Eurographics Symposium on Geometry Processing 2022
M. Campen and M. Spagnuolo
(Guest Editors)

Volume 41 (2022), Number 5

TinyAD: Automatic Differentiation in Geometry Processing Made Simple

P. Schmidt[1] J. Born[1] D. Bommes[2] M. Campen[3] L. Kobbelt[1]

[1]RWTH Aachen University, Germany [2]University of Bern, Switzerland [3]Osnabrück University, Germany

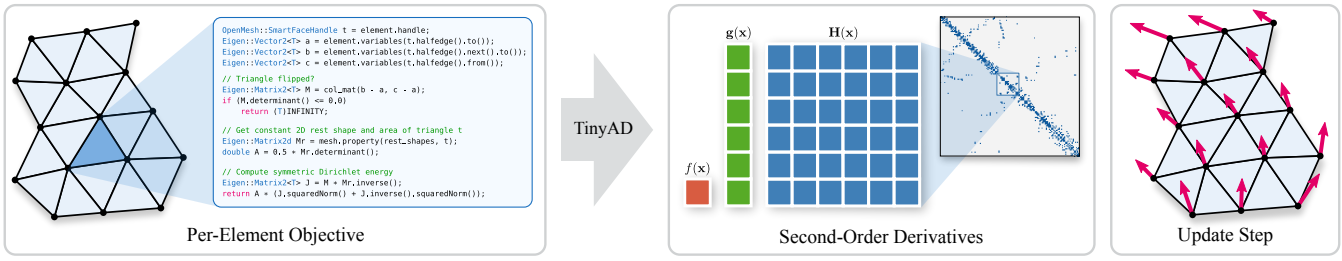

Figure 1: *TinyAD offers automatic differentiation targeted at geometry processing problems on meshes. Left: A user implements an objective function in C++ using the Eigen library (here per triangle, but arbitrary mesh primitives or stencils can be chosen as elements). Center: TinyAD efficiently computes second-order derivatives per element and automatically assembles the global gradient and sparse Hessian matrix. Right: The user computes an update step (here per vertex) using a standard linear solver. Via this simplified access to derivatives, we enable quick exploration of complex objectives in research workflows.*

Abstract

Non-linear optimization is essential to many areas of geometry processing research. However, when experimenting with different problem formulations or when prototyping new algorithms, a major practical obstacle is the need to figure out derivatives of objective functions, especially when second-order derivatives are required. Deriving and manually implementing gradients and Hessians is both time-consuming and error-prone. Automatic differentiation techniques address this problem, but can introduce a diverse set of obstacles themselves, e.g. limiting the set of supported language features, imposing restrictions on a program's control flow, incurring a significant run time overhead, or making it hard to exploit sparsity patterns common in geometry processing. We show that for many geometric problems, in particular on meshes, the simplest form of forward-mode automatic differentiation is not only the most flexible, but also actually the most efficient choice. We introduce TinyAD: a lightweight C++ library that automatically computes gradients and Hessians, in particular of sparse problems, by differentiating small (tiny) sub-problems. Its simplicity enables easy integration; no restrictions on, e.g., looping and branching are imposed. TinyAD provides the basic ingredients to quickly implement first and second order Newton-style solvers, allowing for flexible adjustment of both problem formulations and solver details. By showcasing compact implementations of methods from parametrization, deformation, and direction field design, we demonstrate how TinyAD lowers the barrier to exploring non-linear optimization techniques. This enables not only fast prototyping of new research ideas, but also improves replicability of existing algorithms in geometry processing. TinyAD is available to the community as an open source library.

CCS Concepts

• ***Mathematics of computing*** → *Automatic differentiation; Mathematical software;*
• ***Computing methodologies*** → *Mesh models;*

1. Introduction

Many tasks in geometry processing are, at their core, instances of non-linear (possibly non-convex) optimization problems. Countless examples can be found: in mesh parametrization (e.g. for texturing [PTH*17, LKK*18], structured remeshing [LCBK19], or surface mapping [APL14, SBCK19]), in mesh deformation (e.g. for animation [SdGK19], quad mesh planarization [LPW*06], developable surface approximation [SGC18, SAJ20], registration [HAWG08], or surface reconstruction [NJJ21]), in direction field design (e.g. for remeshing [JFH*15] or fabrication [SFCBCV19]), in numerical simulation and engineering [MGS*21], and in many other branches of geometry processing.

Research in these areas often involves (1) formulating the right optimization problem, (2) developing a suitable solution scheme, and (3) implementing a working prototype. These steps are typically interdependent, need to be iterated, and are—more often than not—driven by working around technical restrictions. Allowing flexible prototyping (e.g. to be able to quickly try different objective functions and solver details) is an ongoing effort and can greatly influence the success of research endeavors.

A major practical roadblock in non-linear optimization is finding and implementing the derivatives of an objective function, in particular second-order derivatives. Even though they can provide a significant advantage in optimization, second-order derivatives are often avoided as they can be excessively tedious to compute. In the worst case, research ideas remain unexplored as finding, implementing, and debugging derivatives is just too time-consuming.

The classical approach to differentiation is by analytically deriving expressions for the gradient and Hessian on paper and manually implementing them in code [Sch19]. Even for functions of moderate complexity, this often turns out to be a slow and error-prone process: researchers need to verify their derivations, debug their code, and start from scratch when the objective function changes.

A slightly more automated approach is symbolic differentiation and code generation in computer algebra systems like Maple, Mathematica, Matlab or SymPy [MSP*17]. While this automates the main differentiation workload, it is still a two-step process that requires switching between different software ecosystems, leading to manual intervention or additional automation efforts. Moreover, symbolic differentiation of complex objectives tends to produce excessively large expressions that are slow to generate and to compile.

Many of these drawbacks do not apply when performing automatic differentiation directly in the target language, e.g. via operator overloading mechanisms in C++. However, existing solutions can still come with a number of challenges, for example when they limit the subset of available language features, incur a run time overhead, or restrict a program's control flow. In particular, it is common to not allow dynamic loops or branching in the differentiated code [DSJ*22]. This can be worked around by explicitly enumerating and differentiating all possible branches (which quickly becomes infeasible) or by differentiating only the current branch at run time [WG12] (which can be expensive). Both cases induce added implementation effort. Furthermore, many automatic differentiation libraries support first-order derivatives only [HP13, Hog14, Jak19, Yan21], are difficult to integrate, or add a significant run time overhead. Moreover, direct support for mesh-based problems is a rare feature, meaning that their typical sparsity patterns have to either be inferred from a computation graph (often slow) or managed manually (causing extra boilerplate code).

Navigating these restrictions and finding the right automatic differentiation package can be a daunting task. General-purpose libraries can be heavy-weight and difficult to use, while too specific libraries from other fields might be challenging to repurpose for geometry processing tasks. This is in stark contrast to adjacent fields such as machine learning, where problem-specific automatic differentiation libraries (e.g. PyTorch or TensorFlow) have been contributing enormously to the tremendous progress of this research area. We argue that simple and easy-to-integrate automatic

```cpp
// Read disk-topology mesh using OpenMesh
OpenMesh::TriMesh mesh = read_mesh("armadillo_disk.obj");

// Set up function with 2D vertex positions as variables.
auto func = TinyAD::scalar_function<2>(mesh.vertices());

// Add objective term per triangle. Each connecting 3 vertices.
func.add_elements<3>(mesh.faces(), [&] (auto& element)
{
    // Element is evaluated with either double or TinyAD::Double<6>
    using T = TINYAD_SCALAR_TYPE(element);

    // Get variable 2D vertex positions of triangle t
    OpenMesh::SmartFaceHandle t = element.handle;
    Eigen::Vector2<T> a = element.variables(t.halfedge().to());
    Eigen::Vector2<T> b = element.variables(t.halfedge().next().to());
    Eigen::Vector2<T> c = element.variables(t.halfedge().from());

    // Triangle flipped?
    Eigen::Matrix2<T> M = col_mat(b - a, c - a);
    if (M.determinant() <= 0.0)
        return (T)INFINITY;

    // Get constant 2D rest shape and area of triangle t
    Eigen::Matrix2d Mr = mesh.property(rest_shapes, t);
    double A = 0.5 * Mr.determinant();

    // Compute symmetric Dirichlet energy
    Eigen::Matrix2<T> J = M * Mr.inverse();
    return A * (J.squaredNorm() + J.inverse().squaredNorm());
});

// Projected Newton
Eigen::VectorXd x = tutte_embedding(mesh);
for (int i = 0; i < max_iters; ++i)
{
    auto [f, g, H_proj] = func.eval_with_hessian_proj(x);
    Eigen::VectorXd d = TinyAD::newton_direction(g, H_proj);
    if (TinyAD::newton_decrement(d, g) < eps)
        break;
    x = TinyAD::line_search(x, d, f, g, func);
}
```

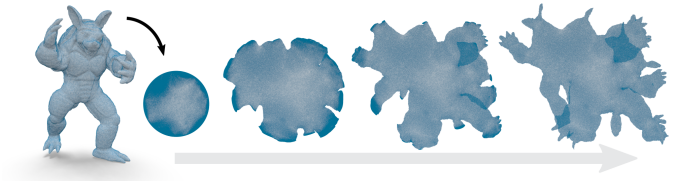

Figure 2: *Fully functional implementation of a mesh parametrization algorithm using TinyAD, optimizing the symmetric Dirichlet energy via a projected-Newton method.*

differentiation tools, tailored to typical optimization problems on meshes, can also vastly accelerate research in geometry processing. Such tools not only lower the barrier to exploring novel ideas and formulations, but also improve replicability of existing works by reducing re-implementation effort, as we demonstrate in Section 4.

