iskra: A System for Inverse Geometry Processing

ANA DODIK, MIT CSAIL, USA
AHMED H. MAHMOUD, MIT CSAIL, USA
JUSTIN SOLOMON, MIT CSAIL, USA

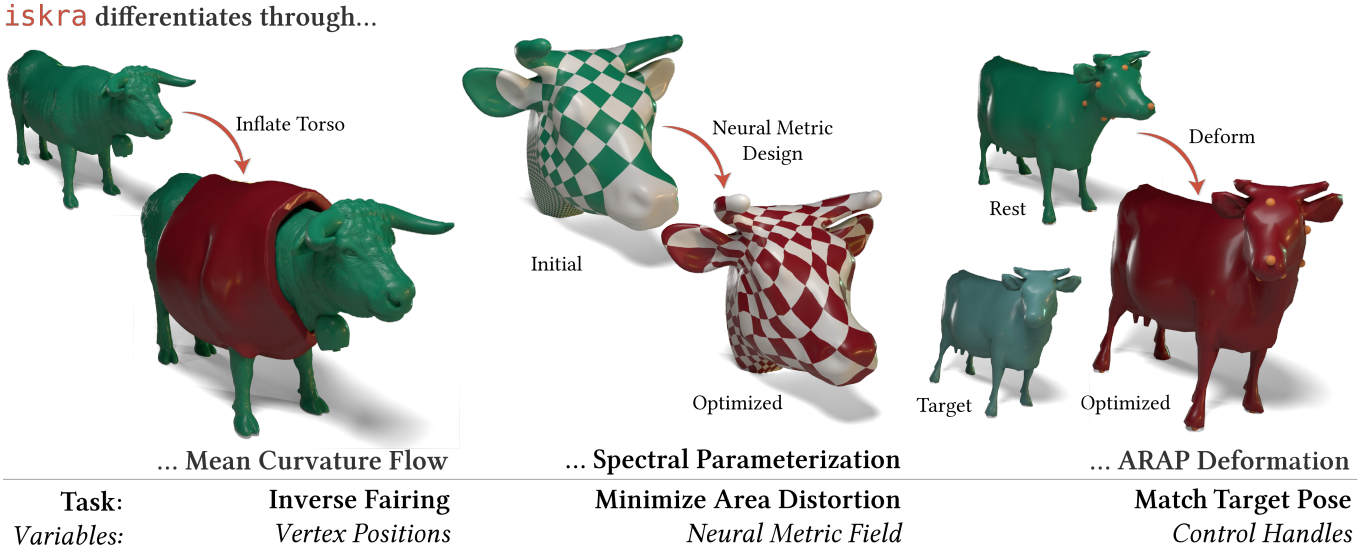


Fig. 1. Our system embeds geometry processing algorithms within larger differentiable pipelines. Users can optimize vertex positions to run a fairing flow backwards, a mesh's intrinsic metric to reduce area distortion during parameterization, or control handles to match a deformed mesh to a target pose.

We propose a system for differentiating through solutions to geometry processing problems. Our system differentiates a broad class of geometric algorithms, exploiting existing fast problem-specific schemes common to geometry processing, including local-global and ADMM solvers. It is compatible with machine learning frameworks, opening doors to new classes of inverse geometry processing applications. We marry the scatter-gather approach to mesh processing with tensor-based workflows and rely on the adjoint method applied to user-specified imperative code to generate an efficient backward pass behind the scenes. We demonstrate our approach by differentiating through mean curvature flow, spectral conformal parameterization, geodesic distance computation, and as-rigid-as-possible deformation, examining usability and performance on these applications. Our system allows practitioners to differentiate through existing geometry processing algorithms without needing to reformulate them, resulting in low implementation effort, fast runtimes, and lower memory requirements than differentiable optimization tools not tailored to geometry processing.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**; **Graphics systems and interfaces**; *Neural networks*; Texturing; Animation; • **Mathematics of computing** → **Automatic differentiation**; **Mathematical software**; *Mathematical optimization.*

ACM Reference Format:
Ana Dodik, Ahmed H. Mahmoud, and Justin Solomon. 2026. iskra: A System for Inverse Geometry Processing. *ACM Trans. Graph.* 45, 4, Article 112 (July 2026), 20 pages. https://doi.org/10.1145/3811371

Authors' Contact Information: Ana Dodik, anadodik@mit.edu, MIT CSAIL, Cambridge, MA, USA; Ahmed H. Mahmoud, ahdhn@mit.edu, MIT CSAIL, Cambridge, MA, USA; Justin Solomon, jsolomon@mit.edu, MIT CSAIL, Cambridge, MA, USA.

ACM 1557-7368/2026/7-ART112
https://doi.org/10.1145/3811371

1 Introduction

Much of geometry processing consists of optimization problems over mesh data that are easily expressed in mesh-specific differentiable systems [Herholz et al. 2024, 2022; Mara et al. 2021]. For example, these systems address the classical problem of deforming a mesh by minimizing a distortion measure while constraining a set of control vertices. However, what if we wish to use the deformed solution as part of a larger program, e.g., finding the positions of the controls such that the deformed mesh best approximates some target? We call this an *inverse problem*, consisting of an *inner optimization* that solves for the deformed mesh and an *outer optimization* which requires the derivatives of deformed mesh with respect to the control vertices (Figure 9).

Inverse geometry processing problems introduce two coupled challenges that make existing systems unsuitable for this case. The first involves operating on mesh data, which requires system support for sparse and irregular operations and workloads. The second has to do with the diversity of optimization problems and problem-specific solvers found in geometry processing. Besides standard

sparse linear systems, geometry processing software often solves highly non-linear or non-convex problems using specialized methods that leverage structures specific to the problem.

Given these requirements, existing tools for inverse problems imply trade-offs. The first class of tools offers simple *declarative* interfaces for specifying the forward problem in some canonical form, which they automatically optimize using a pre-established solver. These tools leverage the narrow canonical form of the inner problem to provide derivatives for the outer problem [Agrawal et al. 2019; Lai et al. 2023; Pineda et al. 2022]. This paradigm offers a concise interface for the class of problems these system target at the cost of constraining the problem and solver class. Going back to our initial example, solving the seminal as-rigid-as-possible (ARAP) deformation problem [Igarashi et al. 2005; Sorkine and Alexa 2007] with quadratic programming systems [Pan et al. 2024] or generic optimization libraries [Pineda et al. 2022] foregoes the popular alternating algorithm typically used to optimize ARAP problems Sorkine and Alexa [2007] and may not be possible unless the (nonconvex) ARAP objective can be massaged into an acceptable canonical form. Thus, this class of systems prevents the user from relying on specialized solvers that exploit the problem's structure.

Alternatively, *imperative tools* offer a lower-level interface, allowing users to specify optimization loops themselves. Here, one might be tempted to simply rely on an off-the-shelf framework, e.g., PyTorch or Jax, to implement an inner optimization loop, which the framework will "unroll" during the backward pass to provide derivatives for the outer problem. Alas, this process stores intermediate data with each iteration, quickly blowing up the memory for many problems. Moreover, it prevents us from relying on highly-optimized black-box solvers for the forward pass that are not implemented in the framework, e.g., ARPACK for eigenproblems [Lehoucq et al. 1998]. Be they declarative or imperative, support for sparse computation on meshes is rare in systems for inverse problems, as many are meant for domains like machine learning or image processing where sparsity is not a key consideration [Chen et al. 2018; Lai et al. 2023; Li et al. 2018].

In bringing support for sparse computation on meshes to inverse problem systems, we must choose a programming abstraction—see Section 2 for an overview of existing options. Representing mesh data in a dense memory layout and exposing it to the user via the tensor abstraction would come with major benefits: it would enable rapid prototyping and make the boundaries to numerical and machine learning ecosystems porous, e.g. via the `DLPack` interface. Such interoperability would be crucial for inverse geometry processing specifically, as it would allow the outer optimization loop to be anything ranging from inverse rendering of the mesh, to neural network dictated mesh variables or optimization objectives. At the same time, relying on this abstraction is not entirely straight-forward and would necessitate designing high-level user abstractions for sparse computation patterns common to meshes (§5.1).

Taking stock of constraints, we distill the following design goals:

- **Versatility.** Our system must support differentiating through the broad spectrum of geometry processing tasks and offer flexibility in how they are solved and differentiated.
- **Usability.** It must integrate into a clean interface for differentiable programming, geometry processing, and inverse problems.
- **Interoperability.** Geometric problems must compose seamlessly with numerical computing and machine learning ecosystems.
- **Efficiency.** Our system should be sparsity-aware and integrate with both CPU and GPU workflows.

To this end, we introduce `iskra`, an *imperative* tensor-based system for inverse geometry processing. It offers a high-level interface for differentiable geometry processing, staying interoperable with the broader numerical computing ecosystem and offering sparse CPU or GPU execution. Our system introduces a simple high-level interface that allows the user to differentiate through common geometry processing solvers. Practically, this means that `iskra` allows geometric problems to be embedded in machine learning pipelines (§7.2). Our system supports differentiating through common geometry processing tasks and solvers including sparse linear systems, local-global solvers, sparse eigenproblems (§5.2), as well as more esoteric methods that exploit problem-specific structures (§7.3, §7.4).

In particular, there is a rich body of work on specialized solvers [Bourquat et al. 2022; Manson and Schaefer 2011; Solomon et al. 2015, 2014], reflecting the collective knowledge and expertise of the past 30 years of the geometry processing literature. The imperative nature of our system allows users to differentiate through geometry problems without reformulating seminal approaches. The user must only specify one iteration of a fixed-point or root-finding problem, which our system then differentiate through using the *adjoint method* (§5.2). Thus, to avoid locking the user into one specific approach, we offer flexibility in separating forward and backward pass implementation as well as the specifics of each.

We provide a clean interface that integrates geometric primitives with the tensor-based abstraction. Since it is built on top of PyTorch, our system has first-class interoperability with machine learning ecosystems and follows standard practices that enable CPU/GPU integration. To make the sparse geometry workflows compatible with a dense tensor framework, our system exposes a set of mesh-specific *gather-scatter* primitives and offers custom data types and operators for differentiable sparse linear algebra. It also offers an extensive list of routines needed to build geometry processing algorithms (§5.1).

We demonstrate the versatility of our system on a set of problems, each of which requires differentiating through a different kind of solver. These include a geometric flow [Desbrun et al. 1999] with a sparse linear system (§7.1), mesh parameterization [Mullen et al. 2008] with a sparse eigensystem (§7.2), mesh deformation with a highly nonlinear and non-convex problem [Sorkine and Alexa 2007] solved via a custom local-global approach (§7.3), and geodesic computation [Edelstein et al. 2023] using a custom ADMM-based [Boyd et al. 2010] optimizer (§7.4). Our implementation is open-source and available at https://github.com/anadodik/iskra.

In this paper, we make the following contributions:

- a system for efficient inverse geometry processing that integrates cleanly with differentiable and numerical computing ecosystems;
- a robust suite of differentiable geometry operators and optimizers with flexible way of specifying the backward pass;
- a set of diverse example applications demonstrating practicality, performance, and ease of use; and
- a quantitative evaluation and benchmark of the system, its various components, and user trade-offs.

Scope. Our system design reflects certain trade-offs. Our goal is to balance efficiency while prioritizing interoperability, generality and usability, not to aim for peak performance or memory utilization. Moreover, we limit our scope to exclude dynamic changes in sparsity or mesh connectivity during solver iterations, as well as physics simulation-specific considerations such as collision detection or contact handling. We leave for future work to integrate with `iskra` low-level approaches to geometry, dynamic topology changes, and physics-specific considerations

2 Related Work

Prior work spans two largely orthogonal research directions. The first is systems for differentiable geometry processing aiming to integrate automatic differentiation into geometry processing with focus on performance, flexibility, and interoperability. The second focuses on differentiating through optimization problems more broadly by developing general frameworks for backpropagation through solvers. We review these lines of work separately as they address complementary aspects of the challenges our system solves. We also review works on differentiating through ODE and PDE solvers.

2.1 Geometry Processing Differentiation Systems

Contemporary geometry processing software for triangle and tetrahedral meshes falls into one of two categories. The first, exemplified by the seminal `libigl` library, offers high-level interfaces based on the matrix abstraction, focusing on practicality and ease of use and offering a rich geometry processing toolbox [Botsch et al. 2002; Jacobson et al. 2021; Jacobson and Panozzo 2017; Project 2025; Sellán et al. 2023; Vaxman et al. 2025]. This set of tools often leaves out differentiability and GPU acceleration necessary for machine learning and inverse problem interoperability.

The second camp focuses on speed, often via GPU acceleration. These GPU-based systems rely on low-level abstractions that require the user to specify every operation over a local neighborhood of a mesh element (e.g., for each vertex, perform an operation on its neighboring triangles) [Herholz et al. 2024; Mahmoud et al. 2021, 2025; Yu et al. 2022]. Such low-level abstractions lead to inflexibility and incompatibility with tensor abstractions used within the machine learning ecosystem [Mahmoud et al. 2021; Yu et al. 2022].

Recently, several attempts have been proposed to introduce differentiability as a first-class primitive for triangle and tetrahedral mesh processing. TinyAD [Schmidt et al. 2022] made the observation that forward-mode automatic differentiation is faster for geometry processing workload. Such a system can be easily integrated with `libigl` and OpenMesh [Botsch et al. 2002] to make them differentiable. However, TinyAD lacks GPU acceleration and does not integrate with modern machine learning frameworks. Other systems with similar limitations include Fernández-Fernández et al. [2025]; Herholz et al. [2024, 2022]; Mara et al. [2021], which offer efficient differentiation but provide limited support for interoperability with ML ecosystems. Crucially, since the goal of these systems is to differentiate in service of solving geometry processing problems, they only allow for differentiability over a individual operations over mesh-based data, which is insufficient if we wish to *invert* a geometry processing problem, i.e., differentiate through solutions to geometric optimizations (see §3).

More broadly, existing solutions for differentiable geometry processing fall into two categories: (1) general-purpose automatic differentiation systems that are flexible but inefficient for sparse geometric operators (e.g., PyTorch, JAX) and (2) efficient but narrowly-scoped systems that lack GPU acceleration, ML interoperability, or support for differentiable optimization. In contrast to both extremes, our work focuses on usability, interoperability, and abstraction flexibility, rather than focusing solely on raw differentiation speed or mesh processing throughput.

