Algebraic Filtering of Surfaces from 3D Medical Images with Julia

Miroslav Jirik[1,3], Antonio DiCarlo[2], Václav Liska[1,3], Alberto Paoluzzi[2],

[1] Charles University, Faculty of Medicine in Pilsen, Biomedical Center, jirik@lfp.cuni.cz
[2]Roma Tre University, Department of Mathematics and Physics, Rome, Italy,
[3]NTIS, Faculty of Applied Sciences, University of West Bohemia,

Corresponding author: Miroslav Jirik, jirik@lfp.cuni.cz

Abstract. In this paper we introduce a novel algebraic filter, based 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 of the Linear Algebraic Representation (LAR) Surface extraction filter (LAR-SURF) is defined as a *chain*, i.e. a vector from a linear space of chains, here subsets of voxels, represented in coordinates as a sparse binary vector. The output is produced by a linear mapping between spaces of 3- and 2-chains, given by the boundary operator $\partial_3 : C_3 \to C_2$. The only data structures used by this approach are sparse arrays with one or two indices, i.e. sparse vectors and sparse matrices. This work is based on LAR algebraic methods and is implemented in Julia language, natively supporting parallel computing on hybrid hardware architectures.

Keywords: Medical 3D, Computational Topology, Linear Algebraic Representation, LAR, Julia, Surface Extraction
DOI: https://doi.org/10.3722/cadaps.2020.xxx-yyy

1 Introduction

Isosurface extraction to produce geometric models of surfaces from volumetric data is important in many applications. It is often used for interactive visualization of medical data and/or for flow modeling [21].

The most popular algorithm used for surface extraction from volume images is called *Marching Cubes* (MC). The algorithm was described by Lorentsen and Cline [13] in 1987. A survey of algorithms MC-inspired has been published in 2006 [15]. The algorithm is based on considering the small cubes defining the volumetric image. Each corner vertex of each cube is related to input volumetric data, typically as average of the incident voxel data. MC traverses the data cube-by-cube, and constructs a triangulated iso-surface by using a lookup table depending on pattern of transitions between values of adjacent cube vertices. The main disadvantages of this method are time requirements, ambiguity, and holes generation. Some of them were discovered shortly after the algorithm was introduced. In 1991 Nielson and Hamman described an Asymptotic Decider to solve the ambiguity problem on the faces of the cube. Natarajan noted that the ambiguity problem also occurs with uniform samples [14]. In 1995 Chernyaev extended the number of lookup cases to 33 [5]. More recently, the algorithm was updated by Custodio, Pesco, and Silva to enhance the quality of iso-surface triangulation [7].

Some alternative methods have been developed, including a method for surface extraction from a grid of field values using particle attraction; a system was described by Crossno and Angel in [6]. A graph processing that tracks the boundary cell-face adjacencies is described in [12]. Some parallel algorithms for iso-surface extraction are discussed in [1]. A data-parallel algorithm, implemented in OpenCL, that runs entirely on the GPU is presented in [23]. A Linear Algebraic Representation approach, parallelized using the OpenCL framework on Linux, was introduced in [16].

In the present paper we discuss a distributed approach for surface extraction, where the LAR-SURF (Linear Algebraic Representation Surface Extraction) filter is based on basic linear algebra and algebraic topology, using linear spaces C_p of chains (of cells) of dimension $0 \leq p \leq 3$ and the boundary matrix $[\partial_3] : C_3 \to C_2$.

Input volumetric data are represented by a 3D voxel array and can be generated, e.g., by segmentation of a computed tomography (left image of Fig. 4). A decomposition of the input volumetric data into small submatrices called *bricks* is performed, then the binary coordinate vector of each *segment* (mathematically, a chain) of voxels is generated, and its boundary is computed by multiplication times the boundary matrix. The resulting output is a sparse binary vector holding the LAR representation of the boundary surface. Such embarrassing parallel data decomposition is used to independently compute within each of the bricks the boundary patches, that are finally joined and smoothed via the Taubin algorithm [26].

The present paper is organized as follows. Section 2 provides the basic topological and geometrical concepts needed to understand the LAR-SURF method, including the building of boundary matrices, the map from Cartesian indices to linear indices, and the Taubin smoothing method. Section 3 discusses the parametric design of the unit block filtered by the parallel algorithm, including the block decomposition, the sparsity rate of the used sparse arrays, and the block-level parallelism. Section 4 is related to the algorithm implementation in Julia, and in particular to a discussion of the parallel workflow. Section 5 presents some examples of algorithm execution on the liver and the hepatic portal system. Section 7 shortly describes the next extensions of this approach, in particular the implementation with Julia's support for GPU parallelism and the multi-segmentation of medical images.

2 Background

Some basic concepts of solid modeling, and in particular the foundational idea of representation scheme, as well as few basic concepts of algebraic topology, are shortly introduced in this section, including the computation of the matrix of a boundary operator between chain spaces.

2.1 Representation Scheme

A *representation scheme* for solid modeling is a mapping between a space of mathematical models and a space of symbolic representations, like generated by a formal grammar. Solid pointsets (i.e., "r-sets") are defined in [20] as compact (bounded and closed) regular and semianalytic[1] 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* [20], 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 the *enumerative*

[1]Semianalytic sets, studied in algebraic geometry, are solutions of systems of polynomials; are closed under (regularized) set union, intersection and difference, and so constitute a Boolean algebra [18].

representation of cellular decompositions of organs and tissues of interest [16], in particular, as subsets *of 3D volume elements* (voxels) from the 3D medical image.

2.2 Linear Algebraic Representation

The *Linear Algebraic Representation* (LAR), introduced in [9], aims to represent the *chain complex* [8, 17] generated by a piecewise-linear *geometric complex* embedded either in 2D or in 3D. This representation provides a minimal characterization of geometry and topology of a cellular complex, through (a) the embedding mapping $\mu : C_0 \to \mathbb{E}^d$ of 0-cells (vertices), and (b) a description of d-cells and/or $(d-1)$-cells as subsets of vertices. When evaluated, it is able to return the whole chain complex:

$$C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta_2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta_1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta_0}{\leftrightarrows}} C_0. \tag{1}$$

i.e., the whole sequence of graded linear *chain spaces* C_p, and any linear *boundary* ∂_p and *coboundary* δ_p map, with $\delta_p = \partial_{p-1}^\top$ 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 bases of chain spaces. 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. Scalars from $\{0, 1\}$ are used for non-oriented complexes.

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

It is possible to show 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 representations of solids. In this paper we apply LAR methods to computation of boundary representations of solid models from segmentation (labeling) of 3D medical images. To display a triangulation of boundary faces in their proper position in space, the information required is contained in the *geometric chain complex* (GCC):

$$\mu : C_0 \to \mathbb{E}^3,\ (\delta_0, \delta_1, \delta_2) \qquad \equiv \qquad \texttt{(geom, top) = (V, (EV, FE, CF))}$$

Note that ordered pairs of letters from `V,E,F,C`, correspond to `V`*ertices*→*Edges*→`F`*aces*→`C`*ells* into the `C`*olumn*→`R`*ow* order of matrix maps of linear operators. The GCC allows to transform the (possibly non connected) boundary 2-cycle of manifold surfaces as a standard B-rep [22]. The geometry `geom` is given by the embedding matrix `V` of vertices (0-cells); the topology `top` by the three sparse matrices (`EV`, `FE`, `CF`) of coboundaries $(\delta_0, \delta_1, \delta_2)$ of the chain complex describing a space arrangement [19].