Once derivatives are available, the task of finding a suitable optimization algorithm remains. Efficient non-linear (and in particular non-convex) optimization tends to require problem-specific solution strategies that are not always provided by black-box solvers. Examples of custom solver details implemented in geometry processing works include manifold optimization [RS15], problem-specific pre-conditioning [KGL16, CBSS17, SBCK19], varying problem size [JSP17], varying variable representation [SCBK20], varying objective functions [LYNF18], alternating optimization [ESBC19], derivative manipulation and transport [SCBK20], custom line search strategies (e.g. involving mesh modifications) [CCS*21, SCBK20], and many more. Common optimization libraries (e.g. IPOPT [WB06]) follow a declarative paradigm, that

is, they strive for abstraction between a user-defined problem description and a black-box solver. However, this may raise the barrier to quickly explore different solver modifications. In contrast, we demonstrate that, once derivatives are available, it is relatively straightforward to manually implement basic Newton-style solvers, which can then form the basis for various modifications, such as the ones listed above. Together with automatic differentiation, this approach enables quick prototyping and allows for flexible adjustment of both the problem formulation and solver details.

1.1. Contribution

In this work, we

- show that the simplest form of forward-mode automatic differentiation is the one best suited for geometry processing tasks,
- introduce TinyAD[1], a lightweight and fast header-only C++ library for second-order automatic differentiation on top of Eigen,
- provide an interface to model the typical sparsity patterns of mesh-based problems, which is easy to use with a variety of different mesh representations and libraries (e.g. OpenMesh, Geometry Central, polymesh, or libigl-style matrices),
- demonstrate how to produce compact implementations of a number of well-known non-linear geometry processing methods.

TinyAD performs per-element automatic differentiation in forward mode. It depends only on the Eigen library [GJ*10], supports arbitrary looping and branching, and draws its run time performance from differentiation at compile time. Note that we do not mathematically or algorithmically introduce a new type of automatic differentiation, but a practical realization tailored towards geometric problems. Similarly, we do not propose new optimization algorithms, but provide the tools necessary to quickly implement such methods. Using examples from surface mesh parametrization (Section 4.1), volume mesh deformation (Section 4.2), frame field design (Section 4.3), and manifold optimization (Section 4.4), we demonstrate how C++ implementations of non-linear geometry processing methods can look very similar to their pseudocode or formulas in a paper.

2. Background & Related Work

For context, we first recap the main steps of unconstrained non-linear (non-convex) optimization in the style of Newton's method. We then give an overview over different types of automatic differentiation and their implementations.

2.1. Non-linear Continuous Optimization

Consider a continuous objective function $f : \mathbb{R}^n \to \mathbb{R}$ to be minimized with respect to a variable vector $\mathbf{x} \in \mathbb{R}^n$. The first- and second-order derivatives of $f(\mathbf{x})$ are the gradient $\mathbf{g}(\mathbf{x}) \in \mathbb{R}^n$ (vector of partial derivatives $\partial f/\partial x_i$) and the Hessian $\mathbf{H}(\mathbf{x}) \in \mathbb{R}^{n\times n}$ (symmetric matrix of partial derivatives $\partial^2 f/\partial x_i \partial x_j$).

Descent methods start from an initial point $\mathbf{x}$ and then iteratively perform update steps $\mathbf{x} \leftarrow \mathbf{x} + s\mathbf{d}$ that strictly decrease the objective value. A descent direction $\mathbf{d} \in \mathbb{R}^n$ can be found by stepping towards the minimum of a quadratic approximation of f around the current point $\mathbf{x}$, via the linear system $\mathbf{Hd} = -\mathbf{g}$ (Newton's method). If f is non-convex at $\mathbf{x}$, $\mathbf{H}$ needs to be replaced by a positive-definite matrix $\mathbf{H}_+$, obtained e.g. via a modification of $\mathbf{H}$ (projected-Newton's method). The step size $s \in \mathbb{R}^{>0}$ is chosen per iteration by sampling successively smaller steps until $f(\mathbf{x} + s\mathbf{d})$ yields sufficient decrease in the objective value (backtracking line search). Iterations stop when a convergence criterion is fulfilled, e.g. when the Newton decrement $\sqrt{-\mathbf{d}^\mathsf{T}\mathbf{g}}$ falls below a threshold. Note that for some $\mathbf{x} \in \mathbb{R}^n$, $f(\mathbf{x})$ needs to be evaluated together with its derivatives $\mathbf{g}(\mathbf{x})$ and $\mathbf{H}(\mathbf{x})$ (once per iteration) and at some $\mathbf{x}$ without its derivatives (once per sample during the line search). For a broader overview and a more thorough treatment of continuous optimization algorithms see e.g. [BV04] or [NW06].

2.2. Automatic Differentiation of Objective Functions

Given a fixed $\mathbf{x} \in \mathbb{R}^n$ and an implementation of $f : \mathbb{R}^n \to \mathbb{R}$, the task of automatic differentiation is to compute $\mathbf{g}(\mathbf{x}) \in \mathbb{R}^n$ and $\mathbf{H}(\mathbf{x}) \in \mathbb{R}^{n\times n}$. We give a short overview of different approaches to this problem and refer to textbooks [GW08, Nau11] or recent reports [Mar19] and tutorials [Sch19] for a broader perspective.

An execution of f can be viewed via its *computation graph*: A directed acyclic graph, in which each node is a scalar value existing at a certain time during program execution (Figure 3). There are n input nodes with values x_i, any number of intermediate nodes, and an output node with value $f(\mathbf{x})$. The set of incoming edges at a node corresponds to an operation producing the node's value based on its arguments. A program execution traverses this graph in a topological ordering, i.e., it performs a single forward pass. Note that in the presence of control flow statements (e.g. branching or loops) the structure of the computation graph itself depends on $\mathbf{x}$.

Automatic differentiation techniques can be seen as attaching additional derivative information to the nodes of the computation graph, such that, after one or multiple passes through the graph, the sought derivatives can be read off. We categorize different approaches along three criteria: (1) the mode of differentiation (forward vs. backward), (2) the time of differentiation (before compilation, at compile time, at run time), and (3) whether or not the computation graph needs to be explicitly represented.

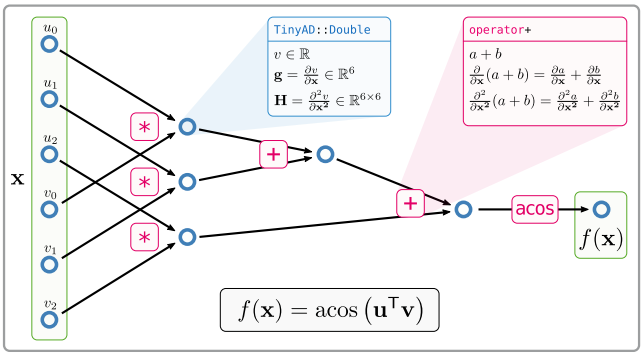

Figure 3: *Computation graph evaluating* $\operatorname{acos}(\mathbf{u}^\mathsf{T}\mathbf{v})$ *with* $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$ *based on the variable vector* $\mathbf{x} \in \mathbb{R}^6$. *Forward-mode differentiation attaches to each node its derivatives with respect to the input* $\mathbf{x}$.