While systematic approaches to *differentiable* geometry processing are well researched, to our knowledge there are no systems specifically focusing on *inverse* geometry processing, i.e., the systematic differentiation through geometric optimizations. Nonetheless, the geometry processing literature has already made use of differentiable optimizations in an ad-hoc manner. Existing work in geometry that differentiates through linear solvers can be found in Wang et al. [2023a]; Wang and Solomon [2021], through eigenproblems in Cosmo et al. [2019]; Smirnov and Solomon [2021], and through geodesic distances in Li et al. [2024]; Verninas et al. [2026].

2.2 Systems for Differentiation Through Optimization

Many systems allow differentiation through optimization routines, i.e., backpropagating through solutions of optimization problems. For example, Theseus [Pineda et al. 2022] is an application-agnostic framework for differentiable nonlinear least squares. Like our system, it strives for interoperability within the machine learning ecosystem. However, because it is not specialized for geometry processing, its performance degrades when differentiating through complex operations such as sparse linear solvers or eigensolvers.

CVXPYLayers [Agrawal et al. 2019] is a domain-specific language that differentiates through disciplined convex programs by converting the problem specification into certain canonical forms. From there, CVXPYLayers differentiates through these problems without explicitly backpropagating through the operations of the solver. Instead, they use implicit differentiation formulas specific to canonicalized versions of certain convex problems. While this facilitates backpropagation, the user must define problems as disciplined convex programs, which might not be always possible. Even when possible, their system does not scale, even to moderate-sized meshes (§7.4).

Similarly to CVXPYLayers, ∇-Prox [Lai et al. 2023] introduces a domain-specific language and compiler for differentiable proximal algorithms, targeted mainly for large-scale image optimization. ∇-Prox reduces a high-level problem specification to a canonical proximal form and compiles it into a differentiable solver that supports unrolling, implicit differentiation, and hybrid learning schedules. Their design enables efficient differentiation through iterative proximal algorithms and allows for rapid prototyping. However, the system is tailored to dense, regular image operations and does not address sparse, irregular operations arising in geometry processing, specifically on triangle and tetrahedral meshes.

BPQP [Pan et al. 2024] is a differentiable convex optimization layer that reformulates the backward pass through a convex program as

a simplified quadratic program by exploiting the structure of the KKT conditions. BPQP computes gradients via a first-order QP. This reformulation reduces the cost of implicit differentiation compared to unrolling the forward solver or differentiating through a full KKT system. Prior to BPQP, OptNet [Amos and Kolter 2017] differentiates through quadratic programs by implicitly differentiating the full KKT system associated with a primal–dual interior-point solve, making OptNet's backward pass relatively straightforward.

2.3 Differentiation Through ODEs and PDEs

A complementary line of work to ours enables differentiation through continuous time/space of dynamical systems. These systems backpropagate through the solution of ordinary and partial differential equations, implementing the simulator as a layer that can be differentiated implicitly. For example, Neural ODE [Chen et al. 2018] backpropagates through black-box ODE solvers using the adjoint method, avoiding backpropagation through steps of the solver to enable end-to-end training of models defined by continuous dynamics. For more information, refer to Kolter et al. [2020].

Subsequent systems such as ChainQueen [Hu et al. 2019] and DiffTaichi [Hu et al. 2020] extended this idea to differentiable physics simulation by relying on compilers to support backpropagating through PDE-based solvers for fluids, soft bodies, and rigid-body dynamics—notably with no support for meshes. Systems like DiffPD [Du et al. 2021] and that of Huang et al. [2024] provide differentiable systems specially tailored to soft-body simulation. Much like the differentiable optimization systems, these methods offer a fixed set of pre-chosen soft-body solvers [Du et al. 2021; Huang et al. 2024]. For information about recent advances on differentiable physics simulators, see Coros et al. [2021]; Newbury et al. [2024].

These systems enable differentiating through time-dependent soft-body solvers on *volumetric* domains with collision handling, and primarily target optimizing for physical parameters. In contrast, our focus is on inverse geometry processing over static triangle and tetrahedral meshes. Geometry processing problems do not fit neatly into a unified declarative form like elasticity does [Du et al. 2021; Hu et al. 2019; Huang et al. 2024]. Even if these systems were modified to support codimension-1 meshes, expressing surface ARAP as a general nonlinear optimization problem requires reformulating the objective function, much like for Theseus [Pineda et al. 2022] (§7.3). Moreover, two of our examples—eigenproblems (§7.2) and geodesics on curved domains (§7.4)—are not handled by prior systems. Lastly, it is common that the systems in this class do not permit specifying custom forward solvers [Du et al. 2021; Hu et al. 2019; Huang et al. 2024]; in contrast, ours tackles general inverse problems, including linear systems, eigenproblems, fixed points, and proximal iterations. As a result, the abstractions and optimizations developed for differentiable simulation do not directly address the sparsity in geometry processing, nor the broader class of geometry processing problems that our system supports.

3 Background

Inverse geometry processing requires differentiating through a broad variety of optimization problems. This section briefly summarizes the mathematical backbone of our system, namely implicit differentiation and its implementation via the adjoint method. For more information on the topic, see the textbook by Strang [2007].

3.1 Adjoint Method

Suppose $g(\,\cdot\,)$ denotes a solver for a geometry processing problem with parameters x; we will use $x \in \mathbb{R}^m$ to denote the input to $g(\,\cdot\,)$ and $y \in \mathbb{R}^n$ to denote the output $y = g(x)$. As an example, we may be interested in finding the sensitivites of deformed vertex coordinates (y) to changes in control handles (x) in as-rigid-as-possible deformation. The adjoint method allows us to calculate these derivatives under mild conditions, given that we can write down a differentiable function $f : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^n$ implicitly relating inputs x of g to outputs y:

$$f(x; y) = 0, \tag{1}$$

where we assume that root-finding on f *uniquely* determines y from x. We use this notation without loss of generality, assuming matrix and tensor valued x and y are vectorized. For notational convenience, we allow for multiple variables on either side of the semicolon in f, implicitly assuming they are concatenated to form the two vectors x and y; we always differentiate the symbols after the semicolon with respect to those before it.

Using gradient-based optimization of objectives that involve g requires finding the Jacobian $\partial y/\partial x \in \mathbb{R}^{n\times m}$. Rather than working directly with g, which is typically a complicated map, the implicit function theorem states that if the Jacobian $\partial f/\partial y \in \mathbb{R}^{n\times n}$ is invertible at an x, then y can be locally treated as a function of x, denoted $y(x)$. By the chain rule, taking a total derivative of $f(x; y(x)) = 0$ with respect to x yields

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} = 0,$$

which leads to the desired derivative:

$$\frac{\partial y}{\partial x} = -\left(\frac{\partial f}{\partial y}\right)^{-1}\frac{\partial f}{\partial x}. \tag{2}$$

This is a solver-agnostic formula, meaning the implementation of g used to compute y from x (e.g., direct solver, iterative method, nonlinear optimization) does not affect the correctness of Equation (2)—what matters are the partial derivatives of f.

Assuming y is fed into a scalar-valued energy to be minimized $\ell : \mathbb{R}^n \to \mathbb{R}$, we can use the adjoint method as an efficient computational procedure for obtaining the derivatives of ℓ with respect to x. Namely, it states that the derivative

$$\frac{\partial \ell}{\partial x} = \underbrace{-\frac{\partial \ell}{\partial y}\left[\frac{\partial f}{\partial y}\right]^{-1}}_{\partial \ell/\partial f}\frac{\partial f}{\partial x}, \tag{3}$$

can be computed by computing $\partial \ell/\partial f$ first, before computing the remaining terms. This is efficient because $\partial \ell/\partial y \in \mathbb{R}^{1\times n}$ always has fewer or equal elements than $\partial f/\partial x \in \mathbb{R}^{n\times m}$, and corresponds to standard backward-mode differentiation.

This approach illustrates a broad pattern where the backward pass through a solver is defined mathematically by the implicit relation, not by the implementation details of the forward algorithm. This viewpoint enables efficient, sparsity-preserving differentiation for a wide class of geometry processing tasks. Indeed, the adjoint

method has been applied to a variety of inverse tasks all throughout computer graphics and geometry processing [Huang et al. 2024], albeit its use usually involves tedious and manual computation of the necessary derivative formulas. Our system automates this computation for geometric applications.

4 Programming Model

In `iskra`, we consider optimization problems of the form

$$\min_{\theta \in \mathbb{R}^p} \ell\big(g(\varphi(\theta))\big), \tag{4}$$

where φ maps *outer* optimization variables $\theta \in \mathbb{R}^p$ to quantities defined over a simplicial mesh; g denotes a *geometric solver* that maps these quantities to a solution y implicitly defined by a system of equations on the mesh (e.g., a linear solve, a fixed-point iteration); and ℓ is an objective defined on the output of g. Both φ and ℓ are composed from arbitrary differentiable operations, including not only geometry processing operations, but also neural network components. As a concrete example, inputs to g could include vertex positions, boundary conditions, and finite element operators and the corresponding output could be the geodesic distance function.

From a user's perspective, expressing an inverse geometry problem in `iskra` follows the straightforward three-step pattern outlined in the next three sections.

4.1 Specify Parameters and Geometry

The user first specifies the mesh and the *outer* optimization variables θ. In code, these are ordinary differentiable PyTorch tensors and can be anything from mesh data to neural network parameters. The user provides the map $\theta \mapsto \varphi(\theta)$, where $\varphi(\theta)$ is, e.g., a geometric quantity defined on mesh elements or a sparse matrix. Constructing geometric quantities is straightforward as `iskra` provides both a low-level scatter-gather interface for working with mesh-structured data (§5.1), as well as high-level primitives, such as element-wise quantities discrete differential operators (§5.2).

In `iskra`, mesh topology consists of sets of faces each of which is assigned a unique index and a canonical orientation. We notate the set of d-dimensional faces, i.e., those containing $(d+1)$ vertices, as $\mathbb{F}^d$. These are $\mathbb{C}, \mathbb{T}, \mathbb{E}, \mathbb{V} \subset \mathbb{N}$ for tetrahedra, triangles, edges, and vertices. Their relationships are defined via tensors such as $\mathrm{EV} \in \mathbb{V}^{|\mathbb{E}|\times 2}$ or $\mathrm{VV} \in \mathbb{V}^{2|\mathbb{E}|\times 2}$; for more details see §6.1. A mesh's geometry is most often defined in terms of vertex positions $\boldsymbol{v} \in \mathbb{R}^{|\mathbb{V}|\times 3}$. More generally, each mesh element can be equipped with an attribute, geometry is only a special case.

Continuing with our introductory example of ARAP deformation [Igarashi et al. 2005; Sorkine and Alexa 2007], we assume a triangular mesh and define a set of control handles $\mathbb{H} \subset \mathbb{V}$ with positions $\boldsymbol{h} \in \mathbb{R}^{|\mathbb{H}|\times 3}$. As a reminder, our goal is to optimize the handle positions such that the deformed mesh best approximates some target mesh with vertices $\boldsymbol{t}$. See Figure 2 for a visual illustration.

In `iskra`, loading meshes and obtaining $\boldsymbol{t}$ and $\boldsymbol{v}$ can be carried out as follows:

```
mesh, _ = Mesh.from_path(mesh_path, dtype=dtype, device=device)
verts = mesh.geom.vertices  # v

target_mesh, _ = Mesh.from_path(mesh_path, dtype=dtype, device=device)
target_verts = target_mesh.geom.vertices  # t
```

We load the handles and initialize their positions to the corresponding vertex positions in the undeformed mesh. Next we define them as optimization variables θ following standard PyTorch practice, by marking them as differentiable and passing them to our optimizer of choice, in this case (stochastic) gradient descent (SGD):

```
handle_idx = load_handles(handles_path)  # ℍ
handles = verts[handle_idx]  # h
handles.requires_grad_(True)  # differentiable
optimizer = SGD([handles], lr=lr)  # make h into θ
```

In this simple example, φ is the identity map.

4.2 Define Implicit Geometric Layer

The next step for the user is defining the solver $y \leftarrow g(x)$. To support our goal of versatility, our system is intentionally general and unopinionated at this level. Once such a core is in place, more specialized convenience utilities can be layered on top. At the lowest interface level, users can define a differentiable g by specifying

(1) a method for computing g during the forward pass,
(2) an implicit relationship f between the inputs and outputs of g, used to automatically create g's backward pass.

More specifically, for a given g, the user specifies a corresponding differentiable implicit relation $f(x; y) = 0$ as described in §3, giving g the name *implicit geometric layer*.

This separation of concerns allows g to be an arbitrary function, including calls to external low-level libraries, such as the optimized sparse linear algebra routines like cuDSS [NVIDIA Corporation 2025] and Cholmod [Chen et al. 2008]. The user implements the implicit relation f using a combination of standard PyTorch routines and `iskra`'s geometry processing features, making it so the system can use automatic differentiation to assemble an adjoint method backward pass. At the lowest level, `iskra` exposes a minimal interface in which users directly specify the forward solver and its corresponding implicit relation. `iskra` also supports other common patterns for defining g, including linear solves, spectral problems, and fixed-point iterations that arise in geometric processing.

For ARAP specifically, in addition to our optimization variables, we need a set of auxiliary variables, namely the cotan weights $\boldsymbol{c} \in \mathbb{R}^{|\mathbb{E}|}$ which we can map onto each oriented vertex pair, thus $\hat{\boldsymbol{c}} \in \mathbb{R}^{2|\mathbb{E}|}$, and assemble into the Laplacian matrix $L \in \mathbb{R}^{|\mathbb{V}|\times|\mathbb{V}|}$:

```
faces = mesh.topo.faces  # 𝔽
vert_vert = vertex_adjacency(faces)  # VV

weights = cotan_weights(verts, faces, clamp_min=1e-5)  # c
vert_vert_weights = edge_to_vertex_adjacency(weights)  # ĉ
lap = laplacian_from_weights(weights, faces)  # L
```

In ARAP, it is common to pre-factor the Laplacian for efficiency. All of our ARAP results were generated with this optimization, but we choose to omit it here for didactic purposes.

