Finite Algebras for Solid Modeling using Julia's Sparse Arrays[⋆,⋆⋆]

Alberto Paoluzzi[a], Vadim Shapiro[b], Antonio DiCarlo[a], Giorgio Scorzelli[c] and Elia Onofri[a]

[a]Roma Tre University, Rome, RM, Italy
[b]University of Wisconsin, Madison & International Computer Science Institute (ICSI), Berkeley, California, USA
[c]Scientific Computing and Imaging Institute (SCI), Salt Lake City, Utah, USA

ARTICLE INFO

Keywords:
Computational Topology
Chain Complex
Cellular Complex
Solid Modeling
Constructive Solid Geometry (CSG)
Linear Algebraic Representation (LAR)
Arrangement
Boolean Algebra

ABSTRACT

An early research in solid modeling led by Herbert Voelcker at the University of Rochester and later at Cornell suggested that every solid representation scheme corresponds to an algebra, where the elements of the algebra are solid representations constructed and edited using operations in the algebra. For example, every CSG representation describes an element in a finite Boolean algebra of closed regular sets, whereas every boundary representation describes an element of a vector space of 2-chains in an algebraic topological chain complex. In this paper, we elucidate the precise relationships (functors) between all algebras used for CSG and boundary representations of solids. Based on these properties, we show that many solid modeling operations, including boundary evaluation, reduce to straightforward algebraic operations or application of identified functors that are efficiently implemented using point membership tests and sparse matrix operations. To fully exploit the efficacy of the new algebraic approach to solid modeling, all algorithms are fully implemented in Julia, the modern language of choice for numerical and scientific computing.

1. Introduction

Foundations of modern solid modeling systems have been laid in the 1970's by the Production Automation Project directed by Herb Voelcker at University of Rochester (Voelcker and Requicha, 1977, 1993). From the early beginnings it was widely accepted that two complementary mathematical models of solidity are important: the Boolean algebra of *r*-sets (closed regular and semi-analytic sets) that are closed under regularized set operations, and the algebraic topological model of topological (cellular) polyhedra that governs properties of most discretizations and boundary representations (b-reps). In particular, the latter are formally computer representations of orientable bounding 2-cycles in the linear space of 2-chains defined over a valid cell complex (Requicha, 1977; Paoluzzi et al., 2020).

1.1. Motivation

The algebraic operations and the properties of these algebraic systems are central to validity and completeness of their respective models and representations. Fortunately, the two models are mathematically compatible: a key result states that a homogeneously 2D topological polyhedron Y in $\mathbb{E}^3$ is a 2-cycle and is semi-analytic iff there exists a unique r-set X such that $\partial X = Y$ (Requicha, 1977). This implies that these two representations can be always converted into each other — exactly, within numerical precision — which in turn shaped the architecture of all modern CAD systems (Shapiro, 2002).

Despite its obvious importance, this theoretical connection between the point set and topological cellular models is only existential, but not constructive, because it does not suggest how the correspondence between the two types of models can be created and maintained in practice. This results in a widely recognized *robustness* problem, which manifests itself in unanticipated algorithmic failures due to inconsistent logical decisions in presence of numerical errors (Hoffmann et al., 1988; Hoffmann, 1989).[1] There are at least two types of such logical inconsistencies. (a) Boundary representations in most commercial systems are implemented using specialized data structures that often impose additional mathematical limitations and assumptions, thus undermining the fundamental premise of solid modeling. For example, the widely used algebra of manifold data structures is closed under so-called Euler operators but

⋆This work was partially supported from Sogei S.p.A. – the ICT company of the Italian Ministry of Economy and Finance, by grant 2016-17.
⋆⋆V.S. was supported in part by National Science Foundation grant CMMI-1344205 and National Institute of Standards and Technology.
*Corresponding author
ORCID(s): 0000-0002-3958-8089 (A. Paoluzzi)

[1]Robustness should not be confused with accuracy, exactness, or "correctness" of numerical computations, particularly because many geometric operations, such as intersection, are not continuous under small perturbations and hence are not formally computable (Edalat and Lieutier, 1999).

is not capable of representing many valid CSG representations of non-manifold solids. (b) Most boundary evaluation algorithms are designed for and are limited by such specialized data structures. Typically, following the boundary evaluation procedures in Voelcker and Requicha, tentative faces and edges are generated by intersecting boundaries of input primitives and are classified against the target CSG expression, followed by merging and simplification of their union (Shapiro, 2002). Correctness and performance of such algorithms depend on the input primitives in the CSG representation, on the form and type of the CSG representation itself, as well as on the specific data structures and algorithms. In fact, recently, the approach has been extended to primitives bounded by polygonal meshes, including meshes that may be invalid and self-intersecting (Zhou et al., 2016). The correctness of all such algorithms depends implicitly on the assumed algebraic properties above, which are not explicitly enforced or represented. As a result, many of the algorithms are limited to particular assumptions about the data structures (manifoldness, connectivity, etc.) and depend on heuristic numerical computation steps that undermine robustness and scalability of the boundary evaluation algorithms.

In contrast, this paper shows that, by formulating boundary evaluation in terms of algebraic operations and functors between the relevant algebras, all logical operations are cleanly separated from numerical computations and remain consistent by construction at all times.

1.2. Previous work

There have been several attempts to make the connection between CSG representations and cellular models more explicit and algorithmic, either in the context of representation conversions (Shapiro, 1991, 1997; Requicha and Rossignac, 1992) or representation unification and generalization (Armstrong et al., 1999; Rossignac and O'Connor, 1989; DiCarlo et al., 2014a). Rossignac and O'Connor proposed SGC (Selective Geometric Complex) data structure to construct and represent most types of combinatorial representations (Rossignac and O'Connor, 1989). Rossignac and Requicha proposed CNRG (Constructive Non-regularized Geometry) representation, a counterpart generalization of CSG modeled as collection of regions (Rossignac and Requicha, 1991). An ambitious attempt to unify point set and combinatorial representations, as well as many algorithms, through common semantics formulated in terms of canonical cellular decompositions is described in (Armstrong et al., 1999), but the effort fell short of providing a path towards practical implementation of such semantics. More generally, in contrast to the approach described in this paper, none of these earlier approaches takes advantage of or explicitly enforces the algebraic properties of the underlying cellular representations.

A common denominator of all such approaches is their reliance on decomposition of space into sign-invariant subsets $\{X_i\}$, where sign-invariance means that every point in each X_i has identical classification (`in`, `on`, `out`) with respect to a given *fixed set of primitives.* In his doctoral dissertation, supervised by Herb Voelcker, Shapiro observed the isomorphism between decompositions of space and finite algebras of sets that are represented uniquely and canonically by the union of atoms in the decomposition (Shapiro, 1991). He also established a hierarchy of such algebras corresponding to different representations schemes and described a number of previously unsolved representation conversions using the correspondence between algebras and decompositions (Shapiro, 1991). In particular, the sign-invariant sets serve as atoms of the *finite* Boolean algebra of all sets representable by these primitives and standard set operations (Halmos, 1963), and each set can be represented by a union of such atoms in a canonical disjunctive form. When decomposition is an arrangement of n primitives, the atoms are relatively open disjoint sets of all dimensions in the corresponding partition of space, and the Boolean algebra contains all sets describable by Boolean expressions using these primitives. On the other hand, if the boundary of every sign-invariant set is also the union of other sign-invariant sets in the partition, the Boolean algebra becomes a *closure algebra*, and the atoms become cells in the cell complex decomposition of space that subsumes all possible cellular representations of the closed sets in the algebra, including their boundary representations. The closed regular sets in this algebra form a smaller Boolean algebra in its own right, closed under the regularized set operations. The atoms of this algebra are defined by regularized intersections of primitives and their complements that form a quasi-disjoint decomposition of space. Any CSG representation X using the given set of primitives can be represented also by a union of atoms, thus providing a unique canonical representation for X.

The relationship between the space decompositions giving rise to the closure algebra (containing all possible cellular structures and boundary representations) and the Boolean algebra of closed regular sets (containing all possible CSG representations) brings us a step closer to establishing direct computational links between the two representations. This goal is finally achieved in this paper by expressing this relationship in a common mathematical language (DiCarlo et al., 2014a) and computational representation: LAR (Linear Algebraic Representation). It represents

the sign-invariant space decomposition corresponding to the closure algebra as a chain complex, using linear algebra with sparse matrices. It also explicitly represents and enforces validity of all cellular representations, including boundaries and homogeneous solids using 2- and 3-dimensional chains respectively. LAR encodes all geometric and topological data in terms of sparse matrices and replaces incidence and boundary computations by linear transformations. In (Paoluzzi et al., 2020), the authors showed how LAR can be efficiently constructed for arbitrary arrangement of cell complexes in $\mathbb{E}^3$, starting from enumeration of vertices (unevaluated LAR) of the cells of polyhedral primitives, and algorithmically constructing the sparse (co)boundary matrices of the underlying chain complex (evaluated LAR). We demonstrate in this paper that this approach allows to compute algebraically the boundary of *every* Boolean combination of n primitives.

1.3. Paper outline

In this paper, we show that Linear Algebraic Representation (LAR) (DiCarlo et al., 2014b), initially constructed for an arrangement of polyhedral solids, allows efficient, general, and robust boundary evaluation for any CSG expression in a finite Boolean algebra generated by topological polyhedral solids. Furthermore, once the LAR representation is constructed, this boundary evaluation does not place any assumptions or limitations on the topology of the represented solids, and does not require any specialized data structures or algorithms beyond simple point membership test and sparse matrix multiplication. To fully exploit the efficacy of this new algebraic approach to solid modeling, all algorithms are fully implemented in *Julia*, the modern language of choice for numerical and scientific computing.

In particular, this paper describes algebraic ideas and tools needed to carry out entirely the programmatic approach to solid modeling discussed in this introduction. In particular, the background Section 2 summarizes basic concepts about cell decompositions, chain complexes, polyhedral arrangements, and Boolean algebras, and finally summarizes the best aspects of Julia scientific programming.

Section 3 contains the main contributions of the paper; it discusses the complete computational pipeline, including the algorithmic steps needed to construct the LAR representation of polyhedral arrangements in $\mathbb{E}^3$, and to establish isomorphism between the 3-chains in this arrangements and the Boolean algebra of closed regular sets, whose atoms directly correspond to the columns of the matrix of boundary operator from 3-chains to 2-chains.

A summary of contributions of the approach is given in Section 4.1, including scope, methods, and new results. A brief discussion of the presented ideas in the current panorama and of future prospect concludes the paper in Section 4.2. Two Appendices A and B are dedicated, respectively, to supporting materials, including the preliminary design of a DSL (Domain Specific Language) for CSG in Julia, and to explicitly showing some computations of simple solid modeling examples.

2. Background

In this section we provide a set of definitions for the basic concepts used in this paper. It contains no new material and may be skipped at first reading, or used as reference for concepts and terminology deployed in later sections. In particular, we will introduce complexes of *cells* and *chains*, the cellular decompositions of the ambient Euclidean space, called *arrangements*, discuss algebraic properties of *Constructive Solid Geometry* (CSG) representations, and introduce geometric programming using Julia. We restrict out attention to dimensions *two* and *three*, and to topological polyhedra, *i.e.* to curved triangulable polyhedra that are homeomorphic to piecewise-linear (PL) polyhedra. Our implementation and most examples in the paper rely on PL-polyhedra, but the approach is applicable to general topological polyhedra, under additional assumptions discussed in Section 4. In Appendix, we discuss additional small and easy-to-understand scripts of *Julia*.

2.1. Cell and Chain complexes

A *complex* is a graded set $S = \{S_i\}_{i\in I}$ *i.e.* a family of sets, indexed in this paper over $I = \{0, 1, 2, 3\}$. We use two different but intertwined types of complexes, and specifically complexes of *cells* and complexes of *chains*. Their definitions and some related concepts are given in this section. Greek letters are used for the cells of a space partition, and roman letters for chains of cells, coded as either (un)signed integers or sparse arrays of (un)signed integers.

2.1.1. Cell complexes

d-Manifold A *manifold* is a topological space that resembles a flat space locally, *i.e.*, near every point. Each point of a d-dimensional manifold has a neighborhood that is homeomorphic to $\mathbb{E}^d$, the Euclidean space of dimension d. Hence, this geometric object is often referred to as d-manifold.

Cell A *p-cell* σ is a p-manifold with boundary ($0 \leq p \leq d$) which is piecewise-linear, connected, possibly non convex, and not necessarily contractible. This definition refers to cellular complexes used in this paper and is different from other ones because a cell is neither simplicial, nor convex, nor contractible. In our theory, cells may contain internal holes; cells of CW-complexes (Hatcher, 2002) are, conversely, contractible to a point. We deal with Piecewise-Linear (PL) cells of dimensions 0, 1, 2, and 3, respectively. It should be noted that 2- and 3-cells may contain holes, while remaining connected. In other words, our cells are p-polyhedra, *i.e.* segments, polygons and polyhedrons embedded in two- or three-dimensional space.

Cellular complex A *cellular p-complex* is a finite set of cells that have at most dimension p, together with all their r-dimensional boundary faces ($0 \leq r \leq p$). A *face* is an element of the PL boundary of a cell, that satisfy the *boundary compatibility* condition. Two p-cells α, β are said boundary-compatible when their point-set intersection contains the same r-faces ($0 \leq r \leq p$) for both α and β. A cellular p-complex is said *homogeneous* when each r-cell ($0 \leq r \leq p$) is face of a p-cell.

Skeleton The s-skeleton of a p-complex Λ_p ($s \leq p$) is the set $\Lambda_s \subseteq \Lambda_p$ of all r-cells ($r \leq s$) of Λ_p. Every skeleton of a regular complex is a regular subcomplex. The difference $\Lambda_r - \Lambda_{r-1}$ of two skeletons is the set U_r of r-cells.

Support space The support space $|\Lambda|$ of a cellular complex is the point-set union of its cells.

Characteristic function Given a subset S of a larger set A, the characteristic function $\chi_A(S)$, also called the *indicator function*, is the function defined to be identically one on S, and zero elsewhere. (Rowland, 2005).

LAR Geometric Representations The **unevaluated LAR** (DiCarlo et al., 2014b) introduced the use of sparse binary arrays to represent and compute the algebraic topology of cellular complexes, i.e., linear spaces of (co)chains, and linear (co)boundary operators.

The Linear Algebraic Representation (LAR) of a piecewise-linear (PL) p-complex Λ_p in the Euclidean space $\mathbb{E}^d$ ($p \leq d$) is given by (a) an embedding map $\mu : \Lambda_0 \to \mathbb{E}^d$, and by (b) the matrices $K_r := [\chi_{\Lambda_0}(\lambda)]$, with $\lambda \in U_r$, $0 \leq r \leq p$. The map $\chi_{\Lambda_0}(\lambda)$ is the *characteristic function* applied to cell λ (as vertex subset) resulting in a binary sequence of length $\#\Lambda_0$.

This function characterizes every cell through the binary string denoting the subset of its "vertices". In unevaluated LAR each r-cell $u_r \in U_r$ (ordered set) is associated to a row of the sparse binary *characteristic matrix* K_r. See DiCarlo et al. (2014a). The **evaluated LAR** (Paoluzzi et al., 2020) is discussed in Section 2.4.

2.1.2. Chain complexes

A *p-chain* can be seen, with some abuse of language, as a collection of p-cells. In this sense we write $U_p = \Lambda_p - \Lambda_{p-1}$ for the set of *unit* p-chains ($1 \leq p \leq d$), and $C_p = \mathscr{P}(U_p)$ for the space of p-chains, where $\mathscr{P}$ is the power set.

Linear chain space The set $C = \oplus C_p$, direct sum of chain spaces, can be given the structure of a graded vector space (see section 2.1.2 and Arnold, 2018, pp. 11–12) by defining sums of chains with the same dimension, and products times scalars in a field, with the usual properties.

Chain bases As a linear space, each C_p contains a set of irreducible generators. The natural *basis* $U_p \subset C_p$ is the set of *independent* (or *elementary*) chains $u_p \in C_p$, given by singleton elements. Consequently, every chain $c \in C_p$ can be written as a linear combination of this basis with field elements, and is uniquely generated. Once the basis is fixed, *i.e.*, U_p is ordered, the unique unsigned coordinate representation of each $\{\lambda_k\} =: u_k \in C_p$ is a binary array with one non-zero element in position k, and all other elements 0. The ordered sequence of scalars may be drawn either from $\{0, 1\}$ (unsigned representation) or from $\{0, 1, -1\}$ (oriented representation). With abuse of language, we often call p-cells the elements of U_p.

Graded vector space A *graded vector space* is a vector space V expressed as a direct sum of spaces V_p indexed by integers in $[0, d] := \{p \in \mathbb{N} \mid 0 \leq p \leq d\}$:

$$V = \oplus_{p=0}^{d} V_p. \tag{1}$$

A linear map $f : V \to W$ between graded vector spaces is a *graded map* of degree k if $f(V_p) \subset W_{p+k}$ for each p.

Chain complex A *chain complex* is a graded vector space V furnished with a graded linear map $\partial : V \to V$ of degree -1 called *boundary operator*, which satisfies $\partial^2 = 0$. In other words, a chain complex is a sequence of vector spaces C_p and linear maps $\partial_p : C_p \to C_{p-1}$, such that $\partial_{p-1} \circ \partial_p = 0$. The notation $C_\bullet$ is used in this paper for the chain complex over the binary field $\{0, 1\}$, and $C_\bullet^{\circlearrowleft}$ for the oriented chain complex over the ternary field $\{0, 1, -1\}$, used to get oriented boundaries.

Cochain complex A *cochain complex* is a graded vector space V furnished with a graded linear map $\delta : V \to V$ of degree $+1$ called *coboundary operator*, which satisfies $\delta^2 = 0$. That is to say, a cochain complex is a sequence of vector spaces C^p and linear maps $\delta^p : C^p \to C^{p+1}$, such that $\delta^{p+1} \circ \delta^p = 0$. In the rest of this paper we identify chains and cochains, so we use subscripts for both spaces (motivation in subsection A.1.1).

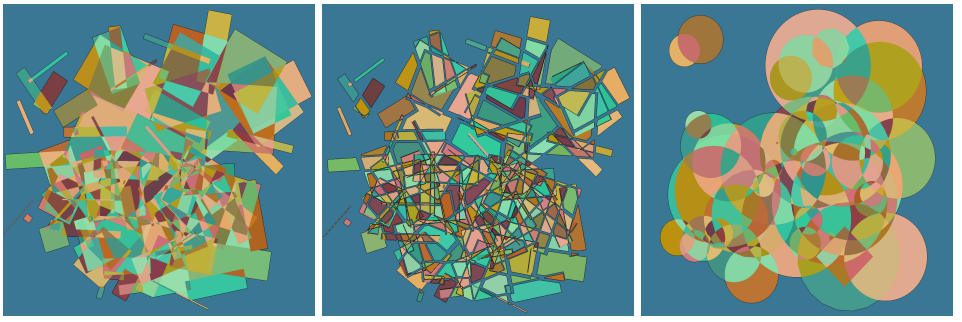
Operator matrices The matrices of boundary and coboundary operators (their transpose) are very sparse, with sparsity growing linearly with the number n of rows (sparse columns in Julia). Sparsity may be defined as one minus the ratio between non-zeros and the number of matrix elements. It is fair to assume that the non-zeros per row in each K_p matrix are bounded by a small constant independent on n, hence the number of non-zero elements grows linearly with n. With common data structures (Cimrman, 2015) for sparse matrices, the storage cost $O(n)$ is linear with the number of cells, with $O(1)$ small cost per cell that depends on the storage scheme.

2.2. Chains and Arrangements

The word *arrangement* is used in combinatorial geometry and computational geometry and topology as a synonym of space partition. Construction of arrangements of lines, see Dimca (2017), segments, planes and other geometrical objects is discussed in Fogel et al. (2007), with a description of CGAL software (Fabri et al., 2000), implementing 2D/3D arrangements with Nef polyhedra (Bieri, 1995) by Hachenberger et al. (2007). A review of papers and algorithms concerning the construction and counting of cells may be found in the chapter on Arrangements in the "Handbook of Discrete and Computational Geometry" (Goodman et al., 2017). Arrangements of polytopes, hyperplanes and d-circles are discussed in Björner and Ziegler (1992).

