juliacon

# Julia Plasm Geometry for Solid Modeling

Alberto Paoluzzi1[1] and Giorgio Scorzelli[2]

[1]Roma Tre University, Rome, Italy
[2]Scientific Computing Institute, University of Utah, USA

## ABSTRACT

In this presentation, we introduce the Julia implementation of the geometric functional language PLaSM (Programming Language for Solid Modeling), a geometry extension of the FL research language developed by Backus and his group at IBM Almaden. Implemented as a Julia package, Plasm.jl exploits the Julia programming environment to empower AEC (Architecture, Engineering, Construction) users and BIM (Building Information Modeling) developers with a robust DSL (Domain Specific Language) for architectural design and CAD. In this presentation, we synthesize the background, architecture, and data structures of Plasm.jl, which is well-founded on combinatorial topology, algebraic homology, and Boolean binary algebra. We also provide images and code demonstrating how simple the composition and application of geometric operators are to build solid objects and environments.

## Keywords

Julia, Geometry, PLaSM, PyPlasm, Plasm.jl, BIM, CAD, DSL

## Contents

## 1. Introduction

We present a synthesis of the Julia embodiment of the PLaSM programming DSL, a geometry-oriented, comprehensive collection of specialized operators and data types designed to assist AEC users and/or BIM programmers in generating complex parametric solid objects and assemblies. This is accomplished through the specialized tools offered by this Julia package for symbolic modeling.
Cellular models, often called *meshes* in engineering, design, and graphics, utilize *discrete cells* to represent a geometric domain. In fields such as geospatial mapping, computer vision, robotics, graphics, finite element analysis, medical imaging, 3D printing, solid modeling, and geometric design, computing incidences, adjacencies, and mesh cell orderings typically depend on various incompatible data structures and algorithms. Plasm utilizes only Julia's sparse matrices and vectors and a new homological method for building any Boolean expressions with union, intersection, and difference, and for visualization of models.
This package also provides a general mechanism (Julia dictionaries) for exporting models characterized by colors, textures, materials, and more, including boolops among lower-level components.
While Julia has many simple, efficient, and beautiful packages for elementary geometric shapes and constructions and visualization of numerical simulation results, it yet lacks general-purpose tools for building and discretizing complex and large-scale geometric assemblies. Also lacks Boolean solid operations, boundary evaluation and computation of surfaces, volumes, and inertia of models and/or their parts. Additionally, Plasm can be embedded in the Jupyter platform to document design choices within digital notebooks.

### 1.1 FL origin of Plasm.jl package

In 1991, the Italian Research Council (CNR) financed a research program called "Edilizia" to promote and support national efforts to industrialize building construction. In this program, the Sapienza CADLab won a funding award sufficient to create and sustain a CS research group in geometric programming around the PLaSM project. In the same year, on the track of Backus' Turing Award, the functional programming group at IBM Almaden published the FL manual for function-level programming using combinatory logic.

### 1.2 First implementations

Developed under Italy's 'Edilizia' Finalized Project, Plasm may emerge as a pioneering metalanguage for Building Information Modeling (BIM). It evolved through multiple programming languages while retaining its core focus: multidimensional geometry and functional programming.
The PLaSM Group implemented the few combinators of FL *algebra of programs* in Common Lisp, introducing a geometric type alongside the native ones: functions, numbers, characters, and sequences [**?**, **?**]. The resulting DSL revealed an incredible descriptive power to generate multidimensional geometric models and assemblies in a few code lines [**?**]. Years later, we opted for Scheme and a C++ implementation of geometry kernel. After the move of A.P. to the new Roma Tre University and a SUR Award from IBM supporting all hardware and software (Catia licences) of a new PLM Laboratory, G.S. implemented the Pyplasm library, used to teach Computer Graphics and Geometric programming in Python and to model archeological sites like the emperor palaces on Palatine hill, S. Stefano Rotondo in Rome, and the leaning Pisa tower. Finally, while rethinking Booleans, we started porting the platform to Julia [**?**], where the geometry engine Plasm.jl and the book [**?**] provide more recent and better results.

## 2. Solid representations and Julia data types

The foundations of solid modeling technology were established between the 70s and 80s. After 40 years, at Roma Tre we started exploring new paradigms [**?**, **?**, **?**], not based on computational geometry algorithms and data structures, but on novel abstractions taking into account structures and operators of algebraic topology and linear algebra, as is happening in modern AI.

A *representation scheme* of solids is a mapping $r : M \to R$ between a space of mathematical models $M$ and a space of representations $R$ generated by a computer grammar.

### 2.1 Plasm representation scheme

Plasm.jl is a Julia package that maps any user-defined geometric expression to a binary expression of set algebras of geometric atoms, which may be resolved by linear operators on ALU engines.
The mathematical domain of the representation scheme is the category of *chain complexes*. The representation codomain space consists of *Julia data types* where evaluated expressions are stored.
In simple terms, any *model* is a chain (subset of cells in 0, 1, 2, or 3-dimensional linear chain space), and any *representation* is an instance of Julia's Hpc or Lar data types described below.
Our approach and Julia implementation originate from over fourteen years of research and experimental implementations focused on innovatively addressing the fundamental issue of solid modeling, i.e., Boolean operations.

### 2.2 Julia user-defined data types

Julia Plasm is based on three user-defined data types, enriched by several constructor and conversion methods:

***Hpc*** *Hierarchical Polyhedral Complex*, for hierarchical assembly, Boolean definitions, and interactive visualization on GPUs.

***Lar*** *Linear Algebraic Representation*, for instancing and computing with boundary-based cellular complexes and chain operators.

***Geo*** *Geometry*, for enumeration of 0- , . . . , $d$-chains in hierarchical space domains, like octrees and potrees, and for $n$D partitions in convex or non-convex cells, together with Hpc or Lar objects.

## 3. Geometric operators

In FL, the function-level approach focuses on composition, with new functions defined by combining existing tasks in various ways to create novel functions. This methodology yields a programming style centered on function-valued expressions. Few basic geometric operators in Plasm are defined without explicit arguments, departing from the applicative style. Most geometric operators utilize the functional style based on multiple applications supported by the Julia syntax, such as the ∘ composition operator.

### 3.1 Affine Transformations

In geometric modeling and computer graphics, it is helpful to distinguish between a space of vectors and a space of points (supported by vectors). In Plasm, any geometric object (Hpc or Lar) is defined in a local Euclidean space. Any affine assembly or Boolean expression is described in the coordinate frame of its first object. Internally, we use homogeneous normalized coordinates that are transparent to the users, with the homogeneous one in the first (no last) position.

[width=]figs/cuboid
Fig. 1: Multidimensional `hpc` assembly: (a) `STRUCT` object in default `VIEW`; (b) `VIEWCOMPLEX` of `LAR(hpc)` object.

In the following, we show three instances of application of `CUBOID`, and an assembly of `Hpc` type, shown on a white background, and assembled using the PHIGS+ semantics of `STRUCT`.

