LAR-ABC, a representation of architectural geometry From concept of spaces, to design of building fabric, to construction simulation

Alberto Paoluzzi, Enrico Marino, and Federico Spini

Roma Tre University

Abstract. *This paper discusses the application of LAR (Linear Algebraic Representation) scheme to the architectural design process. LAR is a novel representation scheme for geometric design of curves, surfaces and solids, using simple, general and well founded concepts from algebraic topology [Dicarlo et al. 2014]. LAR supports all topological incidence structures, including enumerative (images), decompositive (meshes) and boundary (CAD) representations. It is dimension-independent, and not restricted to regular complexes. Furthermore, LAR enjoys a neat mathematical format, being based on chains, the domains of discrete integration, and cochains, the discrete prototype of differential forms, so naturally integrating the geometric shape with the supported physical properties. The LAR representation find his roots in the design language PLaSM [Paoluzzi et al. 1995; Paoluzzi 2003], and is being embedded in Python and Javascript, providing the designer with powerful and simple tools for a geometric calculus of shapes. In this paper we introduce the motivation of this approach, discussing how it compares to other mixed-dimensionality representations of geometry and is supported by open-source software projects. We also discuss simple examples of use.*

1 Introduction

Geometric models, called mesh in computational science and engineering, are of primary importance in old and novel problem areas, such as architectural geometry, material science or biomedicine. In particular, we see that novel applications require the convergence of shape synthesis and analysis, from computer graphics, computer imaging, and computer-aided geometric design, with discrete meshing of domains used for physical simulations.

In this paper we discuss a novel linear algebraic representation, supporting topological methods to represent and process mesh connectivity information for dimension-independent cellular complexes, including simplicial, cuboidal, and polytopal cells. This representation works even with the awkward domain partitions—with non-convex and/or non-manifold cells, and cells non homeomorphic to balls—that might arise in Building Information Modeling (BIM) and Geographic Information Systems (GIS) applications. Simplicial and cuboidal

cell complexes provide the standard mesh representation used in most science and engineering simulations, whereas complexes of possibly non-convex or non-contractible cells may be needed to represent the built environment in software applications for the Architecture, Engineering, and Construction (AEC) sector.

Our approach provides a unified representation where concepts and techniques from computer graphics (graphics primitives), geometric modelling (curves and surfaces) and solid modelling (solids and higher dim manifolds) converge with those from computational science (structured and unstructured meshes) in a common computational structure. This representation may be used in novel and highly demanding application areas[1], and may be supported by modern silicon-based APIs[2] on last and next generation hardware.

The descriptive power of a representation scheme is measured by the extent of its domain. In this sense, the LAR scheme — based on *chain complexes*, where chains are sets of cells — is defined on a quite large mathematical domain, and may represent a wide class of finite cell decompositions of a space. The LAR scheme may denote symbolically a large class of geometric shapes in $\mathbb{E}^d$, and makes use of basic concepts from algebraic topology (mostly from mod two homology), and of standard representations from basic computational algebra. The aim is to provide a representation that can support (at least in principle) all topological and geometric queries and constructions that may be asked to the corresponding model. LAR can be used from initial concept of spaces, to the additive manufacturing of design models, to the meshing for CAE analysis, to the detailed design of components of building fabric, to the BIM processing of quantities and costs.

2 Definitions

A central notion in solid modelling is the concept of representation scheme, a mapping from mathematical solid models to actual computer representations.

A compact topological subspace is a *convex cell* if it is the set of solutions of affine equalities and inequalities. A *face* of a cell is the convex cell obtained by replacing some of the inequalities by equalities. A *facet* of a cell is a face defined by just one equality. The *dimension n* of a *n*-cell is that of its *affine hull*, the smallest affine subspace that contains it.

A c*onvex-cell complex* or *polytopal complex P* is a finite union of convex cells in $\mathbb{E}^n$ such that: (i) if A is a cell of P, so are the faces of A; (ii) the intersection of two cells of P is a common face of each of them. A simplicial (respectively, cuboidal) complex is a polytopal complex where all cells are simplices (respectively, cuboids).

A *filtration* is an indexed set S_i of subobjects of an algebraic structure S, such that $i < j$ implies $S_i \subset S_j$. A *CW-structure* on the space X is a filtration $\emptyset = X^{-1} \subset X_0 \subset X_1 \subset \cdots \subset X = \bigcup_{n \in N} X_n$, such that, for each n, the space X_n is homeomorphic to a space obtained from X_{n-1} by attachment of n-cells of X in $\Lambda_n = \Lambda_n(X)$. A space

[1] It is being tested within the IEEE P3333.2 – Standard for Three-Dimensional Model Creation Using Unprocessed 3D Medical Data.

[2] A prototype implementation with OpenCL and WebCL is on the way.

endowed with a CW-structure is a *CW-complex*, also said a *cellular complex*. The space X_n is called the *n-skeleton* of X. Each subcomplex of X is a closed subset of X. A cellular complex is *finite* when it contains a finite number of cells.

The dimension of a complex is the maximal cell dimension. The r-skeleton X_r is the subcomplex formed by the cells of dimension $\leq r$. The 0-skeleton coincides with the set $V(X)$ of *vertices*.

Let be given a Hausdorff space X, and a finite cellular complex $\Lambda(X)$ such that $X = \Lambda_0 \cup \cdots \cup \Lambda_d$, and let $M_p := M_n^m(\mathbb{Z}_2)$ be binary matrices, with $m = \sharp\Lambda_p$ rows, and $n = \sharp\Lambda_0$ columns.

Definition 1 (Math models) *The LAR domain is defined on the set M of* chain complexes *supported by a finite cellular complex* $\Lambda(X)$.

Definition 2 (Computer representations) *The codomain of the LAR scheme is the set R of d-tuples of CSR sparse matrix representation*[3]

$$CSR(M_n^m(\mathbb{Z}_2)), \quad n = \sharp\Lambda_0, m = \sharp\Lambda_p$$

where d is the dimension of the space $X = \bigcup_p \Lambda_p$, *and* $1 \leq p \leq d$.