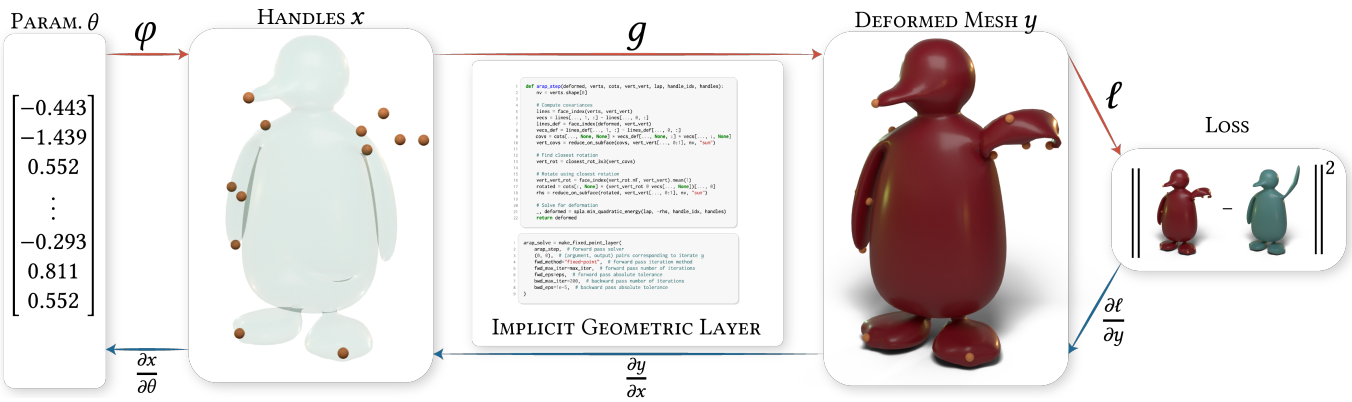


Fig. 2. A typical pipeline in `iskra`: the user maps a parameter vector θ to mesh quantities x, in this case, control handles. Next the user defines an *implicit geometric layer*, which solves a forward (inner) geometric problem, such as ARAP deformation; `iskra` automatically provides gradients in the process. Finally, the user uses the solution to the inner problem in an outer problem with loss function ℓ.

Next, we define f. In `iskra`, we can use ARAP's iterative optimization loop to define the implicit relationship:

```
def arap_step(deformed, verts, cots, vert_vert, lap, handle_idx, handles):
    nv = verts.shape[0]

    # Compute covariances
    lines = face_index(verts, vert_vert)
    vecs = lines[..., 1, :] - lines[..., 0, :]
    lines_def = face_index(deformed, vert_vert)
    vecs_def = lines_def[..., 1, :] - lines_def[..., 0, :]
    covs = cots[..., None, None] * vecs_def[..., None, :] * vecs[..., :, None]
    vert_covs = reduce_on_subface(covs, vert_vert[..., 0:1], nv, "sum")

    # Find closest rotation
    vert_rot = closest_rot_3x3(vert_covs)

    # Rotate using closest rotation
    vert_vert_rot = face_index(vert_rot.mT, vert_vert).mean(1)
    rotated = cots[:, None] * (vert_vert_rot @ vecs[..., None])[..., 0]
    rhs = reduce_on_subface(rotated, vert_vert[..., 0:1], nv, "sum")

    # Solve for deformation
    _, deformed = spla.min_quadratic_energy(lap, -rhs, handle_idx, handles)
    return deformed
```

We can convert this into an implicit relationship between the relevant optimization variables with a simple wrapper:

```
def f(handles, deformed):
    return deformed - arap_step(deformed, ..., handles)
```

Finally, we are ready to assemble g. For now, we assume access to an ARAP solver `arap_solver(handles)`, be it one we created using `arap_step`, or one of the many accelerated methods that optimize objectives like ARAP [Bourquat et al. 2022; Manson and Schaefer 2011; Peng et al. 2018]. In forming g, we must specify which arguments of f serve as x and which as y, i.e., which arguments should be treated as functions of which other arguments. In this case, we wish to find the derivative of `deformed`, in position 1, w.r.t. `handles`, in position 0. While our system allows the user to specify which iterative solver to use for the linear system in the adjoint formula, we rely here on the default GMRES [Saad and Schultz 1986] solver:

```
arap_solve = make_adjoint_layer(
    arap_solver,  # forward solver
    f,  # implicit relationship
    in_args=0,  # which arguments correspond to x
    out_args=1,   # which arguments correspond to y
    bwd_max_iter=200,  # number of adjoint solver steps
    bwd_rel_tol=1e-5,  # adjoint solver relative tolerance
)
```

This is intentionally a very general formulation meant to cover the broadest possible range of inverse problems one might wish to solve—as long as there is a known implicit relationship between the inputs and outputs of the solver and $\frac{\partial f}{\partial y}$ is invertible at a given x, our system can assemble a backwards pass.

However, we can offer greater usability in special cases by leveraging the fact that g and f are oftentimes more closely related. For example, g is often best implemented as a fixed-point iteration $y \leftarrow \widetilde{f}(x; y)$, e.g., a contraction map converging to y (see §7.3, §7.4). Our system also offers a higher-level interface for this specific case, automatically generating the forward pass from $\widetilde{f}$, and generating the backward pass through the fact that this fixed-point problem is isomorphic to $f(x; y) = \widetilde{f}(x; y) - y = 0$. For our running example, `iskra` allows us to create a fixed-point solver from `arap_step` by specifying which arguments and outputs correspond to y:

```
arap_solve = make_fixed_point_layer(
    arap_step,  # forward pass solver
    (0, 0),  # (argument, output) pairs corresponding to iterate y
    fwd_method="fixed-point",  # forward pass iteration method
    fwd_max_iter=max_iter,  # number of fixed-point iterations
    fwd_abs_tol=abs_tol,  # forward pass absolute tolerance
    bwd_max_iter=200,  # number of adjoint solver steps
    bwd_rel_tol=1e-5,  # adjoint solver relative tolerance
)
```

Regardless of exact interface, the user writes their code by composing `iskra`'s sparse mesh-specific operators and `iskra` attaches to each forward pass solver a corresponding implicit backward rule, thus avoiding naively differentiating through internal iterations.

4.3 Compose a Geometry Objective

Having defined θ, φ, and our implicit geometric layer g, the user specifies a scalar objective ℓ as a composition of differentiable operations. Typical choices can include standard geometry energies, e.g., symmetric Dirichlet, various smoothness energies, geodesic distance penalties. Our system is flexible enough to allow the user to incorporate more complex machine learning components, such as neural network-based energies.

In our specific scenario, we wish to match the deformed vertices produced by g to the target vertices $\boldsymbol{t}$,

$$\ell(\boldsymbol{t}, \theta) = \frac{1}{|\mathbb{V}|}\|\boldsymbol{t} - g(\varphi(\theta))\|_2^2, \tag{5}$$

where we take the L_2 norm instead of the mass-matrix norm to simplify exposition. From the user's point of view, this example objective consists only of ordinary code written in PyTorch where the only difference is that some tensors result from implicit geometric layers:

```
optimizer.zero_grad()
deformed = arap_solve(
    verts, vert_vert_weights, vert_vert, lap, handle_idx, handles
)
loss = ((deformed - target_verts) ** 2).sum(-1).mean()
loss.backward()
```

For a more complex version of the problem involving real-world motion capture data [Mahmood et al. 2019], refer to §7.3 and Fig. 11.

Having already chosen the outer optimizer (e.g., SGD, Adam), the user can perform gradient-based optimization over θ by iteratively computing the loss and stepping with the optimizer. The outer optimization loop is therefore written entirely in standard automatic differentiation style, while the complexity of differentiating through solvers is handled internally by `iskra`.

In our example, we note that the `arap_step` function hides within it two nested adjoint problems, namely the sparse linear solve within `spla.min_quadratic_energy` and the `polar3x3` matrix decomposition. This is permissible in `iskra`, as many geometry pipelines solve smaller optimizations as part of a larger problem.

4.4 Capabilities

As we have seen, `iskra` uses implicit differentiation of $f(x; y) = 0$ to obtain gradients $\partial\ell/\partial\theta$ and integrates them to the outer optimizer, regardless of the optimization type. We emphasize the breadth of the set of problems covered by our formulation. Specifically, some of the problems that fall under this umbrella are linear systems (§7.1), eigenproblems (§7.2), and problems solvable via local-global solvers (§7.3) or ADMM iterations (§7.4). Moreover, our system also allows for flexibility in how we solve g's backward pass. This has significant impact on efficiency compared to other approaches, both in terms of memory requirements and speed (see §7).

To amplify usability and efficiency of our system, we include special functionality for two pervasive operations that have structure beyond the generic case. In particular, we provide bespoke layers that implement two strongly-structured forms of f that appear in nearly all geometry processing algorithms:

- **Linear implicit layers**, where the forward pass solves

$$A\,\boldsymbol{y} = \boldsymbol{b}, \tag{6}$$

with the accompanying implicit relationship:

$$\boldsymbol{f}(A, \boldsymbol{b}; \boldsymbol{y}) = A\,\boldsymbol{y} - \boldsymbol{b} = 0. \tag{7}$$

See §5.2 for details.

- **Spectral implicit layers**, where the forward pass solves

$$\begin{aligned} AU &= UB\,\mathrm{diag}(\boldsymbol{\lambda}), \\ U^\top BU &= I, \end{aligned} \tag{8}$$

where the columns of $U \in \mathbb{R}^{n\times k}$ and entries in $\boldsymbol{\lambda} \in \mathbb{R}^k$ represent eigenvectors and eigenvalues of the generalized eigenvalue problem defined by $A \in \mathbb{R}^{n\times n}$, and $B \in \mathbb{R}^{n\times n}$. The eigenvector problem is special not only because of its ubiquity in geometry processing, but also because it admits many ways of differentiating through its solutions. Therefore, our system allows the user to switch between these options. We include a thorough study of the different trade-offs (§7.2). See §5.2 for examples.

5 `iskra` System Design

`iskra` is organized around two tightly coupled modules: a *differentiable geometry module* and an *implicit-function module*. These intertwined modules work together to enable inverting geometry processing problems. The *differentiable geometry module* (§5.1) is responsible for representing meshes, mesh relations, local geometric operators, and offers additional support for irregular data and geometry-specific quantities like (dual) quaternions. The *implicit-function module* (§5.2) is responsible for differentiating through solvers using the adjoint method, as discussed previously (§3, §4).

The design of the geometry module serves as a foundation for the implicit-function module. Supporting a broad range of existing geometry algorithms requires us to give full control to the user; typically, such low-level control would imply a higher implementation burden, standing in an apparent contradiction with our other stated design goal of usability. This tension informs the design of our geometry module. It aims to be as compact and efficient as possible as a way of alleviating the user effort for expressing existing algorithms within the implicit-function module. Thus, the representations of mesh relations, operators, and data movement are chosen to preserve computational sparsity central to geometric workloads during implicit differentiation, all while integrating naturally with the broader tensor-based ecosystem. Crucially, neither is sufficient by itself: their interaction enables `iskra` to achieve the stated design goals of versatility, usability, interoperability, and efficiency.

5.1 Geometry Module

The geometry module of `iskra` is responsible for representing meshes, mesh-based data, and constructing local geometric operators. Rather than introducing a specialized domain-specific language or execution model [Mahmoud et al. 2021; Yu et al. 2022], which sacrifice flexibility and interoperability for raw speed, `iskra`'s tensor-centric design necessitates a different approach. While sharing the tensor abstraction with other systems [Jacobson et al. 2021; Jacobson and Panozzo 2017; Sellán et al. 2023], `iskra` builds upon them by systematizing operations on mesh data represented this

way. Specifically, inspired by Yu et al. [2022], `iskra` expresses geometry processing as a combination of *gather* and *scatter-reduce* operations defined on mesh topology. A key realization in our design is that the scatter-gather paradigm maps quite naturally to the tensor abstraction, even when it comes to mesh attribute data.

`iskra`'s design thinks of mesh faces in a hierarchy. For example, in a one-dimensional polyline mesh, line-segments (1D faces) are at the highest-level and contain within them vertices (0D faces). Therefore, we say that a vertex is a *subface* of the line-segment. We use this idea to describe the relation between edges (1D faces) and triangles (2D faces). Our system's topology interface allows us to extract all topology relationships commonly found in geometry processing, including extracting lower levels of the hierarchy from the higher ones, taking into account orientations when needed.

Mesh-based operations are expressed using explicit *gather* and *scatter-reduce* transformations to move across this hierarchy. Conceptually, these operations follow two complementary patterns:

- *Gather* operations read attributes defined on a subface of a mesh element and collect them all on the face.
- *Scatter-reduce* operations distribute and accumulate attributes defined on a face to its subfaces.

Thus, gather operations move data up the hierarchy, whereas scatter-reduce operations move data down the hierarchy. In combination, these patterns can express common local geometry processing tasks, such as assembling discrete differential operators, computing per-element energies, or accumulating forces. These operations are element-agnostic, i.e., the same abstraction applies uniformly to vertices, edges, triangles, or tetrahedra. In `iskra`, they are implemented as tensor operations, allowing the same code to run on CPU or GPU and to participate in automatic differentiation.

A simple example demonstrating this approach is computing normals on a 1D mesh embedded in 2D. First, using the edge-to-vertex relationship EV, we gather all of the 2D vertex positions into a 2×2 tensor on each edge. Next, we compute the per-edge normal vectors. Lastly, we scatter an edge's normal vector onto the vertices belonging to that edge before finally normalizing. We can express this operation in `iskra` as:

```
# Gather vertex positions onto edges using EV:
lines = face_index(verts, edge_vert)

# Compute per-edge normal:
vecs = lines[..., 1, :] - lines[..., 0, :]
orth = torch.broadcast_to(torch.tensor([1.0, -1.0]), vecs.shape)
face_normals = vecs[..., (1, 0)] * orth_vector

# Scatter-reduce from edges to vertices using EV:
normals = reduce_on_subface(face_normals, edge_vert, verts.shape[0], "sum")
normals = normalize(normals, dim=-1)
```

This simple example demonstrates the basic programming model of our system, albeit more complicated operations are possible. For example, by default, the system assumes all attributes are vector-valued and decides on the scattering strategy based on that. In some situations, ambiguities arise, e.g., a $|\mathbb{T}| \times 3 \times 3$ tensor could either represent a 3×3 matrix per face, or a 3-vector per corner of a face. Our system allows the user to specify the dimensionality of the element attribute and decides the scattering strategy based on this.

In addition, due a lack of fully-fledged support for sparsity in PyTorch, we provide the user with a custom `SparseTensor`, implemented using the *coordinate* sparse format, offering standard utilities common to tensor based frameworks such as indexing, slicing, reshaping, concatenation, as well as elementwise and binary operations.

Discussion. By representing mesh relations through explicit sparse gather and scatter operations, `iskra` simultaneously achieves three design goals. First, it preserves the computational sparsity inherent in discrete geometry operations. Second, it aligns mesh-based computation with the dataflow model expected by modern automatic differentiation systems. Third, it provides a uniform interface that supports a wide range of geometric operators without introducing a separate execution or programming model. Our design inherits trade-offs common to tensor-based frameworks, namely it favors simplicity, composability, and interoperability in lieu of peak mesh traversal performance or compressed memory storage. In practice, we expect our system to integrate with other resource heavy workloads, including, e.g., machine learning and inverse rendering. Due to this, we do not expect our system to be the bottleneck and leave for future work to explore low-level acceleration strategies.

