Locally Solving Linear Systems for Geometry Processing

vorgelegt von
Dipl. Inf.
Philipp Herholz

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Olaf Hellwich
Gutachter:	Prof. Dr. Marc Alexa
Gutachter:	Prof. Dr. Daniele Panozzo
Gutachter:	Prof. Keenan Crane, Ph.D.

Tag der wissenschaftlichen Aussprache: 5. Juli 2019

Berlin 2020

Abstract

Geometry processing algorithms commonly need to solve linear systems involving discrete Laplacians. In many cases this constitutes a central building block of the algorithm and dominates runtime. Usually highly optimized libraries are employed to solve these systems, however, they are built to solve very general linear systems. I argue that it is possible to create more efficient algorithms by exploiting domain knowledge and modifying the structure of these solvers accordingly. In this thesis I take a first step in that direction. The focus lies on Cholesky factorizations that are commonly employed in the context of geometry processing. More specifically I am interested in the solution of linear systems where variables are associated with vertices of a mesh. The central question is: Given the Cholesky factorization of a linear system defined on the full mesh, how can we efficiently obtain solutions for a local set of vertices, possibly with new boundary conditions? I present methods to achieve this without computing the value at all vertices or refactoring the system from scratch. Developing these algorithms requires a detailed understanding of sparse Cholesky factorizations and modifications of their implementation. The methods are analyzed and validated in concrete applications. Ideally this thesis will stimulates research in geometry processing and related fields to jointly develop algorithms and numerical methods rather than treating them as distinct blocks.

Zusammenfassung

Algorithmen im Feld der Geometrieverarbeitung benötigen in vielen Fällen die Lösung linearer Gleichungssysteme, die im Zusammenhang mit diskreten Laplace Operatoren stehen. Häufig stellt dies eine zentrale Operation dar, die die Laufzeit des Algorithmus maßgeblich mitbestimmt. Normalerweise werden dafür hoch optimierte Programmbibliotheken eingesetzt, die jedoch für sehr allgemeine lineare Systeme erstellt wurden. In dieser Arbeit schlage ich vor, effizientere Algorithmen zu konstruieren, indem die Struktur dieser Programmbibliotheken modifiziert und ausgenutzt wird. Der Fokus liegt hierbei auf der Cholesky Zerlegung, die im Kontext der Geometrieverarbeitung häufig Verwendung findet. Dabei interessiere ich mich besonders für Teile der Lösung linearer Gleichungssysteme, die zu einer lokalisierten Region auf dreidimensionalen Dreiecksnetzen gehören. Die zentrale Frage dabei lautet: Angenommen die Cholesky Zerlegung eines linearen Gleichungssystems, definiert auf dem gesamten Netz, ist gegeben. Wie kann man die Lösung an einer kleinen lokalisierten Menge von Knotenpunkten, möglicherweise unter neu gewählten Randbedingungen, effizient bestimmen? In dieser Arbeit zeige ich Wege auf, wie dies, ohne die Lösung an allen Knotenpunkten zu bestimmen oder die Zerlegung komplett neu zu berechnen, erreicht werden kann. Dazu ist es notwendig, die Funktionsweise von Methoden zur Cholesky Zerlegung dünnbesetzter Matrizen zu analysieren, um sie dann in geeigneter Weise zu modifizieren. Die Methoden werden anhand konkreter Anwendungsbeispiele analysiert. Im Idealfall kann diese Dissertation zukünftige Forschung im Bereich der Geometrieverarbeitung und verwandter Felder stimulieren, Algorithmen und numerische Methode gemeinsam zu entwickeln und sie nicht mehr als unabhängige Teile zu verstehen.

Acknowledgments

I am deeply grateful to Marc Alexa who gave me the chance to pursue the journey that lead to this thesis. Doing a PhD is a challenging task, scary at times but always exciting. In many situations it was the continued support and trust that Marc put in me that helped me push through all challenges I was facing. I learned from him how to conduct science and ask the right questions. Instead of just putting pressure on me he was always able to motivate with his genuine excitement for the field. From day one he met me on eye level and created an atmosphere in which anything could be discussed and questioned.

Besides my advisor many people shaped me as a researcher. I was lucky enough to spend six months at MIT with Wojciech Matusik before starting my PhD. He gave me the chance to experience the research environment in one of the worlds most prolific universities. In 2015 I was invited back to Cambridge for an internship at Disney Research under the supervision of David Levin who helped me broaden the scope of my research.

Over the years many colleagues in the Computer Graphics group at TU Berlin helped me on a daily basis. Many discussions and encouragement made working there a great experience. It was always extremely helpful to talk to colleagues that already went through the process I was in. Christian Lessig and Jan-Eric Kyprianidis took their time to guide me through the academic world and where always extremely helpful, especially in deadline times. When David Lindlbauer joined the group in late 2014 I could not know that I would not only meet a new colleague but, together with Alexandra Ion, new friends. Their advise and encouragement

exceeding the academic domain were indispensable during the years.

I am indebted to my parents Christiane and Paul and my brother Christoph for their continued unconditional support. Lisa supported me relentlessly at all times and I am thankful for her loving encouragement. I am grateful for having a close friend like Eugen to talk to about anything at any time.

Contents

1

Introduction

Geometry processing emerged as a field within computer graphics in the last decades. It covers a diverse range of topics like surface reconstruction from point samples, parameterization and mesh deformation with applications in rapid prototyping, interactive mesh modeling, simulation and animation. Being at the intersection of topics from pure mathematics like topology and differential geometry, and ones from computer science like numerical analysis and computer graphics, the field poses unique questions, both challenging and fascinating.

One particular direction of research in geometry processing is concerned with the discretization of concepts from differential geometry on surface meshes. They enable the generalization of tools from signal processing, which are well established in the context of image processing, to discrete geometry. This endeavor is considerably more difficult than equivalent problems in signal processing of image or audio data. In image processing the signal can be represented as data on a highly

regular domain, enabling the use of tools like the fast Fourier transform, filtering and multigrid methods. Moreover, distances and angles are easily computed. In the case of triangle meshes, however, the data is represented as a set of vertices in 3-space along with a set of polygons connecting the vertices. This means the signal is the domain itself, so that the direct application of algorithms from the signal processing toolbox is not possible. A key to deal with this issue is the construction of a proper discretization of the Laplace-Beltrami operator, or Laplacian for short. A surprising number of algorithms in geometry processing make use of this operator in one form or another. As a generalization of the second derivative to surfaces, it allows to define the notions of frequency and eigenmodes in direct analogy to Fourier analysis of one- or multidimensional signals defined over $\mathbb{R}^d$. This enables filtering of surface signals, most prominently for smoothing. From a more geometric point of view, the Laplace-Beltrami operator can be used to compute mean curvature.

Since the Laplace-Beltrami operator is linear, its discretization can be represented as a matrix. In the context of geometry processing this matrix is usually part of a linear system that needs to be solved. In many cases this step dominates the runtime of algorithms and highly optimized off-the-shelf solvers are employed. The idea of this thesis is to open the black boxes these solvers represent. As a first step in this direction, properties of Cholesky factorization algorithms are exploited to design algorithms that are significantly faster and more robust. The main focus lies on localization, that is, solving for the variables of a local surface patch only, possibly while introducing new boundary conditions. The contributions of this thesis fall into three categories:

- The *definition* of a discrete Laplace-Beltrami operator retaining as many properties from the smooth setting as possible by modifying the geometry.
- Efficient *numerical methods* to solve local linear systems involving the discrete Laplace-Beltrami operator.
- Concrete *algorithms* employing the developed numerical methods in the context of heat flow.

In the following I give a brief outlook on the key contributions of this thesis.

1.1 Discrete Laplacians

There are different ways to discretize the Laplace-Beltrami operator from continuous differential geometry on surface meshes. Generally one wants to retain as many properties of the operator from the smooth setting as possible.

Wardetzky et al. [107] proofed that a *perfect* discrete Laplace Operator can not exist in the sense that the construction guarantees all properties of the continuous Laplacian, like symmetry and positive semi-definiteness, for every triangle mesh. This explains the variety of discretization available; depending on the application one has to choose an operator that possesses properties important for that setting. In Chapter 2 I propose a change of perspective and introduce an algorithm that constructs a perfect Laplace operator based on an arbitrary mesh by slightly changing vertex positions if necessary. I argue that mathematical properties of the Laplacian are, in many situations, more important than keeping the exact mesh geometry. Another way of looking at the technique is to associate a Laplacian of a nearby mesh to some input geometry if a perfect one does not exist.

1.2 Localized linear systems

The solution of linear systems involving discrete Laplacians is at the core of many geometry processing algorithms. Linear systems have to be solved once, e.g. to compute a discrete conformal parameterization [26], or several times in the context of physical simulations or Newton based optimization approaches. In many cases, solving linear systems poses, together with the assembly of the Hessian, the main computational burden, making the choice of linear solver crucial. Fortunately, the system matrix is in many cases sparse, positive (semi-)definite and symmetric so that a sparse Cholesky factorization can be constructed leading to efficient linear solves, possibly with multiple right-hand sides. A crucial property of this factorization is the fact that a sparse system matrix usually leads to a sparse Cholesky factor. If this would not be the case storage and runtime requirements

would hinder the use of this technique. Even though the solution of linear systems is a fundamental task, implementations of geometry processing algorithms usually rely on off-the-shelf solvers that are highly efficient but are designed for much more general problems. The systems we are interested in usually have a very distinct structure. In particular, the sparsity patterns of system matrices are related to the combinatorics of an embedded triangle mesh. Moreover, if we are interested in the exact solution of a linear system given its factorization in a few variables only, we have to solve for all of them and then extract the required values. The situation is even more complicated if we also want to impose new boundary conditions for a set of variables. In this case the factorization has to be recomputed. One particular example is mesh deformation. Usually only a small portion of the mesh should be manipulated at a time and the rest should stay fixed. Introducing this kind of boundary conditions makes the factorization of a new system matrix necessary whenever a new region of interest is defined. Even though this matrix is very similar to the original one, for which a factorization exists, the factorization is done from scratch. In Chapter 5 I introduce a method that reuses information from the original factorization in order to compute the new factorization significantly faster. For the case of a local system that does not impose new boundary conditions I present in Chapter 4 a technique that uses selective back substitution to compute the required values without determining all others.

1.3 Applications

I introduce a set of algorithms that benefit from the presented localization techniques, demonstrating that they can be an enabler for practical algorithms.

Building on recent work by Crane et al. [17] we use heat diffusion on meshes to measure distances. The idea is to discretize the common heat diffusion PDE involving the Laplacian using a discrete Laplacian. Varahdan [103] proved that the solution of the heat diffusion equation is directly related to distance. With a notion of distance we can evaluate Voronoi diagrams. In Chapter 3 I present an algorithm using this idea to compute approximate, well-balanced (centroidal) Voronoi diagrams. This technique is among the only ones operating purely intrinsic in the

sense that it does not rely on extrinsic distances induced by the specific embedding of the mesh. Since distance information is relevant only close to the heat source in this application, I use approximate sparse localization techniques to implement the algorithm. In Chapter 6 I demonstrate that techniques similar to those used for heat flow based geodesic distance computations can be employed to compute the direction of the geodesic connecting a given point to a set of neighboring vertices. This enables heat based computation of logarithmic maps on meshes which can be used to define local coordinate systems on the surface in the vicinity of a given point.

1.4 Contributions

The contributions of this thesis fall into three categories. Chapter 2 which introduces the construction of a discrete Laplace operator. Chapters 3 and 6 which introduce concrete algorithms involving discrete Laplacians. Chapters 4 and 5 deal with effective strategies to solve linear systems by modifying Cholesky factorizations. In more detail, they are given by:

Chapter 2: A construction of a discrete Laplace-Beltrami operator enforcing a set of properties from continuous differential geometry, possibly by altering the input geometry.

Chapter 3: An algorithm for the computation of intrinsic centroidal Voronoi diagrams based on heat diffusion.

Chapter 4: A method to locally evaluate the solution of a linear system defined on the surface without the need to compute the solution vector in its entirety.

Chapter 5: An algorithm to update the factorization of linear systems to accommodate changes in a mesh. In particular changing boundary conditions for mesh deformation or parameterization can be incorporated without the need of an expensive refactorization.

Chapter 6: An algorithm to compute exponential maps based on heat diffusion, extending previous approach for computing geodesic distances be-

tween points by the computation of the angle of geodesics relative to a fixed direction in the tangent plane.

1.5 Publications

The presented algorithms have all been published in peer-reviewed journals and were presented at international conferences. I am greatly indebted to my co-authors without whom this work would not have been possible. Therefore I will use the plural form "we" when referring to the contributions in the individual chapters.

- Chapter 2 was published and presented as: Philipp Herholz, Jan Eric Kyprianidis and Marc Alexa. Perfect Laplacians for Polygon Meshes. In *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Geometry Processing)*, volume 34, pages 211–218, Aire-la-Ville, Switzerland, 2015. Eurographics Association. doi:10.1111/cgf.12709.

- Chapter 3 was published and presented as: Philipp Herholz, Felix Haase and Marc Alexa. Diffusion Diagrams: Voronoi Cells and Centroids from Diffusion. In *Computer Graphics Forum (Proceedings of Eurographics)*, volume 36, pages 163–175, Aire-la-Ville, Switzerland, 2017. Eurographics Association. doi:10.1111/cgf.12556.

- Chapter 4 was published and presented as: Philipp Herholz, Timothy A. Davis and Marc Alexa. Localized solutions of sparse linear systems for geometry processing. © Owner/Author | ACM 2017. This is the author's revised and adapted version. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics (Proceedings of Siggraph Asia)*, volume 36, number 6, pages 183:1–183:8, New York, USA, 2017. ACM Press. doi:10.1145/3130800.3130849.

- Chapter 5 was published and presented as: Philipp Herholz and Marc Alexa. Factor Once: Reusing Cholesky Factorizations on Sub-Meshes. © Owner/Author | ACM 2018. This is the author's revised and adapted

version. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics (Proceedings of Siggraph Asia)*, volume 37, number 6, pages 230:1–230:9, New York, USA, 2018. ACM Press. doi:10.1145/3272127.3275107.

- Chapter 6 was published and presented as: Philipp Herholz and Marc Alexa. Efficient Computation of Smoothed Exponential Maps. In *Computer Graphics Forum*, volume 38, number 6, pages 79–90, Aire-la-Ville, Switzerland, 2019. Eurographics Association. doi:10.1111/cgf.13607.

1.6 Notation

The central object of interest in this thesis are surface meshes. They can be thought of as discretizations of 2-manifolds (possibly with boundaries). Because this thesis exclusively deals with surface meshes, the short form "meshes" will also be used. A surface mesh is denoted as $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ or $\mathcal{M} = (\mathcal{V}, \mathcal{F})$ where $\mathcal{V} = (0, 1, 2, \dots, n_v - 1)$ represents a set of vertices. The set of edges $\mathcal{E} = ((i_0, j_0), (i_1, j_1), \dots)$ consists of 2-tuples indexing vertices in $\mathcal{V}$. Similiary faces are represented as a set of n_f tuples $\mathcal{F} = ((i_0, j_0, k_0, \dots), (i_1, j_1, k_1, \dots), \dots)$ each of which indexes vertices of that face in counterclockwise order.

The position of the i-th vertex will be denoted by $\mathbf{v}_i \in \mathbb{R}^d$, the *embedding* of a mesh can therefore be thought of as a map $\mathcal{V} \to \mathbb{R}^d,\ i \mapsto \mathbf{v}_i$ or, alternatively, as a matrix $\mathbf{V} \in \mathbb{R}^{n_v \times 3}$. Each edge (i, j) has an associated edge vector $\mathbf{e}_{ij} = \mathbf{v}_j - \mathbf{v}_i$ and each triangular face (i, j, k) an associated area weighted surface normal

$$\mathbf{a}_{ijk} = \frac{1}{2}(\mathbf{v}_i - \mathbf{v}_k) \times (\mathbf{v}_j - \mathbf{v}_k) \tag{1.1}$$

and area $a_{ijk} = ||\mathbf{a}_{ijk}||$.

Because meshes will be used as to discretize 2-manifolds, the set of possible polygon lists $\mathbb{N}^{n_f \times r}$ needs to be further restricted. In particular I make the following assumptions if no further restrictions are explicitly stated:

- The edges of each polygon form simply connected loops.

- If two faces share an edge both polygons are consistently oriented, i.e. if (k, l) is an edge of the first face then (l, k) is an edge of the second.
- All vertices $\mathcal{N}_i$ connected to a vertex i are connected by a set of edges forming a simple polygon.

Bold capital letters ($\mathbf{A}, \mathbf{B}, \mathbf{C} \ldots$) will be used to refer to matrices while small bold letters ($\mathbf{a}, \mathbf{b}, \mathbf{c} \ldots$) denote vectors and small regular letters (a, b, c) represent scalars. The entries of a matrix $\mathbf{A}$ or a vector $\mathbf{b}$ are referred to as a_{ij} and b_i, respectively.

2

Perfect Laplacians

2.1 Introduction

Discrete approximations of the Laplace-Beltrami operator are at the heart of many mesh processing techniques, such as representation [93], deformation [7], spectral processing [58], or descriptors [10]. For triangle meshes the cotan Laplacian [25, 79] is a common choice because it enjoys many desirable properties. However, as discussed by Wardetzky et al. [107], discrete Laplace operators cannot be *perfect* in the sense that they may fail to possess all properties of their smooth counterpart for an arbitrary triangle mesh. We recall the desired properties in Section 2.2 and explain why they are important for mesh processing applications.

The notion of perfect Laplace operators can be linked to force networks in equilibrium that only pull vertices along edges. This viewpoint allows the characterization of triangle meshes that admit perfect Laplace operators, namely, so called

regular or *weighted Delaunay* meshes. Vertex weights can be used to generalize the cotan Laplacian [39, 40], and for any weighted Delaunay mesh there exists a perfect Laplace operator of this form [41].

The situation is far less developed for general polygon meshes, i.e., meshes with face degrees larger than three. Alexa and Wardetzky [1] provide a construction that reduces to the cotan-Laplace for triangle meshes, but it is unclear for which meshes this operator is perfect. More importantly, there is no obvious generalization for weighted meshes, and therefore the question of existence of perfect Laplace operators for polygon meshes is open. Using the connection to orthogonal duals or force networks we are able to close this gap and generally describe (planar) meshes that admit perfect Laplacians (see Section 2.4).

Our main contribution is an algorithm that generates perfect Laplace operators for *any* polygonal mesh *without changing the combinatorics* but possibly *changing the embedding*. The main ideas for this algorithm are as follows:

1. We identify a perfect Laplace operator for a mesh with given combinatorics with a unique embedding by fixing the boundary. This defines a mapping from the coefficients of the Laplace operator to the edge lengths.

2. By analogy to force networks, we derive a simple update rule for the coefficients such that the edge lengths converge to the desired ones (if possible).

We describe and discuss this algorithm in Section 2.3. An important point of the algorithm is that it is based only on defining boundary conditions and measuring edge lengths of the embedding and thus has a natural extension to (possibly closed) non-planar meshes embedded in higher dimension.

We demonstrate the construction of perfect Laplace operators for polygon meshes by means of self-supporting surfaces with polygonal tiles as well as by computing parameterizations in Section 2.6 and discuss the properties of our construction, in particular in relation to other approaches, in Section 2.7.

2.2 Discrete Laplace operators

The smooth Laplace-Beltrami operator has several properties that naturally translate into properties of a discrete analogue. Loosely following Wardetzky et al. [107], we call a discrete Laplace operator *perfect* if it is locally defined, symmetric, non-negative, affinely invariant (the constant functions are in the kernel), and has linear precision. In the following, we detail these properties and discuss implications.

Let $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ be a polygon mesh. A *(discrete) Laplace operator* $\mathbf{L}$ on $\mathcal{M}$ is defined by symmetric coefficients $\omega_{ij} = \omega_{ji}$ associated to each oriented edge $(i, j) \in \mathcal{E}$, acting on real-valued functions $\mathbf{u} : \mathcal{V} \mapsto \mathbb{R}$, $i \mapsto u_i$ by:

$$(\mathbf{Lu})_i = \sum_{(i,j)\in\mathcal{E}} \omega_{ij}(u_j - u_i)\,. \tag{2.1}$$

This definition identifies all discrete Laplacians that are local and symmetric in the sense of Wardetzky. Being defined on the differences of vertex values, the constant functions are always in the kernel, making the operator affinely invariant.

Non-negativity A Laplace operator is said to be non-negative, if all its coefficients are non-negative and if there is a spanning tree consisting of edges with positive coefficients. In combination with symmetry, this property ensures that (i) the operator is positive semi-definite and (ii) harmonic functions (with respect to the operator and appropriate boundary conditions) obey the maximum principle. This principle states that harmonic functions take their extremal values at the boundary. In the context of our work (and more generally for the application of computing parameterizations) it ensures that Laplacians can be used to obtain embeddings (with convex boundary) of planar graphs [102].

Linear precision In the smooth setting, the Laplace-Beltrami operator $\mathbf{L}$ applied to the embedding of a smooth 2d manifold yields the area gradient [27]. In particular, it vanishes on planar domains. Hence, in the discrete setting we ask that

$(\mathbf{Lu})_i = 0$ if i is an interior vertex and all edges $(i,j) \in \mathcal{E}$ are contained in a plane. Linear precision implies that mean curvature flow modifies the mesh only in normal direction (i.e., there is no tangential smoothing) and parameterization based on the Laplacian preserves meshes in the plane.

Force networks For symmetric Laplace operators of the form in Equation (2.1), the consequences of linear precision and non-negativity can be made more intuitive by considering the (planar) mesh as a force network [70]: each edge is a spring pulling or pushing the incident vertices together or apart from each other. In this picture, the coefficients ω_{ij} take the role of spring constants.

Consider the mesh to be embedded with vertex positions $\mathbf{v}_0, \mathbf{v}_1, \ldots$ and edges $\mathbf{e}_{ij} = \mathbf{v}_j - \mathbf{v}_i$. Then Hook's law takes the form

$$\mathbf{f}_{ij} = \omega_{ij}\, \mathbf{e}_{ij}\,, \tag{2.2}$$

describing the force vector $\mathbf{f}_{ij}$ acting on the spring (i,j). Comparing this to Equation (2.1), we see that applying the Laplace operator to the vertex positions yields the sum of the forces for each node. Hence, linear precision means that forces cancel in each interior node or, in other words, the force network is in equilibrium. The sign of the coefficients ω_{ij} distinguishes between pulling or pushing forces. If all coefficients are positive, vertices are in equilibrium and the springs along all edges are pulling.

No free lunch Based on the force network interpretation of Laplacians, Wardetzky et al. [107] prove that there are triangle meshes without a discrete Laplacian of the form in Equation (2.1) obeying both linear precision and non-negativity (see Figure 2.2 for an example). However, some applications call for a perfect Laplacian: parameterization techniques based on the Laplacian, for example, require an injective mapping; yet this is only guaranteed if the coefficients are non-negative (and the boundary is fixed in convex position). Asking additionally that planar meshes are mapped to themselves requires linear precision. See Section 2.6 for details on parameterization. Further applications include the computation of self

supporting surfaces (Section 2.6) and the fitting of discrete minimal surfaces with fixed convex boundaries.

2.3 Method

In this section, we derive an algorithm for constructing a perfect Laplace operator for a given mesh. As not all meshes admit such an operator, the mesh vertices may be perturbed by the algorithm. The idea of our approach for planar meshes can be intuitively described in terms of force networks: starting from an arbitrary set of positive spring constants, we compute vertex positions by fixing the boundary. Then we compare the resulting edge lengths to the original ones. If an edge is too short we lower the spring constant, yet keeping it positive. If the edge is too long we increase the spring constant.

In the following we describe this idea rigorously, providing a principled way for updating the coefficients. By introducing different boundary conditions, we can extend this idea to meshes embedded into $\mathbb{R}^3$. An analysis of the steady state yields a characterization of the solution.

2.3.1 Planar polygon meshes

Let $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ be a planar 3-connected polygon mesh with single boundary $\partial\mathcal{M} \subset \mathcal{V}$, and let the geometry be given by vertex positions $\hat{\mathbf{V}} \in \mathbb{R}^{n_v \times 2}$, where the positions of vertices appear as row vectors $\hat{v}_i$. We assume that the boundary vertices are in strictly convex position. One may augment the mesh with an outer boundary in the spirit of [57] in order to relax this condition.

Let the coefficients of the Laplace operator and the edge lengths at the k-th iteration of the algorithm be given by

$$\omega_{ij}^{(k)} > 0 \quad \text{and} \quad \left\| \mathbf{e}_{ij}^{(k)} \right\|. \tag{2.3}$$

We start with $\{\omega_{ij}^{(0)} = 1\}$ (or any other positive choice). Then we repeatedly embed the mesh with the given Laplace operator and update the coefficients. The updates preserve positivity of the coefficients, ensuring their positivity for all iter-

ations. This algorithm is summarized as pseudo code in Algorithm 1. The details are discussed in the remainder of this section.

