinsic dimension, as well as the dimension of its embedding space, are specified by the length of the `shape` parameter. See the examples in Figure **??**, but remember that the PLaSM visualiser always embed in 3D the displayed model.

⟨ Multidimensional grid generation 110 ⟩ ≡

```
function vertexDomain(n)
   return hcat([k for k in 0:n-1]...)
end
```

```
function larImageVerts(shape)
   vertLists = [vertexDomain(k+1) for k in shape]
   vertGrid = larVertProd(vertLists)
   return vertGrid
end

function larCuboids(shape, full=false)
   vertGrid = larImageVerts(shape)
   gridMap = larGridSkeleton(shape)
   if ! full
      cells = gridMap(length(shape))
   else
      skeletonIds = 0:length(shape)
      cells = [ gridMap(id) for id in skeletonIds ]
   end
   return convert(Array{Float64,2},vertGrid), cells
end
```
◇

Macro referenced in 94.

8.4.10 Unit tests

"test/jl/largrid.jl" 111 ≡

```
@testset "LarImageVerts Tests" "$shape" for shape in [[3,2,1],[3,2],[10,10,10]]
   @test size(LARLIB.larImageVerts(shape)) == (length(shape),prod(shape + 1))
end
@testset "LarCuboids Tests" begin
    shape = (3,2,1)
    cubes = LARLIB.larCuboids(shape,true)
    verts, cells = cubes
    @test typeof(verts) == Array{Int64,2}
    VV,EV,FV,CV = cells
    @testset "$basis" for basis in [VV,EV,FV,CV]
      @test typeof(basis) == Array{Array{Int64,1},1}
    end
end
```
◇

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

8.5 Cartesian product of cellular complexes

8.5.1 LAR model of cellular complexes

The external representation of a LAR model (necessarily geometrical, i.e. embedded in some $\mathbb{E}^n$, in order to be possible to draw it) is a pair (*geometry*,*topology*), where *geometry* is the list of coordinates of vertices, i.e. a two-dimensional array of numbers, where vertices are given by row, and *topology* is

a list of cells of fixed dimension d. When $d = n$ the model is *solid*; otherwise the model is some emberdded d-skeleton ($0 \leq d < n$).

Binary product of cellular complexes The `larModelProduct` function takes as input a pair of LAR models and returns the model of their Cartesian product. Since this is a pair (*geometry*, *topology*), its second element returns the topological product of the input topologies.

⟨ Cartesian product of two lar models 112 ⟩ ≡

```
function larModelProduct( modelOne, modelTwo )
    (V, cells1) = modelOne
    (W, cells2) = modelTwo

    ⟨ Cartesian product of vertices 113 ⟩
    ⟨ Topological product of cells 114 ⟩

    vertexmodel = []
    for v in keys(vertices)
        push!(vertexmodel, v)
    end
    verts = hcat(vertexmodel...)
    cells = [[v for v in cell] for cell in cells]
    return (verts, cells)
end

function larModelProduct(twoModels)
    modelOne, modelTwo = twoModels
    larModelProduct(modelOne, modelTwo)
end
```

◇

Macro referenced in 94.

Cartesian product of argument vertices The following macro is used to generate a dictionary mapping between integer ids of new vertices and the sets V and W of vertices of the input complexes.

⟨ Cartesian product of vertices 113 ⟩ ≡
```
vertices = DataStructures.OrderedDict();
k = 1
for j in 1:size(V,2)
   v = V[:,j]
    for i in 1:size(W,2)
      w = W[:,i]
        id = [v;w]
        if haskey(vertices, id) == false
            vertices[id] = k
            k = k + 1
        end
    end
end
```
◇
Macro referenced in 112.

Topological product of argument vertices Another macro generates the cells of the topological product, represented as lists of new vertices.

⟨ Topological product of cells 114 ⟩ ≡
```
cells = []
for c1 in cells1
    for c2 in cells2
        cell = []
        for vc in c1
            for wc in c2
                push!(cell, vertices[[V[:,vc];W[:,wc]]] )
            end
        end
        push!(cells, cell)
    end
end
```
◇
Macro referenced in 112.