Construction of boundary matrix ∂_d

First, let us fix an ordering for the cells of a partition of input data, i.e., the matrix of vertex coordinates `V`, and the arrays of arrays of vertex indices specifying edges `EV`, pixels `FV`, and voxels `CV`. i.e. for each 0-, 1-, 2-, and 3-elements of a cell partition `V,E,F,C` of a 3D image. These orderings fix the p-bases for the linear spaces C_p of p-chains ($0 \leq p \leq 3$). The matrix $M_p = (m_{i,j})$ is called the *characteristic matrix* of the p-basis, where each p-cell is expressed as a subset of vertices (0-cells), so that $m_{i,j} = 1$ if and only if the j-th 0-cell c_0^j belongs to the boundary of i-th p-cell $m_{i,j}$, and $m_{i,j} = 0$ otherwise.

The computation of a boundary matrix $[\partial_p]$ begins by computing the compatible product of the two characteristic matrices $M_{p-1}M_p^t$. Let us note that the product of binary matrices is not binary, so by computing

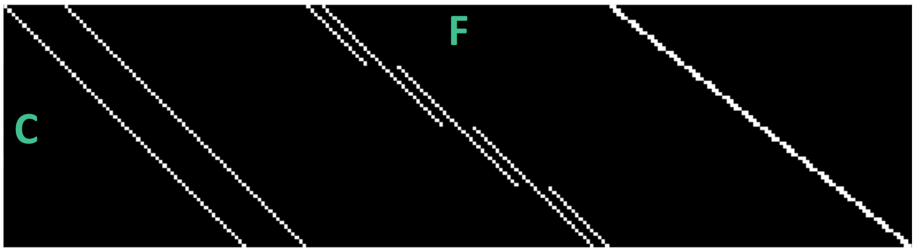


Figure 1: The binary image of *sparse* coboundary matrix $[\delta_2] = [\partial_3]^t : C_2 \to C_3$, built for a small volumetric data (or a brick) with shape $(4,4,4)$. Note that the number of rows equates the size $4 \times 4 \times 4 = 64$ of the voxel set; the number of columns is $d\,n\,(1+n)^{d-1} = 3 \times 4 \times 25 = 300$. Of course, the number of non-zeros per row (cardinality of the facet set of a single voxel) is six, whereas the number of non-zeros per column is generally two, but on boundary facets. The letter F stands for *Faces*, on matrix columns, and the C stands for *Cells* (3-cells) on matrix rows.

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 volume imaging data, made by cubic 3-cells and square 2-cells between adjacent pairs of 3-cells, everywhere we get $n_{i,j} = 4$, we may state $c_2^i \subset \partial c_3^j$. Therefore, in each j column of $M_2M_3^t = (n_{i,j})$, implementing the map `FC : CV` $\to$ `VF`, we have exactly *six rows* where $n_{i,j} = 4$, since a cube (3-chain) has six boundary faces (a 2-chain) on its boundary. Of course, it will contain six non-zero elements for column. It may be worth remembering every 3-cell (voxel) of the volumetric data has exactly six 2-faces.

Finally, consider the linear boundary operator $\partial_p : C_p \to C_{p-1}$. As such, it contains by columns the representation of domain basis elements, expressed as a linear combination of the basis elements of the range space. Therefore, the operator matrix $[\partial_p]$ is readily obtained by setting

$$[\partial_p](i,j) = 1 \quad \text{iff} \quad n_{i,j} = \sum_k m_{i,k}m_{k,j} = \#c_{p-1}^i \quad \text{and} \quad [\partial_p](i,j) = 0 \quad \text{otherwise.}$$

Look for this purpose at the rows of matrix $[\partial_3]^t$ in Figure 1, where the matrix is transposed for better shape ratio. The unit incidence coefficients in $[\partial_3]$, shown as white pixels, are accordingly located by filtering the $n_{i,j}$ elements with value 4. All the other matrix element are set to 0, shown as black pixels.

It is possible to show that *all* the interesting relations of incidence/adjacency between cells of different dimensions can be both computed and efficiently queried by pairwise computing some matrix products, with one of terms possibly transposed, using only the two boundary and coboundary operator matrices $[\partial_p]$ and $[\delta_p]$, and where $[\delta_p] = [\partial_p^\top]$. 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 small brick of shape $(4,4,4)$ is given in Fig. 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. In Julia, our algorithms are most efficiently written in terms of a single linear index `A[i]`, even if `A` is multi-dimensional. Regardless of the array's native indices, linear indices always range from `1:length(A)`.

2.4 Taubin Smoothing

Brick joining Every boundary chain extracted from an image block $\mathbb{B}(i,j,k,n)$ is a *2-cycle*, i.e., a closed 2-chain, defined as a 2-chain with empty boundary. Such 2-cycles are joined together by removing the boundary artifacts, i.e., the double 2-cells at the boundaries of adjacent bricks, after having suitably shifted their indices to an unique linear representation of the whole surface. The resulting raster surface is made by mutually orthogonal raster facets, and must be smoothed in order to get a fair surface.

Surface smoothing A linear time and space algorithm for this purpose is the Laplacian smoothing, which iteratively moves each vertex (0-cell) to the centroid of its neighbors. A well known weakness of this simple algorithm is the asymptotic convergence of the whole mesh to a single point, resulting in unfair size reduction even after few iterations. Conversely, the Taubin smoothing algorithm [27, 28] alternates two Laplacian smoothing steps with *shrink* and *inflate* effects respectively, with the result of delivering pretty invariant sizes and volume of the smoothed mesh. The best results are obtained on meshes which have small variations of edge length and face angles, like for surfaces extracted from 3D raster images, as in our case.

3 Brick-parametric design

In this section we discuss how the input 3D image is handled in order to exploit a data parallelism at brick-level. Specifically, we deal with the image decomposition via the `brick` operator, the brick-to-boundary mapping, and the embedding of the extracted (boundary) 2-cycles from local to global coordinates, in order to merge togheter the various surface patches.

