I❤MESH: A DSL for Mesh Processing

YONG LI, South China University of Technology, China and George Mason University, USA
SHOAIB KAMIL, Adobe Research, USA
KEENAN CRANE, Carnegie Mellon University, USA
ALEC JACOBSON, University of Toronto and Adobe Research, Canada
YOTAM GINGOLD, George Mason University, USA

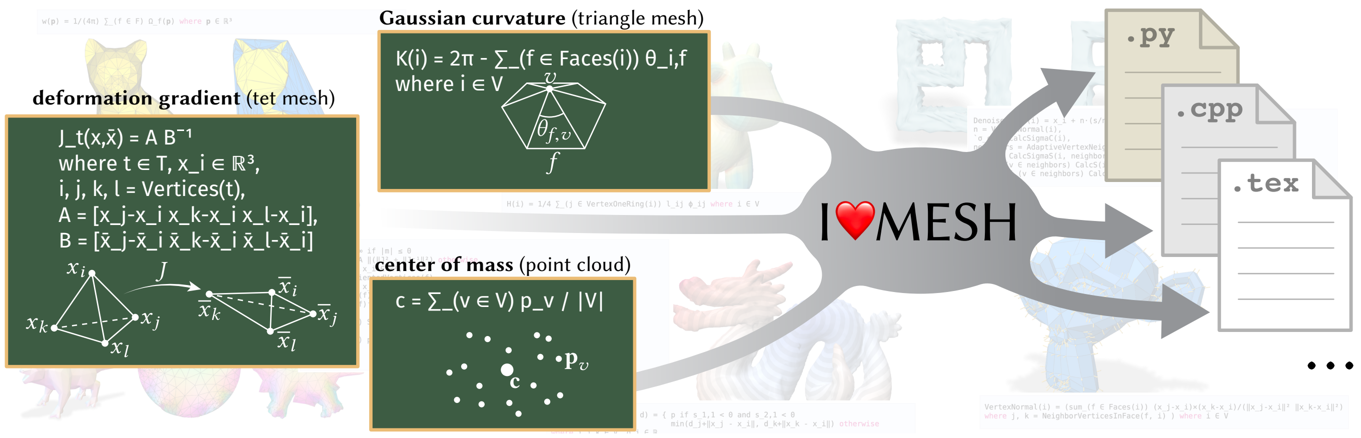

Gaussian curvature (triangle mesh)
K(i) = 2π - ∑_(f ∈ Faces(i)) θ_i,f
where i ∈ V
v
$\theta_{f,v}$
f
deformation gradient (tet mesh)
J_t(x,x̄) = A B⁻¹
where t ∈ T, x_i ∈ ℝ³,
i, j, k, l = Vertices(t),
A = [x_j-x_i x_k-x_i x_l-x_i],
B = [x̄_j-x̄_i x̄_k-x̄_i x̄_l-x̄_i]
x_i x_j x_k x_l J $\bar{x}_i$ $\bar{x}_j$ $\bar{x}_k$ $\bar{x}_l$
center of mass (point cloud)
c = ∑_(v ∈ V) p_v / |V|
$\mathbf{p}_v$
$\mathbf{c}$
I❤MESH
.py
.cpp
.tex
...

Fig. 1. I❤MESH enables rapid exploration of mesh processing algorithms by providing a nexus between high-level math-like notation and compilable code.

Mesh processing algorithms are often communicated via concise mathematical notation (e.g., summation over mesh neighborhoods). However, conversion of notation into working code remains a time consuming and error-prone process which requires arcane knowledge of low-level data structures and libraries—impeding rapid exploration of high-level algorithms. We address this problem by introducing a domain-specific language (DSL) for mesh processing called I❤MESH, which resembles notation commonly used in visual and geometric computing, and automates the process of converting notation into code. The centerpiece of our language is a flexible notation for specifying and manipulating neighborhoods of a cell complex, internally represented via standard operations on sparse boundary matrices. This layered design enables natural expression of algorithms while minimizing demands on a code generation back-end. In particular, by integrating I❤MESH with the linear algebra features of the I❤LA DSL, and adding support for automatic differentiation, we can rapidly implement a rich variety of algorithms on point clouds, surface meshes, and volume meshes.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**; • **Mathematics of computing** → **Mathematical software**; • **Software and its engineering** → **Domain specific languages**.

Authors' addresses: Yong Li, South China University of Technology, China and George Mason University, USA, pressure36@gmail.com; Shoaib Kamil, Adobe Research, USA, kamil@adobe.com; Keenan Crane, Carnegie Mellon University, USA, kmcrane@cs.cmu.edu; Alec Jacobson, University of Toronto and Adobe Research, Canada, jacobson@cs.toronto.edu; Yotam Gingold, George Mason University, USA, ygingold@gmu.edu.

Additional Key Words and Phrases: meshes, geometry processing, domain-specific language

ACM Reference Format:
Yong Li, Shoaib Kamil, Keenan Crane, Alec Jacobson, and Yotam Gingold. 2024. I❤MESH: A DSL for Mesh Processing. *ACM Trans. Graph.* 0, 0, Article 0 (2024), 17 pages. https://doi.org/XX.YYYY/ZZZZZZZ.WWWWWW

1 INTRODUCTION

Geometry processing algorithms entail evaluating or optimizing quantities aggregated over mesh elements. In scientific writing, such algorithms are typically expressed as a mixture of mathematical notation and verbal/rhetorical descriptions. For instance, the area a_v associated with a vertex v of a triangle mesh might be defined by writing *"Let* $a_v = \frac{1}{3}\sum_{f\in\mathcal{N}(v)} Area(f)$*, where* $Area(f)$ *is the area of triangle* f *and* $\mathcal{N}(v)$ *is the collection of triangles containing vertex* v*"*.

To turn this definition into code, one must then carefully translate each term and definition into expressions from a given programming language and/or library. Even within a single language (say, C++) there are numerous libraries, each with its own idiosyncratic interface [Bischoff et al. 2002; Fabri and Pion 2009; Jacobson et al. 2018; Sharp et al. 2019a; Sieger and Botsch 2019]. Hence, algorithm exploration is slow, and achieving consistent, bug-free execution across different platforms is a daunting task.

Similar challenges in the context of numerical linear algebra were recently addressed by the I❤LA DSL [Li et al. 2021, 2022]. Unlike linear algebra, however, formulas found in geometry processing are not purely arithmetic; they must also reference topological data structures (point clouds, polygonal surface meshes, polyhedral volume meshes, etc.), which have no universally agreed upon notation.

Moreover, although authors occasionally provide rigorous definitions for these data structures and their local neighborhoods (e.g., the *star/link/closure* formalism for a simplicial complex), more often they are left undefined. Authors hence rely on readers to interpret notation based on a common cultural context, perhaps in conjunction with annotated figures (Section 3). (See Section 5 for an in-depth study of generality, composability, and syntax in the context of mesh processing DSLs.)

In response to these challenges, I♥MESH provides a translation layer between higher-level notation and lower-level implementation—formalizing and automating the ad-hoc translation process currently performed by expert readers (Figure 1). In particular, it models meshes as *cell complexes* [Hatcher 2002, Chapter 0], which are internally represented via sparse *boundary matrices* [Elcott and Schröder 2005; Friedman 2012]. Users of I♥MESH express mesh operations in terms of math-like syntax, and can define custom mesh stencils/neighborhoods in terms of *cellular sets*, i.e., heterogeneous collections of mesh elements, encoded by sparse vectors. These expressions are then compiled to routines in a general-purpose *host language* (e.g., C++), which can in turn be invoked within, e.g., a larger graphics or geometry processing codebase. We provide a standard library of neighborhoods for a variety of topological data structures, demonstrating that this approach is versatile enough to cover many common use cases. We also implement a variety of geometry processing algorithms (Section 7) which demonstrate how to use the system in an end-to-end fashion.

I♥MESH extends I♥LA into a language we call *H♥rtLang*, adding support for mesh-specific types, operations, and quantities, as well as derivatives of these quantities. To support these features, we add a wide variety of basic language features: better set handling (including set builder notation and type-checking based on inclusion in a specific element set), more functional programming constructs, recursive functions, type-based function overloading, and automatic differentiation. We also introduce syntax and semantics common in mathematical notation, such as function parameters expressed as subscripts, and locally scoped variable definitions. Support for a given host language requires minimal effort: an implementation of cellular sets via sparse linear algebra, plus a few elementary operations (e.g., translation between cellular sets and indicator vectors). Source code and documentation for I♥MESH can be found at https://github.com/iheartla/iheartla.

Note that our goal in this paper is primarily to explore language design for mesh processing. For this reason, performance is a non-goal of I♥MESH, and there are many opportunities for acceleration (including, perhaps, compiling to high-performance backends like *SimIt* [Kjolstad et al. 2016] or *MeshTaichi* [Yu et al. 2022]). To maintain a concise, domain-specific language, procedural logic (e.g., for expressing optimization algorithms) is also out of scope and is instead handled by the general-purpose host language. Finally, we do not consider mutations of the topological data structure (e.g., edge flips), which present future interesting questions—and could also in principle be implemented via the boundary matrix abstraction [Shapero 2023].

2 RELATED WORK

Mesh Data Structures. A cornucopia of mesh data structures has been proposed over the years with varying performance characteristics and representational flexibility. One class of representations are generalizations of linked lists, including half-edge [Bischoff et al. 2002; Campagna et al. 1998; Kettner 1998; Lienhardt 1994; Mäntylä 1989; Muller and Preparata 1978], quad-edge [Guibas and Stolfi 1985] and winged-edge [Baumgart 1972]. These data structures are based on pointer-following to access and enumerate neighbors. While theoretically efficient, operations involving these data structures are highly sequential in nature. Instead, we base I♥MESH on declarative paradigms, boundary matrices and cellular sets, where many mesh operations can be expressed in conventional mathematical notation.

Many mesh processing libraries have been proposed over the years, such as CGAL [Fabri and Pion 2009], OpenMesh [Bischoff et al. 2002], libigl [Jacobson et al. 2018], Geometry Central [Sharp et al. 2019a], and the Polygon Mesh Processing library [Sieger and Botsch 2019]. Those libraries are extremely powerful due to the various built-in algorithms they provide. These libraries are all written in C++, though some have been wrapped for use in other languages. However, a design goal for I♥MESH is to generate code with minimal dependencies, allowing authors to write portable mesh processing expressions that compile to any supported backend. We make use of some of these libraries for file I/O in our example applications.

Boundary matrices have been proposed as an elegant representation for simplicial and cellular complexes [DiCarlo et al. 2014; Edelsbrunner and Harer 2008; Elcott and Schröder 2005; Kaczynski et al. 2004]. These approaches represent mesh connectivity as a collection of sparse matrices; incidence relationships are revealed by matrix multiplication. DiCarlo et al. [2014] proposed the Linear Algebraic Representation (LAR) representation for meshes and studied the theoretical storage and runtime efficiency using compressed sparse row (CSR) matrices for unoriented meshes. Three recent approaches focused on adapting sparse boundary matrix mesh representations for efficient use on the GPU. Zayer et al. [2017] proposed a particular sparse matrix encoding that makes orientation information efficient to extract. Mueller-Roemer et al. [2017] proposed a custom sparse matrix encoding that is more compact and efficient for running on the GPU. RXMesh [Mahmoud et al. 2021] generalizes the "think like a vertex" [McCune et al. 2015] GPU programming model to all three types of triangle mesh elements (vertices, edges, and faces). They propose a custom sparse matrix encoding that is compact and captures orientation information. They decompose the mesh into patches with overlap for better locality. I♥MESH, too, leverages sparse matrices as the fundamental building block for mesh connectivity, since they require only sparse matrices for their implementation, which are already supported by H♥rtLang across all backends. The only extra support needed are functions to create an indicator vector from a set of indices and vice versa. See the supplemental materials for a collection of examples.

Mesh Processing DSLs. Several domain-specific languages (DSLs) have been proposed for meshes. Ebb [Bernstein et al. 2016] proposed a three-layer architecture for physical simulation on CPUs and GPUs by separating simulation code, data structures for geometric domains, and runtimes. Simit [Kjolstad et al. 2016] supports both graph and matrix views of the simulation. The data model uses hypergraphs and tensors as two abstract data structures. Operations such as tensor assemblies and index expressions can be built upon them. More recently, MeshTaichi [Yu et al. 2022] was proposed as a high-level programming model for efficient mesh-based operations. It uses reference-style neighborhood queries to hide the complex indexing system. The compiler partitions the input meshes and prepares the relations via inspecting the code during compile time. In contrast to these DSLs, I❤️MESH focuses on building an expressive, extensible language similar to conventional notation. High performance is not a goal. In the future, the I❤️MESH compiler could target these other languages. See Section 5 for an in-depth characterization of the design space of mesh processing DSLs in graphics.