Space Arrangement Given a finite collection $\mathcal{S}$ of geometric objects in $\mathbb{E}^d$, the arrangement $\mathcal{A}(\mathcal{S})$ is the decomposition of $\mathbb{E}^d$ into connected open cells of dimensions $0, 1, \ldots, d$ induced by $\mathcal{S}$ (Halperin and Sharir, 2017). We are interested in the Euclidean space partition induced by a collection of PL cellular complexes.

Let be given a collection $\mathcal{S}$ of geometric objects. Examples include, but are not limited to: line segments, quads (quadrilaterals), triangles, polygons, meshes, pixels, voxels, volume images, B-reps, *etc.* In mathematical terms, a geometric object is a topological space embedded in some $\mathbb{E}^d$ (Delfinado and Edelsbrunner, 1995). A novel method to compute the topology of their space arrangement $\mathcal{A}(\mathcal{S})$ as a *chain complex* $C_\bullet$ (see Diagram (5)), was introduced in Paoluzzi et al. (2020). 2D arrangements generated by random rectangles and circles are shown in Figure 1.

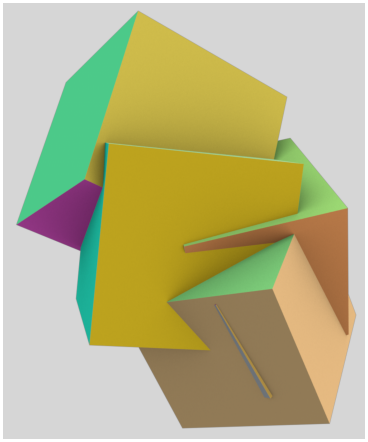


(a) (b) (c)

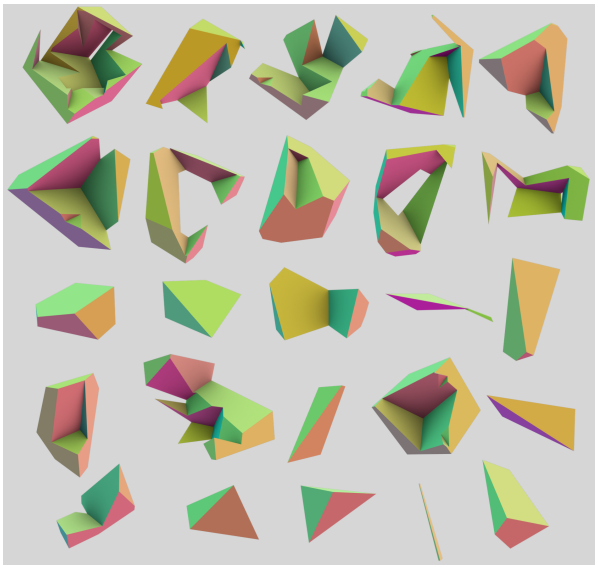
Figure 1: The cells of 2D arrangements generated by (a) rectangles of random size, position and orientation; (b) exploded view of the previous image; (c) random polygonal approximations of 2D circles with random center and radius. Take notice of the fact that 2-cells may be non convex and/or non contractible to a point, i.e., with holes.

Example 2.2.1 (3D arrangement)**.** In Figure 2 we show the arrangement $\mathcal{A}(\mathcal{S})$ generated by the collection $\mathcal{S}$ made

by the thirty-five 2-faces of 5 intersecting random cubes. According to (Paoluzzi et al., 2020), each *3-cell in* $\mathcal{A}(S)$ *is generated* by *a column* of the *sparse matrix* of *boundary map* $\partial_3 : C_3 \to C_2$, with values in $\{0, 1, -1\}$.



(a)



(b)

Figure 2: (a) A collection $\mathcal{S}$ of five random cubes in $\mathbb{E}^3$; (b) the display of 3-cells of the generated $\mathbb{E}^3$ arrangement $\mathcal{A}(S)$ (not in scale, and suitably rotated to better exhibit their complex shape). The quasi-disjoint union of all atoms gives the five cubes. Note that some cells contain holes.

The matrix of any linear map between two linear spaces contains by columns the basis of domain space represented in the basis of the codomain space. Therefore, the columns of $[\partial_3] : C_3 \to C_2$ are 2-cycles, *i.e.* closed chains in C_2. In particular, elementary 3-chains are join-irreducible *atoms* of the CSG algebra with closed regular cells. They may be non contractible to a point (when they contain holes) and non convex. The outer cell is the complement of their union. Any geometric model (out of 2^{25}) from the Boolean CSG algebra generated by these five cubes is made by a subset of those 25 atoms. See Figure 2.

2.3. Cycles and Boundaries

Two greatly useful subspaces are contained within any space C_p of chains: subspaces of cycles and boundaries.

Chain, cycle, and boundary subspaces A *p-cycle* is defined as a p-chain in C_p without boundary, hence it is an element of the kernel Z_p of ∂_p, the red sets in Figure 3. A *p-boundary* is a p-chain which is the boundary of a $(p+1)$-chain, hence it is an element of the image B_p of ∂_{p+1}, the pink sets of Figure 3.

Property 2.3.1 (Columns of ∂_3 matrix are irreducible 2-cycles)**.** In general, the matrix of any linear operator between linear spaces, e.g., $\partial_p : C_p \to C_{p-1}$, gets a sense only when bases have been fixed for both domain and codomain spaces. In this case (orderings are already fixed by the bases) each column of $[\partial_p]$ contains the coordinates of a basis vector of C_p, i.e., by the coefficients of its (unique) representation as linear combination of the C_{p-1} basis. Hence, each columns of $[\partial_3]$ is a closed element (vector) of C_{p-1} linear space. It is closed (i.e., it is a $(p-1)$-cycle) because the constraint that "the boundary of boundary of each p-chain is empty". Therefore, every column of boundary matrices, say the k-th column $[u]_k$, *is a cycle* by definition. Whereas the matrices $[\partial_2]$ and $[\delta_1] = [\partial_2]^t$ are mathematically computable starting from the input data set $[\partial_1]$ (see the Example 2.5.2), the matrix $[\partial_3]$ is totally unknown initially, and was constructed algorithmically (one column at a time) by the Topological Gift Wrapping (TGW) algorithm in 3D, introduced by Paoluzzi et al. (2020), via iteration with $[\partial_2]$ and $[\partial_2]^t$ on input 2-cells, until all scalar coefficients pop up as matrix columns. In Algorithm 8 we provide a generalized version of TGW in dimension d, with input $[\partial_{d-1}]$ and output $[\partial_d^+]$. Not all columns of $[\partial_d^+]$ are linearly independent, and one of them (in particular, the outer $(d-1)$-cycle, boundary of the unbounded exterior cell in $\mathbb{E}^d$ arrangement) must be removed, giving the matrix $[\partial_d]$, where all cycles (columns) are irreducible (independent).

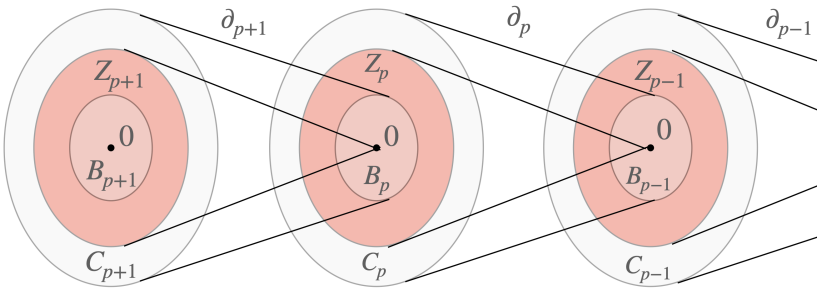


Figure 3: The set of p-boundaries $B_p \subset \partial_{p+1}C_{p+1}$ is a subset of the kernel $Z_p \subset C_p$ of p-cycles of p-chains, since the boundary of a boundary is empty. In other words, $\partial^2 = \partial_p\partial_{p+1} = 0$. The standard practice with linear operators is not to use brackets for application to an argument $(\partial_{p+1}(C_{p+1}))$, and to remove the composition operator $(\partial^2 = \partial\partial)$.

Property 2.3.2 (Rows of ∂_3^+ matrix sum to zero)**.** Each row of ∂_3 matrix corresponds to an irreducible element of the basis $U_2 \subset C_2$. By construction, each basis element $u \in U_2$ is used exactly twice in matrix $[\partial_3^+]$ with opposite coefficients $+1$ and -1, so proving the assertion. Once removed the outer cycle, the set of generators of the 2-cycle subspace Z_2 produced by TGW is linearly independent. In particular, the column of outer (unbound) 2-cycle corresponds to the sum of all the others, changed of sign (if oriented) or not (if not oriented). Differently speaking, any one column (cycle) of $[\partial_3^+]$ is generated by the topological sum of all the others.

Topological Gift Wrapping (Paoluzzi et al., 2020) can be denoted as a function $TGW : C_3 \to Z_2$. The row space goes from a basis of irreducible 2-chains to a basis of irreducible and oriented 2-cycles, plus one more 2-cycle. The extra 2-cycle is reducible to the sum of all the others, and gives the oriented boundary of the "outer" unbound 3-chain.

In (Paoluzzi et al., 2020) an algorithm was presented for transformation of cycles basis to boundaries basis. This algorithm is reported as Algorithm 9 is the Appendix. We observe that two remarks are essential for it, which concern the transformation of cycle chains to boundary chains. The first is that, by construction, the basis of 2-cycles corresponding to $[\partial_3]$ columns are (a) elementary (irreducible) and (b) non intersecting, even in their coordinate chain representation, since they involve different rows (2-cells); the second concerns their cardinalities. It is well known that $Z_p \supseteq B_p$ or, in words, there may be cycles which are not boundaries (see Figure 3). So, the boundary of the outer chain in C_{p+1}, if disconnected, is built in $B_p \subset Z_p$ by summing two or more basis cycles, whose non-zero elements are surely in different row positions by contruction.

Example 2.3.1 (Boundary of concentric spheres)**.** A *sphere* in $\mathbb{E}^d$ is the locus of d-points that have distance r from a fixed d-point, the sphere *center*. The sphere is a closed $(d-1)$-dimensional surface (without boundary). The $(d-1)$-sphere is also the boundary of a solid d-ball, which is the locus of all d-points which have distance less or equal to r from the center point. Let us consider the 3D space partition generated by two 2-spheres S_1 and S_2 with the same center and different radiuses $r_1 > r_2$. There are three solid cells: (a) the *outer unbounded cell* A, *i.e.*, $\mathbb{E}^3$ minus the ball of radius r_1; (b) the solid *shell* B with thickness $r_1 - r_2$; and (c) the *solid inner* ball C of radius r_2. Within the two interfaces, there are four closed irreducible 2-cycles generated as columns of $[\partial_3]$ in this complex, pairwise summing to zero and with opposite orientations. Let us straight denote, from exterior to interior, as

$$[\partial_3^+] = [u_1\ u_2\ u_3\ u_4] \quad \text{and} \quad [\partial_3] = [u_2\ u_3\ u_4], \quad \text{with} \quad u_1 = -(u_2 + u_3 + u_4).$$

If we denote the three "solid" basis elements in $U_3 \subset C_3$ (3-chain space) as A, B, C and the whole space as X, we can express them as Boolean algebra expressions:

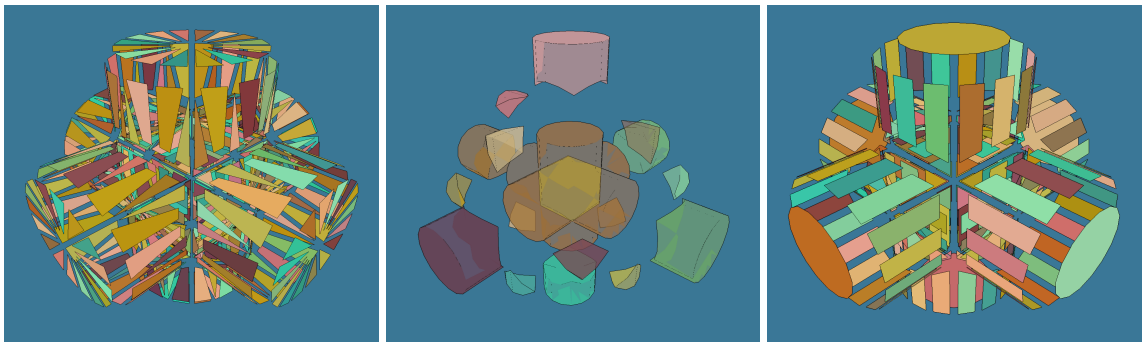
$$X = A + B + C \tag{2}$$

$$A = X - B - C; \quad B = X - A - C; \quad C = X - A - B. \tag{3}$$

In terms of oriented boundary 2-chains it is easy to see that

$$\partial A = u_1; \quad \partial B = u_2 + u_3; \quad \partial C = u_4, \quad \text{with} \quad u_1 + u_2 = u_3 + u_4 = 0, \quad \text{and hence} \quad u_1 + u_2 + u_3 + u_4 = 0.$$

Finally, note that the cycles u_2 and u_3 are not boundaries: $u_2, u_3 \in (Z_2 - B_2) \subset C_2$.



(a) (b) (c)

Figure 4: We show that different shape representations: (a) triangulated b-rep, and (c) LAR b-rep, produce the same $\mathbb{E}^3$ arrangement, and hence the same algebraic atoms (b). We show also that LAR produces a more compact b-rep, through a much smaller $[\partial_3]$ matrix. Note, with a smaller number of rows and of nonzeros, and hence with a faster construction.

Example 2.3.2 (Cycles → Boundaries)**.** The aim of this example is to illustrate that different shape representations will produce the same $\mathbb{E}^3$ arrangement, and hence the same algebraic atoms. In particular, it can be seen that LAR uses a smaller b-rep, hence a smaller $[\partial_3]$ matrix. Using our implementation, we may define a 3D model `assembly = Lar.Struct([ tube, Lar.r(pi/2,0,0), tube, Lar.r(0,pi/2,0), tube])` with three instances of `tube=cylinder()` with $n = 16$ sides, default diameter of length 1, height $h = 2$, and $k = 2$ decompositions in the axial direction, shown in Figure 4. For a description of `Lar.Struct` semantics, the reader may see the similar example 2.5.3. The exploded 2-cycles of the space arrangement, i.e. the atoms given by the 20 columns of $[\partial_3^+]$, the redundant basis of the Z_2 are shown. In the center of Figure 4b readers could see the boundary of outer space, obtainable by *linear combination* of the other 2-cycles, i.e. by *union* of the other atoms.

Remark (Two b-reps comparison). In Figure 4 three cylinders provide the same 3-chain basis, *i.e.* the same atoms, generated either by (1) boundary triangulations (see Figure 4a) or by (2) general LAR faces (see Figure 4c), where we used quadrilaterals on the lateral surface of the three cylinders, and two polygonal approximations of circle at their extremities. The numbers of input faces (2-cells) to be split in order to get a 3-space arrangement are 288 (with boundary triangulations) vs 102 (with LAR). Let us compare the benefits in space and time by the different numbers of faces in Z_2 bases: $(16 \times 2 + 2) \times 3 = 102 << 288 = (16 \times 4 + 2 \times 16) \times 3$, resulting in the second case in almost a third of rows of the $[\partial_3]$ matrix. Of course, the number of columns of $[\partial_3]$ remains the same in both cases.

In solid modeling, the word *shell* is used to denote the maximal connected closed surfaces in a b-rep of a solid object (see, e.g., Paoluzzi et al. (1989)). Here, a shell is every connected 2-cycle of the boundary of a 3-chain, or simply: each of boundary cycles of each connected solid component, including holes. Every *single* shell, including those internal to some unit 3-chain u_i, is obtained by matrix multiplication $[\partial_3^+][u_i]$, with $u_i \in U_3$ represented in coordinates by $\mathbf{e}_i$, the zero vector with only one 1 in position i. The result will be in $B_2 \subset Z_2 \subset C_2$, generating the b-rep of an algebra atom (see Figure 2b and the next section).

It is important to remark again that the matrix $[\partial_3^+]$, as generated by TGW, contains a basis of $Z_3 \subset C_3$, plus one more column, sum of all the others. The Algorithm 9, combining appropriate columns of $[\partial_3^+]$, generates a basis $[\partial_3]$ of the *validity subspace* $B_2 \subset Z_2 \subset C_2$, where each 3-basis element is being expressed as a 2-cycle (b-rep), possibly unconnected. In other words, by construction, each irreducible unit 3-chain $u \in U_3 \subset C_3$ corresponds to a single connected 2-cycle in Z_2. Conversely, the boundary (chain in B_2) of a general (sign-invariant) 3-chain, possibly disconnected, is given by the chain sum of irreducible 2-cycles in $B_2 \subset Z_2 \subset C_2$.

2.4. Algebras

A *semialgebraic set* is a subset of $S \subset R^n$ defined by a finite collection of polynomial inequalities of the form $Q(x_1, ..., x_n) \geq 0$. Finite unions, intersections, complement and projections of semialgebraic sets are still semialgebraic sets.

Finite algebra Let $\mathcal{A}$ be a nonempty set, and operations $\otimes_i : \mathcal{A}^{n_i} \to \mathcal{A}$ be functions of n_i arguments. If for all is A is closed under $\otimes_i$, then the system $\langle \mathcal{A}; \otimes_1, \ldots, \otimes_k \rangle$ is called an *algebra*. Alternatively, we say that $\mathcal{A}$ is a set with operations $\otimes_1, \ldots, \otimes_k$. If $\mathcal{A}$ has a finite number of elements, the algebra is said to be *finite*.

Boolean algebra In mathematics and mathematical logic, *Boolean algebra* is the type of algebra in which the values of the variables are truth values, i.e., their value is either *true* or *false*, usually denoted 1 and 0, respectively. We may think of a finite Boolean algebras $\mathscr{B}$ as a set isomorphic to the power set $\mathscr{P}(X)$ of some finite set X. The power set is naturally equipped with complement, union and intersection operations, which correspond to $-, \vee, \wedge$ operations in Boolean algebra. The complement of A is also denoted as $\overline{A}$.

Finite Boolean Algebra is a Boolean Algebra with a finite number n of atoms. Every finite Boolean algebra $\mathscr{B}$ is isomorphic to the field $\mathscr{P}[n]$, the set of subsets of first n integers, and therefore isomorphic to the Boolean algebra $\mathbf{2}^n$.

Property 2.4.1 (Boolean algebra)**.** Any finite algebra $\mathscr{B}$ is isomorphic to the Boolean algebra $\mathscr{P}(X)$, with the set X containing n elements. Therefore $\mathscr{B}$ can be mapped one-to-one onto the set $\chi_X(\mathscr{P}(X))$ of the images of characteristic functions of $\mathscr{P}(X)$ elements with respect to X, i.e., with the set of strings of n elements in $\{0, 1\}$.

Shapiro (1991) presented a *hierarchy of algebras* to define formally a family of Finite Set-theoretic Representations (FSR) of semi-algebraic subsets of $\mathbb{E}^d$, including representation schemes for solid and non-solid objects, such as B-reps, Constructive Solid Geometry, cellular decompositions, Selective Geometric Complexes, and others.

In this paper, we represent and implement for $d = 2, 3$ the solid Boolean algebras of CSG with closed regular cells, generated by the arrangement of $\mathbb{E}^d$ induced by a collection of cellular complexes with polyhedral cells of dimension $d - 1$ (Paoluzzi et al., 2020).

Generators A set $\mathcal{H}$ generates the algebra $\mathcal{A}$ (under some operations) if $\mathcal{A}$ is the smallest set closed w.r.t the operations and containing $\mathcal{H}$. The elements $h_i \in \mathcal{H}$ are called *generators* of the algebra $\mathcal{A}$. The elementary solid shapes in a CSG expression are the generators of its own CSG algebra.

Atom An *atom* is an element which cannot be decomposed into two proper subsets, like a singleton that cannot be written as a union of two strictly smaller subsets. An atom is a minimal non-zero element; a is an atom iff for every b, either $b \wedge a = a$ or $b \wedge a = 0$. In the first case we say that a belongs to the *structure* of b.