Embedding Since we assume that the polygon mesh $\mathcal{M}$ has a convex boundary, a solution for the equilibrium state of the spring network can be obtained by solving Laplace's equation given by the coefficients ω_{ij} subject to the constraint that the boundary $\{\hat{\mathbf{v}}_i : i \in \partial\mathcal{M}\}$ is held fixed. More specifically, at each iteration k, we solve

$$\boldsymbol{\Omega}^{(k)}\mathbf{V}^{(k)} = \mathbf{b} \tag{2.4}$$

with

$$\Omega_{ij}^{(k)} = \begin{cases} \omega_{ij}^{(k)} & (i,j) \in \mathcal{E},\ i \notin \partial\mathcal{M} \\ -\sum_j \omega_{ij}^{(k)} & i = j,\ i \notin \partial\mathcal{M} \\ 1 & i = j,\ i \in \partial\mathcal{M} \\ 0 & \text{otherwise} \end{cases} \qquad \mathbf{b}_i = \begin{cases} \begin{pmatrix} 0 & 0 \end{pmatrix} & i \notin \partial\mathcal{M} \\ \hat{\mathbf{v}}_i^{\mathsf{T}} & i \in \partial\mathcal{M} \end{cases}$$

Since the coefficients are non-negative by construction, the resulting vertex geometry $\mathbf{V}^{(k)}$ is a *unique embedding*. This follows from Tutte's spring-embedding theorem [102] and its extension by Floater [33]. In particular, uniqueness implies that we have a mapping

$$\omega_{ij} \mapsto \|\mathbf{e}_{ij}\| \tag{2.5}$$

from the Laplace operator coefficients to edge lengths. Note that this mapping is invariant to multiplying all ω_{ij} with a constant positive scalar.

Update rule We want to update the coefficients $\omega_{ij}^{(k)}$ such that $\|\mathbf{e}_{ij}^{(k)}\|$ gets closer to the edge length $\|\hat{\mathbf{e}}_{ij}\|$ of the input mesh. Since the coefficients are positive, we can derive from Hook's law that

$$F_{ij} = \|\mathbf{f}_{ij}\| = \omega_{ij}^{(k)} \left\|\mathbf{e}_{ij}^{(k)}\right\| . \tag{2.6}$$

Algorithm 1: Compute perfect Laplacian

Input: A mesh $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$, coordinates $\hat{\mathbf{V}}$.
Output: Coordinates $\mathbf{V}$, coefficients ω_{ij}.

$\omega_{ij} \leftarrow 1$
for $k = 1, \ldots,$ *max_iters* **do**
  assemble $\boldsymbol{\Omega}^{(k)}$
  solve $\boldsymbol{\Omega}^{(k)}\, \mathbf{V}^{(k)} = \mathbf{b}$
  $\omega_{ij}^{(k+1)} \leftarrow \omega_{ij}^{(k)} \dfrac{\left\|\mathbf{v}_j^{(k)} - \mathbf{v}_j^{(k)}\right\|}{\left\|\hat{\mathbf{v}}_j - \hat{\mathbf{v}}_i\right\|}$
  $\omega_{ij}^{(k+1)} \leftarrow \omega_{ij}^{(k+1)} / \sqrt{\sum_{ij} \left(\omega_{ij}^{(k+1)}\right)^2}$
  if $\left\|\mathbf{v}^{(k)} - \mathbf{v}^{(k-1)}\right\| < \varepsilon$ **then break**
return $\mathbf{V}^{(k)}, \omega_{ij}^{(k)}$

Assuming that forces stay constant for small changes in the spring constants suggests the following update for the spring constants:

$$\omega_{ij}^{(k+1)} = \frac{F_{ij}}{\left\|\hat{\mathbf{e}}_{ij}\right\|}\,. \tag{2.7}$$

Combining Equations (2.6) and (2.7) eliminates the dependence on forces and yields our simple multiplicative update rule for the coefficients:

$$\omega_{ij}^{(k+1)} = \omega_{ij}^{(k)} \frac{\left\|\mathbf{e}_{ij}^{(k)}\right\|}{\left\|\hat{\mathbf{e}}_{ij}\right\|}. \tag{2.8}$$

After updating all coefficients, we scale them uniformly to avoid them to become arbitrarily small or large, potentially causing numerical problems. Note that the fraction of current to original edge lengths is always positive so that $\omega_{ij}^{(k)} > 0$ for all k, as desired.

2.3.2 Polygon meshes in space

The algorithm can be readily generalized to polygonal meshes in 3d ($\hat{\mathbf{V}} \in \mathbb{R}^{n_v \times 3}$). The property of non-negativity carries over directly. Linear precision on the other

hand must be replaced by a condition on the mean curvature normal

$$(\mathbf{L}\mathbf{x})_i = H_i \mathbf{n}_i \,, \tag{2.9}$$

with H_i being the mean curvature and $\mathbf{n}_i$ being the normal at vertex i. The mean curvature normal may be computed using the cotan Laplacian or the area gradient [1]. Condition (2.9) is consistent with the planar case, since discrete mean curvature vanishes for planar vertices. Hence, to compute perfect Laplacians for closed 3d meshes, we replace the right-hand side of Equation (2.4) with

$$\mathbf{b}_i = \begin{cases} (H_i \mathbf{n}_i)^\mathsf{T} & \notin \partial\mathcal{M} \\ \hat{\mathbf{v}}_i^\mathsf{T} & i \in \partial\mathcal{M} \end{cases} . \tag{2.10}$$

2.3.3 Steady state of the solution

We have applied the algorithm to numerous meshes and in all our examples it converged quickly and uniformly to a steady state. An empirical study of convergence will be presented in Section 2.5. Based on this experimental evidence we conjecture that Algorithm 1 converges for arbitrary input meshes with prescribed convex boundary.

It is instructive, however, to inspect the steady state of the solution. Recall that we scale the ω_{ij} uniformly and denote this factor μ. The update in Equation (2.8) reaches a steady state if, for any edge, either

$$\frac{\left\|\mathbf{e}_{ij}^{(\infty)}\right\|}{\left\|\hat{\mathbf{e}}_{ij}\right\|} = \mu^{-1} \quad \text{or} \quad \omega_{ij}^{(\infty)} = 0. \tag{2.11}$$

In particular, if all $\omega_{ij} > 0$ then all original edge lengths are preserved. This means that if a mesh admits a perfect Laplace operator, it is a steady state of our algorithm. If the mesh admits no perfect Laplace operator, a subset of edges (in planar regions) have zero Laplace coefficients while the rest is scaled by a constant factor relative to the original geometry. In the latter case, the lengths of edges corresponding to zero Laplace coefficients are determined by the embedding based on

the edges with positive coefficients. In a sense, the algorithm resorts to a subset of the edges, i.e., a coarsening of the mesh.

2.4 Meshes admitting perfect Laplacians

Unless we can characterize the meshes that admit perfect Laplace operators, it is difficult to claim that the algorithm always performs as expected. In the following, we will therefore extend known results for triangle meshes [107] to the general polygonal case.

Recall that a perfect Laplace operator induces a force network that is in equilibrium where all forces are pulling. Note that, conversely, the existence of such a force network implies the existence of a perfect Laplacian, noting that the spring constants must be positive. So the existence of perfect Laplacians corresponds exactly to the existence of pulling forces such that the nodes are in equilibrium. There is a useful characterization for such meshes: the vertices can be lifted such that their convex hull provides the given combinatorics [3]. In particular, triangle meshes with this property are referred to as *regular*[1], yielding the characterization of meshes that admit perfect Laplace operators in the sense of Wardetzky et al. [107].

However, the notion of meshes admitting lifts into convex position applies more generally to any planar mesh and the meshes that do admit such a lift are called *regular subdivisions*. Jaume and Roter [51] analyze the case of regular subdivisions in detail and in particular distinguish between convex and strictly convex lifts. This distinction also appears in the solutions provided by our algorithm (see Section 2.3.3), as it translates to distinguishing zero and strictly positive coefficients in the Laplace operator. Jaume and Roter also apply their results to the case of force networks, and we can adapt their Proposition 4.4 to derive the following characterization:

> A polygon mesh in the plane admits a perfect Laplace operator if there exists a lift of the vertex positions such that the combinatorics

[1] We avoid the notion of *weighted Delaunay* here because it is limited to triangulations with a single boundary, see for example [24].

of the convex hull of the vertices is a 3-connected spanning subgraph of the mesh.

Any edge not part of the subgraph corresponds to a dual edge of zero length or, equivalently, to a zero coefficient of the Laplacian.

The polygons in regular subdivisions are quite restricted. In particular, since they are projections of polyhedral faces, they are necessarily convex. In this view, it is not surprising that Alexa and Wardetzky [1] implicitly consider a more general construction: any two vertices that are part of the same face may carry a nonzero coefficient. We may characterize the meshes that admit a perfect Laplacian with such additional diagonals as follows: there exists a lift of the vertex positions such that *a subgraph of* the combinatorics of the convex hull of the vertices is a 3-connected spanning subgraph of the mesh. In particular, a polygon mesh in this case admits a perfect Laplace operator if it is contained in a regular triangulation of the vertices.

Note that the natural notion of allowing nonzero coefficients means that faces with higher degree are an obstruction for perfect Laplace operators, while allowing diagonals means that faces with higher degree relax the situation. We discuss the practical consequences of this distinction further in Section 2.7.

2.5 Results

In order to verify the validity of our approach, we performed a series of experiments covering different classes and types of meshes. We check for convergence as described in Algorithm 1 with a value of $\varepsilon = 10^{-6}$. All meshes have been scaled to fit a unit bounding box. For all experiments we conducted, the algorithm converged numerically.

Triangle meshes Our first test case concerned triangular meshes. To this end, we created random triangle meshes by computing tessellations of the unit square

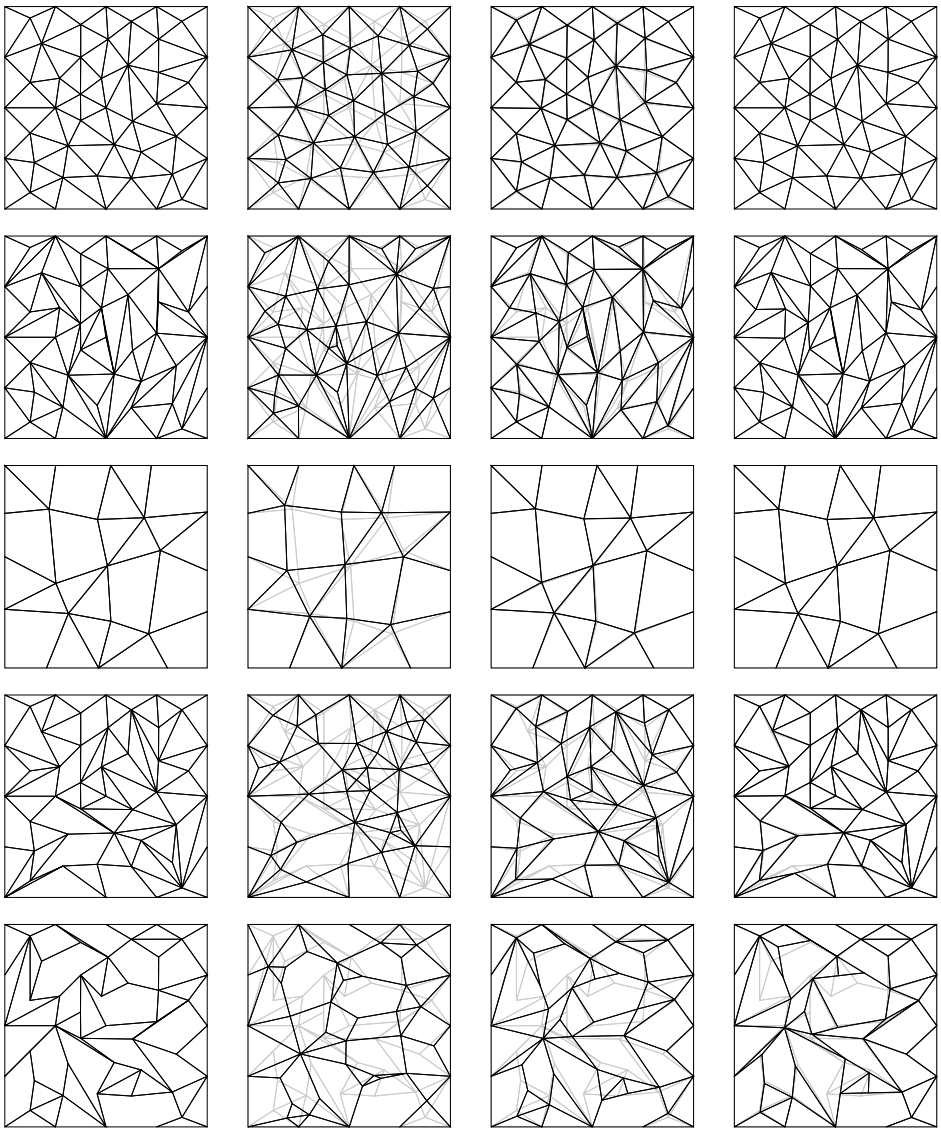


Figure 2.1: *Results created by our approach for different classes of polygon meshes. Each row depicts iterations of our algorithm starting with an input tesselation.*

(a) Delaunay triangulation: Original, 1, 5, and 20 iterations.

(b) Regular triangulation that is not Delaunay: Original, 1, 5, and 20 iterations.

(c) Polygonal mesh with convex faces: Original, 1, 5, and 20 iterations.

(d) Polygonal mesh with non-convex faces: Original, 1, 5, and 50 iterations.

(e) Polygonal mesh with non-convex faces: Original, 1, 5, and 50 iterations.

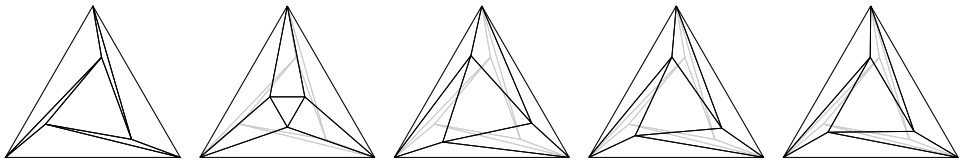


Figure 2.2: *The shadow of the Schönhardt polyhedron, a well-known examples for triangular mesh not admitting a perfect Laplacian. From left to right: Original, 1, 5, 10, 50 iterations of our algorithm.*

and unit circle using the software Triangle [90]. In addition, we created convex tessellations of random point clouds using CGAL [101]. Since the tessellation algorithms create Delaunay triangulations, we also further modified the tessellations by randomly flipping approximately 20% of the edges. While flipping, we ensured that the boundary is preserved and edges are not flipped if the flip would introduce inverted triangles. We tested the algorithm on 1000 random meshes, each with about 50 vertices sampled on a square and a disc. For all cases, we observed that each iteration produces a valid embedding together with an accompanying perfect Laplacian. Moreover, the vertices reached steady state and converged towards the input mesh, achieving good approximations after 5–10 iterations. This indicates that for all these cases the randomly generated meshes were regular, admitting a perfect Laplace operator. Figures 2.1(a) and (b) show two typical examples.

A more challenging test case is given by meshes that do not admit a perfect Laplacian. To this end, we created five manual examples with 8 to 15 vertices, such as the shadow of the Schönhardt polyhedron shown in Figure 2.2. The algorithm handles these cases as well; however, necessarily by changing some edge lengths and thereby moving the vertices (see Section 2.3.3 for details).

While we observe that our approach reaches steady state in all cases, the obtained solutions are not unique. Choosing different initial coefficients typically leads to different results. In particular, for Delaunay meshes with convex boundary, our approach does not recover the cotan Laplacian (which is perfect in this case).

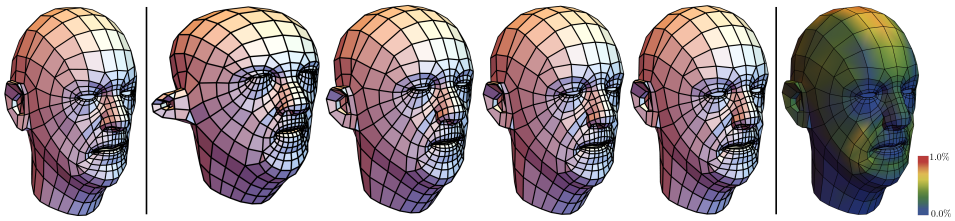


Figure 2.3: *Given a polygonal input mesh (left), our algorithm is applied. Iteration 1,2,5 and 20 are shown (center). The converged result is shown on the right. The color coding illustrates the vertex displacement from input mesh to the result. The maximal vertex displacement is about* 0.85% *of the bounding box diagonal.*

Polygon meshes In a second series of tests we considered planar polygonal meshes. Proceeding as in the previous section, we created random triangle meshes but removed some of the edges. As for triangle meshes, we observed convergence for all 1000 random polygonal meshes tested. The algorithm enforces convex faces in all iterations even in case of non-convex input meshes. Two examples produced by our approach are shown in Figures 2.1(c), (d) and (e).

3D meshes For meshes embedded in 3-space the algorithm performs similar to the 2d case. Figure 2.3 illustrates the convergence for a polygonal input mesh. We tested the algorithm on a number of standard meshes and added artificial noise to assess robustness. For all examples the algorithm converged and produced a valid perfect Laplacian. The next section details applications involving meshes in 3-space.

2.6 Applications

Self-supporting surfaces Several recent works investigate self-supporting surfaces in equilibrium [24, 65, 77]. The central idea is that a network of rigid struts can withstand its own dead load if it is in force equilibrium at the points where struts meet (vertices), except for supported vertices where force can be dissipated.

Denoting the gravitational force acting on vertex i along the z-direction by F_i and the force along the edge $\mathbf{e}_{ij}$ by f_{ij} the condition for structural stability at vertex i becomes

$$\sum_{(i,j)\in E} f_{ij} \frac{\mathbf{v}_j - \mathbf{v}_i}{||\mathbf{v}_j - \mathbf{v}_i||} = \begin{pmatrix} 0 \\ 0 \\ -F_i \end{pmatrix}, \quad \text{with } f_{ij} \geq 0. \tag{2.12}$$

All forces have to be compressive as indicated by the positivity of the forces. Considering $\omega_{ij} = f_{ij}/||\mathbf{v}_j - \mathbf{v}_i||$ we are faced with the problem of computing a perfect Laplacian for a polygon mesh embedded in 3-space. Hence, we can apply our algorithm with appropriate boundary conditions for supported vertices. In contrast to recent work, our algorithm does not rely on interior-point solvers and can be easily implemented using a sparse linear solver. Moreover, our system handles polygon meshes naturally. The results of our algorithm match results of existing methods. Self-supporting surfaces for several input meshes and comparisons to state of the art methods are shown in Figures 2.4 and 2.5, respectively.

In contrast to related work, we do not modify the combinatorics of the mesh which limits the direct applicability of our approach. However, our algorithm can serve as a central building block in a more sophisticated algorithm, driving a remeshing step based on the force distribution or in situations where a certain frame topology needs to be fixed.

Parameterization A classical application of discrete Laplacians is mesh parameterization. Given a fixed convex boundary in the plane and a discrete Laplacian of a polygon mesh embedded in $\mathbb{R}^3$ we can compute an embedding into $\mathbb{R}^2$ by solving Equation (2.4). Using operators with negative coefficients can result in face flips, i.e., the embedding is not injective. For applications like texture mapping this behavior in not acceptable. With our algorithm a flip-free embedding is always guaranteed. This holds for the triangle as well as for the polygonal case. To our knowledge this is the first algorithm to introduce a geometry aware flip-free parameterization technique for polygon meshes without resorting to a triangula-

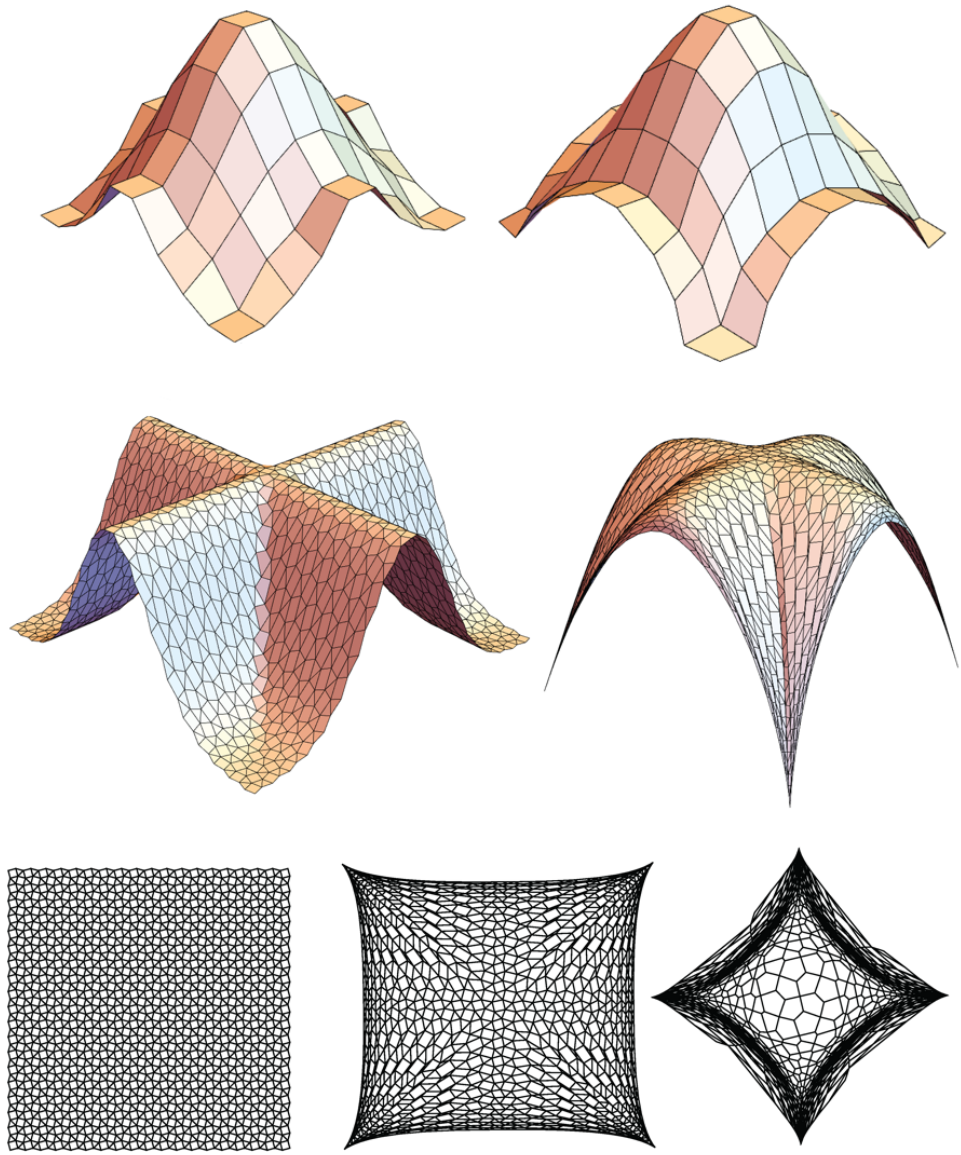


Figure 2.4: *Fitting of self-supporting surfaces to several polygonal meshes. The last row shows the planar projection of the input mesh above, the projection of the self-supporting surface and its dual.*

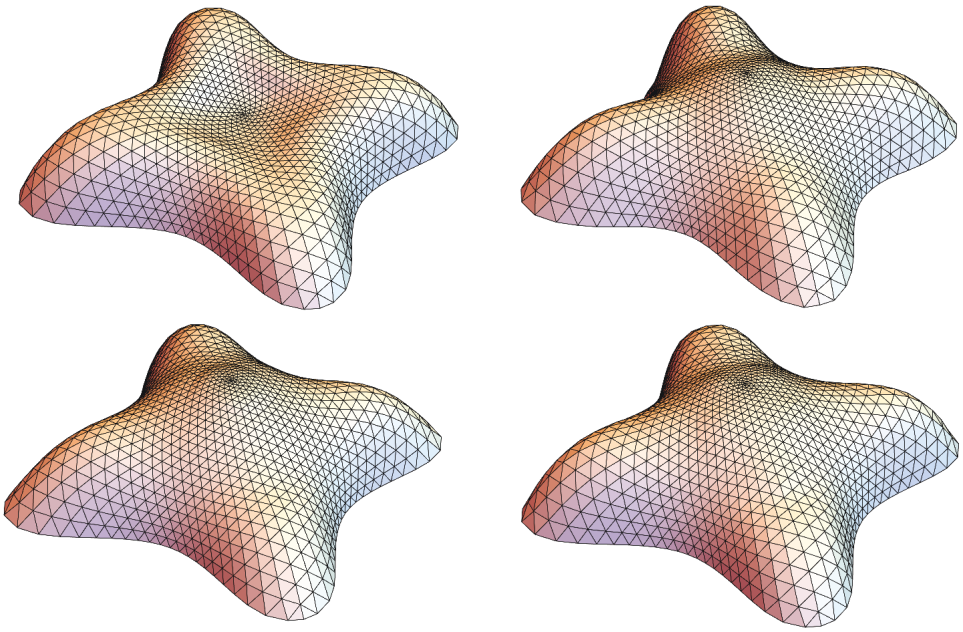


Figure 2.5: *Our algorithm produces self-supporting surfaces comparable to state of the art methods. The input mesh (top, left) is deformed towards a self-supporting surface by our method (top, right), the method by Vouga et al. [104] (bottom, left), and using the technique of de Goes et al. [24].*

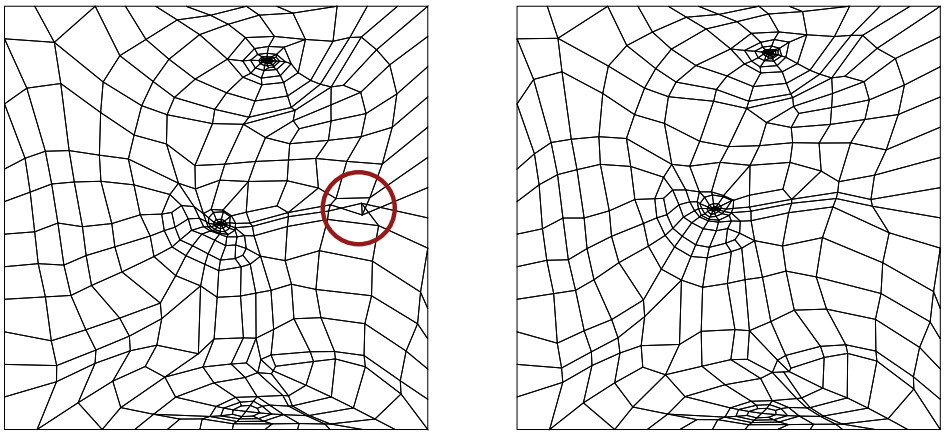


Figure 2.6: *Parameterizing using the Laplace operator of Alexa et al. [1] can introduce flipped faces (left, flipped faces marked red). Our operator theoretically ensures a flip-free embedding. In practice numerical effects need to be taken into account [89].*

tion.