5.2 Implicit-Function Module

Having described the foundation that allows our system to remain efficient even when faced with sparse data flows of geometry processing, we now turn to the implicit function module. The difficulty lies in differentiating through the solver $y \leftarrow g(x)$ from Equation (4). Towards this goal, `iskra`'s *implicit-function module* provides a uniform mechanism for evaluating and differentiating through a broad set of solvers g. This module is responsible for both the forward evaluation of g and the backward propagation of its gradients.

Geometry processing problems give rise to a wide variety of problems and solvers, e.g., linear systems, generalized eigenproblems, alternating local-global schemes, or problem-specific ADMM instances. Rather than designing the system around a single class of optimization problems, the implicit-function module is formulated as generally as possible to support this diversity. At the lowest level, if g solves a known optimization objective the supplied implicit relationship f is used to construct the gradient of the solution with respect to g's inputs. As we will see, in many cases a higher-level interface is possible, e.g., it can be sufficient to specify one step of a fixed-point iteration scheme. Succinctly, our design objective for this module is to accommodate the need for wide diversity of problem-specific solvers, while offering a sufficiently pliable, yet ergonomic experience to geometry processing experts.

To achieve this level of generality, `iskra` adopts an *imperative* formulation for its implicit layers. As covered previously, to generate the backward pass, the user supplies an f that they themselves wrote using functionality from `iskra`'s geometry module. The system does not require the user to specify the objective of g in a declarative form, nor to restrict problems to a specific optimization template. This choice contrasts with declarative approaches, such as CVXPYLayers [Agrawal et al. 2019], ∇-Prox [Lai et al. 2023], or Theseus [Pineda et al. 2022], which demand problems be within the narrow class of either convex, proximal, or non-linear least squares

to derive gradients automatically. Using them involves reformulating optimization problems into the specific declerative form they dictate, which can come with its own challenges (§7).

Such approaches are not well suited to geometry processing workloads which routinely involve highly nonlinear or non-convex energies (§7.3, §7.2), or involve problem with known exploitable structures (§7.4). Taken together, the scope of problems in geometry processing does not admit a single canonical declarative representation nor does it allow for a one-size-fits-all set of solvers. By adopting an imperative formulation, `iskra` allows each solver to be implemented using the most appropriate numerical method while enabling implicit differentiation via automatically derived adjoints.

For each implicit-function layer, the forward pass computes y by solving the implicit equation using an arbitrary, possibly black-box solver g while the backward pass differentiates through its solution using implicit differentiation. This separation allows users to rely on fast specialized solvers even if their implementations are not differentiable. At a high level, the this module consists of:

- an interface for specifying and invoking geometry-aware solvers to compute y from x;
- an interface that allows the user to imperatively specify the implicit relationship $f(x; y) = 0$,
- a set of problem class-specific routines that exploit the sparsity and structures common to that class.

While our applications in §7 follow the structure in Equation (4), in practice our system allows users to stack and nest implicit geometric layers and interleave them with other differentiable workloads.

The core insight enabling our design is that, unlike the forward pass, which can be solved in arbitrary ways depending on the problem, the backward pass is guaranteed to be a sparse linear system. After all, the Jacobian $\partial f/\partial y$ that we must invert is a linear approximation of a function at a given point. This means that, regardless of the complexity of g or f, the backward pass is effectively a (sparse) linear solver (see §3.1). In all but the simplest cases, the Jacobian at any given x is likely an enormous matrix; even with sparsity accounted, using a direct solver would likely be prohibitive for large problems. Instead, we rely on an iterative method, allowing the user to use a customized solver if needed; for more information, see §6.2.

Since our loss ℓ is a scalar-valued function, we can leverage the adjoint method. Practically, this means that given a function $f(x; y)$, our system first constructs two functions that evaluate *vector-Jacobian products*; namely, one that right-multiplies its inputs with $\partial f/\partial y$, and another that right-multiplies with $\partial f/\partial x$. Then we apply the adjoint method: first we compute $\partial \ell/\partial f$ by using an iterative solver that repeatedly applies $\partial f/\partial y^\top$ to $\partial \ell/\partial y^\top$, before incorporating $\partial f/\partial x$ to compute the full expression. This approach to differentiation is used regardless of whether the user relies on the fixed-point interface or the more general low-level one.

The fixed-point interface contributes to the usability of our system, allowing users to solve very broad classes of problems. For example, in addition to the local-global solver in our running ARAP example (§4), our system is demonstrably able to differentiate through problems solvable using ADMM iterations (§7.4). As long as a fixed-point iteration exists, our system can differentiate through it. Specific to this interface is that we allow the user to incorporate additional information into the generation of the forward pass, such as selecting a function's output to serve as the energy objective or specifying tolerances (see §6.2).

Two problem classes stand out in how common they are in geometry processing and in our ability to provide specialized efficient solutions for computing their derivatives, namely sparse linear systems and sparse eigensolvers. Besides being ubiquitous in geometry, linear systems are also mathematically special. They are the only problem class where the Jacobian $\partial f/\partial y$ of the backward pass happens to be the same as the forward problem, opening doors to further optimizations. Eigensystems are special in that they come with a variety of different derivative formulae, each with a different trade-off. Below we describe iskra's special functionality for these two ubiquitous operations.

Differentiating linear solvers. Implementing the adjoint method for a linear system amounts to solving a system with the same matrix as in the forward problem. Differentiating the implicit relationship $\boldsymbol{f}(A, \boldsymbol{b}; \boldsymbol{y}) = A\boldsymbol{y} - \boldsymbol{b} = \boldsymbol{0}$ with respect to its inputs, we obtain:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}} = A, \qquad \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{b}} = -I, \qquad \frac{\partial f_i}{\partial A_{jk}} = \delta_{ij} y_k,$$

where δ is the Kronecker delta. Incorporating this into the adjoint method from Equation (3), we see that solving for $\partial \ell/\partial y$ involves inverting A. Unlike the general adjoint problem from prior sections, users could exploit knowledge about their specific linear system to accelerate the backward pass—provided the right system affordances. For example, it should be possible to exploit known matrix structures such as positive semidefiniteness, to reuse precomutation or to rely on a solver that exploits hardware-specific characteristics. Given how common sparse linear systems are in geometry, staying faithful to our design goals of versatility and efficiency means offering a bespoke solution for this particular case. In doing this, it is imperative that the system offer good defaults, but also allow the user provide their own sparse solver. Therefore, we differentiate through the solutions of linear systems by manually implementing the adjoint method, relying on a user-supplied non-differentiable solver when specified. For example:

```
factors = factor_matrix(mat)
def custom_solver(b):
    return solve_with_factors(factors, b)  # not differentiable

x = linear_solve(mat, b, solver_fn=custom_solver)  # differentiable
```

We demonstrate how this additional flexibility leads to efficiency gains during both the forward and backward passes (§7.1).

Differentiating eigenvectors and eigenvalues. A similarly foundational module of geometry processing is the ability to solve sparse (generalized) eigenvalue problems. With the same rationale of expanding our system's versatility and efficiency, our system provides additional affordances for eigenproblems. However, in contrast to linear systems, differentiating through solutions to eigenproblems—specifically through eigenvectors—is less straightforward. Deriving a closed-form formula for differentiating through a single eigenvector can be done by writing down the equation:

$$\begin{aligned} A\boldsymbol{u}_i &= \lambda_i B\boldsymbol{u}_i, \\ \text{s.t. } \boldsymbol{u}_i^\top B\boldsymbol{u}_i &= 1, \end{aligned} \tag{9}$$

and assembling the corresponding implicit relationship,

$$\boldsymbol{f}(A, B; \boldsymbol{u}_i, \lambda_i) = \begin{bmatrix} A\boldsymbol{u}_i - \lambda_i B\boldsymbol{u}_i \\ \frac{1}{2}(1 - \boldsymbol{u}_i^\top B\boldsymbol{u}_i) \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ 0 \end{bmatrix}, \tag{10}$$

where we treat $\boldsymbol{u}_i$ and λ_i as dependent on $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times n}$. Manually computing the required partial derivatives reveals the following expressions:

$$\begin{aligned} \frac{\partial \boldsymbol{f}}{\partial(\boldsymbol{u}_i; \lambda_i)} &= \begin{bmatrix} A - \lambda_i B & -B\boldsymbol{u}_i \\ -\boldsymbol{u}_i^\top B & 0 \end{bmatrix} \\ \frac{\partial f_j}{\partial A_{kl}} &= \delta_{jk}(u_i)_l \\ \frac{\partial f_j}{\partial B_{kl}} &= \begin{cases} -\lambda_i \delta_{jk}(u_i)_l & j \le n \\ -\frac{1}{2}(u_i)_k(u_i)_l & j = n+1 \end{cases} \end{aligned} \tag{11}$$

On one hand, the system corresponding to $\partial f/\partial y$ is sparse, symmetric, and, assuming no repeated eigenvalues, invertible. On the other, the matrix itself contains both the eigenvalue and eigenvector, meaning that we cannot reuse the same factorization in cases where we need more than one eigenvector. Thus, if more than one eigenvalue-eigenvector pair is required, our system must loop over each pair and factor and solve a different linear system.

The second approach offered by our system is that of Smirnov and Solomon [2021]. Focusing on derivatives of an eigenvector w.r.t. A, further manipulation tells us that they can be computed as follows:

$$\frac{\partial \boldsymbol{u}_i}{\partial A_{kl}} = \sum_{m \neq i} \frac{(u_m)_k (u_i)_l}{\lambda_i - \lambda_m} \boldsymbol{u}_m \tag{12}$$

While this compact formula can be implemented using only dense matrix multiplies, it comes at the cost of needing access to all n eigenvalues, something that is most often prohibitively expensive. To this extent Smirnov and Solomon [2021] introduce an approximation by truncating small eigevalues: if the user requested only the top-k eigenvalues, the method treats the remaining ones as zero.

The remaining approach relies on our system's fixed-point capabilities instead of on hand-derived formulae through a fixed-point equation based on QR-iterations:

$$f(A, B, U) = Q((A - \sigma I)^{-1} U) - U = 0, \tag{13}$$

where Q orthonormalizes its inputs w.r.t. B and chooses a canonical sign for each column, and σ is the *shift-invert* parameter.

Ultimately, the best approach depends on the use-case. In addition to offering such a broad spectrum of approaches, we include a thorough investigation into the performance characteristics of each, including speed, memory consumption, and relationship to the size of A and the number of requested eigenvectors k in §7.2.

6 Implementation Details

Our framework offers numerous utilities related to mesh processing, including various topological operations, point-to-simplex projections and distances, barycentric coordinate computation, normals and (hyper-)volume computation, quaternion and dual quaternion support, and simplex quality metrics. We also offer fully fledged discrete exterior calculus [Crane et al. 2013; Desbrun et al. 2006; Hirani 2003] and finite element toolboxes, representing the discrete operators as sparse matrices. The entire system is differentiable as it is implemented end-to-end in PyTorch.

In this section we focus on the low-level implementation details, namely those of the scatter-gather framework, our implementation of the adjoint method, as well as specialized optimizers required to make differentiable geometry processing integrate well with other optimization pipelines.

6.1 Gather/Scatter

Our scatter-gather framework is suited specifically to meshes, requiring a purpose-built topology toolbox. Given a simplicial mesh, our system can construct the entire face hierarchy upon user request. Each level of the hierarchy consists of three dense tensors connecting it to the level below it. As explained in §4 and 5.1, we assign each d-dimensional face a unique index in $\mathbb{F}^d$ and connect it to $(d-1)$-dimensional faces. Bookkeeping for this is via a set of 3 dense tensors per level of the hierarchy: *face-vertex*, *face-subface*, and *subface-orientation* tensors. For example, for a set of tetrahedra $\mathbb{C}$, we compute the *face-vertex* tensor $\mathrm{C}_V \in \mathbb{V}^{|\mathbb{C}|\times 4}$, the *face-subface* tensor as $\mathrm{CT} \in \mathbb{T}^{|\mathbb{C}|\times 4}$ where $\mathrm{CT}_{ij} \in \mathbb{T}$ is the j^{th} triangle in the i^{th} tetrahedron, and the *subface-orientations* as $\{-1, 1\}^{|\mathrm{CT}|\times 4}$, defined relative to the canonical orientations of triangles in T_V. Moreoever, the triangle CT_{ij} is guaranteed to be opposite vertex $\mathrm{C}_{V\,ij}$: note that this means that, e.g., EV and E_V are not the same tensors. In addition, we provide the connections C_E and vertex adjacencies $\mathrm{VV} \in \mathbb{V}^{2|\mathbb{E}|\times 2}$ for convenience. In addition to supporting open and non-manifold meshes, this representation could also be used to represent meshes consisting of mixed elements.

Gathering data from subfaces onto faces is done via the `face_index` function which accepts a tensor of k-dimensional values, a *face-subface* connection and returns a tensor of the same shape as the face-subface tensor with additional dimensions for the gathered data. Scattering from faces to subfaces is done via the `reduce_on_subface` function, accepting a set of values defined on the faces and a face-subface connection. We implement both using PyTorch's built-in scatter-gather functions.

6.2 Adjoint Method Implementation

As discussed in Section 3.1, our system constructs vector-Jacobian products from f using PyTorch's standard automatic differentiation capabilities(i.e., functionality from `torch.autograd`). Given a function f and the `out_args` parameter, we construct a function that computes vector-Jacobian products with $\partial f/\partial y$. We pass this function to GMRES, which repeatedly applies it to solve for $\partial t/\partial y$. Then we use the `in_args` parameter to construct the second vector-Jacobian product. It is up to the user to ensure that f uniquely determines y in terms of x and that $\partial f/\partial y$ is invertible at x. Since we are dealing with an intentionally broad formulation, we assume no structure on the Jacobian $\partial f/\partial y$, leaving us to rely on the very general GMRES [Saad and Schultz 1986] linear solver for the adjoint equations. While we have not used these features for our experiments, we allow the user to pass in a custom function instead of our GMRES implementation, technically allowing for preconditioning or carefully chosen initializers.

We provide the ability to specify absolute and relative tolerances to GMRES, as well as the maximum number of iterations. If `max_iter` is reached and the tolerances are unmet, the system prints a warning. Setting `verbose=True` exposes debugging information, including absolute and relative solver errors. Alternatively, the user can pass `strict=True` as an argument to GMRES to raise an error in this scenario. See §8 for discussion of GMRES stability. We did not aim to optimize our implementation of GMRES. In particular, the memory required by GMRES scales linearly with the number of iterations performed. We leave optimizing memory usage to future work.

