Title:

# Algebraic Filtering of Surfaces from 3D Medical Images with Julia

Authors:
Miroslav Jirik, `mjirik@kky.zcu.cz`, University of West Bohemia
Alberto Paoluzzi, `paoluzzi@dia.uniroma3.it`, Roma Tre University

Keywords:
Medical 3D, Computational Topology, Linear Algebraic Representation, LAR, Julia, Surface Extraction

DOI: 10.14733/cadconfP.2020.xxx-yyy

## Introduction:

In this paper we introduce an algebraic LAR-SURF filter, based on topological methods, to extract and smooth the boundary surface of any subset of voxels arising from segmentation of a 3D medical image.

Isosurface extraction to produce geometric models from volumetric data is important in many aplications. It is often used for indirect visualization of the medical data or for flow modeling and simulation. Here we discuss an approach based on basic algebraic topology and linear algebra, using linear spaces $C_p$ of chains (of cells) of dimension $0 \leq p \leq 3$ and the boundary matrix $[\partial_3] : C_3 \to C_2$.

The input image segment is defined as a *chain*, i.e. as a vector from a linear space $C_3$ of 3-chains, represented in coordinates as a sparse binary vector.

A decomposition of the 3D image into small submatrices called *bricks* is performed, then the binary coordinate vector of each interesting chain of voxels is generated, and its boundary is computed by matrix multiplication times the boundary matrix producing the binary representation of the boundary surface.

The output boundary surface is produced by a linear mapping between spaces of 3- and 2-chains provided by the boundary operator $\partial_3 : C_3 \to C_2$. In particular, when the input set of voxels is either non 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.

An embarrassing parallel data decomposition is used to compute the boundary surface patches within each brick, that are finally joined and smoothed via the Taubin algorithm [**?** ]. This work is based on LAR (Linear Algebraic Representation) methods [**? ?** ], and is implemented in Julia, natively supporting parallel (distributed or CUDA) computing on hybrid hardware architectures.

## Representation Scheme:

With *Boundary representations* ("*B*-reps"), a solid model is represented through its boundary elements, i.e. faces, edges and vertices; *decompositive/enumerative representations* [**?** ], are a decomposition of either the object's or the embedding space, respectively, into a *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.

The *Linear Algebraic Representation* (LAR), introduced in [? ], 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*:

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

and, in particular, all linear boundary/coboundary maps $\partial_p$ and $\delta_p = \partial_{p-1}^\top$ between chain spaces $C_p$.

The *domain* of LAR is the set of *chain complexes* generated by cell $p$-complexes $(2 \leq p \leq 3)$. The computer data structures 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. 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 $[\partial_p][c_p]$.

To display a triangulation of boundary faces in their proper position in space, the whole information required is contained in the *geometric chain complex*: `(geom,top) = (V,(EV,FE,CF))`. This one is equivalent to the pair made by the embedding function $\mu : C_0 \to \mathbb{E}^3$ and by the coboundary chain $(\delta_0, \delta_1, \delta_2)$, where the geometry `geom` is given by the embedding matrix `V` of vertices (0-cells), and the topology `top` by the three sparse matrices (`EV`, `FE`, `CF`) of coboundaries $(\delta_0, \delta_1, \delta_2)$ of the chain complex.

## Construction of boundary matrix $\partial_p$:

First, let us fix an ordering for all the cells of a partition of a 3D image (with vertices `V`, edges `E`, pixels `F`, and voxels `C`), i.e. for every 0-, 1-, 2-, and 3-cell. These orderings define the $p$-bases for the linear spaces $C_p$ of $p$-chains $(0 \leq p \leq 3)$. We call $\mathtt{M}_p = (m_{i,j})$ the binary *characteristic matrix* of the $p$-basis, where elementary $p$-chains (singleton $p$-cells) are expressed by rows as $C_0$ subsets, so that $m_{i,j} = 1$ if and only if the $j$-th 0-cell $c_0^j$ belongs to the boundary of the $i$-th $p$-cell $c_p^i$, and $m_{i,j} = 0$ otherwise.

Let us note that the product of binary matrices is not binary, so by computing the (sparse) matrix product $\mathtt{M}_{p-1}\,\mathtt{M}_p^t = (n_{i,j})$, where $n_{i,j} = \sum_k m_{i,k} m_{k,j}$, we get for each $n_{i,j}$ the *number of vertices* shared by $c_{p-1}^i$ and $c_p^j$. When this number equates the cardinality of $c_{p-1}^i$, such elementary chain is contained in the boundary of $c_p^j$. In a 3D image, with cubic 3-cells and squared 2-cells in-between, everywhere we get $n_{i,j} = 4$, we may state $c_2^i \subset \partial\, c_3^j$. Therefore, in each $j$ column of $\mathtt{M}_2\,\mathtt{M}_3^t = (n_{i,j})$, we have exactly *six rows* where $n_{i,j} = 4$, since a cube (3-chain) has six boundary faces (2-chains). The unit incidence coefficients in $[\partial_3]$ are set accordingly, by filtering the coefficients with value `4`.