Every matrix $M_p = M_p(\Lambda(X))$ is an *incidence matrix*, i.e. it represents a *characteristic function* $\chi : \Lambda_p \times \Lambda_0 \to \mathbb{Z}_2$, so that $M_p(\mu, \lambda) = 1$ if and only if the cell $\mu \in \Lambda_p$ contains the cell $\lambda \in \Lambda_0$.

The interesting point is that, for a given cellular d-complex $\Lambda = \Lambda(X)$, all the M_p matrices $(1 \leq p \leq d)$ contain the same number of columns, and can be operated by standard matrix multiplication or transposition, providing a simple and convenient tool for computing boundary and coboundary operators and topological relations between cells.

The LAR scheme is a paradigm change in shape representation: whereas topological relations cannot be easily combined, linear operators are readily composed via matrix multiplication.

For *regular polytopal* d-complexes, where each cell is contained in a d-cell, only a single M_d matrix is needed to compute the homology of a space and its (co)boundary operators. Efficient methods exist to compute a vertex-based representation of $(p-1)$-cells from that of p-cells, for instance by using the *qhull* algorithm by [Barber et al. 1996] for polytopal cells, or a $O(1)$ formula for simplices.

Therefore, giving only CSR(M_d) is sufficient to fully denote the chain complex induced by $\Lambda(X)$. For a given decomposition, no ambiguity may arise. Using the language of representation schemes: *on the domain of polytopal complexes, including simplicial and cuboidal ones, all LAR representations are valid*. It is worth noting that CSR(M_d) exactly coincides with the bulk of common representations in Computer Graphics — for instance, the OBJ or PLY file formats.

[3] Compressed Sparse Row (CSR) format, for which efficient implementations on high-performance hardware exist. See [Bell and Garland 2008], [Buluç and Gilbert 2012], and [Lokhmotov 2012].

3 Models, structures, assemblies

In LAR-ABC we make a distinction between geometric *models*, *structures*, and *assemblies*. Some terminology and definitions are given below.

Geometric models A geometric *model* is a pair (*geometry*, *topology*) in a given coordinate system, where *topology* is the LAR specification of highest dimensional cells of a cellular decomposition of the model space, and *geometry* is specified by the coordinates of *vertices*, the spatial embedding of 0-cells of the cellular decomposition of space. From a coding viewpoint, a model is either an instance of the `Model` class, or simply a pair (`vertices`, `cells`), where `vertices` is a two-dimensional array of floats arranged by rows, and where the number of columns (i.e. of *coordinates*) equals the dimension n of the embedding Euclidean space $\mathbb{E}^n$. Similarly, `cells` is a list of lists of vertex indices, where every list of indices corresponds to one of d-dimensional *cells* of the space partition, with $d \leq n$.

Structures A *structure* is the LAR representation of a hierarchical organisation of spaces into substructures, that may be organised into lower-level substructures, and so on, where each part *may* be specified in a *local coordinate system*. Therefore, a structure is given as an *(ordered) list of substructures and transformations* of coordinates, that apply to all the substructures following in the same list. A structure actually represents a *graph of the scene*, since a substructure may be given a name, and referenced more than one time within one or more other structures. The *structure network*, including references, can be seen as an acyclic directed multigraph. In coding term, a structure is an instance of the *Struct* class, whose parameter is a list of either other structures, or models, or transformations of coordinates, or references to structures or models.

Assemblies An assembly is an *(unordered) list of models* all *embedded in the same coordinate space*, i.e. all using the same coordinate system (the *world coordinate system*). An assembly may be either defined by the user as a list of models, or automatically generated by the *traversal* of a structure network. At traversal time, all the traversed structures and models are transformed from their local coordinate system to the world coordinates, that correspond to the coordinate frame of the root of the traversed network, i.e. to the first model of the structure passed as argument to the `evalStruct` function, that implements the traversal algorithm. In few words, we can say that an assembly is the linearised version of the traversed structure network, where all the models are using the world coordinate system.

3.1 The design of LAR-ABC

LAR-ABC is a Python library for geometric design of building objects with the Linear Algebraic Representation (LAR) specialised for Architecture, Building and Construction (ABC). In the present first prototype implementation of the library,

we concentrate on the first two letters of this specification, namely the organisation of spaces (architecture) and the specification of physical, concrete components (building).

Concept design and project plan The client needs and wishes are initially specified by some initial set of requirements, traduced by the architect firm into a design concept, better specified by an initial project plan. When the project plan is accepted by the client, giving a definite shape to the first architectural concept, the model of construction is usually a 2.5D model, made by opaque or transparent 2D surfaces embedded in 3D space. In this stage, LAR-ABC allows for the computation of every topological or geometrical property of interest, including the evaluation of the surface of the building envelope and its partitioning into subsets with different thermal requirements, as well as the computation of the internal volume, and its partitioning into any classes of internal space, and will grant any other geometric computation or simulation (for example of the thermal behaviour) of possible interest for the architect or the client.

Building objects: components and assemblies The LAR description of the topology and its geometric embedding, defined by the position vectors of vertices or control points of the surfaces, makes possible to (mostly) automatically generate a first 3D model of the physical construction, i.e. of the concrete instances of building components. This (semi-)automatic transformation from a 2.5D model formed by surfaces to a 3D model formed by assemblies of solid objects, is obtained using the boundary operator, that allow to discriminate between the various subsystems of the building fabric, i.e. between the horizontal and vertical enclosures, the horizontal and vertical partitions of the interior, the elements of horizontal and vertical communications, and so on, as we show in Section 4.

Operators for assemblies and derived subclasses Remember that an assembly is an unordered *list* of geometric models, i.e. a list of pairs made by vertices and cells, and hence with named components, implementable as a dictionary in Python or as an object in Javascript. Therefore, few specialised higher-level functions are needed to apply the typical operations for models both to basic assemblies and to all derived specialised subclasses, associating specialised semantics to the basic geometric information. Some higher-order general utilities for handling basic geometric assemblies are given for this purpose.