[1] https://github.com/patr-schm/TinyAD

Forward vs. Backward Mode. In *forward mode*, each node carries the derivatives of its value with respect to all input variables. That is, a full gradient vector (and Hessian matrix) is attached to each scalar. As the derivatives of a node can be computed from the derivatives of its predecessors, only a single forward pass is necessary, just as in the original program execution. The final derivatives can then be read from the output node. In contrast, *backward mode* attaches a single scalar to each node: the derivative of the output value $f(\mathbf{x})$ with respect to that node. Since, by the chain rule, this derivative depends on the derivatives of its successor nodes, two passes through the graph are necessary: a forward pass computing the value at each node, followed by a backward pass computing the derivative at each node. Finally, the full gradient vector can be read from the scalar derivatives stored at the input nodes. Second-order derivatives require an additional (e.g. forward) pass [GW08, Ch. 5].

In the case of a scalar objective, which we consider here, asymptotic run time analysis suggests that backward mode is more efficient than forward mode. Already for first-order differentiation, the time complexity of forward mode grows quadratically in the number of inputs, while only linearly in backward mode. This trend persists in the second-order case (cubic vs. quadratic) [Nau11, Ch. 3]. However, in practice, other aspects can outweigh these asymptotic considerations for a sufficiently large range of input dimensions, as we demonstrate in Figure 4. If, in addition, a large problem can be split into small sub-problems with few inputs, as is often the case in geometry processing, we show that forward mode can in fact perform better, while at the same time avoiding the added implementation complexity of backward mode (cf. Section 4).

Differentiation Time. Implementing automatic differentiation via *source code transformation* involves parsing the original source code, symbolically computing derivative expressions, and finally generating derivative code. While this method offers full potential for the optimization of derivative expressions, it is challenging to seamlessly integrate into programming environments [DSJ*22]. Computing derivatives at *compile time*, e.g. via C++ operator overloading or templates, hands the tasks of generating and optimizing derivative code over to the compiler. Alternatively, operator overloading can be used to perform differentiation at *run time*. This offers additional flexibility, e.g. to dynamically track a computation graph, but rules out many performance optimizations as the derivative code is not fully known to the compiler [Sch19].

Computation Graph. Backward-mode differentiation generally requires an *explicit representation* of the computation graph. This could be in form of an abstract syntax tree built by a parser [DSJ*22], a tree built during compile time via expression templates [Hog14], or a tape recorded from a program execution at run time [WG12]. Such explicitly represented computation graphs are often constructed once and then re-used for different inputs $\mathbf{x}$. However, in the presence of variable control flow, i.e. looping and branching based on $\mathbf{x}$, the computation graph can change between function executions and needs to be carefully updated or rebuilt. This often translates into either implementation restrictions on the function f or explicit management of possible branching cases. In contrast, some forward-mode approaches are possible *without* an explicit computation graph, as the control flow of the original program matches the control flow required for differentiation. In other words, instead of tracking derivative expressions, it is sufficient to track derivative values depending on the current input $\mathbf{x}$. In such a setting, variable control flow is naturally supported.

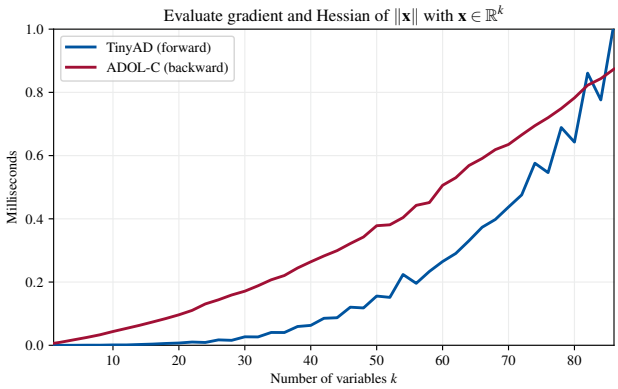

Figure 4: *While, with respect to the number of inputs, backward-mode is asymptotically faster than forward-mode, there is a significant range of small problem sizes for which forward-mode outperforms backward-mode in practice. Here we compute gradients and Hessians of a simple expression (the L2 vector norm). For typical sizes that appear in our application examples, e.g. $k = 6$ and $k = 12$, TinyAD (forward) is faster than ADOL-C (backward) by a factor of 73 and 49, respectively.*

2.3. Automatic Differentiation Software

We focus on methods that support second-order derivatives and compare our approach to three representatives: (1) A concise implementation of forward-mode compile time differentiation, which is part of the Mitsuba renderer [Jak10]. (2) ADOL-C [WG12], a well-established automatic differentiation package in C++ that performs backward-mode differentiation at run time by recording computations on a tape. Active and passive variables, differentiable code sections, and re-taping in case of branching have to be explicitly handled by the user. (3) ACORNS [DSJ*22], a python tool for code transformation in forward or backward mode, which supports a subset of C99 code. It can, with some effort, be integrated into a C++ compiler toolchain, but cannot perform dynamic branching or handle passive variables. The generated code can become prohibitively large in some cases, cf. [DSJ*22, Sec. 3.4].

Many more automatic differentiation packages exist, but often support first-order derivatives only: e.g. Adept [Hog14] (backward, compile time), FastAD [Yan21] (backward, compile time), Enoki [Jak19] (forward or backward, run time), Tapenade [HP13] (forward or backward, code transformation). In the context of machine learning, libraries such as PyTorch [PGM*19] or TensorFlow [AAB*16] (both forward or backward, run time) target computations involving large and typically dense tensors, with a focus on first-order derivatives (although second-order is possible).

Fewer implementations explicitly handle the case of sparse Hessians. ADOL-C [WG12] supports automatic inference of sparsity patterns by solving a graph coloring problem at run time. A recently-described code generator [HTS*22] supports automatic detection of per-element computations and offers sparse second-

order differentiation in backward mode. The authors report striking run time performance, at the cost of limited branching capabilities and a more involved compilation procedure.

Our approach to the dense case is conceptually closest to the one by [Jak10], as we perform compile-time forward-mode differentiation without an explicit computation graph. In the sparse case, we provide a natural interface to express problems via local stencils, removing the need to algorithmically recover sparsity patterns after the fact. Our interface shows similarities to domain-specific languages for sparse computations, such as [KKRK*16], but in contrast stays purely within the realm of the C++ language.

3. TinyAD

We first discuss our second-order automatic differentiation approach for dense, low-dimensional functions $f : \mathbb{R}^k \to \mathbb{R}$ (Section 3.1) and then introduce our interface for the element-wise differentiation of large but sparse problems $f : \mathbb{R}^n \to \mathbb{R}$ (Section 3.2).

3.1. Differentiating Dense Problems

As outlined in Section 2.2, forward-mode differentiation can be achieved by storing at each node of the computation graph not only its value $v \in \mathbb{R}$ but also its derivatives with respect to all inputs $\mathbf{x} \in \mathbb{R}^k$. We also call such a node an *active scalar*, defined as the tuple $(v, \mathbf{g}, \mathbf{H})$ with

$$\mathbf{g} = \frac{\partial v}{\partial \mathbf{x}} \in \mathbb{R}^k \qquad \text{and} \qquad \mathbf{H} = \frac{\partial^2 v}{\partial \mathbf{x}^2} \in \mathbb{R}^{k \times k}.$$

An operation o can access the gradients and Hessians of its arguments and needs to produce the gradient and Hessian of its result. E.g. for a binary operation $o(a, b) = c$ we additionally have to provide expressions for $\partial o / \partial \mathbf{x}$ and $\partial^2 o / \partial \mathbf{x}^2$ to define the map

$$((v_a, \mathbf{g}_a, \mathbf{H}_a), (v_b, \mathbf{g}_b, \mathbf{H}_b)) \mapsto (v_c, \mathbf{g}_c, \mathbf{H}_c).$$

Depending on the operation, this boils down to e.g. applying the first- and second-order chain rule, product rule, quotient rule, etc. The k source nodes of the graph are initialized with $(x_i, \mathbf{e}_i, \mathbf{0})$, where $\mathbf{e}_i \in \mathbb{R}^k$ is the one-hot vector with 1 in the i-th component (since $\partial x_i / \partial \mathbf{x} = \mathbf{e}_i$) and $\mathbf{0} \in \mathbb{R}^{k \times k}$ is the zero-matrix.

We implement active scalars $(v, \mathbf{g}, \mathbf{H})$ in C++ by providing the type `TinyAD::Double` as a replacement for `double`. Each instance of this type carries its own gradient and dense Hessian. We further implement first- and second order derivatives via C++ operator overloading for a broad set of operations (elementary operations, powers, logarithms, trigonometric functions, hyperbolic functions, complex arithmetic, ...), which can also easily be extended by users.