There is a large body of work in scientific computing concerned with DSLs for expressing and solving partial differential equations (PDEs) over regular grids (e.g., Devito [Lange et al. 2016], Firedrake [Rathgeber et al. 2016], and ExaStencils/ExaSlang [Lengauer et al. 2020]) and irregular meshes (e.g., Liszt [DeVito et al. 2011], FEniCS [Baratta et al. 2023], PyOP2 [Rathgeber et al. 2012], and HighPerMeshes [Alhaddad et al. 2022]). Devito [Lange et al. 2016] utilizes the SymPy [Meurer et al. 2017] package to generate optimized parallel C code from high-level symbolic operators defined for finite differences. Firedrake [Rathgeber et al. 2016] and FEniCS [Baratta et al. 2023] automate the numerical solution of PDEs via the finite element method. It separates the local discretization of mathematical operators from their parallel execution over the mesh. Both rely on a high-level DSL for expressing finite element discretizations [Alnæs et al. 2014]. ExaStencils/ExaSlang [Lengauer et al. 2020] support advanced multigrid solver generation with four layers of abstraction ranging from the formulation in continuous mathematics to a full, automatically generated implementation. Liszt [DeVito et al. 2011] is designed to build portable mesh-based PDE solvers. It applies program analysis to determine the appropriate stencil and track field phases. PyOP2 [Rathgeber et al. 2012] portably applies numerical kernels in parallel over an unstructured mesh. PyOP2 treats all mesh entities as sets, with kernels written in a subset of C99. HighPerMeshes [Alhaddad et al. 2022] supports a number of heterogeneous platforms with a C++-embedded DSL that relies on template metaprogramming. These systems all focus on solving PDEs while taking efficient advantage of compute resources (via advanced solvers, memory layout efficiency, and parallelism). In contrast, we are concerned with flexibly expressing the mesh expressions and neighborhoods common in the computer graphics literature. Implementing or interfacing with PDE solvers is out of scope for our work.

Math Notation Languages. A variety of programming languages have been proposed with varying emphasis on syntax resembling conventional mathematical notation. Fortran, MATLAB, Mathematica, Fortress [Allen et al. 2005], Lean [de Moura et al. 2015], Julia [Bezanson et al. 2017], and I❤️LA [Li et al. 2021]. We chose to build I❤️MESH on I❤️LA for several reasons. First, its syntax is closer to conventional notation than the others (including juxtaposition multiplication, def-use analysis for automatic code ordering, and Unicode operators). Second, it is run-time agnostic. I❤️LA can generate code with minimal dependencies for multiple backends (C++/Eigen, Python/NumPy/SciPy, MATLAB). I❤️LA and its backends already support the sparse linear algebra we require. Third, it outputs LaTeX for aesthetic typesetting of user's expressions. Lastly, its source code is small and open source.

N_j denotes the set of adjacent vertices of p_j	$a_j = \dfrac{\sum_{i \in N_j} \alpha_{ij} a_{ij}}{\sum_{i \in N_j} \alpha_{ij}}$	[Brunel et al. 2021]
$N_R(k)$ is the R-ring neighborhood of the graph node k	$\mathbf{f}'_k = \mathbf{b}_k + \sum\limits_{l \in \mathcal{N}_R(k)} a_{k,l} \mathbf{K}_l \mathbf{f}_l$	[Habermann et al. 2021]
A_i denotes the neighborhood set of vertex $v_i \in V$	$\boldsymbol{v}_i - \dfrac{1}{d_i} \sum\limits_{j \in A_i} \boldsymbol{v}_j = \delta_i$	[Song et al. 2020]
C_i are the tetrahedral cells adjacent to the vertex i	$\boldsymbol{F}_i \approx \dfrac{\sum_{k \in C_i} \frac{w_k}{r_k} \boldsymbol{F}(\boldsymbol{r}_k)}{\sum_{j \in C_i} \frac{w_j}{r_j}} = \sum\limits_{k \in C_i} w'_k \boldsymbol{F}(\boldsymbol{r}_k)$	[James 2020]
$N(p)$ defines the patches neighboring p	$E_O = \sum\limits_{p \in P} \sum\limits_{q \in \mathcal{N}(p)} \sum\limits_{i=0}^{N-1} (\alpha_{p,i} - \alpha_{q,i})$	[Chandran et al. 2022]
prev(i) and *next*(i) are the two segments adjacent to s_i	$\ell'_i = w_{\text{prev}(i)} + \ell_i + w_{\text{next}(i)}$	[Campen et al. 2019]
$N(v)$ is the set of faces incident to the vertex v	$\kappa_v := 2\pi - \sum\limits_{f \in N(v)} \theta_v^f$	[Stein et al. 2020]
t_1 and t_2 are triangles adjacent to edge e	$E_p = \left(\sum\limits_{e \in E} w_e \Vert f_{t_1} - f_{t_2} \Vert_2^p \right)^{\frac{1}{p}}$	[Zhang et al. 2020]

Fig. 2. Mesh-based expressions from SIGGRAPH papers 2019–2022. The formulas involve neighborhoods defined verbally, shown at left.

3 BACKGROUND

We conducted a study to understand how the computer graphics community communicates *mesh-based expressions*, i.e., quantities expressed in terms of the discrete elements of a mesh (such as vertices or edges), and values associated with those elements (such as positions or lengths). We collected such expressions from papers published over four years of SIGGRAPH proceedings (2019–2022). We were particularly interested in how authors model and notate mesh connectivity, subsets of mesh elements, and refer to values associated with mesh elements.

Connectivity. Authors commonly model mesh connectivity in terms of discrete sets of elements. For instance, one might use $\mathcal{M} = (V, E, F)$ to denote a triangular surface mesh with vertex set V (which is just an abstract set of points, with no associated data), edge set E, and triangle set F, respectively. A similar representation can be used for volumetric meshes (e.g., by adding a set T of tetrahedra, or a set C of more general polyhedral cells), or for point clouds (e.g., by omitting F and perhaps using E to encode the nearest neighbors of each point). Though not often considered in graphics, this basic set-based construction is compatible with more rigorous treatment of mesh topology in terms of *simplicial complexes* [Edelsbrunner 1999], or more generally *cell- or CW-complexes* [Hatcher 2002, Chapter 0]. Alternatively, one might name only the highest-dimensional elements, from which the lower-dimensional elements can be inferred. E.g., for a triangle mesh with no isolated edges or vertices, the set F contains all information about connectivity. In practice, it is hence quite common to model a triangle mesh as a pair $\mathcal{M} = (V, F)$ where F is the face set, but V is now a concrete list of vertex coordinates, rather than abstract points.

Values. More generally, values associated with mesh elements are typically modeled as either sets indexable by mesh elements, or equivalently, as functions from the set of vertices V to a space of values (like $\mathbb{R}^n$). For example, triangle areas might be expressed as $a_f \in \mathbb{R}_{\geq 0}$ for each $f \in F$, which can also be viewed as a map $a : F \to \mathbb{R}$.

Matrix Representation. Some authors assume from the outset that both connectivity and values are already encoded as matrices. For instance, the faces of a triangle mesh might be interpreted as a matrix $F \in \mathbb{Z}^{|F|\times 3}$, where $|F|$ is the number of triangles, and matrix entries provide indices into the list of vertices. Likewise, vertex coordinates might be expressed as a matrix $V \in \mathbb{R}^{|V|\times 3}$.

Neighborhooods. The notation above is sufficient for specifying values associated with individual elements. However, it does not provide a mechanism for specifying incidence relationships between mesh elements, which are needed to define mesh neighborhoods (sometimes called *stencils*), and in turn, build up larger expressions. Some examples from recent SIGGRAPH papers are shown in Figure 2. A significant challenge is that common terminology often has ambiguous or overloaded meaning—for instance, the *1-ring* of a vertex $v \in V$ might refer only to neighboring vertices (perhaps with or without v), or it might mean all mesh elements incident on v (including both edges and triangles). This lack of specificity can be problematic for correctly translating mathematical expressions into code. Although more precise symbolic notation for incidence relationships exist in some settings (such as *star, link, and closure* [Edelsbrunner and Harer 2008]), they are quite rare in computer graphics (appearing in no paper from our survey). Apart from a few exceptions (e.g., Sharp et al. [2019b, Section 2.1]), authors often do not explicitly define notation for iteration over neighborhoods—assuming that it will be understood from context and prior knowledge.

However, in order to perform automatic transformations, I❤MESH must be given formal definitions of neighborhoods. One possibility would be to parse existing natural language definitions (as already given in many papers). Although this approach could be quite valuable, it is also quite difficult to implement reliably—for instance, even state of the art large language models still struggle with formal reasoning [Xu et al. 2023]. Instead, neighborhood definitions are expressed in H❤rtLang itself, in terms of operations on boundary matrices (Section 3.1) and vectors that encode sets of mesh elements. This mechanism enables users to model iteration over mesh neighborhoods as they see fit; we also provide a library of mesh neighborhoods common across geometry processing. We describe I❤MESH's layered design in Section 4.

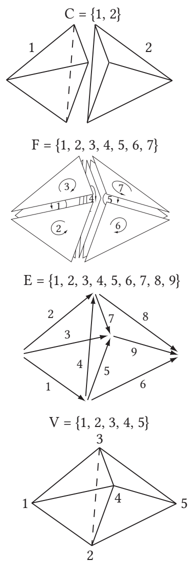
3.1 Boundary Matrices

We use *boundary matrices* as the basic internal representation for mesh connectivity in I❤MESH. For any mesh data structure, the kth boundary matrix ∂^k encodes which k-dimensional mesh elements are adjacent to which $(k-1)$-dimensional elements, as well as the relative orientation of these elements. For instance, in a triangle mesh with connectivity $\mathcal{M} = (V, E, F)$ the matrix ∂^2 is an $|E| \times |F|$ matrix with nonzero entries $\partial^2_{e,f} = \pm 1$ for each edge $e \in E$ contained in each triangle $f \in F$. The sign of the entry is positive if the orientations agree (e.g., the direction of the edge is consistent with the winding direction of the face) and negative otherwise. Orientation information is essential for many geometry processing tasks—for instance, to compute consistently-oriented surface normals on a triangle mesh. Note however that the *relative* orientation information needed to define boundary matrices is still available even for globally nonorientable geometry (such as a Möbius band).

Element indices. The boundary matrix construction assumes that each mesh element of the same dimension is assigned a unique index from a contiguous range starting at 1. In this sense, the element sets (V, E, etc.) can also be viewed as subsets of the integers. Indices can be assigned arbitrarily, since changing the indexing simply permutes the matrix rows/columns—and hence does not impact the meaning of the entries. Figure 3 shows an example of a tetrahedral mesh and its corresponding boundary matrices.

Extracting neighborhoods. The boundary of a set of elements can be obtained by multiplying a boundary operator with an *indicator vector* for those elements, i.e., a vector equal to one at each index corresponding to an element in the set, and zero otherwise. Formally, this indicator vector encodes the set of elements as a *chain*, i.e., a formal linear combination of mesh elements [Hatcher 2002, Chapter 2]. For example, in Figure 3, the matrix-vector product $\partial^3(1,1)^\top$ yields a vector with non-zeros corresponding to the exterior faces of the two tetrahedra (all but face 4), with values oriented consistently with these tetrahedra. Similarly, the boundary of a single triangle is its three oriented edges (circulating around the triangle), and the boundary of an edge is its two endpoints (negative at the “tail”, and positive at the “head”). The *co-boundary* of a k-dimensional element, comprised of incident $k+1$-dimensional elements, can likewise be obtained by multiplying an indicator by the transpose of a boundary matrix. For example, the non-zeros of $(\partial^1)^\top(0,0,0,0,1)^\top$ correspond to the edges 6, 8, 9 incident on vertex 5. Finally, multiplying the absolute value of all $k-1$-dimensional boundary matrices by an indicator vector for a single element (i.e., a Kronecker delta) gathers all the vertices of a k-dimensional element. For instance, $|\partial^1||\partial^2||\partial^3|(1,0)^\top$ gathers all vertices (of all edges of all faces) of the first tetrahedron in Figure 3.

Formally, boundary matrices unambiguously determine a CW-complex, up to reparameterization of the gluing maps. Hence, they are sufficiently expressive to encode most of the mesh data structures commonly used in geometry processing, including point clouds,



$$\partial^3 = \begin{pmatrix} -1 & \\ 1 & \\ -1 & \\ 1 & -1 \\ & 1 \\ & -1 \\ & 1 \end{pmatrix}$$

$$\partial^2 = \begin{pmatrix} 1 & 1 & & & & & \\ -1 & & 1 & & & & \\ & -1 & -1 & & & & \\ 1 & & & 1 & 1 & & \\ & 1 & & -1 & & 1 & \\ & & & & -1 & -1 & \\ & & 1 & 1 & & & 1 \\ & & & & 1 & & -1 \\ & & & & & 1 & 1 \end{pmatrix}$$

$$\partial^1 = \begin{pmatrix} -1 & -1 & -1 & & & & & & \\ 1 & & & -1 & -1 & -1 & & & \\ & 1 & & 1 & & & -1 & -1 & \\ & & 1 & & 1 & & 1 & & -1 \\ & & & & & 1 & & 1 & 1 \end{pmatrix}$$

Fig. 3. Element sets and boundary matrices for a simple tetrahedral mesh (adapted from [Elcott and Schröder 2005]).

simplicial meshes and many kinds of polygonal/polyhedral meshes. From a data structure point of view, they are strictly more general than a list of top-dimensional cells (such as a vertex-face adjacency matrix), and can represent most complexes representable by linked list-like data structures (halfedge, corner table, etc.), apart from a few corner cases (e.g., a single edge glued together at endpoints to form a loop). Hence, boundary matrices provide an attractive internal representation of connectivity in I❤MESH—while still supporting a language where most final mesh expressions can be written without explicit reference to boundary matrices.