3.2 Boundary operator

The LAR representation of the cellular complex $\Lambda(X)$ in Figure 1d, with $\Lambda_1 = \{e_0, e_1, \ldots, e_9\}$, and with non-convex faces $\Lambda_2 = \{f_0, f_1\}$, is:

```
FV = [[0,1,3,5,6,7],[0,2,3,4,5,6]]
EV = [[0,1],[0,2],[0,6],[1,3],[2,3],[3,5],[4,5],[4,6],
    [5,7],[6,7]]
```

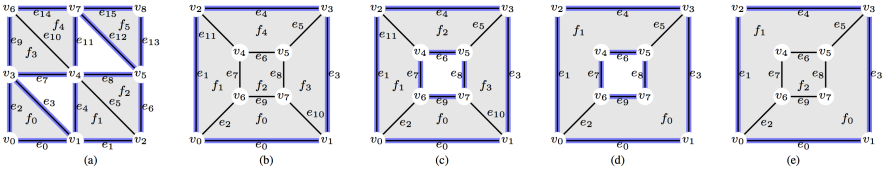


Figure 1: Cellular 2-complexes and their (blue) boundaries: (a) simplicial complex; (b,c) cuboidal complexes; (d,e) cellular complexes with non-convex 2-cells.

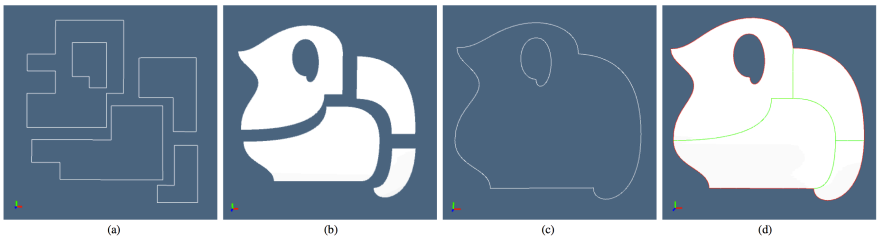


Figure 2: A cellular complex $B = \Lambda(X)$, with curved 2-cells non necessarily homeomorphic to the 2-ball: (a) input Bézier polygons, defined by cycles of control points of Bézier curves of various degrees; (b) exploded 2-cells; (c) boundary 1-chain; (d) boundary 1-chain (red) and interior 1-chain (green). See Appendix A for a detailed presentation of the model, including the actual Python code.

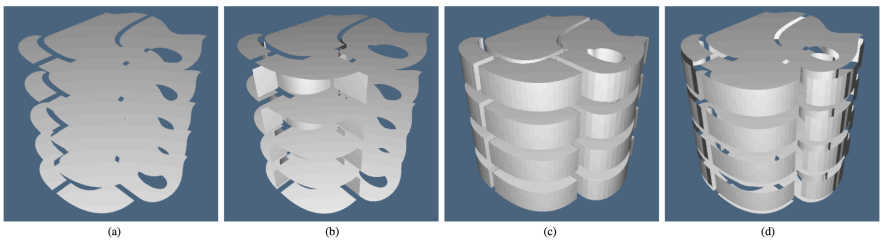


Figure 3: A cellular complex $B = \Lambda(X)$, with curved 2-cells non necessarily homeomorphic to the 2-ball: (a) input Bézier polygons, defined by cycles of control points of Bézier curves of various degrees; (b) exploded 2-cells; (c) boundary 1-chain; (d) boundary 1-chain (red) and interior 1-chain (green). See Appendix A for a detailed presentation of the model, including the actual Python code.

The corresponding binary matrices M_2 and M_1, indexed on the columns by 0-cells, and indexed on the rows by either by 2- or 1-cells are, respectively:

$$M_1 = \begin{pmatrix} 1&1&0&0&0&0&0&0\\ 1&0&1&0&0&0&0&0\\ 1&0&0&0&0&0&1&0\\ 0&1&0&1&0&0&0&0\\ 0&0&1&1&0&0&0&0\\ 0&0&0&1&0&1&0&0\\ 0&0&0&0&1&1&0&0\\ 0&0&0&0&1&0&1&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&1 \end{pmatrix}, \quad M_2 = \begin{pmatrix} 1&1&0&1&0&1&1&1\\ 1&0&1&1&1&1&1&0 \end{pmatrix}$$

In order to compute the matrix representation of the boundary operator $\partial_2 : C_2 \to C_1$ with respect to the standard bases (see [Dicarlo et al. 2014] for detail), the edge-face incidence matrix is computed first:

$$[I_{1,2}] = M_1 M_2^t = \begin{pmatrix} 2&1&2&2&1&2&1&1&2&2\\ 1&2&2&1&2&2&2&2&1&1 \end{pmatrix}^t.$$

where the (i, j) entry, by definition equal to $\langle \mu_i, \lambda_j \rangle = \mu_i(\lambda_j)$, denotes the number of vertices shared by the (elementary) chains $\mu_i \in C_1$ and $\lambda_j \in C_2$.

Let us recall, from [Dicarlo et al. 2014], that

$$[\partial_p]_{ij} = \begin{array}{ll} 1 & \text{if } A_{p-1,p}(i,j) = \sharp \mu_i = 2 \\ 0 & \text{otherwise.} \end{array}$$

Hence we get:

$$[\partial_2] = \begin{pmatrix} 1&0&1&1&0&1&0&0&1&1\\ 0&1&1&0&1&1&1&1&0&0 \end{pmatrix}^t$$

Finally, the boundary 1-chain is computed as the $(\mathrm{mod}\, 2)$ product of $[\partial_2]$ times the coordinate representation of the *total 2-chain* Λ_2 as element of C_2:

$$[\partial_2][\Lambda_2] = [\partial_2]\mathbf{1} \bmod 2 = [1,1,0,1,1,0,1,1,1,1] = \{e_0, e_1, e_3, e_4, e_6, e_7, e_8, e_9\}$$

The result is shown in blue colour in Figure 1d.

4 Example: apartment block design

A simple example of housing design is discussed in this section. The concept design of the dwelling is produced by using a vectorial drawing program, and shown in Figure 4. The input file is parsed, producing the LAR `model` given in the script below, as a pair `V,FV` of vertices `V` and 2-cells `FV`.