In our fixed-point implementation, we provide additional utilities. The user can specify absolute and relative tolerances for the fixed-point solver. By default, these are compared to the difference between consecutive iterates; alternatively, the user can designate one of the iteration function's outputs as the optimization objective. More information can be exposed via `verbose=True`, and if the user prefers certainty, they can set `max_iter=None`, which enforces the tolerances be fully satisfied. We use fixed-point tolerances in §7.4.

We offer two sparse linear solvers to the user by default: CUDSS [NVIDIA Corporation 2025] and CHOLMOD [Chen et al. 2008]. Our default choice is to rely on CHOLMOD on the CPU and CUDSS on the GPU. Both solvers allow reusing parts of the factorization process, a common pattern in geometry research. For the eigensolver, we rely on ARPACK [Lehoucq et al. 1998] via SciPy [Virtanen et al. 2020] on the CPU and custom power-iteration code on the GPU.

Lastly, we include two specialized optimizers to the user that are tailored to geometry applications, a simple line-search procedure, and a so-called Sobolev H^1 preconditioner that allows us to use gradient-based optimization on vertex data [Karátson and Lóczi 2005; Neuberger 1985; Osher et al. 2022]. We use these optimizers in Sections 7.1 and 7.4.

7 Applications

In this section, we demonstrate how our system achieves its stated goals by implementing a number of inverse geometry processing problems and comparing them with alternative approaches. We demonstrate versatility by showing its application to four different geometry processing problems, each with a different set of requirements. We illustrate usability and interoperability by comparing our code to the code needed to solve these problems in declarative systems. Moreover, our system is interoperable with machine learning pipelines, as evidenced through its integration with a neural field [Xie et al. 2022] in §7.2. Lastly, for each problem, we include a thorough timing comparison with an alternative system compatible with that problem class.

Our experiments aim to demonstrate creative applications enabled by our framework, in addition to validation and benchmarking. They show how our system enables efficient implementation of unexplored classes of inverse geometric algorithms. For existing practical applications of inverse geometry processing, see Section 2.1.

First, we demonstrate how to invert a simple fairing problem [Desbrun et al. 1999] (§7.1). Doing so repeatedly within an optimization loop necessitates fast solving of sparse linear systems; as we will see, the additional flexibility of our linear system interface enables the user to exploiting problem-specific structures to extract greater performance (Figure 5). Next, we show how our system enables differentiating through the solution of a parameterization algorithm formulated as an eigenproblem [Mullen et al. 2008], offering a thorough comparison of various formulations. We then demonstrate our system's performance on the nonlinear, non-convex ARAP problem [Sorkine and Alexa 2007] from §4, including an example of how `iskra` could be used to fit a deformation of a registered mesh to real motion capture data (Fig. 11). Finally, we compare our system's performance on the quadratic programming *regularized geodesic distance problem* (RGD) [Edelstein et al. 2023]. ARAP and RGD both come with bespoke solvers proposed by the geometry processing community, namely ARAP's local-global solver and RGD's custom ADMM scheme, each of which outperforms general-purpose optimization systems for its problem class. As we will see, using these schemes via our fixed-point interface serves the dual purpose of performance benefits and allowing practitioners to differentiate through these algorithms without needing to reformulate them.

7.1 Inverse Mean Curvature Flow

Our system allows us to differentiate through the fairing problem proposed by Desbrun et al. [1999]. If applying mean curvature flow to a mesh thins it out, then the inverse problem would recover an inflated version of the mesh. Differentiating through mean curvature flow is a highly ill-posed problem and using it to inflate a mesh requires additional regularization, in our case, in the form of Sobolev H^1 preconditioning (§6.2). Here, we aim to demonstrate a species of mesh stylization following, e.g., *Cubic Stylization* [Liu and Jacobson 2019] or *Homunculus Warping* [Reinert et al. 2012]. The problem's simplicity makes for a perfect test of the various methods for differentiating through linear solvers.

Stating the problem formally, θ corresponds to vertex positions $\boldsymbol{v} \in \mathbb{R}^{|\mathbb{V}|\times 3}$, φ is identity, and $\boldsymbol{t} \in \mathbb{R}^{|\mathbb{V}|\times 3}$ are target vertex positions. Given a mass matrix as M, integrated Laplacian as L, and timestep as τ, we define $\boldsymbol{v}' \leftarrow g(\boldsymbol{v})$ as the result of applying mean curvature fairing to $\boldsymbol{v}$, i.e., solving $(M - \tau L)\boldsymbol{v}' = M\boldsymbol{v}$. To inflate a mesh, we initialize $\boldsymbol{t}$ and $\boldsymbol{v}$ to the vertex positions of the input mesh; thus, applying mean curvature flow to an optimized $\boldsymbol{v}$ should result in the original vertex positions. Our implicit relationship f corresponding to this problem is

$$\boldsymbol{f}(\boldsymbol{v}; \boldsymbol{v}') = (M(\boldsymbol{v}) - \tau L(\boldsymbol{v}))\boldsymbol{v}' - M(\boldsymbol{v})\boldsymbol{v} = \boldsymbol{0}, \tag{14}$$

where $L(\boldsymbol{v})$ and $M(\boldsymbol{v})$ denote the cotangent Laplacian and diagonal mass matrix, which are computed from the vertex positions $\boldsymbol{v}$. Our *outer* optimization problem is

$$\arg\min_{\boldsymbol{v}} \ \frac{1}{2}\|\boldsymbol{t} - g(\boldsymbol{v})\|_M^2, \tag{15}$$

where $\|\boldsymbol{w}\|_M^2 = \mathrm{Tr}(\boldsymbol{w}^\top M \boldsymbol{w})$ represents the discrete inner product under the M-metric for some $\boldsymbol{w} \in \mathbb{R}^{|\mathbb{V}|\times 3}$. This problem is (nearly) ill-posed in general, giving some flexibility in how we design the optimization procedure. Specifically, for a finite number of steps, we can influence the solution by smoothing out the gradients of $\boldsymbol{v}$ using a Sobolev preconditioner as defined in §6.2.

This problem demonstrates several features of our system. First, because $\boldsymbol{v}$ changes during optimization, we must recompute the Laplacian matrix every iteration. A naive implementation of the

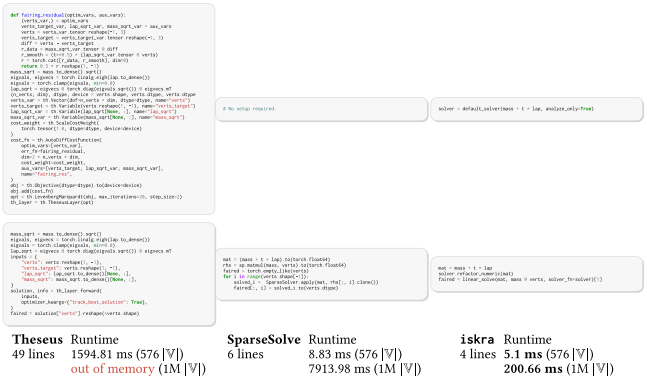
Fig. 3. Mesh inflation via inverse mean curvature flow. The Sappho mesh is inflated by treating it as the solution to the mean curvature flow at some time t and optimizing for the initial condition that led to that solution. Larger t results in more inflation.

adjoint rule for linear systems would have us refactor the matrix from scratch every step, a potentially costly operation. However, having domain knowledge of the problem, we could exploit the fact that the sparsity patterns of M and L remain constant throughout by reusing the *symbolic* factorization—i.e., the one that depends on the sparsity pattern alone—if only the system we were using allowed us to do so. Our system is versatile enough for this purpose, allowing the user to specify a custom linear solver to be used during the forward and backward passes, and manually recompute the *numeric* part of the factorization each iteration, while accounting for L's dependence on $\boldsymbol{v}$. The `iskra` code required for this purpose remains compact, as shown in Figure 4.

We offer two points of comparison for this problem. The first is Theseus [Pineda et al. 2022], a non-linear least squares *declarative* library. Our inner problem can be easily reformulated as a non-linear least squares and should therefore be solvable by Theseus. The second is SparseSolve [Jacobson 2025b], an imperative library meant exclusively for differentiating through sparse linear systems via the CHOLMOD solver [Chen et al. 2008]. Both natively integrate with PyTorch and support CPU and GPU execution.

Figure 4 visually compares the code required to solve this problem for each of the methods, as well as the execution times for two meshes. Figure 5 compares the execution speed on a range of meshes as a function of vertex count. As shown, our system is significantly faster and more ergonomic than Theseus on this problem—implementing it in `iskra` requires less code and is two orders of magnitude faster on average. While SparseSolve is comparable to our system in sheer number of lines of code required, and both systems rely on CHOLMOD for CPU execution in this test, `iskra` is able to exploit problem-specific strategies to further accelerate execution, resulting in an order of magnitude speed up. Here, `iskra` is able to seamlessly reuse the factorization in the forward mode and seamlessly switch to the specialized cuDSS solver [NVIDIA Corporation 2025] for GPU execution. Our point in comparing performance characteristics of the different solvers is *not* that our system uses a faster solver by default, but that it gives the user flexibility to choose the solution best suited for their problem.

Even though our method is in scope of what Theseus can do, we see that even for such a simple problem, declarative methods can result in a significantly higher implementation effort. Their system asserts a specific least-squares declarative form for the problem, and, while our problem mathematically fits into this formulation, integrating it with Theseus requires reformulating it: what should amount to a simple sparse linear solve becomes a pipeline involving multiple matrix square roots.



Theseus	Runtime	**SparseSolve**	Runtime	**iskra**	Runtime
49 lines	1594.81 ms (576 $\lvert\mathbb{V}\rvert$)	6 lines	8.83 ms (576 $\lvert\mathbb{V}\rvert$)	4 lines	**5.1 ms** (576 $\lvert\mathbb{V}\rvert$)
	out of memory (1M $\lvert\mathbb{V}\rvert$)		7913.98 ms (1M $\lvert\mathbb{V}\rvert$)		**200.66 ms** (1M $\lvert\mathbb{V}\rvert$)

Fig. 4. Code comparison for inverse fairing. Top row shows the setup needed for each framework, and the bottom compares executing one step of fairing.

Because they rely on a pre-determined set of solvers, Theseus and SparseSolve cannot exploit specialized solution strategies, such as, e.g., reusing the symbolic part of the factorization. Instead, we are left to rely on one of the pre-selected solvers like Levenberg-Marquardt for Theseus or CHOLMOD with pre-chosen settings for SparseSolve, significantly limiting their performance ceilings. Efficiency issues are far worse for Theseus as it is not designed with sparse geometry operators in mind. Attempting to use a sparse matrix in its problem definition makes the software crash, leaving us to resort to making all operators dense, which in turn negatively impacts memory and speed. In fact, we were unable to use Theseus for meshes larger than 600 vertices. From this example, we

can conclude that due to their rigid implementions of the adjoint rule, previous systems have limited versatility for the problem of inverse fairing. In contrast, our system is designed with flexibility and sparsity in mind, which in turn converts into greater efficiency.

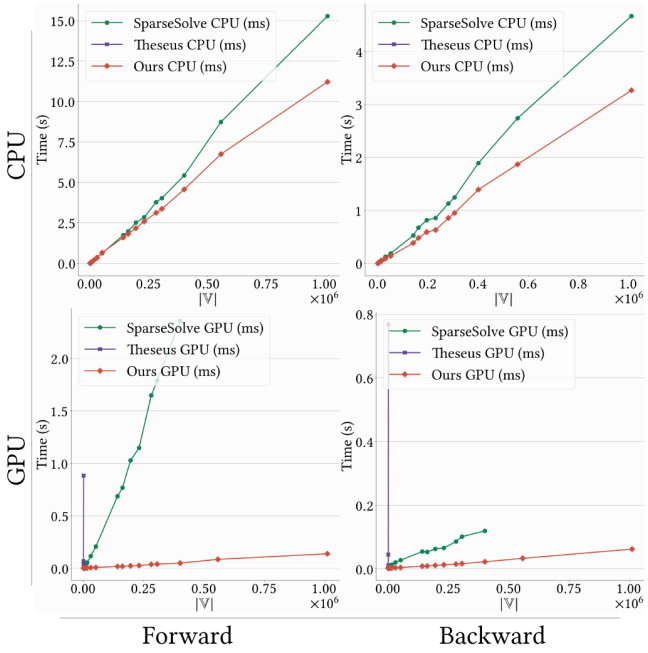


Fig. 5. Runtime comparison of linear solver strategies across the CPU and GPU. Using a declarative method comes with severe limitation in runtime and memory consumption: Theseus did not succeed at differentiating through systems involving meshes with > 642 vertices. A rigid system that pre-defines a solver is also insufficient for geometry processing as it is often possible to make use of problem-specific strategies.

7.2 Inverse Parameterization

Next, we demonstrate our system's capacity for differentiating through sparse eigenproblems. The spectral conformal parameterization (SCP) algorithm [Mullen et al. 2008] has two major benefits over its competitors: it is fast as it only requires solving a sparse eigenproblem, and its solutions are guaranteed to not fold over themselves locally. However, this comes at the cost of the quality of the map: In Figure 7, we see that small areas of a texture get assigned to large areas of the mesh, i.e., one pixel on the texture can become stretched over a large area while others are bunched up in a comparatively minuscule region. Other methods optimize energies meant to reduce this type of distortion, but require additional care to ensure they do not fold over, as well as specialized optimization algorithms [Rabinovich et al. 2017; Smith and Schaefer 2015; Wang et al. 2023b].