2.7 Discussion

Our approach provides a perfect Laplace operator for any input mesh, albeit at the cost of potentially moving the vertices. In the following, we briefly discuss the consequences, particularly in light of related approaches.

2.7.1 Modifying combinatorics

If triangle meshes with convex boundary are Delaunay, all triangle angles are smaller than $\pi/2$ and the cotan operator is non-negative. An interesting theorem due to Rippa [81] states that general planar triangle meshes can be turned into Delaunay triangulations by incrementally flipping edges violating the Delaunay property. Bobenko and Springborn [6] construct perfect Laplacians for triangle meshes embedded in $\mathbb{R}^3$ by generalizing this theorem using intrinsic edge flips, i.e., they flip edges only for the generation of the Laplace operator—the operator may still be applied to the original mesh. Although these algorithm are guaranteed to terminate, it might take up to n_v^2 edge flips in the planar case [30] and even more for intrinsic edge flips on non planar meshes. Perhaps more importantly, in some applications it might be important to fix combinatorics, such as for different vertex geometries that represent different instances of a mesh. In a sense both approaches are related: they apply an operator derived from different underlying combinatorics, while we may apply an operator derived from different geometry; so in both cases the Laplace operator was derived from a metric that is inconsistent with the input.

2.7.2 Modifying geometry

The work of Mullen et al. [75] is similar in spirit to ours. Using non-linear optimization, they search for a weighted cotan Laplacian exhibiting good numerical behavior and mostly non-negative weights. Optionally, vertex positions are optimized as well. Although non-negativity is not implied, the results are impressive and can significantly improve the accuracy of computations involving the dis-

crete Laplacian. By contrast, our approach guarantees non-negativity and, perhaps more importantly, generalizes to polygonal meshes. We also think it is conceptually simpler and offers a more straightforward implementation.

2.7.3 Using diagonals

As mentioned earlier, Alexa and Wardetzky [1] implicitly consider all diagonals in a face. Interestingly, while this improves flexibility and makes it easier to generate non-negative Laplacians, their construction equally uses all diagonals, resulting in negative coefficients for almost all faces.

A downside of our approach is that using only the original edges, a perfect Laplacian requires convex faces. We could use all diagonals in our construction but at the expense of being able to guarantee an embedding in the planar case. This might be acceptable especially for the practical case of meshes in $\mathbb{R}^3$. In fact, we have experimented with using diagonals and the results look reasonable.

There is, however, a good reason for avoiding diagonals also in the case of meshes in $\mathbb{R}^3$: in a non-planar face, positive Laplace coefficients for the diagonals imply that the area gradient flow would reduce the area by folding the polygon rather than making it planar (as desired). This suggests that the choice of a Laplace operator may be application dependent.

3

Diffusion diagrams

3.1 Introduction

The notion of *being closer* is a fundamental geometric relation, which gives rise to Voronoi cells and centroids. It is commonly defined based on a *distance function*. In the Euclidean domain, smallest distances are continuous and can be easily computed. Consequently, bisectors for Voronoi tessellations or centroids are also stable and fast to compute.

On curved surfaces, the most natural way to define distance is the length of the shortest path between two points. This path is a geodesic and the corresponding distance is called geodesic distance. Geodesic distances to a fixed point is generally not continuous, and their computation is more involved and potentially unstable. This has motivated alternative definitions of distance for curved surfaces [4, 60, 78, 92].

Our main observation is that the property of *being closer* can be computed based on a smooth function on the surface that may not satisfy the properties of a distance measure. A smooth function might overcome stability problems that are present for exact distance functions and might be easier to compute. We suggest to use *heat diffusion* for this purpose. An important property of heat diffusion is that it represents a monotone function of distance in Euclidean domains for infinitesimal small timesteps. Computing heat diffusion on a discrete domain is well understood and amounts to solving a linear system. We recall basic properties of the continuous and discrete heat equation in Section 3.2.

We analyze the definition of bisectors (or Voronoi cells) based on heat values in Section 3.4. We show that heat values yield Voronoi diagrams that are identical for Euclidean and spherical domains. An important consequences of defining the bisectors based on heat values, and not a derived measure of distance, is that they converge quadratically. We demonstrate that generating curved Voronoi cells from heat is asymptotically as accurate as defining them from discrete geodesics, while being much faster.

We also use heat to define centroids of cells (see Section 3.5). We can show that for large time values the heat centroid converges against Euclidean centroids. For curved domains we find that heat centroids provide more stable (and more intuitive) results than geodesic centroids because of the underlying smooth function.

All computations are based on the same linear system, which is factorized in a preprocessing step. Generating Voronoi tessellations for arbitrary generators or finding centroids of arbitrary cells only require back substitutions. Moreover, we explain in Section 3.6 how to localize the solve – this means we can compute heat values only where needed for comparison with other heat values. This idea extends existing methods in numerical linear algebra libraries and should be of interest for other applications in geometry processing.

We demonstrate the relevance of our approach by tessellating large meshes with centroidal Voronoi tessellations (CVT) and computing barycentric coordinates on surfaces (see Section 3.7). Lastly, we discuss more potential applications and future work in Section 3.8.

3.2 Discrete heat diffusion

Given a domain Ω, the *heat kernel* $k_{\mathbf{p}} : \mathbb{R}^+ \times \Omega$ describes how much heat diffuses from a fixed point $\mathbf{p} \in \Omega$ to a point $\mathbf{q} \in \Omega$ in time $t \in \mathbb{R}^+$. On a Riemannian manifold without boundary the heat kernel is the fundamental solution of the *heat equation*

$$\begin{cases} \frac{\partial k_{\mathbf{p}}}{\partial t} & = \Delta_{\mathbf{q}} k_{\mathbf{p}}(t, \mathbf{q}) \\ k_{\mathbf{p}}(0, \mathbf{q}) & = \delta(\mathbf{p} - \mathbf{q}), \end{cases} \tag{3.1}$$

where $\Delta_{\mathbf{q}}$ is the Laplace-Beltrami operator in $\mathbf{q}$ and δ represents the Dirac delta. For $\Omega = \mathbb{R}^d$ the heat kernel is explicitly given by the formula

$$k_{\mathbf{p}}(t, \mathbf{q}) = \frac{1}{(4\pi t)^{d/2}} \exp\left(-\frac{\|\mathbf{p} - \mathbf{q}\|^2}{4t}\right). \tag{3.2}$$

Inverting this expression one obtains Varadhan's formula [103]

$$\lim_{t \to 0} \sqrt{-4t \log k_{\mathbf{p}}(t, \mathbf{q})} = \|\mathbf{p} - \mathbf{q}\|, \tag{3.3}$$

revealing the close connection of the heat kernel $k_{\mathbf{p}}(t, \mathbf{q})$ to the distance to $\mathbf{p}$ which also generalizes to manifolds away from the cut locus [103].

We are interested in heat diffusion on discrete surfaces embedded in $\mathbb{R}^3$, represented as triangulated surfaces.

Every point $\mathbf{p} \in \mathcal{M} = (\mathcal{V}, \mathcal{F})$ on the mesh can be represented using barycentric coordinates $\mathbf{b}(\mathbf{p}) \in \mathbb{R}^{n_v}, \mathbf{b}(\mathbf{p})_i \geq 0, \sum_i \mathbf{b}(\mathbf{p})_i = 1$ with respect to the mesh vertices such that

$$\mathbf{p} = \mathbf{V}^\mathsf{T} \mathbf{b}(\mathbf{p}). \tag{3.4}$$

Note that $\mathbf{b}(\mathbf{p})$ can be chosen to have at most three nonzero entries for any point $\mathbf{p}$ on the mesh which are defined by a face containing $\mathbf{p}$.

To solve Equation 3.1 numerically on a triangle mesh we discretize the Laplacian using the cotan operator [79], following the same approach as Crane et al. [17].

Using their notation, the cotan Laplacian is represented by $\mathbf{L} = \mathbf{M}^{-1}\mathbf{L}_C$ where the diagonal (lumped) mass matrix $\mathbf{M} \in \mathbb{R}^{n_v \times n_v}$ and $\mathbf{L}_C \in \mathbb{R}^{n_v \times n_v}$ are given by

$$(\mathbf{L}_C\mathbf{u})_i = \frac{1}{2} \sum_{(i,j)\in\mathcal{E}} (\cot \alpha_{ij} + \cot \beta_{ij})(u_j - u_i) \tag{3.5}$$

$$M_{ll} = \frac{1}{3} \sum_{(i,j,k)\in\mathcal{F},\ l\in(i,j,k)} a_{ijk}. \tag{3.6}$$

Here $\mathbf{u} \in \mathbb{R}^{n_v}$ is a vector of coefficients with respect to the piecewise linear finite element basis. Performing an implicit Euler step of length t for Equation 3.1 amounts to solving the sparse, symmetric, positive-definite system (for an appropriate choice of t)

$$(\mathbf{M} - t\mathbf{L}_C)\mathbf{u} = \mathbf{M}\mathbf{h}. \tag{3.7}$$

In order to model a discrete Dirac delta, the right-hand side has to represent a point with unit energy which leads to

$$\mathbf{h} = \mathbf{M}^{-1}\mathbf{b}(\mathbf{p}). \tag{3.8}$$

In the following sections we will also consider more general initial conditions. Namely we want to generalize the heat source to a surface patch $\mathcal{C} \subset \mathcal{M}$. With the characteristic function $\chi_{\mathcal{C}}(\mathbf{q})$ the initial condition can be formulated as

$$k_\chi(0, \mathbf{q}) = \chi_{\mathcal{C}}(\mathbf{q}). \tag{3.9}$$

Practically this amounts to setting

$$\mathbf{h} = \frac{\int_{\mathcal{C}} \mathbf{b}(\mathbf{p})\, d\mathbf{p}}{\int_{\mathcal{C}} 1\, d\mathbf{p}}. \tag{3.10}$$

Here, the denominator is just the area of the set $\mathcal{C}$. Computing this integral is easy for piecewise linear sets, as they can be triangulated. For convenience, assume the mesh has been triangulated such that the interior of each triangle is either completely inside or completely outside the set $\mathcal{C}$. Each interior triangle $(i, j, k) \in \mathcal{F}$

contributes

$$\mathbf{h}_{ijk} = \frac{1}{3}\left(\mathbf{e}_i + \mathbf{e}_j + \mathbf{e}_\mathbf{k}\right) \tag{3.11}$$

where $\mathbf{e}_i$ denotes a canonical basis vector. The vector $\mathbf{h}_C$ representing the set $\mathcal{C}$ can then be build by summing up over the faces, i.e. $\mathbf{h}_C = \sum_{(i,j,k)\in\mathcal{C}} \mathbf{h}_{ijk}$. We can easily solve Equation 3.7 repeatedly for different initial heat distributions if we have the Cholesky factorization of the system matrix $\mathbf{M} - t\,\mathbf{L}_C$. Computing heat diffusion then comes down to one pass of forward and back substitution for the case of a singular heat source as well as for an area heat source.

3.3 Related work

We focus on the computation of CVTs and geodesics on meshes embedded in $\mathbb{R}^3$.

Geodesics on meshes In principle, any algorithm computing geodesic distances on meshes can be used to generate Voronoi diagrams. One of the earliest approaches generalizes fast marching on regular grids to 3d triangle meshes [87]. However, this method is inaccurate in presence of obtuse triangles. Several strategies to deal with this problem have been proposed, e.g. [55], leading to approximate solutions. Mitchell et al. [73] describe an algorithm to compute exact geodesic distances on triangle meshes which has been implemented by Surazhsky et al. [99]. Chen et al. [13] also propose a method to compute exact geodesic distances which has been improved by Xin et al. [108] outperforming both the latter and Surazhsky et al.'s code.

Diffusion Solutions to the heat equation have been extensively used in machine learning and geometry processing, for example by Benjamin et al. [5] in the context of mesh segmentation.

The central idea is to compute the fundamental solution to (3.1) on a manifold.

Given a discrete Laplacian $\mathbf{L}$ this solution can be expressed as

$$d_t^2(\mathbf{p}, \mathbf{q}) = \sum_k f(\lambda_k)(\varphi_k(\mathbf{p}) - \varphi_k(\mathbf{q}))^2 \tag{3.12}$$

$$f(\lambda) = \exp(-2t\lambda) \tag{3.13}$$

where λ_i and φ_i are eigenvalues and eigenvectors of the Laplacian $\mathbf{L}$, respectively. Using a discrete Laplacian based on the Gaussian kernel, Coifman et al.[15] introduced the term *diffusion distance* for values of d for applications in machine learning. Altering the function f yields different notions of distance, notably *commute time distance* for $f(\lambda) = 1/\lambda$ [34] and *biharmonic distance* for $f(\lambda) = 1/\lambda^2$ [60].

Computing these distances up to a specified accuracy is possible using only a subset of eigenvectors rendering these methods feasible on meshes of moderate size. Patané et al. [78] compute fundamental solutions of the heat kernel by approximating the matrix exponential using Padé approximation. Our method can be understood as using a Padé approximation of order $[0, 1]$. Recently, Crane et al. [17] introduced a fast approximate method to compute geodesic distances based on heat diffusion. The central idea is to diffuse an initial heat distribution by solving a sparse linear system and integrating the normalized gradients of the solution. Iterating the normalization and integration step, Belyaev et al.[4] propose an algorithm tailored to the task of finding the shortest distance to the boundary of a domain in $\mathbb{R}^2$ or $\mathbb{R}^3$.

These methods inspired our use of heat diffusion to measure distance. In contrast to Crane et al. we use heat diffusion directly as a "pseudo", distance similar to Solomon et al. [92].

Centroidal Voronoi tessellations on meshes Many methods for the construction of CVTs on triangle meshes reduce the problem to the Euclidean case. Alliez et al. [2] use a planar conformal parameterization of the mesh to map a CVT with respect to a spatially varying distance function on $\mathbb{R}^2$ to the mesh. This approach requires cutting the mesh into a disk. This is a challenging task in itself and might introduce misalignments at the seams. Another common idea is to restrict CVTs

in $\mathbb{R}^3$ to a surface mesh (see for example [110]). Whenever the cells are large relative to the variation of the metric this method fails to capture the distances on the surface. For example using few generators to define intrinsic barycentric coordinates (see Section 3.7) really requires an intrinsic method. Moreover, close sheets and self-intersection can not be handled by this method.

One of the few practical intrinsic methods for the computation of CVTs was presented by Wang et al. [105]. They employ a modified and parallelized version of Xin et al.'s [108] implementation of geodesic distances to rapidly compute intrinsic Voronoi diagrams. The computation of cell centroids relies on the repeated evaluation of the exponential map at the cell vertices. Approximating centroids on planar projections of the cells is problematic in terms of convergence and quality for large cells with high variation in curvature. Our method in contrast produces cell centroid that are well defined and insensitive to noise and cell size.

3.4 Voronoi cells

The notion of *being closer* may be used to tessellate a domain based on a set of points in that domain. Concretely, given a domain Ω as well as a set of points $\mathbf{p}_i \in \Omega, i \in \{1, \ldots, n\}$, called *generators*, the *Voronoi tessellation* is the decomposition of the domain Ω into *Voronoi cells* $\mathcal{C}_i \subset \Omega$ consisting of the points $\mathbf{x} \in \Omega$ that are not further from $\mathbf{p}_i$ than from any other of the generators $\mathbf{p}_j$:

$$\mathcal{C}_i = \left\{ \mathbf{x} \in \Omega, d(\mathbf{x}, \mathbf{p}_i) \leq d(\mathbf{x}, \mathbf{p}_j), i \neq j \right\}. \tag{3.14}$$

Here $d : \Omega \times \Omega \mapsto \mathbb{R}$ is a distance measure on Ω. The set of points $\mathbf{x}$ that are elements of more than one cell $\mathcal{C}_i$, i.e. lie in the intersection of two or more cells are called the *Voronoi diagram*. For non-degenerate configurations it consists of *edges*, which are the intersections of exactly two Voronoi cells, and *corners*, which are the intersections of three or more Voronoi cells.

We define a Voronoi cell based on heat diffusion. As heat values decrease from the

source, we use

$$\mathcal{C}_i(t) = \left\{ \mathbf{x} \in \Omega, k_{\mathbf{p}_i}(t, \mathbf{x}) \geq k_{\mathbf{p}_j}(t, \mathbf{x},), i \neq j \right\}. \tag{3.15}$$

A motivation for this definition is that it is equivalent to the above in Euclidean domains because $k_{\mathbf{p}}(t, \mathbf{q})$ is monotone in the squared distance $\|\mathbf{p} - \mathbf{q}\|^2$ (see Equation 3.2).

Discrete approximation For computing a Voronoi cell on a triangle mesh we compute a piecewise linear approximation of the heat equation as discussed in Section 3.2. For each generator $\mathbf{p}_i$ we compute the corresponding heat distribution, which can be written as

$$(\mathbf{M} - t\mathbf{L})\,\mathbf{U} = (\mathbf{b}(\mathbf{p}_0), \mathbf{b}(\mathbf{p}_1), \ldots), \tag{3.16}$$

so that the i-th column of $\mathbf{U}$ represents the heat values corresponding to generator $\mathbf{p}_i$. The rows of $\mathbf{U}$ correspond to the different heat values in each vertex. For an arbitrary point $\mathbf{q}$ on the mesh we can find the heat values by computing $\mathbf{U}^\mathsf{T}\mathbf{b}(\mathbf{q})$. This allows us to define the Voronoi cell $\mathcal{C}_i$ as the set of points for which the i-th component in the vector of heat values is larger than any other component:

$$\mathcal{C}_i = \left\{ \mathbf{x} \in \Omega, \left(\mathbf{U}^\mathsf{T}\mathbf{b}(\mathbf{x})\right)_i \geq \left(\mathbf{U}^\mathsf{T}\mathbf{b}(\mathbf{x})\right)_j, i \neq j \right\}. \tag{3.17}$$

Based on this information we approximate the Voronoi diagram. First we associate each vertex with a label

$$l_i := \arg\max_j u_{ij} \tag{3.18}$$

corresponding to the closest generator. If all vertices of a face are assigned the same label the face belongs to the corresponding generator and does not contain parts of the Voronoi diagram. We further distinguish two situations. If two vertices i and j carry the same label different from the label of the third vertex k, we identify

points of the Voronoi diagram on edges (ik) and (jk) by linear interpolation:

$$bu_{il_i} + (1-b)u_{k\,l_i} = bu_{il_k} + (1-b)u_{kl_k} \tag{3.19}$$

$$\Rightarrow \quad b = \frac{u_{kl_k} - u_{kl_i}}{u_{il_i} - u_{kl_i} - u_{il_k} + u_{kl_k}} \tag{3.20}$$

the final point can then be computed as $\mathbf{s}_{ik} = b\mathbf{v}_i + (1-b)\mathbf{v}_k$. $\mathbf{s}_{jk}$ is determined analogously. Connecting these points across the triangle yields our local approximation of the Voronoi diagram. The third case covers triangles with three different labels. In this situation we compute points on each edge via interpolation like in the previous case. All three points are then connected to a central point that is defined using barycentric coordinates

$$\mathbf{s}_{ijk} = \frac{1}{u_{il_i} + u_{jl_j} + u_{kl_k}}(u_{il_i}\mathbf{v}_i + u_{jl_j}\mathbf{v}_j + u_{kl_k}\mathbf{v}_k). \tag{3.21}$$

Note that this is just an approximation, the piecewise linear function defined by the heat values might even intersect in an point outside the triangle. This could be alleviated by a more sophisticated intersection analysis of the linear functions at the cost of some performance. For sufficiently dense tessellations we have found our approximation to perform very well.

Numerical convergence Now that we have a process in place that computes the Voronoi diagram, we ask about the numerical accuracy of this procedure. Computing exact geodesics results in exact intrinsic Voronoi diagrams with respect to the surface mesh. However, if the surface mesh discretizes an underlying smooth surface this might not be the result we are interested in. Topological noise, for example, will influence geodesic distances dramatically while diffusion methods are potentially more robust to these artifacts. Moreover, coarse tessellations will have a strong effect on geodesic distances. Consider the tessellated sphere in Figure 3.1 (inset below).

Experimentally we measure the derivation of the Voronoi diagrams from the exact result on the unit sphere with the following locations as generators: $\mathbf{p}_\circ =$

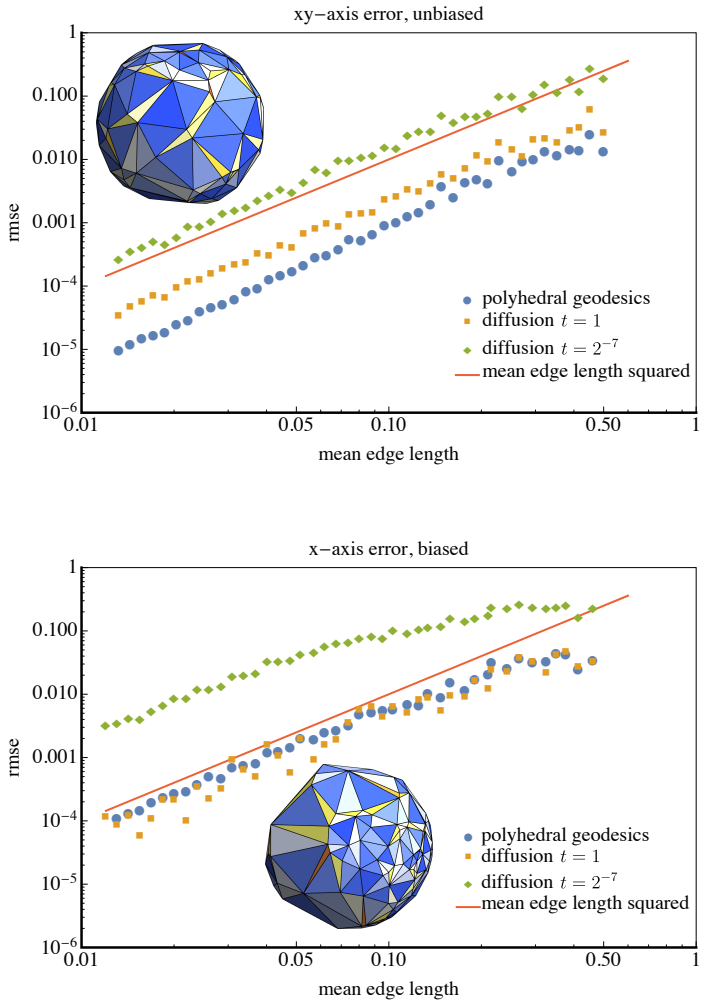


Figure 3.1: *Errors of bisectors approximated on triangulations of a sphere relative to mean edge length. The left is based on uniform vertex distributions with one generator at the pole and another on the equator. The right plot is based on a distribution biased along the one axis and the generators on opposite points on this axis. Both plots show the errors of exact polyhedral geodesics and heat diffusion using $t = 2^0$ and $t = 2^{-7}$. Examples of the meshes used in the experiments are shown as insets.*

$(1, 0, 0)^\mathsf{T}$, $\mathbf{p}_1 = (-1, 0, 0)^\mathsf{T}$, $\mathbf{p}_2 = (0, 1, 0)^\mathsf{T}$. For this example the ground truth is easily established. We measure the errors of the bisectors between $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$. Let $\mathbf{x}$ be a point on the approximated bisector, then we can measure its squared error as

$$e^2(\mathbf{x}) = \mathbf{x}^\mathsf{T} \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} \tag{3.22}$$

as all bisectors are contained in planes through the origin.