Usage. In the simplest scenario, a user chooses the number of variables k, assigns an index i to each variable, and then performs any number of computations using the available operators:

```
using ADouble = TinyAD::Double<2>; // Scalar type w.r.t. k = 2 variables
ADouble x0(3.5, 0); // Variable with index 0
ADouble x1(5.0, 1); // Variable with index 1
ADouble z = -log(0.5 * sqrt(x0 * x0 + x1 * x1));
```

Derivatives can be read via `z.grad` and `z.Hess` from any active scalar at any time, not only from the final result.

```
// Choose autodiff scalar type for 3 variables
using ADouble = TinyAD::Double<3>;

// Init a 3D vector of active and a 3D vector of passive variables
Eigen::Vector3<ADouble> x = ADouble::make_active({0.0, -1.0, 1.0});
Eigen::Vector3<double> y(2.0, 3.0, 5.0);

// Compute angle and retrieve gradient and Hessian w.r.t. x
ADouble angle = acos(x.dot(y) / (x.norm() * y.norm()));
Eigen::Vector3d g = angle.grad;
Eigen::Matrix3d H = angle.Hess;
```

Figure 5: *Computing derivatives of the angle between a pair of 3D vectors (one variable, one constant). Active scalars are used inside Eigen types and freely mixed with passive scalars. This gives access to a large number of Eigen's vector and matrix operations, which are differentiated by TinyAD on a per-scalar level.*

TinyAD is natively integrated with the Eigen library [GJ*10]. Derivatives are represented as Eigen matrices and `TinyAD::Double` itself can be used as the scalar type within Eigen matrices and vectors. This automatically gives access to differentiating common matrix and vector operations (products, determinants, normalization, inversion, etc.). Furthermore, active (`TinyAD::Double`) and passive (`double`) types can be freely combined in expressions, see Figure 5.

Control Flow. Because no explicit representation of the computation graph needs to be tracked, unrestricted value-dependent control flow is possible. This allows general run-time branching and loop statements based on variables. In Figure 6, we showcase an example that can only be easily implemented due to this flexibility: We evaluate piecewise-linear distortion of a triangulated polygon, whose triangulation connectivity depends on the distortion measure itself. That is, the objective function first chooses the optimal polygon triangulation before computing its distortion in a differentiable way. In our single-pass forward-mode setting this is straightforward to implement, whereas in any backward-mode or expression-based environment, this kind of dynamic branching would only be possible at a higher (implementation or run time) cost.

Performance. The run time performance of TinyAD stems from the fact that all derivative expressions are known at compile time. Derivative values inside `TinyAD::Double` are represented via statically-sized matrices and are subject to code vectorization by Eigen. Moreover, all derivative expressions (involving values, gradients and Hessians) are available for compiler inlining. That is, the C++ compiler has access to all these computations at once and automatically performs major optimizations. Even though backward-mode differentiation is known to be faster for large k, these practical arguments outweigh asymptotic considerations for a sufficient range of k (as we also demonstrate in Figure 4 and Figure 7).

With the tools provided so far it is already possible to differentiate and optimize dense problems of small to moderate size or to integrate TinyAD into other systems that require differentiation of small sub-problems. Besides `TinyAD::Double<k>`, any standard floating point type can be used via e.g. `TinyAD::Scalar<k, float>`, based on precision and run time requirements. If only first-order derivatives are required, a gradient-only mode is available via `TinyAD::Scalar<k, double, false>`. The restriction of having to choose `k` at compile time can be lifted by using `TinyAD::Scalar<Eigen::Dynamic, double>` at a run time cost.

```cpp
// Two variables per polygon vertex
constexpr int k = 2 * n_vertices;

// Objective function to be evaluated with TinyAD::Double<k> or double
template <typename T> T objective(const Eigen::Vector<T, k>& _x)
{
    // Compute optimal triangulation w.r.t. current x
    Eigen::MatrixXi F = optimal_triangulation(_x);

    // Compute objective value w.r.t. optimal triangulation
    T f = 0.0;
    for (int i = 0; i < (int)F.rows(); ++i)
        f += triangle_distortion(_x, F(i, 0), F(i, 1), F(i, 2));
    f += penalty(_x);

    return f;
}

// Init vector x with 2D vertex positions
Eigen::Vector<double, k> x = ...

// Evaluate objective with or without derivatives
TinyAD::Double<k> f_active = objective(TinyAD::make_active(x));
double f_passive = objective(x);
```

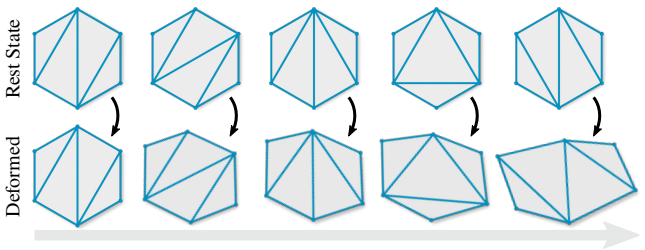

Figure 6: *Polygon deformation with optimal triangulation. The objective function dynamically chooses the least-distorted triangulation among all possible triangulations. Our forward-mode setting without explicit computation graph allows a very compact implementation of such value-dependent control flow. Backward-mode approaches, in contrast, would either require to only differentiate the per-triangle case (adding the burden of manual re-indexing) or to re-build the computation graph in each evaluation (adding a run-time overhead). Here, we also show how to use C++ templates to allow function evaluations with and without derivatives.*

3.2. Differentiating Sparse Problems

In many optimization problems on meshes, a scalar-valued objective function $f : \mathbb{R}^n \to \mathbb{R}$ is *partially separable* [NW06, Ch. 7.4], i.e., it is a sum of *element functions*

$$f(\mathbf{x}) = \sum_j f_j(\mathbf{x}_j)$$

where each f_j only depends on $k \leq n$ entries of $\mathbf{x}$. Per element j, we write the k-dimensional sub-vector of accessed variables as $\mathbf{x}_j = \mathbf{I}_j \mathbf{x}$, where $\mathbf{I}_j \in \{0,1\}^{k \times n}$ removes the unused entries from $\mathbf{x}$.

As differentiation distributes over sums, low-dimensional gradients $\mathbf{g}_j \in \mathbb{R}^k$ and Hessians $\mathbf{H}_j \in \mathbb{R}^{k \times k}$ can be efficiently computed per element (Section 3.1). These are then assembled to form the full n-dimensional gradient and the (usually sparse) Hessian of f via the index mapping $\mathbf{I}_j^\mathsf{T}$:

$$\mathbf{g} = \sum_j \mathbf{I}_j^\mathsf{T} \mathbf{g}_j \qquad \text{and} \qquad \mathbf{H} = \sum_j \mathbf{I}_j^\mathsf{T} \mathbf{H}_j \mathbf{I}_j.$$

We provide the class `ScalarFunction` to model and differentiate such functions. In particular, we offer a convenient way of expressing the sparsity pattern of a problem without having to maintain the typical boilerplate code of an explicit index mapping. The user supplies a set of *variable handles*, a set of *element handles*, and the code to be executed for each element. By accessing a number of variable handles in this code, the user fully defines all element-variable relationships and as such the problem's sparsity pattern.

Depending on the problem, variables and elements might correspond to mesh primitives (e.g. vertices, edges, faces) or to more complex entities (e.g. edge pairs, hinges, patches, etc...). For example, in a planar parametrization problem the variable handles are chosen to be the mesh vertices, with a *variable dimension* of 2. That is, each vertex identifies 2 scalar variables (its u,v-coordinates). The objective could be a distortion energy summed over triangle elements, with an *element valence* of 3, i.e., each triangle depends on 3 vertices, and thus on $k = 6$ scalar variables.

Usage. The user constructs a function by specifying the variable dimension and passing a list of variable handles:

```cpp
auto func = TinyAD::scalar_function<2>(mesh.vertices());
```

Handles can be of one of various supported types from common mesh libraries, or integers. Adding support for additional handle types is easily possible. Next, the user decides on the maximum element valence and passes a list of element handles as well as a lambda function to be evaluated per element:

```cpp
func.add_elements<3>(mesh.faces(), [] (auto& element) { ... });
```

Multiple types of objective terms can be added by calling `add_elements<>()` repeatedly. Inside a per-element lambda function, variables (here vertex positions) can then be accessed via

```cpp
Eigen::Vector2<T> p = element.variables(vh);
```

where `vh` is a variable handle. This function call returns a vector of the variable dimension and records an element-variable relationship (i.e. builds the index map $\mathbf{I}_j$). The scalar type `T` of the returned variables will either be `TinyAD::Double<k>` or `double`, depending on whether derivatives are to be computed or not. Passive variables, constants, or any other kind of data can be freely accessed (via C++ lambda captures) at any point. Finally, the user evaluates the function at a point `x` by calling one of various methods

```cpp
double f = func.eval(x);
auto [f, g] = func.eval_with_gradient(x);
auto [f, g, H] = func.eval_with_derivatives(x);
auto [f, g, H_proj] = func.eval_with_hessian_proj(x);
...
```

which return e.g. the function value `f`, the gradient `g`, the sparse Hessian `H`, or its modification `H_proj` after elementwise positive-definite projection. Elements are evaluated in parallel using OpenMP [DM98]. Figure 2 shows a complete usage example.

