Fast algebraic filtering of surfaces from 3D medical images with Julia

Miroslav Jiřík and Alberto Paoluzzi

Abstract

In this paper we introduce a novel algebraic LAR-SURF filter, well founded on algebraic topology methods, to extract and smooth the boundary surface of any subset of voxels arising from the segmentation of a 3D medical image. The input is defined as a *chain*, i.e. as a vector from a linear space of 3-chains, represented in coordinates as a sparse Boolean vector. The output is produced as the result of the mapping via the linear boundary operator $\partial_3 : C_3 \to C_2$ between linear spaces of 3- and 2-chains. In particular, when the input set of voxels is either not (4-)connected, or contains one or more empty regions inside, LAR-SURF generates a non connected set of closed surfaces, i.e. a set of 2-cycles—using the language of algebraic topology. The only data structures used by this approach are sparse arrays with one or two indices, i.e. sparse vectors and matrices. This work is based on LAR (Linear Algebraic Representation) methods, and is implemented in Julia language, natively supporting parallel computing on hybrid architectures.

Contents

1 Introduction

2 Background

2.1 Representation Schemes

A *representation scheme* for solid modeling is a mapping between a space of mathematical models and a space of symbolic representations, often generated by a formal grammar. Solid pointsets (i.e., 'r-sets') are defined [**?**] as compact (bounded and closed) regular and semianalytic subsets of the d-space. A large number of representation schemes were defined in the past forty years, including the two main classes of (a) *boundary representations* ('B-reps'), where the solid model is represented through a representation of its boundary elements, i.e. faces, edges and vertices, and (b) *decompositive/enumerative representations*, that are a decomposition of either the object or the embedding space, respectively, into a well-defined cellular complex. In particular, a boundary representation provides a cellular decomposition of the object's boundary into cells of dimension zero (vertices), one (edges), and two (faces). Medical imaging can be classified as *enumerative representation* of cellular decompositions of organs and tissues of interest, in particular, as subsets *of 3D volume elements* (voxels) from the 3D image.

2.2 Linear Algebraic Representation

The *Linear Algebraic Representation* (LAR) aims to represent the *chain complex* generated by a piecewise-linear geometric complex embedded either in 2D or in 3D. In few words, it gets a minimal characterization of geometry and topology of a cellular complex, i.e. the embedding mapping $\mu : C_0 \to \mathbb{E}^d$ of 0-cells (vertices), as well a description of $(d-1)$-cells as subsets of vertices, and is able to return the whole chain complex

$$C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta_2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta_1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta_0}{\leftrightarrows}} C_0.$$

and, in particular, any basis for linear chain spaces C_p, and any linear boundary/coboundary map ∂_p and $\delta_p = \partial_{p+1}^\top$ between them. The domain of LAR is the set of chain complexes generated by cell d-complexes ($2 \leq d \leq 3$). The computer representations of LAR are sparse binary matrices to represent both the operators and the chain bases. Note that in algebraic topology a p-chain is defined as a linear combination of p-cells with scalars from a field. When the scalar coefficients are from $\{-1, 0, +1\}$, a chain may represent *any (oriented) subset of cells* from the cellular complex.

We may therefore get the $(p-1)$-boundary $\partial_p \gamma_p$ of *any* p-chain γ_p, by multiplication of the coordinate representation $[\partial_p]$ of the boundary operator times the coordinate representation $[\gamma_p]$ of the chain in terms of such scalars, i.e. by a matrix-vector product $[\partial_p][\gamma_p]$.

It is easy to understand that the LAR representation scheme is very expressive, i.e. that it has a large domain, including collections of: line segments, quads, triangles, polygons, meshes; pixels, voxels, volume images; B-reps, enumerative and decompositive representa-

tions of solids. In this paper we apply LAR methods to computation of boundary representations of solid models from segmentation (labeling) of 3D medical images.

2.2.1 Construction of ∂_d boundary matrix

Once fixed an ordering for all the cells (vertices, edges, pixels, and voxels), i.e. for 0-, 1-, 2-, and 3-elements of a cell partitioning of a 3D image, i.e., once fixed the p-bases for linear spaces C_p of p-chains ($0 \leq p \leq d$), we call $M_p = (m_{i,j})$ the *characteristic matrix* of the p-basis, expressed as subsets of 0-cells, where we have that $m_{i,j} = 1$ iff the j-th 0-cell belongs to the i-th p-cell, and $m_{i,j} = 0$ otherwise.