The choice to use boundary matrices is also compatible with the larger design of H❤rtLang, which is already built on top of sparse matrix representations. The only additional support needed for I❤MESH are functions to create an indicator vector from a set of element indices and vice versa—see Section 4.2.

4 PROGRAMMING MODEL AND LANGUAGE DESIGN

We introduce our programming model via an example. Consider the following expression for discrete Gaussian curvature on a triangle mesh. For each vertex, the Gaussian curvature can be defined as the angle defect:[1]

$$K(v) = 2\pi - \sum_{f \in faces(v)} \theta_{v,f} \tag{1}$$

where $\theta_{v,f}$ is the angle between incident edges of the face f at the vertex v.

The following is a complete implementation in I❤MESH. It generates a *module* containing the function $K(v)$ that takes a mesh vertex as a parameter.

```
1  ElementSets from MeshConnectivity
2  Faces, NeighborVerticesInFace from Neighborhoods(M)
3  arccos from trigonometry
4
5  M: FaceMesh
6  V, E, F = ElementSets( M )
7  x_i ∈ ℝ³
8
9  K(v) = 2π - ∑_(f ∈ Faces(v)) θ_v,f where v ∈ V
10
11 θ_i,f = arccos((x_j-x_i)·(x_k-x_i)/(‖x_j-x_i‖ ‖x_k-x_i‖))
12         where i ∈ V, f ∈ F, j,k = NeighborVerticesInFace(f,i)
```

[1]We follow the definition from Geometry Central [Sharp et al. 2019a], which does not divide by the local area.

The first three lines import built-in packages. The `MeshConnectivity` module provides access to the element sets and boundary matrices of a mesh (Section 4.1). The `Neighborhoods` module is initialized with the input mesh M declared on line 5. The module's functions are tied to M and will perform checks to ensure that only elements of M are passed as parameters (Section 4.1.1). Lines 6 and 7 access the element sets of M and declare a second input parameter x of per-vertex positions. Line 9 is a direct translation of Equation 1. It makes use of the neighborhood function `Faces`, which maps a vertex in M to the set of incident faces. H❤rtLang supports function overloading based on parameter types (Section 4.3.2), so `Faces` behaves appropriately when called with an edge, face, or cell. Line 11 defines the incident angle, which wasn't specified in Equation 1. Note that H❤rtLang allows definitions to occur out of order. The θ function takes its parameters as subscripts. Its parameters are defined as members of V and F. The compiler also verifies that θ's parameters belong to these specific element sets of M. The θ function makes use of another neighborhood function, `NeighborVerticesInFace`, which returns the next two vertices in f in counterclockwise order. When compiled, I❤MESH generates LaTex for typesetting.

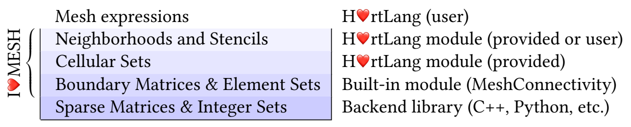
Usage from C++. The following C++ code uses the module generated by I❤MESH to compute the curvature for each vertex.

```
1  // Load a mesh from disk.
2  Eigen::MatrixXi F;
3  std::vector<Eigen::Vector3d> x;
4  LoadOBJ("mesh.obj", x, F);
5  // Create boundary matrices.
6  iheartmesh::TriangleMesh M(F);
7  // Copy boundary matrices into the I❤MESH data structure.
8  iheartmesh::FaceMesh fm(M.bm1, M.bm2);
9  // Initialize the Gaussian curvature module.
10 GaussianCurvature module(fm, x);
11 std::vector<double> gaussian_curvature(x.size());
12 for (int i = 0; i < x.size(); ++i) {
13     // Compute per-vertex curvature.
14     gaussian_curvature[i] = module.K(i);
15 }
```

In this snippet, line 6 creates boundary matrices for a triangle mesh with a provided `iheartmesh::TriangleMesh` helper class, and line 8 initializes the corresponding I❤MESH mesh data structure to pass as a parameter to the module. `iheartmesh::FaceMesh` is a lightweight C++ class provided with I❤MESH containing the mesh data (Section 4.2). Line 10 initializes the C++ class generated by the I❤MESH compiler with the mesh and positions. Line 14 calls the Gaussian curvature function K as an ordinary C++ method for each vertex.

4.1 Architectural Layers

I❤MESH has a layered architecture (Fig. 4). Users write mesh expressions in H❤rtLang that are compiled by I❤MESH. A typical user's neighborhood and stencil functions are included in a library provided with I❤MESH. However, we cannot anticipate every kind



	Mesh expressions	H♥rtLang (user)
I♥MESH	Neighborhoods and Stencils	H♥rtLang module (provided or user)
	Cellular Sets	H♥rtLang module (provided)
	Boundary Matrices & Element Sets	Built-in module (MeshConnectivity)
	Sparse Matrices & Integer Sets	Backend library (C++, Python, etc.)

Fig. 4. I♥MESH's layered architecture supporting user-written mesh expressions. H♥rtLang is the extension of I♥LA with many new language features. I♥MESH contains the MeshConnectivity built-in module in addition to a library of neighborhood and stencil functions. Much of I♥MESH is written in H♥rtLang itself. This allows users to write and share their own neighborhood and stencil functions as needed by accessing the the cellular set or boundary matrix layers of I♥MESH. This also facilitates adding support for additional backend languages.

of mesh stencil or neighborhood a user may need. Because much of I♥MESH is written H♥rtLang itself, users can write and share functions that iterate over new kinds of own neighborhoods as needed. This extensible architecture also minimizes the effort needed to generate code for additional backends.

4.1.1 Boundary Matrices and Elements Sets. The core mesh data structure in I♥MESH consists of boundary matrices and element sets. The built-in `MeshConnectivity` module provides access to a mesh's element sets, boundary matrices, and functions for creating indicator vectors from an element set and an element set from an indicator vector (via the vector's non-zero elements). Boundary matrices are sparse matrices. Although element sets are all encoded as sets of integers, element sets of different dimension (vertices, edges, faces, and cells) are distinct, incompatible types in I♥MESH. Moreover, I♥MESH tracks which mesh an element set originated from. This metadata allows type checking function parameters so that, for example, θ's parameter declared `v∈V` can be guaranteed to have originated from the mesh `M`, since `V,E,F = ElementSets(M)`. This data flow type checking also allows a function to be declared as only taking elements from a subset (e.g., non-boundary elements). The compiler tracks the originating mesh with an *owner* property attached to matrices and sets. Incompatible mesh origins raise an I♥MESH compile-time error. Set membership cannot in general be tracked at compile time, so it is checked at run-time.

4.1.2 Cellular Sets. We use the term *cellular set* to simply mean any collection of mesh elements, of possibly heterogeneous type—e.g., a single cellular set might contain vertices *and* faces. In the layer above boundary matrices, I♥MESH provides a library of basic operations on cellular sets. For instance, functions `Vertices`, `Edges`, `Faces`, and `Cells` have an overloaded meaning, based on the type of their input and the relative dimension:

- Given a `CellularSet`, they yield the elements within that set of a specific type. For example, applying `Edges` to a cellular set containing vertices, edges, and faces will yield only the edges.
- Given a higher-dimensional element (or homogeneous set of such elements), they yield its lower-dimensional components. For example, applying `Vertices` to a single triangle f will produce the three vertices of f; applying it to a set containing two adjacent triangles will yield their four shared vertices.
- Likewise, given a single lower-dimensional element (or homogeneous set of such elements), they yield the higher-dimensional elements containing it. For example, applying `Edges` to a single vertex v yields all edges e containing v; applying it to a set of vertices yields all edges incident on any vertex in that set.

All of these operations are trivial to implement in terms of boundary matrices. For instance, the following H♥rtLang code defines the cellular set operations for a polygonal mesh:

```
ElementSets, BoundaryMatrices, UnsignedBoundaryMatrices, NonZeros,
       IndicatorVector  from MeshConnectivity

M: FaceMesh

V, E, F = ElementSets( M )
`∂⁰`, `∂¹` = BoundaryMatrices(M)
`B⁰`, `B¹` = UnsignedBoundaryMatrices(M)
`B⁰ᵀ` = `B⁰`ᵀ
`B¹ᵀ` = `B¹`ᵀ

Vertices(S) = S₁ where S: CellularSet
Edges(S)    = S₂ where S: CellularSet
Faces(S)    = S₃ where S: CellularSet

Vertices(f) = vertexset(NonZeros(`B⁰` (`B¹` IndicatorVector({f})))) where f ∈ F
Vertices(G) = vertexset(NonZeros(`B⁰` (`B¹` IndicatorVector(G)))) where G ⊂ F
Vertices(e) = vertexset(NonZeros(`B⁰` IndicatorVector({e}))) where e ∈ E
Vertices(H) = vertexset(NonZeros(`B⁰` IndicatorVector(H))) where H ⊂ E

Edges(v) = edgeset(NonZeros(`B⁰ᵀ` IndicatorVector({v}))) where v ∈ V
Edges(f) = faceset(NonZeros(`B¹` IndicatorVector({f}))) where f ∈ F

Faces(v) = faceset(NonZeros(`B¹ᵀ` (`B⁰ᵀ` IndicatorVector({v})))) where v ∈ V
Faces(e) = faceset(NonZeros(`B¹ᵀ` IndicatorVector({e}))) where e ∈ E
```

For performance, we declare the transpose of `B`^{0} and `B`^{1} and add explicit parentheses around the matrix products to guarantee that matrix-vector products are performed (Section 8.1).

4.1.3 Neighborhoods and Stencils. I♥MESH provides a built-in library of neighborhood and stencil functions. For triangle meshes, this includes a vertex-vertex one ring, edge diamond, the vertices opposite an edge, the oriented vertices in a face, etc. For example, an edge diamond can be written in terms of cellular set operations:

```
Diamond(e) = CellularSet(Vertices(e), {e}, Faces(e)) where e ∈ E
```

This function constructs a cellular set where the vertex set contains the endpoints of `e`, the edge set is just `{e}`, and the face set consists of all faces containing `e`.

In contrast, the next vertex around a face requires orientation information from the boundary matrices two layers away:

```
NextVertexAroundFace(f, v) = vset_1 where v ∈ V, f ∈ F,
  eset = { e for e ∈ Edges(f) if `∂1`_e,f `∂0`_v,e = -1},
  vset = Vertices(eset) - {v}
```

This expression first builds a set `eset` of all edges of face `f` that are oriented as outgoing edges from vertex `v`. The product of boundary matrix elements computes this orientation. There should be only one such edge. This expression then builds `vset`, the set consisting of the edge's two vertices without the vertex parameter `v` itself. `vset` should have a single element; it is returned from the function.

Though somewhat complex, neighborhood and stencil functions need only be written once and can be shared and re-used many times.

4.2 Semantics of Architectural Layers

A cellular set is represented as a tuple containing a set of 0-dimensional vertices, 1-dimensional edges, 2-dimensional faces, and 3-dimensional cells. Some of these sets may be empty—e.g., for a surface mesh, the cell set is empty. Each type within the system is associated with a "source" tag, which tracks its originating mesh. Element sets are integer sets ranging from 1 to their respective dimensions.

The `NonZeros` and `IndicatorVector` functions are used to convert between set-based and vector-based encodings of a collection of mesh elements. Namely, `NonZeros` returns the set of nonzero indices of a sparse column vector; `IndicatorVector` constructs a sparse column vector with a nonzero entry 1 for each element in a given set. Pseudocode for these routines is given in Algorithms 1 and 2. Both functions have overloaded definitions, according to element type, but are implemented using the same backend function.