4.1 LAR model input

Input of the dwelling concept The vertices list `V` contains pairs of coordinates within a local reference frame. The 2-cell list `FV` of references to vertices is given counterclockwise, in order to automatically get a complete LAR representation of topology, i.e. the pair `FV, EV` of 2-cells and 1-cells.

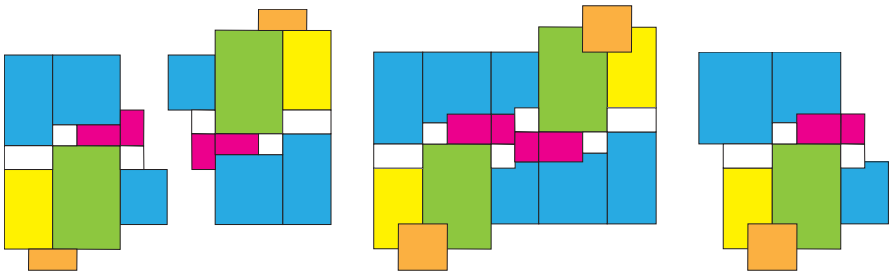


Figure 4: Concept design. living/eating area (green/yellow); bedrooms (cyan), lavatories (magenta); entrance (white).

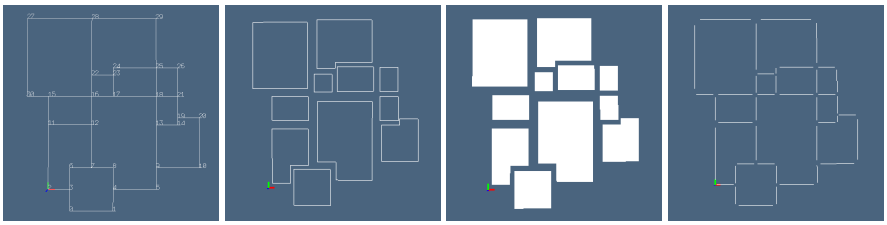


Figure 5: (a) LAR drawing as close polylines: (b) exploded polylines: (c) exploded 2-cells: (d) exploded 1-cells.

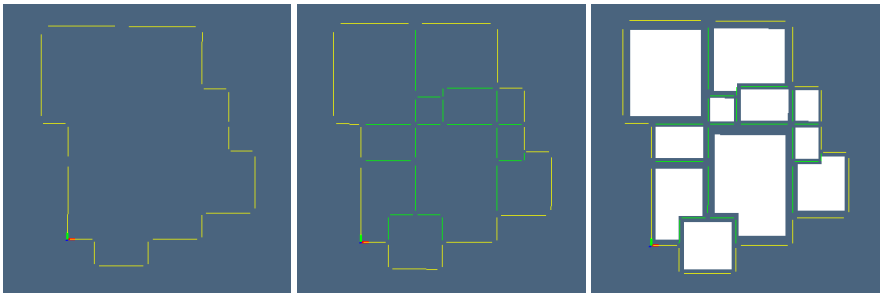


Figure 6: (a) 2-cells, interior 1-chain (green), boundary 1-chain (red): (b) boundary 2-chain: (c) interior 2-chain.

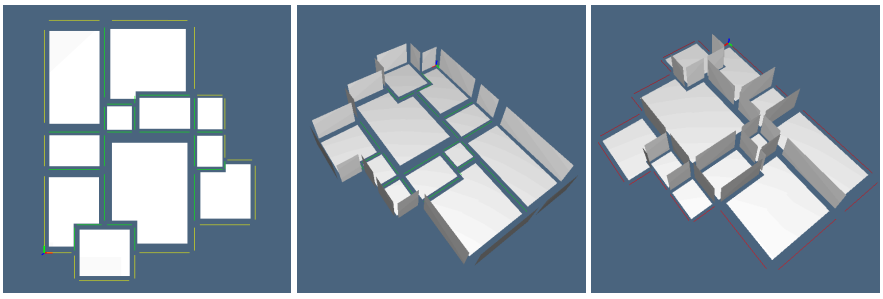


Figure 7: Partitioning of the 2-chains: (a) 2-cells: interior 1-chain (green), boundary 1-chain (yellow); (b) boundary 2-chain; (c) interior 2-chain.

```
V = [[3,-3],
[9,-3],[0,0],[3,0],[9,0],[15,0],[3,3],[6,3],[9,3],[15,3],[21,3],
[0,9],[6,9],[15,9],[18,9],[0,13],[6,13],[9,13],[15,13],[18,10],
[21,10], [18,13],[6,16],[9,16],[9,17],[15,17],[18,17],[-3,24],[6,
24],[15,24],[-3,13]]
```

```
FV = [
[22,23,24,25,29,28], [15,16,22,28,27,30], [18,21,26,25],
[13,14,19,21,18], [16,17,23,22], [11,12,16,15],
[9,10,20,19,14,13], [2,3,6,7,12,11], [0,1,4,8,7,6,3],
[4,5,9,13,18,17,16,12,7,8],[17,18,25,24,23]]
```

Simple data transformations The LAR model of the `dwelling` is simply given by the pair `V,FV`. Some simple transformations of the input data are given below, and displayed in Figure 5. The FL operator `AA` stands for *Apply-to-All* (a function to a list of arguments). The `pyplasm` primitive `TEXT` is used to show the enumeration of vertices in Figure 5a.

```
from myfont import *
dwelling = V,FV
poly = AA(POLYLINE)(lar2polylines(dwelling))
bU = AA(SOLIDIFY)(poly)
EV = face2edge(FV)
VIEW(STRUCT(poly + [T([1,2])(v)(S([1,2])([.1,.1])(TEXT(str(k))))
    for k,v in enumerate(V)]))
VIEW(EXPLODE(1.2,1.2,1)(poly))
VIEW(EXPLODE(1.2,1.2,1)(bU))
VIEW(EXPLODE(1.2,1.2,1)(MKPOLS((V,EV))))
```

4.2 Partitioning the 1-cells

The subdivision of the 1-cells of the complex, between boundary cells and interior cells, is executed by computing the boundary operator ∂_2, and multiplying it by the coordinate representation **1** of the 2D basis of cells.