In this setting, we analyze the approximation error relative to the time step t and the maximal or mean edge length in the mesh. Based on prior analysis of diffusion operators on meshes, we expect that the approximation is better for larger values of t [17], and that it converges quadratically as edge lengths decrease [28]. Note that we cannot expect any better convergence, as all metric properties of the discrete surface are converging at most quadratically. In particular, also discrete geodesics are converging quadratically against their smooth counterpart.

We have generated a set of triangulations of the unit sphere with varying characteristics. In all cases the process starts with a regular octahedron. Then vertices are randomly sampled from the unit sphere. For generation of uniform random points we draw the three coordinates independently from normal distributions and then normalize the result. This results in triangulations with varying triangle quality, including many with obtuse angles (see the insets in Figure 3.1).

We also bias the distribution so that the average density varies along the first coordinate axis. Such spheres with different densities around the points $\mathbf{p}_0$ and $\mathbf{p}_1$ allow us to check if the bisectors still converge to the ground truth, and how much they are affected by differently sized triangles. Using this set of meshes, we measure the error of the bisectors as a function of the edge length based on two values for the time step: $t = 1$ and $t = 2^{-7} \approx 0.008$. For comparison, we also compute the bisectors using polyhedral geodesics, i.e. the exact distance on the piecewise linear surface, based on Kirsanov's implementation [99]. The resulting root mean squared errors relative to the mean edge lengths are shown in Figure 3.1. All plots clearly show the expected quadratic convergence for the diffusion bisectors as well as for polyhedral distances. For the uniformly sampled case the constant is better compared to the biased case. Diffusion using large values for t seems to be slightly

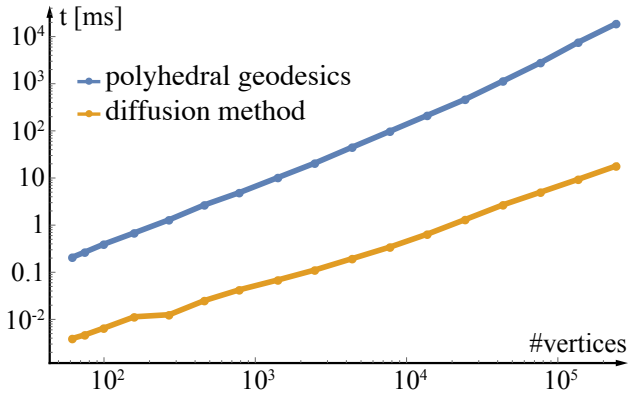


Figure 3.2: *Computing the exact polyhedral distance from one points to each other using the algorithm of Qin et al. [80] takes orders of magnitudes longer than computing heat diffusion given a prefactored matrix. This is true for a range of mesh sizes.*

more robust to variations in triangle size, as can be seen in the slightly better results for the biased vertex distribution for $t = 1$. Statistics based on max-errors and/or max edge lengths show similar characteristics.

Performance The computation of Voronoi diagrams is dominated by the evaluation of heat diffusion. We compare this to the time it takes to compute polyhedral distances using the state of the art method by Qin et al. [80].

The graph in Figure3.2 shows the time needed for computing heat values and polyhedral distances relative to the number of vertices of a planar disk-shaped mesh. For each vertex count we generate a spherical mesh and average timings over all possible seed vertices. Timings include set up costs, such as the matrix factorization for the computation based on diffusion. In this experiment heat diffusion is significantly faster and also scales better with the number of vertices.

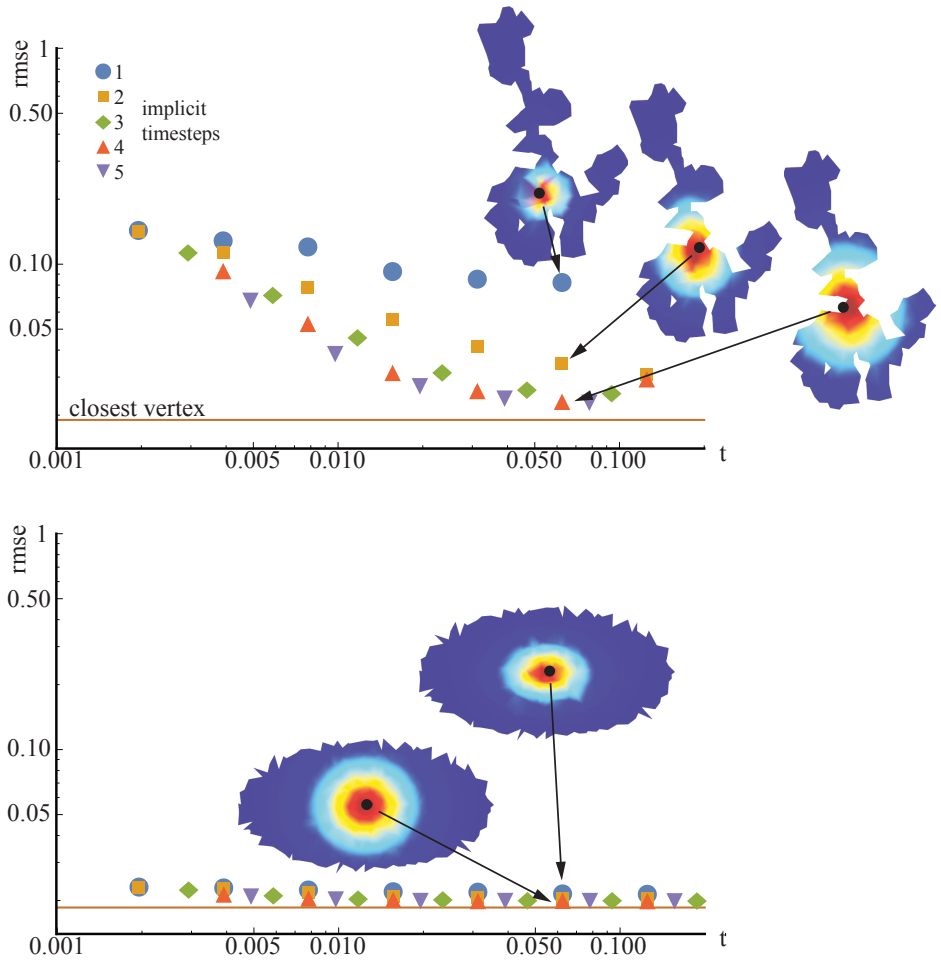


Figure 3.3: *The heat center converges to the Euclidean centroid in the plane for large t. The horizontal line represents the distance to the closes vertex, which is the optimum value for the discrete approximation. The effect is non-linear, so requires several linear time steps. For large t the boundary of the finite disk distorts the result. While the non-linearity is clearly visible for irregular polygons (left), Euclidean centroid and heat centroid are very similar for a wide range of parameters for well-behaved polygons (right).*

3.5 Centroids on surfaces

For Euclidean domains the *centroid* is the center of mass. For more general domains one may employ the notion of *Riemannian center of mass* [52], which can be defined as the point that minimizes the integrated squared distance values to all points in the cell:

$$\mathbf{c}_i = \operatorname*{argmin}_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} d^2(\mathbf{x}, \mathbf{y})\, d\mathbf{x}. \tag{3.23}$$

This definition works with arbitrary choices for the distance function $d(\cdot, \cdot)$. For Riemannian surfaces a common choice for $d(\cdot, \cdot)$ is *geodesic distance*.

Using heat diffusion we define the center as the point in a cell that generates the most heat within the cell when diffused:

$$\mathbf{c}_i(t) = \arg\max_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} k_{\mathbf{y}}(t, \mathbf{x})\, d\mathbf{x}. \tag{3.24}$$

Using the symmetry of the heat kernel (3.2) we have

$$k_{\mathbf{y}}(t, \mathbf{x}) = k_{\mathbf{x}}(t, \mathbf{y}). \tag{3.25}$$

Moreover, because the heat equation (3.1) is a linear PDE we have

$$k_{\mathcal{C}_i}(t, \mathbf{x}) = \int_{\mathcal{C}_i} k_{\mathbf{y}}(t, \mathbf{x})\, d\mathbf{x} \tag{3.26}$$

which describes more general initial conditions for the heat diffusion problem as discussed earlier (3.10). Because we discretize the heat equation as a linear system of equations these properties are exactly captured in the discrete setting.

We can now reformulate the optimization problem

$$\mathbf{c}_i(t) = \arg\max_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} k_{\mathbf{y}}(t, \mathbf{x})\, d\mathbf{x} \tag{3.27}$$

$$= \arg\max_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} k_{\mathbf{x}}(t, \mathbf{y})\, d\mathbf{x} \tag{3.28}$$

$$= \arg\max_{\mathbf{y}\in\Omega} k_{\mathcal{C}_i}(t, \mathbf{y}). \tag{3.29}$$

This means we diffuse heat for all points in the cell $\mathcal{C}_i$ and then look for the point that is hottest after some time t. This interpretation is more appealing because it requires only a single solve for heat values, as described in Section 3.2, followed by finding the point with the largest heat value.

Relation to Euclidean centroids As it turns out, this definition is not just an approximation to Euclidean centroids, it approaches them for large t. To see this, we plug the heat kernel into equation 3.24

$$\mathbf{c}_i(t) = \arg\max_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} \frac{1}{(4\pi t)^{d/2}} \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{4t}\right) d\mathbf{x} \tag{3.30}$$

$$= \arg\max_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{4t}\right) d\mathbf{x}. \tag{3.31}$$

Writing the exponential function as a series leads to

$$\mathbf{c}_i(t) = \arg\max_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} 1 - \frac{\|\mathbf{x}-\mathbf{y}\|^2}{4t} + \frac{\|\mathbf{x}-\mathbf{y}\|^4}{32t^2} - \dots\, d\mathbf{x} \tag{3.32}$$

$$= \operatorname*{argmin}_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} \|\mathbf{x}-\mathbf{y}\|^2 - \frac{\|\mathbf{x}-\mathbf{y}\|^4}{8t} + \dots\, d\mathbf{x}. \tag{3.33}$$

We see that for $t \to \infty$ the first term dominates, i.e.

$$\lim_{t\to\infty} \mathbf{c}_i(t) = \operatorname*{argmin}_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} \|\mathbf{x}-\mathbf{y}\|^2 d\mathbf{x} \tag{3.34}$$

$$= \operatorname*{argmin}_{\mathbf{y}\in\Omega} \int_{\mathcal{C}_i} d^2(\mathbf{x}, \mathbf{y}) d\mathbf{x} \tag{3.35}$$

which is exactly the definition of the Riemannian center of mass (3.23).

Discrete version and numerics The interpretation of setting the cell to constant heat and defining the center as the hottest point carries over nicely to the discrete setting. We again solve Equation 3.7 and set $\mathbf{h}$ as described in Section 3.2. Then we just need to iterate over the elements in the solution vector $\mathbf{u}$ to identify the center. As before, the system matrix can be factored in a preprocess, so finding the center amounts to performing back substitution.

With this procedure in place, we are able to show experimentally that the heat center indeed converges to the centroid in the plane as t grows. For this we generate a tessellation of the unit circle similar to the sphere used before. Then we generate polygons in two different ways: 1) We start a random walk over the vertex graph from the center. Triangles are included if all three vertices have been visited. This leads to irregularly shaped, non-convex polygons, which are not necessarily simply connected. 2) We select vertices inside differently sized and oriented ellipses. These polygons are simply connected, well shaped but not strictly convex. The triangles comprising the polygons define $\mathbf{h}$ and solving for $\mathbf{u}$ yields the center.

In order to properly simulate the behavior of the heat function for large t it is necessary to use a higher order (e.g. Padé) approximation, amounting to performing several time steps: for this we solve for $\mathbf{u}$ based on the initial value $\mathbf{h}$ and then iterate $\mathbf{h} \leftarrow \mathbf{u}$ in Equation 3.7. Note that we only use a higher order approximation for the sake of this experiment. Figure 3.3 shows the result. The horizontal line depicts the error that would result from picking the vertex closest to the true centroid. We see that, indeed, the discrete solution converges to this optimum value as t grows, however, convergence requires several time steps. This behavior should be expected, as the similarity between centroid and heat center is non-linear. The solutions tend to get worse for larger values of t because of boundary effects: the approximation assumes an infinite plane, and once values start to be nonzero at the boundary the approximation is distorted. The two plots suggest that Euclidean centroids are matched for smaller time values t if the polygon is well-shaped (i.e. roughly convex and symmetric).

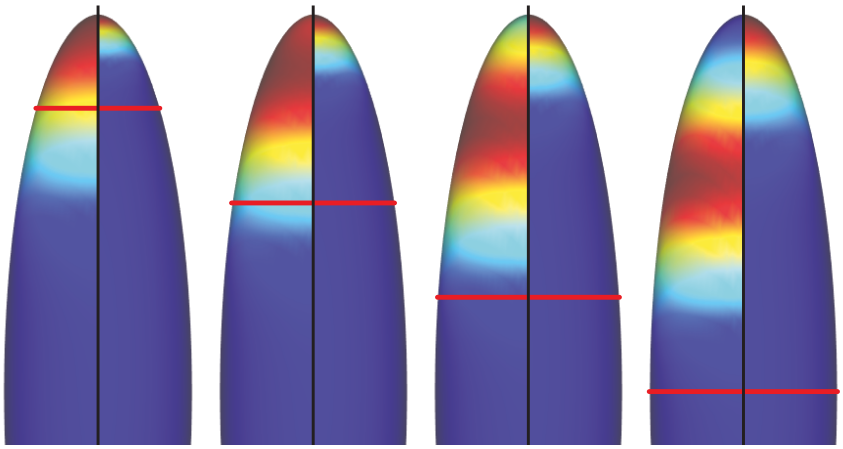


Figure 3.4: *The functional defining the center on an elongated ellipsoid. Cells are defined by cutting the ellipsoid orthogonal to the long axis (cell boundaries are marked in red). The left part shows the sum of squared geodesic distances, the right part diffusion values. Color coding is chosen so that the center is in the red area. Already for moderately large cells, geodesic distances become non-unique and the center(s) moves away from the tip. Diffusion naturally defines the tip to be the center.*

Smoothness and uniqueness Defining the centroid based on geodesic distance leads to problems for larger non-Euclidean cells: given a point $\mathbf{x}$ on the surface, the *cut-locus* of $\mathbf{x}$ is the set of points that have more than one shortest geodesic. This means the function of geodesic distances to $\mathbf{x}$ is non-smooth at the cut locus. This has an important numerical consequence: consider a convex cell around $\mathbf{x}$. Then the geodesic distance function is *not convex* if the cell contains points of the cut locus which means that the centroid is not necessarily unique.

Note that the definition of the center (Equation 3.23) considers all points in a cell. This means the integrand is non-convex if the cut locus of *any of the points in the cell* intersects the cell. Computing the minimum of non-convex function is generally difficult. The fact that the function is not smooth also has consequences: small perturbations of the cell boundary may lead to large, non-continuous jumps of the centroid of the cell.

We illustrate this behavior in Figure 3.4. Here we define a cell by cutting an elon-

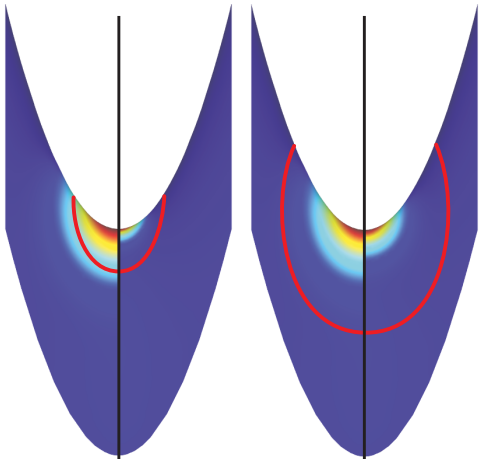


Figure 3.5: *The saddle point is correctly identified as centroid for two different cells (red) using heat diffusion. The left side of each image illustrates the sum of squared distances used to define the Riemannian center of mass* (3.23).

gated ellipsoid orthogonal to the long axis. The left side of each ellipsoid shows the sum of squared geodesic distances, i.e. the functional defining geodesic centroids. The color coding represents the small values with red, so the geodesic centroid is located in the red area. The right side shows the heat functional, color coded with red representing large values. So, again, the center is in the red area. As the cell gets larger, the geodesic functional has a ring of maxima around the center axis and not just fails to uniquely identify a center but also provides a rather unintuitive result. The heat center functional, perhaps not surprisingly given the setup, identifies the tip of the ellipsoid as the center in all cases.

The better behavior of heat centers can be expected in much more general situations: the fundamental solution to the heat function is log-concave in Euclidean domains for all t, and this also holds for positively curved Riemannian manifolds (see [68] for recent results in this area). It follows from the Précopa-Leindler theorem [9] that the integral of log-concave functions over a convex domain is log-concave. So for convex cells with positive curvature the heat centroid is unique. Smoothness of the function also implies it is smoothly varying with the cell bound-

ary; this is useful for optimization with smoothly varying cells.

It is known that the estimates on log-concavity are only slightly perturbed for negative curvature [45]. This means while we may not have a guarantee for uniqueness of the centroid, the functional is always close to be log-concave. Indeed, we have failed to generate examples (with simple convex boundary) that would exhibit more than one local maximum. A generic example of the functional on a saddle for two sizes of a cell symmetric around the saddle point is shown in Figure 3.5. Note that the extremum is in the saddle point, as expected, for geodesic distances as well as diffusion.

3.6 Localized solving

As the mesh in our application is static and the linear system symmetric and positive definite, we compute a Cholesky factor of the matrix as a preprocess. Then the dominating operation for the various solves are forward and back substitutions. We show that forward substitution is generally efficient and the resulting vector is sparse. While back substitution is also quite fast, it still requires looping over all vertices in the mesh. This makes sense, as the solution of heat diffusion is global, i.e. heat values are nonzero everywhere. However, in our context, we typically need the solution only in a certain area close to the heat source to compute bisectors or centroids. We can obtain an approximate solution by setting heat values far away from the source explicitly to zero since they will be relatively small anyways. This way the runtime of back substitution is proportional to the number of values we are actually interested in and not the size of the full mesh.

Selective Cholesky Solve Let $\mathbf{L}\mathbf{L}^\mathsf{T} = \mathbf{M} - t\mathbf{L}_C$ be the (lower triangular) Cholesky factorization[1] of the system matrix computed using a modified version of LDL [19]. Solving for the right-hand side $\mathbf{b} \in \mathbb{R}^{n_v}$ requires solving $\mathbf{L}\mathbf{a} = \mathbf{b}$ with an auxiliary vector $\mathbf{a} \in \mathbb{R}^{n_v}$ and subsequently solving $\mathbf{L}^\mathsf{T}\mathbf{u} = \mathbf{a}$. It can be shown that the auxiliary vector $\mathbf{a} \in \mathbb{R}^{n_v}$ is usually sparse if $\mathbf{b}$ is sparse. A more

[1] We follow the usual convention to refer to the Cholesky factor as $\mathbf{L}$. This matrix is not to be confused with the Laplacian. The cotan Laplacian will be written as $\mathbf{L_C}$.

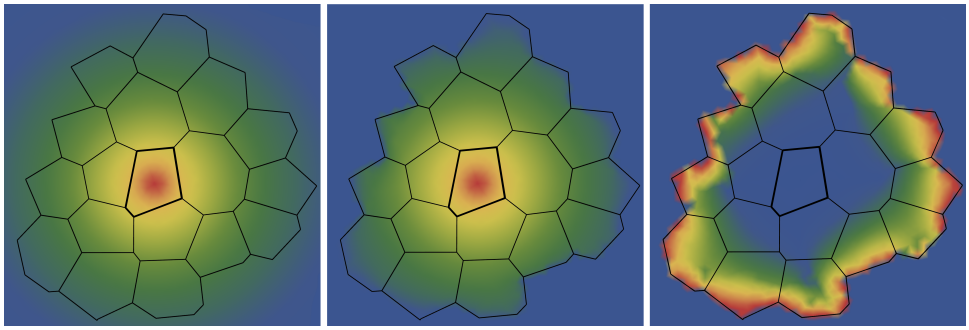


Figure 3.6: *The central Voronoi cell is computed using the heat method. The left image shows the heat distribution, the middle image the result of the selective solve, which restricts heat computation to a neighborhood around the cell. The right image illustrates the logarithm of the (relative) distance between the functions. The mean error is* 1.8% *(max* 16%*).*

detailed discussion of this fact and Cholesky factorizations in general is deferred to the next chapter.

The solution to $\mathbf{L}^\mathsf{T}\mathbf{u} = \mathbf{a}$ is dense for a connected mesh because heat diffusion is a global process. This is unfortunate because it would mean that computational complexity is at least linear in the number of vertices of the whole mesh, even though we are only interested in values relatively close to the source. To alleviate this problem, we use the fact that values far away from the heat source will be close to zero. Setting them to zero explicitly results in an approximate solution but allows us to fix the sparsity structure of the solution $\mathbf{u}$.

More formally, we define the sparsity of $\mathbf{u}$ by an index set $\mathcal{I}$, i.e. allowing nonzero coefficients only for u_i, $i \in \mathcal{I}$. Because the upper triangular matrix $\mathbf{L}^\mathsf{T}$ is given in a row-major sparse matrix format, solving $\mathbf{L}^\mathsf{T}\mathbf{u} = \mathbf{a}$ can be easily restricted to the index set $\mathcal{I}$ by ignoring rows in the complement of $\mathcal{I}$ during back substitution. As a result, the complexity of selective back substitution depends on the number of elements in $\mathcal{I}$ and not on the number of vertices of the mesh. Algorithm 2 details this selective back substitution algorithm. We integrated sparse forward substitution and selective back substitution in our sparse Cholesky solver based on the LDL

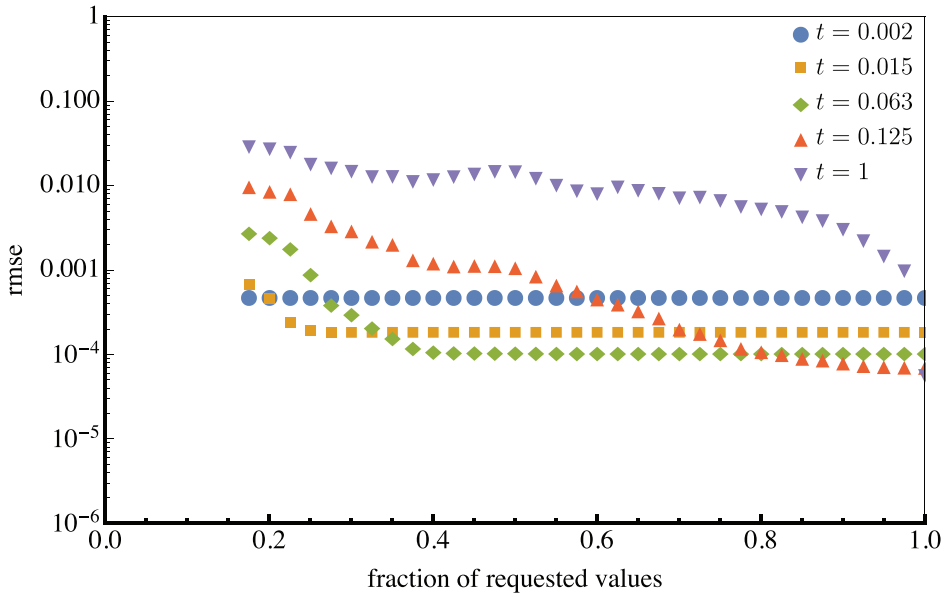


Figure 3.7: *Using the approximate sparse solve introduces error which depends on the neighborhood for which values are requested and the diffusion time parameter t.*

Input:

- Cholesky factor $\mathbf{L}^T \in \mathbb{R}^{n \times n}$ in row-major format (I_R, I_C, V).
- Sparse right-hand side $y \in \mathbb{R}^n$.
- Index set $\mathcal{I}$ sorted in descending order.

Output: Sparse approximate solution y.

```
for k in I do
    for p from I_R[k] to I_R[k + 1] do
        y[k] ← y[k] − V[p] ∗ y[I_C[p]]
```

Algorithm 2: *In the row-major sparse format the matrix is represented by three arrays. The row array I_R contains the position of the first index of each row in the column array I_C, where the position of all nonzeros per row are stored. The array V contains for each column position in I_C the respective matrix entry.*

package [19].