8.5.2 Unit tests

"test/jl/largrid.jl" 115 ≡
```
@testset "LarImageVerts Tests" "$shape" for shape in [[3,2,1],[3,2],[10,10,10]]
   @test size(LARLIB.larImageVerts(shape)) == (length(shape),prod(shape + 1))
end
```
◇
File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

"test/jl/largrid.jl" 116 ≡
```
@testset "LarCuboids Tests" begin
   @testset "LarCuboids Tests" "$shape" for shape in [[3,2,1],[1,1,1],[3,3,10]]
      @test size(LARLIB.larCuboids(shape)[1],2) == prod(collect(shape) + 1)
```

```
        @test length(LARLIB.larCuboids(shape)[2]) == prod(shape)
        cubes = LARLIB.larCuboids(shape,true)
        verts, cells = cubes
        @test typeof(verts) == Array{Int64,2}
        VV,EV,FV,CV = cells
        @testset "$basis" for basis in [VV,EV,FV,CV]
          @test typeof(basis) == Array{Array{Int64,1},1}
        end
      end

      @testset "LarCuboids Tests" "$shape" for shape in [[3,2],[1,1],[3,10]]
        @test size(LARLIB.larCuboids(shape)[1],2) == prod(collect(shape) + 1)
        @test length(LARLIB.larCuboids(shape)[2]) == prod(shape)
        cubes = LARLIB.larCuboids(shape,true)
        verts, cells = cubes
        @test typeof(verts) == Array{Int64,2}
        VV,EV,FV = cells
        @testset "$basis" for basis in [VV,EV,FV]
          @test typeof(basis) == Array{Array{Int64,1},1}
        end
      end

      @testset "LarCuboids Tests" "$shape" for shape in [[3],[1],[10]]
        @test size(LARLIB.larCuboids(shape)[1],2) == prod(collect(shape) + 1)
        @test length(LARLIB.larCuboids(shape)[2]) == prod(shape)
        cubes = LARLIB.larCuboids(shape,true)
        verts, cells = cubes
        @test typeof(verts) == Array{Int64,2}
        VV,EV = cells
        @testset "$basis" for basis in [VV,EV]
          @test typeof(basis) == Array{Array{Int64,1},1}
        end
      end
    end
    ◇
```

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

"test/jl/largrid.jl" 117 ≡

```
    @testset "LARLIB.LarModelProduct Tests" begin
      geom_0 = hcat([[x] for x=0.:10]...)
      topol_0 = [[i,i+1] for i=1:9]
      geom_1 = hcat([[0.],[1.],[2.]]...)
      topol_1 = [[1,2],[2,3]]
      model_0 = (geom_0, topol_0)
      model_1 = (geom_1, topol_1)
      model_2 = LARLIB.larModelProduct(model_0, model_1)
      model_3 = LARLIB.larModelProduct(model_2, model_1)
      @test size(model_2[1],2) == size(model_0[1],2) * size(model_1[1],2)
      @test length(model_2[2]) == length(model_0[2]) * length(model_1[2])
```

```
    @test size(model_3[1],2) == size(model_2[1],2) * size(model_1[1],2)
    @test length(model_3[2]) == length(model_2[2]) * length(model_1[2])
    V,EV = model_0
    @test typeof(V) == Array{Float64,2}
    @test typeof(EV) == Array{Array{Int64,1},1}
    V,EV = model_1
    @test typeof(V) == Array{Float64,2}
    @test typeof(EV) == Array{Array{Int64,1},1}
    V,FV = model_2
    @test typeof(V) ==  Array{Float64,2}
    @test typeof(FV) == Array{Array{Int64,1},1}
    V,CV = model_3
    @test typeof(V) ==  Array{Float64,2}
    @test typeof(CV) == Array{Array{Int64,1},1}

    modelOne = ([0 0 1 1; 0 1 0 1], Array{Int64,1}[[1, 2, 3, 4]])
    modelTwo = ([0 1], Array{Int64,1}[[1, 2]])
    @test LARLIB.larModelProduct(modelOne, modelTwo) == LARLIB.larModelProduct(modelTwo, modelOne)
    @test LARLIB.larModelProduct(modelTwo, modelTwo)[1] == [0  0  1  1; 0  1  0  1]
    @test LARLIB.larModelProduct(modelOne,modelOne)[2][1] == Any[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
end
◇
```

File defined by 99, 101, 103, 105, 108, 111, 115, 116, 117.