Note that, by computing the (sparse) matrix product $(M_{p-1}M_p^t) = (n_{i,j})$, with $n_{i,j} = \sum_k m_{i,k}m_{k,j}$, we get for each n_{ij} the *number of vertices* shared by c_{p-1}^i and c_p^j. When this number equates the cardinality of c_{p-1}^i, this elementary chain is contained on the boundary of c_p^j. In a 3D image, with cubic 3-cells and squared 2-cells, everywhere we get $n_{i,j} = 4$, we may state $c_2^i \subset \partial c_3^j$. In this case, by looking in each j column of $M_2M_3^t = (n_{i,j})$, we have exactly *six rows* where $n_{i,j} = 4$.

Finally, consider the linear graded boundary operator $\partial_p : C_p \to C_{p-1}$. As such, it contains by columns the representation of domain basis elements, expressed as linear combination of the basis elements of the range space. Therefore, the operator matrix $[\partial_d]$ is readily obtained by setting $[\partial_d](i,j) = 1$ iff $n_{i,j} = 4$ and $[\partial_d](i,j) = 0$ otherwise. Of course, it will contain six non-zero elements for column. It may be worth to remember that every 3-cell (voxel) has exactly six 2-faces.

It is possible to show that all the interesting relation of incidence/adjacency between cells of different dimensions can be both computed and efficiently queried by pairwise computing some products, with one of terms possibly transposed, of the two boundary and coboundary operator matrices $[\partial_p]$ and $[\delta_p] = [\partial_p^\top] = [\partial_p]^t$. We may also show that such matrices are *very sparse*, with their sparseness growing rapidly with the dimensions (see Section 3.2). The pattern of non-zeros in matrix $[\partial_3]$ corresponding to a brick of shape $(4, 4, 4)$ is given in Figure 1.

2.3 Multiindices from Cartesian indices

In order to utilize the topological algebra shortly recalled in this paper, we need to explicitly sort the cells of the various dimensions into linearly ordered sequences, possibly according to the linear order their information is linearly accommodated in computer storage.

2.4 Taubin Smoothing

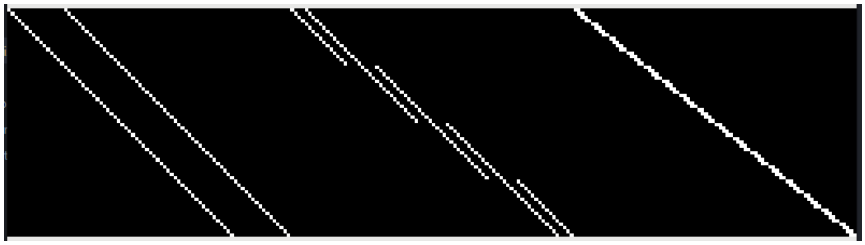

Figure 1: A binary image of the coboundary operator $\delta_2 = \partial^\top : C_2 \to C_3$, built for small 3D images with shape $(4, 4, 4)$. Note that the number of rows equates the cardinality $4 \times 4 \times 4 = 64$ of the voxel set; the number of columns is $d\, n\, (1+n)^{d-1} = 3 \times 4 \times 25 = 300$. Of course, the number of non-zeros per row (cardinality of single voxel facets) is six.

3 Block-parametric design

3.1 Block decomposition

We assume that medical devices produce 3D images with lateral dimensions that are integer multiples of some powers of two, like 128, 256, 512, etc. Any cuboidal portion of image is completely determined by the Cartesian indices of its voxel of lowest and highest indices, and extracted by multidimensional array *slicing* as $Image([\ell_x{:}h_x, \ell_y{:}h_y, \ell_z{:}h_z])$.

For the sake of simplicity, we assume a common size on the three image axes, and the corresponding image portion B, called *block*, as a function of its element of the lowest block coordinates $i, j, k \in \mathbb{N}$ and of block dimension n:

$$B(i, j, k, n) := Image([in{:}in + n, jn{:}jn + n, kn{:}kn + n])$$

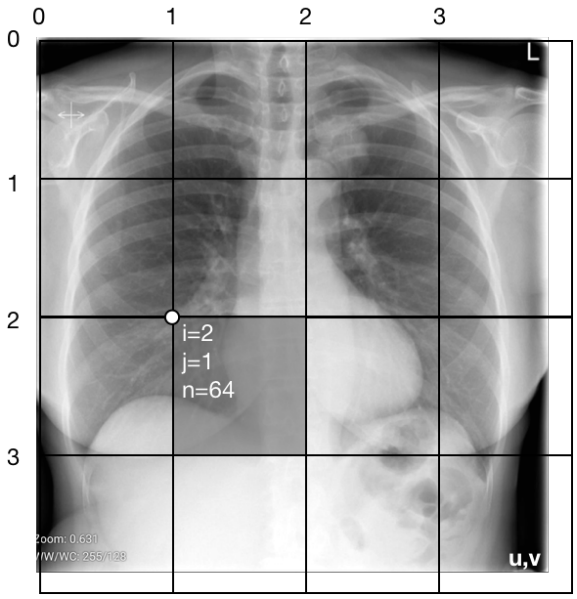

Figure 2: A possible block partitioning of a radiologic image. The evidenced 2D block, of size $n^d = 64^2$, is sliced as $B([2, 1, 64]) = Image([128 : 172], [64 : 128)]$

Figure 2 shows the block decomposition in a 2D image, with positive integer u, v lateral sizes of image. Note that block sides do not necessarily correspond to image edges.

3.2 Block operator

Chain coordinates We are going to treat each image block independently from each other. Hence we map each image subspace $B(i, j, n)$ to the linear *chain* space C_2 of dimension $n \times n$, using coordinate vectors $c_{h,k} \in \mathbb{B}^{n\times n} := \{0, 1\}^{n\times n}$, where the basis element

$c = c_{h,k} \in C_2$ is mapped via Cartesian-to-linear map to the Boolean vector

$$Image(h,k) \mapsto c_{h,k} := [0 \cdots 0\ 1\ 0 \cdots 0] \in \mathbb{B}^{n \times n}$$

for each $0 \leq h, k \leq n$, and where the (single) unit element is in position $nk + h \leq n \times n$.

Therefore, each pixel (or voxel) in a block image will be seen as a basis Boolean vector in C_2, and each subset of image elements, as the corresponding Boolean vector in C_2, with many ones as the cardinality of the subset.

Boundary operator For a fixed block size n, the boundary operator $\partial_d : C_d \to C_{d-1}$, with $d \in \{2, 3\}$, will be constructed once and for all using the algorithm given in [], and inlined in the generated boundary extraction code.

It is easy to see that the operator's matrix $[\partial_d]$ is *very sparse*, since it contains $2 \times d$ non-zero elements (ones) for each column (of length n^d), i.e. 4 ones and 6 ones for the 2D and 3D case, respectively. In fact the matrix of a linear operator between linear spaces contains by columns the basis element of the domain space, represented in the target space. In our case, the former is an image element (2-cube or 3-cube), represented as the chain of its boundary—i.e. either a 1-cycle of 4 edges, or a 2-cycle of 6 faces, respectively.

The number of rows of $[\partial_d]$ equates the dimension of the linear space C_{d-1}, i.e. the number of $(d-1)$-cells—elementary $(d-1)$-chains—in the cellular partition of the image. To compute their number, we act in two steps. (a) First we map one-to-one the n^d d-cells with d adjacent $(d-1)$-cells, so getting $d\,n^d$ distinct basis elements of C_{d-1}. (b) Then we complete this bases by adjoining n^{d-1} boundary elements for each of the d dimensions of the image, so providing further $d\,n^{d-1}$ basis elements for C_{d-1}. The dimension of C_{d-1}, and therefore the number of rows of $[\partial_d]$ matrix is $d\,(n^{d-1} + n^d) = d\,n\,(1+n)^{d-1}$. The number of column equates the number of basis elements of C_d, i.e. the number n^d of block elements.

Sparsity and size of boundary matrix As we have seen, we have $2d$ non-zero elements for each column of $[\partial_d]$, so that their total number is $2d\,n^d$. The number of matrix element is $d\,n\,(1+n)^{d-1} \times n^d$, giving a ratio of

$$\frac{\text{non-zero elements}}{\text{total elements}} = \frac{2d \times n^d}{d\,n\,(1+n)^{d-1} \times n^d} = \frac{2}{n + n^d}$$

Using sparse matrices in CSC (Compressed Sparse Column) format we get a storage size:

$$mem([\partial_d]_{n^d}) = 2 \times \#\text{nzero} + \#\text{columns} = 2 \times 2d\,n^d + n^d = (4d+1)n^d.$$

In conclusion, for block size $n = 64$, the matrix $[\partial_d]$ requires for 2D images $9 \times 64^2 = 36,864$ memory elements, and for 3D images $13 \times 64^3 = 3,407,872$ memory elements. Counting the bytes for the standard implementation of a sparse binary matrix (1 byte for values and 8 bytes for indices) we get $(18d+8)n^d$ bytes, giving 176 KB for 2D and 15.872 MB for 3D.