Numerical experiments Measuring the performance gain using the selective solve method, we find that the performance is almost perfectly linear in the number of requested result values. For error measurement we use again the irregularly sampled unit spheres. This time we generate a Voronoi diagram of moderate complexity, each Voronoi cell covering about 17% of all vertices, and measure the error on all intersections of triangle edges with Voronoi edges. We restrict the set of vertices used for computing the heat distribution for a particular generator: let $\mathbf{p}$ be a generator, then we set all values that are outside a neighborhood covering a certain amount of vertices to zero by skipping the corresponding rows as described in algorithm 2. The result of this experiment is illustrated in Figure 3.7. Clearly, fixing no value at zero gives the most accurate result. Choosing smaller neighborhoods introduces more error. The amount of error is also linked to the time parameter t. For smaller values we know that heat decays faster with distance to the source. Consequently setting values away from the source to zero is a better approximation for small values as compared to larger values of t. For very small values of t the solution is indistinguishable from the full solution as soon as all vertices in the eventual Voronoi cells are being evaluated. This suggest that using smaller values

of t and local solves are a good choice in practice.

3.7 Applications

Intrinsic Voronoi diagrams can be used in a variety of applications. We focus on two scenarios that particularly benefit from smoothness and efficiency of the computation. We have found that the results in these applications are largely independent of the choice of t and that there is no significant benefit of performing several implicit time steps. We have followed the suggestion by Crane et al. [17] and use $t = h^2$, where h is the mean edge length, in all of the following experiments.

3.7.1 Centroidal Voronoi tessellations

Computing an (approximate) centroidal Voronoi tessellation using Lloyd's algorithm proceeds by alternating two steps. Starting with a set of n generators $\mathbf{p}_i^\circ \in \mathcal{M}$ one iterates:

1. constructing the Voronoi tessellation of $\mathbf{p}_1^k, \ldots, \mathbf{p}_n^k$;
2. setting $\mathbf{p}_i^{k+1}$ to the centroids of their respective Voronoi cells.

Each of the two steps can be performed as explained in the previous sections. In the following, we provide details on how to adjust the techniques to the specifics of Lloyd's algorithm. We discuss the results and extend the technique to anisotropic diagrams.

Determining the relevant neighborhood Using the selective solve requires defining a local neighborhood. In the context of Lloyd's algorithm we exploit the Voronoi tessellation of the previous step and use a union of adjacent Voronoi cells as neighborhood. When computing heat diffusion for centroid or distance computation of a cell C_i, we select all adjacent Voronoi cells of this cell. For increased robustness we also include 2nd degree neighbors. The error due to the selective solve is small as illustrated in Fig. 3.6 and in particular, is concentrated

Model	$\lvert\mathcal{V}\rvert$	Generators	T_{fac} (m:s:ms)	Mem (MB)	min. φ_m (deg)	$\overline{\varphi_m}$ (deg)	T/iter (s:ms)
Klein-Bottle	35k	100	0:00:076	14	36.1	52.3	00:660
Bimba	75k	1,000	0:00:202	34	33.7	52.0	00:289
Casting	109k	1,000	0:00:437	54	23.2	50.6	00:564
Dragon	153k	1,500	0:00:479	71	27.6	51.6	00:786
Buste	297k	2,000	0:01:324	149	35.7	52.8	01:951
Kitten	442k	2,000	0:03:069	244	38.4	53.6	03:536
Fertility	499k	2,000	0:03:290	277	35.6	53.2	03:646
Happy	608k	3,000	0:04:397	330	19.0	51.4	07:187
Pierrot	641k	1,000	0:06:011	367	33.8	52.8	03:838
Buddha	669k	2,000	0:06:348	388	28.4	51.7	06:342
Pensatore	886k	2,000	0:09:869	529	33.4	53.0	10:969
Isidore	4.2M	4,000	2:10:836	2867	23.2	50.6	84:000

Table 3.7.1: *Statistics for results presented in Fig. 3.14. All timings were taken on an Intel 3.9 GhZ Core i7 processor using four threads. T_{fac} stands for the time taken during prefactorization. 'Mem" reports the amount of memory required for storing the Cholesky factorization. φ_m is the vector of the smallest interior angles per triangle of the Delaunay triangulation and $\overline{\varphi_m}$ the mean of this angle over all faces. The time T/iter represents the average time for each iteration (construction of Voronoi diagram* and *finding cell centroids). In all cases we ran the algorithm for 100 iterations.*

on the boundary of the region and negligible in the area of interest. In the initial step, we estimate an appropriate radius for each generator and compute the local neighborhood using Dijkstra's algorithm.

Centroids on faces The discrete energies we consider are all based on piecewise linear approximation. This means maxima, and therefore centroids, necessarily coincide with vertices. For convergence of Lloyd's method it is beneficial to move the generators smoothly on the surface over the course of the procedure. A possible solution would be to use higher order finite elements to solve the heat equation, yet this would increase storage and computation requirements of the factorization.

For the purpose of Lloyd's algorithm, we decided to smooth the trajectories of generators by locally fitting bivariate quadratic polynomials to the piecewise linear heat function in the vicinity of the maximum $\tilde{\mathbf{c}}_i$ and to compute the centroid as its unique maximizer.

More specifically, we first determine the heat centroid $\tilde{\mathbf{c}}_i$ as detailed in the previous section. Then we select all faces incident to the vertex $\tilde{\mathbf{c}}_i$ and project them onto the tangent plane at $\mathbf{c}_i$ as defined by the vertex normal at this point. Any choice of vertex normal could be used here, we decided to use the area weighted face normals. Next, we fit a bivariate polynomial to the heat values at the projected vertices. Finally, we determine the triangle that contains the maximum of the polynomial and map this point back to the geometry using barycentric coordinates. If the maximum is not contained in a triangle we use the point on one of the triangles that is closest to the solution.

Even though this is just a local approximation, the quality of the resulting CVT is significantly increased using this method as generators move freely on the surface. We observed an increase of the quality metric $\overline{\varphi_m}$ (see below) of up to 2°.

Parallelization The bulk of computation time ($> 95\%$) is typically spent during back substitution to solve the heat equation. This step can be easily parallelized

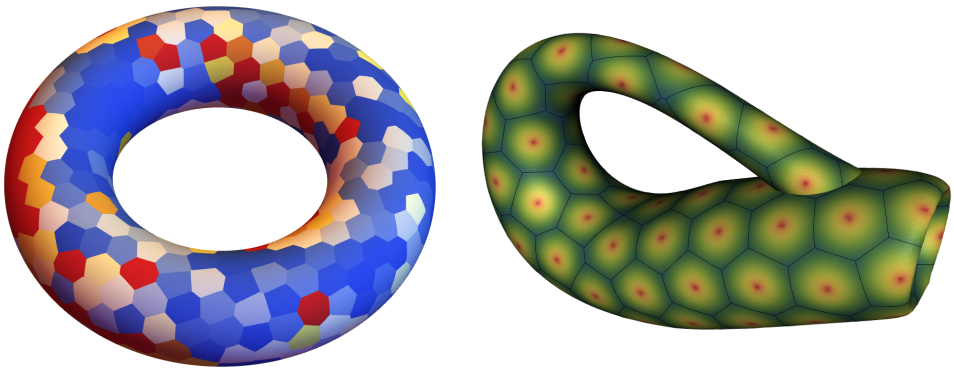


Figure 3.8: *Left: the accuracy of our method is not systematically affected by curvature variation. Right: Our fully intrinsic approach enables the computation of Vornoi diagrams and centroids on surface that can not be injectively embedded.*

since we solve the equation for each centroid (or Voronoi cell) independently. We use `std::thread` for parallelization.

Results We tested our approach on meshes varying in vertex count, triangle quality and curvature. For each mesh we generated a CVT from a random initial distribution of the generators. The quality of the CVT can be measured based on the interior angles of its dual triangulation. For easy comparison with related work we also report the mean and minimum of the *smallest* interior angle per triangle. Table 3.7.1 shows that the resulting tessellations are similar in regularity compared to other approaches. Since the results depend on random initialization, we report the best out of 5 runs with respect to mean interior angle.

In terms of computational efficiency, we note that we have been able to generate an intrinsic CVT on meshes with larger vertex count than reported so far in the literature. To provide a rough comparison to Wang et al. [105] we have run our algorithm on the ARMADILLO model with the same number of generators on comparable hardware (2.6GHz Intel i5, four cores) and found a three-fold speed-up of the time needed per iteration (0.82*s* vs. 2.12*s*). Quality metrics in this experiment turned out to be very similar.

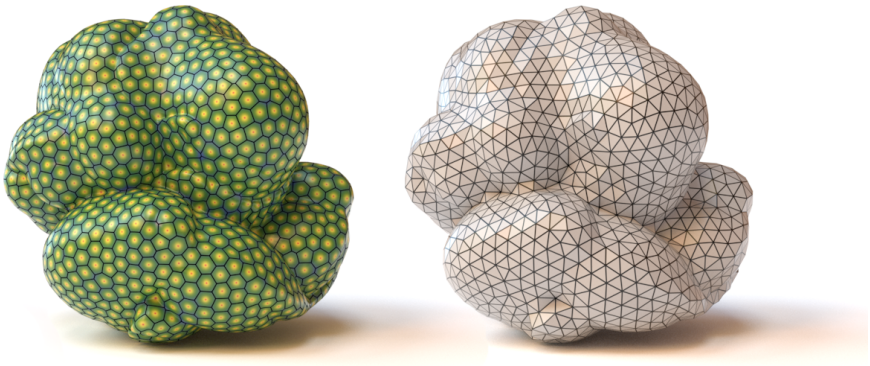


Figure 3.9: *Dual Delaunay triangulation of a CVT computed using our Algorithm.*

The diffusion CVT shows no significant dependence on curvature. To demonstrate this, we have generated 20 and 500 sites on a torus and a sphere (similar mesh resolution) scaled to unit area. After 100 iterations we measured the variance in cell area and obtained values on the order of 10^{-6} for 20 sites and 10^{-8} for 500 sites, for both the torus and the sphere. The inset shows the torus with the sites color coded based on the squared difference to average area. There is apparently no systematic dependence of area on curvature.

The intrinsic nature of the computations allows applying the algorithm to surfaces that contain self intersections, i.e. that are not properly embedded. This is important as many meshes encountered in practice contain self intersections. Figure 3.10 shows an example of a CVT on an instance of the Klein bottle, which is an example of a surface that cannot be embedded in $\mathbb{R}^3$ without selfintersections.

Meshing The regularity of a CVT induces dual Delaunay triangulations with close to regular triangles (see Figure 3.9). This makes our approach attractive for computing high-quality base meshes as needed e.g. in semi-regular remeshing [44]. As mentioned, the intrinsic computation makes the meshing robust against the geometric properties of the embedding, such as proximity of different parts of the surface. For generating meshes of high resolution the source mesh might need to be upsampled, as the Voronoi diagrams cannot have more cells

than the underlying mesh has vertices.

On the other hand, intrinsic computations are costly compared to distance computations in Euclidean space. For surfaces that are simple enough extrinsic approaches [110] would be more appropriate.

Anisotropic centroidal Voronoi tessellations The Laplacian is an intrinsic operator and the cotan discretization consequently only depends on intrinsic properties such as edge lengths and angles. This means our approach is independent of the ambient space and directly extends to higher co-dimension. We exploit this feature to get anisotropic CVTs that consider distances in positions as well as normals: For a mesh with vertices $\mathbf{p}_i$ and corresponding vertex normals $\mathbf{n}_i$ the mapping $f_\lambda : \mathcal{M} \mapsto \mathbb{R}^6, \mathbf{p}_i \to (p_i^0, p_i^1, p_i^2, \lambda n_i^0, \lambda n_i^1, \lambda n_i^2)$ lifts the mesh $\mathcal{M}$ into $\mathbb{R}^6$ [11]. $\lambda \in \mathbb{R}$ controls the relative influence of the normal information. In this immersion, intrinsic distances also account for normal variation. Note that f_λ is not necessarily injective and the mesh will generally have many self intersections.

We construct anisotropic centroidal Voronoi tessellations by evaluating Voronoi cells with respect to the immersion induced by f_λ and cell centroids using the diffusion metric. To fix the relative importance of normals in this computation we normalize all meshes to a unit bounding box. In this setting we chose a value of $\lambda = 0.05$. Figure 3.10 shows to results that demonstrate how edges align with high-curvature regions.

3.7.2 Natural neighbor coordinates

Voronoi diagrams can be used to define barycentric coordinates on meshes using natural neighbor coordinates [91] which are useful e.g. to interpolate data defined at fixed points.

Let $\mathbf{p}_1, \ldots, \mathbf{p}_n$ be points in the Euclidean domain. Barycentric coordinates $b_i(\mathbf{q})$ at a point $\mathbf{q}$ have to fulfill the following requirements:

Positivity: $b_i(\mathbf{q}) \geq 0$ for all i.

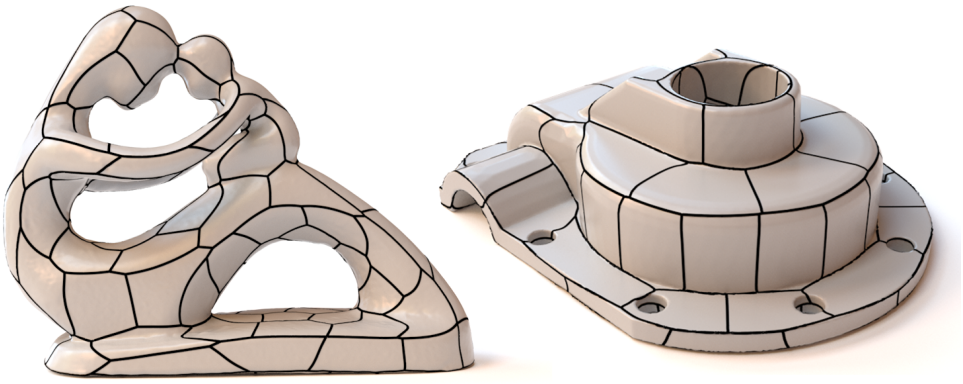


Figure 3.10: *Intrinsic Voronoi diagrams enable interesting applications. Anisotropic CVTs are generated by immersing the surface into $\mathbb{R}^6$ using positions and normals. For non-convex surfaces, this leads to self intersections. The Voronoi diagram on the Klein bottle has been computed in $\mathbb{R}^3$, where no embedding of the surface exists.*

LAGRANGE PROPERTY: $b_i(\mathbf{p}_j) = \delta_{ij}$.

PARTITION OF UNITY: Coordinates at a point sum to 1, i.e. $\sum_{i=1}^{n} b_i(\mathbf{q}) = 1$.

LINEAR PRECISION: The point is the weighted average of the fixed sites:

$$\mathbf{q} = \sum_{i=1}^{n} b_i(\mathbf{q}) p_i.$$

Natural neighbor coordinates define barycentric coordinates using Voronoi diagrams. The basic idea is to first form the Voronoi diagram of the fixed sites $\mathbf{p}_i$ and compute the area of each Voronoi cell a_i. The barycentric coordinates of a point $\mathbf{q}$ are determined as follows: Construct a new Voronoi diagram with the added site $\mathbf{q}$ and compute the cell areas $\tilde{a}_i$ and a_q respectively. The coordinates are now defined as $b_i(\mathbf{q}) = (a_i - \tilde{a}_i)/a_q$. This definition motivates the alternative name *area stealing coordinate*.

Given our fast approximation of Voronoi diagrams on meshes natural neighbor coordinates can be readily generalized to triangle meshes, however, for non-planar meshes linear precision can not carry over to curved domains. Figure 3.11 illustrates barycentric interpolation of texture coordinates. Out method is relatively slow compared to previous work. Rustamov [82] generalizes 2d barycentric coor-

Figure 3.11: *Natural neighbor coordinates can be used to perform texture mapping.*

dinates to the surface case by using the exponential map. This method is limited to sites forming polygons whereas our method can interpolate between arbitrary data points.

Panozzo et al. [76] compute approximate geodesic distances by mapping the mesh to a higher dimensional space where Euclidean distances mimic geodesic distances on the original mesh. This approximation is used to define barycentric coordinates variationally. The resulting algorithm is very fast and robust, however, the precomputation is quite heavy and the approximation might fail. Moreover, negative weights might occur for coarse samplings. Our algorithm has faster precomputation time and can guarantee positive weights at the cost of much slower evaluation.

Blend skinning Another application of barycentric coordinates is blend skinning. Given a set of point handles on a mesh (usually structured in a skeletal hierarchy), the mesh should deform when the handles are moved. The influence of each handle on a given point can be computed using barycentric coordinates. To average transformations given barycentric weights we use dual quaternion skinning [54] as implemented in libigl [50]. Figure 3.12 shows skinning weights com-

Figure 3.12: *Voronoi diagram of handles (left), two weight functions (center), posed model (right).*

puted using the described method and a posed model.

One of the favorable properties of natural neighbor coordinates is their locality. Weights of distant handles are guaranteed to be exactly zero. This feature sets natural neighbor skinning apart from competing methods like Bounded Biharmonic Weights [49] which uses non-linear optimization to compute a set of weights with all required properties. Our method, albeit easy to implement, is comparably slow. For seven handles on the armadillo with 55102 vertices (Fig. 3.12) computing blending weights for a specific vertex takes about 1.1 ms on a 3.9 GHz CPU with four cores, which adds up to about a minute for all vertices. The main reason for this long computing time is the fact that we can not efficiently localize the computation of the heat kernel if we are interested in a very coarse diagram. Natural neighbor coordinates could be made useful in practice by computing the weights on a coarse mesh and subsequently propagating them to the original version.

3.8 Conclusion

We defined Voronoi cells and centroids based on heat diffusion. This enables stable and very efficient computation on various discrete surfaces. Since diffusion is based on the Laplace operator, our algorithms extend to any domain that al-

lows the definition of a Laplacian like polygonal meshes [1] and point clouds [64]. Computing CVT's on point sets using the operator defined in [64] immediately allows us to reconstruct surfaces from points using the dual Delaunay triangulation (see Figure 3.13).

While we have not demonstrated this, the computation of Voronoi tessellations extends to arbitrary generators such as lines or more general shapes. This would allow to efficiently implement corresponding centroidal Voronoi tessellations [67].

An important ingredient for efficiency is the localized back substitution. Back substitution for prefactored Laplace operator matrices is a part of many geometric processing algorithms. Consequently, we believe that many other algorithms could benefit from this idea. As an example, an important ingredient in many recent geometry processing techniques [16, 17, 94] involves solving constant linear systems based on the Laplacian and local non-linear operations. Localized back substitution would be applicable whenever the desired solution is restricted to a local area in the mesh.

So far we have restricted our analysis to plain heat diffusion. It seems promising to use the solutions of other PDEs [4] for defining Voronoi cells or centroids – efficiency of computation would be similar to the heat method as long as a sparse linearization is available.

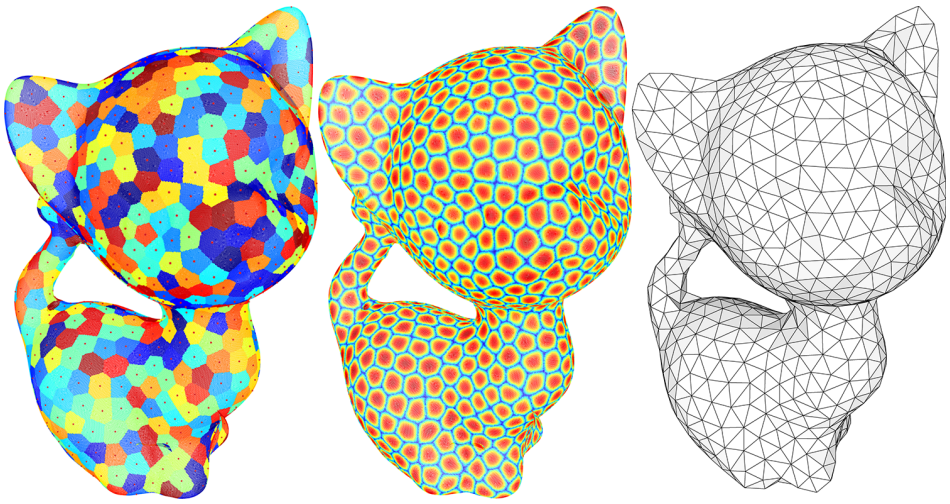


Figure 3.13: *Meshing a point cloud by generating a CVT followed by dualization.*

Figure 3.14: *Results obtained with our algorithm. See Tab. 3.7.1 for timings and statistics.*

4

Localized solutions of sparse linear systems

4.1 Introduction

The central operation in many geometry processing tasks is the (repeated) solution of a sparse linear system [7, 8, 12, 25, 26, 59, 94]. These problems can be efficiently solved by computing a (sparse) factorization of the system matrix and then performing a step of forward and back substitution. This is particularly attractive if several solves against varying right-hand sides are necessary.

Computing a solution vector given the factorization is quite efficient if the factors are sparse, but solving for a simple regular 2d mesh with n vertices still has a time complexity of at least $\mathcal{O}(n \log n)$ [35]. However, many geometry processing operations are local, i.e. a solution to the linear system is required only for a subset of

the mesh, e.g. for generating a parameterization of a patch. Our main contribution lies in describing a method that uses a global factorization to compute solutions for subsets of mesh vertices in time that depends mostly on the size and structure of the subset and not the whole mesh. This approach is favorable to the common solution of setting up a new system for the subset of the mesh and factorizing it.

In order to present our approach we first provide some background on sparse Cholesky factorizations in Section 4.2. Then we explain how sparsity in the right-hand side as well as in the desired solution can be exploited. Sparsity in the right-hand side leads to sparse intermediate solutions – this is commonly exploited in libraries for numerical linear algebra. Our contribution lies in exploiting 'sparsity' in the desired solution. By sparsity we mean that only a subset of the solution vector is desired – while the solution may still be dense, i.e. nonzero in all its elements. We explain how this is possible without resorting to an approximation: the solution for the desired elements is exactly the same as if all elements of the solution were computed. An important point of our analysis and the resulting algorithm is that the sparsity in the solution can be exploited independently from sparsity in the right-hand side (if any). In addition, we explain how our implementation exploits vectorization and cache coherence. The details of the analysis and resulting algorithms are described in Section 4.3.

We first evaluate the effectiveness of the main operations at the common example of generating a conformal parameterization with prescribed boundary. Our approach follows the now classic idea to generate a harmonic embedding [26] and requires only a single linear solve. This allows us to compare the time required for numerical solutions based on the local patch or using the sparse localized solve based on the prefactored global system. We find that our approach provides a significant speed-up in all practical scenarios.

Next we analyze the behavior for extremely sparse vectors at the example of discrete heat diffusion. Here, we assume that the heat source is local (i.e. a single vertex). Then we demonstrate the gain in efficiency if only a single value is desired, either far away or in the vertex itself (the case of auto-diffusion, which is used for many signatures and matching [38, 97]). Both cases show significantly reduced

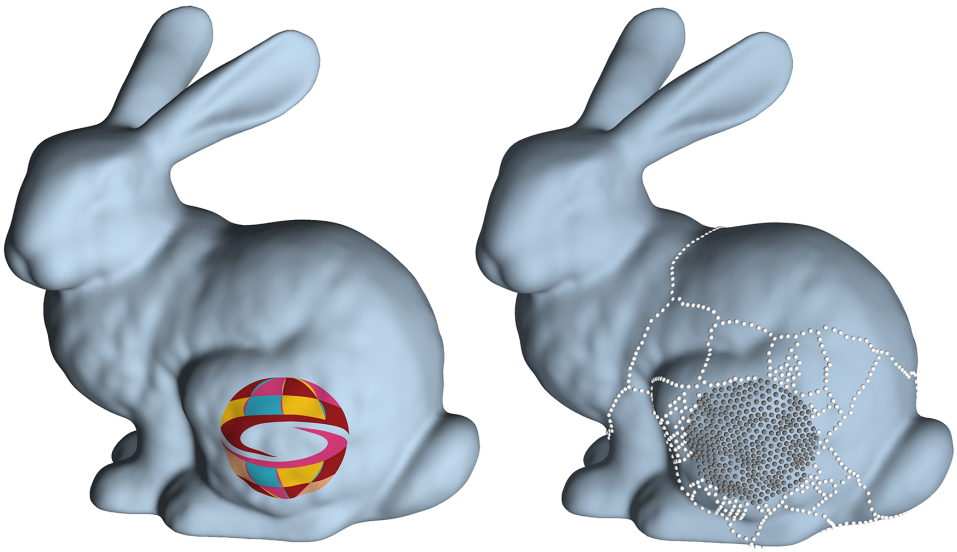


Figure 4.1: *To parameterize a small surface patch (left) a global factorization of the Laplacian can be utilized. For the computation of the exact solution for patch vertices (gray vertices on the right) not all columns of the Cholesky factor have to be considered. More specifically, only those associated with paths in the elimination tree connecting root and patch-vertices are required additionally (white vertices).*

computational cost compared to standard solutions; the scenario of different locations for source and desired heat value allow us to analyze the benefit of exploiting the sparsity in the solution independently from the sparsity in the right-hand side (see Section 4.6).