Analogously to `ScalarFunction`, we also provide an interface `VectorFunction` for vector-valued functions $f : \mathbb{R}^n \to \mathbb{R}^m$, where each element emits a segment of the result vector instead of a single scalar. We use this interface, for example, in Figure 9 to produce the sparse Jacobian $\mathbf{J} \in \mathbb{R}^{m \times n}$.

3.3. Solving Non-linear Problems

With a way of computing derivatives at hand, basic Newton-style solvers for unconstrained optimization can be implemented rather easily. For convenience, we provide a few simple functions such as `TinyAD::newton_direction()` or `TinyAD::line_search()` to get started (see Figure 2, line 35–42). Alternatively, any external solver can be used; either directly or, for example, through one of the non-linear problem interfaces provided by CoMISo [BZK10]. TinyAD can provide derivatives for all values of $\mathbf{x} \in \mathbb{R}^n$ where $f(\mathbf{x})$ is once or twice differentiable. For discontinuous problems it is the responsibility of the optimization algorithm to only request derivatives where these are well-defined.

Constrained Optimization. We demonstrate the use of penalty terms in Figure 8 and barrier terms in Figure 2, 8, 9, 11. Advanced solvers for problems with non-linear or non-convex constraints (e.g. IPOPT [WB06]) require derivatives of a constraint function $\mathbf{c} : \mathbb{R}^n \to \mathbb{R}^m$, e.g., in form of the (sparse) Jacobian $\mathbf{J} \in \mathbb{R}^{m \times n}$, which TinyAD can easily deliver via its `VectorFunction` interface.

4. Applications and Comparison

In the following, we demonstrate the practical value of TinyAD in four application scenarios. We recap basic usage along a parametrization example (Section 4.1) and compare run time performance to other automatic differentiation approaches. A 3D deformation example (Section 4.2) illustrates working with tetrahedral meshes as well as quick exploration of different energy formulations. In Section 4.3 we replicate a complex non-linear method for frame field optimization, in very few lines of code, and compare against a reference implementation. Finally, Section 4.4 demonstrates the flexibility to implement custom optimization schemes at an example from manifold optimization. Run time evaluation was performed on a desktop computer with Intel Core i7-8700 CPU.

Mesh Data Structures. In Section 4.1 we represent meshes via the OpenMesh [BSBK02] library and in all other examples via a matrix format in the style of libigl [JP*18]. Further code examples using other libraries, e.g. GeometryCentral [SC*19] or polymesh [Tre21], are also available[2]. Support for additional mesh representations can easily be added by users.

4.1. Surface Mesh Parametrization

As a first complete application example, we show how to optimize mapping distortion of a planar mesh parametrization in Figure 2. Given a triangle mesh $\mathcal{M} = (\mathcal{V}, \mathcal{T})$ embedded in $\mathbb{R}^3$, the variable vector $\mathbf{x} \in \mathbb{R}^{2|\mathcal{V}|}$ assigns to each vertex a position in $\mathbb{R}^2$, thus defining a piecewise linear map to the plane. Per-triangle distortion is measured via the symmetric Dirichlet energy

$$f(\mathbf{x}) = \sum_{t \in \mathcal{T}} \text{area}_t \left(\|\mathbf{J}_t(\mathbf{x})\|^2 + \|\mathbf{J}_t(\mathbf{x})^{-1}\|^2 \right)$$

with $\mathbf{J}_t = \begin{bmatrix} \mathbf{b}-\mathbf{a} & \mathbf{c}-\mathbf{a} \end{bmatrix} \begin{bmatrix} \mathbf{b}_r-\mathbf{a}_r & \mathbf{c}_r-\mathbf{a}_r \end{bmatrix}^{-1} \in \mathbb{R}^{2\times 2}$ being the map Jacobian between a rest-pose triangle $\mathbf{a}_r, \mathbf{b}_r, \mathbf{c}_r$ in a tangent space of $\mathcal{M}$ and a variable triangle $\mathbf{a}, \mathbf{b}, \mathbf{c}$ in the plane.

[2] https://github.com/patr-schm/TinyAD-Examples

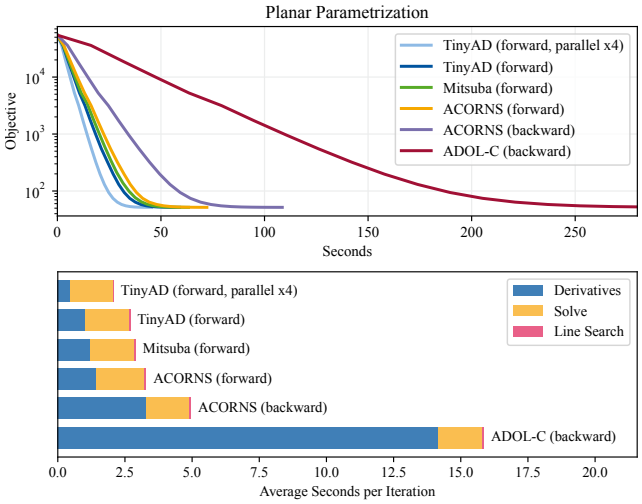

Figure 7: *Run time performance of the parametrization algorithm in Figure 2 on a mesh with 319k triangles. Top: Objective value over wall clock time. TinyAD is slightly faster than other forward-mode approaches and significantly faster than backward-mode alternatives in a single-threaded comparison. Evaluating the set of elements in parallel gives an additional speedup. Bottom: Using TinyAD, the total runtime per iteration is dominated by the linear solve.*

Implementation. In Figure 2 we represent a triangle mesh using the OpenMesh library. A `ScalarFunction` is created by choosing the variable dimension 2 and passing a list of OpenMesh vertex handles. Triangles with element valence 3 are added by passing a range of OpenMesh face handles and a lambda function. Inside this function, `element.handle` provides access to the current triangle (an OpenMesh face handle), and the variable vertex positions $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^2$ are accessed via `element.variables(...)`, which takes an OpenMesh vertex handle as argument. Constant information present in the surrounding scope, e.g. mesh connectivity or the (precomputed) rest shapes of triangles, is simply accessed via lambda captures without any restrictions. Vectors and matrices containing active variables are based on the type `T`, which is instantiated as `TinyAD::Double<6>` when computing derivatives and as `double` during the line search. Run time branching is performed to check if a triangle is flipped or degenerate. The symmetric Dirichlet energy is computed using matrix operations provided by the Eigen library, which in turn invoke per-scalar differentiation by TinyAD. An initial parametrization $\mathbf{x}$ is computed via Tutte's embedding and then optimized via a projected-Newton solver implemented by the user.

While more specialized optimization algorithms for this problem exist, a projected-Newton implementation is a standard baseline for the evaluation of more advanced parametrization methods [RPPSH17, SPSH*17]. Similarly, it can also be the basis for more complex problem settings, e.g. with non-injective initializations [DAZ*20], more involved objectives [SBCK19], or alternating discrete-continuous optimization schemes [LKK*18].

Figure 8: *Implementation of a 3D tetrahedral mesh deformation algorithm. Experimenting with various deformation energies is extremely simple (highlighted box). Soft position constraints are added via a penalty term (top center). A hand-crafted projected-Newton implementation uses a conjugate gradient solver for improved performance (center). Run time is dominated by the linear solve (bottom center).*

Comparison. In Figure 7, we compare the run time of Figure 2 to alternative implementations in which we instead perform the dense per-element differentiation using Mitsuba [Jak10], ACORNS [DSJ*22], and ADOL-C [WG12]. All variants yield the same solution (up to numerics) and spend equal amounts of time for linear solve and line search. Timings of the differentiation step show that for this problem (involving 6×6 Hessians), forward-mode (TinyAD, Mitsuba, ACORNS) significantly outperforms backward-mode (ACORNS, ADOL-C). TinyAD achieves marginally better performance than Mitsuba and ACORNS at significantly reduced implementation effort (cf. Figure 2). While in the ACORNS backward-mode implementation all derivative expressions are known at compile time, the ADOL-C implementation records a tape of the per-element computation (once) and replays it (per triangle, per iteration) at run time, explaining the additional performance difference. We perform comparisons in single-threaded mode and additionally show the (default) parallel version of TinyAD for practical reference.