Subdivide the 1-cells of the concept plan The input to `bUnit_to_eEiP`, to compute the 1D external envelope and interior partitions starting from 2D from building units, is given below and shown in Figures 6, 7, and 8.

```
eE,iP = bUnit_to_eEiP(FV,EV)
eE1D = AA(COLOR(YELLOW))(MKPOLS(V,[EV[e] for e in eE]))
iP1D = AA(COLOR(GREEN))(MKPOLS(V,[EV[e] for e in iP]))
VIEW(EXPLODE(1.2,1.2,1)(bU + iP1D + eE1D))
```

Spiral stair A fully parameterised LAR model of a spiral stair is given below. When instancing the `spiralStair` function, the user may specify the `thickness` of the slab, the major and minor radiuses `R` and `r`, the `riser` height of the step, the stair `pitch`, i.e. the vertical distance after a 2π turn, and the number of steps, corresponding to the angular subdivisions of the stair. Notice that the returned value is a LAR model, i.e. a pair (*vertices, cells*), given by `W,CW`.

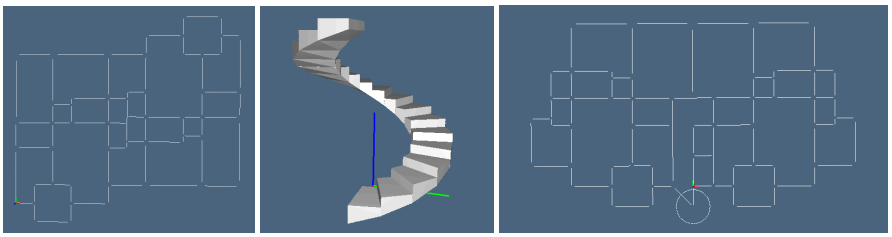


Figure 8: Concept design: (a) aggregation of two building units; (b) fully parametric spiral stair; the plan of a building flat.

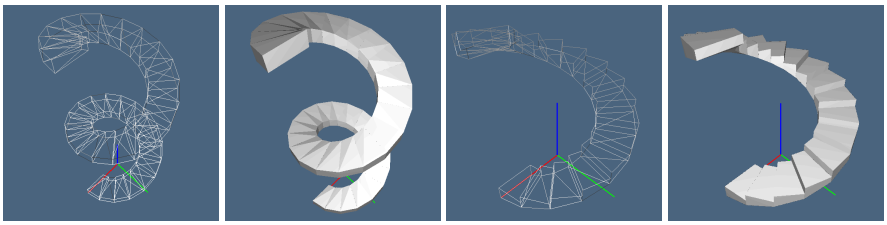


Figure 9: The construction of the spiral stair; (a) wire-frame of two turns of the solid spiraloid; (b) image of the solid spiraloid, as made of polytopal cells; (c,d) one turn of the spiral stair, obtained by suitable translation of the vertices of the spiralled cells.

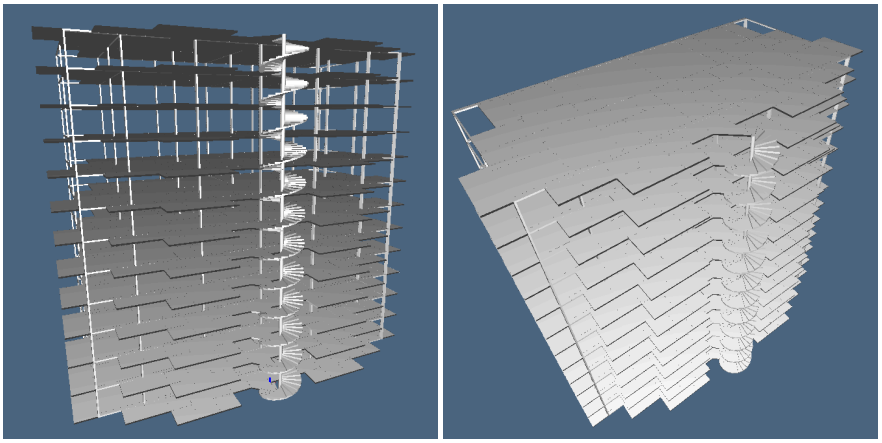
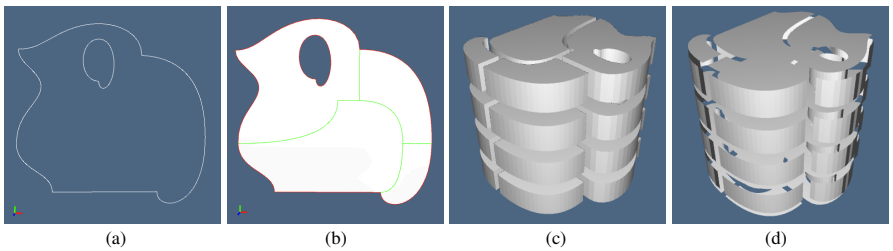


Figure 10: The 3D frame of the apartment block: (a) view from above; (b) view from top.



(a) (b) (c) (d)

Figure 11: A cellular complex $B = \Lambda(X) \subset \mathbb{E}^2$, with curved 2-cells non homeomorphic to the 2-ball: (a) boundary 1-chain; (b) boundary 1-chain (red) and interior 1-chain (green); (c) solid complex $B \times C$, with $C \subset \mathbb{E}$; (d) 2D complex $(B \times \partial C) \cup (\partial B \times C)$.

```
def spiralStair(thickness=0.2,R=1.,r=0.5,riser=0.1,pitch=2.,
                nturns=2.,steps=18):
    V,CV = larSolidHelicoid(thickness,R,r,pitch,nturns,steps)()
    W = CAT([[V[k],V[k+1],V[k+2],V[k+3]]+
        [SUM([V[k+1],[0,0,-riser]]),SUM([V[k+3],[0,0,-riser]])]
        for k,v in enumerate(V[:-4]) if k%4==0])
    for k,w in enumerate(W[:-12]):
        if k%6==0: W[k+1][2]=W[k+10][2]; W[k+3][2]=W[k+11][2]
    nsteps = len(W)/12
    CW =[SUM([[0,1,2,3,6,8,10,11],[6*k]*8])
                for k in range(nsteps)]
    return W,CW
```