Code 1: Unit `interval` (1D), `square` (2D), `cube` (3D), and object `hpc` assembled with `STRUCT`.

```
interval = CUBOID([1])
square = CUBOID([1,1])
cube = CUBOID([1,1,1])
hpc = STRUCT(cube,T(2)(1),square,T(1)(1),interval)
VIEW(hpc)
VIEWCOMPLEX(LAR(cube))
```

The *affine transformation operators* require three applications to the coordinate indices, the transformation parameters, and an `Hpc` input object to generate a transformed `Hpc` output object. Two applications generate a geometry tensor as a partial function. The number of indices is equal to the number of parameters and may range from 1 to $d$. All transformation tensors `T` (Translation), `R` (Rotation), `S` (Scaling), and `H` (Shearing) are multidimensional. All are implemented in Julia as partial functions using a square invertible matrix of normalized homogeneous coordinates, with the normalized one in the first position.

*Translation transformation.*

Code 2: Simple linear `stair`, demonstrating an iterative use of tensors in `STRUCT`. Of course, the number, size, and shape of the `step` model can be parametrized as arguments of a geometric function returning `Hpc` objects.

```
function stairway(lx,ly,lz, n)::Hpc
    step   = QUOTE(lx) * QUOTE(ly) * QUOTE(lz)
    move   = T(1,2,3)(0, 0.8*ly, 0.8*lz)
    ramp   = STRUCT( CAT([[step, move] for k=1:n]) )
end
stair = stairway(.8, .22, .18, 15);
VIEWCOMPLEX(LAR(stair))
```

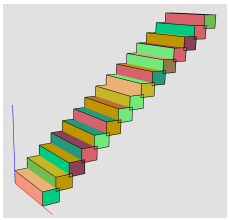
Fig. 2: Number, size, and shape of the `step` model can be parametrized as arguments of a function returning `Hpc` objects.

*Rotation transformation.* Two rotation parameters are used for elementary rotations in *any* dimension, where only one coordinate is changed with the `sin` and `cos` pattern.

Code 3: 2D and 3D rotations.

```
SQUARE(side) = CUBOID([side,side])
obj = R(1,2)(π/4)(SQUARE(1)) # as 3D rot about z
VIEW(obj)

obj1 = R(2,3)(π/3)( CUBE(1) ); # rot about x
obj2 = R(1,3)(π/6)( CUBE(1) ); # rot about y
obj3 = R(1,2)(π/4)( CUBE(1) ); # rot about z
VIEW( obj1 ); VIEW( obj2 ); VIEW( obj3 );
```

Code 4: Unit `interval` (1D), `square` (2D), `cube` (3D), and object `hpc` assembled with `STRUCT`.

```
VIEWCOMPLEX(LAR(cube))
```

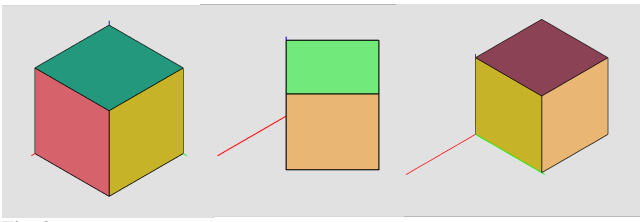

Fig. 3: Three parallel projections of a unit cube: `V = VIEWCOMPLEX ∘ LAR; V(CUBE(1)); V(R(1,2)(π/4)(CUBE(1))); V(R(1,2)(π/2)(CUBE(1)))`.

*Scaling transformation.* As an exciting coding example, we show how to construct an octahedron model simply by starting from the 3D SIMPLEX model. The final `BOOL` operator produces a decompositive representation into four 3D tetrahedra (see Figure **??**).

Code 5: Construction of `octahedron` model.

```
tetra = HPCSIMPLEX(3);
twotetra = STRUCT( tetra, S(1)(-1), tetra );
fourtetra = STRUCT( twotetra, S(2)(-1), twotetra );
octahedron = BOOL(UNION( fourtetra, S(3)(-1),
    fourtetra ))
VIEWCOMPLEX(octahedron,show=["CV"],explode=[2,2,2])
```

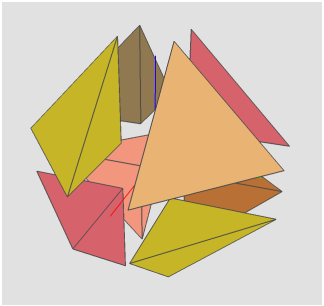

Fig. 4: The `octahedron` object's atoms explosed. Notice that `VIEWCOMPLEX` applies to `Lar` values. The four atoms of the `BOOL` operation are shown exploded (see **??** for details).

It is worth remarking that `BOOL` is an alias for `STRUCT`, where the *boolops* associated with assembly subtrees are memorized inside the associated `Hpc` subtrees `Properties`.

*Shearing transformation.*

Code 6: 3D shearing of the unit cube.

```
shearedcube = H(3)(.2,.3)(CUBE(1))
VIEWCOMPLEX(LAR(shearedcube))
```

It is worthwhile to remark that the `H` tensor, as `R`, `GR`, `S`, `T`, `MAT`, and `HOMO` are dimension independent, so they can be applied to models of whatever embedding dimension $d$ of geometric models.

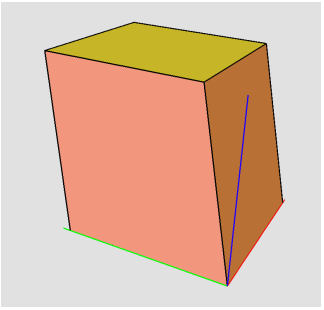

Fig. 5: Unit cube sheared on the (third) coordinate $z$. The $z$ of points does not change.

## 3.2 Cellular and simplicial complexes

A cellular model, or mesh, represents a domain using discrete cells. This section demonstrates that the Plasm representation scheme can be seen as a hybrid of cellular and boundary schemes [**?**]. Specifically, Plasm encapsulates a unique blend of hierarchical cellular schemas based on convex cell nodes defined by their vertices, alongside a flat scheme representing the entire topology of such assemblies. Importantly, chain complex representations avoid the issues and complications associated with geometric non-manifoldness, which need not be considered.

*Cartesian product of complexes.* Let us introduce the definition of the multidimensional *grids* of *cuboidal*, and the general *Cartesian product* `*` of cellular complexes, also referred to as *topological product*, which is commutative and associative like the standard product of numbers. The `*` operator, depending on the input's dimension, generates either *full-dimensional* output complexes or *lower-dimensional* complexes of dimension $d$ embedded in Euclidean $n$-space, with $d \leq n$. We say an object is *solid* when $d = n$.

Code 7: Cartesian Product of 0/1 chain instances.

```
grid1D = QUOTE(N(10)(.1));
grid2D = grid1D * grid1D;
grid3D = grid2D * grid1D;
grid11d = SKELETON(1)(grid2D);
VIEW(grid11d); VIEW(grid2D); VIEW(grid3D)
```

`N(10)(.1)` creates an array of `10` values `0.1`. The operator `QUOTE` transforms it into a 1D complex, i.e., a sequence of line segments. Figure **??** shows pictures of a 2D and 3D cuboidal complex.