4.2 Background

The central operation we are concerned with are solutions of linear systems that are the result of discretizing differential equations over the domain of a triangle mesh $\mathcal{M} = (\mathcal{V}, \mathcal{F})$.

A large number of geometry processing operations involve discrete differential operators. There are many alternatives on how to derive the matrix representations of these operators [1, 31, 72, 79]. The important point is that in all constructions the operators are built from the adjacency structure of the mesh, and the sparsity

pattern is identical to a low order power of the adjacency matrix. Moreover, in most cases the system is, or can be made, symmetric. In other words, the common task is to solve a sparse symmetric system of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ represents values attached to the mesh elements, most commonly the vertices, in which case $\mathbf{A} \in \mathbb{R}^{n_v \times n_v}$.

Sparse Cholesky factorization If the resulting sparse symmetric linear system is positive definite it can be solved using Cholesky factorization

$$\mathbf{A} = \mathbf{LDL}^\mathsf{T} \quad \text{with } \mathbf{L}, \mathbf{D} \in \mathbb{R}^{n_v \times n_v}. \tag{4.1}$$

Here, $\mathbf{D}$ is a diagonal matrix and $\mathbf{L}$ is a lower triangular matrix with unit diagonal commonly called the *Cholesky factor*. Since $\mathbf{A}$ is positive definite all entries of the diagonal matrix $\mathbf{D}$ are positive. The diagonal entries of $\mathbf{L}$ will all have the value one. The alternative definition

$$\mathbf{A} = \mathbf{L}'\mathbf{L}'^\mathsf{T} \quad \text{with } \mathbf{L}' = \mathbf{LD}^{1/2} \tag{4.2}$$

is equivalent to the first one but has some practical advantages when applying so called supernodal methods that will be used in the next chapter. Keeping the explicit matrix $\mathbf{D}$ has the advantage that matrices that are not numerical positive definite, i.e. possesses an eigenvalue that is numerically not positive, can also be handled to a certain extend. In this chapter we will therefore proceed with this variant.

The Cholesky factorization has many favorable properties, for example it is unconditionally stable, allowing permutations for optimizing other properties. In our context a permutation can be selected with the goal to reduce *fill-in*. A fill-in entry is a nonzero that appears in $\mathbf{L}$ but not in $\mathbf{A}$. In particular, for the adjacency structures resulting from triangle meshes it is usually possible to generate sparse Cholesky factors using an appropriate ordering.

Elimination tree The sparsity of the Cholesky factor is important in our context because it leads to efficient solves. Solving a system for $\mathbf{x} \in \mathbb{R}^{n_v}$ given a right-hand

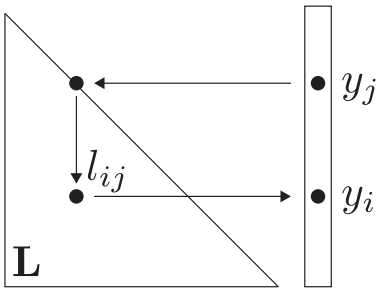


Figure 4.2: *If we know the value y_j to be nonzero and l_{ij} is a nonzero as well we know that y_i has to be a (structural) nonzero.*

side $\mathbf{b} \in \mathbb{R}^{n_v}$ requires a forward substitution (or forward solve) step to compute the auxiliary vector $\mathbf{y} \in \mathbb{R}^{n_v}$

$$\mathbf{Ax} = \mathbf{b} \tag{4.3}$$

$$\Rightarrow \quad \mathbf{LDL}^\mathsf{T}\mathbf{x} = \mathbf{b} \tag{4.4}$$

$$\Rightarrow \quad \mathbf{LDy} = \mathbf{b} \tag{4.5}$$

followed by the back substitution $\mathbf{L}^\mathsf{T}\mathbf{x} = \mathbf{y}$.

Now consider computing the forward solve. The elements of the solution $\mathbf{y}$ are computed in order of increasing index, i.e. $y_0 = b_0/d_{00}$ is computed first, then y_1, and so on.

For a sparse right-hand side $\mathbf{b}$ the vector $\mathbf{y}$ is usually also sparse and knowing the nonzero structure of $\mathbf{y}$ in advance would significantly improve performance — columns in $\mathbf{L}$ corresponding to a zero in $\mathbf{y}$ can simply be ignored.

In what follows we refer to elements as *nonzero* if they potentially take on values different from zero, because their computation depends on other nonzero elements. It is still possible that nonzero elements carry a zero value due to numerical cancellation. Zero values of this kind are commonly not exploited algorithmically.

The key observations for the nonzero structure of $\mathbf{y}$ are the following: (1) if b_i is nonzero then y_i is nonzero (because $\mathbf{D}$ has to be full-rank), and (2) if y_j is nonzero

and there is a nonzero l_{ij}, then y_i will be nonzero, as illustrated in Figure 4.2. Consequently we can find the nonzero structure of $\mathbf{y}$ by traversing the graph of $\mathbf{L}$ that has an *oriented* edge (j, i) for each nonzero l_{ij}. Note that all edges in the graph are directed from a node with smaller index to a node with larger index, reflecting the computation of elements in increasing order of index. The traversal is started in the nodes corresponding to the nonzeros in $\mathbf{b}$, following observation (1). Then the graph is traversed following the oriented edges, reflecting observation (2). The nonzero structure of $\mathbf{y}$ is given by the nodes that are visited during this traversal [37]. If the elimination tree is fairly balanced we can expect the vector $\mathbf{y}$ to be sparse if $\mathbf{b}$ is sparse, albeit with some fill in.

The graph mentioned above is a useful tool for quickly identifying the columns in $\mathbf{L}$ that need to be considered during the solve. To store the graph without compromising on the possibility to quickly identify all dependencies a particular spanning tree of the graph is used: the *elimination tree*. This tree results from keeping only one *outgoing* edge for each node. In particular, in node i we pick the edge (i, j) with the smallest index j. By retaining only one edge per node and since all edges go from nodes with smaller index to larger index, it is clear that in the resulting graph there is at most one path from a node with smaller index to one with higher index. But why is it connected? It is a specific property of the Cholesky factor that whenever node i has outgoing edges (i, j) and (i, k) with $j < k$, then node j has the outgoing edge (j, k). We refer the reader to Liu [62] and Davis [18] for an explanation and proof of this property.

The elimination tree not just stores the connectivity of the graph, it also allows identifying the dependence among nonzero elements by starting the traversal in any of its nodes. By traversing only along the spanning tree we are guaranteed to reach all nodes we would find when traversing the full graph. A space and runtime efficient representation is to simply store for each index j its parent index $\pi(j)$ in the elimination tree.

By exploiting this representation of the elimination tree, computing the nonzero pattern of $\mathbf{y}$ takes an amount of time proportional to the number of nonzeros in $\mathbf{y}$. Simply traverse the tree up to its root starting at all indices that contain a nonzero

in **b** and mark all visited notes. Figure 4.4 illustrates an elimination tree. Consider a right-hand side **b** with nonzeros in positions 1 and 5. Traversing the tree informs us that the vector **y** will have nonzeros in positions $1, 4-8$ and $17-20$. In this example there are nonzeros only in roughly half of the entries, however, for larger trees this ratio will be significantly smaller.

The tree itself can be directly constructed from the input matrix and will be available anyway because it is used during the construction of the Cholesky factor itself and determines its nonzero structure.

Ordering: nested dissection An important factor in the efficiency of computing the intermediate solution for sparse right-hand sides is the (average) length of the sequence $(j, \pi(j), \pi(\pi(j)), \ldots)$ or, in other words, the height of the elimination tree. Of much lesser importance, in our context, is the fill-in: note that a zero fill-in structure might still have an elimination tree with worst case height. The sequence $\pi(j) = j + 1$, for example, corresponds to a dense subdiagonal and would imply that starting from column j all following columns are visited, which is similar to the situation for dense matrices, so the number of columns considered are linear instead of logarithmic.

To limit the height of the elimination tree one can exploit a crucial property of Cholesky factors: If a_{ij} is the leftmost entry in row i of the matrix **A**, all values l_{ik} with $k < j$ in the factor **L** are guaranteed to be zero. For the example in Figure 4.3 this implies that the block in rows $1-8$ will not be connected to block $8-16$ in the matrix graph of **L**, therefore these blocks reside on separate subtrees in the elimination tree. This property can be brought to use by reordering the matrix columns and rows such that blocks are grouped together. In Figure 4.3 (upper right) a *separator* consists of nodes $17, 20, 18, 19$ which are sorted last. All nodes left and right are grouped and sorted before the separator. Proceeding recursively produces a factorization with balanced elimination tree.

This process is called *nested dissection* [36]. Note that finding the best ordering to minimize the tree height and/or to minimize fill-in is an NP-hard problem, so

heuristics must be used. The main goal in nested dissection is to quickly find *good* separators, i.e. consisting of a small number of edges while still generating balanced components, each of which can again be divided by good separators. We note that the graphs of the meshes encountered in geometry processing naturally allow finding good separators. The reason for this is that the triangle mesh is usually *embedded*. This means, adjacency along the surface, which is important for the differential operators, corresponds to adjacency in the graph. This makes it possible to find efficient separators in the graph because graph distance is correlated with euclidean distance to some degree. Davis [18] notes that for many discretization of two-dimensional problems this leads to an overall computational complexity of $\mathcal{O}(n^{3/2})$ with $\mathcal{O}(n \log n)$ nonzeros in $\mathbf{L}$. In Section 4.4 we indeed observe this to be the case for the particular problems we consider here.

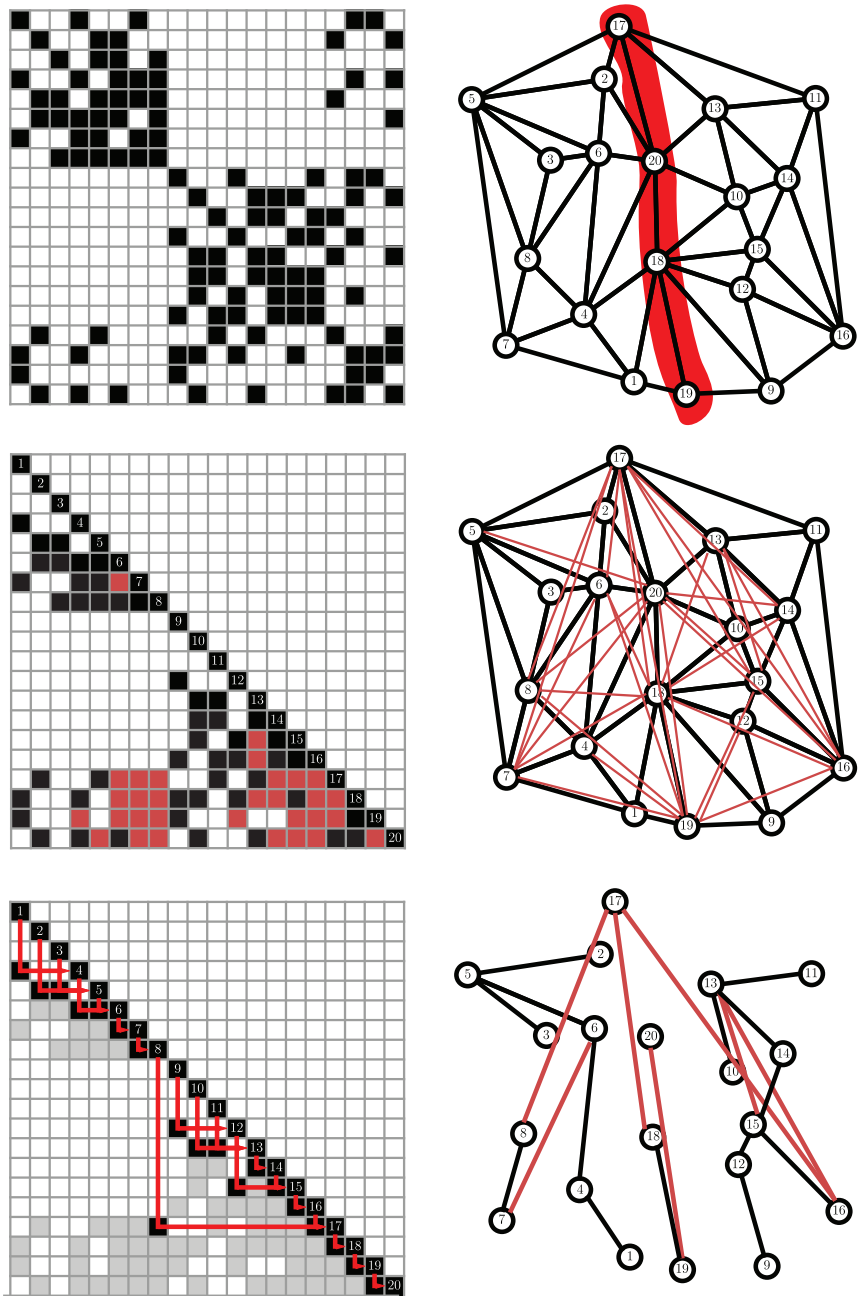


Figure 4.3: *The nonzero structure of a discrete Laplacian (upper left) based on the triangular mesh (upper right). The corresponding Cholesky factor contains elements that were zero in the Laplacian matrix (middle), however, because the elimination tree (lower right and Figure 4.4) is nearly balanced, fill in can be limited. A balanced elimination tree will also facilitate fast solving times for sparse and localized sets of vertices.*

4.3 Local solves

We have seen that it is possible to reorder the matrices commonly encountered in geometry processing such that the Cholesky factors have a small number of fill-edges (the total number of nonzero elements is on the order $\mathcal{O}(n \log n)$ if the elimination tree is fairly balanced). The elimination tree provides an efficient implementation of the forward and back substitution for sparse right-hand sides.

Consider a right-hand side $\mathbf{b}$ with a single nonzero element for index i. The forward solve involves only visiting the elements on the path from i in the elimination tree to the root, which is available as the sequence $(i, \pi(i), \pi(\pi(i)), \dots)$. Choosing $i = 5$ in the example (see Figure 4.3 and 4.4), one ends up with the nodes on the bold black path which represent the only columns that have to be considered. This is not only computationally efficient, it also means the number of nonzero elements in the intermediate solution $\mathbf{y}$ is bounded by the height of the elimination tree or, in other words, $\mathbf{y}$ is very sparse.

What happens if in addition to b_i the element b_j is nonzero? The additional elements in $\mathbf{y}$ are given by the path $(j, \pi(j), \pi(\pi(j)), \dots)$.

If the additional node lies on the path of b_i, e.g. $j = 7$ in our example, there is no extra computational work involved. For a node not on the path the additional work depends on the position in the tree. If the paths for nodes i and j have many nodes in common, e.g. for $j = 1$, the amount of additional computation is very limited.

This is a consequence of the nested dissection ordering, which is constructed such that vertices close in the triangle graph are likely to fall in the same subtree. More generally, the intermediate solution $\mathbf{y}$ is sparse for sparse right-hand sides $\mathbf{b}$ if the nonzero elements in $\mathbf{b}$ are close to each other. This property is also illustrated in Figure 4.1 which shows the subtree of nodes necessary to perform a forward solve for the gray vertices.

In other words, the particular geometry of many problems in geometry processing lends itself well to the sparse solve if nested dissection has been used for ordering.

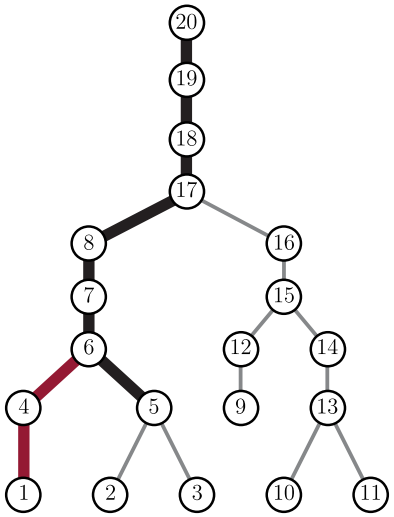


Figure 4.4: *The elimination tree of the factorization depicted in Figure 4.3. Having a nonzero at vertex 5 on the right-hand side* **b** *will result in all nodes on the bold black path to become nonzero in the intermediate vector* **y**. *An additional nonzero at vertex 0 will only add vertices on the red path to this set.*

While these connections between geometry processing and sparse linear solves might not be obvious, they have been routinely exploited by simply calling the right library functions. In the following we extend the possibilities for exploiting sparsity by also considering sparsity in the desired solution vector **x**.

Subset solve Even if the right-hand side **b** and the intermediate vector **y** are sparse, the final solution $\mathbf{x} = \mathbf{L}^{-\mathsf{T}}\mathbf{y}$ is usually dense. In heat diffusion with a single vertex as source, for example, **b** contains only a single nonzero entry but all vertices receive some nonzero quantity of heat, regardless of the timestep (cf. Section 4.6). However, it turns out that any subset of elements in the solution **x** can usually be computed *exactly*, without computing all of them.

Consider performing back substitution with an upper triangular n_v-by-n_v system $\mathbf{L}^{\mathsf{T}}\mathbf{x} = \mathbf{y}$, one row at a time. The right-hand side **y** is sparse, and derives from the forward solve described in Section 4.2. The backsolve accesses **L** column-by-column (or $\mathbf{L}^{\mathsf{T}}$ row-by-row, the i-th step accessing the i-th column of **L**):

for $i = n$ down to 1 **do**

$x_i = y_i$

for each $j > i$ for which $l_{ji} n_e o$ **do**

$x_i = x_i - l_{ji} x_j$

$x_i = x_i / l_{ii}$

At first glance, it would seem that nothing can be gained by considering only a subset of the solution vector **x**.

However, suppose we are interested in the solution only at a subset of the vertices. These vertices correspond to a particular subtree of the elimination tree (see Figure 4.1). By construction, all elements in this subtree are not influenced by the value in any other subtree. In other words, the solution for any subtree of the root node can be computed independently. This argument applies recursively for any subtree. In particular, suppose only a single entry x_i is desired. It suffices to compute all the components of **x** along the path from node i to the root, starting from the root and moving down towards node i. This gives a similar result as the forward solve: starting with the nodes of desired values of **x**, find the reach of these nodes in the elimination tree to determine the components of **x** that must be computed.

In a preprocessing symbolic analysis phase, we traverse the elimination tree from the nodes of all desired values and store the set of visited indices. This index set represents all nodes that could possibly influence the desired values. Visiting the necessary rows in order yields only a subset of the solution, including the desired nodes, and the values in this subset are correct. Note that the symbolic traversal goes up the tree to the root, starting from the desired nodes, but the numeric back substitution traverses the same nodes in opposite order.

Exploiting the elimination tree, the symbolic analysis time for determining which components of **x** are required for computing the desired subset is proportional to the size of the output set (the visited indices).

It is important to note that the sparsity of **b** and the desired subset of **x** play different roles for the efficiency of the computation and can (and should) be set indepen-

dently: the sparsity of $\mathbf{b}$ affects the efficiency of the forward substitution and the sparsity of the intermediate solution $\mathbf{y}$. The desired subset in the solution affects the subsets of rows that need to be evaluated during back substitution. It is possible, and we will exploit this feature, that the nonzeros in $\mathbf{b}$ are disjoint from the set of desired values in $\mathbf{x}$.

Solving for multiple right-hand sides In geometry processing applications geometry is often considered to be the signal. This means we commonly need to solve the same system for the different components of each coordinate, e.g. the x- and the y-vector in a parameterization task. In any case that requires solving for several right-hand sides with the same sparsity (but different numerical values) it is beneficial to perform the solves simultaneously. This has two reasons: First, cache coherence is improved for accessing the data structures that store the factorization. And, second, vectorization can be exploited for the numerical operations.

In our implementation we use templates to provide a convenient way to provide optimized solvers for different situations. Because of the limits of vectorization we limit the number of concurrent solves to 8. In this scenario we observe a 2.5-fold speed-up for 8 concurrent solves compared to 8 sequential ones.

4.4 Poisson problems on patches

One of the most basic tasks in geometry processing is the parameterization of a surface patch. The classic technique is based on elementary results in complex analysis, namely that every topological disk admits a conformal mapping into a disk. The analogue for triangle meshes is the discrete harmonic map [29]. This method minimizes the Dirichlet energy subject to Dirichlet boundary conditions, i.e. fixing boundary vertices in the plane. To compute these mappings, the linear system $\mathbf{L}_C\mathbf{x} = \mathbf{b}$ with $\mathbf{x}, \mathbf{b} \in \mathbb{R}^{n_v \times 2}$ has to be solved, where $\mathbf{L}_C$ represents the cotan Laplacian [79]. The boundary conditions require replacing rows that correspond to boundary vertices by canonical basis vectors and the right-hand side $\mathbf{b}$ is a sparse vector containing the x and y component of the desired positions, respectively.

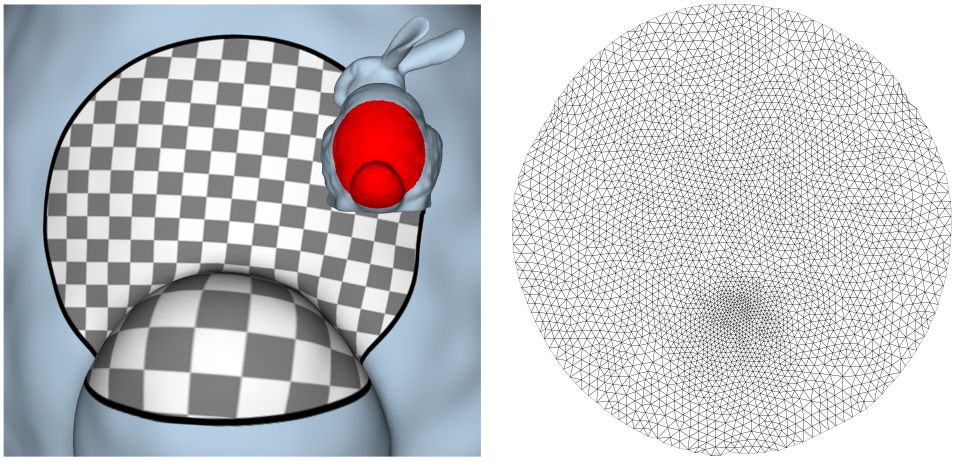
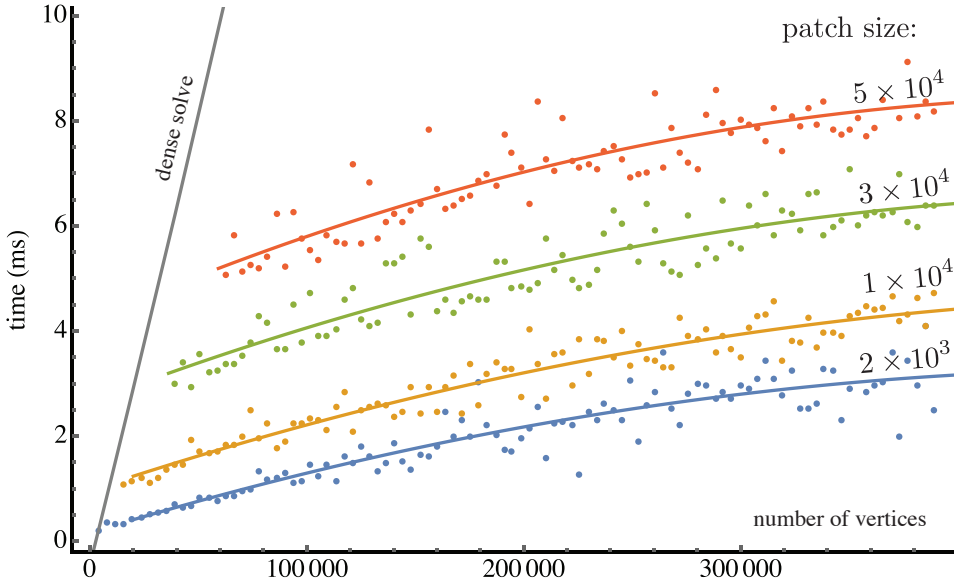


Figure 4.5: *A surface patch parameterized using localized harmonic parameterization.*

We wish to use the factorization for the full mesh $\mathcal{M}$ to efficiently compute a harmonic parameterization of a surface patch $\mathcal{M}_p \subset \mathcal{M}$. The global factorization is unaware of the patch boundary $\partial\mathcal{M}_p$, Dirichlet boundary conditions are therefore not an option. Instead we use Neumann boundary conditions [26] which amounts to prescribing values $(\mathbf{L}_C\mathbf{x})_i = b_i$ for $i \in \mathcal{B}$ where $\mathcal{B} \subset \mathcal{V}$ is the set of boundary vertices. Different choices of Neumann boundary conditions are possible. We suggest generating the Neumann boundary data by laying out the boundary vertices in the plane, for example on a circle, yielding positions $(\mathbf{x}_{\mathcal{B}_0}, \mathbf{x}_{\mathcal{B}_1}, \dots)$ where $(\mathcal{B}_i, \mathcal{B}_{i+1})$ is an edge in the mesh. The right-hand side $\mathbf{b} \in \mathbb{R}^{n_v \times 2}$ can then be defined as

$$\mathbf{b}_i = \begin{cases} \frac{1}{2}(\mathbf{x}_{\mathcal{B}_{j+1}} - \mathbf{x}_{\mathcal{B}_{j-1}})^\perp & \mathcal{B}_j = i \in \mathcal{B} \\ \begin{pmatrix} 0 & 0 \end{pmatrix} & \text{else.} \end{cases} \tag{4.6}$$

These boundary conditions do not guarantee that the boundary vertices are fixed at the positions $\mathbf{x}_{\mathcal{B}_i}$ but tend to produce disc like parameterizations. Sawhney et al. [83] describe a method to generate Neumann boundary conditions that are consistent with Dirichlet boundary conditions, however, their method needs to solve linear systems defined over the surface patch which is the operation we would



patch size:	2000	10000	30000	50000
refactorize & solve	6	42	180	386
conjugate gradients	18	227	1209	2580

Figure 4.6: *Parameterizing surface patches using sparse solves depends on the size of the full mesh. However, even for small patches on large meshes the method still significantly outperforms conjugate gradients and refactorization & solve techniques which do only depend on the patch size (see table, timings in ms).*

like to avoid in the first place.