3.1 Brick decomposition

Let us assume that medical devices produce 3D images with lateral dimensions that are integer multiples of some powers of two, like 128, 256, 512, etc. Therefore, any cuboidal portion of the image is completely determined by the Cartesian indices of its voxels of lowest and highest indices, and is 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 block image $\mathbb{B}$, called *brick*, as function of the element of lowest coordinates $i,j,k \in [0\!:\!n-1]$ and of block lateral size $n \in \mathbb{N}$:

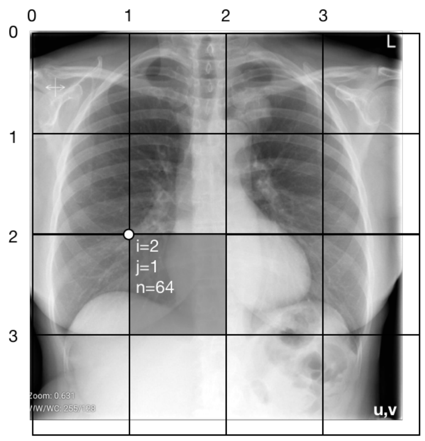
$$\mathbb{B}(i,j,k,n) := image(i\!:\!i+n,\ j\!:\!j+n,\ k\!:\!k+n)$$

Figure 2a shows the brick decomposition in a 2D image, with positive integers (u,v) giving the lateral sizes of image. Note that brick sides do not necessarily correspond to image edges.

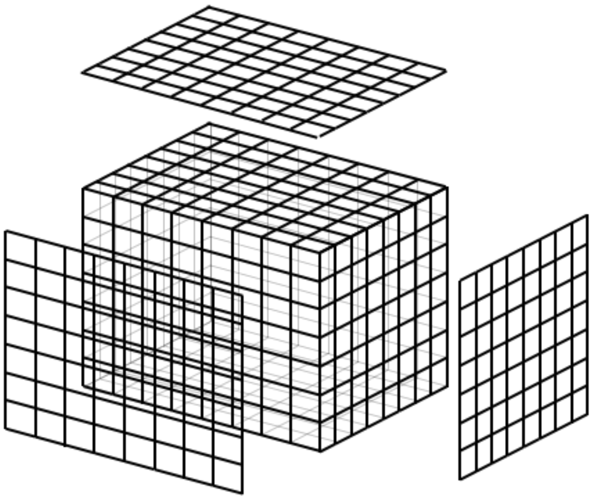
3.2 Brick operator

Chain coordinates We are going to treat each image brick independently from each other. Hence we map each image brick $\mathbb{B}(i,j,k,n)$ to the linear *chain* space C_3 of algebraic dimension $n\times n\times n$, using coordinate vectors $[c] \in \{0,1\}^{n^3}$, where each voxel (basis element $c \in C_3$) is mapped via Cartesian-to-linear map to a vector with n^3-1 zeros and only one element 1. In other words, each voxel in a brick image will be seen as a basis binary vector, and (more interestingly) every subset of brick elements, as the corresponding binary vector (sum of included basis vectors), with many ones as the cardinality of the considered subset.

Boundary operator For a fixed brick 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 [17], and inlined in the generated boundary extraction code.



(a) A possible brick partitioning of a radiologic image. The evidenced 2D brick, of size $n^d = 64^2$, is sliced by $\mathbb{B}([2,1,64]) = Image(128{:}172, 64{:}128)$



(b) Faces on the brick boundary

Figure 2: Brick decomposition

It is easy to see that all matrices $[\partial_d]$ are *very sparse*, since they contains $2d$ non-zeros (ones) for each column (of length n^d), i.e. 4 ones or 6 ones per columns for the 2D and 3D case, respectively. We remind that the matrix of a linear operator between linear spaces contains by columns the basis element of the domain, 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 cells, i.e. either a 1-cycle of 4 edges (2D), or a 2-cycle of 6 faces (3D), 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 of 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 the basis by adjoining n^{d-1} boundary elements for each of the d dimensions of the brick, 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 $d\,(n^{d-1}+n^d) = d\,n^{d-1}\,(1+n)$. The number of columns of $[\partial_d]$ equates the number of basis elements of C_d, i.e. the number n^d of brick elements.

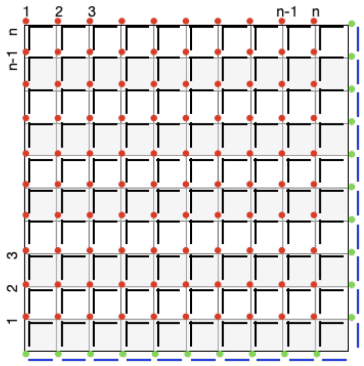


Figure 3: Mapping between 2-cells and 1-cells, used to compute the size of $[\partial_2] : C_2 \to C_1$ for a 2D brick of size $n^d = 10^2$. The dimensions of the chain spaces are: $\dim C_2 = n^d = 100$, $\dim C_1 = dn^d + dn^{d-1} = 200 + 20$.

Sparsity and size of boundary matrix As we have seen, we have $2d$ non-zero elements for each column of $[\partial_d]$ matrix, so that the total number of non-zeros is $2d\,n^d$. The number of matrix element is given by the number of rows ($(d-1)$-cells), times the number of elements per row, i.e., $d\,n^{d-1}\,(1+n) \times n^d$, giving $0 \leq sparsity \leq 1$, with:

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

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

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

In conclusion, for brick size $n = 64$, the matrix $[\partial_d]$ requires for 2D bricks $9 \times 64^2 = 36,864$ memory elements, and for 3D bricks $13 \times 64^3 = 3,407,872$ 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, so requiring 180 KB for 2D brick and 12 MB for 3D brick.

3.3 Brick-to-boundary mapping

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

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

Then, we consider each segment portion $c_{i,j,k} = S \cap \mathbb{B}(i,j,k,n)$ and compute its coordinate vector local to the brick $[c] \in C_3(n^3)$. This one is a sparse binary vector of length n^3. Then, assemble by columns the M representations $[c]_j$ $(1 \leq j \leq M)$ of segment portions into a sparse binary matrix $\mathbf{S}$, of dimension $n^d \times M$, with $d = 3$. Finally, compute a matrix **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 $dn^{d-1}(1+n) \times n^d$. Of course, the **B** sparse matrix has the same column number M of $\mathbf{S}$, because each column contains the boundary representation of the corresponding $S \cap \mathbb{B}(i,j,k,n)$, and the number of matrix rows is the dimension $d\,n^{d-1}\,(1+n)$ of the linear space C_2.

Embedding Two final computational steps are required, to embed the 2-chains in $\mathbb{E}^3$ space, 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 with the vertices of the extracted surface. The simplest solution is to associate four vertices to each 2-cell of the extracted surface, i.e. to each non-zero entry in every column of **B**. The μ function can be computed by identifying, via element position in the column, a triple of integer values $0 \leq x \leq u$, $0 \leq y \leq v$, and $0 \leq z \leq w$ 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 Cartesian to linear coords, in order of not duplicating the output vertices.