## Example: Boundary matrices for grids of cubes

We give here the full Julia code for algebraic computation of $\partial_3$ 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 `Float64` indices of grid cubes.

```
julia> V,CV = Lar.cuboidGrid([3,2,1]);

julia> V
3x24 Array{Float64,2}:
 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
6-element Array{Array{Int64,1},1}:
 [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 utility functions needed to generate the face data `FV` and edge data `EV` from cell data `CV`. In particular, `CV2FV` and `CV2EV` functions apply to all the 3-cells in `CV` the pattern of reference to vertices used by faces and edges of the single 3-cube:

```
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 (`VV`, `EV`, `FV`, `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:

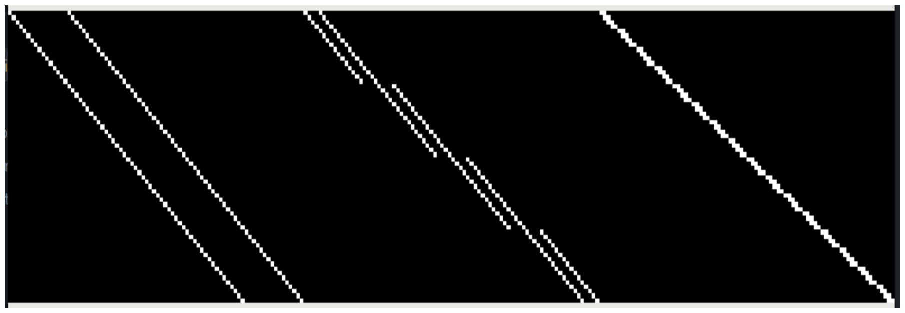
```
VV = [[v] for v=1:size(V,2)]
[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17],
[18], [19], [20], [21], [22], [23], [24]

FV = collect(Set{Array{Int64,1}}(cat(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}}(cat(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 sparse(I,J,Vals)
end
```

$\mathtt{M}_0 = \mathtt{K(VV)};\ \mathtt{M}_1 = \mathtt{K(EV)};\ \mathtt{M}_2 = \mathtt{K(FV)};\ \mathtt{M}_3 = \mathtt{K(CV)}$

*Boundary matrices*:

The boundary matrices between non-oriented chain spaces are computed by *sparse matrix multiplication* followed by *matrix filtering*, produced in Julia by broadcast of vectorized integer division ($.\div$), as follows:

$$\partial_1 = \mathtt{M}_0 * \mathtt{M}_1' = \mathtt{M}_1'$$
$$\partial_2 = (\mathtt{M}_1 * \mathtt{M}_2')\ .\div\ \mathtt{sum}(\partial_1, \mathtt{dims}{=}2)$$
$$\partial_3 = (\mathtt{M}_2 * \mathtt{M}_3')\ .\div\ \mathtt{sum}(\partial_2, \mathtt{dims}{=}2)$$



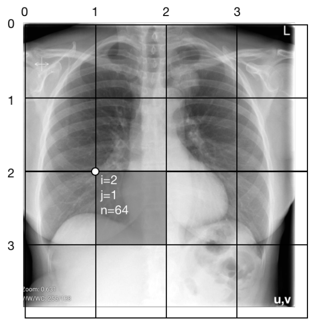


Fig. 1: The binary image of *sparse* coboundary matrix $[\delta_2] = [\partial_3]^t : C_2 \to C_3$, built for a small 3D image with `shape=[4,4,4]`. (a) the number of rows equates the size $4 \times 4 \times 4 = 64$ of the voxel set; (b) the number of columns is $d\, n\, (1+n)^{d-1} = 3 \times 4 \times 25 = 300$; (c) the number of non-zeros per row (size of face set of a single voxel) is 6; (d) the number of non-zeros per column is 2, but on boundary faces.

Fig. 2: A possible block partitioning of a radiologic image. The evidenced 2D block, of size $\mathtt{n}^d = 64^2$, is sliced as $\mathbb{B}$(`[2,1,64]`) = `image([128:172],[64:128)]`. Our topological methods do not require brick-size be submultiple of image-size.