Algorithm 1 IndicatorVector

```
Input: a vertex set $s$                          ▷ Also works for an edge/face/cell set
Output: a vector
  $result \leftarrow zeros(s.source.vertices.length)$
  $result.source \leftarrow s.source$            ▷ Copy the source mesh
  for $index \in s$ do
      $result[index] \leftarrow 1$
  end for
  return $result$
```

Algorithm 2 NonZeros

```
Input: a vertex vector $v$  ▷ Also works for an edge/face/cell vector
Output: a set
  $result \leftarrow \{\}$
  $result.source \leftarrow v.source$            ▷ Copy the source mesh
  for $index \in range(|v|)$ do
      if $v[index] \neq 0$ then
          $result.insert(index)$
      end if
  end for
  return $result$
```

`MeshConnectivity` is the I❤MESH module that provides access to mesh data passed from the host language code. It provides both signed and unsigned boundary matrices. Signed matrices provide orientation information, while unsigned matrices are multiplied when querying neighbors. The following is the complete data structure for a face mesh in the C++ backend. It is initialized with signed boundary matrices, and generates and stores the unsigned matrices and element sets. It provides convenient accessors for this data. The dimensions of mesh elements are accessed by the implementation of `IndicatorVector`, ensuring compatibility for matrix-vector multiplications involving boundary matrices. As this data structure serves as the interface between ambient code and I❤MESH, we want it to be readable and minimally complex. As such, `IndicatorVector` and `NonZero` are implemented as separate helper functions.

```cpp
class FaceMesh {
public:
    typedef std::vector<int> Set;
    typedef Eigen::SparseMatrix<int> Matrix;
    typedef std::tuple<const Set&,const Set&,const Set&> FaceSetTuple;
    typedef std::tuple<const Matrix&, const Matrix&> FaceMatTuple;
    FaceMesh(){};
    FaceMesh(const Matrix& bm1_, const Matrix& bm2_) {
        bm1 = bm1_;
        bm2 = bm2_;
        pos_bm1 = bm1.cwiseAbs();
        pos_bm2 = bm2.cwiseAbs();
        Vi.resize(bm1.rows());
        for (int i = 0; i < bm1.rows(); ++i){ Vi[i] = i; }
        Ei.resize(bm1.cols());
        for (int i = 0; i < bm1.cols(); ++i){ Ei[i] = i; }
        Fi.resize(bm2.cols());
        for (int i = 0; i < bm2.cols(); ++i){ Fi[i] = i; }
    }
    int n_vertices() const{ return bm1.rows(); }
    int n_edges() const{ return bm1.cols(); }
    int n_faces() const{ return bm2.cols(); }
    FaceSetTuple ElementSets() const{ return std::tie(Vi, Ei, Fi); }
    FaceMatTuple BoundaryMatrices() const{ return std::tie(bm1, bm2); }
    FaceMatTuple UnsignedBoundaryMatrices() const{ return std::tie(pos_bm1,
      pos_bm2); }
private:
    Set Vi;
    Set Ei;
    Set Fi;
    Matrix bm1;     // |V|x|E|
    Matrix pos_bm1; // |V|x|E|
    Matrix bm2;     // |E|x|F|
    Matrix pos_bm2; // |E|x|F|
};
```

4.3 H❤rtLang Improvements

We substantially enhanced H❤rtLang with functionality to support a variety of practical mesh processing applications (Section 7).

4.3.1 Automatic Differentiation. Many geometry processing algorithms involve optimization over energies defined on meshes. We extended H❤rtLang with support for taking first and second derivatives (gradients and Hessians) of expressions. These derivatives can be with respect to scalar, vector, or matrix types—or sequences thereof. To do this, H❤rtLang applies automatic differentiation. Many automatic differentiation libraries require users to vectorize their variables, in the sense of enumerating them as a sequence. I❤MESH automates this vectorization for users. The resulting gradient or Hessian dimensions are determined by the total dimensions of the vectorized parameters. The precise enumeration is arbitrary—we use memory storage order—and opaque to users. To allow users to make use of these gradients and Hessians in further algebraic expressions, such as taking a step along the gradient direction, I❤MESH exposes its generalized vectorization operator `vec` and inverse `vec⁻¹`. For example, the following I❤MESH code takes a gradient descent step (line 12) towards minimizing the unweighted Laplacian of edge positions subject to a data term:

```
1  ElementSets from MeshConnectivity
2  Vertices from Neighborhoods(M)
3  vec, vec⁻¹ from linearalgebra
4
5  M: FaceMesh
6  V, E, F = ElementSets( M )
7  x_i ∈ ℝ³
8  y_i ∈ ℝ³
9  L = 1/|E| ∑_( e ∈ E ) ‖ x_i - x_j ‖² where i,j = Vertices(e)
10 D = 1/|V| ∑_( v ∈ V ) ‖ x_v - y_v ‖²
11 g = ∂(L+D)/∂x
12 `x'` = vec⁻¹_x( vec(x) - 0.1g )
```

4.3.2 Naming Enhancements. For mesh-related functions, it is common to use the same function name with different parameter types. For example, the `Vertices` function retrieves the endpoints of an edge, the three vertices of a triangle, or the four vertices of a tetrahedron. We extended H❤rtLang with support for function overloading based on parameter types. This support is based on H❤rtLang's type system, not the backend language. This is important, since H❤rtLang's element sets have the same C++ type in the generated code and Python does not support type-based function overloading at all. We also extended H❤rtLang to support passing parameters as subscripts. This notation appears commonly in the literature, as in Equation 1's θ.

Practitioners often have preferred names and notations for their functions. In support of this, we extended H❤rtLang to allow renaming symbols when importing them from a module. For example, the following snippet shortens a function name for more concise expressions.

```
NextVertexAroundFace as next from MeshConnectivity
```

This example renames a long yet clear name `NextVertexAroundFace` into the short and ambiguous name `next`. Authors often prefer short names in mathematical expressions—even single letters—with verbal descriptions in the text.

4.3.3 Scope and Recursion. We extended H❤rtLang to support locally scoped variable definitions for summations and functions. H❤rtLang's def-use analysis allows these definitions to be written in author-preferred order. We also added support for recursion. Together, these features allow for functional programming.

4.3.4 Improved Set Functionality. H❤rtLang allows users to define sets by filtering other sets with expressions and perform algebraic manipulations like unions and intersections. For example, the following expression collects vertices in the one ring closer than a threshold d:

```
S = {j for j ∈ VertexOneRing(i) if ‖x_i - x_j‖² < d}
```

5 DSL COMPARISONS

We characterized the design space of mesh processing DSLs in graphics to motivate I❤MESH and understand its unique set of attributes. We focused our investigation on I❤MESH, Simit [Kjolstad et al. 2016], Ebb [Bernstein et al. 2016], and MeshTaichi [Yu et al. 2022]. To understand these DSLs and illustrate our findings, we implemented the same mesh processing algorithm (per-vertex mean curvature) in all of them (Appendix A). We summarize our findings in Table 1 and Figure 5. Table 1 identifies, for each DSL, the data structure used for querying mesh element relationships, the approach to storing per-element attributes, built-in mesh types, the space of user-supportable mesh types, and the syntax in which users write their code. Figure 5 displays mesh DSLs in a Venn diagram among several characteristics: high performance, whether neighborhood sets must be defined across two languages, whether user-defined neighborhood sets can be composed easily (e.g., sharable as snippets versus interleaved among other code), whether the language supports global matrix assembly, and whether the syntax resembles conventional mathematical notation.

Appendix A displays the source code for each language, with color coding to highlight lines related to the mesh expression, stencil definition, and driver code. To calculate curvature, we must determine the dihedral angle for each edge. This requires accessing the oriented adjacent triangles for an edge (*oriented edge flaps*). We chose this example since this specific stencil or neighborhood function is not trivially supported and allows us to highlight how users create additional neighborhood functions in each language.

Ebb and Simit both draw inspiration from databases and focus on high-performance simulation. Ebb provides a three-layer architecture, enabling simulation code to be written on top of programmable relational domain libraries. These top two layers are written in the Lua programming language. (The bottom layer is the runtime.) Figure 2 in Bernstein et al. [2016] showcases a mass-spring simulation written in Ebb. Kernel functions can be applied over mesh elements directly. This approach leads to high performance and allows users to support any mesh type. For our dihedral angle example, the built-in domain library for triangle meshes lacks oriented edge flaps. Adding this relation requires the creation of a new domain library. We copied-and-pasted the triangle mesh library and modified some lines to add new, needed relations (Figure 9, `EdgeFlapMesh.t`, yellow highlighted lines). We could have re-architected the included triangle mesh library for inheritance, but one would still run into the problem of code re-use when trying to mix in additional stencils via, for example, multiple inheritance. For this reason, we do not consider Ebb to support composable neighborhood definitions (Figure 5). Ebb's strict layer separation does not allow for lightweight definition of new relations.

Simit emphasizes the local/global distinction between local assembly kernels and high-performance global linear algebra [Bernstein and Kjolstad 2016]. It facilitates a seamless transition between graph and matrix views of the simulation. Figure 7 in Kjolstad et al. [2016] demonstrates how the global matrix can be assembled. Users define hyperedges between sets of structs in the Simit language and bind them in C++. Hypergraphs in Simit allow users to define various stencils on any mesh. Kernel functions can be directly applied to hypergraph sets. To implement the dihedral angle formula in Simit, we defined a hyperedge consisting of the endpoint vertices and adjacent faces given an edge (Figure 9, (a) Simit code, green highlighted lines 13–15). We then prepared the stencils (Figure 9, (a) C++ code, yellow highlighted lines) and bind the data in C++ (Figure 9, (a) C++ code, green highlighted lines). While this approach allows for high-performance execution over any mesh type, stencil implementation must be written entirely by the user in C++. Composition is possible but entirely delegated to the user's approach to defining connectivity; if users choose, they may build C++ libraries defining composable neighborhoods, but the libraries will be tied to the user's C++ mesh data structure, which Simit is agnostic to.

MeshTaichi is designed to accelerate mesh processing by structuring memory access patterns on the GPU. It is a DSL embedded in Python. Its optimizations are conceptually similar to the GPU mesh library RXMesh [Mahmoud et al. 2021]. MeshTaichi partitions the mesh into patches for cache efficiency when accessing per-element attributes. (RXMesh does the same for accessing mesh neighbor relationships instead.) MeshTaichi supports triangle and

Table 1. For each DSL: the data structure used for querying mesh element relationships, how per-element attributes are stored, built-in mesh types, the space of user-supportable mesh types, and the syntax users write their code in. Compatible mathematical objects means various I❤MESH types including matrices, sequences, or functions taking objects. See the text for details.

Mesh DSL	Relation data structure	Attribute storage	Mesh types (built-in)	Mesh types (supported)	Syntax
Ebb	relational models	contiguous arrays	triangle/tetrahedral meshes 2D/3D regular grids polygonal/polyhedral meshes	any	Lua
Simit	hypergraphs	dense and sparse tensors	none	any	MATLAB-like and C++
MeshTaichi	V/E/F/C-V/E/F/C incidence relations	contiguous arrays	triangle/tetrahedral meshes	triangle/tetrahedral meshes	Python
I❤MESH	boundary matrices	compatible mathematical objects	triangle/tetrahedral meshes point clouds polygonal/polyhedral meshes	simplicial (cellular) sets	H❤rtLang

tetrahedral meshes, with no opportunity for extensibility. MeshTaichi provides built-in support for element incidence relationships. However, accessing information beyond the one-ring neighbors necessitates awkward iteration to gather stencil elements. In order to query the orientations for our oriented edge flap neighborhood, we implemented the stencil as interleaved lines of code with the computation (Figure 9, (c) MeshTaichi, yellow highlighted lines). We could have instead separated the stencil by gathering a set of local variables—other DSLs enforce this—at the cost of lengthier code. As in Simit, composition is possible but entirely delegated to the user's approach to defining connectivity. In MeshTaichi, unlike Simit, neighborhoods are written in the same language as computations, which increases compatibility between such code.

I❤MESH is designed to facilitate executing mesh processing expressions as conventionally written in papers. The emphasis is on syntax and flexible neighborhood definitions, rather than performance. Users write mesh expressions and neighborhood functions in H❤rtLang. Mesh relation data is provided and stored as signed boundary matrices (Section 3.1). This is general enough to support the well-studied CW complexes and simplicial sets from algebraic topology. In particular, this is capable of representing triangle/tetrahedral meshes, polygonal/polyhedral meshes, regular grids, point clouds, and more exotic meshes used in mesh processing applications (tufted covers Sharp and Crane [2020] and intrinsic triangulations [Sharp et al. 2021]). However, this is more limited than Ebb's relational models or Simit's hypergraphs, which can represent inconsistent relationships, such as an edge set (pairs of vertices) that doesn't match the triangle set (triplets of vertices). In I❤MESH, one could supply multiple meshes associated with the same element attributes, but not relate elements in one mesh to the other. Attributes can be stored in any compatible mathematical object supported by H❤rtLang, e.g., by subscripting a sequence, matrix, vector, or calling a function. I❤MESH provides many built-in neighborhood functions. New neighborhood relations can be written succinctly and access any of the architectural layers (Figure 9, `OrientedOppositeFaces` in I❤MESH code, yellow highlighted lines). Users have the flexibility to write them where they are needed (as in the example) or create custom libraries of neighborhoods. Neighborhoods can make use of other existing neighborhood functions, fundamental incidence relationships, or the boundary matrices themselves. Users can share individual or libraries of neighborhoods, similar to how BibTeX is shared. I❤MESH provides a library of common neighborhood functions.

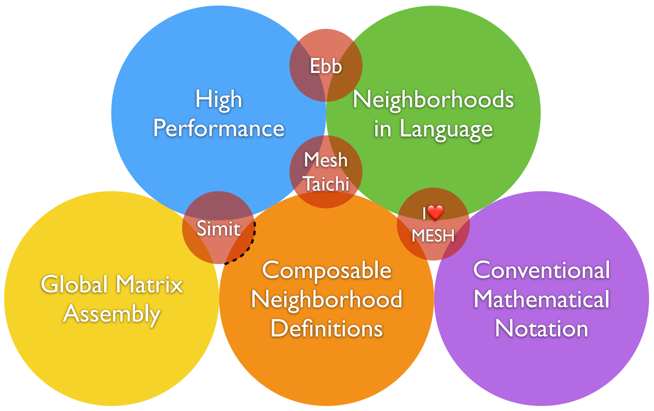

High Performance
Ebb
Neighborhoods in Language
Mesh Taichi
Simit
I❤ MESH
Global Matrix Assembly
Composable Neighborhood Definitions
Conventional Mathematical Notation

Fig. 5. A Venn diagram of mesh processing DSL attributes. Mesh processing DSLs occupy different points in the design space. *High Performance* refers to the intended use of the DSL for efficient computation (on CPUs and GPUs). *Neighborhoods in Language* refers to the ability to write neighborhood or stencil definitions in the same rather than a second language. *Conventional Mathematical Notation* refers to the similarity of the DSL syntax to notation used in, e.g., papers. *Global Matrix Assembly* refers to the ability of the DSL to construct a sparse, global system matrix where each mesh element contributes to a specific block within the matrix. *Composable Neighborhood Definitions* refers to the ability to compose neighborhood and stencil definitions written separately (i.e., an individual neighborhood is composable if it can be easily shared as a code snippet versus interleaved among other code). Simit is agnostic to the user's choice of mesh data structure. As a result, the composability of its neighborhood definitions is dependent on users choosing consistent mesh data structures. I❤MESH emphasizes exploration as the only system with neighborhood accessors written in the language, composable mesh types, and conventional mathematical notation.