Structure of algebra elements We call *structure* of $b \in \mathscr{P}(X)$ the atom subset S such that b is the irreducible union of S. By extension, we also call structure of b the *binary string* associated with the ordered sequence of its atoms (elements of X). In other words, the structure $S(b)$ is the image of the characteristic function $\chi_X(S)$.

Property 2.4.2 (Boolean atoms are unit 3-chains)**.** There is a natural transformation between d-chains defined on a spatial arrangement and the algebra generated by that arrangement. Unit d-chains correspond to atoms of the algebra; the $[c]$ coordinate representation (bit array) of any d-chain c generates the coordinate representation in boundary space: $[\partial_d][c] = [b] \in B_{d-1} \subset C_{d-1}$. Of course, when $d = 3$, for the properties of the matrix of the linear operator between linear spaces, the matrix $[\partial_3] : C_3 \to C_2$ contains the domain space basis (unit 3-chains) represented in target space.

LAR Geometric Complex It is worthwhile to remark that, in order to display a triangulation of boundary faces in their proper position in space, the whole information required *(geometry & topology)* is contained within the **evaluated LAR** data structure or *Geometric Complex* (GC), by the arrangement of space given as a pair (Paoluzzi et al. (2020)):

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

where `V` has type `Matrix{Real}` with 3 rows and $\#C_0$ columns, and `(CF, FE, EV)` are sparse coboundary matrices. A GC transforms the (possibly non connected) boundary 2-cycle of a Boolean result (see Example 3.4.1) into a complete b-rep of the solid. The ordered pairs of letters from `V,E,F,C`, correspond to the coboundary sequence V*ertices*→E*dges*→F*aces*→C*ells* expressed through the C*olumn*→R*ow* order of matrix maps of operators.

2.5. Using the Julia language

One of main decision of the research and development work leading to this paper, was to use the Julia language for scientific programming. Julia has many merits, having reached optimally the main goals posed to its design, and

shortly reported here from (Bezanson et al., 2017, 2018). Julia combines features of *productivity languages*, such as Python or MATLAB, with characteristics of *performance-oriented languages*, such as C++ or Fortran. Syntactically is easy and fast to write and debug, and enjoys a great collection of packages.

In particular, the language features are: (a) an expressive type system, allowing optional type annotations; (b) multiple dispatch using these types to select implementations; (c) metaprogramming for code generation; (d) dataflow type inference algorithm allowing types of most expressions to be inferred; (e) aggressive code specialization against runtime types; (f) Just-In-Time (JIT) compilation using the LLVM compiler framework; and (g) Julia's carefully written libraries that leverage the language design.

Only few syntax notions are needed in order to allow the unknowledgeable reader to understand the few lines of code in the following pages. Julia strongly resembles Python for generic code, and MATLAB for linear algebraic calculus with matrices and vectors. A broadcast dot operator applies any operator or function to all the elements of an array. Multiple dispatch allows for specializing any function with multiple methods using arguments different for number and/or type. Task-based control flows for parallel execution are natively provided, like booth cooperative multitasking and thread-based preemptive multi-tasking. Julia's multithreading-based model provides the ability to schedule Tasks simultaneously on more than one thread or CPU core, sharing memory. Summing up, Julia ease the implementation of complex algorithms by providing native support for concurrency and fast computation by design. To implement concurrency and multithreading is just a matter of following few style rules.

2.5.1. Generation of a sparse characteristic matrix

The remaining of the section aims to both give a very short synthesis of the Julia syntax, while introducing the LAR representation that we have used for our experiments and the prototype implementation of the algorithms discussed in the paper. The Julia syntax is very similar to python, but instruction blocks are not visually delimited, and require either explicit `end` or `begin ... end`. For the algebraic syntax with vectors and matrices Julia strongly resembles MATLAB. Julia is optionally typed, so the user may not decorate the declarations of parameters with explicit types. In most cases the compiler may infer them for code optimization.

Characteristic matrices The special usefulness of characteristic matrices $\boldsymbol{K}_p$ $(0 \leq p \leq d)$, resides in the fact that they have the same number $n = \#\Lambda_0$ of columns, for every p. The property may be used to multiply any two of them, after having transposed the second one. Following this operation with some simple "filtering" of non-zero values, this allows us to compute all the 3×3 binary relations between the three boundary objects `V`, `E`, `F` (DiCarlo et al., 2014b).

Algorithm 1 (Characteristic matrices)**.** .

```
# Sparse matrix generation from a set of triplets (i,j,x) of non-zeros, given as a triple (I,J,X) of arrays
function K( CV )
    I = vcat( [ [k for h in CV[k]] for k=1:length(CV) ]...)
        # vcat maps arrayofarrays to single array
    J = vcat( CV...) # splat operator ... transforms the array elements to function arguments
    X = Int8[1 for k=1:length(I)]
        # Type Int8 (capital) defines the memory map of array elements
    return SparseArrays.sparse(I,J,X) # sparse is a function of SparseArrays package
end
```

The so-called "characteristic matrices" are used in *unevaluated* LAR to denote the cells of a cellular complex as binary vectors, *i.e.*, as rows of binary matrices.

The code snippet of Algorithm 1 computes the characteristic matrix for any sequence `CV` of cells, represented "by-vertices", i.e., as array of arrays of vertex indices, as introduced in Example 2.5.1. We consider this one the simplest user-interface for the interactive input of a shape using a CLI (Command Line Interface). Of course, this format is also very simple to implement in a GUI for shape definition.

In particular, the Julia function `K` returns a `SparseArrays` matrix providing the characteristic $\boldsymbol{K}_r$ matrix, given as input an array `CV` specifying each r-cell as array of vertex indices. The embedding of cells, *i.e.*, the affine map that locate them in $\mathbb{E}^d$, will be specified by a $d \times n$ array `V`, where n is the number of 0-cells. See `V` in Example 2.5.1.

Example 2.5.1 (Cellular 2-complex data (continues on Example 2.5.2))**.** The input data to generate the geometric representation of the 2D cellular complex in 2.5.2 and Figure 5b follows. The embedding function $\gamma : U_0 \to \mathbb{E}^2$ has

Julia type `Matrix{Float64}`; the matrices of operators δ_0 and δ_1 are given here as arrays of array, and converted to CSC (Compressed Sparse Column) representation in LAR implementation:

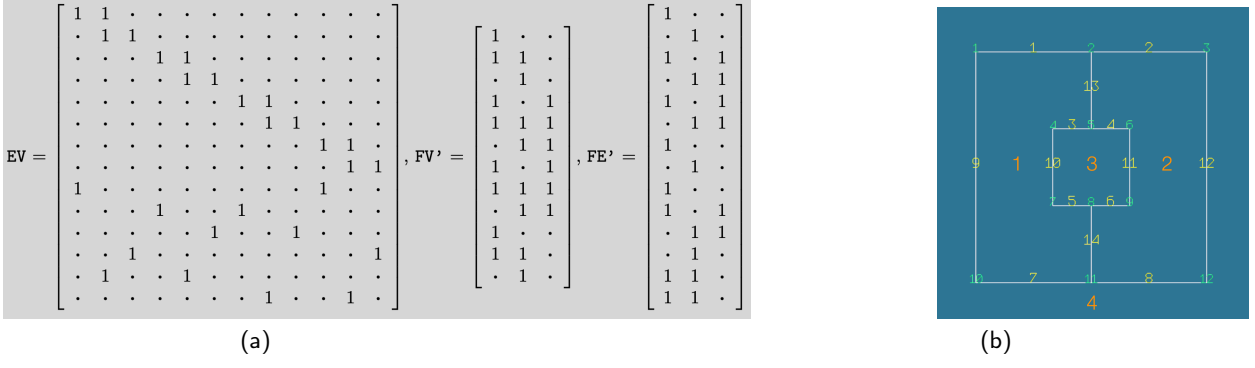
```
# γ : U_0 → E^2   12×2 full matrix of 2D vertex coords
V =  [0.0 1.5 3.0 1.0 1.5 2.0 1.0 1.5 2.0 0.0 1.5 3.0 ;
      0.0 0.0 0.0 1.0 1.0 1.0 2.0 2.0 2.0 3.0 3.0 3.0 ]

# δ_0 ≡ χ_{U_0}(U_1) := Edges as array of arrays of Vertex indices
EV = [[1,2],[2,3],[4,5],[5,6],[7,8],[8,9],[10,11],[11,12],
      [1,10],[4,7],[6,9],[3,12],[2,5],[8,11]]

# K_2 ≡ χ_{U_0}(U_2) := Faces as array of arrays of Vertex indices
FV = [[1,2,4,5,7,8,10,11],[2,3,5,6,8,9,11,12],[4,5,6,7,8,9]]
```

Example 2.5.2 (Characteristic matrices)**.** The sparse binary characteristic matrices K_1= `K(EV)` and K_2= `K(FV)` are generated from data of cellular 2-complex 2.5.2 using the Algorithm 1. Here we show the corresponding dense matrices, for the sake of readibility. Of course, the storage space of a sparse matrix is linear with the total number `nnz` of non-zero elements, and the sparsity, defined as $0 \leq 1 - \texttt{nnz}/nm \leq 1$, grows with the number nm of cells, *e.g.*, the `EV` array length. The apex symbol "'" stands for matrix transpose (more generally, for matrix adjoint).

```
julia> K_1 = K(EV)  #   ≡ [δ_0] = [∂_1^t]  sparse matrix from input EV to compute the 2D arrangement A(EV)
14x12 SparseMatrixCSC{Int8, Int64} with 28 stored entries: # output from Julia's REPL

julia> K_2' = K(FV)'  #   sparse matrix from input FV to compute the 2D arrangement A(EV)
12x3 SparseMatrixCSC{Int8, Int64} with 22 stored entries: # output from Julia's REPL

julia> EF = (K(EV) * K(FV)').÷2  #  (K_1 * K_2').÷2 ≡ [∂_2] = [δ_1^t]
14x3 SparseMatrixCSC{Int8, Int64} with 22 stored entries: # output from Julia's REPL
```

The

' stands for matrix transposition, ÷ for integer division, .÷ for broadcasting the application of the operator to all elements of array. Also, `SparseMatrixCSC{Int8,Int64}` is the type of output sparse matrices. `REPL`, that stands for (Read, Eval, Print, Loop), is the standard interactive interface for terminal interaction. The `REPL` prompt is `julia>`.



(a) (b)

Figure 5: (a) Sparse matrices (shown as dense) of topological operators $\delta_0 = \partial_1^t$, K_2^t, $\delta_1 = \partial_2^t$; (b) the 2D cellular complex generated by data of Example 2.5.1. The complex has $\gamma_0 = 12$, $\gamma_1 = 14, \gamma_2 = 4$, with 4 faces including the outer face, so that the Euler characteristic is $\gamma = 12 - 14 + 4 = 2$ as expected. Note that the exterior face (4) is included in the count.

2.5.2. Traversal of a shape hierarchy

Complex solid shapes are usually defined as hierarchical assemblies of either solid primitives or more complex shapes, each defined in a local coordinate system. Most graphics and modeling systems implement this semantics as a hierarchical graph, where affine geometry within the nodes is defined in local systems, and arcs are associated to affine transformations that move the whole subgraph rooted in the ending node onto the coordinate system of the first node of the arc. A well known recursive algorithm is used to traverse in preorder the graph, using a so-called CTM (Current Transformation Matrix) to bring all geometry in the (global) coordinates of the root (world coordinates).

A double traversal is executed by our computational pipeline (see Algorithm 2) when evaluating a CSG tree, where

geometry is normally on the leaves in local coordinates, and non-leave nodes may contain either affine transformations or Boolean operators to be applied to the rooted subtree.

Algorithm 2 (Traversal of hierarchical CSG forms)**.** .

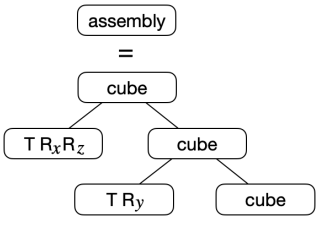
(a) DFS preorder traversal to get all solid objects (generators) in root coordinates, and
(b) DFS postorder traversal of the CSG expression to output a parsed syntax tree with symbolic placeholders (unbound Julia symbols) for algebra generators, whose Boolean coordinate vectors will be computed later in the computational pipeline.

In Algorithm 2 we split the transformation of coordinates from the evaluation of Boolean forms, and execute two combined Depth First Search (DFS) traversal. The algorithm is executed in both linear time and space, since the traversal is $O(n)$. The placeholder will be substituted by Boolean vectors of CSG algebra generators, to be computed later (see Algorithm 4) in the computational process.

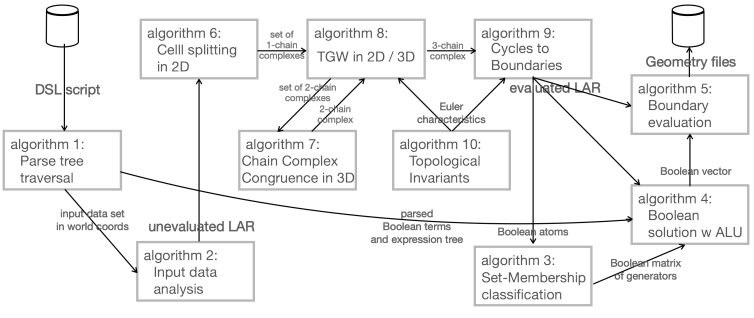
In our prototype implementation we make use of a hierarchical user-defined data-type called `Struct` (different from the Julia's `struct`). The semantics of `Lar.Struct()` is similar to that of PHIGS structures (Kasper and Arns, 1993; Paoluzzi, 2003). In particular, a function `struct2lar()` is applied to a hierarchical `assembly` in order to get a single unevaluated LAR model, used in the following to compute the space arrangement, and the whole data pipeline shown in Figure 10.

Example 2.5.3 (Space arrangement from `assembly` tree)**.** Consider the `assembly` constructed below by putting together three instances of the unit cube, suitably rotated and translated. The `Lar` constructor `cuboidGrid` of grids of cubes with "shape" `[m,n,p]`, returns the geometry `V` along with, if the optional parameter is `all=true`, the whole collection `VV`, `EV`, `FV`, and `CV` of p-cells, with $0 \leq p \leq 3$, represented "by-vertices". Only `V,FV,EV` (vertices, faces, edges) are actually needed by the Boolean generation in 3D. It might be useful to remember that the *shape* of a multi-dimensional array, in Julia, Python, and other computer languages, is a tuple or array with numbers of rows, columns, pages, etc. of data elements within the array. The *length* of shape tells the array dimensions: 1=vector, 2=matrix, etc. The variable `assembly` is bound to the Julia's immutable composite type generated by the function `Lar.Struct()`, which provides the syntax needed to define hierarchical geometric structures used to define the dimension-independent partition of the embedding space.

```
julia> using LinearAlgebraicRepresentation;
julia> m,n,p = 1,1,1;
    Lar = LinearAlgebraicRepresentation;
    V,(VV,EV,FV,CV) = Lar.cuboidGrid([m,n,p], all=true);
    cube = V,FV,EV;
julia> assembly = Lar.Struct([ cube,  # same semantics of PHIGS structures (Kasper and Arns, 1993)
    Lar.t(.3,.4,.25), Lar.r(pi/5,0,0), Lar.r(0,0,pi/12), cube,
    Lar.t(-.2,.4,-.2), Lar.r(0,pi/5,0), Lar.r(0,pi/12,0), cube ]);
julia> W, EV, FE, CF, boolmatrix = Lar.arrangement(assembly);
```



(a) Tree of the CSG `assembly`.



(b) A graphical depiction of the system design.

Figure 6: Graphical representation of the whole process to transform a CSG tree into files describing the computed geometry. See also Figure 10.

Let us note that `assembly` is a linearization of preorder DFS of the tree. The `Lar.arrangement()` function applied to `assembly` returns the `(geometry,topology)` of 3D space partition generated by it. `Geometry` is given by the embedding matrix `W` of old and new 0-cells, and `topology` by the sparse matrices `CF`, `FE`, `EV`, *i.e.*, by $\delta_2, \delta_1, \delta_0$, of the chain complex generated by the $\mathcal{A}$(`assembly`) arrangement. See section 2.4 (LAR Geometric Complex).

3. Solid Algebras and Boundary Chains

In this section we introduce and discuss a new algebraic method for boundary evaluation of any CSG (Constructive Solid Geometry) expression tree, constructed with regularized operators of union, intersection, and difference on a variable number of polyhedral d-solids (d = 2, 3). The result is obtained in five main steps (see also Figure 10):

1. By evaluating the input expression and producing an unevaluated LAR of generator primitives as well as the expression AST (Abstract Syntax Tree) to be used later, after having transformed the CSG form generators into binary strings in a Boolean array;
2. by computing an independent set of $(d-1)$-cycles generating the d-space arrangement induced by the input;
3. by reducing the given CSG expression into an equivalent Boolean expression with vectors of zeros and ones ('false' and 'true') associated to generators, without any reduction to canonical disjunctive normal form;
4. by evaluating the Boolean expression using bitwise native operators of Julia compiler, so producing a single resulting Boolean vector X to be interpreted as coordinate representation of a 3-chain;
5. finally, the binary vector of the Boolean result is used as the coordinate vector in linear space C_3 of a 3-polyhedron, and is converted to any standard b-rep (see Algorithm 5) using sparse matrix multiplications times the matrices of $\partial_3 : C_3 \to C_2$, $\partial_2 : C_2 \to C_1$, and $\partial_1 : C_1 \to C_0$.

Our method is implemented in Julia using sparse matrices and vectors: the computational evaluation of every possible solid expression with given solid generators is reduced to an equivalent logical expression of a finite set algebra over the cells of a spatial arrangement, and solved by native bitwise operators.

We show that the *structure* of each term of this algebra is characterized by a discrete set of points, each one computed once and for all in the interior of each atom. Set-membership classifications (SMC) with respect to such single internal points of atoms, computes the *structure* of any algebra term, and in particular transforms each solid variable (each term) of every Boolean formula, into a sparse *logical array* of length m, equal to the number of atoms.

Computing the product of two sparse general matrices (SpGEMM) is a fundamental operation in various combinatorial and graph algorithms as well as various bioinformatics and data analytics applications for computing inner-product similarities. In the last few years many optimized algorithms have been deviced on multi- and many-core architectures and GPUs.

3.1. Compendium of topological CSG method

For the sake of user convenience and understanding, we give here a paradigmatic summary of our method. For simplicity, we discuss few examples in 2D, that show the bulk of it, and are fairly readable by any scientific and technical reader. For this purpose, we need (a) a structure diagram (see Figure 7), to characterize and extract information contained in chain complexes, and (b) few simple Examples, in order to discuss our method step-by-step. Two primitive 2D solid *Ring* and *Rectangle* are shown in Figure 8, and both some CSG forms and the corresponding oriented b-reps are given in the following examples. Consider the boundary 1-cells e_i of input shapes to generate an arrangement of $\mathbb{E}^2$, as in Figure 8, according to (Paoluzzi et al., 2020). In Figure 8, the 0-cells, 1-cells, and 2-cells are denoted v_i, e_h and u_k, respectively.

From CSG forms to oriented B-reps Two chain complexes with linear spaces and linear transformations can be defined in $\mathbb{E}^2$ using either a binary or a ternary field of scalar coefficients:

(a) the *unsigned* linear spaces C_p ($p \in \{2, 1, 0\}$) over the binary field $\langle\{0, 1\}, +\bmod 2, \times\rangle$;

(b) the *signed* linear space $C_p^{\circlearrowleft}$ over the ternary field $\langle\{0, 1, -1\}, +\bmod 3, \times\rangle$.

We propose the diagram in Figure 7, where $\mathscr{A}(S)$, $\mathscr{P}[n]$, and $\mathscr{B}$ stand respectively for (a) the arrangement of $\mathbb{E}^2$ generated by a set S of primitive shapes, (b) the powerset of natural numbers $[n] := \{1, 2, \dots n\}$, where n is the number of Boolean atoms in $\mathscr{A}(S)$, and (c) the finite Boolean algebra generated by them. The following connected examples show some important computations over topological polyhedra in a 2D environment for simplicity.