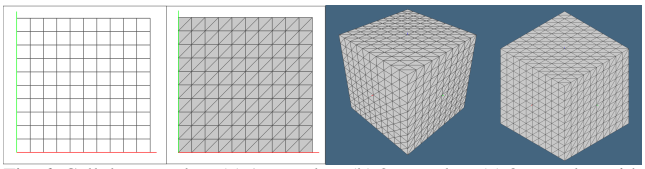

Fig. 6: Cellular complex: (a) 1-complex; (b) 2-complex; (c) 3-complex with perspective projection; (d) 3-complex with isometric parallel projection.

We recall that a convex 3D cell may be represented (a) as the convex hull of its vertices (in Julia Plasm by the `Hpc` data type), or by its Brep, boundary representation used locally or globally:

Code 8: Direct gereration of any-dimensional cuboidal complex.

```
grid = CUBOIDGRID([2,3,1]) # Lar
VIEWCOMPLEX(grid,explode=[1.2,1.2,2])

julia> grid.C
Dict{Symbol, Vector{Vector{Int64}}} with 5 entries:
  :CF => [[1, 2, 4, 9, 11, 13], [2, 3, 7, 10, 15,...
  :CV => [[1, 2, 3, 4, 5, 6, 7, 8], [1, 2, 7, 8, ...
  :FV => [[1, 2, 3, 4], [1, 2, 7, 8], [1, 2, 13, ...
  :EV => [[1, 2], [1, 3], [1, 7], [1, 9], [1, 13],...
  :FE => [[1, 2, 6, 9], [1, 3, 7, 17], [1, 5, 8, ...
```

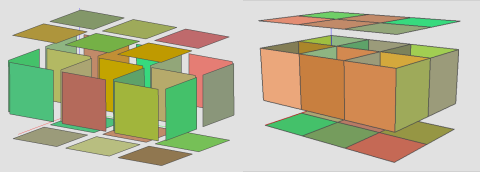

Fig. 7: Non-manifold internal brep, important with new materials models.

A small 3D `grid` is exploded in Figure **??**. The complexes generated by `CUBOIDGRID` are strongly non-manifold, an essential issue in designing a *new representation* for solid modeling. The `Plasm` DSL and its Julia structures make the distinction between manifold and non-manifold objects and models obsolete. Using chain complex models, *manifoldness* and its reverse property are not needed.

*Simplicial extrusion of complexes.* The unit cube can be decomposed into six well-assembled unit tetrahedra (3-simplices), looking at Figure **??**, where a standard 2-simplex in $\mathbb{E}^2$ is extruded orthogonally in $\mathbb{E}^3$ to generate three standard 3-simplices, thus producing a half-cube partition.

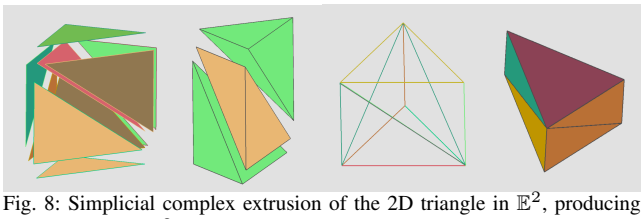

Fig. 8: Simplicial complex extrusion of the 2D triangle in $\mathbb{E}^2$, producing the half-cube in $\mathbb{E}^2$ as a complex with three tetrahedra: (a) 2-skeleton of exploded half-cube; (b) exploded 3-cells (tetrahedra); (c) 1-skeleton (set of 1-cells); (b) 2-skeleton (set of 3-cells).

A classic `PLaSM` object `model` is a `Pair` (vertices, cells) to be extruded, whereas `pattern` is an array of numbers for lateral measures of the **extruded model**, which enjoys one added dimension. The pattern's elements are assumed as either *solid* or *empty* measures, according to their (+/-) sign.

Code 9: Multiple Hpc model extrusion.

```
V = [[0.,0] [1,0] [2,0] [0,1] [1,1] [2,1] [0,2] [1,2] [2,2]];
FV = [[1,2,4],[2,3,5],[3,5,6],[4,5,7],[5,7,8],[6,8,9]];
pattern = repeat([1,.2,-2],outer=4);
model = (V,FV)
W,FW = EXTRUDESIMPLICES(model, pattern);
VIEW(MKPOL(W,FW))
VIEWCOMPLEX(LAR(MKPOL(W,FW)))
```

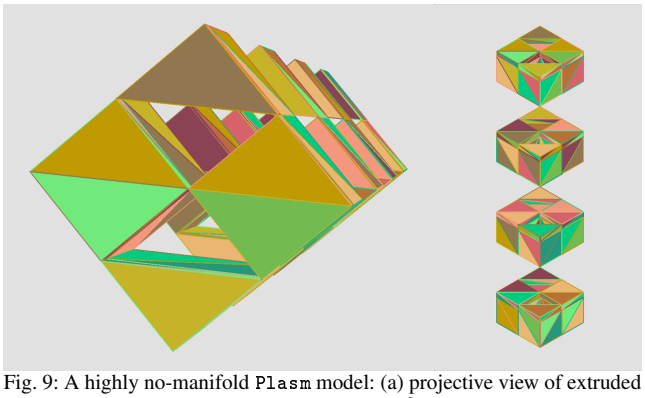

Fig. 9: A highly no-manifold `Plasm` model: (a) projective view of extruded 3-complex; (b) parallel view of 3-complex in $\mathbb{E}^3$.

*Simplicial grids.*
Geometric grid structure made by simplicial cells of the same dimension aligned on a cuboidal layout grid of 1D, 2D, 3D, etc., elements. For this construction, we use a multidimensional combinatorial formula to produce the output complex in $\Omega(n)$, linear with the output size.
This operator is mainly used for regular *domain decomposition*, to `MAP` any $d$ coordinate functions to create curved structures, such as proper curves, curved surfaces, and curved solids. The curved complex is easily generated by applying the `MAP` operator of Julia Plasm to some `Hpc` grid, so that all vertex points are coherently transformed without changing the grid topology.

Code 10: Multidimensional grid generation.

```
grid = MKPOL(SIMPLEXGRID([10,10,1]))::Hpc
VIEWCOMPLEX(LAR(SKELETON(1)(grid))::Lar)
```

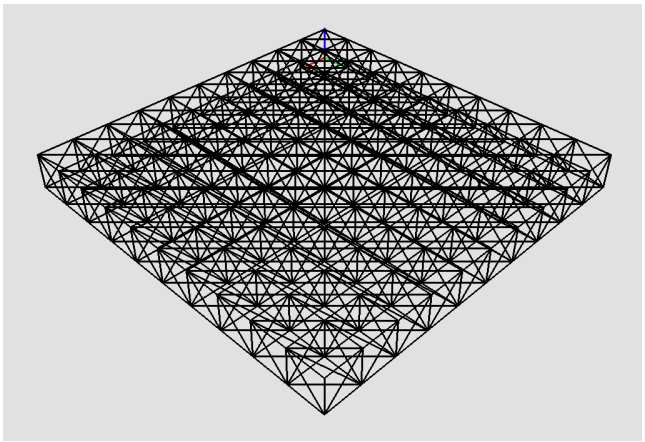

Fig. 10: The `SKELETON` operator extracting here the 1-complex from the well-assembled 3-complex generated by the `SIMPLEXGRID` operator.