6 IMPLEMENTATION

We implemented I❤MESH as a fork of I❤LA [Li et al. 2021], inheriting the core compiler infrastructure for H❤rtLang. I❤MESH was designed to minimize backend-specific development effort, allowing it to scale to more backends. Most of our changes were written to the Python-based compiler itself. The upper layers of I❤MESH were written in H❤rtLang. Each target language interfaces with I❤MESH by creating boundary matrices from the input data. These must be created by the user and passed when initializing their generated I❤MESH module. I❤MESH provides convenience routines in the target languages for creating these data structures from various kinds of mesh data. We have written such routines for triangle meshes (all backends) as well as polygon meshes, tetrahedral meshes, and point clouds (C++). Our point cloud mesh construction defines edge connectivity based on radius or k-nearest neighbors. (Two points are considered neighbors if either one of them has the other as a k-nearest neighbor.) We do this efficiently with a KD-Tree built using the nanoflann library [Blanco and Rai 2014].

I♥MESH employs reverse mode automatic differentiation from the autodiff library [Leal 2023] to obtain derivatives. Derivative support is currently only implemented in the C++ backend. (Full MATLAB and Python support are left for the future.) The autodiff library provides a `var` type to monitor variables with which to differentiate. In the generated code, any symbol whose definition makes use of one of these variables must itself be an autodiff `var` type. I♥MESH generates code assuming that module parameters are of double type and only converts them to `var` types inside the module when derivatives are encountered in the source code. In lieu of creating a top-down Directed Acyclic Graph (DAG) tracking all symbols' interacting with derivative operations, I♥MESH relies on generic programming with a simple rule. The left-hand side of a derivative operator is of double type, while all other newly defined non-integer symbols are of `var` type. Additionally, the system generates C++ templated functions so that functions make use of and return double or `var` type depending on the parameters. This approach is conservative and may create more `var` types than necessary. In the future, we plan to explore AST-based automatic differentiation [Guenter 2007; Laue 2022], removing any backend library requirements.

I♥MESH heavily relies on set operations when collecting neighbor and stencil elements. In C++, set support is implemented as a sorted `std::vector` with `STL` set operations for efficiency (versus `std::set` and `std::unordered_set`).

7 RESULTS

To demonstrate I♥MESH's breadth, we implemented a variety of geometry processing techniques across different representations (Figures 6 and 7) and verified correctness of the generated code by comparing the output to reference implementations. Table 2 summarizes these examples. Please see the supplemental material for the full I♥MESH source code for each example, along with generated C++, an easily-compiled C++ driver program, and the typeset generated LaTeX. The I♥MESH compiler itself is also included.

Vertex Normals. This example demonstrates I♥MESH's support for three kinds of meshes and many kinds of neighborhoods. We implemented multiple vertex normal techniques for triangle meshes, polygon meshes, and point clouds (Figure 6). For triangle and polygon meshes, we implemented a classic angle-weighted formula due to Max [1999]. For each face incident at a vertex, the expression accesses the next and previous neighboring vertices around the oriented face. For triangle meshes, we also implemented formulas for Gaussian and mean curvature normals from Crane et al. [2013], which we discuss under Discrete Curvature. Point clouds lack faces, so the normal estimation is entirely different. We implemented a classic technique for estimating unoriented normals as the third

Table 2. Information about our mesh processing examples.

- V⟶F: given a vertex, find its incident faces.
- V⟶V: given a vertex, find vertices sharing an edge with it.
- FV⟶VO: given a face and one of its vertices, find the oriented vertex-vertex neighbors of the vertex in the face.
- F⟶VO: given a face, find its oriented vertices.
- C⟶VO: given a cell, find its oriented vertices.
- VV⟶FO: given two edge-sharing vertices, find faces in the diamond oriented with the first face containing the vertices in the given orientation.
- VV⟶VO: given two vertices sharing an edge, find the remaining two vertices in the diamond with the same orientation as VV⟶FO.

Bilateral filtering finds vertex-vertex neighbors within a given radius. Geodesic distance finds vertex-vertex k-ring neighborhoods.

Example	Mesh types	Neighborhoods	Lines of I♥MESH	Derivatives
Vertex Normals	Triangle mesh	V⟶F, FV⟶VO	8	No
	Polygon mesh	V⟶F, FV⟶VO	8	No
	Point cloud	V⟶V	12	No
Discrete Curvature	Triangle mesh	V⟶F, FV⟶VO, V⟶V, VV⟶FO, F⟶VO	29	No
Winding Number	Triangle mesh	F⟶VO	15	No
Bilateral Filtering	Triangle mesh	V⟶V*, FV⟶VO, V⟶F	32	No
Geodesic Distance	Triangle mesh	V⟶V*, FV⟶VO	21	No
Mesh Parameterization	Triangle mesh	F⟶VO	25	Yes
Mesh Deformation	Tetrahedral mesh	C⟶VO	39	Yes
Mass Spring	Tetrahedral mesh	V⟶V	24	No

principal component of each vertex and its vertex neighbors [Mitra and Nguyen 2003]. The C++ driver code for these examples is minimal, directly calling code generated by I♥MESH to obtain each vertex normal.

Discrete Curvature. We implemented Meyer et al. [2003]'s discrete Gaussian and mean curvature expressions for triangle meshes and Crane et al. [2013]'s formulas for Gaussian and mean curvature normals. This example demonstrates several additional neighborhoods in I♥MESH such as those required for computing corner angles opposite edge flaps. Figure 6 shows these curvatures and curvature normals. The C++ driver code for these examples is also minimal.

Winding Number. We implemented the generalized winding number formula [Jacobson et al. 2013] and applied it to fill holes in a triangle mesh (Figure 6). The formula involves a summation over all faces of a triangle mesh and computing a quantity requiring each face's oriented vertices. The C++ code loads a triangle mesh and a constrained Delaunay tessellation (CDT) of that mesh's convex hull, before using the I♥MESH module to compute the generalized winding number at each tetrahedron's barycenter, then extracting a closed triangle mesh as the boundary of tetrahedra with winding number $\leq \frac{1}{2}$.

Bilateral Filtering. We implemented the bilateral filtering algorithm introduced in Fleishman et al. [2003] for denoising meshes. This example required implementing a new recursive neighborhood function in H♥rtLang that progressively expands the set of one-ring vertex neighbors until all desired vertices have been collected.

Figure 6 displays a noisy input mesh and the smoothed output after ten iterations. The C++ in this example is minimal.

Geodesic Distance. We implemented the geodesic distance algorithm described by Calla et al. [2019], which uses a fast marching method to update the geodesic distance from a point in a growing wavefront. To enable the propagation of topological level sets, we wrote a neighborhood function that retrieves the sequence of k-ring neighboring vertices for a given set of vertices, with $k = 1$ initially. In this example, we use C++ for the outer loop iteration, which calls the propagation function written in I♥MESH in each step. Figure 7 visualizes the geodesic distance to a point on a dragon mesh.

Mesh Parameterization. This example demonstrates I♥MESH's automatic differentiation abilities for parameterizing a triangle mesh. The algorithm works by minimizing the distortion of a mesh mapped to the plane via the symmetric Dirichlet energy [Schreiner et al. 2004; Smith and Schaefer 2015]. The energy is computed in I♥MESH for each face and makes use of its oriented vertices to calculate the map's Jacobian. To optimize the energy, we obtain the gradient and Hessian of the energy automatically via automatic differentiation, projecting the Hessian to be positive semi-definite (psd) [Teran et al. 2005]. I♥MESH doesn't support automatically gathering only the non-zero elements of each face's energy, so the Hessian matrix is large.[2] As a result, the psd projection was too slow when written in H♥rtLang, since H♥rtLang can't express the logic for a gather/scatter step to exploit the particular structure of the matrices [Schmidt et al. 2022]. Instead, we use a callback function written in C++. The only other C++ code is the Newton's method and line search outer loop as in [Schmidt et al. 2022]. Figure 7 displays the optimized result for an animal model.

Tetrahedral Mesh Deformation. We implemented a variety of energies from Schmidt et al. [2022] to demonstrate tetrahedral mesh neighborhoods and automatic differentiation. We implemented four distinct energy functions: symmetric Dirichlet [Schreiner et al. 2004; Smith and Schaefer 2015], exponential symmetric Dirichlet [Rabinovich et al. 2017], AMIPS, and conformal AMIPS [Fu et al. 2015]. The gradient and Hessian are automatically computed and projected in a similar manner to the mesh parameterization example. Figure 7 presents the optimized outcome for a tessellated cube.

Mass Spring. To demonstrate a physical simulation, we implemented a mass-spring system with explicit integration [Baraff and Witkin 1998] directly in I♥MESH. This example operates on a tetrahedral mesh, which has internal edges for structural support. The I♥MESH module is initialized with the rest state, mass, stiffness, damping, and floor height. The I♥MESH module exposes two functions, one which computes internal forces and one which time steps the system forward. The C++ driver code simply calls the two routines in alternating order. The internal forces for each vertex were computed by considering its neighboring vertices within a one-ring structure; these forces are then used to compute the velocity of each vertex. We use conditional assignments to handle boundary conditions during force application. A video of our simulation is provided in the supplemental material.

8 OBSERVATIONS, LIMITATIONS, AND FUTURE WORK

Implementing these examples required writing relatively few lines of H♥rtLang (Table 1). We verified the correctness of I♥MESH's generated code by comparing the output to reference implementations. It is unclear how to quantify the lines of driver C++ code. Code to load a mesh and visualize the output is lengthy but boilerplate. The examples which used more specialized C++ did so to access specific functionality unsupported by I♥MESH directly. Two examples (parameterization, deformation) made use of C++ optimization routines written outside I♥MESH. Clearly, this impacts the portability of these examples. One direction of future work would be to automatically choose an appropriate optimization routine. However, we would then either need to tie our output to a per-platform optimization package (introducing a similar dependency as our current solution) or else include our own per-platform implementation of optimization routines. These examples also used C++ code to project Hessian matrices to be positive, semi-definite. We initially wrote this routine in H♥rtLang, but were unable to access the small, dense submatrix necessary for reasonable performance. Two of the examples (geodesic distance, bilateral filtering) made use of dynamic neighborhoods. These were implementable in H♥rtLang as recursive functions with set algebra operations. They may have been more naturally described with imperative logic. Our support for recursive functions in theory allows for any logic to be written in H♥rtLang, but this is impractical in general. I♥MESH also does not support topology-modifications, such as edge collapses.

8.1 Performance

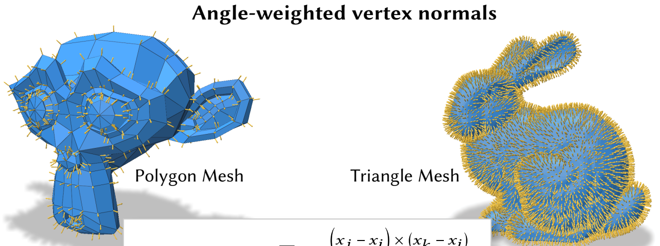
High performance is not a goal of I♥MESH. Instead, we aim for the compiler to produce backend code sufficient for prototyping, and ideally with reasonable scaling characteristics. Users can use I♥MESH to quickly explore potential approaches; the generated code can be subsequently optimized by hand if users wish. Our examples run in a few seconds (normals, curvature, winding number, geodesic distance, mass-spring) to minutes (bilateral filtering, parameterization, deformation). Of the lengthier examples, all but deformation take $<$ 20 seconds per iteration.

We evaluated the basic scaling of I♥MESH by measuring the cost of finding a vertex's vertex neighbors ($|\partial^1||\partial^1|^\top v$), which should run in constant time (on average).[3] We found two $O(n)$ confounders in our naive expression. First, sparse matrix · sparse vector multiplication should only be constant-time with compressed column matrix storage (CSC). Since the default storage order is CSC, this computation involving the transposed matrix · vector product uses, in effect, compressed row format (CSR). Storing a copy of the transposed boundary matrices prevents this. This can be easily done in H♥rtLang itself. Second, without parentheses, the generated code may compute a matrix · matrix product followed by a matrix · vector product. Inserting parentheses prevents this, as in $|\partial^1|(|\partial^1|^\top v)$.