3.4 Brick-level parallelism

In the computational pipeline discussed in this paper, several steps can be efficiently performed in parallel at the image-brick level, depending on the embarrassingly data-parallel nature of the problem. In particular,

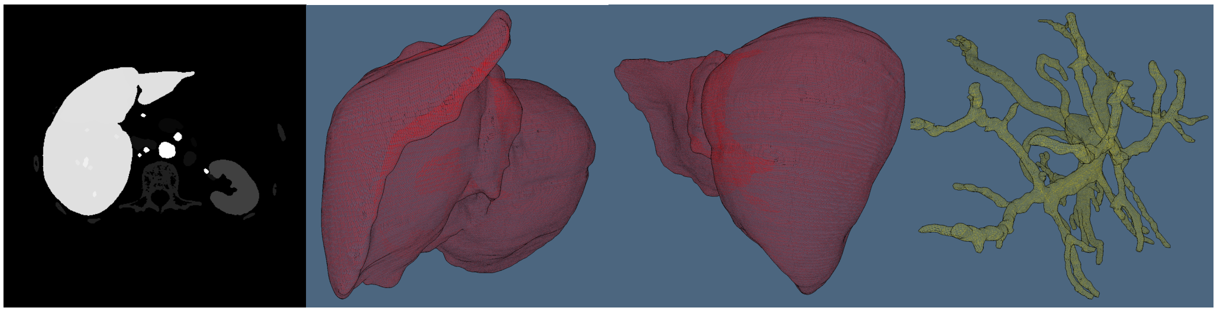


Figure 4: The left image shows one slice with segmented organs from the Ircad dataset [25]. The other three images show the surface of the liver and portal vein generated by `lar-surf.jl` package.

little effort is needed to separate the problem into several parallel tasks $S_{i,j,k}$, using multiarray slicing. The granularity of parallelism, depending on the brick size n, is further enforced by the computation of a single boundary matrix $[\partial_d(n)]$, function only of brick size n, so that the initial communication cost of broadcasting the matrix to compute nodes can be carefully controlled, and finely tuned depending on the system architecture. The whole approach is appropriate for SIMD (Single Instruction, Multiple Data) hybrid architectures of CPUs and GPUs, since only the initial brick setup of boundary matrix and image slices, as well the final collection of computed surface portions, require inter-process communication.

4 Julia implementation

The computer code is implemented in Julia language [3] according to the workflow described below, whose stages are parallelized and/or optimized in various ways. The workflow scheme can be seen on Fig. 7. The implementation is available on GitHub [11] and our `LarSurf.jl` package can be installed using a standard Julia package register.

4.1 Parallel workflow

Workflow setup The functions in this preliminary step include:

1. the input of 3D medical image $\mathcal{I}$ with *shape* (ℓ_1, ℓ_2, ℓ_3), such that: $\mathcal{I} = [\ell_1] \times [\ell_2] \times [\ell_3]$, where $[\ell_k] = [1, 2, \dots, \ell_k]$;
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 the *size* of 3D image brick $\mathcal{B}$. With default *size* $= 64$, 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 the default *number* of bricks, of size 64^3, is $m = 8 \times 8 \times 4 = 256$, for standard medical images $512 \times 512 \times 256$;
4. computation of sparse brick boundary matrix $[\partial_B]$, where `Int8` and `Int64` are the types for values and indices, returning a sparse matrix value of type `SparseMatrixCSC{Int8}{Int64}`, stored by Compressed Sparse Column (CSC) format. The storage of $[\partial_B]$ (for *n* = 64) requires about 12 MB;
5. creation of either a local or distributed `channel` to implement a producer/consumer model of parallel/distributed computation, depending on available resources;

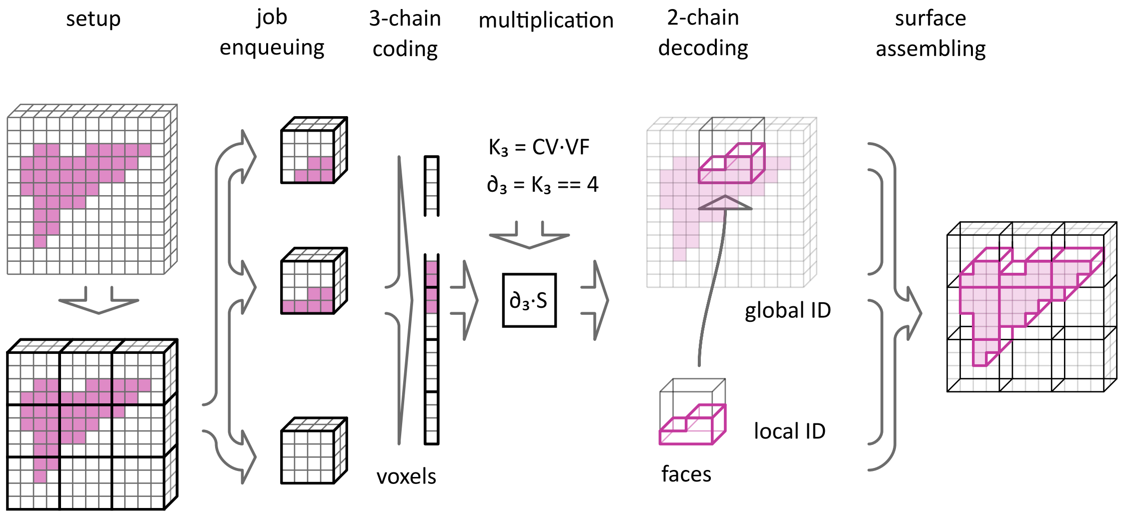


Figure 5: Workflow of LAR-SURF algorithm

6. distribution of matrix $[\partial_B]$, of default size 12 MB, to all available nodes/cores (Julia workers), using the Julia macro `@eveywhere`.

Job enqueuing Communication and data synchronization are managed through *Channels*, which are the FIFO conduits that may 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 of this stage follow:

1. extraction, from image arrays of the block views, depending on 3 Cartesian indices;
2. transform each block *from global* $[\ell_1] \times [\ell_2] \times [\ell_3]$ to *local coordinates* $[n] \times [n] \times [n]$;
3. further transform of each *foreground voxel* $\nu \in \mathcal{S} \subseteq \mathcal{I}$ from Cartesian to linear coordinates, using the suitable Julia's library functions.
4. enqueuing the job (as a sequence of integer positions for the non-zeros image elements aligned in a memory buffer of proper `Channel` type).

3-Chain encoding The interesting part of the *Image* $\mathcal{I}$ is called *Segment* $\mathcal{S}$. The goal of the whole *workflow* is to extract a *boundary model* of $\mathcal{S}$ from $\mathcal{I}$. The portion of $\mathcal{S}$ inside $\mathcal{B}$, will be denoted as $\mathcal{S}(B)$. Each block $\mathcal{B}$ of the 3D image must by converted into the *coordinate representation* of a vector $\nu \in C_3$ in the linear space of 3-chains.

In coordinates local to $\mathcal{B}$, once fixed an ordering from Cartesian to linear coordinates, this vector is represented by a *binary array* of length $size^3$. With $size = 64$, we have $64^3 = 262144$, with a non-zero value (i.e. 1) for each foreground voxel in $\mathcal{S}(B)$. Therefore, the coded segment portion $\mathcal{S}(B)$ results with a space occupancy of about 262 KB if encoded as a full array (i.e. including the zero values). Whether encoded as a sparse vector, its space occupancy will correspondingly decrease.