Chapter 9

Utilities

9.1 Overview

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

"lib/jl/utilities.jl" 118 ≡
⟨ Utilities 120, ... ⟩
◇

9.1.1 Tests

As usual every function has some unit tests.

"test/jl/utilities.jl" 119 ≡
```
using Base.Test
using LARLIB

```
⟨ Utilities tests 121, ... ⟩
◇

9.2 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 120 ⟩ ≡
```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.2.1 Tests

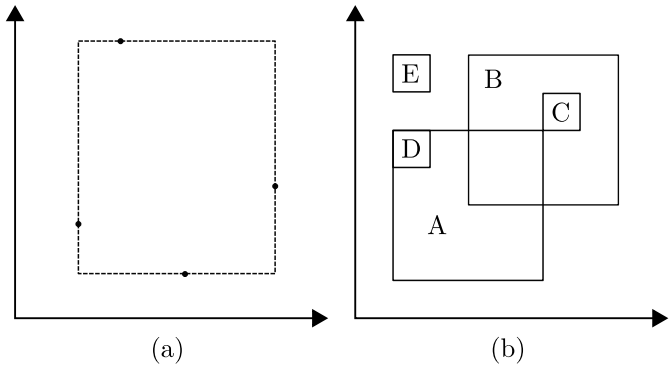


Figure 9.1: (a) is a visualization of the test for bboxes building, (b) for bbox containment.

⟨ Utilities tests 121 ⟩ ≡

```
@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 121, 123.
Macro referenced in 119.

9.3 Face area calculation

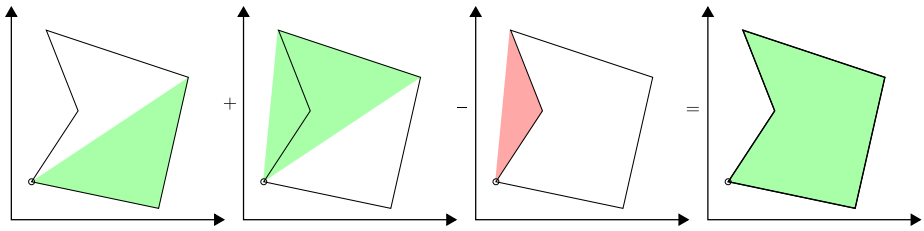


Figure 9.2: A visual representation of the face area calculation algorithm. The area of the face is the sum of the areas of each triangle which can be build using the pivot vertex and the other vertices of the face

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.

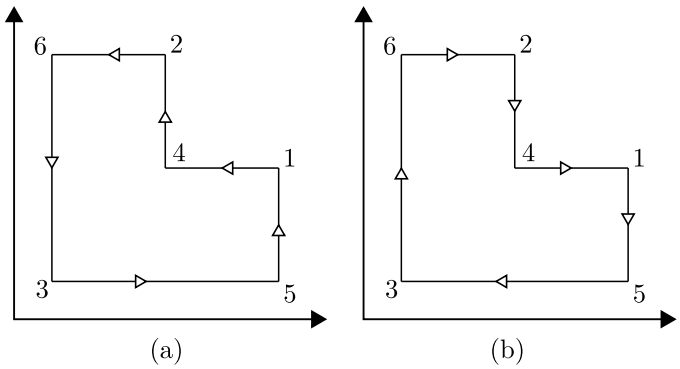
⟨ Utilities 122 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.3.1 Tests



The two faces drawn above they must have complimentary area.

⟨ Utilities tests 123 ⟩ ≡

```
    @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 121, 123.
Macro referenced in 119.

9.4 Skeletal merge

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

⟨ Utilities 124 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.5 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 125 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.6 FV building

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

⟨ Utilities 126 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.7 Boundaries building

⟨ Utilities 127 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.8 Vertex equality utilities

Vertex comparison must be performed using floating-point fixed error [ref. 3.2.1].

⟨ Utilities 128 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.9 Full triangulation

⟨ Utilities 129 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.10 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 130 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

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 131 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

9.11 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 132 ⟩ ≡

```
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 120, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132.
Macro referenced in 118.

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.