3.3 Block boundary mapping

Here we refer directly to the 3D case. Let us call *segment* the bulk content S of interest within the input 3D image of size (u, v, w). We aim to compute the segment boundary $\partial_3 S$. First we set the size n of the block, in order to decompose the input $Image(u, v, w)$ into a fair number

$$M = \lceil u/n \rceil \times \lceil v/n \rceil \times \lceil w/n \rceil \simeq \frac{uvw}{n^3}$$

of blocks. Then we consider each image portion $c_{i,j,k} = S \cap B(i, j, k, n)$ and compute its (binary) coordinate representation $[c]_{i,j,k} \in C_3(n, n, n)$. This one is a sparse binary vector of length n^3. Then assemble the M representations c of segment portions into a sparse binary matrix $\mathbf{S}$, of dimension $n^d \times M$. Finally compute a matrix $\mathbf{B}$ of boundary portions of S, represented by columns as chain coordinate vectors in C_2:

$$\mathbf{B} = [\partial_3(n)]\,\mathbf{S}.$$

where the boundary matrix has dimension $d\,n\,(1+n)^{d-1} \times n^d$. Of course, the $\mathbf{B}$ sparse matrix has the same column number M of $\mathbf{S}$, because each column contains the boundary representation of the corresponding $S \cap Box_{i,j,k} i, j, k$, and the number of rows of the operator, equal to the dimension of the linear space C_2.

Embedding A final computational step is needed, in order to embed the 2-chains in Euclidean space $\mathbb{E}^3$ and to assemble the whole resulting surface. In particular, we need to compute the *embedding function* $\mu : C_0 \to \mathbb{E}^3$, where C_0 is the space of 0-chains, one-to-one corresponding to the vertices of the extracted surface. The simplest solution is to associate four 0-cells to each 2-cell of the extracted surface, i.e. to each non-zero entry in every column of $\mathbf{B}$. The μ function can be computed by identifying, via element position in the column, a triple of integer values $0 \le x \le u$, $0 \le y \le v$, and $0 \le z \le w$ for each vertex of the 2-cell. The mapping can be implemented using a dictionary, that will store the inverse coordinate transformation used at the beginning, i.e. the one from linear to Cartesian coords, in order of not duplicating the output vertices.

Surface assembling All boundary surface subsets $B_{i,j,k}(S) = \partial_3 S \cap \mathrm{Box}_{i,j,k}$, provided by columns of $\mathbf{S}$, are embedded in the same coordinate space. In formal terms:

$$\mathtt{Lar}(S) := (\mathtt{Geom}(S), \mathtt{Top}(S)) = (\mathtt{V}, \mathtt{CV}),$$

where, with respect to the *chain complex* $C_3 \to C_2 \to C_1 \to C_0$ induced by the input image Im and segment $S_{i,j,k}$, we get

$$\mathtt{Geom} := \mu(C_0(_{i,j,k})) = \mathtt{V}, \tag{1}$$

$$\mathtt{Top} := C_3(S) = \mathbf{S} \mapsto \mathtt{CV}. \tag{2}$$

and

$$\texttt{Lar}(B_{i,j,k}) := (\texttt{Geom}, \texttt{Top}) = (\texttt{W}, \texttt{FW}), \tag{3}$$

$$\texttt{Geom} := \mu(C_0(B_{i,j,k})) = \texttt{W} \subset \texttt{V}, \tag{4}$$

$$\texttt{Top} := C_2(B_{i,j,k}) = \mathbf{B}_{i,j,k} \mapsto \texttt{FW} \subset \texttt{FV}. \tag{5}$$

A translation transformation applied to each vertex subset $\texttt{W}_{i,j,k}$ with translation vector $\mathbf{t} = [i, j, k]$ will therefore move it in the final space position, so finally giving

$$\texttt{Lar}(B) = \oplus_{i,j,k}\texttt{Lar}(\partial_3 S_{i,j,k})) = \oplus_{i,j,k}(\texttt{W}, \texttt{FW}).$$