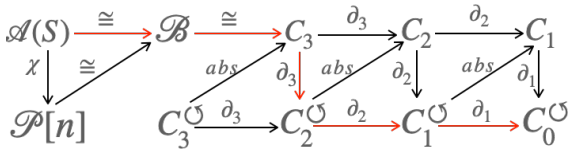
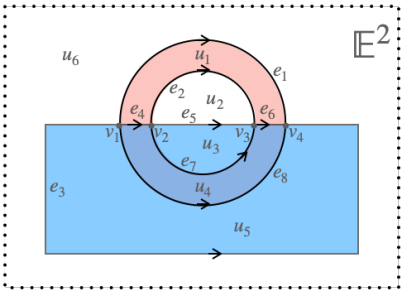


Figure 7: The diagram of functors between the unsigned chain complex $C_\bullet$ over a binary field of scalars, and the signed chain complex $C_\bullet^{\circlearrowleft}$ over a ternary field of scalars. In red the computational pipeline proposed in this paper.

Example 3.1.1 (From $\mathbb{E}^2$ arrangement to b-rep of algebraic forms)**.** In Figure 7 the same symbol is used for arrows ∂_p between signed or unsigned chain spaces. Of course, their matrix representations depend on the choice of the bases in both domain and codomain spaces. The path followed by our computational pipeline in 3D is drawn in red in the diagram. The matrix of mapping $\partial_2 : C_2 \to C_1^{\circlearrowleft}$ is shown on the right of Figure 8; it contains by column the basis elements of C_2 space represented in $Z_1^{\circlearrowleft} \subset C_1^{\circlearrowleft}$ subspace of 1-cycles, i.e. with coordinates in $\{0, 1, -1\}$. Some CSG forms and corresponding oriented b-reps are given in Example 3.1.2. Note in Example 3.1.3 that $\sum_{i=1}^{6}$ b-rep$(u_i) = 0$.

Example 3.1.2 (Boundary matrix of 2-complex)**.** It is easy to see that the $\mathbb{E}^2$ arrangement of Figure 8 produces the boundary matrix $[\partial_2] : C_2 \to Z_1$, from unoriented chains in C_2 to oriented 1-cycles in $Z_1 \subset C_1^{\circlearrowleft}$. A basis for the linear subspace of 1-boundaries $B_1 \subset Z_1$ is produced by Algorithm 9 and built in matrix $[\partial_2] : C_2 \to B_1$ by sum of columns corresponding to $u_2 + u_6$.



(a)

∂_2	u_1	u_2	u_3	u_4	u_5	u_6
e_1	-1					1
e_2	1	-1				
e_3					1	-1
e_4	1			-1		
e_5		1	-1			
e_6	1			-1		
e_7			1	-1		
e_8				1	-1	

(b)

Figure 8: (a) The arrangement of $\mathbb{E}^2$ generated by a solid rectangle in 2D and by a solid ring in 2D. (b) The boundary mapping ∂_2 between the basis $U_2 \subset C_2$ of unoriented 2-chain space and the subspace $Z_1^{\circlearrowleft} \subset C_1^{\circlearrowleft}$ of oriented 1-cycles. The sign-invariant domains of the quasi-disjoint partition of $\mathbb{E}^2$ is a finite Boolean algebra where atoms are indicated as $u_1, \ldots, u_6$.

It may be worthwhile to note that any Boolean term, say $Rectangle = u_3 \cup u_4 \cup u_5$ is also coordinate representation $[0, 0, 1, 1, 1, 0]$ of the corresponding *chain* in C_2, as well its *structure* in a finite Boolean algebra. In particular, we get the boundary transformations, shown in the following, from some simple CSG forms to oriented b-rep models. The orientation of basis elements $e_i \in C_1^{\circlearrowleft}$ is defined conventionally: e_i is positively oriented when: $e_i = v_k - v_h$, $k > h$.

$$\text{CSG}(rectangle) = u_3 \cup u_4 \cup u_5 = (u_3 + u_4 + u_5) \in C_2 \mapsto [\partial_2][0, 0, 1, 1, 1, 0]^t = [0, 0, 1, -1, -1, -1, 0, 0]^t$$
$$\equiv \text{b-rep}(rectangle) = (e_3 - e_4 - e_5 - e_6) \in B_1 \subset Z_1 \subset C_1^{\circlearrowleft}$$
$$\text{CSG}(ring) = u_1 \cup u_4 = (u_1 + u_4) \in C_2 \mapsto [\partial_2][1, 0, 0, 1, 0, 0]^t = [-1, 1, 0, 0, 0, 0, -1, 1]^t$$
$$\equiv \text{b-rep}(ring) = (-e_1 + e_2 - e_7 + e_8) \in B_1 \subset Z_1 \subset C_1^{\circlearrowleft}$$
$$\text{CSG}(rectangle \cap ring) = u_4 \in C_2 \mapsto [\partial_2][0, 0, 0, 1, 0, 0]^t = [0, 0, 0, -1, 0, -1, -1, 1]^t$$
$$\equiv \text{b-rep}(rectangle \cap ring) = (-e_4 - e_6 - e_7 + e_8) \in B_1 \subset Z_1 \subset C_1^{\circlearrowleft}$$
$$\text{CSG}(rectangle \cup ring) = u_1 \cup u_3 \cup u_4 \cup u_5 = (u_1 + u_3 + u_4 + u_5) \in C_2 \mapsto [\partial_2][1, 0, 1, 1, 1, 0]^t = [-1, 1, 1, 0, -1, 0, 0, 0]^t$$
$$\equiv \text{b-rep}(rectangle \cup ring) = (-e_1 + e_2 + e_3 - e_5) \in B_1 \subset Z_1 \subset C_1^{\circlearrowleft}$$
$$\text{CSG}(rectangle - ring) = u_3 \cup u_5 = (u_3 + u_5) \in C_2 \mapsto [\partial_2][0, 0, 1, 0, 1, 0]^t = [0, 0, 1, 0, -1, 0, 1, -1]^t$$
$$\equiv \text{b-rep}(rectangle - ring) = (e_3 - e_5 + e_7 - e_8) \in B_1 \subset Z_1 \subset C_1^{\circlearrowleft}$$

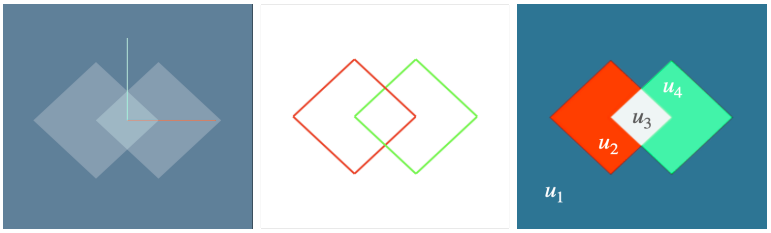
The linear space C_2 has dimension $6 = \#U_2$, equal to the number $M = 6$ of atoms of the solid algebra generated by the arrangement of $\mathbb{E}^2$ induced by the two primitive shapes *rectangle* and *ring*. The 1-cell representations of unit 2-cells are by column in $[\partial_2] : C_2 \to C_1$. M is also the length of bit strings giving their Boolean *structure*. The number of all distinct algebraic forms, i.e., the size of this solid algebra is of course 2^M.

Example 3.1.3 (Oriented boundary chain of basis elements)**.** With respect to Figure 8, the b-rep of each unit (i.e. basis) 2-chain (a.k.a. atom) is simply computed by (sparse) matrix-product of $[\partial_2]$ times the unit column vectors u_k ($1 \leq k \leq 6$) of each basis element, as shown in the following.
The b-reps of Boolean 2D atoms $u_k \subset \mathbb{E}^2$ and their coherently oriented boundary 1-cycles in C_1 are computed below:

$$\text{b-rep}(u_1) := \partial_2 u_1 = [\partial_2][u_1] \mapsto -e_1 + e_2 + e_4 + e_6 \qquad \text{b-rep}(u_4) := \partial_2 u_4 = [\partial_2][u_4] \mapsto -e_4 - e_6 - e_7 + e_8$$
$$\text{b-rep}(u_2) := \partial_2 u_2 = [\partial_2][u_2] \mapsto -e_2 + e_5 \qquad \text{b-rep}(u_5) := \partial_2 u_5 = [\partial_2][u_5] \mapsto e_3 - e_8$$
$$\text{b-rep}(u_3) := \partial_2 u_3 = [\partial_2][u_3] \mapsto -e_5 + e_7 \qquad \text{b-rep}(u_6) := \partial_2 u_6 = [\partial_2][u_6] \mapsto e_1 - e_3$$

3.2. Isomorphism between arrangements and algebras

The strong relationship beetween algebras and arrangements is well known—see, for example, Orlik and Terao (1994). Conversely, the authors do not know a similar correspondence between arrangements and (co)chains. Therefore, the major contribution of this paper is to demonstrate that there exists an homomorphism between the Constructive Solid Geometry with regularization (*i.e.* a finite solid algebra with operations of complement, union, intersection, and difference) and the linear space C_3 of (co)chains associated by the arrangement of Euclidean space induced by a collection of $(d-1)$-objects, with basis given by the columns of the matrix of linear map $\partial_3 : C_3 \to C_2$.



(a) (b) (c)

Figure 9: (a) Two input 2-complexes; (b) (4+4) 1-cells (line segments) in 1-skeletons of input, generate (c) *four* 2-cells in $\mathbb{E}^2$: blue (u_1); red (u_2); white (u_3); green (u_4). The element u_1 in basis $U_2 \subset C_2$ is the *outer* 2-cell Ω, complement of the union of the others.

Example 3.2.1 (2D space arrangement)**.** Generated in $\mathbb{E}^2$ by two overlapping single-cell 2-complexes. Figure 9c shows the arrangement of $\mathbb{E}^2$ into four subsets: the red region A, the green region B, the overlapping region $A \cap B$ and the outer region $\overline{A \cup B}$, *i.e.*, the rest of the plane. The set of *atoms* of the Boolean algebra $\mathscr{B}$ is one-to-one with this arrangement: the four regions are the four atoms of the $\mathscr{B}$ algebra, and there are $2^4 = 16$ distinct elements $S \in \mathscr{B}$. The *structure* of each element $S \in \mathscr{B}$ is a *union* of $\mathscr{B}$ atoms; as a chain in C_2, it is a *sum* of basis elements.

Example 3.2.2 (Boolean algebra)**.** Let us consider Table 1, that contains by columns the coordinate representation of the 2-chains A, B, and Ω in the chain space C_2. In algebraic-topology notations, with oriented chains:

$$A = u_2 + u_3, \quad B = u_3 + u_4, \quad \Omega = u_1 = -(u_2 + u_3 + u_4).$$

The Boolean algebra of sets is represented here by the power set $\mathscr{A} = \mathscr{P}(U_2)$, with $U_2 = \{u_1, u_2, u_3, u_4\}$, whose $2^4 = 16$ binary terms of length $\#U_2 = 4$ are given in Table 2, together with their semantic interpretation in set algebra. Of course, the coordinate vector representing the universal set, *i.e.* the whole topological space $X := \mathbb{E}^2$ generated by $[\partial_2]$ columns is given by $[X] = [1, 1, 1, 1]^t$, including the outer cell $\overline{A \cup B}$. Let us remember that the *complement* of A, denoted $-A$ or $\overline{A}$, is defined as $X \setminus A$ and that the *difference* operation $A \setminus B$ is defined as $A \cap \overline{B}$.

Table 1
Truth table (Example 3.2.2) associating the unit 2-chains $u_i \in U_2$ $(1 \le i \le 4)$, basis of linear space C_2, to rows of the table, and solid objects A, B and $\Omega = \overline{A \cup B}$ to columns. See Example 9c for images of the $\mathbb{E}^2$ space and the arrangement induced by colored 1-skeletons of the 2-complexes in Figure (b) of Example 3.2.1.

	Ω	A	B
u_1	1	0	0
u_2	0	1	0
u_3	0	1	1
u_4	0	0	1

Table 2
Truth table (Example 3.2.1) provides the complete enumeration of elements S of the finite Boolean algebra $\mathcal{A} = \mathbf{2}^4$ generated by two solid squares A, B and four atoms u_1, u_2, u_3, u_4. The same binary representation holds for coordinates of elements of chain space C_2 with basis $U_2 = (u_1, u_2, u_3, u_4)$. associated with the arrangement $\mathcal{A}(S)$, with $S = \{\partial(A), \partial(B)\}$.

U_2	$X = \mathbb{E}^2$	A	B	$A \cup B$	$\overline{A \cup B}$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \oplus B$	$\overline{A \setminus B}$	$\overline{B}$	$\overline{B \setminus A}$	$\overline{A}$	$\overline{A \oplus B}$	$\overline{A \cap B}$	$\emptyset$
u_1	1	0	0	0	1	0	0	0	0	1	1	1	1	1	1	0
u_2	1	1	0	1	0	1	0	0	1	0	1	1	0	0	1	0
u_3	1	1	1	1	0	0	1	0	0	1	0	1	0	1	0	0
u_4	1	0	1	1	0	0	0	1	1	1	0	0	1	0	1	0

3.3. Computational Pipeline

This section summarizes the whole computational process producing an evaluated LAR model of a CSG expression tree with values defined in Julia by: solid primitives; nested *assemblies* of data structures denoted as `LAR.Struct()`; affine transformations; and the macro `@CSG` including Boolean ops (see the Section A.2 in the Appendix).

3.3.1. Overview: models and representations

a. The set of *models* M of our representation scheme is the family of sets of piecewise-linear (PL) polyhedra even unconnected and/or non-manifold. Collections of 3-polyhedra are one-to-one with (coherently-oriented) 3-chain complexes, with p-chain bases of p-cells $(0 \le p \le 3)$ that are connected, but possibly non-manifold and non-contractible. They are also one-to-one with arrangements of $\mathbb{E}^3$ and hence with Boolean algebras of sign-invariant cells of such arrangements.

b. Given any collection of *cellular polyhedra* in this domain, the computation of induced $\mathbb{E}^3$ arrangement is summarized by:

   (a) extract the 2-skeletons of polyhedra and merge efficiently (via interval-trees) all their 2-cells;
   (b) compute in 2D the *local* chain complex $C_2 \leftrightharpoons C_1 \leftrightharpoons C_0$ on each 2-cell σ of the input data set by the TGW algorithm (Paoluzzi et al., 2020), given here as Algorithm 8;
   (c) identify via topological congruence (DelMonte et al., 2020) 3D multiple instances of p-cells $(0 \le p \le 2)$;
   (d) finally compute the *global* chain 2-complex partitioning the whole space $\mathbb{E}^3$, and the matrix $[\partial_2]$ of the boundary operator $\partial_2 : C_2 \to C_1$.

c. The TGW algorithm in 3D, with input the matrix $[\partial_2]$, produces the matrix $[\partial_3^+]$ whose columns are a basis of C_3 expressed through the target subspace Z_2 of 2-cycles. The cycles which are boundary of holes are linearly combined, to get the set of generators spanning $B_2 \subset Z_2 \subset C_2$, the subspace of 2-chains that are boundaries of 3-chains (see Figure 3), in particular of atoms of $\mathbb{E}^3$ partition, including the outer space.

d. The B_2 elements are one-to-one with the 3D subsets of a partition, i.e., an arrangement of $\mathbb{E}^3$ and this is one-to-one with the atoms of a Boolean solid algebra, produced by our original collection of PL polyhedra, *generators* of this algebra. Each generator ~~solid~~ has a *structure* made by disjoint union of some atoms, and is representable as a binary string, whose length equals the total number of atoms.

e. The *validity set* (the set of all *valid* representations) of our representation scheme is the linear space B_2, i.e., the subspace of 2-cycles that are boundary of some *coherently oriented* 3-chain (*i.e.* of some PL 3-polyhedra).

The *domain* of the scheme is $\mathrm{dom}(\delta_2|_{B_2})$, with $\delta_2 : C_2 \to C_3$, capturing the idea of solidity using oriented boundaries.

f. The set of 2^n binary strings with length n (the number of atoms) provides a Boolean algebra of sets. Here we have operations of union and intersection, as well as difference and complementation. This algebra is isomorphic to the Boolean Algebra of PL polyhedra provided by our generators, and hence isomorphic to the solid algebra that is called Constructive Solid Geometry (CSG).

g. In conclusion, every CSG expression tree may be converted into an equivalent expression, including binary vectors of length n, round brackets, Boolean operator symbols, and solved efficiently bitwise by the Julia optimizing compiler. The resulting binary vector $[x]$, the coordinate representation of a 3-chain within the linear chain space C_3, is converted by matrix multiplication $[\partial_3][x]$ into a 2-boundary in B_2, our b-rep representation of the CSG expression. This one may be converted into any external geometry format through multiplications with $[\partial_2]$ and $[\partial_1]$ matrices.

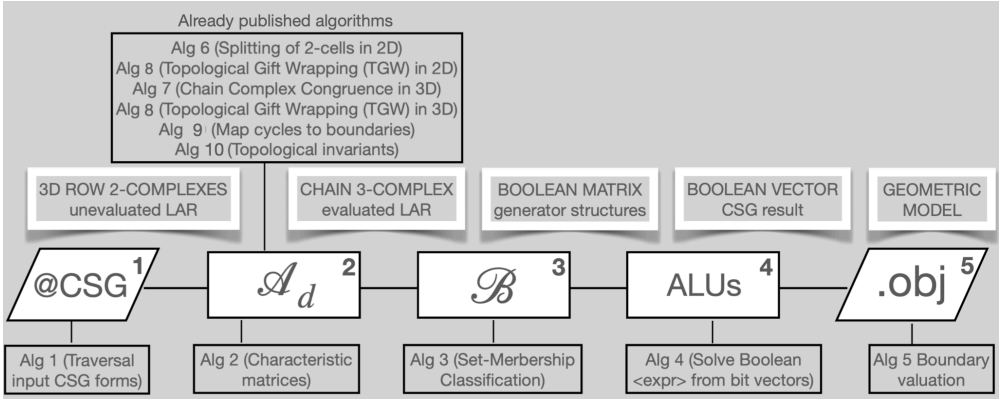


Figure 10: Evaluation pipeline (Algorithm 0) from Julia `@CSG` macros to a geometric model: (1) the evaluation process starts from a (hierarchical) `@CSG <expr>`; (2) computation of space arrangement; (3) evaluation of Boolean form generators into atomic binary sequences; (4) bitwise computing of Boolean result by available Arithmetic-Logical Units; (5) output of 3D geometric model. Algorithms 1-5 are in the main text; Algorithms 6-11 in Appendix A. See also Figure 6b.

Algorithm 0 (Evaluation Pipeline)

The evaluation of a 3D Boolean functional form `@CSG <expr>`, into an *evaluated* LAR model `(V,(CF,FE,EV))`, made by a $3 \times n$ array `V` and by a triple `(CF,FE,EV)` of sparse arrays, providing the geometric embedding $C_0 \to \mathbb{E}^3$, and a chain complex $\delta_2, \delta_1, \delta_0$, consists of several tasks (see Figure 10):

1. *Evaluation of data and functional forms*. **Input**: a single `@CSG` macro block, possibly delimited by `begin...end` clause, including solid terms and (nested) component `@CSG` clauses. **Output**: parse tree of compound `@CSG` expression; ordered tuple of variable names associated to input objects in a single coordinate frame: $(\mathtt{X}_1, \ldots, \mathtt{X}_N)$.
2. *Computation of space arrangement*. **Input**: tuple of N terms (generators) in world coordinates: $(\mathtt{X}_1, \ldots, \mathtt{X}_N)$. **Output**: sparse boundary matrices `(FC,EF,VE)` as $[\partial_3], [\partial_2], [\partial_1]$, and vertex matrix `V`. The columns of $[\partial_3^+]$, generated by algorithm 8 give all the irreducible cycles of $Z_2 \subset C_2$. Via Algorithm 9, (computing boundaries from cycles), the columns of $[\partial_3]$ are associated with the atoms spanning B_2 and generating the `@CSG` algebra.
3. *Evaluation of Boolean terms*. **Input**: the b-rep of M atoms of CSG algebra, generated by the chain complex `(V,(FC,EF,VE))`, and input $(\mathtt{X}_1, \ldots, \mathtt{X}_N)$ terms. **Output**: $M \times N$ matrix `B::Matrix{Bool}` whose column $\mathtt{B}_j$ gives the bit vector of length M (the number of atoms) representing the $\mathtt{X}_j$ generator of `@CSG` algebra.
4. *Bitwise computing of Boolean function*. **Input**: the `Bool` matrix `B`; the parse tree of `@CSG` expression. **Output**: a single `BitArray` of length M. The $1s$ in this array denote the unit 3-chains (atoms) whose quasi-disjoint unions (generators) produce the Boolean solution vector, *i.e.*, the binary *value* of the evaluated `@CSG` form.
5. *Boundary evaluation of output 3-chain*. **Input**: the 3-chain (binary) coordinate representation `X` of the Boolean result; the chain complex `W,(FC,EF,VE)`. **Output**: the boundary representation of the Boolean result, possibly exported in a standard vector graphics format `*.<ext>`, like, e.g., the file formats `*.OBY`, `*.PLY`, or `*.DAE`. Our software prototype may currently export in files `*.OBY`, both 2D and 3D.