[2]The Hessian is also dense, though theoretically sparse. Automatic sparse Hessian computation is an area of active research and out of scope for I♥MESH.

[3]A manifold mesh vertex has, on average, six neighbors. For this experiment, we used a k-NN mesh with $k = 6$.

Angle-weighted vertex normals

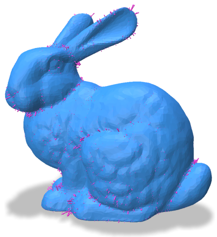


$$VertexNormal\,(i) = \sum_{f \in Faces(i)} \frac{(x_j - x_i) \times (x_k - x_i)}{\|x_j - x_i\|^2 \|x_k - x_i\|^2}$$
$$\text{where } j, k = NeighborVerticesInFace\,(f, i)$$
$$\text{where } i \in V$$

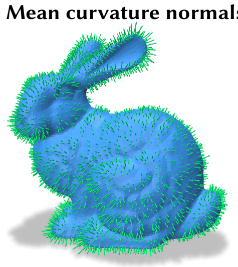
polygon_normal.hlang/vertex_normal.hlang:

```
VertexNormal(i) = (∑_(f ∈ Faces(i)) (x_j- x_i)×(x_k-x_i)/(‖x_j-x_i‖² ‖x_k-x_i‖²)
where j, k = NeighborVerticesInFace(f, i) ) where i ∈ V
```

Gaussian curvature normals



Mean curvature normals



$$KN\,(i) = \frac{1}{2}\left(\sum_{j \in VertexOneRing(i)} \frac{\phi_{i,j}}{l_{i,j}}\,(x_j - x_i)\right) \text{where } i \in V$$

$$HN\,(i) = \frac{1}{2} \sum_{j \in VertexOneRing(i)} (cot(\alpha) + cot(\beta))\,(x_i - x_j)$$
$$\text{where } k, p = OppositeVertices\,(i, j)\,, cot(\alpha) = cot\,(k, j, i)\,, cot(\beta) = cot\,(p, i, j)$$
$$\text{where } i \in V$$

vertex_normal.hlang:

```
KN(i) = 1/2 (∑_(j ∈ VertexOneRing(i)) ϕ_ij/l_ij (x_j - x_i))
```

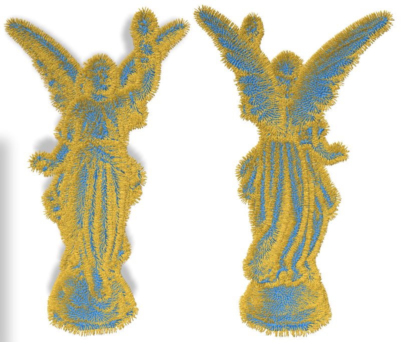
```
HN(i) = 1/2 (∑_(j ∈ VertexOneRing(i))(`cot(α)` + `cot(β)`) (x_i - x_j)
where k, p = OppositeVertices(i, j),
`cot(α)` = cot(k, j, i),
`cot(β)` = cot(p, i, j) ) where i ∈ V
```

Point cloud normals (unoriented)

point_cloud.hlang:

```
Normal(v) = vv_*,3 where v ∈ V,
N = VertexOneRing(v),
p̄ = ∑(_(n ∈ N) x_n)/|N|,
d = {x_v-p̄ for v ∈ N},
m_i,* = d_i,
u, `∑`, vv = svd(m)
```

$$Normal\,(v) = vv_{*,3}$$
$$\text{where}$$
$$v \in V$$
$$N = VertexOneRing\,(v)$$
$$\bar{p} = \frac{\sum_{n \in N} x_n}{|N|}$$
$$d = \{x_v - \bar{p} \mid v \in N\}$$
$$m_{i,*} = d_i$$
$$u, \Sigma, vv = \text{svd}\,(m)$$



Gaussian curvature

Mean curvature

$$K\,(i) = 2\pi - \sum_{f \in Faces(i)} \theta_{i,f} \text{ where } i \in V$$

$$H\,(i) = \frac{1}{4} \sum_{j \in VertexOneRing(i)} l_{i,j}\phi_{i,j} \text{ where } i \in V$$

discrete_curvature.hlang:

```
K(i) = 2π - ∑_(f ∈ Faces(i)) θ_i,f where i ∈ V
```

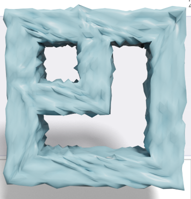
```
H(i) = 1/4 ∑_(j ∈ VertexOneRing(i)) l_ij ϕ_ij where i ∈ V
```

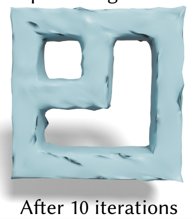
Bilateral Filtering

bilateral_denoising.hlang:

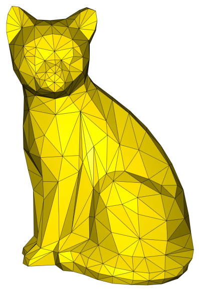
```
DenoisePoint(i) = x_i + n·(s/norm)  where i ∈ V,
n = VertexNormal(i),
`σ_c` = CalcSigmaC(i),
neighbors = AdaptiveVertexNeighbor(i, {i}, `σ_c`),
`σ_s` = CalcSigmaS(i, neighbors),
s = ∑(_(v ∈ neighbors) CalcS(i, v, n, `σ_c`, `σ_s`)),
norm = ∑(_(v ∈ neighbors) CalcNorm(i, v, n, `σ_c`, `σ_s`))
```

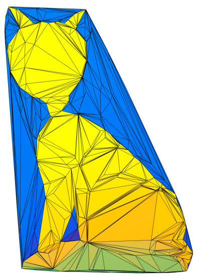
$$DenoisePoint\,(i) = x_i + n \cdot \left(\frac{s}{norm}\right)$$
$$\text{where}$$
$$i \in V$$
$$n = VertexNormal\,(i)$$
$$\sigma_c = CalcSigmaC\,(i)$$
$$neighbors = AdaptiveVertexNeighbor\,(i, \{i\}, \sigma_c)$$
$$\sigma_s = CalcSigmaS\,(i, neighbors)$$
$$s = \sum_{v \in neighbors} CalcS\,(i, v, n, \sigma_c, \sigma_s)$$
$$norm = \sum_{v \in neighbors} CalcNorm\,(i, v, n, \sigma_c, \sigma_s)$$


Input triangle mesh


After 10 iterations

Generalized Winding Number


Input triangle mesh

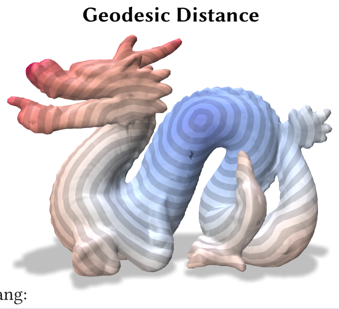

Generalized winding number values inside the convex hull's constrained Delaunay tesselation

winding_number.hlang:

```
w(p) = 1/(4π) ∑_(f ∈ F) Ω_f(p) where p ∈ ℝ³
```

$$w\,(\mathbf{p}) = \frac{1}{4\pi} \sum_{f \in F} \Omega_f\,(\mathbf{p}) \text{ where } \mathbf{p} \in \mathbb{R}^3$$

Fig. 6. Results implemented using I❤️MESH. See Section 7 for details. See the supplemental material for the complete source code.



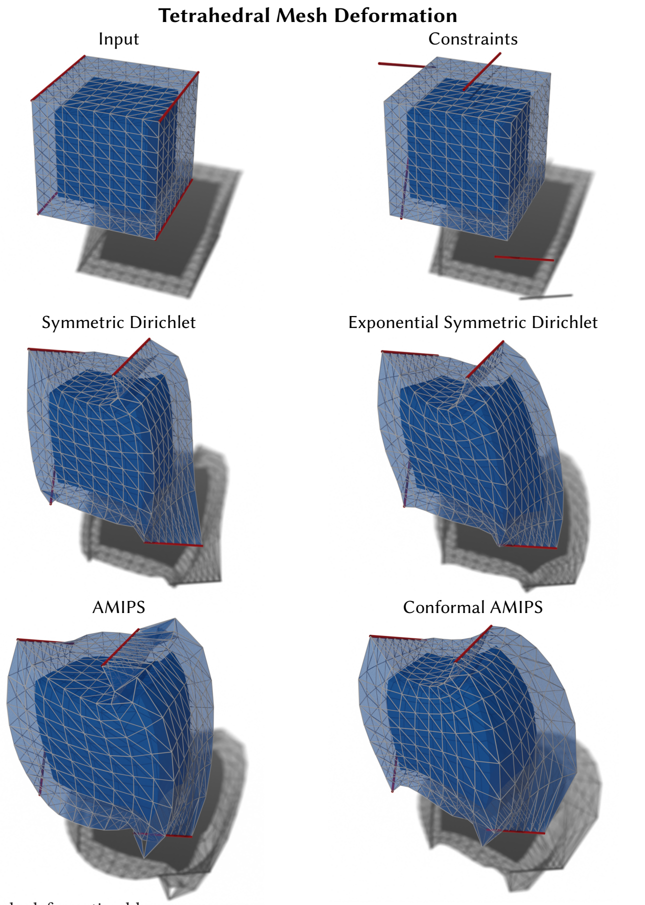
geodesic_distance.hlang:

```
UpdateStep(i, j, k, d) = { p if s_1,1 < 0 and s_2,1 < 0
                           min(d_j+‖x_j - x_i‖, d_k+‖x_k - x_i‖) otherwise
where i,j,k ∈ V, d_i ∈ ℝ,
X = [x_j-x_i x_k-x_i],
t = [d_j d_k]ᵀ,
Q = (XᵀX)⁻¹,
`1` = [1 ; 1],
p = (`1`ᵀQt + sqrt((`1`ᵀQt)² - `1`ᵀQ`1` · (tᵀQt - 1)))/ (`1`ᵀQ`1`),
n = XQ(t- p ·`1`),
s = QXᵀn
```

$$UpdateStep\left(i,j,k,d\right)=\begin{cases}p & \text{if } s_{1,1}<0 \text{ and } s_{2,1}<0\\ \min\left(d_j+\left\|x_j-x_i\right\|,d_k+\left\|x_k-x_i\right\|\right) & \text{otherwise}\end{cases}$$

where

$$i,j,k\in V$$
$$d_i\in\mathbb{R}$$
$$X=\begin{bmatrix}x_j-x_i & x_k-x_i\end{bmatrix}$$
$$t=\begin{bmatrix}d_j & d_k\end{bmatrix}^T$$
$$Q=\left(X^TX\right)^{-1}$$
$$1=\begin{bmatrix}1\\1\end{bmatrix}$$
$$p=\frac{1^TQt+\sqrt{\left(1^TQt\right)^2-1^TQ1\cdot\left(t^TQt-1\right)}}{1^TQ1}$$
$$n=XQ\left(t-p\cdot 1\right)$$
$$s=QX^Tn$$



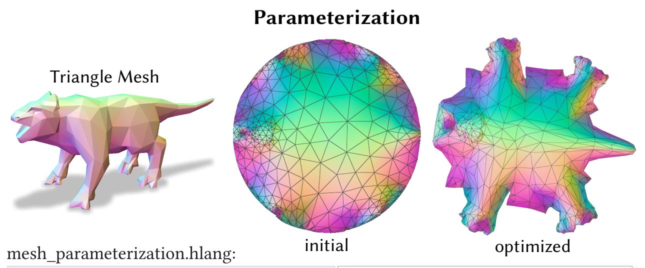
mesh_deformation.hlang:

```
CAMIPS(s, x) = { ∞ if |m| ≤ 0
                 vol_abcd ‖(‖J²/|J|^⅔) otherwise
where s ∈ C, x_i ∈ ℝ³,
a, b, c, d = OrientedVertices(s),
m = [x_b-x_a x_c-x_a x_d-x_a],
J = m `m_r`(s)⁻¹

E2 = w ∑_j ‖bp_j - x_(bx_j)‖²

e = ∑_(i ∈ C) S_i(x) + E2
G = ∂e/∂x
H = ∑_(i ∈ C) psd(∂²S_i(x)/∂x²) + psd(∂²E2/∂x²)
```

$$CAMIPS\left(s,x\right)=\begin{cases}\infty & \text{if } |m|\leq 0\\ vol_{a,b,c,d}\left(\frac{\|J\|^2}{|J|^{\frac{2}{3}}}\right) & \text{otherwise}\end{cases}$$

where

$$s\in C$$
$$x_i\in\mathbb{R}^3$$
$$a,b,c,d=OrientedVertices\left(s\right)$$
$$m=\begin{bmatrix}x_b-x_a & x_c-x_a & x_d-x_a\end{bmatrix}$$
$$J=mm_r\left(s\right)^{-1}$$
$$E2=w\sum_j\left\|bp_j-x_{bx_j}\right\|^2$$
$$e=\sum_{i\in C}S_i\left(x\right)+E2$$
$$G=\frac{\partial e}{\partial x}$$
$$H=\sum_{i\in C}psd\left(\frac{\partial^2 S_i\left(x\right)}{\partial x^2}\right)+psd\left(\frac{\partial^2 E2}{\partial x^2}\right)$$



mesh_parameterization.hlang:

```
S(f, x) = { ∞ if |m| ≤ 0
            A ‖(‖J² + ‖J⁻¹‖²) otherwise
where f ∈ F, x_i ∈ ℝ²,
a, b, c = OrientedVertices(f),
m = [x_b-x_a x_c-x_a],
A = ½ |`m_r`(f)|,
J = m `m_r`(f)⁻¹

e = ∑_(i ∈ F) S_i(x)

H = ∑_(i ∈ F) psd(∂²S_i(x)/∂x²)

G = ∂e/∂x
```

$$S\left(f,x\right)=\begin{cases}\infty & \text{if } |m|\leq 0\\ A\left(\|J\|^2+\left\|J^{-1}\right\|^2\right) & \text{otherwise}\end{cases}$$

where

$$f\in F$$
$$x_i\in\mathbb{R}^2$$
$$a,b,c=OrientedVertices\left(f\right)$$
$$m=\begin{bmatrix}x_b-x_a & x_c-x_a\end{bmatrix}$$
$$A=\frac{1}{2}\left|m_r\left(f\right)\right|$$
$$J=mm_r\left(f\right)^{-1}$$
$$e=\sum_{i\in F}S_i\left(x\right)$$
$$H=\sum_{i\in F}psd\left(\frac{\partial^2 S_i\left(x\right)}{\partial x^2}\right)$$
$$G=\frac{\partial e}{\partial x}$$

Fig. 7. Results implemented using I❤️MESH. See Section 7 for details. See the supplemental material for the complete source code.