The above is a slab scaffolding well-known to engineers. Because the structure is based on a triangulated framework (with triangles being inherently stable and resistant to deformation), and because these rigid cells are assembled in a regular grid, the load is efficiently distributed throughout the entire assembly.

*Volume integration of polynomials.* Plasm.jl also includes functions for calculating domain integrals of polynomials on piecewise-linear polyhedra [**?**] from a triangulation of the boundary of the domain model, an important solid modeling tool. In particular, Plasm has simple primitives for determining the mechanical properties of solids, including area, volume, centroid, products, and moments of inertia of models, presented in both scalar and tensor forms.
Hopefully, readers will appreciate the remarkable compactness of the classic FL-based language PLaSM ported to Julia.

Code 11: Boundary triangulation primitives.

```
BREP(obj::Hpc) = CONS([S1,CAT \circ S2])(
    get_oriented_triangles(obj))
BREP(obj::Lar) = CONS([S1,CAT \circ S2])(
    get_oriented_triangles(MKPOL(obj.V,
        obj.C[:CV])))
```

Here, we have shown a compact API and FL-based implementation for a minimal `Plasm` interface that converts `Hpc` and `Lar` data objects into the model data structure recognized by the native `PlaSM` language.
The simplest test examples are given below, as usual when writing unit tests for software development. Now compute some integrals on `Hpc` dataset:

Code 12: Simple test examples of the integration module `Plasm/src/integr/`: volume, surface, and centroid tests.

```
obj = CUBE(2) # => 3D cube of side 2
VOLUME(BREP(obj::Hpc)) # => 8
SURFACE(BREP(obj::Hpc)) # => 24
CENTROID(BREP(obj))'
1x3 adjoint(::Vector{Float64}) with eltype Float64:
 1.0  1.0  1.0
```

Note that the operator `BREP` only works with `Hpc` objects. A `Lar` object must first be transformed into the `Hpc` data type.

### 3.3 Parametric curves, surfaces, and solids

This section is based on concepts essential for understanding computer-generated parametric curves and surfaces. Specifically, we introduce the concepts of *curve* and *surface* as point-valued functions of one or two variables, respectively.
Plasm restricts its curved geometric objects to manifold ones, where the neighborhood of any point is topologically equivalent to a small circle of the same dimension. This suffices to define limited regular surface *patches*, the familiar curved objects in CAD and BIM. Each Plasm patch of a $d$-manifold is generated by mapping a vector function of $d$ coordinate functions over a cellular decomposition of the patch domain. To generate a 2D surface, i.e., a two-dimensional manifold embedded in $\mathbb{E}^3$, we need a vector-valued function with three coordinate functions of two parameters: $S(u, v) = [x(u, v), y(u, v), z(u, v)]$. Some examples follow.

Code 13: Symbolic example of Plasm-generated manifold patch. Of course, this coding structure may be modified when useful to produce a simpler user interface.

```
patch = MAP(mapping)(domain)
```

Code 14: Transfinite Bézier patch defined by four Bézier curves. The corresponding surface patch is shown in Figure **??**. Note that the argument Bézier functions may have any degree.

```
C0 = BEZIER(S1)([[0,0,0],[10,0,0]])
C1 = BEZIER(S1)([[0,2,0],[8,3,0],[9,2,0]])
C2 = BEZIER(S1)([[0,4,1],[7,5,-1],[8,5,1],[12,4,0]])
C3 = BEZIER(S1)([[0,6,0],[9,6,3],[10,6,-1]])
VIEW(MAP(BEZIER(S2)([C0,C1,C2,C3]))(
    INTERVALS(1.0)(10) * INTERVALS(1.0)(10)))
```

Code 15: Specialized transfinite `Bézier mapping`. Five control points, hence a quartic Bezier curve. Unit interval domain.

```
controlpoints = [[0,0],[1,0],[0.5,2],[2,1],[2,2]]
curve  = BEZIER(S1)(controlpoints)
domain = INTERVALS(1.0)(32)
VIEW(STRUCT(
    COLOR(CYAN)(MAP(curve)(domain)),
    COLOR(YELLOW)(POLYLINE(controlpoints)), FRAME2))
```

Many more curves, surfaces, and splines, including non-uniform B-splines and non-uniform rational B-splines (NURBs), have been ported to Plasm.jl from the classic PLaSM [**?**]. Two simple examples are given here.

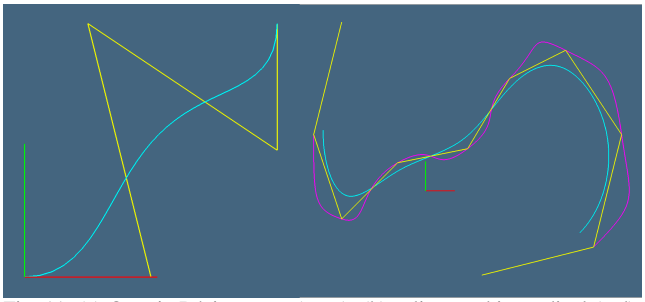

Fig. 11: (a) Quartic Bézier curve (cyan); (b) splines: cubic cardinal (red); B-spline (cyan) with the same control polygon (yellow)

Code 16: Spline cardinal and uniform B-spline.

```
domain = INTERVALS(1.0)(20)
points = [[-3.0,6.0],[-4.0,2.0],[-3.0,-1.0],[-1.0,
    1.0],[1.5,1.5],[3.0,4.0],[5.0,5.0],[7.0,2.0],
    [6.0,-2.0],[2.0,-3.0]]
poly = COLOR(YELLOW)(POLYLINE(points))
cardinal_spline = COLOR(RED)(SPLINE(CUBICCARDINAL(
    domain))(points));
uniform_Bspline = COLOR(CYAN)(SPLINE(CUBICUBSPLINE(
    domain))(points));
VIEW(STRUCT(poly,cardinal_spline,uniform_Bspline))
```

Parametric curves can be combined in many ways to create parametric surfaces, solids, or higher-dimensional manifolds. Practical classes of surfaces are discussed, in [**?**], such as the *profile product* surfaces, which include rotational surfaces; the *ruled* surfaces, encompassing generalized cylinders and cones; and the surfaces generated by *tensor product* of curves or splines. The straightforward combinatorial semantics of the Julia Plasm package may provide valuable insights to the reader with respect to tensor operations and transfinite combinations.

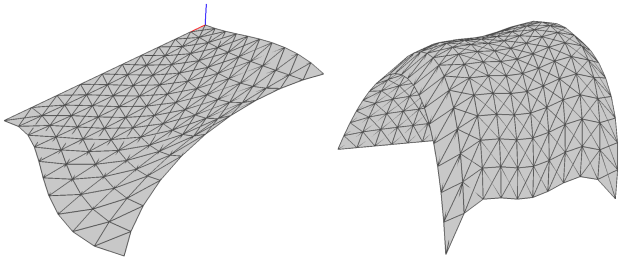

Fig. 12: Transfinite surfaces: (a) Bezier surface defined by four different degrees boundary curves; (b) Coons patch defined by four generic boundary curves (`S1, S2` are used to select the `domain` points) first/second coordinate.