We are now interested in the solution of $\mathbf{L}_C\mathbf{x} = \mathbf{b}$, evaluated at vertices $\mathcal{V}_p \subset \mathcal{V}$ of the surface patch. Note that this system also defines the positions of the vertices $\mathcal{V} \setminus \mathcal{V}_p$ outside of the surface patch. The benefit of our approach is that their positions do not have to be computed which has no effect on neither the solution inside the patch nor the computational complexity of the sparse subset solve applied to the interior of the patch. Figure 4.5 shows an example of a surface patch parameterized using our localization technique.

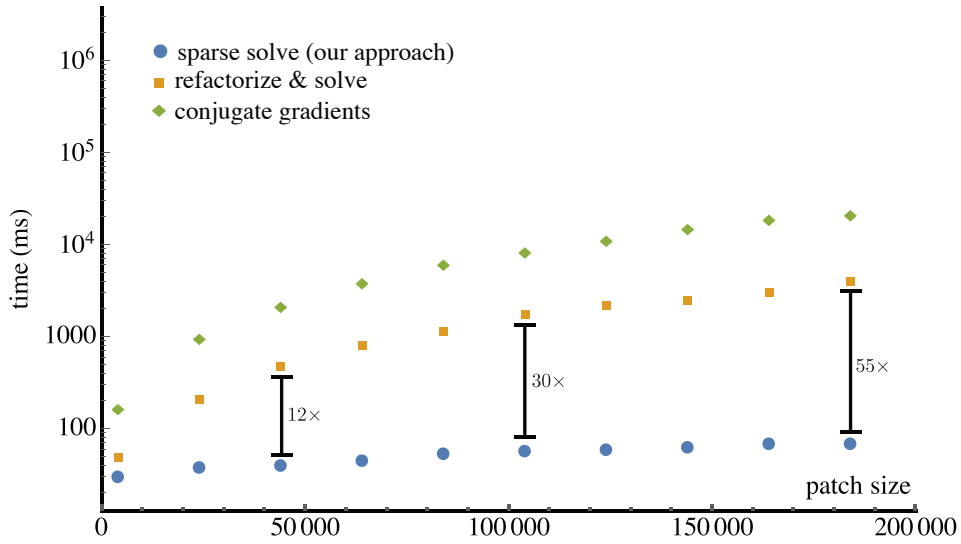


Figure 4.7: *For a mesh with 4×10^6 vertices patches of different size are parameterized. Despite the relatively large size of the full mesh compared to the surface patches, sparse solves are still orders of magnitude faster than local approaches. Because the number of columns involved in a sparse solve depend on the pattern of the right-hand side, we show the average of 100 runs for each patch size.*

Evaluation We conduct two experiments to evaluate the computational efficiency of our approach relative to pertinent parameters. We compare the performance to the two solutions suggested in the literature, namely computing the solution inside the patch using conjugate gradients, or generating a new system matrix for the patch, factorizing the matrix, and computing the solution using the sparse factorization.

We base our implementation on Eigen[43]. Sparsity is exploited whenever possible. In particular, we also exploit the sparsity of the right-hand side for solving of the locally factored system. For conjugate gradients we use diagonal preconditioning (the default in Eigen). For reference we also include timings for forward and back substitution without exploiting sparsity.

In the first experiment we consider patches with a fixed number of vertices and

vary the mesh size. The idea is that traditional approaches would be affected only by the size of the patch, but not by the size of the surrounding mesh. This is different in our approach, as the height of the elimination tree and, consequently, the local solve, depend on the size of the mesh.

We generate patches with vertex counts 2×10^3, 1×10^4, 3×10^4 and 5×10^4 using BFS on the vertex-edge graph of the triangle mesh. The table in Figure 4.6 shows the running times of the traditional solutions, which were indeed nearly constant across all experiments. The graph in Figure 4.6 shows the performance of our approach. As expected, the runtime increases with the size of the mesh, seemingly involving a logarithmic dependence. The local variation in runtime is due to the varying alignment of the patches with the nested dissection structure. In other words, if the patch happens to align nicely with the separators used in nested dissection, the sparsity in the right-hand side and the patch results in fewer nonzeros in the solution and reduced cost. If the patch intersects many such separators at a high level, the performance degrades. Importantly, the performance of our approach is significantly better than for traditional approaches, even in unfavorable constellations.

In the second experiment we take a fixed mesh with 4×10^6 vertices and vary the patch size. We take a relatively large mesh, as smaller meshes lead to smaller elimination trees and would be to the advantage of our approach. In this scenario we consider comparably small patches since larger patches are to the disadvantage of the traditional techniques. The graph in Figure 4.7 shows the result of this experiment: for all but the smallest patch sizes our approach outperforms the traditional techniques by one or more orders of magnitude.

Consistently with the literature on the numerical solution of the 2d Poisson problem, the direct solver based on sparse factorization outperforms the iterative solver based on conjugate gradients.

Reusing the factorization of the mesh Laplacian obviously requires building this factorization in a preprocess. Figure 4.8 illustrates the time required for meshes of different size. For triangle meshes reordered by nested dissection the asymp-

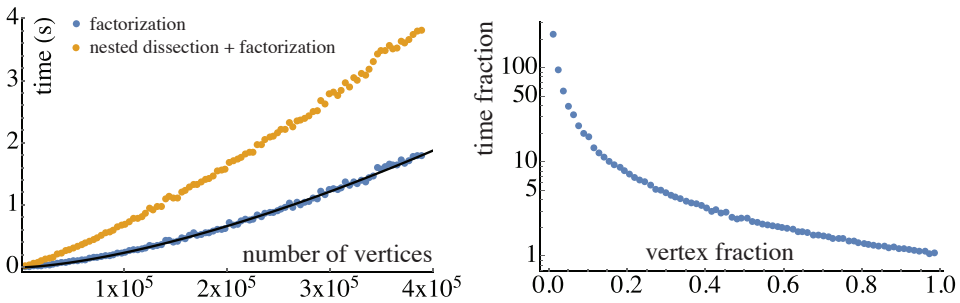


Figure 4.8: *Time required for Cholesky factorization and nested dissection (Metis) reordering (left). Time ratio of local patch factorization and factorization time for the full mesh over vertex fraction of the patch (right). Time fraction quantifies how many local factorizations can be computed until it becomes beneficial to compute and exploit a global factorization.*

totic runtime of $\mathcal{O}(n^{3/2})$ [42, 11.1.7] can be observed (see fitted curve). The second plot also includes the time taken for nested dissection reordering (using Metis [53]).

In many geometry processing applications the factorization of the full mesh is necessary anyway, and it could simply be reused for localized solves. If only few small patches are processed it may be faster to compute the factorizations for the subsets. The amortization of a full factorization depends on the size of the patch and the full mesh. Figure 4.8 (right) illustrates the ratio of local and global factorization time as a function of the relative size of the patch. For example, factorizing the Laplace matrix of a patch consisting of roughly 20% of the vertices is 10 times faster then factorizing the matrix for the whole mesh.

Compared to the times necessary for factorization, solving requires negligible time. So if indeed the overall processing time is of importance, one would have to estimate how many local solutions on patches of what size are required in order to exploit the benefit of local vs. global factorization. To stay with the previous example, if less than 10 local geometry processing problems have to be solved on patches consisting of 20% of the vertices in the mesh it is better to compute local factorizations. For more than 10 problems it is better to compute a global factorization.

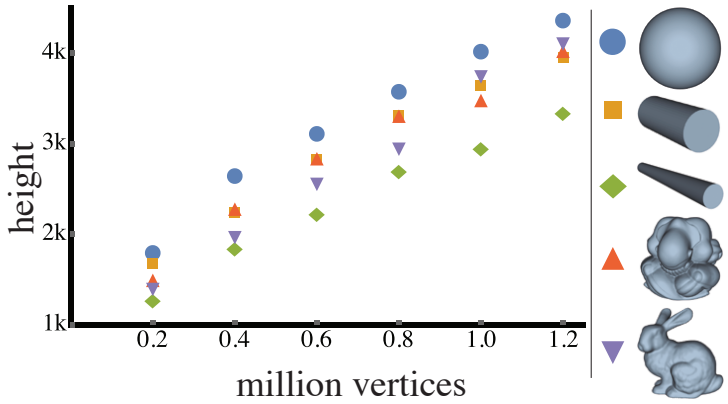


Figure 4.9: *The height of the elimination tree for different mesh resolutions, revealing the influence of mesh geometry on performance.*

Influence of geometry The efficiency of the proposed approach relies on a balanced elimination tree. While embedded models usually allow for a very efficient nested dissection reordering, results might differ with geometry. A high resolution mesh of a thin cylinder will allow for a very small separator dividing the vertices into two roughly equal sets. The mesh of a sphere, on the other hand, will have larger separators and consequently a denser Cholesky factor and higher elimination tree. The height of the elimination tree is the main factor determining the time needed for back substitution. To asses how much geometry affects performance we compare total tree height for different meshes at different resolutions in Figure 4.9. As expected the spherical mesh seems to be among the worst and the elongated cylinder among the best cases for the presented algorithm. Other meshes are ranked somewhere in between. For the evaluation in the previous paragraph we used spherical meshes at different resolution.

Non-linear parameterization More recent techniques for parameterizing patches are non-linear and typically involve iterative solutions of sparse linear systems. Some popular techniques are based on a fixed system and only modify the right-hand side throughout the iterations: As-rigid-as-possible parameteri-

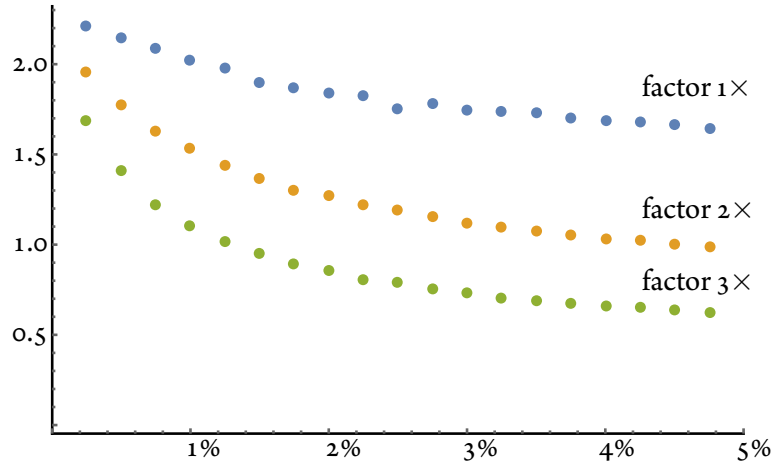


Figure 4.10: *Performance ratio of the exact and approximate sparse solve with respect to (circular) patch size (as percentage of all vertices). The radius needs to be enlarged for the approximate method, here we use a factor of one, two and three. Only for patches covering more than 5% of the surface and a factor of > 2 the exact method will be faster.*

zation [63] optimizes the per-triangle transformation to be rigid or a similarity transformation. They report up to 10 iterations for the minimization of their energy. Spectral-conformal parameterizations [74] are based on extracting the Fiedler vector from a Laplacian system. Computing this eigenvector can be done based on inverse iterations quite efficiently using our approach, as the Cholesky factors allow quick computation of the iterates. Inverse iteration is known to converge extremely rapidly, with a good starting guess in well under 5 iterations [48].

Given the performance data we have shown, even for these techniques it is beneficial to exploit a global prefactorization and then use the local sparse subset solve we provide.

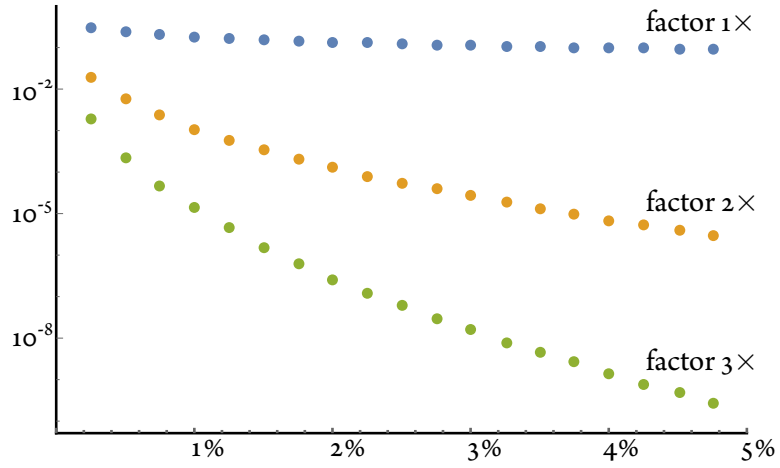


Figure 4.11: *The accuracy of the approximate solve increases with the amount of additional values computed.*

4.5 Comparison to approximate sparse solving

Chapter 3 introduces techniques to compute Voronoi diagrams and cell centroids based on heat diffusion. These techniques can be used to approximate centroidal Voronoi diagrams by means of Lloyd iterations. Naturally the solution to the involved diffusion problems is only interesting close to the source which is why they can potentially profit from the localized solution technique introduced in this Chapter. Here we compare the approximate sparse solution technique described in 3.6 with the new exact approach.

Both methods do not evaluate the result for all variables during back substitution. The approximate sparse solve uses the fact that values decay fast with distance from the source and just assumes that they are zero outside a certain neighborhood. The accuracy of the result depends on how well the assumption is actually fulfilled which depends on the diffusion time step and the size of the neighborhood. Because only values associated with vertices in a neighborhood are considered the performance of this method is largely unaffected by the size of the full mesh.

The exact sparse solution technique introduced in this chapter computes the correct result for a set of vertices and thus does not require to extend the neighbor-

hood, however, additional values also have to be computed as dictated by the elimination tree. The amount of these values depends on the size of the mesh, as analyzed in Section 3.6. This makes the exact technique more versatile since it applies to any sparse linear system for which a Cholesky factorization exists but performance will depend on the ratio of neighborhood and mesh size.

Naturally the question arises which technique is better suited for the computation of (centroidal) Voronoi diagrams. We can expect that for quite fine diagrams with small cell sizes, small time steps and large meshes the approximate technique will be very accurate and faster than the exact one, for larger cells the exact method should perform better. For a quantitative analysis we conduct the following experiment: For a triangulation of the sphere we randomly select circular patches containing a certain ratio of all vertices. We choose the sphere since this geometry can be considered to be among the worst cases for elimination tree height as detailed in Section 4.5. For each patch we perform heat diffusion using all vertices as source and set 500 times the squared average edge length as time step, as would be necessary for the computation of accurate cell centroids (cf. Section 3.5). All computations are performed using the exact sparse method and the approximate method with a patch radius increased by a factor of one, two and three. In Figure 4.11 we report the performance of the approximate method divided by the performance of the exact one. Figure 4.10 illustrates the average relative error of the approximate solve. Using only the neighborhood itself will naturally lead to a quite large error, increasing this radius brings the error down to an acceptable level. A factor of at least two is therefore recommended. The exact method will then be faster for cell sizes larger than 5% of the full mesh. For a factor of one the exact method can never be faster since the rows of $\mathbf{L}^\mathsf{T}$ considered during back substitution are a superset of the rows considered for the approximate solve. We can conclude that, except for very coarse diagrams, the approximate method will perform faster and is accurate enough. For more general problems or diffusion problems with large step size the exact method is to be preferred.

4.6 Applications

The autodiffusion function The auto-diffusion function $ADF_t(x)$ [38] measures the amount of heat that stays at a source after diffusing for time t. The more general heat kernel signature $HKS(x)$ [97] combines several values of the auto-diffusion function for varying timesteps t. These functions are used to generate descriptors and have a variety of applications, particularly in shape matching.

The standard way to compute the auto-diffusion function or heat kernel signature is based on computing a set of eigenvectors of the discrete Laplace-Beltrami operator. However, it is clear that the value of $ADF_t(\mathbf{v}_i)$ could also be computed by solving heat diffusion for $\mathbf{b} = \mathbf{e}_i$ and considering the result in x_i. If a standard approach to solving the linear equation is used, the solution would provide all values in $\mathbf{x}$. Then computing the auto-diffusion amounts to effectively inverting the diffusion operator, which is more expensive than computing a few eigenvectors. Computing the HKS would mean to compute the inverses for a set of matrices of the form $\mathbf{M} - t\mathbf{L}_C$ with varying t, which appears to not be attractive.

Using our approach, computing the value of the solution only for x_i is a best case scenario. What our approach effectively does is computing only the main diagonal of the inverse operator. This can be done at a fraction of the cost of computing the full inverse. For the HKS, we can exploit the common sparsity structure of all operators: Note that the elimination trees in the factorizations of $\mathbf{M} - t\mathbf{L}_C$ are independent of t. This means we can effectively solve for the values in one vertex with varying t in parallel, by traversing the elimination tree and computing the values based on different numerical values for the factorizations. This approach becomes particularly attractive if the auto-diffusion values are needed only in a sparse subset of the vertices, because the complexity is trivially linear in the number of desired values (whereas the computation of eigenvectors depends on the mesh size, even if the signatures are evaluated only on a subset of the vertices).

Computing the autodiffusion function using our method on a mesh with 53000 vertices (Figure 4.12 left) took 11 seconds while solving the system each time using classical forward/back substitution would take 3 minutes and 45 seconds. Com-

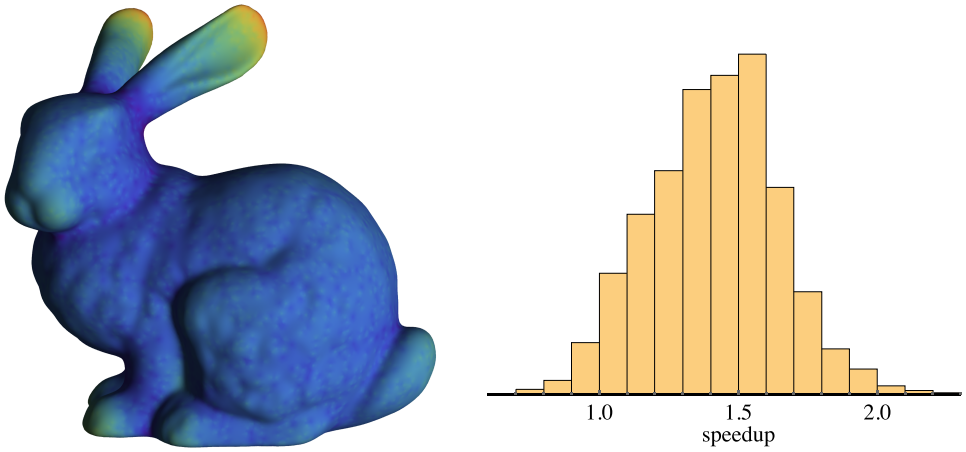


Figure 4.12: *Left: The result of computing the autodiffusion function with our method. Right: Histogram of speedup for pairwise heat exchange using our technique.*

puting 300 eigenfunctions using ARPACK to compute the autodiffusion function as described in [38] takes 39 seconds.

We wish to stress that computing the solution of the linear equation for only the source vertex is fundamentally different from restricting the computation to a submesh around the vertex i. For small time steps the results might be fairly similar, for larger t values, however, heat diffusion on a small disk like mesh around i will behave fundamentally different from the global solution. The sparse solve computes the exact results no matter what value of t is set. Moreover, the amount of computation is exactly the same for all time steps t.

Pairwise heat exchange The case of autodiffusion is optimal for the sparse solve. All that is required is to walk down one path in the elimination tree for the forward solve and move back up for the back solve. In many cases we are interested in the distance or heat exchanged by a small set of points, for example when computing embeddings [76]. If we are interested in how much heat is exchanged by two disjoint sets of points or two single points, the obvious approach is to compute the subtree of the elimination tree with respect to the union of both

sets. This can be quite efficient if both sets share most of the subtree. If this is not the case it is beneficial to use different subtrees for the forward and back solve respectively. We implemented this technique and conducted an experiment to assess its benefit. On a mesh we randomly select pairs of points. Depending on how many nodes in the paths traced to the root of the elimination are shared we expect different speedups. For 10^4 pairs on a mesh with 2×10^5 vertices we show a histogram of the achieved speedup (Figure 4.12 right). In almost all cases runtime significantly profits from using two separate subtrees. In a small number of cases, where points are quite close on the mesh, setup cost for the second tree outweighs the benefit by a small margin.

4.7 Discussion & conclusions

Geometry processing applications commonly require solving symmetric, positive-definite linear systems $\mathbf{Ax} = \mathbf{b}$. In many situations either the right-hand side $\mathbf{b}$ and/or the set of solution values of interest in $\mathbf{x}$ corresponds to a small *and* localized set of vertices. We have shown how to systematically exploit the resulting sparsity of $\mathbf{x}$ and $\mathbf{b}$ when the Cholesky factorization of $\mathbf{A}$ is provided without compromising on numerical accuracy. As a result of the nested dissection reordering strategy, the speedup over a full solve can be significant when the sparse sets correspond to spatially close vertices. Even if this is not the case, improved solving times can usually be observed. Due to the small overhead, our strategy will exhibit nearly identical running times as the full solve not exploiting sparsity in the worst case. We have demonstrated the efficiency of local solutions on representative applications in geometry processing. For these we find that using sparse subset solves on prefactored system matrices can improve the runtimes by orders of magnitude.

Localization is not always straightforward. For example, fixing a subset of arbitrary vertices (such as in many modeling scenarios) requires updating the system matrix, which may not be very efficient [23]. The next Chapter will introduce a technique to tackle this kind of problem.

5

Reusing Cholesky factorizations on submeshes

5.1 Introduction

Many interactive modeling operations require solving a linear system on a subset of a mesh. As a guiding example for this type of operation we take *mesh deformation*: the user marks a region of interest and a set of handle vertices; moving the handle vertices will then affect the vertex positions in the region of interest, while the vertices outside the region of interest stay fixed. Many approaches for this operation are based on minimizing a quadratic (or nonlinear) deformation energy

subject to position constraints of all fixed vertices [8, 94, 96]:

$$\begin{aligned} \underset{\mathbf{x}\in\mathbb{R}^{n_v\times 3}}{\operatorname{argmin}} \quad & \operatorname{Tr}\left(\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}\right) \\ \text{s.t. } & \mathbf{x}_i = \mathbf{x}_i^c \text{ for } i \in \mathcal{B} \end{aligned} \tag{5.1}$$

Note that these are actually three separate optimization problems, one for each dimension. Here the index set $\mathcal{B}$ contains the constrained vertices. The matrix $\mathbf{A} \in \mathbb{R}^{\mathbf{n_v}\times\mathbf{n_v}}$ is symmetric and positive semi-definite, in many cases the mesh Laplacian [7, 72, 79] or a low power of it. The order of vertices in $\mathbf{x}$ is arranged so that the set of unconstrained vertices $\mathcal{I}$ and the fixed vertices $\mathcal{B}$ form blocks:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_\mathcal{I} \\ \mathbf{x}_\mathcal{B} \end{pmatrix}, \qquad \mathbf{A} = \begin{pmatrix} \mathbf{A}_{\mathcal{I}\mathcal{I}} & \mathbf{A}_{\mathcal{I}\mathcal{B}} \\ \mathbf{A}_{\mathcal{B}\mathcal{I}} & \mathbf{A}_{\mathcal{B}\mathcal{B}} \end{pmatrix}. \tag{5.2}$$

The minimization can be performed by substituting $\mathbf{x}_\mathcal{B}$ with the constrained vertex positions $\mathbf{x}_\mathcal{B}^c$ and solving the resulting system

$$\begin{pmatrix} \mathbf{A}_{\mathcal{I}\mathcal{I}} & \mathbf{A}_{\mathcal{I}\mathcal{B}} \end{pmatrix} \begin{pmatrix} \mathbf{x}_\mathcal{I} \\ \mathbf{x}_\mathcal{B}^c \end{pmatrix} = 0 \quad \Rightarrow \quad \mathbf{A}_{\mathcal{I}\mathcal{I}}\mathbf{x}_\mathcal{I} = -\mathbf{A}_{\mathcal{I}\mathcal{B}}\mathbf{x}_\mathcal{B}^c. \tag{5.3}$$

Because $\mathbf{A}$ is positive definite or can be regularized the matrix $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ will inherit this property, and the system is commonly solved by computing the *Cholesky* factorization $\mathbf{A}_{\mathcal{I}\mathcal{I}} = \mathbf{L}\mathbf{L}^\mathsf{T}$ followed by performing forward and back substitution. This is particularly effective if the system is sparse. The sparsity of the resulting factor can be further improved by reordering.