4.2. Volume Mesh Deformation

Similarly, we show how to deform a tetrahedral mesh $\mathcal{M} = (\mathcal{V}, \mathcal{T})$ in $\mathbb{R}^3$ with respect to position constraints. Based on the per-tetrahedron Jacobian $\mathbf{J} \in \mathbb{R}^{3\times3}$, we implement different elastic deformation energies: e.g. the (exponential) symmetric Dirichlet energy $\exp\left(\|\mathbf{J}\|^2 + \|\mathbf{J}^{-1}\|^2\right)$, the AMIPS energy $\exp\left(\frac{1}{2}\left(\frac{\|\mathbf{J}\|^2}{\det\mathbf{J}} + \frac{1}{2}\left(\det\mathbf{J} + \det\mathbf{J}^{-1}\right)\right)\right)$, and the conformal AMIPS energy $\frac{\|\mathbf{J}\|^2}{\det(\mathbf{J})^{\frac{2}{3}}}$. Manually implementing the 12×12 Hessians of these energies (even with the aid of symbolic differentiation tools) is a non-trivial and error-prone task. Here, we are able to write each of these in a single line (see Figure 8). Additionally, we formulate position constraints for some vertices by adding soft penalty terms.

In this example, the tetrahedral mesh is represented by a "libigl-style" $|\mathcal{V}| \times 3$ matrix of vertex positions and a $|\mathcal{T}| \times 4$ matrix of vertex indices forming one tetrahedron per row. Variable and element handles are row indices of these matrices, created via `TinyAD::range(num_handles)`.

We again optimize this problem via projected-Newton, this time using a (diagonally-preconditioned) conjugate gradient solver for the linear system. In Figure 8, we run our implementation on an example input from [RPPSH17]. For all energies the algorithm converges within 5 to 15 iterations, and the per-iteration run time is dominated by the linear solve.

4.3. Frame Field Optimization

We provide a compact implementation of the Integrable PolyVector Fields algorithm for frame field optimization [DVPSH15]. Given an input frame field (two tangent vectors per face) on a triangle mesh, the method performs a heavy non-linear optimization to guide the field towards a nearby curl-free (and thus locally integrable) field. The formulation avoids explicit matchings between vectors in adjacent triangles by encoding them as the roots of complex polynomials.

The variable vector $\mathbf{x} \in \mathbb{R}^n$, with $n = 4|\mathcal{T}|$, represents two tangent vectors $\alpha, \beta \in \mathbb{C}$ per triangle. Different per-edge objective terms promote smoothness, curl reduction, and order-preservation among adjacent triangles. In addition, per-face terms provide regularization and injectivity guarantees. The final objective function is a sum of squared residuals $f(\mathbf{x}) = \sum_{j=1}^{m} \left(r_j(\mathbf{x})\right)^2$ and is optimized via Gauss-Newton iterations based on the residual vector $\mathbf{r} \in \mathbb{R}^m$ and its Jacobian $\mathbf{J} = \partial\mathbf{r}/\partial\mathbf{x} \in \mathbb{R}^{m\times n}$.

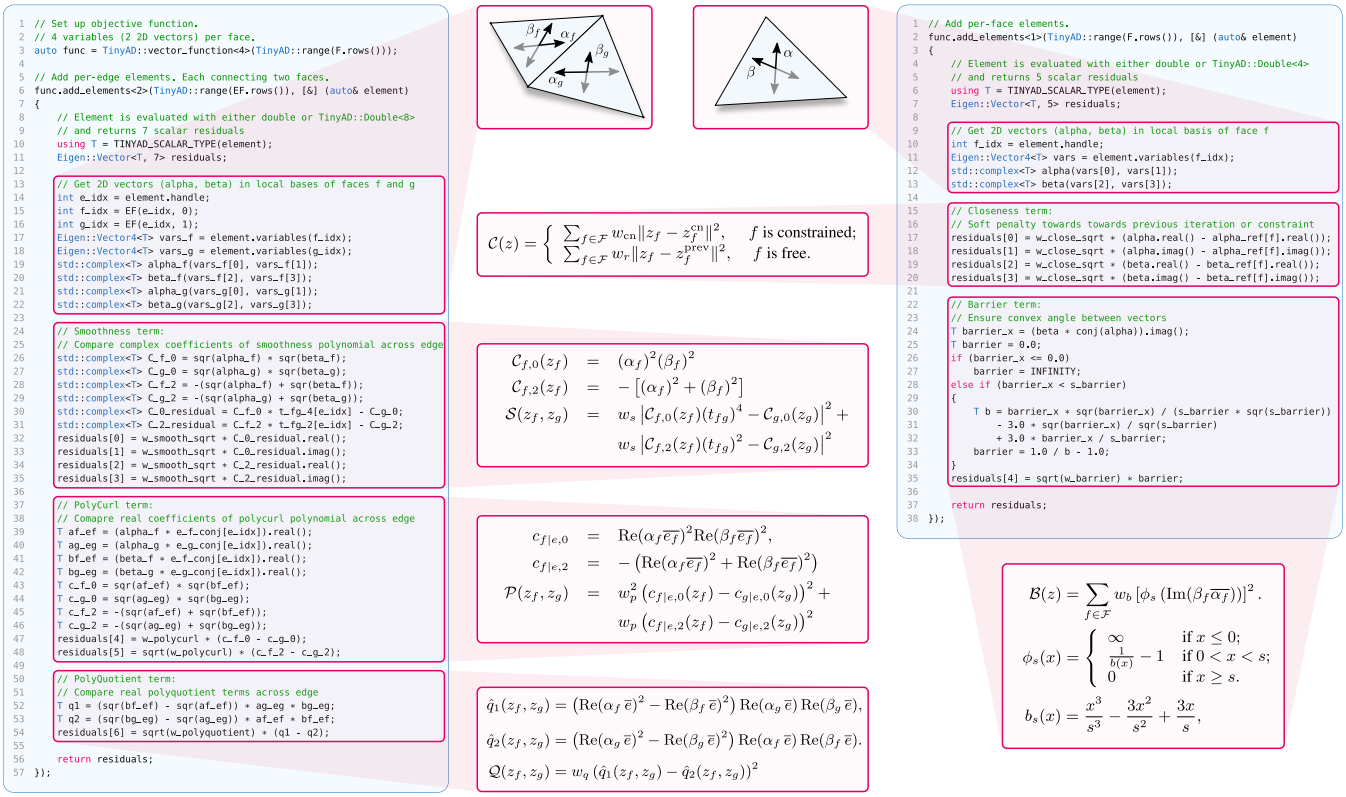

Figure 9: *Complete implementation of the objective function of Integrable PolyVector Fields [DVPSH15]. Being able to express per-edge terms (left) and per-face terms (right) directly in complex arithmetic and without much boilerplate code or index mapping, the implementation looks just like the formulas in the original paper (center). Here, we use* `TinyAD::VectorFunction` *to emit multiple residual terms per element.*

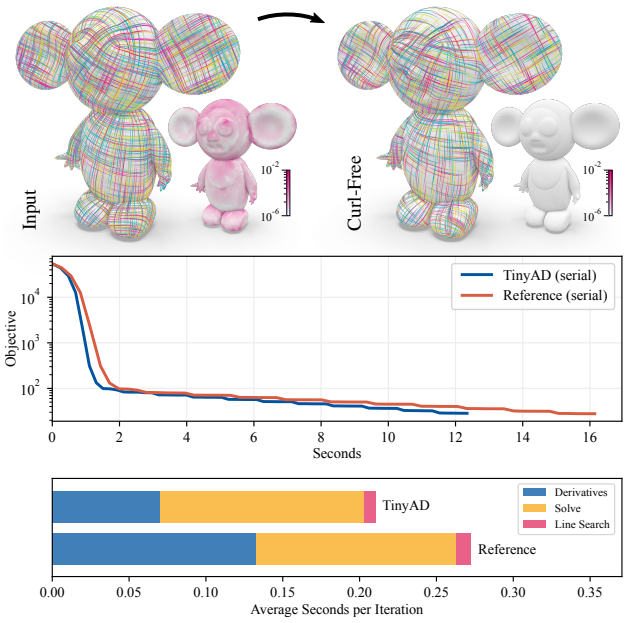

Figure 10: *Results of curl reduction algorithm in Figure 9. Top: Heat map shows the PolyCurl term before and after optimization. Center & bottom: Our significantly simpler implementation via TinyAD is slightly faster than the reference implementation [V*18]. Staircase pattern in the plot is due to a weight decay scheme.*

The authors of [DVPSH15] provide Matlab scripts for symbolic differentiation of the individual terms, spelled out in real arithmetic. In a C++ reference implementation [V*18], significant programming effort is spent on evaluating the resulting expressions as well as assembling and updating the system matrix in each iteration. In contrast, using TinyAD, we are able to implement the objective function directly in C++, express terms in complex arithmetic, and compute derivatives automatically. The resulting code in Figure 9 is very short and closely resembles the formulas in the original paper [DVPSH15]. Moreover, it is self-contained, as neither writing nor running this code depends on external (commercial) tools or switching between software ecosystems. We are convinced that this not only accelerates prototyping, but also fosters replicability and reproducibility in geometry processing research.