### 3.4 Minkowsky operators

The Minkowski sum (or difference) of two convex polyhedra is the convex hull of the points obtained by replicating the vertices of one polyhedron around every vertex of the other one. Plasm was born multidimensional from the very beginning. Hence, the object representations and the affine transformations, particularly the shearing and the Cartesian products, are dimension-independent. Therefore, exciting and powerful methods for growing the intrinsic dimension of geometrical objects can be defined by bringing them into higher dimensions, then transforming affinely, and finally projecting back into the original embedding space. For example, the object shown in Figure **??** is appropriately extruded in $\mathbb{E}^6$ space, and then projected back in $\mathbb{E}^3$.

Code 17: House 3D model by Minkowsky `OFFSET`.

```
verts = [[0.,0.,0.],[3,0,0],[3,2,0],[0,2,0],[0,0,
    1.5],[3,0,1.5],[3,2,1.5],[0,2,1.5],[0,1,2.2],
    [3,1,2.2]]
cells = [[1,2],[2,3],[3,4],[4,1],[5,6],[6,7],[7,
    8],[8,5],[1,5],[2,6],[3,7],[4,8],[5,9],[8,9],
    [6,10], [7,10],[9,10]]

House = MKPOL(verts, cells)::Hpc
VIEWCOMPLEX( LAR(House) )
VIEWCOMPLEX( LAR(OFFSET([0.2, 0.2, 0.5])(House)) )
```

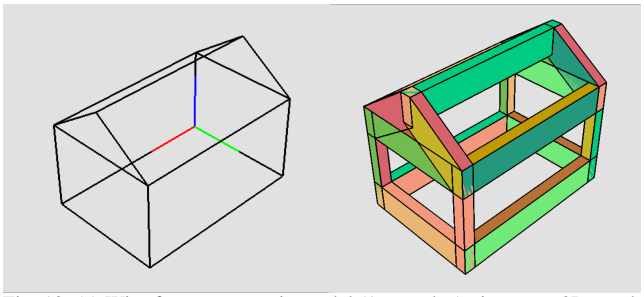

Fig. 13: (a) Wire-frame geometric model (1-complex) given as a 3D graph in `(verts,cells)`; (b) the generated 3-complex through `OFFSET` operator.

### 3.5 Enumerative schemes and geometric combinators

A solid model may be described by enumerating the set of solid cells in some partitioning of the embedding space. It is possible to distinguish between schemes using sparse Boolean matrices as shape parcel enumerators and schemes based upon hierarchical space decompositions, say *quadtrees* and *octrees*. Such representations are approximations of the object's space occupancy, even for linear polyhedra.

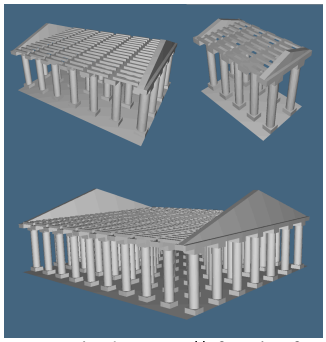

Fig. 14: The fully parameterized `temple()` function features some parameters that are interconnected through algebraic equations. Let's note the number of side and front columns and their relationship to column height and gable width. Additionally, observe the mixture of 3D and 2D objects, where `PROJECT(2)()` and `BOX(3)(temple)` generate the ground rectangle of the right size.

This scheme is also used in Plasm, e.g., by the `CUBOIDGRID` and `SIMPLEXGRID` generator functions, which accommodate basic topological cells like small cubes or simplices within textured (even curved) shapes used, for example, in 3D printing and new multi-material shapes.

The hierarchical enumerative approach can be simulated in some sense by using the operators `LEFT`, `RIGHT`, `UP`, and `DOWN` in 2D and 3D; `TOP`, and `BOTTOM` in 3D, always between `Hpc` objects, able to be easily converted into graph structures in computer memory. In Plasm, the enumerative scheme may be extended to grids that alternate similar solid shapes and empty intervals of embedding space. This approach would dramatically accelerate the easy and fast development of preliminary design models, mainly in architecture and civil engineering.

## 4. Visualization & interaction platforms

The `src/viewer.jl` module of `Plasm.jl` provides an interactive geometry viewer for visualizing and interacting with the shapes generated by the Plasm codes. It supports rendering on a laptop terminal screen and within the HTML interface of a web browser. The web-based viewer was developed to display Plasm models online and facilitate writing rich-text examples and exercises in a web notebook, eliminating the need to install any software.

### 4.1 GLFW-based multi-platform 3D viewer

As the many examples clearly showed, a geometric computing platform needs powerful and easy methods to visualize and interact with an object's model under development. Currently, the primitives `VIEW` and `VIEWCOMPLEX` allow the user to present interactive environments with many graphical features, including colors, perspective, and parallel projective presentation.

Two submodules extend this functionality by integrating external libraries. `Viewer.glfw.jl` leverages GLFW, an open-source, multi-platform library for developing OpenGL, OpenGL ES, and Vulkan applications. Meanwhile, `viewer.meshcat.jl` utilizes `MeshCat.jl`, a remotely-controllable 3D platform built on `three.js` library for web applications. `MeshCat.jl` allows local visualization and interaction with Plasm models directly in a web

browser, such as within a `Jupyter` notebook controlled by a Julia server, local or remote.

### 4.2 Notebook environment execution

The platform design includes multiple execution and visualization environments that support different user needs and computational requirements. With its OpenGL GLFW viewer, the full-featured desktop application enables researchers to access native user interfaces and local computing resources for detailed model interaction and intensive tasks.
But `Plasm.jl` also operates without restrictions in Jupyter notebook environments [**?**]. In this case, the geometric backend communicates to the graphical front-end via a message queue. It renders the results in a simple browser by leveraging its JavaScript engine, which has advanced rendering capabilities and can render even very large geometric models [**?**].
Julia users of Plasm geometric applications can execute notebooks through either a local machine server or a remote server operating on high-performance computing infrastructure. Furthermore, users benefit from dual-mode accessibility because they can select the most suitable interface and computational backend according to their needs between local analysis and collaborative resource-intensive computations on remote servers.

### 4.3 Two-dimensional SVG input

Plasm lacks, by design, a graphical user interface, but it can import 2D designs from SVG files. Scalable Vector Graphics (SVG) is the W3C standard for 2D vector graphics on the Web based on XML. The SVG standard includes support for several primitives, interactive exchange from mouse clicks and touch events, as well as animated visualization in response to user actions. The primitives can be filled with color or a gradient. The wireframe drawing (stroke) can also be rendered with a specific width, possibly using solid Plasm primitives. Some features will be ported to Plasm in a future version.
To extend Plasm's capabilities with Scalable Vector Graphics (SVG) support, we developed a Julia parsing module that processes SVG files by (*) parsing the XML structure using the XML.jl library [XML], (*) traversing the XML tree, and applying systematically and accumulating transformations to ensure all graphical elements are embedded in a global coordinate space.
All the core SVG shapes (e.g., rectangles, circles, ellipses, lines, polylines, polygons) and complex path data are converted into standardized polyline or polygon representations. For curved elements (including path segments like Bezier curves and elliptical arcs) user can set a configurable number of points to sample and generate the polyline approximation.