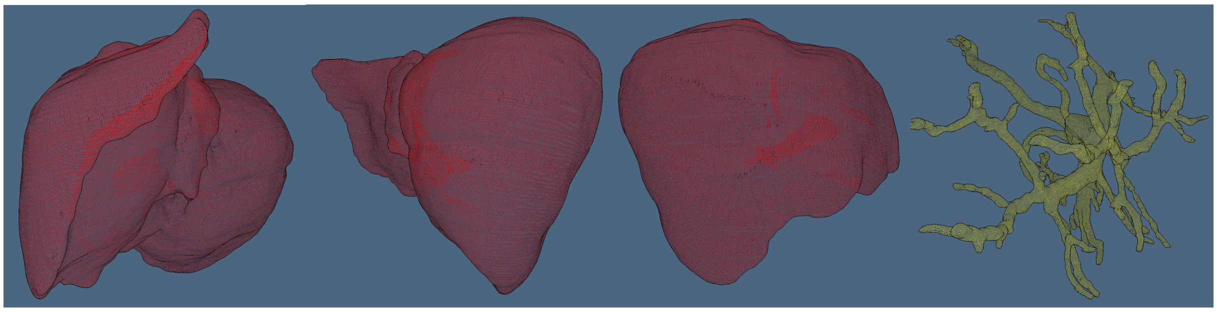


Fig. 3: Images of a pig liver, and portal veins with biggest sections, generated by `lar-surf.jl` package.

Brick-level parallelism:

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. Any cuboidal portion of image is completely determined by the Cartesian indices of its voxels of lowest and highest indices, and extracted by multidimensional array *slicing* as `image[`$\mathtt{l}_x$`:`$\mathtt{h}_x$`,`$\mathtt{l}_y$`:`$\mathtt{h}_y$`,`$\mathtt{l}_z$`:`$\mathtt{h}_z$`]`. For the sake of simplicity, we assume a common size on the three image axes, and the corresponding image portion $\mathbb{B}$, called *brick*, as a function of its element of the lowest brick coordinates `i,j,k` $\in$ `[1:n]` and the brick lateral size `n` $\in \mathbb{N}$:

$$\mathbb{B}(i,j,k,n) := \texttt{image[in:in+n, jn:jn+n, kn:kn+n]}$$

In our computational pipeline, 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 brick size $n$, is further enforced by the computation of a single boundary matrix $[\partial_d(n)]$ 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.

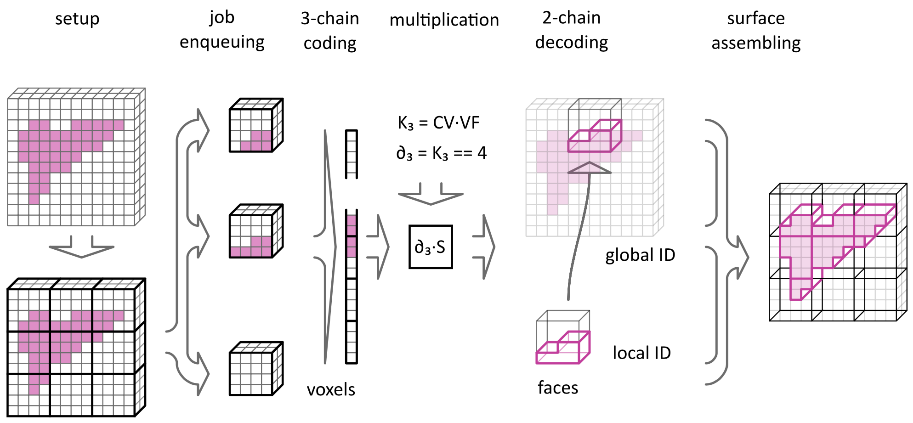


Fig. 4: Workflow of LAR-SURF algorithm

The whole approach is appropriate for SIMD 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. Taubin smoothing method [] is applied to final result.

Conclusions:

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 matrix, and the sparse matrix-vector multiplication are the main computational kernels of this approach. We may show a good speed-up over standard marching-cubes algorithms. The current implementation employs Julia's channels for multiprocessing. The computational pipeline is being strongly improved to gain a much greater speed-up using native Julia implementation of `CUDA` programming platform, and the Julia's `SuiteSparseGraphBLAS.jl` framework [? ] 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.

References:

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

[] 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. ISSN 1545-5955. URL http://dx.doi.org/10.1109/TASE.2009.2021342.

[] Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. Linear algebraic representation for topological structures. *Comput. Aided Des.*, 46:269–274, January 2014. ISSN 0010-4485. doi: 10.1016/j.cad.2013.08.044. URL http://dx.doi.org/10.1016/j.cad.2013.08.044.

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

[] Alberto Paoluzzi, Antonio DiCarlo, Francesco Furiani, and Miroslav Jirik. Cad models from medical images using LAR. *Computer-Aided Design and Applications*, 2016. URL http://dx.doi.org/10.1080/16864360.2016.1168216. Being published.

[] A. Paoluzzi, V. Shapiro, A. DiCarlo, F. Furiani, G. Martella, and G. Scorzelli. Topological computing of arrangements with (co)chains. *ACM Transactions on Spatial Algorithms and Systems*, November 2019. Submitted for publication.

[] Aydin Buluk, Tim Mattson, Scott McMillan, Jose Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, 2017. URL http://dx.doi.org/10.1109/IPDPSW.2017.117.