The final stage of our approach to constructive solid geometry consists in generating a representation of solid arguments as linear combinations of unit 3-chains, *i.e.*, of 3-cells of the space partition generated by the input. The U_3 basis of 3-chains is represented in $C_2^{\circlearrowleft}$, i.e. the linear space of chains over the ternary field of coefficients $\{0, 1, -1\}$ by the columns of the ∂_3 matrix. See Section 3.1 for detailed computational example. This set of 2-cycles can be seen as a collection of point-sets, including the whole outer space and the empty set. In this sense, it generates both a discrete topology of $\mathbb{E}^3$ and, via the Stone Representation Theorem (Stone, 1936), a finite Boolean algebra $\mathscr{B}$ over C_3 elements.

3.3.2. Set-Merbership Classification (SMC)

The LAR representation of atoms, i.e., join-irreducible elements of distributive lattice $\mathscr{L}(S) \cong \mathcal{A}(S_{d-1})$ as discrete point-sets (see Section 2.4), is used to map the *structure* of each term $X \in S$ to the *set algebra* $\mathscr{B} \cong \mathcal{A}(S)$. Assume that: (a) $X \in S$; (b) a partition of $\mathbb{E}^d$ into subsets $A_i \in \mathcal{A}(S)$ is known; (c) a one-to-one mapping $A_i \mapsto p_i$ between atoms and single internal points is available; (d) a SMC oracle, *i.e.*, a set-membership classification (Tilove, 1980) test is available.

A naive approach to SMC, where a single point p_k (in the interior of the atom $A_k \in \mathscr{L}(\mathcal{H})$) is tested against *all* input solid terms $X \in \mathcal{H}(S)$ may be computed in quadratic time $O(MN)$, where M is the number of atoms, and N is the number of input solid terms X. An efficient $O(M \log N)$ procedure is developed by using two one-dimensional interval-trees for the arrangement $\mathcal{A}(S_{d-1})$ of $\mathbb{E}^3$, in order to execute the SMC test only against the terms in the subset of elementary 2-cells in $\mathcal{I}_i \subseteq U_2 \subset C_2$, whose containment boxes intersect the vertical ray_i from the test point p_i.

Algorithm 3 (Set-Membership Classification)**.**

Require: $\mathscr{A} := \{A_i\}; \quad i \in \{1 \dots M\}$ ▷ Atoms of space arrangement ≡ basis of Boolean algebra
Require: $S := \{\partial X_j\}; \quad j \in \{1 \dots N\}$ ▷ Collection of shapes, generators of $\mathscr{A}(S)$ space arrangement
Ensure: $\mathscr{B} := B[i, j] \in \{0, 1\}; \quad (i, j) \in M \times N$ ▷ Structure matrix in $\mathscr{B}$ Boolean algebra of input shapes
1: $B \leftarrow zeros(\texttt{Int}, M, N)$ ▷ initialize output matrix
2: **for all** $i \in M$ **do**
3: $p_i \leftarrow \mathbf{i}A_i$ ▷ compute internal point to $atom[i]$
4: $ray_i \leftarrow \text{int−tree}(x_i) \cap \text{int−tree}(y_i)$ ▷ Vertical ray by intersection of two queries on interval-trees
5: $\mathcal{I}_i \leftarrow ray_i$ ▷ $\mathcal{I}_i \subset S$ of input 2-cells of possible intersection with ray_i
6: **for all** $\sigma \in \mathcal{I}_i$ **do** ▷ input 2-cell $\sigma \in \mathcal{I}_i$ of possible intersection
7: **if** $(\pi(\sigma) \cap ray_i \in \mathbf{i}\sigma) \wedge (\sigma \in \partial X_j)$ **then** ▷ $\sigma \cap ray_i$ is within σ interior on X_j b-rep
8: $B[i, j] \leftarrow B[i, j] + 1$ ▷ increment intersection counter of ray_i with input shape X_j
9: **end if**
10: **end for**
11: **end for**
12: $B \leftarrow B \,.\%\, 2$ ▷ Vector transform B::Bool ← B::Int

The *structure* of term X in the finite algebra $\mathscr{B}(S)$ can be computed using a *generate-and-test* procedure. Such a SMC test is simple and does not involve the resolution of "on-on ambiguities" (Tilove, 1980), because of the choice of using an *internal* point p_i in each atom A_i. Set Membership Classification (SMC) is executed via a ray-polygon intersection test in 3D, actually through ray-plane intersection, followed by point-polygon-containment test in 2D. The topological operator **i** return a generic interior point of the point-set it is applied to.

In our current implementation in 3D, the SMC test is executed by intersecting a ray from p_i with the planes containing the 2-cells of atoms in $\mathcal{I}(p_i)$, and testing for point-polygon-containment in these planes (via maps to the $z = 0$ subspace). In summary, we decompose $\mathbb{E}^d$ into join-irreducible elements A_i of algebra $\mathscr{A}(S)$, and represent each A_i via a single point p_i. By the Jordan curve theorem, an odd (transversal) intersection number of the *vertical* ray for p_i with boundary 2-cells of X_j produces an oracle answer about the query statement $p_i \in X_j$. If `true`, the intersection counter $B[i, j]$ is incremented. At the very end, the (sparse) integer matrix B is transformed to binary via the remainder operator (%) applied element-wise (dot operator ".") to all B matrix elements. An odd number i implies $A_i \subseteq X_j$. An even number i gives the converse result.

The procedural SMC algorithm works once for each atom A_i, and computes both a single internal point p_i and the subset of generators 2-cells $\mathcal{I}_i$ whose box is intersected by ray_i. Using the intersection of two queries (on x_i and y_i of

p_i) over two interval trees, many generators X_j may be excluded in $O(M \log N)$ time, detailed tests on planes of faces being executed only when the surely negative answers of intersection were already excluded.

3.3.3. Binary representation of Boolean terms

In practice, we construct a binary representation (of the structure) of each term X of a solid Boolean expression, as a subset of U_3 (basis of 3-chains). Remember that, by construction, U_3 partitions both $\mathbb{E}^3$ and the input solid objects.

Bit vectors A subset Y of X can be identified with an indexed family of bits with index set X, and the bit indexed by $x \in X$ being 1 or 0 according to whether or not $x \in Y$. The Boolean algebra $\mathscr{B} = \mathscr{P}(X)$ of the power set of X can be defined equivalently as a set of *bit vectors*, all of the length $n = \#X$, with $\#\mathscr{B} = 2^n$.

Mapping generators to bit vectors To translate a CSG formula $\mathcal{H}$ including a solid term X to machine language it is sufficient (once computed the $[\partial_3]$ matrix, and hence the U_3 basis) for each unit 3-chain $u_k \in U_3$, to test for *set-membership* a single *internal* point $p_k \in u_k$, by checking if $p_k \in X$. In the affirmative case the k-th bit of coordinate vector $[X] \in \mathbf{2}^M \cong \mathscr{P}(U_3)$ is set to `true`. Of course, this translation may be done in parallel for all atoms A_k in the space arrangement $\mathcal{A}(U_3)$ (see Algorithm 3).

Abstract Syntax Tree (AST) This paragraph is mainly extracted from Julia developer's documentation. In Julia, an AST is a lisp-like expression returned by the parser and manipulated by macros. Front-end ASTs consist almost entirely of `Exprs` and atoms (e.g. symbols, numbers). Expressions (`Exprs`) are represented as parenthesized lists using operators and/or symbols. Some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. Some operators (+ and *, e.g.) use N-ary parsing; chained calls are parsed as a single N-argument call. They are immutable and 'interned', i.e., hashed by the language implementation. Julia and Lisp have in common the ability to represent the language's code as a data structure in the language itself (homoiconicity). Symbols in Julia are the same as in Lisp or Scheme. In particular, a symbol is a way to represent a language variable in metaprogramming. In order to construct expressions in Julia that use variables, the AST use unbound symbols, that are bound in this project by Algorithm 4, that provides a fairly high view of machine-level operations.

Algorithm 4 (Mapping generators to characteristic vector of CSG solution)**.**

Require: $parsetree$ :: `AST` $\wedge$ $BoolMatrix$ :: `SparseMatrix{Bool}{Int}` ▷ parsed expression tree *and* Boolean generators as columns of input $BoolMatrix$
Ensure: $boolvector$ :: `SparseVector{Bool}{Int}` ▷ Binary output value of input CSG expression
  $vars$::`Vector{Symbol}` $\leftarrow extractsymbs(parsetree)$ ▷ extract unbound symbols from AST
  for all $j, boolvect \in enumeratecols(BoolMatrix)$ **do** ▷ for all columns in BoolMatrix
    $vars[j] \leftarrow boolvect$ ▷ bound Boolean generator vectors to the symbols of variables in AST
  end for
  $IRcode \leftarrow$ `just-in-time (JIT) compilation of bound AST` ▷ generation of intermediate code (IR)
  $boolvector \leftarrow$ `ALUrun(`$IRcode$`)` ▷ bit-wise computation of Boolean solution of initial CSG expression

Example 3.3.1 (Boolean matrix)**.** The variable `boolmatrix` (see last expression of Example 2.5.3) is a `Matrix` of 8×4 `Bool` values, and maps the 8 atoms of the $\mathbb{E}^3$ arrangement to the 4 generators of the associated solid algebra.

In other words, `boolmatrix` contains by column the results of Set Membership Classification (SMC, Algorithm 3) for a single internal point of each 8 atomic 3-cells (rows) of the $\mathbb{E}^3$ partition, against the outer space Ω and each $\mathtt{K}_1, \mathtt{K}_2, \mathtt{K}_3$ cube instance (matrix columns). Of course, each cube was suitably mapped to world coordinates by `Struct` evaluation. Also, note `boolmatrix[1,1] = true` by the containment of a single its point in the first atom $\Omega \equiv$ outer space of the arrangement. The remaining elements of the first row and column are all `false` because the same point is contained neither in $\mathtt{K}_1, \mathtt{K}_2, \mathtt{K}_3$, nor in any A_i atom ($i > 1$). The sparse `boolmatrix` is shown as dense `Matrix` below.

```
julia> Bmat = Matrix(boolmatrix)
8x4 Matrix{Bool}:
true false false false
false false false true
false true true false
false true true true
false true false false
false false true false
false true false true
false false true true
```

Example 3.3.2 (Boolean generators)**.** Continuation of example 3.3.1. Our cube instances $\mathtt{K}_1, \mathtt{K}_2, \mathtt{K}_3$ are extracted here from `boolmatrix` columns into variables `A,B,C`, so describing how each one is partitioned by (ordered) 3-cells in U_3. The whole space is given by $X = \Omega \cup \mathtt{A} \cup \mathtt{B} \cup \mathtt{C}$. The reader may check that bit-wise union of all `Bmat` columns is `true`.

```
julia> A,B,C = boolmatrix[:,2], boolmatrix[:,3], boolmatrix[:,4]
(Bool[false, false, true, true, true, false, true, false],
 Bool[false, false, true, true, false, true, false, true],
 Bool[false, true, false, true, false, false, true, true])
```

3.3.4. Bitwise resolution of set algebra expressions

When the input initial solids have been mapped to arrays of Booleans, now called `A`, `B`, and `C`, any expression of their finite Boolean algebra is evaluated by logical operators, that operate by comparing corresponding bits of variables.

Bitwise operators In particular, the Julia language offers bitwise logical operators *and* (`&`), *or* (`|`), *xor* ($\veebar$), and *complement* (`!`), as well as a dot mechanism for applying elementwise any function to arrays. Hence, we can write expressions like `(A .& B)` or `(A .| B)` that are evaluated *bitwise* and return the result in a new `BitArray` vector variable. We remark that these operators can be used also in *prefix* and *variadic* form. Hence, bitwise operators can be applied at the same time to any finite number of variables. Parallel application is implicit in the language.

Example 3.3.3 (Boolean formulas)**.** Continuation of examples 3.3.1 and 3.3.2. Some assignment statements follow, in order to show simple examples of Julia's syntax with Boolean expressions and vectors (1-dim arrays). In particular, the `AminBminC` variable contains the value of the intersection of the first term `A` with complements of other terms `B` and `C`, with `A`, `B`, `C` of Example 3.3.2, giving the set difference `(A\B)\C`. In other words, the variable `AminBminC` contains the result of the set difference denoted `&(A,` $\overline{\mathtt{B}}$, $\overline{\mathtt{C}}$`)`, mapped into the geometric model shown in Figure 11, made by a single atom (see variable `AminBminC` in the last line below).

```
julia> AorB = A .| B;
julia> AandB = A .& B;
julia> AxorB = A .⊻ B;
julia> AorBorC = A .| B .| C = .|(A, B, C);
julia> AandBandC = A .& B .& C = .&(A, B, C);

julia> AminBminC = A .\ B .\ C = .&(A, .!B, .!C)
8-element BitArray{1}:
[false, false, false, false, true, false, false, false]
```

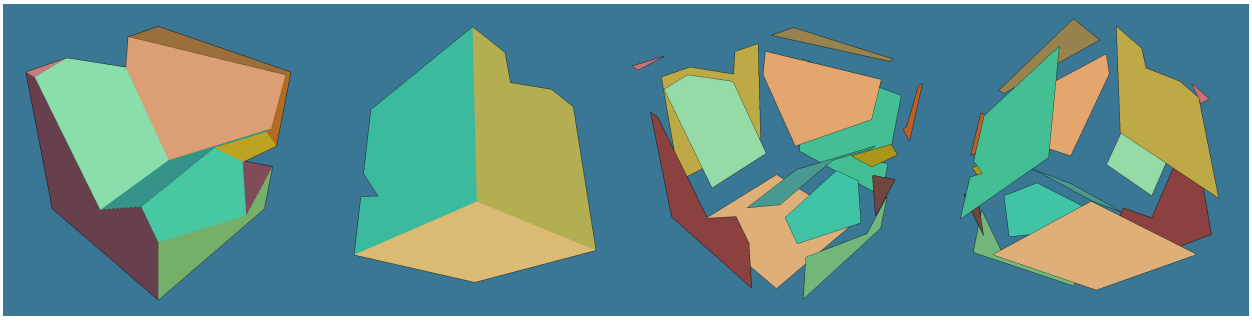
3.4. Boundary computation

Remember from Section 2.1.2 the notation $C_\bullet$ for the chain complex over the binary field $\{0, 1\}$, and $C_\bullet^{\circlearrowleft}$ for the *oriented* chain complex over the ternary field $\{0, 1, -1\}$, to get oriented boundaries. Actually, the boundary matrices generated by the TGW algorithm 8 are maps $C_3 \to C_2^{\circlearrowleft}$ in 3D, and $C_2 \to C_1^{\circlearrowleft}$ in 2D.

In most cases, the target geometric computational environment is able to display—more in general to handle—a solid model only by using some boundary representation, typically a triangulation. It is easy to get such a representation by multiplying the matrix of 3-boundary operator $\partial_3 : C_3 \to C_2^{\circlearrowleft}$ times the coordinate vector in C_3 space of the solid expression, say `X`, computed as a binary term of our set algebra. Once obtained in this way the signed coordinate vector of the solid object's boundary, *i.e.*, the 2-chain made by its oriented 2-cells (faces), these must be collected by columns into a sparse "face matrix", and translated to the corresponding matrix of oriented 1-cycles of edges, by right multiplications of $[\partial_2]$ times the face matrix.

Property 3.4.1 (Storage space of LAR Geometric Complex)**.** The topology of a LAR 3-complex is fully represented by the operators $\delta_2, \delta_1, \delta_0$, *i.e.*, by the sparse arrays `(CF,FE,EV)`, providing the incidences between vertices, edges, faces, and 3-cells (atoms) for both b-reps and cellular representations. If a b-rep `(FE,EV)` is used, LAR storage is 1/2 of the *winged-edge* representation by (Baumgart, 1972), often used as a storage standard (Woo, 1985) for solid modeling, and equal ($2 \times$ `FE`) to *half-edge* (Muller and Preparata, 1978), widely used in Computational Geometry, of course counting over the same cells.

It may be useful to remark that the triangulation of faces, given here at the end of the explicit computation of boundary elements in Algorithm 5, is *only* needed for display purpose, and often left to the graphics hardware.



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

Figure 11: Boundary surface of Boolean difference $(A \backslash B) \backslash C$ of three cubes, with 2-cells in different colors: (a) view from the front; (b) view from the back; (c) front with exploded 2-cells; (d) back with exploded 2-cells. Note that 2-cells of the resulting boundary may be non-convex.

Algorithm 5 (Boundary evaluation)**.**

Require: ∂_1, ∂_2, ∂_3 :: `SparseMatrix{Int}` $\vee$ `X` :: `SparseVector{Int}` $\neq 0$
Ensure: `CSG_model` :: `.obj`

```
∂1, ∂2, ∂3                                  ▷ chain complex input from computation of E^3 arrangement
X                                           ▷ input from julia evaluation of Boolean CSG form
triangleSet ← []
boundaryChain ← ∂3 * X                      ▷ boundaryChain ∈ B2 ⊂ C2
for all i,c ∈ enumerate(boundaryChain) do     in it for all unit 2-chain in it
    z[i] ← c                                ▷ coord repr of a 2-cell: sparse zero vector with z[i]=c
    faceChain ← ∂2 * z                      ▷ faceChain ∈ B1 ⊂ C1
    triangleSet ← triangleSet ∪ triangulate(ordered(∂1 * faceChain))
end for
writeln(CSG_model.obj, triangleSet)
display(CSG_model.obj)
```

The last stage of the computational pipeline (see Figure 10 and Algorithm 5) starts from the Boolean solution X of the binary form generated by Julia evaluation of the target CSG form, after substituting each column of the Boolean structure matrix B to the corresponding generator object of the algebra $\mathscr{B}(\mathcal{A})$ induced by the space arrangement $\mathcal{A}(S)$. Remind that S is the collection of geometric objects (2-chains) generated in world coordinates by the traversal of a hierarchical `assembly` of geometric primitives (Paoluzzi et al., 2020).

The solution vector X depends of course on the syntax tree of the Boolean formula generated by Algorithm 2 in Section 2.5.2, which produced both geometry in world coordinates and the target syntax tree. The vector X is computed element-wise (bit-to-bit) by the hardware of the ALUs, or bitwise in parallel on many-core architectures like GPUs.

Example 3.4.1 (Boundary of solid expression)**.** Continuation from Example 3.3.3. The Julia variable `AminBminC`, from which we start, contains the Boolean vector computed at the end of Example 3.3.3. Here we show the very simple Julia computations needed to generate the oriented boundary polygons (2-cells) of the solid solution of the CSG expression `A - B - C`, displayed in Figure 11, being the values in `A,B,C` already computed in Example 3.3.2.

First, the value of the variable `AminBminC` is converted (from Boolean to binary) and stored within the binary vector `difference`—the coordinate representation of a *3-chain*—by the vectorized constructor "`Int8.`". Then the *oriented boundary 2-cycle* of the Boolean expression is computed and stored in a variable (named `boundary`) through multiplication times the boundary matrix $[\partial_3] \equiv$ `CF'`.