1. each encoding task produces either a full or sparse binary vector. With full or sparse arrays depending by one index, we get either 262 KB or less per job, correspondingly;
2. special format for sparse CSC (Compressed Sparse Column) vectors can be used, since the *value* data for non-zeros does 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 $8 \times \texttt{nnz}$ kB storage) is needed;
3. prepare subsequences of such data vectors (non-zero linear row indices), in order to feed efficiently the available processor threads. In case of the 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 [4] 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 default setting of the algorithm (thize of the block 64^3; the size of the image $512^2 \times 256$), or more in case of either smaller blocks or image greater than the standard one, this stage must be evidently parallelized and carefully tuned, possibly by using the GPU, if available.

1. the total speed of this stage is strongly dependent on the hardware available, on the granularity of bricks, and on the choice between dense/sparse storage of encoded 3-chains;
2. the compute elements or threads is fed without solution of continuity in a *dataflow* process. This parallel operation is, according to our preliminary experiments, the critical one of the whole workflow, since any ΔT (either positive or negative) in this stage contributes 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 ones, depending on the size of the memory buffers used to feed the CPU cores or the GPUs and get their results. Its elementary steps follow:

1. reading of position of ones (non-zeros) in the 2-chain as linear indices of rows;
2. conversion from linear indices to Cartesian indices in coordinates local to the $\mathcal{B}$ 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.

Julia's vectorized pipeline data-flow was the more appropriate implementation model for the job of workers.

Assembling and artifact filtering The results of the previous stages can be described as a *collection of sets* of *geometric quadrilaterals (quads)*, each one encoded as an array of quadruples of integer indices, pointing to the linear array of grid vertices associated to the image block $\mathcal{B}$. In other words, *all quads* of **each job** are now given in the **same** *local coordinates*. Besides putting each partial surface $\mathcal{S}(B) = (\texttt{V}_B, \texttt{FV}_\sigma)$ 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 $\mathcal{B}$:

1. translate each array $\texttt{FV}_\sigma$, of type `Lar.Cells`, by summing to each vertex index the linearized offset of the Cartesian coordinates (i, j, k) of the $\mathcal{B}$'s *reference vertex*, i.e. the one with lowest *Cartesian coordinates* within the $\mathcal{B}$ block.

2. remove both instances of *double quads* generated by `Lar` software at the block boundaries (see Fig. 2b). 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 was used, which does not require any sorting nor searching on the assembled array of quads. The output set of 2-cells (or triangles) is just rewritten, discarding the elements pointing to all vertices with one coordinate equal to a block boundary. A proper software filter was applied for this purpose.

Smoothing The final smoothing of the generated surfaces cannot be performed block-wise since this would introduce smoothing artifacts at the block boundaries. Anyway, the Taubin smoothing [26] 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 consists in computing a new position as a proper average of its neighborhood vertices, i.e. by applying a discrete Laplacian operator. Some appropriate sets of workers may so be assigned 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 type `Channel`;

2. vectorized computation of proper averages of near vertices;

3. job dequeuing, by recovering finished tasks from a channel and assembling the results into the embedding function $\texttt{V} : C_0 \to \mathbb{E}^3$, providing an array of type `Lar.Points` of `Float64` $\times$ 3, with vertex coordinates by column.

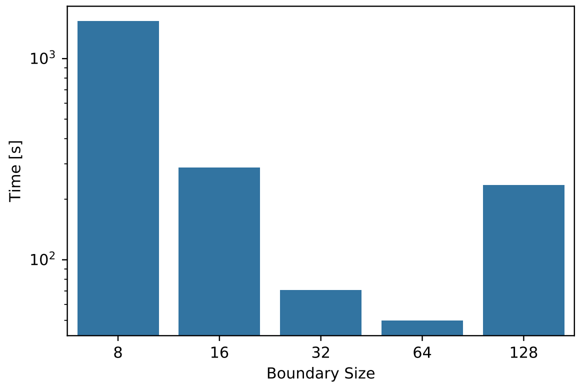
4.2 Performance analysis

Boundary matrix size The size of the boundary matrix is a critical parameter of the LAR-SURF method. To determine optimal size of boundary matrix the experiment on artificial data was performed (Fig. 6a). The size of experimental data is set to $512 \times 512 \times 512$ and it is derived from a typical size of Computed Tomography medical images. Computation is done on the Tesla DGX-1 machine.

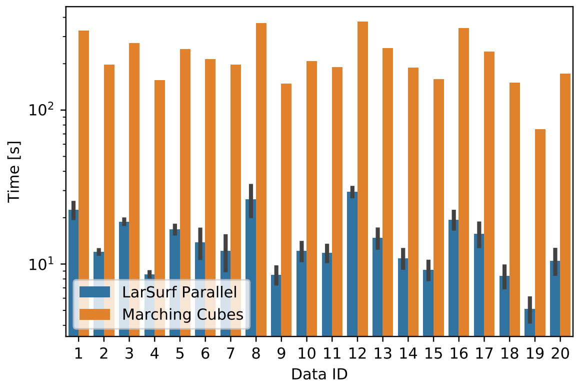
According to the experiment the fastest computation is with the boundary matrix with size $64 \times 64 \times 64$. This is an expected result. The larger boundary matrix is too big to fit in CPU's cache memory.

Comparison with the Marching Cubes algorithm To compare the time requirements of LAR-SURF with Marching Cubes implemented in Python we performed an experiment on Ircadb dataset [24]. Dataset contain 20 Computed Tomography images (see table 1) with xy-resolution from 0.56 mm to 0.87 mm and z-resolution from 1.0 mm to 4.0 mm. The number of slices is each series varies from 74 to 260 and the size of each slice is 512×512. The dataset contains manually segmented liver, portal vein, and other structures. We performed surface extraction of the liver with Marching Cubes and LAR-SURF. The time required for computation can be seen in Fig. 6b.

Based on the t-test with $\alpha = 0.99$, $p = 8.735 \times 10^{-24}$ and $s = -16.67$ it can be shown that the mean of time consumed by LAR-SURF is significantly lower from time consumed by Marching Cubes.



(a) Time requirements of the LAR-SURF filter used on artificial volumetric data with various sizes of boundary matrix



(b) Time requirements of the LAR-SURF filter and Marching Cubes on the Ircad dataset. Error bars show the 95% confidence interval

Figure 6: Performance analysis of the LAR-SURF filter.