3.4 Block-level parallelism

In the computational pipeline introduced in this paper, several steps can be efficiently performed in parallel at image-block level, depending on the embarassingly data parallel nature of the problem. In particular, little effort is needed to separate the problem into a number of parallel tasks $S_{i,j,k}$, using multiarray slicing. The granularity of parallelism, depending on the block size n, is further enforced by the computation of a single boundary matrix $[\partial_d(n)]$, in turn depending on n, so that the initial communication cost of broadcasting the matrix to nodes can be carefully controlled, and finely tuned depending on the system architecture. The whole approach is appropriate for SIMD hybrid architectures of CPUs and GPUs, since only the initial block setup of boundary matrix and image slices, as well the final collection of computed surface portions, require inter-process communication.

3.5 Julia implementation

The computer code is implemented in Julia language [] according to the workflow described below, whose stages are parallelized and/or optimized in various ways.

Workflow setup The functions in this preliminary step include:

1. input of 3D medical image $\mathcal{I}$ dimensions ℓ_1, ℓ_2, ℓ_3, such that: $\mathcal{I} = [\ell_1] \times [\ell_2] \times [\ell_3]$, where $[\ell_k] = (1, 2, \ldots, \ell_k)$;

2. analysis of resources available in the computational environment, including operating system, type and number of compute nodes (processors, cores, GPUs), number of cores per node, RAM and caches amounts;

3. depending on the above, best decision for *size* of 3D image blocks B (bricks); defaults to: $size = 64$; hence the number of bricks will be $n = \lceil \ell_1/size \rceil \times \lceil \ell_2/size \rceil \times \lceil \ell_3/size \rceil$. Hence default value: $n = 256$, for images $\mathcal{I} = 512 \times 512 \times 256$;

4. computation of a Julia's sparse boundary matrix $[\partial_B]$, returning a value of type `SparseMatrixCSC{Int8}{Int64}`, where `Int8` and `Int64` are the types for values and indices, respectively, stored by Compressed Sparse Column (CSC) format; the average $[\partial_B]$ (for $size = 64$) is about

5. creation of a either a local or distributed channel to implement a producer/consumer model of parallel/distributed computation, depending on available resources;

6. distribution of matrix $[\partial_B]$, of size 45 MB, to all available nodes/cores (workers) using the macro `@eveywhere`.

The number of non-zeros within the sparse matrix $[\partial_{64^3}]$ is $\texttt{nnz} = 4792266$, for a memory size of $9 \times \texttt{nnz} + 8 * 262144 \simeq 45$ MB. The memory size of the sparse matrix is computed as $8 + 1$ bytes for non-zero element (i.e. 6 per row), plus 8 bytes per column.

Job enqueuing Communication and data synchronization may be managed through *Channels*, which are the FIFO conduits that provide producer/consumer communication. Overall execution time can be improved if other tasks can be run while a task is being executed, or while waiting for an external service/function to complete. The single work items follows:

1. extraction of the block views from image arrays depending on 3 indices;

2. transformation of each job from global to local coordinates;

3. further transform of each forground voxel from Cartesian to linear coordinates, using the suitable Julia's library functions.

4. enqueing of the job (a sequence of integer positions for the non-zeros image elements aligned in a memory buffer).

3-Chain encoding Each block B of the 3D image must by converted into the coordinate representation of a vector of the linear space C_3 of 3-chains. In coordinates local to B, once fixed an ordering from Cartesian to linear coords, this vector is represented by a binary array of length $64^3 = 262144$, with a non-zero value (i.e. 1) for each foreground voxel in the interesting image portion in B. Therefore, the coded portion of the segment $\mathcal{S}$ inside B, denoted as $\mathcal{S}(B)$, results with a space occupancy of about 262 KB if encoded as a full array (i.e. including the zero values), When encoded as a sparse vector, its space occupancy will correspondingly decrease.

1. each job encoding produces either a full or sparse binary vector. With full arrays depending by one index, we get 262 KB per job;

2. special format for sparse CSC (Compressed Sparse Column) vectors can be used, since the *value* data for non-zeros do not need storage. Hence only a single 1-array of `Int64` row positions (with total length equal to the number of non-zeros in the block, with $8 \times$ `nnz` KB storage) is needed;

3. prepare subsequnces of such data vectors (non-zero row indices in linear indices), in order to feed efficiently the available processor threads. In case of presence of one/more GPUs, a smaller size of the block—and hence of the boundary matrix and the encoded 3-chain vectors—and then much higher vector numbers, are preferable for speed.