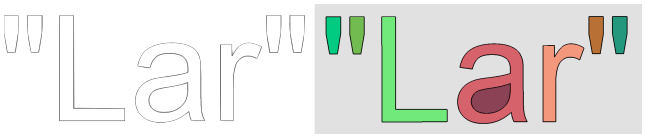


Fig. 15: Images generated by `Plasm`: (a) one-dimensional `Hpc` model generated by the expression `SVG("filename.svg")`. Note that the model of the `"Lar"` string includes several curved splines; (b) atoms in different colors generated by `ARRANGE2D` primitive. The `outer` atom and cell (irreducible to a point, see Section **??**) is the ivory background.

The parser also tries to preserve the most important style attributes, including fill color, stroke color, and stroke width, to store in `Properties` field of data objects. Color definitions, whether specified as hexadecimal values, RGB/RGBA functions, or named colors, are consistently converted into four-component RGBA (Red, Green, Blue, Alpha) arrays, thus facilitating their direct use in subsequent rendering or analysis pipelines within Plasm.
For the polygon PL-filling from low-degree spline curves, we provided in-house using the classic PLaSM algorithm [**?**] with Boolean `XOR` of planar adjacent stripes generated by the polygon profile of a spline curve. Of course, the `src/pdf/` module contains the initial development stage and requires much more work, extensive testing, and time.
Importing and parsing `.svg` files generated by interactive design environments and stored in local memory or the cloud is a crucial link between the Plasm language running the `REPL` interpreter on a local or remote terminal and the CAD extrusion of file content. At present, we have only performed a few experiments interfacing the geometric language with this web tool, but it is considered a high-priority project for the `Plasm` platform.

### 4.4 PLY format reading and writing

To enhance its versatility and facilitate the integration of existing 3D assets, Plasm.jl allows importing external geometric models using the Polygon File Format (PLY) [**?**]. The PLY format is a common standard for representing 3D data, mainly originating from 3D scanners or other modeling software. PLY support allows users to bring these external geometries into the Julia computational environment.
First, we parse .ply files to extract essential geometric information, including vertex coordinates and face definitions. After reading this raw data, Plasm.jl translates it into internal data structures, typically the form of cellular complex provided by the Julia Plasm `Lar` data type. This conversion process ensures that the imported model, regardless of its origin, is represented in a manner consistent and compatible with `Plasm.jl`'s native geometric objects, allowing further elaboration.
By transforming external data into its internal format, the Julia Plasm package allows users to import models obtained from a wide variety of sources, significantly extending the platform's interoperability and practical application scope in research and development workflows involving pre-existing 3D datasets.

## 5. Arrangements and Boolean Algebra

This section synthesizes the Plasm innovative approach to the Boolean algebra of solid models, rooted in the algebraic topology of piecewise-linear geometry. Specifically, Plasm implements computational topology algorithms to uncover the two- or three-dimensional space partitions (called arrangement by combinatorial topologists) formed by collections of 1D and 2D geometric objects, respectively [**?**].
Julia's sparse arrays, with their standard algebraic operations, were the principal data types for our computational program. Their extensive use [**?**] for topology representation allowed us to handle general cellular complexes that are homeomorphic to $d$-polyhedra, representing triangulable spaces that can be non-convex and multiply connected.
Sparse arrays, with their standard algebraic operations, were essential data structures for our computational program. They permitted us to handle general cellular complexes homeomorphic to $d$-polyhedra, representing triangulable spaces that can be non-convex and multiply connected. The computation of *space arrangements*

was the preliminary step toward generating Boolean solid algebras and the binary resolution of any Boolean expression within.

### 5.1 Space homology pipeline

To clarify, it may be beneficial to revise the mathematical notation for "chain complex", which we restate here with our data structure notation for the user's convenience. For sake of reader comprehensibility with respect to the mathematics, we translate the names of sparse matrices of topological operators, and use `EV: V` $\rightarrow$ `E`, `FE: E` $\rightarrow$ `F`, and `CF: F` $\rightarrow$ `C`, with relational meanings from our topological operators (`V, E, F, C` stand for vertices, edges, faces, and cells, respectively).

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

$$\mathtt{C} \underset{\mathtt{FC}}{\overset{\mathtt{CF}}{\leftrightarrows}} \mathtt{F} \underset{\mathtt{EF}}{\overset{\mathtt{FE}}{\leftrightarrows}} \mathtt{E} \underset{\mathtt{VE}}{\overset{\mathtt{EV}}{\leftrightarrows}} \mathtt{V}.$$

Many 2D unit tests have been run during the development and the Plasm implementation of 2D and 3D algorithmic pipelines. In Figure **??**, we show two executions with fifteen squares with random rotation centers (the left-bottom vertex) and angles.

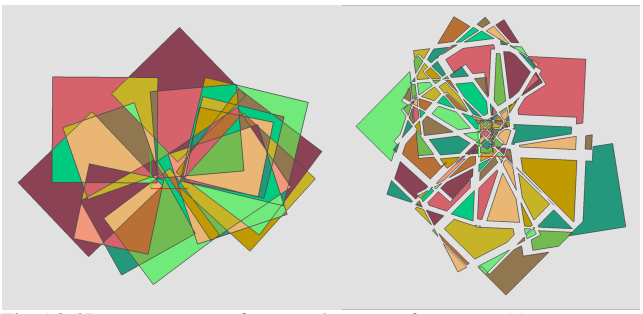

Fig. 16: 2D arrangements of two random sets of squares : (a) arrangement atoms in different colors; (b) atoms exploded about the origin.

Similar test examples were developed for implementing the `ARRANGE3D` function, which was later joined with the companion `ARRANGE2D` function into a single `ARRANGE` operator used within the computational Julia Plasm environment `BOOL` to evaluate Boolean solid formulas in 2D or 3D.

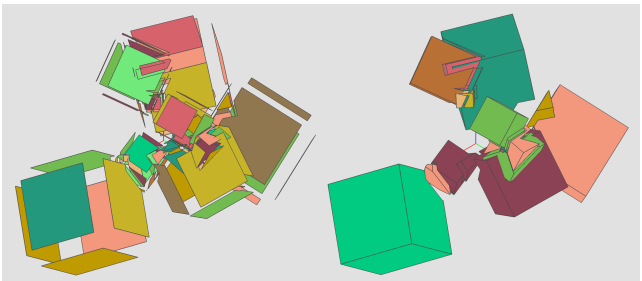

Fig. 17: 3D arrangements of ten random cubes: (a) 2-cells of the arrangement atoms; (b) atoms in different colors exploded about the origin.