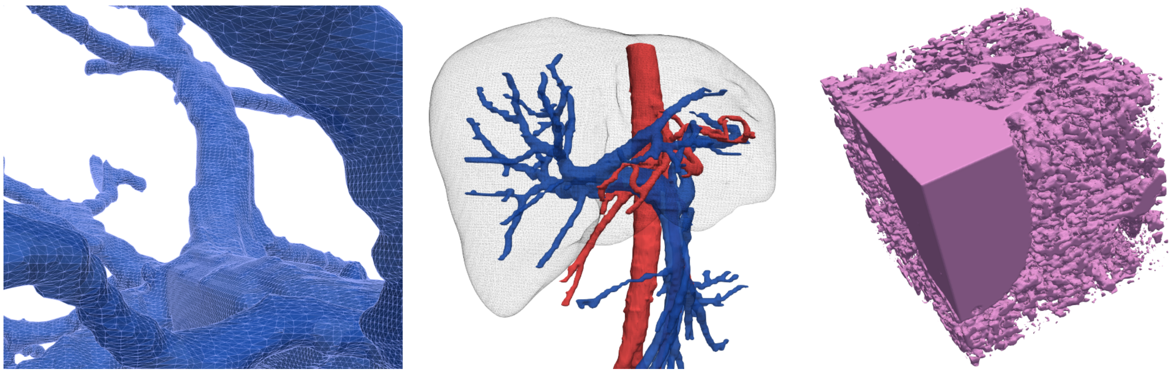


Figure 7: Triangulated surfaces of the macroscopic and microscopic structures of the liver extracted by the LAR-SURF algorithm. (a) Detail of the portal vein (PV) with resolution (??) $1.6 \times 0.57 \times 0.57$ $[mm]$. (b) model of the human liver, with portal vein and hepatic artery connected to colon and stomach. (c) microvasculature of a pig liver, based on corrosion cast of interlobular veins prepared by Eberlova [10]. The size of the specimen is 0.936 $[mm]$ along each axis and the resolution of the Micro-CT data is 4.682 μm. Note that brick boundaries look flat.

5 Examples

Use of the `LarSurf.jl` package can be seen on listings 1 where the liver segmentation with 2865131 voxels from the Ircad dataset is used as an input for our surface extraction algorithm. The size of 3D volumetric image is $129 \times 512 \times 512$ and the voxel resolution is $1.6 \times 0.57 \times 0.57$ [mm]. The output liver surface model is formed by 182124 triangles and the number of vertices is 90822. The visualization can be seen in Fig. **??**.

Listing 1: Get surface from DICOM volumetric data

```
using Distributed
using Pio3d  # Read 3D data from DICOM files
addprocs(3)  # set number of processors
using LarSurf

LarSurf.lsp_setup([64, 64, 64])  # set block size

# read data from DICOM files
datap = Pio3d.read3d("3Dircadb1.1/MASKS_DICOM/liver")
segmentation = datap["data3d"]
voxelsize_mm = datap["voxelsize_mm"]

# get surface
V, FV = LarSurf.lsp_get_surface(segmentation, voxelsize_mm)
FVtri = LarSurf.triangulate_quads(FV)

# do smoothing and save data
Vs = LarSurf.Smoothing.smoothing_FV_taubin(V,FV,0.5,-0.2,40)
objlines = LarSurf.Lar.lar2obj(Vs, FVtri, "liver.obj")
```

The portal vein surface extraction can be performed with a small change of input path in code. The 3D image resolution is the same. The number of input voxels is 103533. The output surface is created by 90822 vertices and 182124 triangles.

The right bottom image of Fig. **??** is the surface of the microvasculature of pig liver. The volumetric image is based on Micro-CT data of corrosion casts of pig liver. [10]. The size of the visualized data is $100 \times 100 \times 100$ voxels and the size of the voxel is 4.682 μm. The number of triangles is 544784 and the number of vertices is 272826.

6 Discussion of method

The greatest amount of other methods for extraction of boundary surfaces from 3D data arrays—including [29] and [30]—use implicit functions, defined by first averaging upon 3D mesh vertices the lighting or brightness of incident voxels, and then by applying some marching cube algorithm in order to traverse and to triangulate the iso-valued boundary patches.

Conversely, we use a binary labeling of the voxel sets of interested image segment, and the standard medical 3D image array as solid representation. The set of boundary facets is extracted through spMV (sparse matrix vector multiplication) of the $[\partial_3]$ matrix times the binary vector labeling the voxels of the segment. The boundary matrix is computed once and for all, sent to all workers (cores or nodes), then used in parallel for all image brick extractions. This algebraic method can be immediately extended to multi-material processing, as well as to design multi-material tissue layering for 3D printing.

In particular, we might organize a multi-material (or multi-organ) extraction, simply by multiplication of the boundary matrix $[\partial_3] : C_3 \to C_2$ times a binary matrix where the non-zeros elements on each column represents either one of the materials, or one of the organs to be algebraically extracted by the filter. The two surface patches common to two adjacent segments will be exactly coincident, and share the same geometry and topology, with the only exception of each patch contours, where the influence of adjacent patches induces each instance to be separated and round-off.

The main advantage of the approach discussed in this paper is given by its very algebraic nature; our implementation does not need any kind of algorithmic graph traversal, of difficult parallelization, but only the execution of standard algebraic computing kernels, and in particular the spMV and the spMspM (sparse matrix–sparse matrix multiplication) kernels, nowadays efficiently implemented by tensor processors on Nvidia hardware. As we have shown, the parallelization, both shared-memory and distributed, is also fairly easy and with good speed-up.

7 Conclusion

We introduced a Julia implementation of an algebraic filter to extract from 3D medical images the boundary surface of some specific image segment, described as a 3-chain of voxels. Translations from Cartesian indices of cells to linearized indices, the computation of the sparse boundary matrices, and the sparse matrix-vector multiplication are the main computational kernels of this approach.

The implementation of the LAR-SURF filter is available in the open-source repository and it can be installed using standard Julia package manager [11].

We showed a good speed-up over marching-cubes algorithms. The existing implementation employs Julia's channels for multiprocessing. Our performance experiment showed an optimal size of the brick size. Parallelization makes a large portion of spared computational cost. Moreover, we expect additional improvement in the future because our approach is appropriate for SIMD (Single Instruction, Multiple Data) 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.

Currently, the computational pipeline is being strongly improved to gain a greater speed-up using native Julia implementation `CUDA.jl` of Nvidia programming platform [2] , and Julia's `SuiteSparseGraphBLAS.jl` framework [4] for graph algorithms with the language of linear algebra. In particular, we are extending its use pattern in order to work with general cellular complexes.

ACKNOWLEDGEMENTS

This work was supported by Charles University Research Centre program UNCE/MED/006 "University Center of Clinical and Experimental Liver Surgery" and Ministry of Education project ITI CZ.02.1.01/0.0/0.0/17_048/0007280: Application of modern technologies in medicine and industry. The research was also supported by the project LO 1506 of the Czech Ministry of Education, Youth and Sports.

A Appendix

Here we provide, for the sake of readers, a list of symbols used in the paper, and a small set of related definitions. A description of public dataset used in the experiments, and a coding example of computation of the sparse $[\partial_3]$ boundary matrix in Julia is finally given.