Complexes with non-convex and curved cells Since the LAR representation is purely topological, the actual shape of the cells does not matter. Examples of non-linear and non-convex cells are given in Figures 11b, 11c, 11d.

```
# input of vertex locations (numbered from zero)
V = [[5.,29.],[17.,29.],[8.,25.],[11.,25.],[14.,25.],[0.,
23.],[5.,23.],[17.,23.],[27.,23.],[0.,20.],[5.,20.],[8.,
19.],[11.,19.],[11.,17.],[14.,17.],[0.,16.],[5.,16.],[14.
,16.],[17.,16.],[23.,16.],[0.,10.],[14.,10.],[23.,10.],[
27.,10.],[0.,6.],[5.,6.],[5.,3.],[20.,3.],[23.,3.],[20.,
0.],[27.,0.]]
```

Edges and faces describe the 2D and 1D cells partitioning the geometry, named here according to their meaning, i.e. `FV` (faces-by-vertices) and `EV` (edges-by-vertices), respectively. Edge lists are ordered and will be rendered as Bézier curves.

```
# input of faces as lists of control points indices
FV = [[0,1,2,3,4,5,6,7,9,10,11,12,13,14,15,16,17,18,20,
21],[7,8,18,19,22,23],[17,18,19,20,21,22,24,25,26,27,28],
[22,23,27,28,29,30]]
# input of edges as lists of control points indices
EV = [[5,6,0,1,7],[7,18],[18,17],[17,21,20],[20,15,16,10,
9,5], [12,11,2,3],[3,4,14,13,12],[7,8,23],[22,23],[22,19,
18],[22,28,27],[26,27],[26,25,24,20],[23,30,29,27]]
```

A *canonical* (numerically ordered) description of 2-cells is used for `FV`, without any 'topological' ordering. Conversely, the control points of Bézier curves must be ordered according to the geometric shape of edges (point-valued polynomials of various degrees).

5 Results and conclusions

The main idea developed in this paper is to characterize the cells of a CW-complex decomposition of a space by the subsets of 0-cells they contain. This choice allowed us to represent a large class of solid models, including spaces formed by assembling curves, surfaces, solids and higher-dimensional cells, under rather weak assumptions on the cell topology and geometry. When connectedness of cells is assumed, the LAR domain coincides with the domain of the SGC representation introduced by [Rossignac and O'Connor 1990], allowing cells non contractible to a

point. If cells are convex pointsets, then every LAR representation is valid, and requires as input no more than the description of the highest-dimensional cells. Using algebraic methods and mod two (co)homology, we were able to fully compute the model topology, i.e. all the relations of incidence/adjacency betweeen cells, as well as the chain-complex supported by the space partition, through the algebraic product of sparse binary matrices. Finally, our research was also driven by the search for techniques allowing the convergence of methods from graphics, geometric design, solid modeling and computational science, as well as by the search for a fast implementation with advanced tools for parallel computation over distributed and/or accelerated geometric processors. Representing geometry with sparse binary matrices seems to us a major progress on this direction. Both `lar-cc` and `pyplasm` projects are being implemented as opensource software, and can be downloaded from GitHub.

References

BACKUS, J., WILLIAMS, J., WIMMERS, E., LUCAS, P., AND AIKEN, A. 1989. FL language manual, parts 1 and 2. Tech. rep., IBM Research Report.

BACKUS, J., WILLIAMS, J. H., AND WIMMERS, E. L. 1990. An introduction to the programming language FL. In *Research topics in functional programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 219–247.

BACKUS, J. 1978. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM 21*, 8, 613–641.

BARBER, C. B., DOBKIN, D. P., AND HUHDANPAA, H. 1996. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw. 22*, 4 (Dec.), 469–483.

BELL, N., AND GARLAND, M. 2008. Efficient sparse matrix-vector multiplication on CUDA. Tech. Rep. NVR-2008-004, NVIDIA Corp, December.

BULUÇ, A., AND GILBERT, J. R. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal of Scientific Computing (SISC) 34*, 4, 170 – 191.

DICARLO, A., PAOLUZZI, A., AND SHAPIRO, V. 2014. Linear algebraic representation for topological structures. *Comput. Aided Des. 46* (Jan.), 269–274.

LOKHMOTOV, A. 2012. Implementing sparse matrix-vector product in OpenCL. In *OpenCL tutorial, 6th Int. Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*.

PAOLUZZI, A., PASCUCCI, V., AND VICENTINO, M. 1995. Geometric programming: a programming approach to geometric design. *ACM Trans. Graph. 14*, 3 (July), 266–306.

PAOLUZZI, A. 2003. *Geometric Programming for Computer Aided Design*. John Wiley & Sons, Chichester, UK.

ROSSIGNAC, J. R., AND O'CONNOR, M. A. 1990. SGC: a dimension-independent model for pointsets with internal structures and incomplete boundaries. In *Geometric modeling for product engineering*. North-Holland.

A Appendix: Implementation example

In this appendix we discuss a straightforward implementation of the LAR scheme, and discuss step by step the generation of geometric models in Figure 11. They should be considered as simple BIM (Building Information Modeling) applications in the AEC (Architecture, Engineering, Construction) domain. The coding is in Python, which exhibits its full power in rapid prototyping developments. The *Pyplasm* package for *geometric programming* (see [Paoluzzi 2003]) is used, with higher-level functionals from FL ([Backus 1978; Backus et al. 1989; Backus et al. 1990]). *Pyplasm* operators are written all-caps.