Let us recall that `CF` (*i.e.*, Faces $\rightarrow$ Cells) is the sparse matrix of coboundary operator $\delta_2 : C_2 \rightarrow C_3$, and that $[\delta_2]^t = [\partial_3]$, which in Julia is `CF'`.

```
# X ≡ A - B - C = AminBminC
# difference equal to intersection of complements
julia> AminBminC = .&(A, .!B, .!C)
8-element BitArray{1}:
[false, false, false, false, true, false, false, false]
```

```
julia> difference = Int8.(AminBminC)
8-element Array{Int8,1}:
[0, 0, 0, 0, 1, 0, 0, 0]

julia> boundary = CF' * difference
47-element Array{Int8,1}:
[1, 0, -1, 0, 0, 0, -1, 0, 1, 0, 1, 0, 0, 0, 1, -1, 0, -1, 0, -1, 0, 0, 1, 0, 0, 0, -1, 0, 0,
     0, 0, 0, -1, 0, 0, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 0, 0]
```

Finally, we remind that in oriented chain notation it is possible to write the oriented boundary of solid `difference` using the following expression, where f_k stands for kth face (2-cell, or unit 2-chain) of the computed $\mathbb{E}^3$ arrangement:

$$\text{boundary} \mapsto f_{A\setminus B\setminus C} = f_1 - f_3 - f_7 + f_9 + f_{11} + f_{15} - f_{16} - f_{18} - f_{20} + f_{23} - f_{27} - f_{33} + f_{38} - f_{43}$$

The actual reduction to the triangulated boundary (needed only for display) is obtained using also `FE'` and `EV'` sparse matrices and a constrained Delaunay triangulation (CDT) algorithm (see Algorithm 5).

Remark (From solid chains to B-reps). Of course, the 14 non-zero elements in the `boundary` coordinate array of the `AminBminC` variable, provide the coherently oriented boundary 2-cells of the solid result $X \in C_3$, stored in `AminBminC`. Each of them is transformed into a (possibly non connected) 1-cycle by the `FE'` sparse matrix, *i.e.*, by the 2-boundary matrix $[\partial_2] = [\delta_1]^t$. Finally, every 1-cycle is transformed into one or more sequences of 0-cells, using the `EV'` matrix, *i.e.*, by using $[\partial_1] = [\delta_0]^t$. The 0-cells are then cyclically ordered, and used to generate sequences of 3D points via the embedding matrix `V`, which provides the vertex coordinates by column. This last step gives the ordered input for face triangulation using a CDT in 2D (Constrained Delaunay Triangulation) algorithm (Shewchuk, 2002).

4. Conclusion

In this section we provide a digest of the scope of the paper and our technical contributions. A simple low-dimensional (2D) review of the topological method proposed was given in Section 3.1.

4.1. Summary

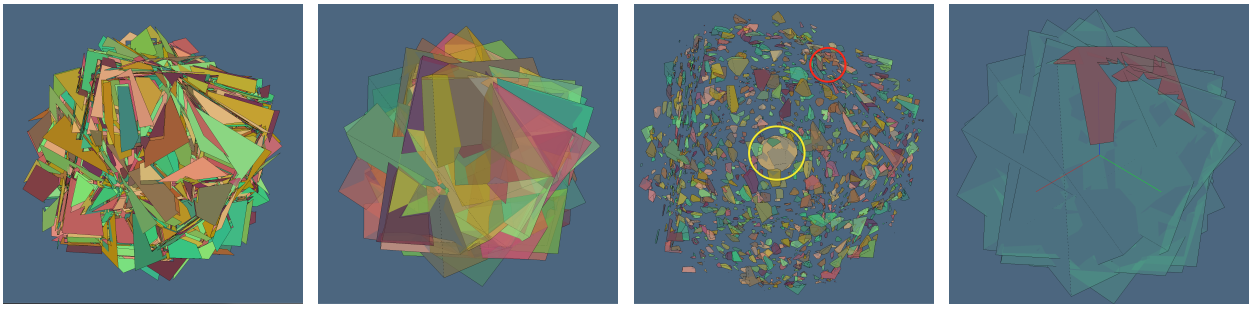
The *domain* of our representation scheme is currently restricted to the Boolean Algebra of PL polyhedra, which can be also disconnected and/or non-manifold. The elements of this algebra are in one-to-one correspondence with (coherently-oriented) 3-chains complexes, using basis p-chains (p-cells, $p = 0, \dots, 3$) that are connected, but possibly non-manifold and non-contractible. While the restriction to PL solids simplifies the initial implementation, the proposed algebraic approach is topological in nature, and does not depend on shape linearity. For instance, Example 3.1.2 uses topological polyhedra with non linear geometry.

Given any collection of *models* in this domain, using the approach from Paoluzzi et al. (2020), we construct the evaluated LAR (Linear Algebraic Representation) representation of their arrangement in E^3, including the 3-chain complex with linear ∂_p operators represented by sparse matrices. In particular, the columns of matrix $[\partial_3]$, including the boundary cycles of the exterior space, supply a basis of the linear subspace Z_2 of 2-cycles. Some irreducible 2-cycles in Z_2 are linearly combined to get a basis $B_2 \subset Z_2 \subset C_2$ of bounding 2-cycles, spanning the subspace of 2-chains that are boundaries of 3-chains.

We have shown that the 3-cells and elementary (basis) 3-chains in LAR are in one-to-one correspondence to the atoms in the Boolean algebra of closed regular sets (CSG representations), and that the elements of B_2 are in one-to-one correspondence with the boundaries of such sets. It follows that each 3D solid in this Boolean algebra may be represented either by a bit string indicating its structure in terms of atoms, i.e by a coherently oriented 3-chain, or by its oriented bounding 2-cycle, via multiplication with the matrix of $\partial_3 : C_3 \to C_2$.

Finally, we have described a fully implemented procedure for computing the boundary representation (bounding 2-cycle) of any variadic CSG expression in this algebra — a variadic function being a function of indefinite arity, i.e., one which accepts a variable number of arguments. The algebra generators are first converted into equivalent binary strings representing their Boolean atoms, round brackets to denote explicit precedence in CSG expression, and Boolean operator symbols, then solved efficiently by the Julia optimizing compiler. The resulting binary vector X, coordinate representation of a 3-chain in chain space C_3, is converted by matrix product $[\partial_3]X$ into a 2-cycle in B_2, the oriented boundary representation of the CSG expression, and finally may be converted into any given geometry format.

Example 4.1.1 (Solid algebra 3D)**.** The arrangement of $\mathbb{E}^3$ shown in Figure 12 is a 3-complex with 2208 vertices, 5968 edges, 5360 faces, and 1600 solid cells. The Euler characteristic χ is $\chi_0 - \chi_1 + \chi_2 - \chi_3 = 2208 - 5968 + 5360 - 1600 = 0$. This count includes the outer (unbounded) 3-cell. We recall that the Euclidean d-space is topologically equivalent to the d-sphere minus one point. the Euler characteristic of the d-sphere is $\chi = 1 + (-1)^d = 2$ or 0, for either even or odd space dimension d. It is worthwhile considering the complex shape of some 3-cells and 2-cells (red circle in Figure 12c-d). Such complicated shapes Figure 12d) are handled by LAR as straightforwardly (unordered list of vertex indices) as triangles and tetrahedra in 3D.



(a) Fragmented faces after the 2-cell splitting;

(b) Solid 3-cells assembled to give the solid union;

(c) Exploded set of atoms of Boolean algebra associated to this arrangement;

(d) One single (very complex) 3-cell evidenced.

Figure 12: Arrangement of $\mathbb{E}^3$ produced by 8 concentric unit cubes randomly rotated about the origin. The input data S_2 is made by $6 \times 8 = 48$ square 2-cells. The input configuration is close to the worst case $O(n^2)$ for Boolean operations, since every 2-cell is intersected by most of the other ones. The central largest atom signed in yellow in subfigure (c) is the intersection of all the cubes, and is an approximated 3-ball.

4.2. Significance and Outlook

Solid Modeling was conceived by engineers as a rigorous and universal language for geometry-based engineering. Mathematically, at its core, all solid models are computer representations of elements of some algebraic systems that are constructed and maintained by algorithms corresponding to the operations in the respective algebras. While this view is widely accepted as a good practice in solid and geometric modeling, it is rarely explicitly articulated or used directly to design data structures and algorithms. Furthermore, when multiple representations are used simultaneously, the relationships between them is often relegated to heuristic and ill-defined notions of “representation conversion," “translations,” and “interoperability" – undermining the rigor and robustness of solid modeling systems. In contrast, the results in this paper demonstrate that algebraic properties can be used directly to construct computationally efficient representations and algorithms, as well as to maintain correspondence between different representations. As a result, all higher level operations and applications correspond to formal algebraic expressions in the corresponding algebraic systems, increasing expressive power and improving the robustness of the implementations.

This paper, in particular, focuses on the algebraic relationships between CSG and boundary representations of PL solids, as summarized in the diagram in Figure 7. We show that the (regularized) arrangement of a given set of primitives is isomorphic to the finite Boolean algebra of regularized sets containing all possible CSG representations, which in turn is isomorphic to the linear vector space of 3-chains C_3 over the binary field $\langle\{0, 1\}, +\bmod 2, \times\rangle$. The latter are transformed by the boundary operation ∂_3 into oriented 2-boundaries in $B_2 \subset C_2^{\mathcal{O}}$ over the ternary field $\langle\{0, 1, -1\}, +\bmod 3, \times\rangle$ that corresponds to usual boundary representations. Based on these algebraic relations, any and all operations on CSG and boundary representations reduce to compositions of operations within the respective algebras and isomorphims between the algebras. In particular, we show that boundary evaluation of any CSG representation with a finite number of primitives reduces algebraically to a composition of Boolean operations on bit strings, sparse matrix multiplication, and point membership tests – without sacrificing efficiency or requiring specialized geometric data structures. By sake of completeness, it is worthwhile to remark that our topological methods, while defined over PL polyhedra by simplicity of prototypical implementation, should equally hold over chains of curved cells.

Our results point to a new research direction in solid modeling, where high-level applications are formulated algebraically and automatically compiled (via algebraic transformations) into composition of standard scientific com-

putations that are efficiently implemented using high-performance programming languages and are guaranteed to be correct by construction. The present research may be extended in many directions, in particular with general curved solid, whose incidence relations can be represented by multi-graphs and sparse arrays, and cells are piecewise non-linear. The methods introduced in this paper are applicable to topological polyhedra, and shown here mostly PL only as the follow-up of many experiments and PL prototypical implementations. To extend our topological approach to wider classes of CSG algebras, say, with curved boundaries, two additions are needed. In particular, to substitute our embedding mapping of 0-cells with a general strategy for embedding curved 0-, 1-, and 2-cells and chains. Of course, our face-vs-faces intersection method (Paoluzzi et al., 2020) should be suitably extended, making use of the wide body of knowledge accumulated in curved solid modeling in the past forty years.

Acknowledgement

We are grateful to the three anonymous reviewers for careful reading of the manuscript and sharing their time, efforts, and insights during the two rounds of revisions. Of course, any omissions and errors are our own.

References

Armstrong, C., Bowyer, A., Cameron, S., Corney, J., Jared, G., Martin, R., Middleditch, A., Sabin, M., Salmon, J., Woodwark, J., 1999. Djinn: a geometric interface for solid modelling. Technical Report. Information Geometers Ltd.. Winchester, UK.

Arnold, D.N., 2018. Finite Element Exterior Calculus. volume 93 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

Baumgart, B.G., 1972. Winged edge polyhedron representation. Technical Report Stan-CS-320. Stanford University. Stanford, CA, USA.

Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V.B., Vitek, J., Zoubritzky, L., 2018. Julia: Dynamism and performance reconciled by design. Proc. ACM Program. Lang. 2. URL: https://doi.org/10.1145/3276490, doi:10.1145/3276490.

Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B., 2017. Julia: A fresh approach to numerical computing. SIAM Review 59, 65–98. URL: http://julialang.org/publications/julia-fresh-approach-BEKS.pdf, doi:10.1137/141000671.

Bieri, H., 1995. Nef polyhedra: A brief introduction, in: Hagen, H., Farin, G., Noltemeier, H. (Eds.), Geometric Modelling, Springer Vienna, Vienna. pp. 43–60.

Björner, A., Ziegler, G., 1992. Combinatorial stratification of complex arrangements. J. Amer. Math. Soc. , 105–149.

Cimrman, R., 2015. Sparse matrices in scipy, in: Varoquaux, G., Gouillart, E., Vahtras, O., deBuyl, P. (Eds.), Scipy lecture notes. release: 2022.1 ed.. Zenodo, p. Section 2.5. URL: https://scipy-lectures.org/advanced/scipy_sparse/index.html, doi:10.5281/zenodo.594102.

Delfinado, C., Edelsbrunner, H., 1995. An incremental algorithm for betti numbers of sinplicial complexes on the 3-sphere. Computer Aided Geometric Design 12, 771–784.

DelMonte, G., Onofri, E., Scorzelli, G., Paoluzzi, A., 2020. Local congruence of chain complexes. CoRR abs/2004.00046. URL: https://arxiv.org/abs/2004.00046, arXiv:2004.00046.

DiCarlo, A., Paoluzzi, A., Shapiro, V., 2014a. Linear algebraic representation for topological structures. Computer-Aided Design 46, 269–274. URL: https://doi.org/10.1016/j.cad.2013.08.044, doi:10.1016/j.cad.2013.08.044.

DiCarlo, A., Paoluzzi, A., Shapiro, V., 2014b. Linear algebraic representation for topological structures. Comput. Aided Des. 46, 269–274. URL: http://dx.doi.org/10.1016/j.cad.2013.08.044, doi:10.1016/j.cad.2013.08.044.

Dimca, A., 2017. Hyperplane Arrangements: an Introduction. Springer. URL: https://www.springer.com/gp/book/9783319562209.

Edalat, A., Lieutier, A., 1999. Foundation of a computable solid modeling, in: Proceedings of the 5th ACM Symposium on Solid modeling and Applications, Ann Arbor, Michigan. pp. 278–284.

Fabri, A., Giezeman, G.J., Kettner, L., Schirra, S., Schönherr, S., 2000. On the design of cgal a computational geometry algorithms library. Softw. Pract. Exper. 30, 1167–1202. URL: http://dx.doi.org/10.1002/1097-024X(200009)30:11<1167::AID-SPE337>3.0.CO;2-B, doi:10.1002/1097-024X(200009)30:11<1167::AID-SPE337>3.0.CO;2-B.

Fogel, E., Halperin, D., Kettner, L., Teillaud, M., Wein, R., Wolpert, N., 2007. Arrangements, in: Boissonat, J.D., Teillaud, M. (Eds.), Effective Computational Geometry for Curves and Surfaces. Springer. Mathematics and Visualization. chapter 1, pp. 1–66.

Goodman, J.E., O'Rourke, J., Tòth, C.D. (Eds.), 2017. Handbook of Discrete and Computational Geometry – Third Edition. CRC Press, Inc., Boca Raton, FL, USA.

Hachenberger, P., Kettner, L., Mehlhorn, K., 2007. Boolean operations on 3d selective nef complexes: Data structure, algorithms, optimized implementation and experiments. Comput. Geom. Theory Appl. 38, 64–99. URL: http://dx.doi.org/10.1016/j.comgeo.2006.11.009, doi:10.1016/j.comgeo.2006.11.009.

Halmos, P., 1963. Lectures on Boolean Algebras. Van Nostrand.

Halperin, D., Sharir, M., 2017. Arrangements, in: Goodman, J.E., O'Rourke, J., Tòth, C.D. (Eds.), Handbook of Discrete and Computational Geometry – Third Edition. Chapman and Hall/CRC, New York, NY, USA, p. chapter 5|33 pages.

Hatcher, A., 2002. Algebraic topology. Cambridge University Press.

Hoffmann, C.M., 1989. Geometric and Solid Modeling: An Introduction. Morgan Kaufmann Publishers, San Mateo, CA.

Hoffmann, C.M., Hopcroft, J.E., Karasick, M.S., 1988. Towards implementing robust geometric computations, in: Proceedings of the fourth ACM symposium on Computational geometry, Urbana-Champaign, Illinois, United States. pp. 106–117.

Jarvis, R.A., 1973. On the identification of the convex hull of a finite set of points in the plane. Information Processing Letters , 18–21doi:https://doi.org/10.1016/0020-0190(73)90020-3.

Kasper, J.E., Arns, D., 1993. Graphics Programming with PHIGS and PHIGS PLUS. Hewlett-Packard/Addison-Wesley, Inc., Reading, MA, USA.
Muller, D.E., Preparata, F.P., 1978. Finding the intersection of two convex polyhedra. Theoretical Computer Science 7, 217–236.
Orlik, P., Terao, H., 1994. Commutative algebras for arrangements. Nagoya Mathematical Journal 134, 65 – 73. URL: https://doi.org/, doi:nmj/1118779934.
Paoluzzi, A., 2003. Geometric Programming for Computer Aided Design. John Wiley & Sons, Chichester, UK. URL: https://doi.org/10.1002/0470013885.
Paoluzzi, A., Ramella, M., Santarelli, A., 1989. Boolean algebra over linear polyhedra. Comput. Aided Des. 21, 474–484. URL: http://dl.acm.org/citation.cfm?id=70248.70249.
Paoluzzi, A., Shapiro, V., DiCarlo, A., Furiani, F., Martella, G., Scorzelli, G., 2020. Topological computing of arrangements with (co)chains. ACM Trans. Spatial Algorithms Syst. 7. URL: https://doi.org/10.1145/3401988, doi:10.1145/3401988.
Preparata, F.P., Shamos, M.I., 1985. Computational Geometry: An Introduction. Springer-Verlag New York, Inc., New York, NY, USA.
Requicha, A., 1977. Mathematical models of rigid solids, in: Tech. Memo28, Production Automation Project. University of Rochester, pp. 1–37.
Requicha, A.A.G., Rossignac, J.R., 1992. Solid modeling and beyond. IEEE Comput. Graph. Appl. 12, 31–44. URL: https://doi.org/10.1109/38.156011, doi:10.1109/38.156011.
Rossignac, J.R., O'Connor, M.A., 1989. A dimension-Independent Model for Pointsets with Internal Structures and Incomplete Boundaries. Technical Report Research Report RC 14340. IBM Research Division. Yorktown Heights, N.Y. 10598.
Rossignac, J.R., Requicha, A.A., 1991. Constructive non-regularized geometry. Computer-Aided Design 23, 21–32. URL: https://www.sciencedirect.com/science/article/pii/001044859190078B, doi:https://doi.org/10.1016/0010-4485(91)90078-B.
Rowland, T., 2005. Characteristic function, in: From MathWorld. A Wolfram Web Resource, p. Foundations of Mathematics > Set Theory > Sets. URL: ttps://mathworld.wolfram.com/CharacteristicFunction.html.
Shapiro, V., 1991. Representations of Semi-algebraic Sets in Finite Algebras Generated by Space Decompositions. Ph.D. thesis. Cornell University. Ithaca, NY, USA.
Shapiro, V., 1997. Maintenance of geometric representations through space decompositions. International Journal of Computational Geometry & Applications 7, 21–56.
Shapiro, V., 2002. Solid modeling, in: Farin, G., Hoschek, J., Kim, M.S. (Eds.), Handbook of Computer Aided Geometric Design. Elsevier, pp. 473–518. URL: https://doi.org/10.1016/B978-0-444-51104-1.X5000-X.
Shewchuk, J.R., 2002. Delaunay refinement algorithms for triangular mesh generation. Computational Geometry 22, 21 – 74. URL: http://www.sciencedirect.com/science/article/pii/S0925772101000475, doi:https://doi.org/10.1016/S0925-7721(01)00047-5.
Stone, M.H., 1936. The theory of representations for Boolean algebras. Trans. Am. Math. Soc. 40, 37–111. doi:10.2307/1989664.
Tilove, R.B., 1980. Set membership classification: A unified approach to geometric intersection problems. IEEE Trans. on Computer C-29, 874–883.
Voelcker, H.B., Requicha, A.A., 1977. Geometric Modelling of Mechanical Parts and Processes. Technical Report Production Automation Project Memoranda, 23. University of Rochester. Rochester, N.Y.
Voelcker, H.B., Requicha, A.A.G., 1993. Research in solid modeling at the University of Rochester: 1972-1987, in: Piegl, L. (Ed.), Fundamental Developments of Computer-Aided Geometric Modeling. Academic Press Ltd.,, London, England, p. 203–254.
Woo, T., 1985. A combinatorial analysis of boundary data structure schemata. Computer Graphics & Applications, IEEE 5, 19–27.
Zhou, Q., Grinspun, E., Zorin, D., Jacobson, A., 2016. Mesh arrangements for solid geometry. ACM Trans. Graph. 35, 39:1–39:15. URL: http://doi.acm.org/10.1145/2897824.2925901, doi:10.1145/2897824.2925901.