1.1 Symbol list

$\mathcal{I}$ three-dimensional medical image
ℓ_1, ℓ_2, ℓ_3 dimensions of image
$\mathcal{S}$ segment: a subset of voxels from image segmentation
$\mathcal{B}$ 3D image brick

$size$ lateral dimension of cubic brick $\mathcal{B}$
n number of bricks $\mathcal{B}$ (jobs) in $\mathcal{I}$
[$\partial_{\mathcal{B}}$] boundary matrix for brick $\mathcal{B}$
C_p linear (vector) space of p-chains
$\nu \in C_p$ p-chain
[ν] coordinate representation (binary vector) of ν
$\mathbb{E}^3$ Euclidean 3-space

1.2 Definitions

Boundary model Closed manifold surface of the boundary of a solid model
CSC Compressed Sparse Column format for sparse matrices
Global coordinates Integer linear coordinates of $\mathcal{I}$
Local coordinates Integer linear coordinates of $\mathcal{B}$
Cartesian coordinates Integer triples (i, j, k) one-to-one with voxels
Voxels Individual elements in 3D image (3-cells)
p-**chain** Formal linear combination of p-cells with coefficients in $\{0, 1\}$
Coord. repr. Binary vector (for p-chains) or binary matrix (for chain operators)
Quad Geometric quadrilateral; convex polygon with four vertices
Foreground voxel Individual element of a segment $\mathcal{S}$
Segment Subset of voxels resulting from image segmentation

1.3 3D-Ircadb Dataset

For perfomance analysis the public dataset from Research Institute against Digestive Cancer (IRCAD) [24] was used. The table 1 describe the dataset.

	z-resolution [mm]	xy-resolution [mm]	obj. voxels	size xy	size z
min	1.00000	0.561000	5.832080e+05	512.0	74.000000
mean	1.77750	0.725141	1.894777e+06	512.0	141.150000
50%	1.60000	0.739094	1.760604e+06	512.0	127.000000
max	4.00000	0.873047	3.341433e+06	512.0	260.000000

Table 1: Ircad dataset description [24]. It contains 20 Computed Tomography images of abdomen with manually segmented tissues.

1.4 Basic operations in LAR

Boundary matrices for grids of cubes:

We give here the full Julia code for the algebraic computation of ∂_3 matrix, for a very little grid of unit 3-cubes. Due to the simplicity of the cells (voxels = cubes), a sufficient (geom,top) pair is given below as (`V,CV`), where `CV` is an array of arrays of `Int64` indices of vertices of grid cubes.

```
julia> using LinearAlgebraicRepresentation, SparseArrays
julia> Lar = LinearAlgebraicRepresentation
julia> V, CV = Lar.cuboidGrid([3,2,1])
julia> V
0.0 0.0 0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 3.0 3.0 3.0 3.0 3.0 3.0
0.0 0.0 1.0 1.0 2.0 2.0 0.0 0.0 1.0 1.0 2.0 2.0 0.0 0.0 1.0 1.0 2.0 2.0 0.0 0.0 1.0 1.0 2.0 2.0
0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0
julia> CV
[[ 1, 2, 3, 4, 7, 8, 9,10], [ 3, 4, 5, 6, 9,10,11,12], [ 7, 8, 9,10,13,14,15,16],
 [ 9,10,11,12,15,16,17,18], [13,14,15,16,19,20,21,22], [15,16,17,18,21,22,23,24]
```

Face and Edge Data generation:

In the following, we provide the functions for generating the face data `FV` (vertex indices in faces) with function `CV2FV` and edge data `EV` (vertex indices in edges) with function `CV2EV` from cell data `CV`.

```
function CV2FV( v:: Array{ Int64 } )
    return faces = [[v[1], v[2], v[3], v[4]], [v[5], v[6], v[7], v[8]],
                    [v[1], v[2], v[5], v[6]], [v[3], v[4], v[7], v[8]],
                    [v[1], v[3], v[5], v[7]], [v[2], v[4], v[6], v[8]]]
end
function CV2EV( v:: Array{ Int64 } )
    return edges = [[v[1],v[2]], [v[3],v[4]], [v[5],v[6]], [v[7],v[8]], [v[1],v[3]], [v[2],v[4]],
                    [v[5],v[7]], [v[6],v[8]], [v[1],v[5]], [v[2],v[6]], [v[3],v[7]], [v[4],v[8]]]
end
```

Characteristic matrices:

The function `K` transforms an array of arrays ($\mathtt{VV}, \mathtt{EV}, \mathtt{FV}, \mathtt{CV}$) into a sparse binary characteristic matrix ($\mathtt{M}_0, \mathtt{M}_1, \mathtt{M}_2, \mathtt{M}_3$). A Julia sparse matrix needs three arrays I, J, Vals of rows, columns, values of non-zeros:

```
VV = [[v] for v=1:size(V, 2)];
FV = collect(Set{Array{Int64,1}}(vcat(map(CV2FV, CV)...)))
[[13,15,19,21], [1,2,3,4], [7,9,13,15], [13,14,15,16], [7,8,13,14], [1,2,7,8], [2,4,8,10], [7,8,9,10],
 [3,5,9,11], [8,10,14,16], [15,16,21,22], [9,11,15,17], [3,4,5,6], [17,18,23,24], [11,12,17,18],
 [1,3,7,9], [3,4,9,10], [9,10,15,16], [4,6,10,12], [13,14,19,20], [9,10,11,12], [15,16,17,18],
 [19,20,21,22], [15,17,21,23], [16,18,22,24], [21,22,23,24], [10,12,16,18], [5,6,11,12], [14,16,20,22]]

EV = collect(Set{Array{Int64,1}}(vcat(map(CV2EV, CV)...)))
[[15,17], [16,22], [6,12], [17,23], [18,24], [4,10], [3,4], [13,15], [11,12], [9,15], [13,19],
 [1,7], [5,11], [5,6], [12,18], [8,14], [15,21], [17,18], [1,3], [2,4], [16,18], [2,8], [21,23],
 [20,22], [1,2], [14,16], [10,16], [13,14], [19,21], [7,13], [9,10], [23,24], [11,17], [21,22],
 [3,9], [3,5], [9,11], [7,9], [14,20], [7,8], [22,24], [19,20], [8,10], [15,16], [10,12], [4,6]]

function K(CV)
    I = vcat( [ [k for h in CV[k]] for k =1: length(CV) ]...);
    J = vcat(CV ...);
    Vals = Int8[1 for k=1: length(I)];
    return SparseArrays.sparse(I,J,Vals)
```

```
end
VV = [[k] for k=1:size(V,2)];
M0 = K(VV); M1 = K(EV); M2 = K(FV); M3 = K(CV);
```

Boundary matrices:

The boundary matrices between non-oriented chain spaces are computed by sparse matrix multiplication followed by matrix filtering, produced in Julia by the broadcast of vectorized integer division (.÷):

```
# This code is working with Julia 1.2
partial_1 = M0 * M1'
partial_2 = div.((M1 * M2'), 2)
s = sum(M2, dims=2)
partial_3 = (M2 * M3') ./ s
partial_3 = div.(partial_3, 1)
```

ORCID

Miroslav Jirik http://orcid.org/0000-0002-8002-2079
Alberto Paoluzzi http://orcid.org/0000-0002-3958-8089

REFERENCES

[1] Bajaj, C.L.; Pascucci, V.; Thompson, D.; Zhang, X.Y.: Parallel accelerated isocontouring for out-of-core visualization. Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics, PVGS 1999, 97–104, 1999. http://doi.org/10.1145/328712.319342.