In Figure 10 we show that the added implementation convenience does not come at the cost of runtime performance. In fact, automatic differentiation via TinyAD (serial) even outperforms the manual implementation in each iteration. We perform the same Gauss-Newton step and line search strategy as the reference code and obtain the same solution (up to numerics).

4.4. Manifold Optimization

We additionally demonstrate the flexibility to address complex problem setups with an example from manifold optimization. Here, we optimize an injective embedding of a genus-0 triangle mesh on the unit sphere. We enforce the constraint that each vertex stays on

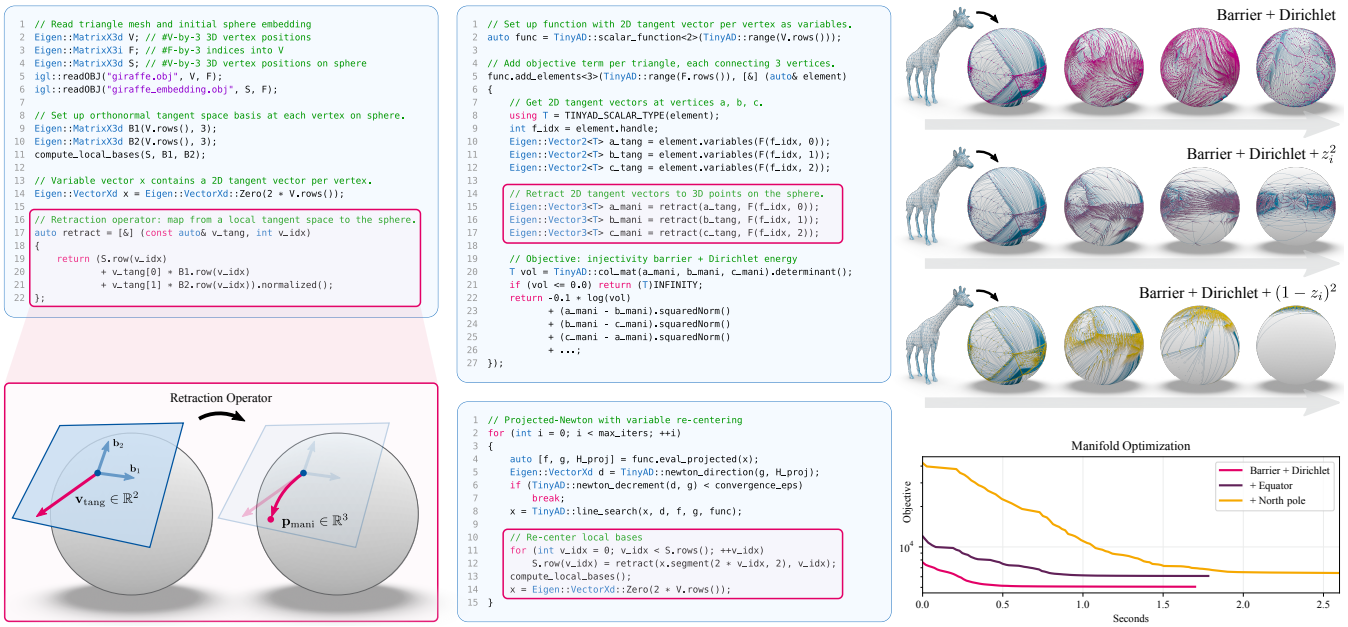

Figure 11: *Implementation of a manifold optimization algorithm, moving vertex positions on the unit sphere. Left: vertex trajectories are parametrized via 2D tangent vectors under a retraction operator. Center top: Per triangle the objective function first applies the retraction operator to its three vertices and then evaluates an energy based on 3D positions. Center bottom: After each iteration, local tangent bases are re-centered at the new vertex positions. Right: Via little adjustments to the objective function, different behaviors can be quickly explored. For illustration purposes we experiment with terms attracting vertices either to the equator or to the north pole.*

the manifold, i.e $\mathbf{v} \in S^2 \subset \mathbb{R}^3$, by parametrizing vertex trajectories via tangent vectors, expressed in local 2D bases. A retraction operator $S^2 \times \mathbb{R}^2 \to S^2$ then traces vertex updates within the curved manifold. After moving to a new position, each vertex receives a new (and arbitrary) tangent space basis. Hence, while the optimization is smooth in the vertex positions on the sphere, the interpretation of the variable vector $\mathbf{x} \in \mathbb{R}^{2|\mathcal{V}|}$ changes in each iteration.

In Figure 11, we show how to implement an objective function as a composition of a retraction operator with a distortion measure. As such, it is automatically differentiated and straightforward to use in a standard Newton-style algorithm. At the end of each iteration, all vertex trajectories are applied, new tangent space bases are computed. Implementations of different retraction operators (e.g. via the exponential map) as well as extensions to other manifolds (e.g. hyperbolic spaces) are possible in the same way.

5. Conclusion

We showed that simple, yet efficient, differentiation of small problems is possible when choosing forward mode (due to its straightforward implementation), compile time differentiation (due to its performance), and avoiding explicit computation graphs (due to branching flexibility). Our interface for per-element differentiation easily transfers these advantages to sparse problems on meshes.

The performance of this approach relies on choosing a static element size: the (maximum) number of variables per element must be known at compile time, which is not necessarily the case for, e.g., per-vertex objectives with dynamic vertex degree. Still, this can often be worked around, e.g., by expressing Laplace-type functions per edge instead of per vertex, by limiting the maximum vertex degree, or potentially by switching to dynamic element size at a run time cost.

We see future potential in providing additional derivative expressions for higher-level operations. For example, matrix expressions are by default differentiated per scalar entry, but more advanced differentiation rules for specific expressions or decompositions can be added by simply overloading functions with TinyAD types. For increased performance it could be worthwhile to also incorporate analytic derivatives of well-known distortion energies, which can then be used as a building block in more complex objectives.

While we focused on the practically most relevant case of first- and second-order derivatives, it is conceptually possible to extend the same approach to e.g. third-order derivatives, by implementing the respective differentiation rules.

Acknowledgements

We thank Anton Florey, Alexandra Heuschling, Dörte Pieper, Joe Jakobi, and Philipp Domagalski for testing and contributing to the development of TinyAD. This work was partially funded by the German Research Foundation within the Gottfried Wilhelm Leibniz programme and partially funded under the Excellence Strategy of the Federal Government and the Länder, as well as by grant IRTG-2379 of the Deutsche Forschungsgemeinschaft (DFG). D. Bommes has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (AlgoHex, grant agreement No 853343). Open access funding enabled and organized by Projekt DEAL.