A. Appendix

The purpose of this appendix is mainly to provide some examples of computational methods with chain complexes, and to remind (mainly from Paoluzzi et al. (2020)) specialized algorithms on such algebraic data structures, normally executed by basic operations with arrays. The interested reader may find it useful for understanding better the current prototype and the future implementation as a domain specific language.

A.1. Relevant matters

In this section we give an outline of concepts and algorithms needed to implement the computational pipeline of Algorithm 0 (Evaluation process) between an input `@CSG` form and its resolution in term of computer representation (b-reps or 2D / 3D meshes) to be used in CAD and graphics applications. Many of these were already introduced and discussed in (Paoluzzi et al., 2020), and are given here for the sake of the reader.

A.1.1. Chain-cochain duality

This §§ is quite technical, and can be skipped at first reading, but justifies our assumption to identify chain and cochain spaces and notations, being this paper interested only to topology and not to functional analysis over chains.

Any chain space, being linear, is associated with a unique dual cochain space. A linear map $L : V \to W$ between linear spaces induces a dual map $L^* : W^* \to V^*$ between their dual spaces. If (and only if) the primal chain space has finite dimension, as in our case, then its dual cochain space has the same dimension, and is therefore linearly isomorphic to the primal. Moreover, the coboundary operator δ^k is the dual of the boundary operator ∂_{k+1} :

$$(\delta^k \omega)g = \omega(\partial_{k+1} g), \qquad \text{for every } \omega \in C^k, g \in C_{k+1}. \tag{4}$$

There exist infinitely many linear isomorphisms between a finite-dimensional linear space and its dual. Each such isomorphism is produced by identifying element-wise a basis of the primal space with a basis of its dual.

By identifying the natural bases of chain and cochain spaces, we express to be only interested to combinatorial topology aspects of cell complexes. As a consequence, the boundary and coboundary matrices are related by transposition, and given the coordinate vector of a p-cell, provide both the incident $p-1$ and $p+1$ cells, respectively. Being only interested in topological properties, we have adopted the most straightforward choice, identifying elementwise the natural bases of the corresponding chain and cochain spaces. Under the selected identification, we have that the matrix $[\delta_{p-1}]$, representing δ_{p-1} in the natural bases of C_{p-1} and C_p, equals the transpose of the matrix $[\partial_p]$, representing ∂_p in the natural bases of C^p and C^{p-1}, so that the two relations, critical for our approach, follow:

$$
\begin{aligned}
&[\delta_{p-1}] = [\partial_p]^t, \\
&C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta_2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta_1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta_0}{\leftrightarrows}} C_0, \qquad \text{where} \qquad \partial C_{p-1} \circ \partial C_p = 0, \quad \text{and} \quad \delta C_{p+1} \circ \delta C_p = 0.
\end{aligned} \tag{5}
$$

Property A.1.1 (Minimal representation)**.** The LAR cellular representation in Example 2.5.1 is actually redundant, since the map FV ("faces-by-vertices") can be generated from V and EV ("edges-by-vertices") using the methods given in (Paoluzzi et al., 2020), that are summarized in Section 3.3.1. In other words, given any straight embedding of a planar graph in a plane, the cellular 2-complex—*i.e.*, the graph complement—and its arrangement in $\mathbb{E}^2$ are completely specified.

CSC sparse matrix in Julia The compressed sparse column (CSC) format represents a sparse matrix M by three (one-dimensional) full arrays. In Julia, CSC is the preferred (actually unique, by now) storage format for sparse matrices. The sparse M is stored as a `struct` containing: Number of rows, Number of columns, Column pointers, Row indices of stored values, and Stored values, typically nonzeros.

A.1.2. Published algorithms and data structures

Most of algorithms given in this section were already published in Paoluzzi et al. (2020), and are summarized here for the sake of reader convenience. The extended version of Algorithm 7 is in the manuscript by DelMonte et al. (2020). See also Figure 10 and Figure 6.

Algorithm 6 (Splitting of 2-cells in 2D)**.** Input: unevaluated LAR; output: set of 1-chain complexes.

Each set $\mathcal{I}(\sigma)$ of $(d-1)$-cells of possible intersection with a fixed $(d-1)$-cell σ, is efficiently computed by combinatorial intersection of $Box(\sigma)$ query results on d different (one for each coordinate) interval-trees (Preparata and Shamos, 1985). In particular, every $\mathcal{I}(\sigma)$ set, for $\sigma \in S_2$ input, is affinely mapped in $\mathbb{E}^3$, moving σ to the $z=0$ subspace. The arrangements $\mathcal{A}(\sigma) = C_\bullet(\sigma)$ are computed in $\mathbb{E}^2$, and then mapped back into $\mathbb{E}^3$.

Algorithm 7 (Chain Complex Congruence (CCC))**.** Input: set of 1(-2)-chain complexes; output: 1(-2)-chain complex.

We have discussed in DelMonte et al. (2020) the block diagonal marshaling matrices $[\Delta_0]$ and $[\Delta_1]$ of local coboundary matrices generated by Algorithm 6. The target of the CCC algorithm is to merge the local chains by using the equivalence relations of ϵ-congruence between 0-, 1-, and 2-cells (elementary chains). To understand all the mathematical details, the interested reader should look at the paper by DelMonte et al. (2020). In particular, we reduce the block-diagonal coboundary matrices $[\Delta_0]$ and $[\Delta_1]$, used as matrix accumulators of the *local* coboundary chains, to the *global* matrices $[\delta_0]$ and $[\delta_1]$, representative of congruence topology, *i.e.*, of congruence quotients between all 0-,1-,2-cells, via elementary algebraic operations on their columns.

1. Discover the *ϵ-nearness* of vertices by calling the function `CSG.vcongruence(V::Matrix;epsilon=1e-6)` on the $3 \times n$ input V of 3D coordinates, which returns the new vertices W and the `vclasses` map of ϵ-congruence. The $3 \times m$ matrix W holds the coordinates of class representatives, mapped to each class centroid.
2. With the Julia function `CSG.cellcongruence` we replace each subset of columns of `Delta_0` sparse matrix corresponding to *ϵ-near vertices*, with their centroid. A new matrix is produced from the array of new vectors. Finally, equal rows of this new matrix, discovered via a dictionary, are substituted by a single representative.
3. The same function `CSG.cellcongruence` is also applied to $[\Delta_1]$, by summing each subset of columns corresponding to each class of *congruent edges*, so generating a new Julia sparse matrix from the resulting set of columns. Then, we reduce every subset of equal rows, if any, to a single row representative of congruent faces.

4. Finally, a higher-level Julia function `CSG.chaincongruence` maps the input data `W`, `Delta_0`, `Delta_1`, into a compact representation `V`, `EV`, `FE` of the chain complex, where `V` : $C_0 \to \mathbb{E}^3$, $\delta_0 : C_0 \to C_1$, and $\delta_1 : C_1 \to C_2$

Algorithm 8 (Topological Gift Wrapping)**.** Input: $(d-1)$-chain complex; output: (d)-chain complex.

The TGW algorithm (Paoluzzi et al., 2020) is summarized here in d-space. The algorithm builds the unknown U_d basis of C_d space one element at a time, as a new output column, built as linear combination of U_{d-1} basis of C_{d-1}. The aim is to discover the unknown atoms of the E^d arrangement, like enclosing them in an envelope petal by petal. The name TGW comes from the similarity with *Gift Wrapping* algorithm by Jarvis (1973), but is not restricted to convex shapes, and uses ∂_{d-1} an δ_{d-2} topological operators. The input is the sparse matrix $[\partial_{d-1}]$; the output is the matrix $[\partial_d^+] : C_d \to Z_{d-1}$, from d-chains to oriented (d-1)-cycles. It is based on the topological property that the intersection of any two d-chains is exactly one or zero $(d-1)$-chain. It follows that every basis $(d-1)$-chain (column of input matrix) must be used exactly twice, summing to $2n$, where n is the number of columns of input matrix $[\partial_{d-1}]$. This fact provides a simple tool for halting the algorithm, that stops when input $(d-1)$-chains have been used $2n$ times. The interested reader may find a through 3D discussion of this algorithm in (Paoluzzi et al., 2020).

1. Initialization: $m, n = [\partial_{d-1}].shape$; $marks = zeros(n)$; $[\partial_d^+] = []$;
2. **while** sum($marks$) $< 2n$ **do**
   (a) select the $(d-1)$-cell seed of the column extraction
   (b) compute boundary c_{d-2} of seed cell
   (c) **until** boundary c_{d-2} becomes empty **do**
      i. $corolla = []$
      ii. **for each** "stem" cell $\tau \in c_{d-2}$ **do**
         a. compute the τ coboundary
         b. compute the new *petal* cell
         c. orient *petal* and insert it in $corolla$
      iii. insert $corolla$ cells in current c_{d-1}
      iv. compute again the c_{d-2} boundary of c_{d-1}
   (d) increment the $marks$ counters of used cells
   (e) append a new column to $[\partial_d^+]$

Algorithm 9 (From cycles to boundaries)**.** Input: (d)-chain complex; output: evaluated LAR (Geometric Complex).

We discuss here how to reduce the matrix $[\partial_3^+]$ of irreducible *2-cycles* generated by the TGW Algorithm 8, into the matrix $[\partial_3]$ of *2-boundaries* of irreducible 3-chains (atoms) of the $\mathbb{E}^3$ arrangement induced by the input.

1. Search, for each connected component k of the 2-complex generated by congruence Algorithm 7, the $[\partial_3^+]$ outer column;
   (a) Repeat the search and elimination for each matrix $[\partial_3^+]_k$ of a component.
      i. In each $[\partial_3^+]_k$ look for the cycle which contains the highest number of non-zeros, *i.e.*, the highest number of 2-cells.
      ii. Before elimination, check that the involved subset of vertices contains the extreme values (max and min) for each coordinate.
      iii. If true, remove this cycle from the component matrix. In the very unlikely opposite case, the second longest cycle will be candidate, and so on.
2. All component matrices $[\partial_3]_k$ without the local outer cycle ($1 \le k \le n$), are composed together columnwise is a single sparse block matrix $[\partial_3] = [[\partial_3]_1 \cdots [\partial_3]_k \cdots [\partial_3]_n]$.

By construction, each irreducible input 3-chain $u \in C_3$ corresponds to a single connected PL 3-cell, possibly non-convex and non path-contractible to a point. When this algorithm terminates, the boundary of a 3-chain may consists of one or more shells, i.e., may possibly be non-connected, made by one or more isolated 2-cycles (corresponding to internal holes and/or unconnected components).

Algorithm 10 (Topological invariants)**.** Input: geometric data structures; output: several types of invariants).

Invariants are predicates (functions that return a Boolean value) that must be satisfied by current values of variables in specific points during the program evaluation process. In particular, topological invariants are evaluated dynamically during execution to catch common numerical errors. The TGW algorithm is particularly fragile with respect to topological errors of this type, so that few invariants are evaluated in various points of the pipeline. The more common invariant is the topological characteristic, or Euler number, both in 2D and in 3D. In fact we have $\mathtt{V}-\mathtt{E}+\mathtt{F}=2$ and $\mathtt{V}-\mathtt{E}+\mathtt{F}-\mathtt{C}=0$ on the 2-sphere and the 3-sphere, topologically equivalent to the plane and the space, respectively. Some invariants are therefore computed:

1. Before 2D splitting, for each input 2-face, we have a "soup" of line segments to mutually intersect, with $\mathtt{E} \leq 2\mathtt{V}$;
2. After 2D splitting and TGW, for each simply connected 2-face component, it is necessarily $\mathtt{V}-\mathtt{E}+\mathtt{F}=2$.
3. Before chain complex congruence, when building $[\Delta_0]$ and $[\Delta_1]$ sparse container matrices, $[\Delta_1][\Delta_0]=0$
4. After CCC we must have, for the quotient topology, that $[\delta_1][\delta_0]=0$ holds, with $\ell=\mathtt{V}$ columns of $[\delta_0]$;
5. Before TGW in 3D, with input $[\partial_2]_{m,n}=[\delta_1]^\top$ we have $m=\mathtt{E}, n=\mathtt{F}$, and $\ell-m+n=p$;
6. After TGW in 3D, the identity $\mathtt{V}-\mathtt{E}+\mathtt{F}-\mathtt{C}=0$ must hold, where $\mathtt{C}=p$.

A.2. Boolean DSL

In this section we discuss a preliminary design of a *Domain Specific Language* (DSL) about the Constructive Solid Geometry (CSG) algebra introduced in this paper, as a Julia's small set of *macros*, being currently under development.

Functional form Let $\langle \mathcal{A}; \otimes_1, \ldots, \otimes_k \rangle$ be an algebra generated by $\mathcal{H} = \{h_1, ..., h_m\}$. A syntactic expression constructed as a valid sequence of operations $\otimes_i$ on n variables x_i denoting elements of $\mathcal{A}$ is called a *functional form* Φ over the algebra $\mathcal{A}$. Note that $\Phi(x_1, ..., x_n)$ is not a function (is not a set of ordered pairs) but *defines* a function of n arguments $\phi : \mathcal{A}^n \to \mathcal{A}$.

Remark (Evaluation process). We use capital Greek letters such Φ, Π, etc. to denote forms, and denote functions over algebras by lower case Greek letters. The distinction between forms and functions is important. Forms and functions over an algebra are formally related by an *evaluation* process, assigning values to the variables in the form, and computing the resulting value of the expression. This relationship is expressed by writing $\phi(x_1, ..., x_n) = |\Phi(x_1, ..., x_n)|$.

Property A.2.1 (Evaluation process)**.** If $\mathcal{A}$ is a finite algebra, there is a finite number of distinct functions over $\mathcal{A}$, but an infinite number of distinct forms, e.g., strongly redundant. If $S \in \mathcal{A}$ is an element of the algebra generated by $\mathcal{H}$, there exists a form Φ over $\mathcal{A}$ such that: $S = |\Phi(\mathcal{H})|$. We then say that S is *describable* in $\mathcal{A}$ by $\mathcal{H}$. In general, $\Phi(\mathcal{H})$ is not unique, but $|\Phi(\mathcal{H})|$ is unique by definition (Shapiro, 1991).

Domain Specific Language The *Domain Specific Language* (DSL) proposed in this section for our CSG algebra is very simple. Its well-formed formulas are made by Julia identifiers of variables, the *assembly* constructor function `Lar.Struct()`, Boolean operation symbols, and round brackets. A CSG expression is coded as a prefix Julia macro `@CSG`, to provide a simpler interface to create Julia objects with complicated structure. The list of object models associated to variables is extracted by traversing the `@CSG` expression. Each object will be located and oriented in world coordinates using the `Lar.Struct` datatype, allowing to combine hierarchical assemblies (cellular complexes in local coordinates) and affine transformations.

Boolean forms Clause is an expression formed from a finite collection of literals that matches other clauses and/or `LAR` models, by matching Boolean combinations of other clauses. It is built using one or more `@CSG` macro, each one with a typed "Struct" occurrence. The semantics of a `Struct`, natively including only "aggregation" and "affine transformation", will be enriched by n-ary union, intersection, difference, and by complement symbols. Let `A, B, C, ...` be either `LAR` models or `Struct` literals or values. We have, with `:union`, :intersect, :diff symbols, and `@CSG` a macro:

```
@CSG (+, A,B,C,...) := Struct(:union,[A,B,C,...])
 @CSG(*, A,B,C,...) := Struct(:intersect,[A,B,C,...])
@CSG (-, A,B,C,...) := Struct(:diff,[A,B,C,...])
```

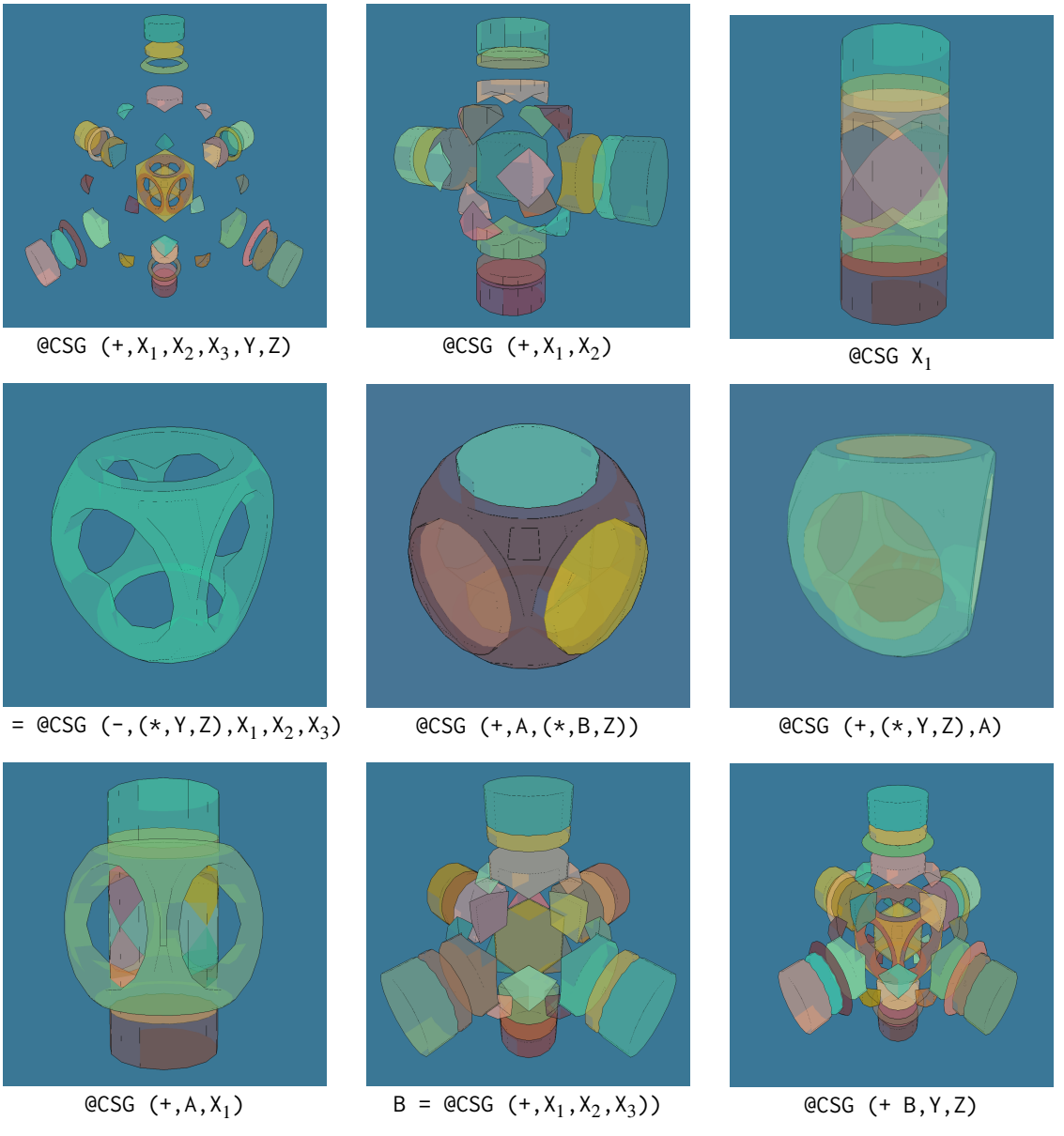


Figure 13: Five input primitives $S = \{X_1, X_2, X_3, Y, Z\}$, with three orthogonal cylinders, a cube and a sphere. The $\mathbb{E}^3$ arrangement they generate is $\mathcal{A}(S)$, shown exploded, together with evaluated DSL formulas (see Section A.2). The value returned by evaluating a Julia `@CSG` macro is of type `GeometricComplex`, *i.e.*, an evaluated LAR (see section 2.4). There are 40 atoms in this algebra, and $2^{40} \simeq 10^{12}$ terms with different *structure*. The coordinate representation of each atom is a 40-element `BitArray` with only one bit equal to 1 (`true`), and all the others equal to 0 (`false`). For computations of CSG forms with large algebras, we use sparse vectors. It is worthwhile to remark that two partial solids are stored as *bit strings* within variables `A` and `B`, and that their symbols may be used in other `@CSG` expressions. Bit strings are both the structure of Boolean forms and coordinate representations of 3-chains. By multiplication times the sparse matrix $[\partial_3]$ we get the b-rep as an oriented 2-cycle (see Example 3.1.1).