[2] Besard, T.; Foket, C.; De Sutter, B.: Effective Extensible Programming: Unleashing Julia on GPUs. IEEE Transactions on Parallel and Distributed Systems, 30(4), 827–841, 2019. ISSN 15582183. http://doi.org/10.1109/TPDS.2018.2872064.

[3] Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review, 59(1), 65–98, 2017. http://doi.org/10.1137/141000671.

[4] Buluc, A.; Mattson, T.; McMillan, S.; Moreira, J.; Yang, C.: Design of the GraphBLAS API for C. In Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017, 643–652, 2017. ISBN 9781538634080. http://doi.org/10.1109/IPDPSW.2017.117.

[5] Chernyaev, E.: Marching cubes 33: Construction of topologically correct isosurfaces. Tech. rep., 1995. http://wwwinfo.cern.ch/asdoc/psdir.

[6] Crossno, P.; Angel, E.: Isosurface extraction using particle systems. Proceedings of the IEEE Visualization Conference, 495–498, 1997. http://doi.org/10.1109/visual.1997.663930.

[7] Custodio, L.; Pesco, S.; Silva, C.: An extended triangulation to the Marching Cubes 33 algorithm. Journal of the Brazilian Computer Society, 25(1), 6, 2019. ISSN 1678-4804. http://doi.org/10.1186/s13173-019-0086-6.

[8] Dicarlo, A.; Milicchio, F.; Paoluzzi, A.; Shapiro, V.: Chain-based representations for solid and physical modeling. IEEE Transactions on Automation Science and Engineering, 6(3), 454–467, 2009. ISSN 15455955. http://doi.org/10.1109/TASE.2009.2021342.

[9] Dicarlo, A.; Paoluzzi, A.; Shapiro, V.: Linear algebraic representation for topological structures. Comput. Aided Des., 46, 269–274, 2014. ISSN 0010-4485. http://doi.org/10.1016/j.cad.2013.08.044.

[10] Eberlova, L.; Liska, V.; Mirka, H.; Tonar, Z.; Haviar, S.; Svoboda, M.; Benes, J.; Palek, R.; Emingr, M.; Rosendorf, J.; others: The use of porcine corrosion casts for teaching human anatomy. Annals of Anatomy-Anatomischer Anzeiger, 213, 69–77, 2017.

[11] Jirik, M.; Paoluzzi, A.: LarSurf.jl package on GitHub, 2020. https://mjirik.github.io/LarSurf.jl/.

[12] Lachaud, J.O.; Montanvert, A.: Continuous analogs of digital boundaries: A topological approach to iso-surfaces. Graphical Models, 2000. ISSN 15240703. http://doi.org/10.1006/gmod.2000.0522.

[13] Lorensen, W.E.; Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. ACM siggraph computer graphics, 21(4), 163–169, 1987.

[14] Natarajan, B.K.: On generating topologically consistent isosurfaces from uniform samples. The Visual Computer, 11(1), 52–62, 1994. ISSN 1432-2315. http://doi.org/10.1007/BF01900699.

[15] Newman, T.S.; Yi, H.: A survey of the marching cubes algorithm. Computers and Graphics (Pergamon), 30(5), 854–879, 2006. ISSN 00978493. http://doi.org/10.1016/j.cag.2006.07.021.

[16] Paoluzzi, A.; Dicarlo, A.; Furiani, F.; Jirik, M.: CAD models from medical images using LAR. Computer-Aided Design, 13(6), 2016. ISSN 0010-4485. http://doi.org/10.1080/16864360.2016.1168216.

[17] Paoluzzi, A.; Shapiro, V.; DiCarlo, A.; Furiani, F.; Martella, G.; Scorzelli, G.: Topological computing of arrangements with (co)chains. Transactions on Spatial Algorithms and Systems (ACM TSAS). Accepted for publication.

[18] Paoluzzi, A.; Shapiro, V.; DiCarlo, A.; Scorzelli, G.; Onofri, E.: Finite Boolean Algebras for Solid Geometry using Julia's Sparse Arrays. https://arxiv.org/pdf/1910.11848.

[19] Paoluzzi, A.; Shapiro, V.; DiCarlo, A.; Scorzelli, G.; Onofri, E.: Finite boolean algebras for solid geometry using julia's sparse arrays, 2019. https://arxiv.org/abs/1910.11848. Eprint in https://arxiv.org/abs/1910.11848.

[20] Requicha, A.G.: Representations for rigid solids: Theory, methods, and systems. ACM Comput. Surv., 12(4), 437–464, 1980. ISSN 0360-0300. http://doi.org/10.1145/356827.356833.

[21] Rohan, E.; Lukes, V.; Jonásová, A.: Modeling of the contrast-enhanced perfusion test in liver based on the multi-compartment flow in porous media. Journal of Mathematical Biology, 77(2), 421–454, 2018. ISSN 14321416. http://doi.org/10.1007/s00285-018-1209-y.

[22] Shapiro, V.: Solid modeling. In G. Farin; J. Hoschek; S. Kim, eds., Handbook of Computer Aided Geometric Design, chap. 20, 473–518. Elsevier Science, 2002.

[23] Smistad, E.; Elster, A.C.; Lindseth, F.: Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs. In Norsk informatikkonferanse, 141–152, 2012. http://www.tapironline.no/last-ned/1050.

[24] Soler, L.: 3D-IRCADb-01 dataset, 2016. https://www.ircad.fr/research/3dircadb/.

[25] Soler, L.: 3D-IRCADb-01 (http://www.ircad.fr/research/3d-ircadb-01/), 2016. http://www.ircad.fr/research/3d-ircadb-01/.

[26] Taubin, G.: Curve and surface smoothing without shrinkage, 1995. http://doi.org/10.1109/iccv.1995.466848.

[27] Taubin, G.: A signal processing approach to fair surface design. In Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '95, 351–358. ACM, New York, NY, USA, 1995. ISBN 0-89791-701-4. http://doi.org/10.1145/218380.218473.

[28] Taubin, G.: Geometric Signal Processing on Polygonal Meshes. In Eurographics 2000 - STARs. Eurographics Association, 2000. ISSN 1017-4656. http://doi.org/10.2312/egst.20001029.

[29] Wang, C.C.L.: Direct extraction of surface meshes from implicitly represented heterogeneous volumes. Comput. Aided Des., 39(1), 35?50, 2007. ISSN 0010-4485. http://doi.org/10.1016/j.cad.2006.09.003.

[30] Wang, C.C.L.: Extracting Manifold and Feature-Enhanced Mesh Surfaces From Binary Volumes. Journal of Computing and Information Science in Engineering, 8(3), 2008. ISSN 1530-9827. http://doi.org/10.1115/1.2960489. 031006.