Code 18: aaaa.

```
hpc = STRUCT([RandomBubble() for I in 1:50])
arrangement = ARRANGE2D(LAR(hpc))
VIEWCOMPLEX(arrangement)
VIEWCOMPLEX(arrangement, explode=[1.5,1.5,1.5])
```

In the above script, we have shown the typical pattern needed for this purpose. First, the set of generators, in this case, 50 random polygons, is correctly assembled within a `Hpc STRUCT` collection; then it is transformed into a `Lar` cellular complex and passed as an argument to the functions `ARRANGE2D` or `ARRANGE3D` in case of three-dimensional dataset, producing all the atoms of the `arrangement` partition.

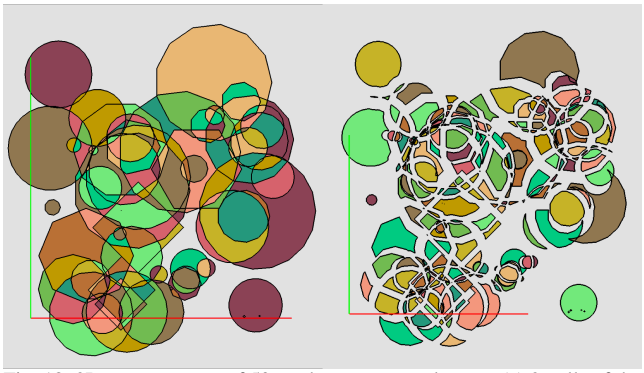

Fig. 18: 2D arrangements of 50 random convex polygons: (a) 2-cells of the arrangement atoms; (b) atoms in different colors exploded about the origin.

### 5.2 Object generators and atoms

*Chain complexes* are an algebraic tool for computing or defining homology. They are used here to algebraically describe a partition of the embedding space induced by a collection of geometric objects, particularly all the cycles (holes) in the dimensions of interest—specifically, from dimension 0 to dimension 3.

For example, let us consider a building with many *hollow spaces* and thin separation surfaces. Architects and engineers have always designed the structure of internal spaces by creating their separation elements and how they relate. In Plasm, we consider the topological abstraction of incidence and adjacency of elementary separation parts by a sequence of spaces of chains (subsets of cells) and linear operators, so fetching the algebraic tool of a chain complex.

In a Boolean algebra, atoms and generators have distinct roles and definitions. In summary, atoms are distinct elements related to the structure and cardinality of Boolean algebra, while generators are elements used to construct or describe this algebra. When considering Boolean strings of finite Boolean algebras, atoms contain only one non-zero element.

`Plasm` may utilize via the `Lar` type very general 2-cells, possibly non-convex and featuring holes. Refer to Figure **??**b. It's important to note that the `partition` object includes all the 3D atoms of the $\mathbb{E}^3$ arrangement generated by the `Plasm` expression `ARRANGE3D(LAR(hpc))` including the holes present in the embedding space $\mathbb{E}^3$.

Conversely, `INNERS(partition)` comprises only the atoms (inners) that do not contribute to the finite boundary of the exterior unbounded atoms `OUTERS(partition)`. In our scenario, this collection is connected and cannot be exploded by `Plasm`; therefore, only one hole exists in the exterior 3D space.

### 5.3 Boolean solid algebras

*The fundamental property.* At the core of the Boolean operations between solid rigid models implemented in our `Plasm` language, there is the following fundamental property:

PROPERTY 1 BOOLEAN ATOMS ARE UNIT 3-CHAINS. *There is a* natural transformation [1] *between d-chains defined on a spatial arrangement and the solid algebra generated by that arrangement.*

Solid objects are often defined as hierarchical assemblies of solid primitives or more complex shapes, each represented 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. Arcs are linked to affine transformations that shift the entire subgraph rooted in the ending node onto the coordinate system of the initial node of the arc.

*Solid Boolean algebra implemented in Plasm.* Here, we clarify the evaluation method of complex `Plasm` expressions of solid algebra, written with `Hpc` objects, variadic `UNION`, `INTERSECTION`, and `COMPLEMENT` operators, as well as `DIFFERENCE` and `XOR`, along with parentheses to modify the order of operations.
Solid objects are typically defined as hierarchical assemblies of solid primitives or more complex shapes, each represented 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. Arcs are linked to affine transformations that shift the entire subgraph rooted in the ending node onto the coordinate system of the initial node of the arc.
The `Plasm` Boolean method executes a Depth First Traversal (DFS) traversal when evaluating the `Hpc` tree of the input Boolean expression, where geometry is stored on the leaves in local coordinates, and non-leave nodes may contain either affine transformations or Boolean operators (see Example **??**).
We have five tasks in sequence:

(1) DFS traversal to get all solid terms (algebra generators) in root coordinates and put local Boolean functions stored as placeholders in property fields of `Hpc` nodes (see Example **??**);
(2) construction of the global equivalent Boolean function made by combining elementary Boolean functions, selector functions, and parentheses (see Example **??**);
(3) execution of the embedding space arrangement, with boundary generation of each oriented atom cycle (`TGW` algorithm)—(see Figure **??**);
(4) construction of the truth table of the Boolean algebra using a point-set membership algorithm (see Example **??**);
(5) application of the Boolean function equivalent to the solid input expression on all minterms of atoms (see Example **??**).

Some solid algebra expressions are computed and visualized in Figure **??** and **??**. A description of the `Plasm` syntax for Boolean solid expression is shown in the examples.

Code 19: Plasm expression of its Boolean solid algebra. Shown in Figure **??**

[1] In category theory, a branch of abstract mathematics, a *natural transformation* provides a way of transforming one functor into another while respecting the internal structure (i.e., the composition of morphisms) of the categories involved. Hence, a natural transformation can be considered a "morphism of functors."

```
cube = T(1,2,3)(-1,-1,-1)(CUBE(2))
cyl  = T(3)(-2)(CYLINDER([0.5,4])(8))
hpc = DIFFERENCE(
   cube,
   UNION(cyl, R(2,3)(π/2), cyl, R(1,3)(π/2), cyl)
)
result = BOOL(hpc)
VIEWCOMPLEX(result, explode=[3,3,3])
VIEWCOMPLEX(result, explode=[3,3,3], show=["CV"])
```

Code 20: Plasm expression of its Boolean solid algebra. Shown in Figure **??**

```
hpc = UNION(
   cube,
   UNION(cyl, R(2,3)(π/2), cyl, R(1,3)(π/2), cyl)
)
result = BOOL(hpc)
```

## 6. References


[1] J. Backus, J.H. Williams, and E.L. Wimmers. An introduction to the programming language FL. In D.A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, Reading, MA, 1990.

[2] John Backus, John H. Williams, Edward L. Wimmers, Peter Lucas, and Alexander Aiken. FL LANGUAGE MANUAL. PARTS 1 AND 2. Technical Report RJ 7100 (67163), IBM Almaden Research Center, 1989.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[4] C. Cattani and A. Paoluzzi. Boundary integration over linear polyhedra. *Computer-Aided Design*, 22(2):130–135, 1990. doi:https://doi.org/10.1016/0010-4485(90)90007-Y.

[5] Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. Linear algebraic representation for topological structures. *Computer-Aided Design*, 46:269–274, 2014. doi:10.1016/j.cad.2013.08.044.