Inverse geometry processing offers a way to merge these two approaches. Our goal is to use SCP as the inner algorithm that maps an input mesh to the plane. For the outer optimization we minimize some other distortion energy—e.g., the symmetric Dirichlet energy [Schreiner et al. 2004; Smith and Schaefer 2015]—by training

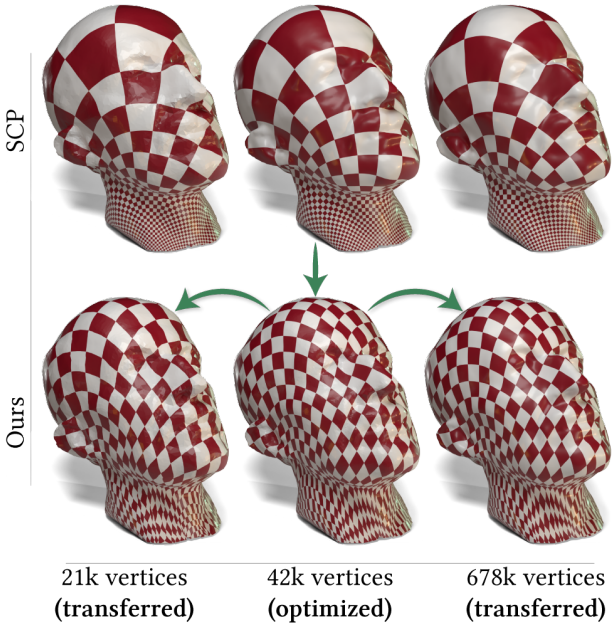


Fig. 6. Left-to-right: same mesh at three different resolutions of a level-of-detail hierarchy. We optimize a neural metric field on the middle-level mesh to reduce area distortion during SCP. Having optimized once, we transfer the metric to other levels by evaluating the field on other meshes.

a neural network to predict an intrinsic metric for SCP to operate in. In other words, our goal is to design a metric that results in a less distorted parameterization compared to SCP, while retaining SCP's suitable properties. For our proof-of-concept, we opted for a simple neural field [Xie et al. 2022] approach, effectively using the neural network as a smooth parameterized interpolant and optimizing on one specific mesh.

This application illustrates how our system enables new pipelines driven by ambient neural fields. Unlike traditional methods, using a neural field allows us to transfer the optimized metric onto a remeshed version of the same geometry, e.g., transferring the parameterization down a level-of-detail mesh hierarchy as shown in Figure 6, a setting not handled by existing approaches. While the purpose of our experiment is this new setting (rather than outperforming the single-mesh parameterization literature), we note that some parameterization methods already do solve *linear* inverse problems, e.g., Wang et al. [2023a], without being transferrable.

SCP is an intrinsic algorithm, meaning it takes as input a set of edge lengths $\boldsymbol{l} \in \mathbb{R}^{|\mathbb{E}|}$ that satisfy certain properties, such comprising a discrete metric with no degenerate elements, and outputs a set of 2D coordinates $\boldsymbol{u} \in \mathbb{R}^{|\mathbb{V}|\times 2}$; thus, we write $\boldsymbol{u} \leftarrow g(\boldsymbol{l})$. In our example, θ are the parameters of a neural field operating in ambient space that we will use to predict a new discrete metric. A simple approach is to predict a displacement vector per mesh vertex and use the displaced vertex positions to compute the discrete metric. Therefore, our neural network takes as input a point in ambient space $\boldsymbol{p} \in \mathbb{R}^3$ and outputs an offset $\boldsymbol{o} \in \mathbb{R}^3$: for edge i connecting vertices k and l, the new edge lengths are $l'_i = \|(\boldsymbol{p}_j + \boldsymbol{o}_j) - (\boldsymbol{p}_k + \boldsymbol{o}_k)\|$. Since we

only ever evaluate our neural network on the mesh vertex positions, we can view the composition of its evaluation with the computation of the edge lengths as $\varphi : \theta \mapsto l'$. We use a multilayer perceptron with 3 hidden layers of 64 neurons each, and use a harmonic input encoding [Mildenhall et al. 2021] with 5 basis functions.

Fig. 7. Spectral conformal parameterizations produce high area distortion. By differentiating through SCP, we can optimize the mesh's metric such that SCP's output under the new metric is less distorted.

Given $\boldsymbol{l'}$, SCP constructs a discrete *conformal* Laplacian $L_c \in \mathbb{R}^{2|\mathbb{V}|\times 2|\mathbb{V}|}$ and solves the generalized eigenproblem

$$L_c \ \mathrm{vec}(\boldsymbol{u}) = \lambda B \ \mathrm{vec}(\boldsymbol{u}), \tag{16}$$

where B is SCP's diagonal "boundary matrix," and $\mathrm{vec}(\cdot)$ flattens $\boldsymbol{u}$ into a $\mathbb{R}^{2|\mathbb{E}|}$ vector. Our parameterization is then the eigenvector $\boldsymbol{u}$ corresponding to the smallest non-zero eigenvector; for more information, refer to Mullen et al. [2008]. In code, the user computes the cotangent weights and calls `iskra`'s spectral implicit layer:

```
def scp(lengths, face_edge, n_vertices, faces, bdr_idx, boundary_mat):
    cot = cotan_weights_intrinsic(lengths, face_edge)
    conf_lap = conformal_laplacian(cot, n_vertices, faces, bdr_idx)
    _, evecs = eigsh(
        conf_lap,
        M=boundary_mat,
        k=3,
        sigma=-1e-12,
        bwd_method="fixed-point",
        bwd_max_iter=25,
    )
    uv = evecs[..., 0:1].reshape(2, -1).mT
    return uv
```

`iskra` implements the forward pass via a sparse generalized eigensolver and allows the user to choose one of a number of different approaches for computing the backward pass; see below for a comparison of different methods. In practice, we precompute and cache purely topological matrices needed to construct the conformal Laplacian, excluding them here for the sake of exposition.

Our outer optimization objective is the well known symmetric Dirichlet energy [Schreiner et al. 2004; Smith and Schaefer 2015]. Given the Jacobian J_i of the map between the rest configuration of triangle $i \in \mathbb{F}$ and its parameterization, its rest area $\boldsymbol{a_i}$, we can write the symmetric Dirichlet energy as:

$$\ell(\boldsymbol{l'}) = \sum_{i\in\mathbb{F}} \boldsymbol{a}_i(\|J_i\|_F + \|J_j^{-1}\|_F), \tag{17}$$

This can be implemented in `iskra` via a gather operation followed by standard PyTorch linear algebra. We optimize this objective using the Adam optimizer with an initial learning rate of 10^{-3}, for a total of 2000 steps. Unlike other methods for optimizing the symmetric Dirichlet energy [Rabinovich et al. 2017; Smith and Schaefer 2015], our method requires no specialized line-search procedure during gradient descent as going through SCP guarantees injectivity.

Eigenstudy. Our system implements four methods for differentiating through eigensystems. We compare the methods in Figure 8 in terms of runtime and accuracy of differentiating through Laplacian eigenvectors, as functions of mesh size and the number of requested eigenvectors.

The first algorithm directly implements Equation 11, solving a different linear system for each eigenvector requested by the user. We treat this method as ground truth for accuracy as it involves no approximations. While it is fast for a small number of eigenvalues, the runtime explodes for a large number of eigenvectors and/or a large mesh, especially since this method has to refactor a large matrix for each requested eigenvector.

The simplest approach *unrolls* power iterations, i.e., stepping back through each iteration of the power iteration algorithm. For our experiments, we used 100 iterations. While fairly accurate, this method requires high memory use; we were unable to run it on meshes with more than 30K vertices for three eigenvectors, and on meshes larger than 162K vertices for 50 eigenvectors. This makes this method of limited practical usefulness, especially since the isolated workload in this test case was not integrated into a larger computational pipeline.

Next is the method of Smirnov and Solomon [2021], which *truncates* the number of eigenvalues used to evaluate the closed-form Equation 12. The method is the fastest as it only inverts one sparse matrix. However, this comes at the cost of accuracy if the number of requested eigenvectors is small relative to the mesh size; conversely, it is well suited for problems where the user requests many eigenvectors relative to mesh size.

Lastly, we offer an algorithm that relies on our *fixed-point* functionality to differentiate through the QR relationship in Equation 13. We set the number of *GMRES* iterations to 100and GMRES tolerances to 0 to match the number of unrolled power-iterations in the unrolled setting. Figure 8 shows that this method offers a good middle ground: it is comparably accurate to other approximate methods and is strictly more accurate than truncation on all but one mesh. While it is slower than truncation, it is comparable to the remaining

methods on small meshes and faster on large meshes across both test conditions. In comparison to the *individual* method, *fixed-point* can pre-factor once and reuse the solver across iterations.

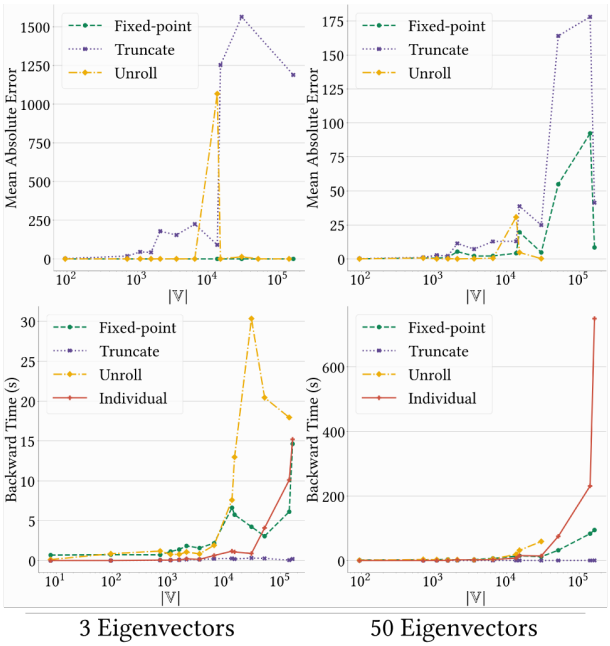


Fig. 8. Comparison of eigenvector derivative computation methods across mesh scales and number of eigenvectors. Top row compares accuracy with respect to *Individual*. Missing data on for the *Unroll* method is due to the fact that stepping back through power iterations requires more memory than available on our system for large enough problems.

As evidenced by our experiments, unrolling power iterations is impractical, *individual* differentiation should be used for small meshes and small numbers of eigenvectors, *truncated* differentiation should be used for large numbers of eigenvectors, and *fixed-point* differentiation offers a reasonable trade-off. All experiments were performed on the CPU.

7.3 Inverse ARAP

We now return to our introductory example of inverting the ARAP problem. Formally, assuming we have cotangent Laplacian weights $\boldsymbol{c}$, the optimization problem solved by ARAP is defined as:

$$\begin{aligned} \underset{\boldsymbol{v}' \in \mathbb{R}^{|\mathbb{V}|\times 3}, \{R_1,\ldots,R_{|\mathbb{V}|}\}}{\arg\min} \quad & \sum_{(i,j)\in \mathbb{VV}} \boldsymbol{c}_{ij} \left\| \left(\boldsymbol{v}'_i - \boldsymbol{v}'_j\right) - R_i(\boldsymbol{v}_i - \boldsymbol{v}_j) \right\|^2 \\ \text{s.t.} \quad & \boldsymbol{v}'_k = \boldsymbol{h}_k, \ k \in \mathbb{H} \\ & R_i \in SO(3), \ i = 1, \ldots, |\mathbb{V}|, \end{aligned} \tag{18}$$

where $\mathbb{H}$ is the set of handles with positions $\boldsymbol{h}$ and R_i are per-vertex rotation matrices. We have already shown how to implement the specialized local-global ARAP solver of Sorkine and Alexa [2007] using our system's fixed-point interface and use it to solve the inverse problem from Equation 5 (§4.2). We demonstrate the results of this optimization on number of meshes in Figures 1 and 9. All results were computed by running the outer optimization loop for 500 iterations. The number of GMRES iterations was kept at 200.

We also include a modification to the problem in Figure 11, which uses motion capture data from the AMASS dataset [Mahmood et al. 2019]. First, we take motion capture markers on the registered body mesh in rest pose and find the input mesh vertex closest to each marker. The goal is to match those vertices to markers captured during the person jumping by fitting a reduced order deformation model to the new set of markers. We compute the loss in (5) between this subset of vertices and the markers, instead of on the full mesh. Notably, the number of ARAP handles is much smaller than the number of markers.

ARAP can be relaxed into a nonlinear least squares problem [Mara et al. 2021], allowing us to compare with Theseus [Pineda et al. 2022] once again. Unlike in §7.1, ARAP's nonlinear least squares formulation can naturally be written without sparse matrices, meaning we manually convert our geometry operators into dense operations. Table 1 includes a comparison on a mesh of a sphere with 642 vertices: we ran both implementations for a total of 25 optimization steps and report the forward and backward pass runtimes, as well as the peak memory use across both CPU and GPU back-ends. Here we see how our system's versatility leads to greater efficiency. As expected, being able to use the specialized algorithm of Sorkine and Alexa [2007] as opposed to a generic solver accelerates the forward pass. Moreover, we see how the generality of our system leads to orders of magnitude better memory characteristics and an order of magnitude faster backward pass. We were unable to run Theseus on meshes significantly larger than 642 vertices as its memory allocations exceeded our computer's 32GB or RAM memory.

Table 1. Despite ARAP falling under the class of problems solvable by Theseus, `iskra`'s imperative approach remains orders of magnitude faster. Note Theseus did not complete executing on larger meshes due to excessive memory requirements.

		Forward (s)	Backward (s)	Memory (GB)
CPU	Theseus	43.69	135.38	18.22
	`iskra`	0.32	0.027	0.67
GPU	Theseus	1.18	2.39	13.76
	`iskra`	0.095	0.018	0.019

7.4 Inverse Geodesic Distances

In this section, we use our system's fixed-point implicit layers to differentiate through geodesic distances, in the process demonstrating our system's support for the large class of geometry processing problems solvable via ADMM iterations [Boyd et al. 2010; Gabay and Mercier 1976; Glowinski and Marroco 1975] While prior work relied on hand-derived formulae for differentiating through geodesic distances [Benmansour et al. 2010; Li et al. 2024], our method automates this derivation for its users. In the *inner* problem, we are optimizing for a *regularized* distance function $u \in \mathbb{R}^{|\mathbb{V}|}$ starting from some set of vertices $\mathbb{S} \subset \mathbb{V}$. It is regularized because it trades off satisfying the geodesic equations and, in our case, the Dirichlet

Fig. 9. Results of using our system to solve the inverse ARAP problem on a range of meshes.