A.1 Shape input

The model data were entered through an editor for SVG (simple vector graphics), the W3C standard for vector graphics on the web. Several tools exist for interactively editing a SVG shape. We used the free web application *svg-edit* within a browser. Once parsed the input file, the points were scaled by a $1/10$ factor and snapped to a grid of integers, to compensate for low precision of the free input tool, that did not provides any snap, finally generating a list of points, given as a list of pairs of floats, named **V**, and two lists of lists of point indices.

```
# input of vertex locations (numbered from zero)
V = [[5.,29.],[17.,29.],[8.,25.],[11.,25.],[14.,25.],
[ 0.,23.], [5.,23.],[17.,23.],[27.,23.],[0.,20.],[5.,
20.],[8.,19.],[11.,19.],[11.,17.],[14.,17.],[0.,16.],
[5.,16.],[14.,16.],[17.,16.],[23.,16.],[0.,10.],[14.,
10.],[23.,10.],[27.,10.],[0.,6.],[5.,6.],[5.,3.],
[20.,3.],[23.,3.],[20.,0.],[27.,0.]]
```

Edges and faces describe the 2D and 1D cells partitioning the geometry, and are named here according to their meaning, i.e. **FV** (faces-by-vertices) and **EV** (edges-by-vertices), respectively. Edges and faces are given as lists of control points indices. Edge lists are ordered and will be rendered as Bézier curves.

```
# input of edges as lists of control points indices
EV = [[5,6,0,1,7],[7,18],[18,17],[17,21,20],[20,15,16,
10,9,5], [12,11,2,3],[3,4,14,13,12],[7,8,23],[22,23],
[22,19,18],[22,28,27],[26,27],[26,25,24,20],[23,30,29,
27]]

# input of faces as lists of control points indices
FV = [[0,1,2,3,4,5,6,7,9,10,11,12,13,14,15,16,17,18,
20,21],[7,8,18,19,22,23],[17,18,19,20,21,22,24,25,26,
27,28],[22,23,27,28,29,30]]
```

A *canonical* (numerically ordered) description of 2-cells is used here, without any 'topological' ordering. Conversely, the control points of Bézier curves must be

ordered according to the geometric shape of edges (point-valued polynomials of various degrees). Of course, this ordering does not affect the topological computations, but determines the geometric embedding of the shape. Notice, from Figure 11, that the 2-cells are *non convex*, and that one of them is even non contractible to a point.

A.2 Kernel functions

Only a few simple functionalities are required to implement the LAR scheme. As discussed in the paper, **FV** and **EV** are a CSR representation of binary matrices. In particular, kernel functions are needed to: (a) set-up the internal representation of CSR matrix format; (b) display the matrices in a readable way; (c) execute matrix transposition and matrix product operations; (d) update the value of non-zero matrix elements.

Conversion to standard CSR format We use the **sparse** subpackage, provided by the *Scipy* package for scientific computing in Python, to handle sparse matrices efficiently. Faster implementations in OpenCL and WebCL for use from C^{++} and JavaScript, respectively, will be available soon. Presently, the matrix conversion to the internal format is executed by the **csr** function.

Display of binary CSR matrices The display of a binary matrix starting from the CSR format is provided by the **csr2mat** function.

```
# characteristic matrices
print "FV =\n", csr2mat(csr(FV))
>>> FV =
[[11111111011111111110110000000000]
 [00000001100000000011001100000000]
 [00000000000000000111111011111100]
 [00000000000000000000000110001111]]

print "EV =\n", csr2mat(csr(EV))
>>> EV =
[[11000111000000000000000000000000]
 [00000001000000000010000000000000]
 [00000000000000000110000000000000]
 [00000000000000000100110000000000]
 [00000100011000011000100000000000]
 [00110000000110000000000000000000]
 [00011000000011100000000000000000]
 [00000001100000000000000010000000]
 [00000000000000000000000110000000]
 [00000000000000000001100100000000]
 [00000000000000000000000100001100]
 [00000000000000000000000000011000]
 [00000000000000000000100011110000]
 [00000000000000000000000010001011]]
```

CSR matrix transposition and product Only two basic computational linear algebra operators are used, namely (a) to *transpose* a CSR matrix and (b) to *multiply* two CSR matrices. The computation of the $\mathbf{EF} : C_2 \to C_1$ operator is just implemented as follows. It is transposed below to **FE** for the sake of space. Notice that the product of binary matrices is an integer matrix, whose entry (i, j) denotes the number of

vertices connoting the incidence of the edge j upon the face i. For instance, the first column below tell us that the (curved) edge 0 has 5 control vertices belonging to the face 0, whereas only 1 vertex belongs to face 1.

```
# product and transposition
EF = csrProd(csr(EV), csrTrans(csr(FV)))
FE = csrTrans(EF)
print "FE =\n", csr2mat(FE)
>>> FE =
[[5 2 2 3 6 4 5 1 0 1 0 0 1 0]
 [1 2 1 0 0 0 0 3 2 3 1 0 0 1]
 [0 1 2 3 1 0 0 0 1 3 3 2 4 1]
 [0 0 0 0 0 0 0 1 2 1 3 1 0 4]]
```

Computation of (co)boundary operators The function **maxFilter** is used to get either the matrix $[\partial_2]$ from **EF** or the matrix $[\delta_1]$ from **FE**. Of course, a boundary edge belongs to a face iff all edge vertices belong to (or generate a curved edge of) the face. Hence we search each row of the boundary matrix (column of the coboundary matrix) for its relative maxima element. The reader should compare columnwise the **FE** matrix above with the **coboundary** matrix below.

```
# boundary and coboundary operators
boundary = maxFilter(EF)
coboundary = csrTrans(boundary)
print "coboundary =\n", csr2mat(coboundary)
>>> coboundary =
[[1 1 1 1 1 1 1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 1 1 1 0 0 0 0]
 [0 0 1 1 0 0 0 0 0 0 1 1 1 0]
 [0 0 0 0 0 0 0 0 0 1 0 1 0 1]]
```

A.3 Making solid cells from b-reps

The coordinate representation of boundary 1-chains of 2-cells is given by the rows of $[\delta_1]$, i.e. by the rows of the **coboundary** matrix. Let first compute the boundary 1-chain of **_2cells** (from rows of the coboundary matrix). Here numbers denote *edge numerals*.

```
# boundary 1-chains of unit 2-chains
_1chains = format(coboundary,shape="csr")
_2cells = [_1chains[k].tocoo().col.tolist()
    for k in range(len(FV))]
print "_2cells =\n", _2cells
>>> _2cells =
[[0,1,2,3,4,5,6],[1,7,8,9],[2,3,9,10,11,12],[8,10,13]]
```