[6] A. Paoluzzi. *Geometric Programming for Computer Aided Design*. John Wiley Sons, Chichester, UK, 2003.

[7] Alberto Paoluzzi and Giorgio Scorzelli. *BIM geometry with Julia Plasm – Functional language for CAD programming*. Wiley Nature, 2025. In print.

[8] Alberto Paoluzzi, Vadim Shapiro, Antonio DiCarlo, Francesco Furiani, Giulio Martella, and Giorgio Scorzelli. Topological computing of arrangements with (co)chains. *ACM Trans. Spatial Algorithms Syst.*, 7(1), October 2020. doi:10.1145/3401988.

[9] Alberto Paoluzzi, Vadim Shapiro, Antonio DiCarlo, Giorgio Scorzelli, and Elia Onofri. Finite algebras for solid modeling using julias sparse arrays. *Computer-Aided Design*, 155:103436, 2023. doi:https://doi.org/10.1016/j.cad.2022.103436.

[10] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. Using the jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–2, 2017. doi:10.1109/JCDL.2017.7991618.

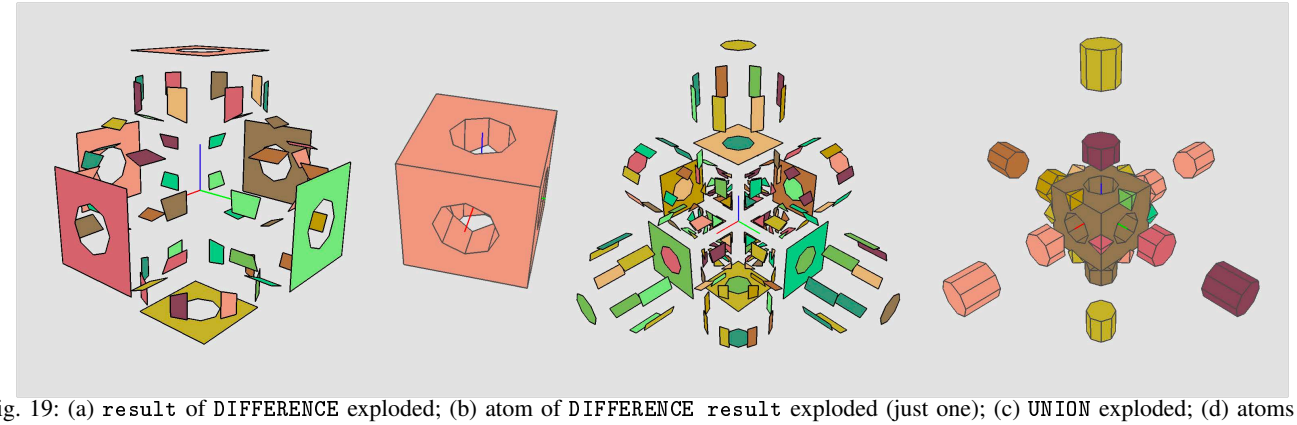

Fig. 19: (a) `result` of `DIFFERENCE` exploded; (b) atom of `DIFFERENCE result` exploded (just one); (c) `UNION` exploded; (d) atoms of `UNION result` exploded.

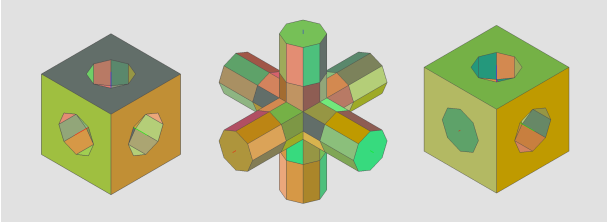

Fig. 20: (a) `c\(x1∪x2∪x3)`; (b) `x1∪x2∪x3`; (c) `c\(x2∪x3)`, where `c`, `x1`, `x2`, `x3` are the algebra generators.

[11] A.A.G. Requicha. Representations for rigid solids: Theory, methods and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[12] The MathWorks, Inc. The ply format. `https://de.mathworks.com/help/vision/ug/the-ply-format.html`, 2025. Accessed: 2025-05-15.

[13] W. Usher and V. Pascucci. Interactive visualization of terascale data in the browser: Fact or fiction? In *IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)*, 2020.

## APPENDIX

### A. Parametric building frame

Parametric models and topological operators demonstrate how to create flexible and reusable designs using parametric functions and topological transformations. Geometric mapping of complexes and grids lets us explore techniques to map and manipulate geometric data structures effectively. Look at Figure **??**.

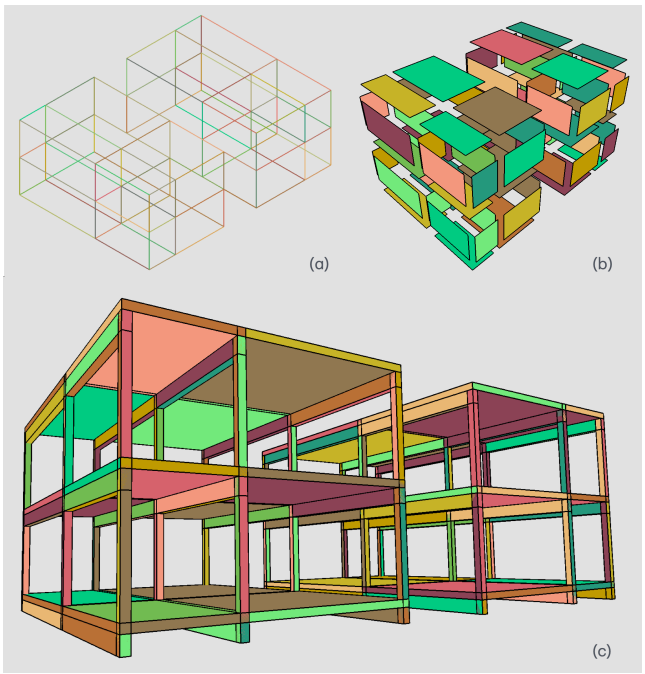


Fig. 21: (a) SK(1)(idea); (b) exploded object 2-complex; (c) structural framexyz model from the project named idea in the following Julia Plasm fragment. All the generating Plasm code is shown in Figure **??**.

```
using Plasm
X = GRID([2.4,4.5,-3,4.5,2.4]);
Y = GRID([7,5]);
Z = GRID([3,3]);
idea = X * Y * Z;
VIEW(SK(1)(idea))   # Fig. 1a
VIEWCOMPLEX(LAR(idea), explode = [1.2,1.2,2.0] )    # Fig. 1b

building110 = X * Y * SK(0)(Z);
building1_101 = SK(1)(SK(0)(X)*SK(1)(Y)*SK(1)(Z));
building1_011 = SK(1)(SK(1)(X)*SK(0)(Y)*SK(1)(Z));

floors = OFFSET([.2,.2,.2])(building110);
framex = OFFSET([.2,.2,.2])(building1_011);
framey = OFFSET([.2,.2,-.4])(building1_101);
framexyz = STRUCT(framex, framey, floors);
VIEWCOMPLEX(LAR(framexyz))                  # Fig. 1c
```

Fig. 22: The Julia Plasm code fragment producing all the images of Fig. **??**
.