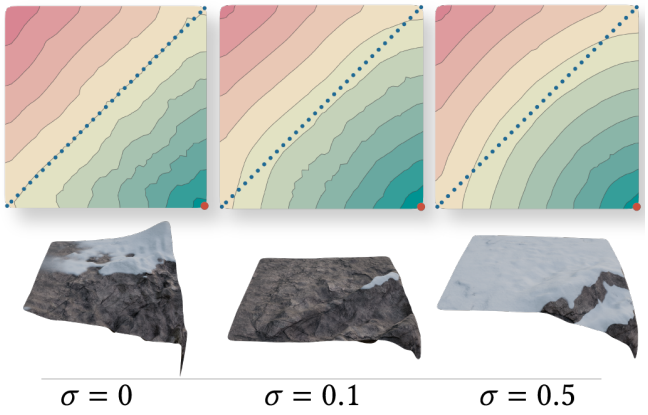


Fig. 10. Our system can differentiate through regularized geodesic distances, facilitating procedural terrain design. Plots in the top row show distances to their bottom-right corners (red). These correspond to a birds-eye view of the heightfield in the bottom row. The objective is to make the center-line points (blue) equidistant from the start point, in combination with a smoothness regularizer governed by σ. Different columns correspond to different regularization strengths.

energy via a chosen parameter β, Assuming vertex areas $m \in \mathbb{R}^{|\mathbb{V}|}$, and the discrete gradient operator G and Laplacian L, we can phrase this problem (with an optional smoothing term controlled by $\beta \geq 0$) as follows

$$\begin{aligned} \min_{u} \quad & -m^\top u + \frac{\beta}{2} u^\top L u \\ \text{s.t.} \quad & \|(Gu)_f\| \leq 1, \quad \forall f \in \mathbb{F} \\ & u_s = 0, \quad \forall s \in \mathbb{S}. \end{aligned} \tag{19}$$

As with previous examples, geometry processing practitioners have proposed specialized solvers that exploit problem-specific structure—in this case a fast and concise ADMM scheme proposed by Edelstein et al. [2023], who introduced formulation (19). This scheme alternates three steps: solving for u, solving for a auxiliary variable $z = Gu$, and solving for the dual variable $\lambda \in \mathbb{R}^{|\mathbb{F}|}$. Assuming a set of algorithm specific constants ρ and β_k, and the divergence operator D, this procedure can be compactly implemented in in `iskra` as:

```
# step 1: u-minimization
b = vert_areas - div @ y.flatten() + rho * div @ z.flatten()
u = linear_solve(lap, b, solver=solver)[1] / (beta + rho)

# step 2: z-minimization
grad_u = (grad @ u).reshape(*z.shape)
z_new = (1 / rho) * y + grad_u
norm_z = torch.linalg.vector_norm(z_new, axis=0)
z_new = z_new / torch.where(norm_z >= 1, norm_z, 1)

# step 3: dual update
y_new = y + rho * (beta_k * grad_u + (1 - beta_k) * z - z_new)
```

In practice, we also implemented the computation of primal and dual residuals, heuristic ρ modulation, and early termination criteria proposed in [Edelstein et al. 2023], bringing up the number of lines of code from 7 to 25 for the inner loop. The user defines g by passing this code to our fixed-point interface (§4.2), specifying which variables act as iterates and which outputs represent the residuals.

For our *outer* optimization, we procedurally design a terrain, e.g., for a videogame. Our mesh is a flat plane embedded in 2D, and our optimization variables are the heights of a heightfield defined on that mesh, $\theta := \zeta \in \mathbb{R}^{|\mathbb{V}|}$; φ is the identity map. We design the terrain such that the distance a character needs to traverse from its corner to any point on its diagonal equals some constant u_c. Our objective is composed from 3 terms: a distance term, a smoothness term weighed by a factor σ, and a non-negativity term:

$$\arg\min_{\zeta} \underbrace{\sum_{i \in \mathbb{C}} (u_c - g(\zeta)_i)^2}_{\text{center-distance}} + \underbrace{\sigma \zeta^\top L \zeta}_{\text{smoothness}} + \underbrace{0.1 \sum_{i \in \mathbb{V}} \text{relu}(-\log(\zeta_i + 1))}_{\text{non-negativity}},$$

where $\mathbb{C}$ denotes the set of vertices on the heightfield's diagonal. We optimize the outer loop using H^1 descent while constraining $\zeta_s = 0$ for all $s \in \mathbb{S}$ to eliminate translational ambiguity. We show the results produced by our system after 500 iterations in Figure 10 for a range of smoothness values σ.

This problem is a quadratic program, solvable systems designed for solving and differentiating through this broad class of problems. Here we include a comparison with the seminal CVXPYLayers system [Agrawal et al. 2019] which builds on the CVX optimization package [CVX Research 2012], running all methods with their default parameters. Implementing this problem in CVXPYLayers required us to reformulate the problem in two ways. First, it required us to manually rewrite sparse matrix operations in terms of

dense tensor operations. Second, we had to convert our problem into the so-called disciplined parameterized programming paradigm of CVXPYLayers [Agrawal et al. 2019]. This is a non-trivial exercise, requiring the user to reason about the "affineness" of each operation in the inner loop—for more detail see [Agrawal et al. 2019].

CVXPYLayers successfully solves this problem on a terrain mesh with 2048 faces. Both methods acheived a comparable loss on the outer problem after 500 iterations with a mutual difference of 10^{-4}. Due to the problem reformulation necessitated by CVXPYLayers, the amount of code required by both methods ended up being comparable, and so were the forward pass runtimes. On the CPU our forward pass on average took 0.299s (ranging from 0.150s to 1.163s), compared to CVXPYLayers 0.301s (0.194s to 0.661s). However, our backward pass was faster: on average our method took 3.670s (0.150 to 6.919s), whereas CVXPYLayers took 12.335s, (5.538 to 13.407s). The variation in the runtimes are due to the early stopping critaria of both methods, and, in our case, of GMRES during the backward pass. While comparable on sheer runtime, our method outperforms CVXPYLayers by an order of magnitude when it comes to memory use. Simply applying one step of Loop subdivision [Loop 1987] results in CVXPYLayers attempting to allocate more memory than available on our 32GB machine, leading it to crash. In contrast, our system's memory usage remains below 1GB for this test case.

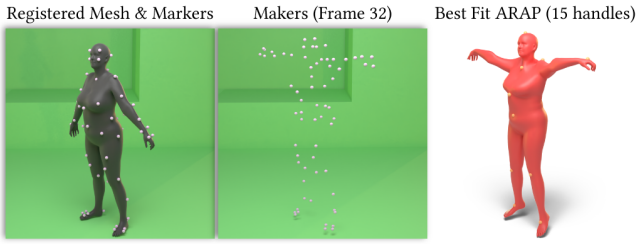


Fig. 11. Fitting an ARAP deformation to match motion capture markers.

8 Discussion

Our results thoroughly demonstrate our system in relation to its design goals of versatility, usability, interoperability, and efficiency. In particular, we measure versatility by solving problems involving sparse linear systems, sparse eigendecompositions, local-global iterations, and ADMM formulations, each representative of a large class of geometry problems. As shown in §7.1 and §7.3, given a formulation of the desired implicit relationship f, `iskra`'s code is often more succinct than its declarative counterparts, demonstrating its usability. Experiments like the one in §7.2 showcase interoperability by directly integrating with machine learning architectures through PyTorch. In all of our experiments, `iskra` is either comparable to, or strictly faster than alternatives.

Design goals like ours are often in tension with one another. For example, usability typically inhibits low-level control, which otherwise would imply greater implementation effort. Surprisingly, however, our system satisfies all the design goals simultaneously. In fact, succinctly expressing complex geometric algorithms *necessitated* additional affordances for simplicity and expressivity. These affordances include our tensor-based hierarchical scatter-gather formulation, sparse linear algebra functionality, and high-level utilities for fixed-point iteration algorithms. These features are crucial for geometry processing, allowing practitioners to differentiate through existing specialized algorithms without needing to reformulate them.

Moreover, we have demonstrated how our system naturally integrates with machine learning workflows (§7.2), opening doors to a broad class of geometry processing pipelines. While our example in §7.2 offers a proof-of-concept integration with a neural field, future directions may combine implicit geometric layers with larger-scale data-driven machine learning workflows. Extrapolating from this example, a future data-driven model may provide priors and then rely on implicit geometric layers to guarantee that geometric constraints are satisfied. This direction is particularly compelling as it promises to marry two, until now, mostly disjoint worlds in geometry processing: data-driven and optimization-based methods.

Despite our framework's simplicity and concision in practical examples, we achieve versatility by requiring the user to provide an implicit relationship. In practice, this is desirable because we wish to use existing geometry processing formulations. However, one could imagine adding utilities similar to our fixed-point interface, e.g., one that translates a declarative-style constrained quadratic program specification into an implicit function (like CVX [CVX Research 2012; Grant and Boyd 2008]) or another that creates a proximal iteration to be plugged into our fixed-point interface (like ∇-Prox [Lai et al. 2023]). While we focused on existing geometry processing algorithms, these extensions may enable rapid prototyping of completely new geometry processing techniques.

Furthermore, one could likely extend our system to physics simulation, taking inspiration from, e.g., Huang et al. [2024]. We suspect that implementing collision detection or friction could be accomplished using differentiable utilities in the geometry module, orthogonal to the rest of our system—e.g., §7.4 already implements a naïve penalty forcing terrain heights to be nonnegative. Differentiating through ordinary differential equations is currently possible either by unrolling multiple implicit layers or by specifying the system within our fixed-point interface; we leave for future work implementing time integrators within our custom backward pass interface. Lastly, we only handle dynamic topology changes *outside* the implicit function interface: the vector-Jacobian product used to construct the backward pass does not see topology changes that may have happened during the forward pass. Differentiable topology changes (see, e.g., Rakotosaona et al. [2021]) remain an avenue for future work.

While the generality of the off-the-shelf GMRES optimizer allows us to differentiate through a broad class of problems, future work could look into faster or more robust approaches. For systems with multiple solutions, GMRES produces the "Drazin inverse" solution, making it somewhat more predictable than naïve direct solvers. Singular Jacobians are still possible, but we did not encounter any empirically. We did experience poorly-conditioned Jacobians, e.g., in the geodesics example, where many GMRES iterations were necessary for convergence. In this case, we encountered the warning

stating GMRES failed to converge within the chosen iteration count—since gradient descent can recover from minor gradient perturbations, however, we did not ultimately find this to be an issue. While `iskra` allows for custom solvers for the backward pass, Jacobi preconditioning was overall slower despite needing fewer iterations; we leave derivation of more effective preconditioning strategies to future work.

While not explored here, our method can also be combined with other differentiation utilities in geometry processing, such as libraries that allow users to construct sparse Hessians from energies defined on meshes [Jacobson 2025a]. Our system may also benefit from future innovation in differentiable mesh systems that bridge the gap between low-level high-performance GPU data structures [Mahmoud et al. 2021; Yu et al. 2022] and tensor-based machine learning systems. Moreover, we foresee possible extensions and integrations with differentiable physics simulation and contact handling [Hu et al. 2020; Macklin 2022], higher-order or non-simplicial finite elements [Schneider et al. 2019], and algorithms that involve dynamic topology changes [Huang et al. 2024]. Lastly, our system is likely compatible with inverse rendering [Ansel et al. 2024; Nimier-David et al. 2019], enabling end-to-end differentiable graphics applications.

In providing a system for the automatic computation of adjoints, we bring the same kind of simplicity and ease of experimentation to inverse problems in geometry processing that automatic differentiation frameworks have brought to other domains. Our system opens doors to new geometry processing pipelines, unifying machine learning approaches with classical optimization-based algorithms. It does so in a compact, fast, memory efficient manner, without requiring practitioners to reformulate their algorithms.

Acknowledgments

We thank Brandon Amos, Oded Stein, and Jonathan Ragan-Kelley for helpful early discussions. The MIT Geometric Data Processing Group acknowledges the generous support of National Science Foundation grants IIS2335492 and OAC2403239, from the CSAIL Future of Data and FinTechAI programs, from the MIT–IBM Watson AI Laboratory, from the Wistron Corporation, from the MIT Generative AI Impact Consortium, from the Toyota–CSAIL Joint Research Center, and from Schmidt Sciences.

References

A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. 2019. Differentiable Convex Optimization Layers. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Article 858, 13 pages. https://dl.acm.org/doi/10.5555/3454287.3455145

Brandon Amos and J. Zico Kolter. 2017. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 136–145.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. https://doi.org/10.1145/3620665.3640366

F. Benmansour, G. Carlier, G. Peyré, and F. Santambrogio. 2010. Derivatives with respect to metrics and applications: subgradient marching algorithm. *Numer. Math.* 116, 3 (Sept. 2010), 357–381. https://doi.org/10.1007/s00211-010-0305-8

M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. 2002. OpenMesh – a generic and efficient polygon mesh data structure. In *1st OpenSG Symposium*. https://www.graphics.rwth-aachen.de/media/papers/openmesh1.pdf

Pierre Bourquat, David Coeurjolly, Guillaume Damiand, and Florent Dupont. 2022. Hierarchical mesh-to-points as-rigid-as-possible registration. *Computers & Graphics* 102 (2022), 320–328. https://doi.org/10.1016/j.cag.2021.10.016

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. 2010. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine learning* 3, 1 (2010), 1–122. https://doi.org/10.1561/2200000016

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2018. Neural Ordinary Differential Equations. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 6572–6583. https://dl.acm.org/doi/10.5555/3327757.3327764

Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw.* 35, 3, Article 22 (Oct. 2008), 14 pages. https://doi.org/10.1145/1391989.1391995

Stelian Coros, Miles Macklin, Bernhard Thomaszewski, and Nils Thürey. 2021. Differentiable simulation. In *SIGGRAPH Asia 2021 Courses* (Tokyo, Japan) *(SA '21)*. Association for Computing Machinery, New York, NY, USA, Article 3, 142 pages. https://doi.org/10.1145/3476117.3483433

Luca Cosmo, Mikhail Panine, Arianna Rampini, Maks Ovsjanikov, Michael M. Bronstein, and Rodolá Rodolá. 2019. Isospectralization, or How to Hear Shape, Style, and Correspondence. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 7521–7530. https://doi.org/10.1109/CVPR.2019.00771

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)*. Association for Computing Machinery, New York, NY, USA, Article 7, 126 pages. https://doi.org/10.1145/2504435.2504442

Inc. CVX Research. 2012. CVX: Matlab Software for Disciplined Convex Programming, version 2.0. https://cvxr.com/cvx.