References

[AAB*16] ABADI M., AGARWAL A., BARHAM P., BREVDO E., CHEN Z., CITRO C., CORRADO G., ET AL.: TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint* (2016). 4

[APL14] AIGERMAN N., PORANNE R., LIPMAN Y.: Lifted bijections for low distortion surface mappings. *ACM Transactions on Graphics 33*, 4 (2014). 1

[BSBK02] BOTSCH M., STEINBERG S., BISCHOFF S., KOBBELT L.: OpenMesh – a generic and efficient polygon mesh data structure. 7

[BV04] BOYD S., VANDENBERGHE L.: *Convex Optimization*. Cambridge University Press, 2004. 3

[BZK10] BOMMES D., ZIMMER H., KOBBELT L.: Practical mixed-integer optimization for geometry processing. In *Proceedings of the International Conference on Curves and Surfaces* (2010). 7

[CBSS17] CLAICI S., BESSMELTSEV M., SCHAEFER S., SOLOMON J.: Isometry-aware preconditioning for mesh parameterization. *Computer Graphics Forum 36*, 5 (2017). 2

[CCS*21] CAMPEN M., CAPOUELLEZ R., SHEN H., ZHU L., PANOZZO D., ZORIN D.: Efficient and robust discrete conformal equivalence with boundary. *ACM Transactions on Graphics 40*, 6 (2021). 2

[DAZ*20] DU X., AIGERMAN N., ZHOU Q., KOVALSKY S. Z., YAN Y., KAUFMAN D. M., JU T.: Lifting simplices to find injectivity. *ACM Transactions on Graphics 39*, 4 (2020). 7

[DM98] DAGUM L., MENON R.: Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering 5*, 1 (1998). 6

[DSJ*22] DESAI D., SHUCHATOWITZ E., JIANG Z., SCHNEIDER T., PANOZZO D.: ACORNS: An easy-to-use code generator for gradients and Hessians. *SoftwareX* (2022). 2, 4, 8

[DVPSH15] DIAMANTI O., VAXMAN A., PANOZZO D., SORKINE-HORNUNG O.: Integrable polyvector fields. *ACM Transactions on Graphics 34*, 4 (2015). 8, 9

[ESBC19] EZUZ D., SOLOMON J., BEN-CHEN M.: Reversible harmonic maps between discrete surfaces. *ACM Transactions on Graphics 38*, 2 (2019). 2

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3. http://eigen.tuxfamily.org, 2010. 3, 5

[GW08] GRIEWANK A., WALTHER A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008. 3, 4

[HAWG08] HUANG Q.-X., ADAMS B., WICKE M., GUIBAS L. J.: Non-rigid registration under isometric deformations. *Computer Graphics Forum 27*, 5 (2008). 1

[Hog14] HOGAN R. J.: Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software 40*, 4 (2014). 2, 4

[HP13] HASCOET L., PASCUAL V.: The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software 39*, 3 (2013). 2, 4

[HTS*22] HERHOLZ P., TANG X., SCHNEIDER T., KAMIL S., PANOZZO D., SORKINE-HORNUNG O.: Sparsity-specific code optimization using expression trees. *ACM Trans. Graph. 41*, 5 (2022). 4

[Jak10] JAKOB W.: Mitsuba renderer, 2010. http://www.mitsuba-renderer.org. 4, 5, 8

[Jak19] JAKOB W.: Enoki: structured vectorization and differentiation on modern processor architectures, 2019. https://github.com/mitsuba-renderer/enoki. 2, 4

[JFH*15] JIANG T., FANG X., HUANG J., BAO H., TONG Y., DESBRUN M.: Frame field generation through metric customization. *ACM Transactions on Graphics 34*, 4 (2015). 1

[JP*18] JACOBSON A., PANOZZO D., ET AL.: libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/. 7

[JSP17] JIANG Z., SCHAEFER S., PANOZZO D.: Simplicial complex augmentation framework for bijective maps. *ACM Transactions on Graphics 36*, 6 (2017). 2

[KGL16] KOVALSKY S. Z., GALUN M., LIPMAN Y.: Accelerated quadratic proxy for geometric optimization. *ACM Transactions on Graphics 35*, 4 (2016). 2

[KKRK*16] KJOLSTAD F., KAMIL S., RAGAN-KELLEY J., LEVIN D. I., SUEDA S., CHEN D., VOUGA E., KAUFMAN D. M., KANWAR G., MATUSIK W., ET AL.: Simit: A language for physical simulation. *ACM Transactions on Graphics 35*, 2 (2016). 5

[LCBK19] LYON M., CAMPEN M., BOMMES D., KOBBELT L.: Parametrization quantization with free boundaries for trimmed quad meshing. *ACM Transactions on Graphics 38*, 4 (2019). 1

[LKK*18] LI M., KAUFMAN D. M., KIM V. G., SOLOMON J., SHEFFER A.: OptCuts: Joint optimization of surface cuts and parameterization. *ACM Transactions on Graphics 37*, 6 (2018). 1, 7

[LPW*06] LIU Y., POTTMANN H., WALLNER J., YANG Y.-L., WANG W.: Geometric modeling with conical meshes and developable surfaces. *ACM Transactions on Graphics 25*, 3 (2006). 1

[LYNF18] LIU L., YE C., NI R., FU X.-M.: Progressive parameterizations. *ACM Transactions on Graphics 37*, 4 (2018). 2

[Mar19] MARGOSSIAN C. C.: A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 9*, 4 (2019). 3

[MGS*21] MAKATURA L., GUO M., SCHULZ A., SOLOMON J., MATUSIK W.: Pareto gamuts: Exploring optimal designs across varying contexts. *ACM Transactions on Graphics 40*, 4 (2021). 1

[MSP*17] MEURER A., SMITH C. P., PAPROCKI M., CERTÍK O., KIRPICHEV S. B., ROCKLIN M., KUMAR A., IVANOV S., ET AL.: SymPy: Symbolic computing in Python. *PeerJ Computer Science 3* (2017). 2

[Nau11] NAUMANN U.: *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2011. 3, 4

[NJJ21] NICOLET B., JACOBSON A., JAKOB W.: Large steps in inverse rendering of geometry. *ACM Transactions on Graphics 40*, 6 (2021). 1

[NW06] NOCEDAL J., WRIGHT S.: *Numerical Optimization*. Springer Science & Business Media, 2006. 3, 6

[PGM*19] PASZKE A., GROSS S., MASSA F., LERER A., ET AL.: PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems 32* (2019). 4

[PTH*17] PORANNE R., TARINI M., HUBER S., PANOZZO D., SORKINE-HORNUNG O.: Autocuts: simultaneous distortion and cut optimization for UV mapping. *ACM Transactions on Graphics 36*, 6 (2017). 1

[RPPSH17] RABINOVICH M., PORANNE R., PANOZZO D., SORKINE-HORNUNG O.: Scalable locally injective mappings. *ACM Transactions on Graphics 36*, 4 (2017). 7, 8

[RS15] RAY N., SOKOLOV D.: On smooth 3D frame field design. *arXiv preprint* (2015). 2

[SAJ20] SELLÁN S., AIGERMAN N., JACOBSON A.: Developability of heightfields via rank minimization. *ACM Transactions on Graphics 39*, 4 (2020). 1

[SBCK19] SCHMIDT P., BORN J., CAMPEN M., KOBBELT L.: Distortion-minimizing injective maps between surfaces. *ACM Transactions on Graphics 38*, 6 (2019). 1, 2, 7

[SC*19] SHARP N., CRANE K., ET AL.: geometry-central, 2019. https://www.geometry-central.net. 7

[SCBK20] SCHMIDT P., CAMPEN M., BORN J., KOBBELT L.: Inter-surface maps via constant-curvature metrics. *ACM Transactions on Graphics 39*, 4 (2020). 2

[Sch19] SCHROEDER C.: Practical course on computing derivatives in code. In *ACM SIGGRAPH Courses*. 2019. 2, 3, 4

[SdGK19] SMITH B., DE GOES F., KIM T.: Analytic eigensystems for isotropic distortion energies. *ACM Transactions on Graphics 38*, 1 (2019). 1

[SFCBCV19] SAGEMAN-FURNAS A. O., CHERN A., BEN-CHEN M., VAXMAN A.: Chebyshev nets from commuting PolyVector fields. *ACM Transactions on Graphics 38*, 6 (2019). 1

[SGC18] STEIN O., GRINSPUN E., CRANE K.: Developability of triangle meshes. *ACM Transactions on Graphics 37*, 4 (2018). 1

[SPSH*17] SHTENGEL A., PORANNE R., SORKINE-HORNUNG O., KOVALSKY S. Z., LIPMAN Y.: Geometric optimization via composite majorization. *ACM Transactions on Graphics 36*, 4 (2017). 7

[Tre21] TRETTNER P.: polymesh, 2021. https://gitlab.vci.rwth-aachen.de:9000/ptrettner/polymesh. 7

[V*18] VAXMAN A., ET AL.: Directional: A library for directional field synthesis, design, and processing, 2018. https://avaxman.github.io/Directional. 9

[WB06] WÄCHTER A., BIEGLER L. T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming 106*, 1 (2006). 2, 7

[WG12] WALTHER A., GRIEWANK A.: Getting started with ADOL-C. In *Combinatorial Scientific Computing*. Chapman-Hall CRC Computational Science, 2012. 2, 4, 8

[Yan21] YANG J.: FastAD: Expression template-based C++ library for fast and memory-efficient automatic differentiation. *arXiv preprint* (2021). 2, 4