B. Appendix: Julia Examples

Julia's 2-array (matrix) names made by two characters from `V,E,F,C`, which stand for *vertices*, *edges*, *faces*, and *volumes*, in turn for 0-, 1-, 2-, and 3-cells, respectively, will be often used instead of mathematical simbols ∂_p and δ_q.

In such cases we always represent the maps: `YX : X → Y`, and `YX' = XY`.

B.1. Simple 2D example

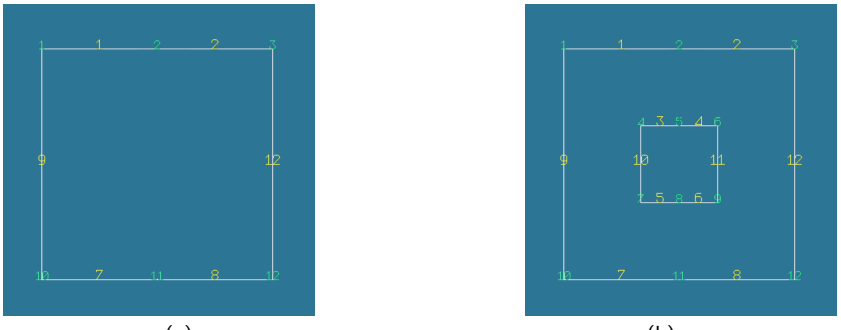
In the following example we build the sparse matrix $[\partial_2]$ of operator $\partial_2 : C_2 \to C_1$ from multiplication of characteristic matrices `K(EV)` and the transposed `K(FV)`. For details see DiCarlo et al. (2014b). Some matrices are written transposed here for sake of space. The function `K`, to contruct sparse characteristic matrices, is given in Algorithm 1.

Example B.1.1 (1-boundary)**.** Continuation from Examples 2.5.1 and 2.5.2. Variables `b1` and `b2` in this example are the 1-cycles in Figures 14a and 14b, generated by product of 2-boundary operator matrix $[\partial_2] \equiv$ `EF`, times a 2-chain (`[1 1 1]'` and `[1 1 0]'`, respectively). The Julia function `Sparsearrays.findnz` reports two arrays of indices and non-zeros of a sparse vector, and these are used as indices for the `EV` array, to extract the edges of the 1-cycle. Function `findnz` is the inverse of the `sparse` and `sparsevec` functions of Julia's `SparseArrays` package, which retrieves the inputs used to create the sparse array.

```
julia> Matrix( K(EV) * K(FV)')'
 2 1 2 1 2 1 2 1 2 2 0 0 2 2
 1 2 1 2 1 2 1 2 0 0 2 2 2 2
 0 0 2 2 2 2 0 0 0 2 2 0 1 1

julia> EF = (K(EV) * K(FV)') .÷ 2;

julia> Matrix(EF')
3x14 Array{Int64,2}:
 1 0 1 0 1 0 1 0 1 1 0 0 1 1
 0 1 0 1 0 1 0 1 0 0 1 1 1 1
 0 0 1 1 1 1 0 0 0 1 1 0 0 0

julia> b1 = (EF * [1 1 1]') .% 2
1x14 Array{Int64,2}:
 1 1 0 0 0 0 1 1 1 0 0 1 0 0

julia> b2 = (EF * [1 1 0]') .% 2
1x14 Array{Int64,2}:
 1 1 1 1 1 1 1 1 1 1 1 1 0 0

julia> SparseArrays.findnz(b1)[2]
[1, 2, 7, 8, 9, 12]

julia> SparseArrays.findnz(b2)[2]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
julia> EV[findnz(b1)[2]]
 [1, 2]
 [2, 3]
 [10, 11]
 [11, 12]
 [1, 10]
 [3, 12]

julia> EV[findnz(b2)[2]]
 [1, 2]
 [2, 3]
 [4, 5]
 [5, 6]
 [7, 8]
 [8, 9]
 [10, 11]
 [11, 12]
 [1, 10]
 [4, 7]
 [6, 9]
 [3, 12]
```

Note also that in the product matrix `A = EF` of characteristic matrices, like in row 1 of left snippet, a term $a_{ij} = e^i \times f_j$ denotes the *number* of 0-cells shared by chains e^i and f_j.



(a) (b)

Figure 14: The 1-boundaries $C_1 \ni$ `[c1] = [`∂_2`][b1] = [1,1,0,0,0,0,1,1,1,0,0,1]'` and $C_1 \ni$ `[c2] = [`∂_2`][b2] = [1,1,1,1,1,1,1,1,1,1,0,0]'` of two 2-chain vectors `[b1] = [1,1,1]'`, and `[b2] = [1,1,0]'`, with `b1,b2` $\in C_2$. Note that the `b2` $\in C_1$ cycle is disconnected, *i.e.* reducible to the sum of two irreducible 1-cycles.

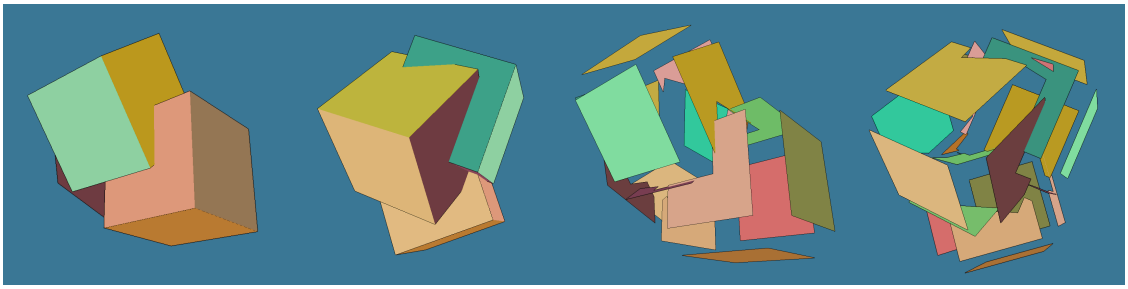
B.2. Computational examples

This section provides the reader with some simple examples of solid modeling *programming style* with Julia and its sparse and non-sparse arrays. We believe that giving a look to a few simple concrete examples is useful for understanding our computational approach and its possible developments. Sections B.2.1 and B.2.2 aim at showing the sequence of spaces and transformations that finally produce a solid model when evaluating a Boolean expression through a function application `Lar.bool3d(assembly)`. This one is the function of our current prototype which implements the whole computational pipeline, connecting all the subsystems shown in Figure 6b. This function features as formal parameter a single value `assembly`, of `Lar` type `Struct`.

We show in Section B.2.3 that the exactness property ($\partial^2 = 0$) of any chain complex can be used to check the accuracy of calculations. The *Euler characteristic* of the `union` solid model generated in Example 2.5.3 is stepwise computed in Section B.2.4, where we use the chain maps $\partial_3, \partial_2, \partial_1$ to obtain the sets (and numbers) of *faces*, *edges* and *vertices* belonging to the boundary of the `union` solid, which is a closed 2-manifold of genus zero (see Figure 15). As we said before, a specific API and a mini DSL for CSG expressions based on Julia's macros and metaprogramming are being planned (see Section A.2).

B.2.1. Variadic union

The term variadic stands for taking a variable number of arguments; especially, taking arbitrarily many arguments.



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

Figure 15: Boolean union $A \cup B \cup C$ of three cubes (from `assembly` of Example 2.5.3), with 2-cells in different colors: (a) view from the front; (b) view from the back; (c) front with exploded 2-cells; (d) back with exploded 2-cells. For clarity sake, only the boundary 2-cells are displayed. The space partition generated other 2-cells in the interior, of course.

In order to compute the `union` of three affinely transformed instances of the unit cube, we consider the `assembly` expression given in Example 2.5.3. First we get the $\mathbb{E}^3$ space partition and the structure matrix of generators produced by the function `Lar.bool3d` applied to `assembly` object; then we combine the logical arrays `A`, `B`, and `C`, building the value of `BitArray` type for the `union` variable, that stores the logical representation of the specific 3-chain. Let us remark that the bitwise "or" operator ("|") is applied in a vectorized way to arrays, by inserting a dot character:

```
julia> W, (EV, FE, CF), boolmatrix = Lar.bool3d(assembly);
julia> A,B,C = boolmatrix[:,2],boolmatrix[:,3],boolmatrix[:,4]
julia> union = .|(A, B, C);
julia> @show union;

union = Bool[false, true, true, true, true, true, true, true]
```

Finally, the boundary 2-cycle `faces` is generated by multiplication of the sparse matrix $[\partial_3]$ (*i.e.*, `CF'`) times the binary converted `union`. The mapping `Bool` $\rightarrow \{0, 1\}$ is applied via a vectorized application of the `Int8` constructor:

```
julia> faces = CF' * Int8.(union);
julia> @show faces;
faces = [1, 0, -1, 0, 0, 0, -1, 0, 1, 0, 1, 0, 0, 0, 1, -1, 0, -1, 0, 0, 1, -1, 0, -1, 1,
0, 0, 0, 1, 1, 0, -1, 0, 0, 1, 0, -1, 0, 0, 0, -1, 1, 0, 1, 0, 0, -1]
```

With $f_k \in U_2$, where U_2 is the basis of chain space C_2 generated by $\mathcal{A}$ (`assembly`), we may write in chain notation:

$$\begin{aligned} \texttt{faces} \mapsto f_{A \cup B \cup C} = & f_1 - f_3 - f_7 + f_9 + f_{11} + f_{15} - f_{16} - f_{18} + f_{21} - f_{22} - f_{24} \\ & + f_{25} + f_{29} + f_{30} - f_{32} + f_{35} - f_{37} - f_{41} + f_{42} + f_{44} - f_{47} \end{aligned} \tag{6}$$

The boundary representation, *i.e.*, the subset of boundary's oriented faces, is given by the 2-cycle in Equation (6). They are transformed into a boundary triangulation, needed for graphic display, by proper use of $[\partial_2]$ = FE' and $[\partial_1]$ = EV'.

B.2.2. Meaning of ∂ matrices

Continuation from Section B.2.1. By definition, the matrix $[\partial_2]$ = FE' = EF contains by columns the U_2 basis expressed as an ordered sequence of irreducible 1-cycle vectors. Signed values provide cyclic ordering of 1-cycles of edges, easily extendible to higher dimensions. It is often stated that cyclic ordering of polygon sides is not extendable to higher dimensions. On the contrary, this can be easily done, using d-cycles. With $e_h \in U_1$, we have:

```
FE'[:, 1] ↦ f1 = e1 -e2 +e4 -e5 +e6 -e7 +e8
FE'[:, 2] ↦ f2 = e3 +e7 -e8
...  ...  ...
FE'[:,47] ↦ f47 = e64 -e69 +e77 -e81 -e86 +e88
```

Each 1-cell e_h is mapped to an ordered pair of 0-cells $v_j \in U_0$, via the boundary matrix $[\partial_1] = [\delta_0]^t$ = EV':

```
EV'[:, 1] ↦ e1 = v2 - v1
EV'[:, 2] ↦ e2 = v6 - v3
...  ...  ...
EV'[:,49] ↦ e49 = v29 - v26
```

The geometric embedding in $\mathbb{E}^3$ of the $A \cup B \cup C$ model generated in Section B.2.1 is provided by the coordinate array W $\in \mathbb{R}^3_{49}$, with 3 rows and 49 columns. The coordinate array V embedding the initial `assembly` model, consisting of three non (yet) intersected cubes, has instead dimension 3×24.

B.2.3. Correctness checks

A computational approach based on chain complexes offers unique tools for checking the accuracy of calculations, that are correct by construction, since both $\partial^2 = 0$ and $\delta^2 = 0$ hold, when applied to any chain. In words, *every boundary is a cycle,* or equivalently: the *chain complex is exact*. Also, we know that the boundary of every solid is a possibly non-connected closed surface, hence a 2-cycle. This holds if and only if the construction of $[\partial]$ matrices is done correctly. In the following example, we start from the chain of boundary `faces` of Section B.2.1.

```
julia> pairs = [(f,sign) for (f,sign) in enumerate(CF' * Int8.(union)) if sign != 0];
julia> faces = map(prod, pairs); # standard map of functional languages
julia> @show faces; # @show is a macro to view values into variables
faces = [1,-3,-7,9,11,15,-16,-18,21,-22,-24,25,29,30,-32,35,-37,-41,42,44,-47]
```

We obtain the following 2-chain as boundary 2-cycle. The cardinality of `faces` array in `union` boundary is $\chi_2 = 21$.

$$\begin{aligned} \texttt{faces} \mapsto f_{A\cup B\cup C} = & f_1 - f_3 - f_7 + f_9 + f_{11} + f_{15} - f_{16} - f_{18} + f_{21} - f_{22} - f_{24} + f_{25} + f_{29} + f_{30} \\ & - f_{32} + f_{35} - f_{37} - f_{41} + f_{42} + f_{44} - f_{47} \end{aligned} \tag{7}$$

The edge subset on the boundary of the `union` solid is computed by: (a) transforming the `faces` array of signed indices of 2-cells into the `COORD` representation (`rows,cols,vals`) of a sparse matrix (Cimrman, 2015); (b) holding a single boundary face (2-cell) per column in `facemat`; (c) multiplying this matrix times the $[\partial_2]$ operator, thus obtaining the new sparse matrix `edges4face`, holding a face 1-cycle per column; and, finally, (d) extracting only the positive instance of boundary edges belonging to the boundary faces. The number of boundary edges $\chi_1 = 57$ is one half of the non-zero terms in the sparse vector `edges4face`. The other half has the opposite sign, so that the total sum is zero:

```
julia> nonzeros = hcat([[abs.(face),k,sign(face)] for (k,face) in enumerate(faces)]...);
julia> facemat = sparse([nonzeros[k,:] for k=1:size(nonzeros,1)]...); # load 2-cells in facemat
julia> edges4face = FE' * facemat; # generate 1-cells in sparse edges4face array
julia> rows,cols,vals = findnz(edges4face); # extract (I,J,X) from sparse array edges4face
julia> edges = [e*sign for (e,sign) in zip(rows,vals) if sign==1];
     # filter positive edge indices
julia> @show edges; # edge indices in EV array of array
edges = [1,4,6,8,10,14,18,20,9,23,26,27,2,34,30,36,38,5,29,24,43,15,42,28,47,51,53,54,21,52,57,
        48,61,62,64,50,59,66,55,58,40,63,68,77,79,80,35,78,83,72,88,7,74,87,69,81,86]
```

We remark again that, without filtering out the terms of negative `sign`, we would get an `edges` array of signed indices summing to zero, according to the constraint $\partial^2 = 0$. This attests to the exactness of calculations.

B.2.4. Euler characteristic

Euler characteristic of the connected boundary of a solid of genus g in $\mathbb{E}^3$ is defined as

$$\chi(g) = \chi_0 - \chi_1 + \chi_2 = 2 - 2g,$$

where χ_0, χ_1, χ_2 are, respectively, equal to the number of vertices, edges and faces on the boundary of the solid. In our case, continuing as test-case the Example started in Section B.2.3, the number of boundary faces is $\chi_2 = 21$, according to Eq. (7). The `edges` indices given in Section B.2.3 determine the 1-chain $e_{A\cup B\cup C}$ (or, more precisely, the non-independent 1-cycle generated by the independent 2-cycles of boundary faces). The cardinality of the `edges` array provides $\chi_1 = 57$. Note that, in order to counting the edges, we consider only the positive instances of 1-cells, since they appear in pairs (positive and negative) in a closed and coherently orientedcellular 2-complex.

$$\begin{aligned}\texttt{edges} \mapsto e_{A\cup B\cup C} = {} & e_1 + e_4 + e_6 + e_8 + e_{10} + e_{14} + e_{18} + e_{20} + e_9 + e_{23} + e_{26} + e_{27} + e_2 + e_{34} + e_{30} + e_{36} + e_{38} + e_5 + e_{29} + e_{24} \\ & + e_{43} + e_{15} + e_{42} + e_{28} + e_{47} + e_{51} + e_{53} + e_{54} + e_{21} + e_{52} + e_{57} + e_{48} + e_{61} + e_{62} + e_{64} + e_{50} + e_{59} + e_{66} + e_{55} \\ & + e_{68} + e_{77} + e_{79} + e_{58} + e_{40} + e_{63} + e_{80} + e_{35} + e_{78} + e_{83} + e_{72} + e_{88} + e_7 + e_{74} + e_{87} + e_{69} + e_{81} + e_{86}\end{aligned} \tag{8}$$

The final script computes the 0-chain $v_{A\cup B\cup C}$ of vertices on the boundary of `union` solid, with cardinality $\chi_0 = 38$.

```
julia> nonzeros = hcat([[abs(e),k,sign(e)] for (k,e) in enumerate(edges)]...);
julia> edgemat = sparse([nonzeros[k,:] for k=1:size(nonzeros,1)]...);
julia> verts = sort(collect(Set(findnz(copEV' * edgemat)[1])));
julia> @show verts;
verts = [1,2,3,4,5,6,7,8,9,10,12,15,17,18,19,20,21,24,25,26,27,30,31,32,33,34,
         35,36,37,38,39,41,44,45,46,47,48,49]
```

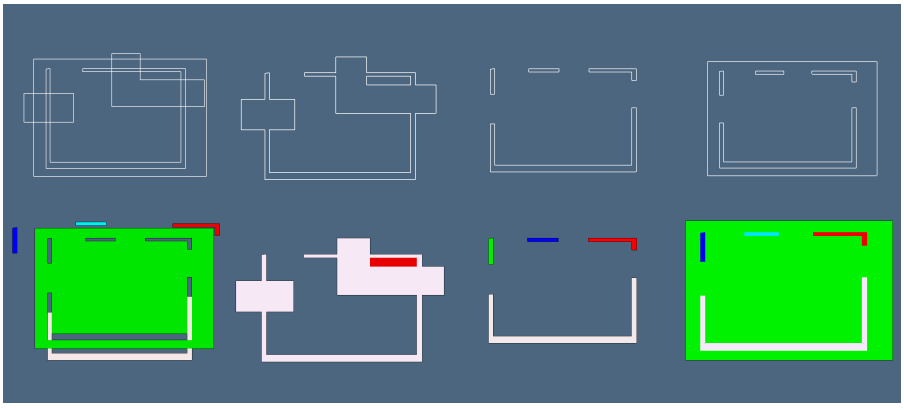
In chain notation:

$$\begin{aligned}\texttt{verts} \mapsto v_{A\cup B\cup C} = {} & v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8 + v_9 + v_{10} + v_{12} + v_{15} + v_{17} + v_{18} + v_{19} + v_{20} + v_{21} + v_{24} + v_{25} + v_{26} \\ & + v_{27} + v_{30} + v_{31} + v_{32} + v_{33} + v_{34} + v_{35} + v_{36} + v_{37} + v_{38} + v_{39} + v_{41} + v_{44} + v_{45} + v_{46} + v_{47} + v_{48} + v_{49}\end{aligned} \tag{9}$$

The generated `union` solid model has topological genus $g = 0$ (see Figure 15). Therefore, the test of correctness provided by checking the Euler characteristic χ via Eqs. (7), (8), and (9), gives the correct answer:

$$\chi(\texttt{union}) = \chi_0 - \chi_1 + \chi_2 = 38 - 57 + 21 = 2$$

Example B.2.1 (2D variadic Booleans)**.** Figure 16 shows some simple examples of 2D variadic Booleans, obtained by applying the Boolean operators "`.|`", "`.-`", "`.!`", and "`.&`" to a `Struct(assembly)` expression without affine transformations, using 2-cells (B-reps of generators) imported from SVG (Simple Vector Graphics).



(a,b) (c,d) (e,f) (h,g)

Figure 16: (a,b) start-to-end: from polygon input via SVG to (exploded) difference of outer box and interior walls; (c,d) boundary of union and union of some shapes; (e,f) difference 2-complex and its boundary; (g,h) boundary of outer box minus the previous 2-complex, and the corresponding 2-complex.