Mathieu Desbrun, Eva Kanso, and Yiying Tong. 2006. Discrete differential forms for computational modeling. In *ACM SIGGRAPH 2006 Courses* (Boston, Massachusetts) *(SIGGRAPH '06)*. Association for Computing Machinery, New York, NY, USA, 39—-54. https://doi.org/10.1145/1185657.1185665

Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. 1999. *Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow*. ACM Press/Addison-Wesley Publishing Co., USA, 317–324. https://doi.org/10.1145/311535.311576

Tao Du, Kui Wu, Pingchuan Ma, Sebastien Wah, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. 2021. DiffPD: Differentiable Projective Dynamics. *ACM Trans. Graph.* 41, 2, Article 13 (Nov. 2021), 21 pages. https://doi.org/10.1145/3490168

Michal Edelstein, Nestor Guillen, Justin Solomon, and Mirela Ben-Chen. 2023. A Convex Optimization Framework for Regularized Geodesic Distances. , Article 2 (July 2023), 11 pages. https://doi.org/10.1145/3588432.3591523

José Antonio Fernández-Fernández, Fabian Löschner, Lukas Westhofen, Andreas Longva, and Jan Bender. 2025. SymX: Energy-based Simulation from Symbolic Expressions. *ACM Trans. Graph.* 45, 1, Article 5 (Oct. 2025), 19 pages. https://doi.org/10.1145/3764928

Daniel Gabay and Bertrand Mercier. 1976. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications* 2, 1 (March 1976), 17–40. https://doi.org/10.1016/0898-1221(76)90003-1

Roland Glowinski and Americo Marroco. 1975. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de Dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique* 9, R2 (1975), 41–76. https://www.numdam.org/item/M2AN_1975__9_2_41_0/

Michael Grant and Stephen Boyd. 2008. Graph implementations for nonsmooth convex programs. In *Recent Advances in Learning and Control*, V. Blondel, S. Boyd, and H. Kimura (Eds.). Springer London, London, 95–110. https://doi.org/10.1007/978-1-84800-155-8_7

Philipp Herholz, Tuur Stuyck, and Ladislav Kavan. 2024. A Mesh-based Simulation Framework using Automatic Code Generation. *ACM Trans. Graph.* 43, 6, Article 215 (Dec. 2024), 17 pages. https://doi.org/10.1145/3687986

Philipp Herholz, Xuan Tang, Teseo Schneider, Shoaib Kamil, Daniele Panozzo, and Olga Sorkine-Hornung. 2022. Sparsity-Specific Code Optimization using Expression Trees. *ACM Trans. Graph.* 41, 5, Article 175 (Oct. 2022), 19 pages. https://doi.org/10.1145/3520484

Anil N. Hirani. 2003. *Discrete Exterior Calculus.* Ph. D. Dissertation. California Institute of Technology. http://resolver.caltech.edu/CaltechETD:etd-05202003-095403

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).

Yuanming Hu, Jiancheng Liu, Andrew Spielberg, Joshua B. Tenenbaum, William T. Freeman, Jiajun Wu, Daniela Rus, and Wojciech Matusik. 2019. ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics. In *2019 International Conference on Robotics and Automation (ICRA)* (Montreal, QC, Canada). IEEE Press, 6265–6271. https://doi.org/10.1109/ICRA.2019.8794333

Zizhou Huang, Davi Colli Tozoni, Arvi Gjoka, Zachary Ferguson, Teseo Schneider, Daniele Panozzo, and Denis Zorin. 2024. Differentiable solver for time-dependent deformation problems with contact. *ACM Trans. Graph.* 43, 3, Article 31 (May 2024), 30 pages. https://doi.org/10.1145/3657648

Takeo Igarashi, Tomer Moscovich, and John F. Hughes. 2005. As-Rigid-as-Possible Shape Manipulation. *ACM Trans. Graph.* 24, 3 (July 2005), 1134––1141. https://doi.org/10.1145/1073204.1073323

Alec Jacobson. 2025a. indexed_sum: fast sparse Hessians in PyTorch. https://github.com/alecjacobson/indexed_sum. https://github.com/alecjacobson/indexed_sum GitHub repository, accessed 22 Jan 2026.

Alec Jacobson. 2025b. *pytorch-sparse-solve: Differentiable sparse linear solver for PyTorch.* https://github.com/alecjacobson/pytorch-sparse-solve GitHub repository, accessed January 14, 2026.

Alec Jacobson et al. 2021. gptoolbox: Geometry Processing Toolbox. http://github.com/alecjacobson/gptoolbox.

Alec Jacobson and Daniele Panozzo. 2017. libigl: prototyping geometry processing research in C++. In *SIGGRAPH Asia 2017 Courses* (Bangkok, Thailand) *(SA '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 172 pages. https://doi.org/10.1145/3134472.3134497

J Karátson and L Lóczi. 2005. Sobolev gradient preconditioning for the electrostatic potential equation. *Computers & Mathematics with Applications* 50, 7 (2005), 1093–1104. https://doi.org/10.1016/j.camwa.2005.08.011

J. Zico Kolter, David Duvenaud, and Matt Johnson. 2020. Deep Implicit Layers: Neural ODEs, Deep Equilibrium Models, and Beyond. NeurIPS 2020 Tutorial. https://implicit-layers-tutorial.org/ Video presentation available at https://slideslive.com/38935810/deep-implicit-layers-neural-odes-equilibrium-models-and-beyond.

Zeqiang Lai, Kaixuan Wei, Ying Fu, Philipp Härtel, and Felix Heide. 2023. ∇-Prox: Differentiable Proximal Algorithm Modeling for Large-Scale Optimization. *ACM Transactions on Graphics (TOG)* 42, 4 (Aug. 2023), 1–19. https://doi.org/10.1145/3592144

R. B. Lehoucq, D. C. Sorensen, and C. Yang. 1998. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719628

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph.* 37, 4, Article 139 (July 2018), 13 pages. https://doi.org/10.1145/3197517.3201383

Yue Li, Logan Numerow, Bernhard Thomaszewski, and Stelian Coros. 2024. Differentiable Geodesic Distance for Intrinsic Minimization on Triangle Meshes. *ACM Trans. Graph.* 43, 4, Article 91 (July 2024), 14 pages. https://doi.org/10.1145/3658122

Hsueh-Ti Derek Liu and Alec Jacobson. 2019. Cubic stylization. *ACM Trans. Graph.* 38, 6, Article 197 (Nov. 2019), 10 pages. https://doi.org/10.1145/3355089.3356495

Charles T. Loop. 1987. Smooth Subdivision Surfaces Based on Triangles. https://api.semanticscholar.org/CorpusID:116150707

Miles Macklin. 2022. Warp: A High-performance Python Framework for GPU Simulation and Graphics. https://github.com/nvidia/warp. NVIDIA GPU Technology Conference (GTC).

Naureen Mahmood, Nima Ghorbani, Nikolaus F. Troje, Gerard Pons-Moll, and Michael J. Black. 2019. AMASS: Archive of Motion Capture as Surface Shapes. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 5442–5451. https://doi.org/10.1109/ICCV.2019.00554

Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2021. RXMesh: A GPU Mesh Data Structure. *ACM Transactions on Graphics* 40, 4, Article 104 (Aug. 2021), 16 pages. https://doi.org/10.1145/3450626.3459748

Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2025. Dynamic Mesh Processing on the GPU. *ACM Transactions on Graphics* 44, 4, Article 136 (July 2025), 19 pages. https://doi.org/10.1145/3731162

Josiah Manson and Scott Schaefer. 2011. Hierarchical Deformation of Locally Rigid Meshes. *Computer Graphics Forum* 30, 8 (2011), 2387–2396. https://doi.org/10.1111/j.1467-8659.2011.02074.x

Michael Mara, Felix Heide, Michael Zollhöfer, Matthias Nießner, and Pat Hanrahan. 2021. Thallo – Scheduling for High-Performance Large-Scale Non-Linear Least-Squares Solvers. *ACM Trans. Graph.* 40, 5, Article 184 (Oct. 2021), 14 pages. https://doi.org/10.1145/3453986

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2021. NeRF: representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (Dec. 2021), 99–106. https://doi.org/10.1145/3503250

Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. 2008. Spectral Conformal Parameterization, In Proceedings of the Symposium on Geometry Processing. *Computer Graphics Forum*, 1487–1494. https://doi.org/10.5555/1731309.1731335

JW Neuberger. 1985. Steepest descent and differential equations. *Journal of the Mathematical Society of Japan* 37, 2 (1985), 187–195. https://doi.org/10.2969/jmsj/03720187

Rhys Newbury, Jack Collins, Kerry He, Jiahe Pan, Ingmar Posner, David Howard, and Akansel Cosgun. 2024. A Review of Differentiable Simulators. *IEEE Access* 12 (July 2024). https://doi.org/10.1109/ACCESS.2024.3425448

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *ACM Trans. Graph.* 38, 6, Article 203 (Nov. 2019), 17 pages. https://doi.org/10.1145/3355089.3356498

NVIDIA Corporation. 2025. cuDSS: Release 0.7.1. https://developer.nvidia.com/cudss/.

Stanley Osher, Bao Wang, Penghang Yin, Xiyang Luo, Farzin Barekat, Minh Pham, and Alex Lin. 2022. Laplacian smoothing gradient descent. *Research in the Mathematical Sciences* 9, 3 (Aug. 2022), 55. https://doi.org/10.1007/s40687-022-00351-1

Jianming Pan, Zeqi Ye, Xiao Yang, Xu Yang, Weiqing Liu, Lewen Wang, and Jiang Bian. 2024. BPQP: a differentiable convex optimization framework for efficient end-to-end learning. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. 77468–77493. https://doi.org/10.52202/079017-2463

Yue Peng, Bailin Deng, Juyong Zhang, Fanyu Geng, Wenjie Qin, and Ligang Liu. 2018. Anderson acceleration for geometry optimization and physics simulation. *ACM Trans. Graph.* 37, 4, Article 42 (July 2018), 14 pages. https://doi.org/10.1145/3197517.3201290

Luis Pineda, Taosha Fan, Maurizio Monge, Shobha Venkataraman, Paloma Sodhi, Ricky TQ Chen, Joseph Ortiz, Daniel DeTone, Austin Wang, Stuart Anderson, Jing Dong, Brandon Amos, and Mustafa Mukadam. 2022. Theseus: A Library for Differentiable Nonlinear Optimization. In *36th Conference on Neural Information Processing System (NeurIPS 2022)*.

The CGAL Project. 2025. *CGAL User and Reference Manual* (6.1 ed.). CGAL Editorial Board. https://doc.cgal.org/6.1/Manual/packages.html

Michael Rabinovich, Roi Poranne, Daniele Panozzo, and Olga Sorkine-Hornung. 2017. Scalable locally injective mappings. *ACM Transactions on Graphics (TOG)* 36, 2, Article 16 (April 2017), 16 pages. https://doi.org/10.1145/2983621

Marie-Julie Rakotosaona, Noam Aigerman, Niloy J. Mitra, Maks Ovsjanikov, and Paul Guerrero. 2021. Differentiable surface triangulation. *ACM Trans. Graph.* 40, 6, Article 267 (Dec. 2021), 13 pages. https://doi.org/10.1145/3478513.3480554

Bernhard Reinert, Tobias Ritschel, and Hans-Peter Seidel. 2012. Homunculus Warping: Conveying importance using self-intersection-free non-homogeneous mesh deformation. *Computer Graphics Forum* 31 (Sept. 2012), 2165–2171. https://doi.org/10.1111/j.1467-8659.2012.03209.x

Youcef Saad and Martin H. Schultz. 1986. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput.* 7, 3 (1986), 856–869. https://doi.org/10.1137/0907058

P. Schmidt, J. Born, D. Bommes, M. Campen, and L. Kobbelt. 2022. TinyAD: Automatic Differentiation in Geometry Processing Made Simple. *Computer Graphics Forum* 41, 5 (Oct. 2022), 113–124. https://doi.org/10.1111/cgf.14607

Teseo Schneider, Jérémie Dumas, Xifeng Gao, Denis Zorin, and Daniele Panozzo. 2019. PolyFEM. https://polyfem.github.io/.

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

Silvia Sellán, Oded Stein, et al. 2023. gptyoolbox: A Python Geometry Processing Toolbox. https://gpytoolbox.org/.

Dmitriy Smirnov and Justin Solomon. 2021. HodgeNet: Learning Spectral Geometry on Triangle Meshes. *ACM Trans. Graph.* 40, 4, Article 166 (July 2021), 11 pages. https://doi.org/10.1145/3450626.3459797

Jason Smith and Scott Schaefer. 2015. Bijective parameterization with free boundaries. *ACM Transactions on Graphics (TOG)* 34, 4, Article 70 (July 2015), 9 pages. https://doi.org/10.1145/2766947

Justin Solomon, Fernando de Goes, Gabriel Peyré, Marco Cuturi, Adrian Butscher, Andy Nguyen, Tao Du, and Leonidas Guibas. 2015. Convolutional wasserstein distances: efficient optimal transportation on geometric domains. *ACM Trans. Graph.* 34, 4, Article 66 (July 2015), 11 pages. https://doi.org/10.1145/2766963

Justin Solomon, Raif Rustamov, Leonidas Guibas, and Adrian Butscher. 2014. Earth mover's distances on discrete surfaces. *ACM Trans. Graph.* 33, 4, Article 67 (July 2014), 12 pages. https://doi.org/10.1145/2601097.2601175

Olga Sorkine and Marc Alexa. 2007. As-Rigid-As-Possible Surface Modeling. In *Geometry Processing*, Alexander Belyaev and Michael Garland (Eds.). The Eurographics Association. https://doi.org/10.2312/SGP/SGP07/109-116

Gilbert Strang. 2007. *Computational Science and Engineering* (1st ed.). Wellesley-Cambridge Press, Philadelphia, PA. https://epubs.siam.org/doi/abs/10.1137/1.9780961408817

Amir Vaxman et al. 2025. Directional: A library for Directional Field Synthesis, Design, and Processing. https://doi.org/10.5281/zenodo.3338174

Hippolyte Verninas, Caner Korkmaz, Stefanos Zafeiriou, Tolga Birdal, and Simone Foti. 2026. Parallelised Differentiable Straightest Geodesics for 3D Meshes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (Feb. 2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Yu Wang, Minghao Guo, and Justin Solomon. 2023a. Variational quasi-harmonic maps for computing diffeomorphisms. *ACM Trans. Graph.* 42, 4, Article 130 (July 2023), 26 pages. https://doi.org/10.1145/3592105

Yu Wang, Minghao Guo, and Justin Solomon. 2023b. Variational quasi-harmonic maps for computing diffeomorphisms. *ACM Transactions on Graphics (TOG)* 42, 4, Article 130 (July 2023), 26 pages. https://doi.org/10.1145/3592105

Yu Wang and Justin Solomon. 2021. Fast quasi-harmonic weights for geometric data interpolation. *ACM Trans. Graph.* 40, 4, Article 73 (July 2021), 15 pages. https://doi.org/10.1145/3450626.3459801

Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. 2022. Neural fields in visual computing and beyond. *Computer Graphics Forum* 41, 2 (May 2022), 641–676. https://doi.org/10.1111/cgf.14505

Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. 2022. MeshTaichi: A Compiler for Efficient Mesh-Based Operations. *ACM Trans. Graph.* 41, 6, Article 252 (Nov. 2022), 17 pages. https://doi.org/10.1145/3550454.3555430