Notice that **_2cells** is a list representation of boundary 1-chains, whereas **cells2D** below is a list of geometric values, whose "exploded" value is shown in Figure 11a.

```
# 2D cell complex as a list of polyline structures
cells2D = [ STRUCT([ POLYLINE([V[v]
    for v in EV[edge]]) for edge in cell ])
    for cell in _2cells ]
VIEW(EXPLODE(1.2,1.2,1)(cells2D))
```

A *solid* representation of 2-cells is computed by applying the **SOLIDIFY** algorithm[4] to the piecewise-linear approximation of 1D edges (generated by the **bezier** function) of the face boundary. An exploded view of the 'solid 2-cells' is shown in Figure 11b.

```
# approximation with 20 segments of a Bezier curve
def bezier(points):
    return MAP(BEZIERCURVE(points))(INTERVALS(1)(20))

# 2D cell complex as a list of solid cells
cells2D = [ SOLIDIFY(STRUCT([ bezier([V[v]
  for v in EV[edge]]) for edge in cell]))
    for cell in _2cells ]
VIEW(EXPLODE(1.2,1.2,1)(cells2D))
```

A.4 Boundary and interior computation

The *boundary* and *interior* 1-cells of the complex are computed in this section. The CSR rep of the *total 2-chain* (the set of all 2-cells) is put in **_2chain** as a column matrix.

```
# coordinate rep of the largest 2-chain
_2chain = csrTrans(csr([range(len(FV))]))
```

Then the boundary 1-chain, as a list of triples (row, column, 1), is computed by the **csrFilter(1, 0)(csrProd(boundary, _2chain))** expression, and the 1-chain **_1boundarycells** is constructed. A **boundary1D** piecewise-linear approximation of the boundary is generated and displayed in Figure **??**c.

```
# boundary 1-chain computation
_1boundarycells = [triple[0] for triple
    in csrFilter(1,0)(csrProd(boundary,_2chain))]
print "1-boundary =\n", _1boundarycells
>>>1-boundary =
[1,2,3,4,6,7,8,9,10,11,13,14,15,16,18,19,20]

boundary1D= AA(bezier)([[V[v] for v in EV[e]]
    for e in _1boundarycells ])
VIEW(STRUCT(boundary1D))
```

Finally, the chain **_1interiorcells** is computed as the complement to **_1boundarycells** and displayed in color in Figure **??**d.

```
# computation of interior 1-cells
_1interiorcells = [e for e in range(len(EV))
    if e not in _1boundarycells]
print "1-interior =", _1interiorcells
>>> 1-interior = [0,5,12,17]

interior1D = AA(POLYLINE)([[V[v] for v in EV[e]]
    for e in _1interiorcells])
VIEW(STRUCT(AA(COLOR(RED))(boundary1D) +
    cells2D + AA(COLOR(GREEN))(interior1D)))
```

[4]The interested reader may find it on page 615 of [Paoluzzi 2003] book.

A.5 Assembling the 3D model

The extruded 3D models of Figure **??** are computed in this section. A 0-complex and an 1-complex, both embedded in $\mathbb{E}^1$, are needed in order to perform the extrusion of the 2D model as a Cartesian product of pointsets of dimension 2, 1, and 0.

Computation of the 0-complex A list **V_0** of five 1D points gives the z-elevation heights. The CSR binary matrix **VV_0** provides the description of the highest dimensional cells (vertices), by vertices itself. Of course, it coincides with the identity matrix 5×5. The complex **floors2D** generated by Cartesian product $\Lambda_2(X) \times \Lambda_0(Y)$ cell pairs is shown in Figure 3a.

```
# computation of a 0-complex
V_0 = [[0.],[50.],[100.],[150.],[200.]]
VV_0 = AA(LIST)(range(len(V_0)))
print "VV_0 =", VV_0
>>> VV_0 = [[0],[1],[2],[3],[4]]

cells0D = AA(MK)([V_0[v[0]] for v in VV_0])
floors2D = AA(PROD)(CART([cells2D,cells0D]))
VIEW(EXPLODE(1.2,1.2,1.5)(floors2D))
```

Computation of the 1-complex Analogously, **EV_1** describes the edges by means of vertices. The matrix of boundary operator **boundary1** $:= [\partial_1]$ equates by definition the matrix [**VE_1**], since the max() value of each [**VE_1**]'s row is 1.

```
# computation of a 1-complex
EV_1 = [[0,1],[1,2],[2,3],[3,4]]
VE_1 = csrTrans(csr(EV_1))
print "VE_1 =\n", csr2mat(VE_1)
>>> VE_1 =
[[1 0 0 0]
 [1 1 0 0]
 [0 1 1 0]
 [0 0 1 1]
 [0 0 0 1]]
boundary1 = VE_1  # bydef: max(VE_1[v]) == 1, forall v
```

The function **format** is from *Scipy*'s **sparse** package, and transforms a set of (row,column,value) triples into its internal format (**"csr"**). The **.tocoo().col.tolist()** method composition extracts the columns (edge indices) of each unit 1-chain.

```
# 1D cell complex
chains_1 = csrTrans(boundary1)
chains_1 = format(chains_1,shape="csr")
cells_1 = [chains_1[k].tocoo().col.tolist()
    for k in range(len(EV_1))]
print "cells_1 =", cells_1
>>> cells_1 = [[0,1],[1,2],[2,3],[3,4]]

cells1D = [ POLYLINE([ V_0[v] for v in edge ])
    for edge in cells_1 ]
VIEW(EXPLODE(2,2,2)(cells1D))
```

Cartesian products Finally, the sets of cells of various dimensions are generated and variously assembled by the below script, using the *Pyplasm*'s Cartesian product of pointsets. The various complexes (implemented as lists of *Pyplasm*'s geometric values) are displayed exploded in Figures 10c and 10d, respectively.

```
# Cartesian product complexes
boundary2D=AA(PROD)(CART([boundary1D,cells1D]))
interior2D=AA(PROD)(CART([interior1D,cells1D]))
VIEW(EXPLODE(1.1,1.1,1.5)(interior2D+floors2D))
VIEW(EXPLODE(1.1,1.1,1.5)(boundary2D+floors2D))
solid3D = AA(PROD)(CART([cells2D,cells1D]))
VIEW(EXPLODE(1.1,1.1,1.5)(solid3D))
```