SpMM Multiplication According to the current literature [] it is more convenient to execute SpMV (sparse matrix-vector) multiplications than SpMSpV (sparse matrix-sparse vector) multiplications. Since we have 256 such jobs (one multiplication per block) to perform in the standard setting of the algorithm[1], or even more, in case of either smaller blocks or image greater than the standard ones, this stage must be evidently parallelized and carefully tuned, possibly by using the available GPUs.

1. Various multiplication algorithms are being experimented, using several packages for sparse algebra and/or customized implementations;

2. the total speed of this stage will strongly depend on both the hardware available, on the granularity of blocks, and on the choice between dense/sparse storage of encoded 3-chains;

[1] Size of block 64^3; size of image $512^2 \times 256$.

3. anyway, the compute elements or threads will be feed without solution of continuity in a *dataflow* process. This parallel operation is, according to our preliminary experiments, the critical one of the whole workflow, in the sense that any ΔT (either positive or negative) in this stage will contribute to the total time T.

2-Chain decoding Each multiplication of $[\partial_B] : C_3 \to C_2$, times a 3-chain $\nu \in C_3$, produces a 2-chain $\sigma \in C_2$, i.e. the coordinate representation of the boundary vector $\sigma \in C_2$. The inverse of the coding algorithm is executed in the present stage. This process can also be partially superimposed in time with the previous one, depending on the size of the memory buffer used to feed the CPU cores or the GPUs. The elementary steps follow:

1. conversion from position of ones (or non-zeros) in the 2-chain to row linear indices;
2. conversion from linear indices to Cartesian indices in coordinates local to the B block, using the appropriate library functions;
3. conversion from each Cartesian index value to a suitably oriented (i.e. with proper attitude) geometry quadrilateral (or pair of triangles) in local coordinates.

A Julia's vectorized pipeline seems the more appropriate implementation model for the job of each worker.

Assembling and artifact filtering The results of the previous stages can be described a a collection of sets of geometric quads, each one encoded as an array of quadruples of integer indices, pointing to the linear array of grid vertices associated to the standard image block B. In other words, *all quads* of **each job** are now given in the **same** *local coordinates*. Other than to put each partial surface $\mathcal{S}(B) = (\mathtt{V}_B, \sum_{\sigma \in B} \mathtt{FV}_\sigma)$ in the global coordinate system of the image, the present stage must eliminate the redundant boundary features possibly generated at the edges of the partial surface $\mathcal{S}(B)$ within each block B such that $B \cap \mathcal{I} \neq \emptyset$.

1. translate each array `FV`, of type `Lar.Cells`, by summing to each linear vertex index the linearized offset of the Cartesian coordinates $(n, m, p)(B)$ of the vertex with lowest coordinates of its block B.
2. remove both instances of double quads generated by `Lar` software at the block boundaries (see Figure **??**). They are artifacts generated by the decomposition of the whole image into a number of blocks of tractable size.
3. a smart strategy of removal of such artifacts can be used, which does not require any sorting nor searching on the assembled array of quads. It will consist of arranging each block with all three dimensions increased by one, so that each 2-adjacent pair of blocks will covering each other for a full side of blocks of depth one. The details of this artifact filtering will be elucidated in Section **??**.

Smoothing The final smoothing of the generated surfaces cannot be performed blockwise, since this would introduce smoothing artifacts at the block boundaries. Anyway, the Taubin smoothing [] can be performed in parallel, since for each vertex in the final surface (except eventually the ones on the image $\mathcal{I}$ boundaries) it essentially consist in computing a new position as a proper average of its neighborhood vertices, i.e. by applying a discrete Laplacian operator. Some appropriate set workers may so be assigne the task of generating iteratively a new position for the vertices they take cure of. In particular, we have:

1. Job enqueuing, by writing sets of integers (global linear indices of vertices) in array buffers of appropriate type `Channel`;
2. iterated vectorized computation of proper averages of closed vertices;
3. job dequeuing, by recovering finished tasks from a channel and assembling the results into the embedding function $\mathtt{V} : C_0 \to \mathbb{E}^3$ providing an array of type `Lar.Points` of `Float64` $\times$ `3`, with vertex coordinates by column.

3.5.1 Code optimization

3.5.2 Performance analysis

4 Examples

5 Conclusion