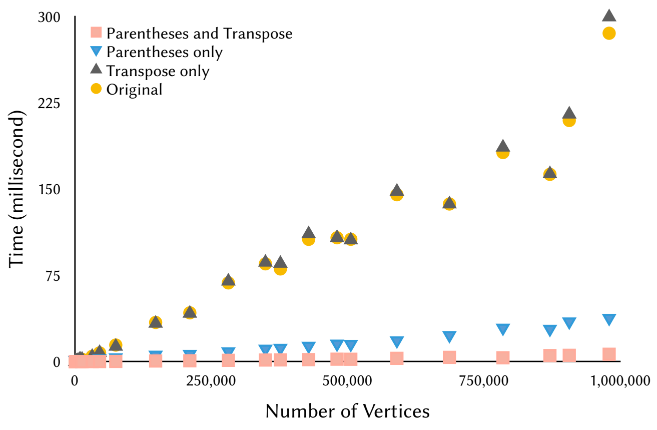


Fig. 8. The average time for computing a per-vertex normal in different conditions (Section 8.1). *Parentheses* signifies that the matrix-vector product takes precedence and is performed before other operations. *Transpose* refers to storing a pre-transposed boundary matrix for efficient sparse matrix-vector multiplication.

We measured these performance improvements by computing per-vertex normals for various $k = 6$ nearest-neighbor point clouds with our C++/Eigen backend. Figure 8 plots the time to compute a single vertex's normal in each condition for meshes with varying numbers of vertices (up to 1 million). Each measurement was obtained by computing 100 normals for each input, repeated 10 times, and then averaging. As shown, the largest improvement is due to avoiding the matrix-matrix multiplication with appropriate parentheses. There is no performance improvement from using the stored CSC transpose when performing matrix-matrix multiplication, because efficient matrix-matrix multiplication requires converting one side to CSR format and the other to CSC. The linear confounding factor from "transposed-matrix" · vector multiplication is comparatively smaller. With both modifications, performance appears virtually constant, though we objectively measured a small remaining linear factor (4×10^{-6} milliseconds per mesh vertex). For comparison, computing point cloud normals in the same way using hand-written C++ takes 0.004 ms versus 7 ms per vertex. (The C++ implementation used pre-computed vertex adjacency stored in arrays.) We have found it relatively straightforward to express neighborhood functions with boundary matrices and element sets. However, it is not always easy to maintain optimal complexity.

While not a goal, this experiment points to the potential for the I❤LA compiler to drastically improve performance via a small set of standard optimizations such as common subexpression elimination. One direction worth pursuing is generating code for high-performance languages like MeshTaichi [Yu et al. 2022] and Halide [Ragan-Kelley et al. 2013]. We would also like to add source-to-source automatic differentiation support to H❤rtLang. This would eliminate a dependency from the generated code and avoid the high runtime cost.

8.2 Authoring Support

In the spirit of experimenting with mesh expressions, we would like to explore automatic generation of mathematical illustrations. We envision illustrations similar to and leveraging Penrose [Ye et al. 2020] by displaying a mesh annotated with expressions from the I❤MESH code. We would also like to integrate I❤MESH with H❤rtDown [Li et al. 2022] to create a notebook-like playground for experimenting with mesh expressions. I❤MESH provides support for the discrete mesh expressions commonly written in geometry processing papers. Many papers also include a continuous formulation of their problem and solution. A future language could allow authors to express the continuous version of their problem, while selecting which I❤MESH discretization to use to implement it.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their suggestions. Alec Jacobson was supported in part by the Sloan Foundation and the Canada Research Chairs Program. Keenan Crane was supported in part by the David and Lucile Packard Foundation, NSF awards 1943123 and 2212290, and gifts from Meta Reality Labs and Google, Inc. Yotam Gingold was supported in part by a gift from Adobe Inc.

REFERENCES

Samer Alhaddad, Jens Förstner, Stefan Groth, Daniel Grünewald, Yevgen Grynko, Frank Hannig, Tobias Kenter, Franz-Josef Pfreundt, Christian Plessl, Merlind Schotte, Thomas Steinke, Jürgen Teich, Martin Weiser, and Florian Wende. 2022. The HighPerMeshes framework for numerical algorithms on unstructured grids. *Concurrency and Computation: Practice and Experience* 34, 14 (2022), e6616.
Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, and others. 2005. The Fortress language specification. *Sun Microsystems* 139, 140 (2005).
Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Software* 40, 2 (Feb. 2014), 1–37. https://doi.org/10.1145/2566630
David Baraff and Andrew Witkin. 1998. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 43–54.
Igor A. Baratta, Joseph P. Dean, Jørgen S. Dokken, Michal Habera, Jack S. Hale, Chris N. Richardson, Marie E. Rognes, Matthew W. Scroggs, Nathan Sime, and Garth N. Wells. 2023. DOLFINx: the next generation FEniCS problem solving environment. preprint. https://doi.org/10.5281/zenodo.10447666
Bruce G Baumgart. 1972. *Winged edge polyhedron representation*. Technical Report. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.
Gilbert Louis Bernstein and Fredrik Kjolstad. 2016. Perspectives: why new programming languages for simulation? *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–3.
Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–12.
Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.
Botsch Steinberg Bischoff, M Botsch, S Steinberg, S Bischoff, L Kobbelt, and Rwth Aachen. 2002. OpenMesh–a generic and efficient polygon mesh data structure. In *In OpenSG Symposium*.
Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann.
Camille Brunel, Pierre Bénard, and Gaël Guennebaud. 2021. A time-independent deformer for elastic contacts. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–14.
Luciano A Romero Calla, Lizeth J Fuentes Perez, and Anselmo A Montenegro. 2019. A minimalistic approach for fast computation of geodesic distances on triangular meshes. *Computers & Graphics* 84 (2019), 77–92.
Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. 1998. Directed edges—a scalable representation for triangle meshes. *Journal of Graphics tools* 3, 4 (1998), 1–11.
Marcel Campen, Hanxiao Shen, Jiaran Zhou, and Denis Zorin. 2019. Seamless parametrization with arbitrary cones for arbitrary genus. *ACM Transactions on*

Graphics (TOG) 39, 1 (2019), 1–19.

Prashanth Chandran, Loïc Ciccone, Markus Gross, and Derek Bradley. 2022. Local anatomically-constrained facial performance retargeting. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–14.

Keenan Crane, Fernando de Goes, Mathieu Desbrun, and Peter Schröder. 2013. Digital Geometry Processing with Discrete Exterior Calculus. In *ACM SIGGRAPH 2013 courses* (Anaheim, California) *(SIGGRAPH '13)*. ACM, New York, NY, USA, 126 pages.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 1–12.

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

Herbert Edelsbrunner. 1999. *Simplicial Complexes.* Technical Report. https://people.eecs.berkeley.edu/~jrs/meshpapers/edels/L-06.pdf

Herbert Edelsbrunner and John Harer. 2008. *Computational Topology: An Introduction.*

Sharif Elcott and Peter Schröder. 2005. Building Your Own DEC at Home. In *ACM SIGGRAPH 2005 Courses (SIGGRAPH '05)*. Association for Computing Machinery, New York, NY, USA, 8–es. https://doi.org/10.1145/1198555.1198667 event-place: Los Angeles, California.

Andreas Fabri and Sylvain Pion. 2009. CGAL: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 538–539.

Shachar Fleishman, Iddo Drori, and Daniel Cohen-Or. 2003. Bilateral mesh denoising. In *ACM SIGGRAPH 2003 Papers*. 950–953.

Greg Friedman. 2012. Survey article: an elementary illustrated introduction to simplicial sets. *The Rocky Mountain Journal of Mathematics* (2012), 353–423.

Xiao-Ming Fu, Yang Liu, and Baining Guo. 2015. Computing locally injective mappings by advanced MIPS. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–12.

Brian Guenter. 2007. Efficient symbolic differentiation for graphics applications. In *ACM SIGGRAPH 2007 papers (SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 108–es. https://doi.org/10.1145/1275808.1276512

Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM transactions on graphics (TOG)* 4, 2 (1985), 74–123.

Marc Habermann, Lingjie Liu, Weipeng Xu, Michael Zollhoefer, Gerard Pons-Moll, and Christian Theobalt. 2021. Real-time deep dynamic characters. *ACM Transactions on Graphics (ToG)* 40, 4 (2021), 1–16.

Allen Hatcher. 2002. *Algebraic Topology.* Cambridge University Press.

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–12.

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Doug L James. 2020. Phong deformation: a better C 0 interpolant for embedded deformation. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 56–1.

Tomasz Kaczynski, Konstantin Michael Mischaikow, and Marian Mrozek. 2004. *Computational homology.* Number v. 157 in Applied mathematical sciences. Springer, New York.

Lutz Kettner. 1998. Designing a data structure for polyhedral surfaces. In *Proceedings of the fourteenth annual symposium on Computational geometry*. 146–154.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–21.

Michael Lange, Navjot Kukreja, Mathias Louboutin, Fabio Luporini, Felippe Vieira, Vincenzo Pandolfo, Paulius Velesko, Paulius Kazakas, and Gerard Gorman. 2016. Devito: Towards a generic Finite Difference DSL using symbolic Python. In *2016 6th workshop on python for high-performance and scientific computing (PyHPC)*. IEEE, 67–75.

Soeren Laue. 2022. On the Equivalence of Automatic and Symbolic Differentiation. http://arxiv.org/abs/1904.02990 arXiv:1904.02990 [cs].

Allan Leal. 2023. The autodiff library. https://autodiff.github.io.

Christian Lengauer, Sven Apel, Matthias Bolten, Shigeru Chiba, Ulrich Rüde, Jürgen Teich, Armin Größlinger, Frank Hannig, Harald Köstler, Lisa Claus, et al. 2020. Exastencils: advanced multigrid solver generation. In *Software for Exascale Computing-SPPEXA 2016-2019*. Springer International Publishing, 405–452.

Yong Li, Shoaib Kamil, Alec Jacobson, and Yotam Gingold. 2021. I Heart LA: Compilable Markdown for Linear Algebra. *ACM Transactions on Graphics (TOG)* 40, 6 (Dec. 2021).

Yong Li, Shoaib Kamil, Alec Jacobson, and Yotam Gingold. 2022. HeartDown: Document Processor for Executable Linear Algebra Papers. In *ACM SIGGRAPH Asia (Conference Papers)*. https://doi.org/10.1145/3550469.3555395

Pascal Lienhardt. 1994. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry & Applications* 4, 03 (1994), 275–324.

Ahmed H Mahmoud, Serban D Porumbescu, and John D Owens. 2021. RXMesh: a GPU mesh data structure. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16.

Martti Mäntylä. 1989. Advanced topics in solid modeling. In *Advances in Computer Graphics V*. Springer, 49–74.

Nelson Max. 1999. Weights for computing vertex normals from facet normals. *Journal of graphics tools* 4, 2 (1999), 1–6.

Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103.

Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H Barr. 2003. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and mathematics III*. Springer, 35–57.

Niloy J Mitra and An Nguyen. 2003. Estimating surface normals in noisy point cloud data. In *Proceedings of the nineteenth annual symposium on Computational geometry*. 322–328.

J. S. Mueller-Roemer, C. Altenhofen, and A. Stork. 2017. Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs. *Computer Graphics Forum* 36, 5 (Aug. 2017), 59–69. https://doi.org/10.1111/cgf.13245

D. E. Muller and F. P. Preparata. 1978. Finding the intersection of two convex polyhedra. *Theoretical Computer Science* 7, 2 (Jan. 1978), 217–236. https://doi.org/10.1016/0304-3975(78)90051-8

Michael Rabinovich, Roi Poranne, Daniele Panozzo, and Olga Sorkine-Hornung. 2017. Scalable Locally Injective Mappings. *ACM Trans. Graph.* 36, 2 (April 2017). https://doi.org/10.1145/2983621 Place: New York, NY, USA Publisher: Association for Computing Machinery.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. 2016. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)* 43, 3 (2016), 1–27.

Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. 2012. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 1116–1123.

Patrick Schmidt, Janis Born, David Bommes, Marcel Campen, and Leif Kobbelt. 2022. TinyAD: Automatic Differentiation in Geometry Processing Made Simple. In *Computer graphics forum*, Vol. 41. Wiley Online Library, 113–124.

John Schreiner, Arul Asirvatham, Emil Praun, and Hugues Hoppe. 2004. Inter-Surface Mapping. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 870–877. https://doi.org/10.1145/1015706.1015812 Place: New York, NY, USA Publisher: Association for Computing Machinery.

Daniel Shapero. 2023. An Ergonomic Approach to Topological Transformations of Unstructured Meshes. *International Meshing Roundtable (Research Note)* (2023).

Nicholas Sharp and Keenan Crane. 2020. A Laplacian for Nonmanifold Triangle Meshes. *Computer Graphics Forum (SGP)* 39, 5 (2020).

Nicholas Sharp, Keenan Crane, et al. 2019a. geometry-central. www.geometry-central.net.

Nicholas Sharp, Mark Gillespie, and Keenan Crane. 2021. Geometry processing with intrinsic triangulations. *SIGGRAPH'21: ACM SIGGRAPH 2021 Courses* (2021).

Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019b. Navigating intrinsic triangulations. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–16.

Daniel Sieger and Mario Botsch. 2019. The Polygon Mesh Processing Library. http://www.pmp-library.org.

Jason Smith and Scott Schaefer. 2015. Bijective parameterization with free boundaries. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–9.

Steven L Song, Weiqi Shi, and Michael Reed. 2020. Accurate face rig approximation with deep differential subspace reconstruction. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 34–1.

Oded Stein, Alec Jacobson, Max Wardetzky, and Eitan Grinspun. 2020. A smoothness energy without boundary distortion for curved surfaces. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–17.

Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. 2005. Robust Quasistatic Finite Elements and Flesh Simulation. In *Proceedings of the 2005 ACM*

SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '05). Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/1073368.1073394 event-place: Los Angeles, California.

Fangzhi Xu, Qika Lin, Jiawei Han, Tianzhe Zhao, Jun Liu, and Erik Cambria. 2023. Are large language models really good logical reasoners? a comprehensive evaluation from deductive, inductive and abductive views. *arXiv preprint arXiv:2306.09841* (2023).

Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 144–1.

Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. 2022. MeshTaichi: A Compiler for Efficient Mesh-Based Operations. *ACM Transactions on Graphics (TOG)* 41, 6 (2022), 1–17.

Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. 2017. A GPU-Adapted Structure for Unstructured Grids. *Computer Graphics Forum* 36, 2 (May 2017), 495–507. https://doi.org/10.1111/cgf.13144

Paul Zhang, Josh Vekhter, Edward Chien, David Bommes, Etienne Vouga, and Justin Solomon. 2020. Octahedral frames for feature-aligned cross fields. *ACM Transactions on Graphics (TOG)* 39, 3 (2020), 1–13.

A SOURCE CODE COMPARISONS

We implemented per-vertex mean curvature using I❤️MESH, Simit, Ebb, and MeshTaichi. Figure 9 displays the source code in each language. Driver code is highlighted in green. Stencil code is highlighted in yellow. Mesh expressions are highlighted in pink.

```cpp
using namespace simit;
int main(int argc, char **argv){
  simit::init("cpu", sizeof(double));
  Mesh mesh;
  mesh.load("path/to/mesh");
  Set points;
  Set faces(points, points, points);
  Set ds(points, points, faces, faces);
  simit::FieldRef<double,3> x = points.addField<double,3>("x");
  simit::FieldRef<double,3> fn = faces.addField<double,3>("fn");
  simit::FieldRef<double> c = points.addField<double>("curvature");
  std::vector<ElementRef> pr;
  for(auto vertex : mesh.v) {
    ElementRef point = points.add();
    pr.push_back(point);
    x.set(point, vertex);
  }
  typedef std::pair<std::tuple<int, int>, int> key_pair;
  std::map<std::tuple<int, int>, int> f_map;
  std::vector<ElementRef> fr;
  int index = 0;
  for(auto t : mesh.t) {
    ElementRef face = faces.add(pr[t[0]], pr[t[1]], pr[t[2]]);
    fr.push_back(face);
    for (int i = 0; i < 3; ++i) {
      f_map.insert(key_pair(std::make_tuple(t[i], t[(i+1)%3]), index));
      auto s = f_map.find(std::make_tuple(t[(i+1)%3], t[i]));
      if (s != f_map.end()) {
        ds.add(pr[t[i]], pr[t[(i+1)%3]], fr[index], fr[s->second]);
        ds.add(pr[t[(i+1)%3]], pr[t[i]], fr[s->second], fr[index]);
      }
    }
    index++;
  }
  Program program;
  program.loadFile("path/to/curvature.sim");
  Function cal = program.compile("compute_curvature");
  cal.bind("verts", &points);
  cal.bind("faces", &faces);
  cal.bind("diamonds",  &ds);
  cal.init();
  cal.runSafe();
  return 0;
}
```

C++ code

```
element Vert
  x  : vector[3](float);
  curvature  : float;
end

element Face
  fn  : vector[3](float);
end

element Diamond
end

extern verts:set{Vert};
extern faces : set{Face}(verts,verts,verts);
extern diamonds : set{Diamond}(verts, verts, faces, faces);

func face_normal(inout s: Face, inout p : (Vert*3))
  s.fn = cross(p(1).x - p(0).x, p(2).x - p(0).x);
end

func angle(inout d: Diamond, inout q:(v1:Vert,v2:Vert,f1:Face,f2:Face))
  var len = norm(q.v2.x - q.v1.x);
  var e = (q.v2.x - q.v1.x)/len;
  var t = atan2(dot(e, cross(q.f1.fn,q.f2.fn)), dot(q.f1.fn,q.f2.fn));
  q.v1.curvature = q.v1.curvature + len * t / 4.0;
end

export func compute_curvature()
  apply face_normal to faces;
  map angle to diamonds;
end
```

Simit code

(a) Simit

```python
ti.init(arch=getattr(ti, 'cpu'))
mesh = Patcher.load_mesh("/path/to/mesh", relations=['FV', 'VF', 'VV'])
mesh.verts.place({'x' : ti.math.vec3, 'curvature' : ti.f32})
mesh.verts.x.from_numpy(mesh.get_position_as_numpy())
@ti.kernel
def vertex_mean_curvature():
  ti.mesh_local(mesh.verts.x, mesh.verts.curvature)
  for v in mesh.verts:
    for neighbor in v.verts:
      n1 = ti.Vector([0.0, 0.0, 0.0])
      n2 = ti.Vector([0.0, 0.0, 0.0])
      for f in v.faces:
        if f.verts[0].id == v.id and f.verts[1].id == neighbor.id:
          n1 = (neighbor.x - v.x).cross(f.verts[2].x - v.x)
        elif f.verts[1].id == v.id and f.verts[2].id == neighbor.id:
          n1 = (neighbor.x - v.x).cross(f.verts[0].x - v.x)
        elif f.verts[2].id == v.id and f.verts[0].id == neighbor.id:
          n1 = (neighbor.x - v.x).cross(f.verts[1].x - v.x)
        elif f.verts[0].id == neighbor.id and f.verts[1].id == v.id:
          n2 = (v.x - neighbor.x).cross(f.verts[2].x - neighbor.x)
        elif f.verts[1].id == neighbor.id and f.verts[2].id == v.id:
          n2 = (v.x - neighbor.x).cross(f.verts[0].x - neighbor.x)
        elif f.verts[2].id == neighbor.id and f.verts[0].id == v.id:
          n2 = (v.x - neighbor.x).cross(f.verts[1].x - neighbor.x)
    len = (neighbor.x - v.x).norm()
    e = (neighbor.x - v.x)/len
    alpha = ti.math.atan2(e.dot(n1.cross(n2)), n1.dot(n2))
    v.curvature += alpha * len / 4
vertex_mean_curvature()
```

(c) MeshTaichi

```lua
-- adapted from ebb/domains/trimesh.t
... copied lines omitted ...
local EdgeFlapMesh = {}
function EdgeFlapMesh:build_edges(vs)
  ... copied lines omitted ...
  for i = 1, mesh:nTris() do
    neighbors[vs[i][1]+1][vs[i][2]+1] = i-1
    neighbors[vs[i][2]+1][vs[i][3]+1] = i-1
    neighbors[vs[i][3]+1][vs[i][1]+1] = i-1
  end
  ... copied lines omitted ...
  local e_face = {}
  local e_indices = {}
  local e_dual = {}
  for k = 1, mesh:nVerts() do e_indices[k] = {} end
  for i = 1, mesh:nVerts() do
    for j,_ in pairs(neighbors[i]) do
      ... copied lines omitted ...
      table.insert(e_face, neighbors[i][j])
      e_indices[i][j] = n_edges
      n_edges = n_edges + 1
    end
  end
  for i = 1, mesh:nVerts() do
    for j,_ in pairs(neighbors[i]) do
      table.insert(e_dual, e_indices[j][i])
    end
  end
  ... copied lines omitted ...
  mesh.edges:NewField('face', mesh.triangles):Load(e_face)
  mesh.edges:NewField('dual', mesh.edges):Load(e_dual)
  ... copied lines omitted ...
end

-- adapted from ebb/domains/ioOff.t
function EdgeFlapMesh.LoadEdgeFlapMesh(path)
  ... copied lines omitted ...
  return EdgeFlapMesh.LoadFromLists(position_data_array,tri_data_array)
end
```

EdgeFlapMesh.t

```lua
import "ebb"
local L = require "ebblib"
local ioOff = require "ebb.domains.ioOff"
local PN    = require "ebb.lib.pathname"
local EdgeFlapMesh = require 'EdgeFlapMesh'
local mesh  = EdgeFlapMesh.LoadEdgeFlapMesh("path/to/mesh")
mesh.vertices:NewField('mean_curvature', L.double)
mesh.triangles:NewField('n', L.vec3d)
local ebb compute_normal ( t : mesh.triangles )
  t.n = L.cross(t.v[1].pos - t.v[0].pos, t.v[2].pos - t.v[0].pos)
end
local ebb vertex_mean_curvature ( v : mesh.vertices )
  for e in v.edges do
    var neighbor = e.head
    var f1 = e.face
    var f2 = e.dual.face
    var len = L.length(neighbor.pos - v.pos)
    var e = (neighbor.pos - v.pos)/len
    var cur = atan2(L.dot(e, L.cross(f1.n, f2.n)), L.dot(f1.n, f2.n))
    v.mean_curvature += cur * len / 4
  end
end
mesh.triangles:foreach(compute_normal)
mesh.vertices:foreach(vertex_mean_curvature)
```

curvature.t

(b) Ebb

```
atan2 from trigonometry
ElementSets from MeshConnectivity
VertexOneRing, OrientedVertices, EdgeIndex from Neighborhoods(M)
M : FaceMesh
x_i ∈ ℝ^3
V, E, F = ElementSets( M )

N(f) = ((x_j- x_i)×(x_k-x_i))/(‖(x_j-x_i)×(x_k-x_i)‖) where f ∈ F,
  i,j,k = OrientedVertices(f)

l_i,j = ‖x_j - x_i‖ where i,j ∈ V

ϕ_i,j = atan2(e⋅(`n_1`×`n_2`), `n_1`⋅`n_2`) where i, j ∈ V,
  e = (x_j-x_i)/‖x_j-x_i‖,
  `f_1`, `f_2` = OrientedOppositeFaces(i, j),
  `n_1` = N(`f_1`),
  `n_2` = N(`f_2`)

H(i) = 1/4 ∑_(j ∈ VertexOneRing(i)) l_ij ϕ_ij where i ∈ V

`∂⁰`, `∂¹` = BoundaryMatrices(M)
OrientedOppositeFaces(i, j) = tuple(f1_1, f2_1) where i,j ∈ V,
  e = EdgeIndex(i, j), fs = Faces(e),
  f1 = { f for f ∈ fs if `∂¹`_e,f `∂⁰`_i,e = -1 },
  f2 = fs - f1
```

I❤MESH code

```cpp
int main(int argc, const char * argv[]) {
    Eigen::MatrixXd meshV;
    Eigen::MatrixXi meshF;
    igl::readOBJ("path/to/mesh", meshV, meshF);
    TriangleMesh triangle_mesh;
    triangle_mesh.initialize(meshF);
    FaceMesh face_mesh(triangle_mesh.bm1, triangle_mesh.bm2);
    std::vector<Eigen::Matrix<double, 3, 1>> P;
    for (int i = 0; i < meshV.rows(); ++i) {
        P.push_back(meshV.row(i).transpose());
    }
    heartlib ihla(face_mesh, P);
    std::vector<double> mean_curvature(meshV.rows());
    for (int i = 0; i < meshV.rows(); ++i) {
        mean_curvature[i] = ihla.H(i);
    }
}
```

C++ code

(d) I❤MESH

Fig. 9. Source code comparisons of vertex mean curvature. See Section 5 for a discussion.