While solving a sparse system given its Cholesky factorization is very fast, the necessary preordering and factorization steps are expensive computations. This can lead to problems for operations that are intended for real-time interaction. *Our central idea is to reuse the ordering and factorization of a linear system defined on the complete mesh to construct the factorization of a linear operator on a submesh.* The first idea is straightforward: The reordering that is necessary for the computation of Cholesky factors can be reused for linear systems on submeshes by just keeping the vertex indices in the same relative order. This avoids the time consuming

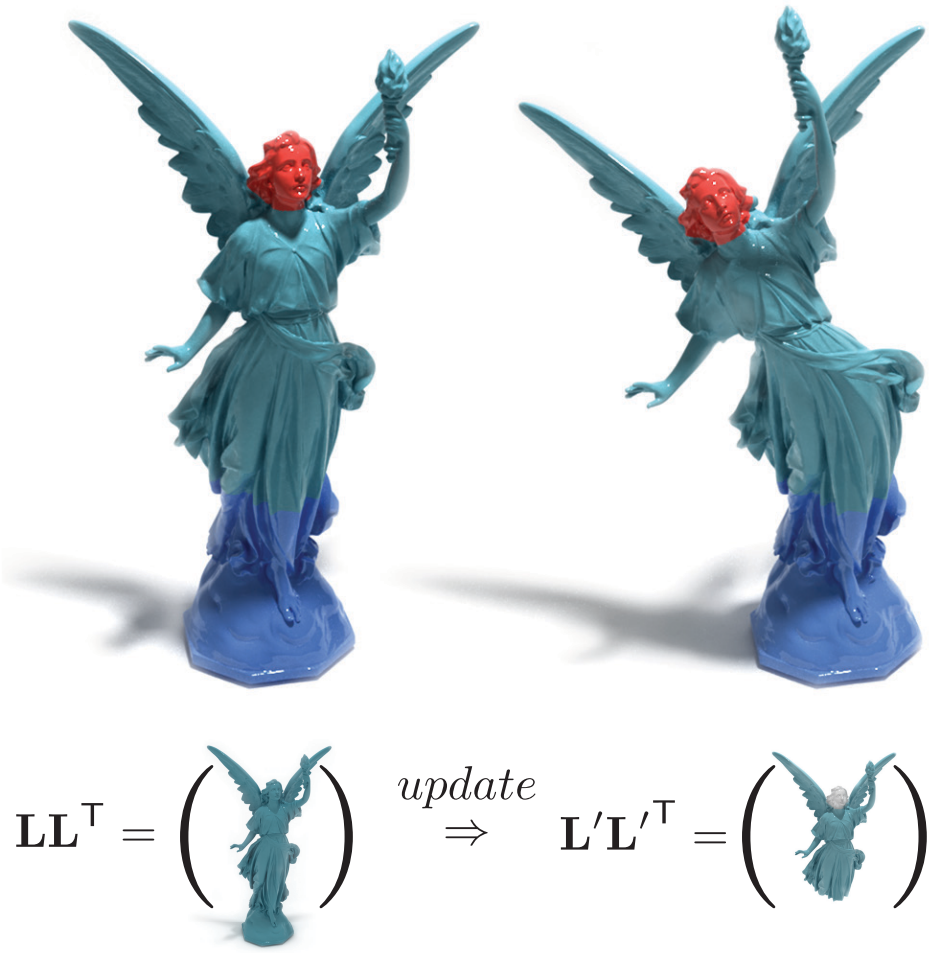


Figure 5.1: *Deforming a mesh based on selecting a region of interest (green) and a handle (red) commonly requires solving a sparse linear system. This is typically done using the Cholesky factorization. The computation time for the factorization depends on the number of vertices in the region of interest, and may prohibit interactive use for large meshes. We propose a method that exploits the factorization of the operator* $\mathbf{L}$ *on the whole mesh for computing the new factorization of* $\mathbf{L}'$ *on the submesh, instead of computing it from scratch. For the depicted mesh with one million vertices our update algorithm computes a new factorization in* 0.47 *seconds whereas the factorization of the desired system matrix for the region of interest using Cholmod takes* 3.4 *seconds.*

ordering step for the local problem.

More significantly, we observe that the factors for the local problem can be derived from the factors of the global problem by a series of rank-one updates (Section 5.4). This updates can be efficiently performed by copying data from the global factor and running a modified Cholesky factorization algorithm only on a subset of columns.

In Section 5.5 we first show that the nonzero structure of the desired local Cholesky factor can be easily deferred from the global factor. This observation is non-trivial as it is not generally true for arbitrary sparse factorizations. Based on this observation we develop an update strategy that is significantly faster than general purpose rank-one updates [21].

We compare the efficiency of our approach with complete refactorization for a range of different meshes and differently sized submeshes. Reusing the ordering already provides a speedup for solving local problems. Our update strategy results in significant further progress. Compared to the common call to a matrix library, which would perform preordering and factorization, our update strategy is significantly faster. We present detailed results in Section 5.7. In general, every algorithm that requires solving several constrained local problems on a mesh can benefit from our approach. This includes parameterization [26, 74], local mesh filtering [25] and constrained heat diffusion [17].

5.2 Related work

Hecht at al. [46] suggest a method which is similar in spirit to ours. They apply partial refactorization to Cholesky factors in a nonlinear finite element simulation of elastic objects. During the simulation the problem is repeatedly linearized, leading to sparse linear systems. The main observation is that in many cases the simulated object and therefore the linear system only changes locally. These changes in turn only affect certain parts of the Cholesky factor, which can be partially refactored. This leads to an approximation that is good enough for small time steps. The system is completely factored only sporadically. Like our approach, this method

leverages the block structure of the Cholesky factor of a matrix that has been rearranged using nested dissection reordering. The authors report a two to three fold speedup compared to full refactorization at every frame.

In a similar vein, Yeung et al. [111] consider updates to a linear FEM system when the structure of a simulated mesh is changing, e.g. during a surgical simulation. They augment the matrix with additional rows and columns to account for new constraints effectively modifying columns of the original matrix. The new system can be solved by using the original factorized matrix in conjunction with an iterative scheme. This method is very well suited for continuous changes in mesh topology. After a certain number of iterations, however, a new factorization has to be computed. Our approach on the other hand is targeted towards incorporating many constraints at once and would not be competitive if changes have to be made one at a time. In that sense this method is complementary to ours.

Modifying a given Cholesky factorization has been an active research area for many years. The central observation is that the factorization of a given matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ can be modified to become the factorization of a matrix $\mathbf{A}' \in \mathbb{R}^{n \times n}$ with much less effort than computing the new factorization as long as $\mathbf{A}'$ has been obtained from $\mathbf{A}$ by certain simple modifications. Not all modifications are allowed since $\mathbf{A}'$ has to stay symmetric and positive definite. One class of modifications that necessarily respects these properties are rank-one updates:

$$\mathbf{A}' = \mathbf{L}'\mathbf{L}'^{\mathsf{T}} = \mathbf{A} + \mathbf{v}\mathbf{v}^{\mathsf{T}} = \mathbf{L}\mathbf{L}^{\mathsf{T}} + \mathbf{v}\mathbf{v}^{\mathsf{T}}. \tag{5.4}$$

with $\mathbf{v} \in \mathbb{R}^n$. This type of modification appears quite naturally, e.g. when adding an observation in a regression model that has to be solved using least squares. Davis et al. [21] provide an efficient implementation of this method for sparse matrices. Special care has to be taken when the nonzero structure of the factor changes, which can generally happen. The method is very efficient if updates are performed one at a time. If several updates are necessary at once, i.e. $\mathbf{A} + \mathbf{V}\mathbf{V}^{\mathsf{T}}$ with $\mathbf{V} \in \mathbb{R}^{n \times k}$, the update can be performed slightly faster than consecutive rank-one updates, however, the amount of speedup is limited to about a factor of two because updates can only be efficiently processed in small batches [22]. If the update is of

very low rank (i.e. k is very small) this method might actually be faster than our approach. However, this assumption is not true for almost all use cases considered here. In Section 5.7 we compare the performance of our approach to multi-rank updates as implemented in Cholmod.

It is one of our central observations that in our particular scenario, where only very specific updates have to take place, the nonzero structure is unaffected by the updates. We show how this simplifies the operation and allows for much faster multi-rank updates.

5.3 Background: Sparse Cholesky factorization

Our approach is based on the Cholesky factorization of the linear system. We make use of several properties of sparse Cholesky decompositions, its common data structures, and the basic factorization algorithm. We already covered the basics of Cholesky factorizations in Section 4.2, here we will provide some further details of the factorization procedure itself as far as they are required for understanding our algorithm. For practical reasons we use the variant $\mathbf{A} = \mathbf{L}\mathbf{L}^\mathsf{T}$ of the decomposition in this chapter and do not explicitly represent the diagonal matrix $\mathbf{D}$ (cf. Section 4.2).

Left-looking Cholesky factorization In principle there are many ways to compute Cholesky factorizations. A popular method is the so called left-looking algorithm that computes the factor $\mathbf{L}$ column by column using the already computed columns to the left – hence the name. Assume the first $i-1$ columns of the factor have already been computed and one wants to determine the i-th column $(l_{ii}, \boldsymbol{\ell}_i)$ (see Figure 5.3). For the diagonal value l_{ii} we get

$$\mathbf{l}_1 * \mathbf{l}_1^\mathsf{T} + l_{ii}^2 = a_{ii} \tag{5.5}$$

$$\Rightarrow \quad l_{ii} = \sqrt{a_{ii} - \mathbf{l}_1 * \mathbf{l}_1^\mathsf{T}}. \tag{5.6}$$

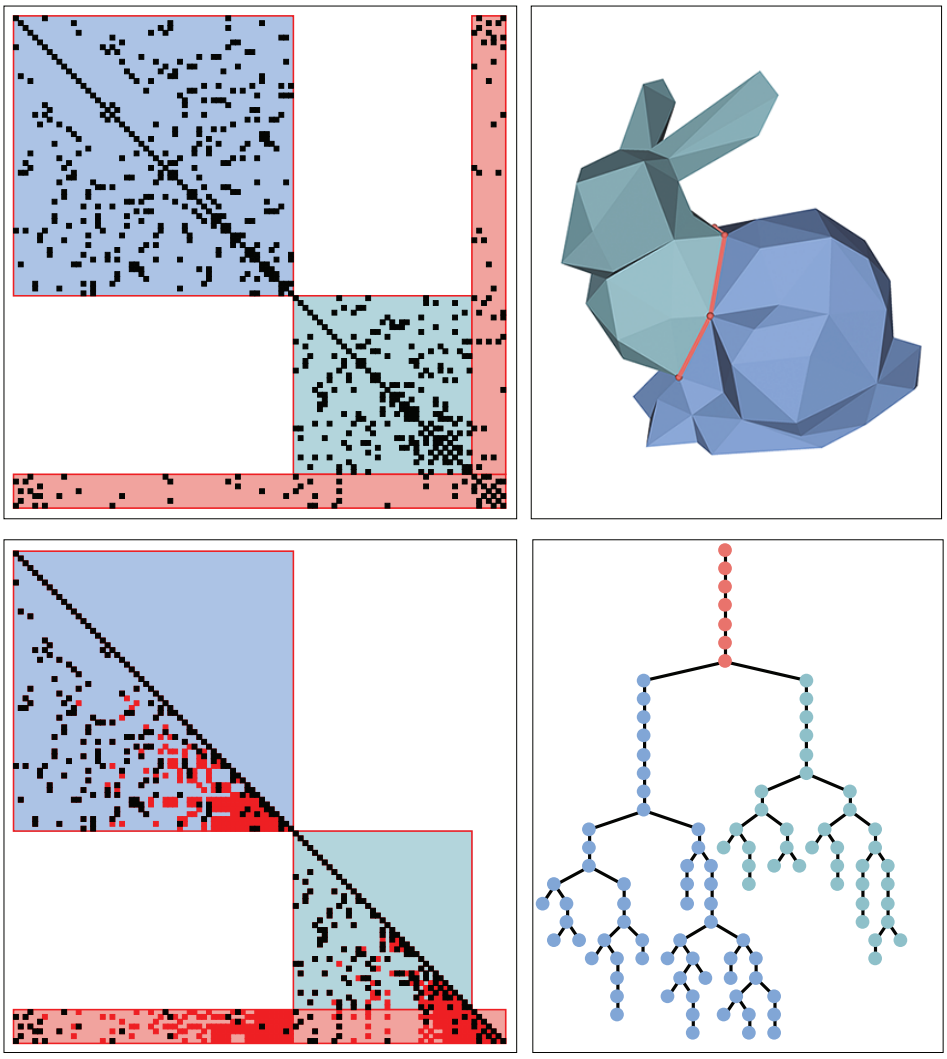


Figure 5.2: *Nested dissection reordering hierarchically divides the mesh into two balanced parts separated by a small divider (top right). The mesh Laplacian will then exhibit a block structure (top left). These blocks are preserved in the Cholesky factor (bottom left) which has additional nonzero entries only inside these blocks (red pixels). The block structure yields a fairly balanced elimination tree (bottom right).*

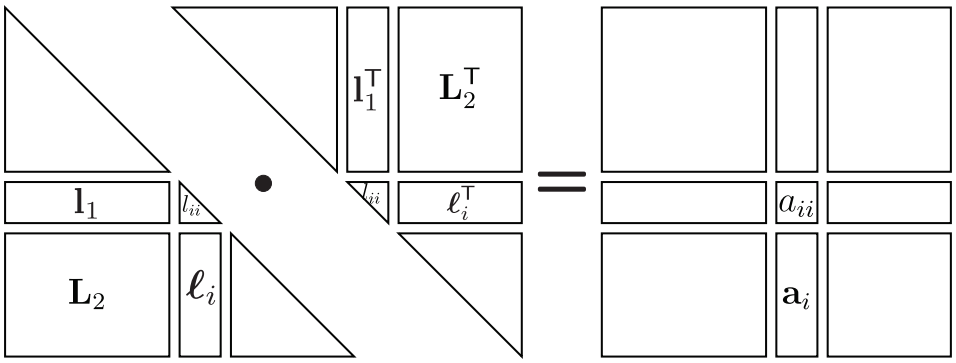


Figure 5.3: *Schematic illustration of the left-looking Cholesky algorithm. The algorithm subsequently computes the columns ℓ_i by using already computed values to its left in the factor.*

Expressing $\mathbf{a}_i$ with the matrix product on the left in Figure 5.3 provides a relationship for ℓ_i:

$$\mathbf{L}_2 * \mathbf{l}_1^\mathsf{T} + l_{ii} * \ell_i = \mathbf{a}_i \tag{5.7}$$

$$\Rightarrow \quad \ell_i = \frac{1}{l_{ii}} \left(\mathbf{a}_i - \mathbf{L}_2 * \mathbf{l}_1^\mathsf{T} \right). \tag{5.8}$$

If the matrix is positive (semi-)definite, the square root in the expression for l_{ii} is real-valued. In many computer graphics application the matrix might actually not be positive definite because of a rank deficit. Due to numeric errors the square root might produce complex values. This can be prevented by regularizing the system matrix effectively adding a small multiple of the identity matrix to the operator. Most importantly for our setting, this algorithm can be efficiently implemented for sparse matrices. Equation (5.8) also demonstrates why the block structure of $\mathbf{A}$ is preserved. If the k-th row of $\mathbf{L}$ does not contain a nonzero until column i and the same is true for $\mathbf{A}$, the value of $l_{k,i}$ will also be zero.

Sparse matrix representation Sparse matrices can be represented in several ways with advantages for different access patterns or modifications. For Cholesky factors and system matrices we use the compressed column format which maintains three arrays: one that holds all nonzero values in column major order, a sec-

ond one of the same size that stores the corresponding row indices and a smaller array that contains pointers to the start of each column in the row and value array respectively. In 5.9 we depict an example matrix (left) and its compressed column representation.

$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 3 & 0 \\ 4 & 0 & 5 \end{pmatrix} \qquad \begin{bmatrix} 0 & 2 & 3 & 5 \end{bmatrix} \quad \text{column array}$$
$$\begin{bmatrix} 0 & 2 & 1 & 0 & 2 \end{bmatrix} \quad \text{row array} \qquad (5.9)$$
$$\begin{bmatrix} 1 & 4 & 3 & 2 & 5 \end{bmatrix} \quad \text{value array}$$

Storing a sparse matrix this way it is generally very fast to visit all nonzeros per column. To traverse a matrix row is much less efficient. Alternatively the matrix can be represented in a compressed row major order by interchanging the role of rows and columns. All operations performed during the sparse Cholesky factorization, as described in [18], are carefully designed with the compressed column format in mind.

Elimination Tree In Section 4.2 the elimination tree of a Cholesky factor was introduced. Given a sparse right-hand side $\mathbf{b}$ this data structure allowed us to determine the nonzero pattern of the vector $\mathbf{y}$ in $\mathbf{L}\mathbf{y} = \mathbf{b}$. The elimination tree can also be used to determine dependencies among columns in the Cholesky factor.

Considering the left-looking Cholesky factorization algorithm one can see that the values of the i-th column of the Cholesky factor depend only on a sparse set of columns to the left because ℓ_i is sparse (see equation 5.8). Knowing what columns depend on each other is the key to our efficient factor update: Changing a set of columns in $\mathbf{A}$ only affects columns that depend on them in the Cholesky factor.

To find all columns that depend on a column i numerically one can simply traverse the elimination tree starting at node i up to its root and collect all visited nodes. A proof of this fact can be found in [62].

This makes an update quite efficient and also illustrates why changing a column usually does not affect too many others: when modifying a column in the blue

block of Figure 5.2 (lower left) all columns in the green block remain unaffected because they reside on a separate sub-tree.

Sparse Cholesky The left-looking Cholesky factorization can be efficiently implemented for compressed column matrices. The computation proceeds in two phases. The symbolic phase builds the elimination tree and uses it to analyze the nonzero pattern of the Cholesky factor. During the numeric phase the actual values are computed and stored in the prepared sparse matrix. Running the left-looking Cholesky factorization proceeds by constructing $\mathbf{L}$ column by column. The computation is dominated by the evaluation of equation (5.8) which involves the multiplication of the block $\mathbf{L}_2$ with the row vector $\boldsymbol{\ell}_1$. Due to the compressed column matrix format we do not have an efficient way to access the nonzero pattern of a row. In this setting, however, we scan the rows $\boldsymbol{\ell}_i$ of the matrix one after another which enables us to use a simple pointer per column that is incremented whenever a value in the current row is accessed. With this pointer one has access to the value of a column in that row. However, this technique is only beneficial if we know the nonzero pattern of $\boldsymbol{\ell}_i$ because every column pointer needs to be checked for every row otherwise. This nonzero pattern can be found quite efficiently by traversing the elimination tree, starting from all nonzeros in $\mathbf{a}_i$ (see [18]).

In our implementation we take a slightly different route. During the symbolic phase we not only compute the combinatorial structure of the matrix in compressed column form but also in compressed row form. This gives us easy access to each column in a row and makes row access more efficient – at the cost of storage. Moreover, this row information is needed during our partial update procedure.

Supernodal Cholesky When inspecting the elimination tree one can identify strings of nodes that have only one child. Since traversing the tree determines the nonzero structure of each column it can be shown that columns of the Cholesky factor in such a string have two properties: They are immediate neighbors and they have, apart from diagonal entries, the same nonzero pattern as illustrated in Figure

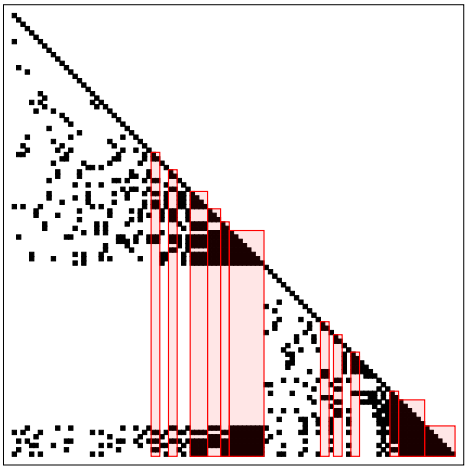


Figure 5.4: *Cholesky factors commonly include consecutive columns that share the same nonzero pattern (excluding the diagonal), here shown as red blocks. Aggregating these columns and storing the row information only once is known as the supernodal compressed column representation.*

5.4. It is therefore beneficial to aggregate these columns into so called *supernodes* to store the common row information only once and keep the values of such a group of columns as a dense matrix. The inset highlights supernodes of a Cholesky factor in red. Supernodes are not only convenient for storage, they can also be exploited during computation, leveraging fast dense matrix kernels (BLAS / LAPACK). Our implementation based on Cholmod makes extensive use of this fact. However, for clarity of exposition we describe our algorithm in terms of simple columns and hide the implementation details of the supernodal version of the algorithm. For more detail see the book by Davis [18].

5.4 Approach

Our goal is to use the factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^\mathsf{T}$ of the (global) matrix for a local problem on the same mesh. Note that $\mathbf{A}$ has been reordered to optimize the sparsity of the factor $\mathbf{L}$; changing the order of $\mathbf{A}$ would require recomputing the factorization. This means the common approach of ordering the system such that the unconstrained and constrained vertices form blocks, as described in the introduction, is

infeasible.

Nonetheless, $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ and $\mathbf{A}_{\mathcal{I}\mathcal{B}}$ can still be expressed in terms of subblocks of $\mathbf{L}$. Let a_{ij} be an arbitrary matrix entry of $\mathbf{A}$. This entry can be represented by the inner product $a_{ij} = \mathbf{l}_i^\mathsf{T}\mathbf{l}_j$ of two rows of the Cholesky factor $\mathbf{L}$. These two rows can be split into columns for $\mathcal{B}$ and $\mathcal{I}$, which can be interpreted as computing the inner product in a different order:

$$a_{ij} = \mathbf{l}_i^\mathsf{T}\mathbf{l}_j = \sum_k l_{ik}l_{jk} = \sum_{k\in\mathcal{I}} l_{ik}l_{jk} + \sum_{k\in\mathcal{B}} l_{ik}l_{jk} = \mathbf{l}_{i\mathcal{I}}^\mathsf{T}\mathbf{l}_{j\mathcal{I}} + \mathbf{l}_{i\mathcal{B}}^\mathsf{T}\mathbf{l}_{j\mathcal{B}}. \tag{5.10}$$

Putting this together for all entries a_{ij} of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ and $\mathbf{A}_{\mathcal{I}\mathcal{B}}$ yields:

$$\mathbf{A}_{\mathcal{I}\mathcal{I}} = \mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{I}\mathcal{I}}^\mathsf{T} + \mathbf{L}_{\mathcal{I}\mathcal{B}}\mathbf{L}_{\mathcal{I}\mathcal{B}}^\mathsf{T} \tag{5.11}$$

$$\mathbf{A}_{\mathcal{I}\mathcal{B}} = \mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{B}\mathcal{I}}^\mathsf{T} + \mathbf{L}_{\mathcal{I}\mathcal{B}}\mathbf{L}_{\mathcal{B}\mathcal{B}}^\mathsf{T}. \tag{5.12}$$

Note that if the system was ordered such that the indices in $\mathcal{I}$ came first, the zero structure of $\mathbf{L}$ would imply $\mathbf{L}_{\mathcal{I}\mathcal{B}} = 0$ and $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ would be sufficient to solve the local problem. In general, however, this is not the case and Equation (5.11) is the key to generate the desired factor: we are updating the Cholesky factor $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ to become the factorization of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$.

It has been observed [20] that a low-rank update will only affect a small set of columns. Instead of using the general purpose rank-one update procedure for sparse Cholesky factorizations, which is prohibitively slow when changing a larger set of columns, we identify the set of columns that will change and rerun the factorization algorithm only on these columns, while simply copying columns that remain unchanged from the matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$.

5.5 Sparse updates

Our central observation is that in order to compute the Cholesky factor of a submatrix $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ the submatrix of the factor $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ has to be modified by means of a low-rank update, affecting only a small number of columns in most cases. Moreover, the set

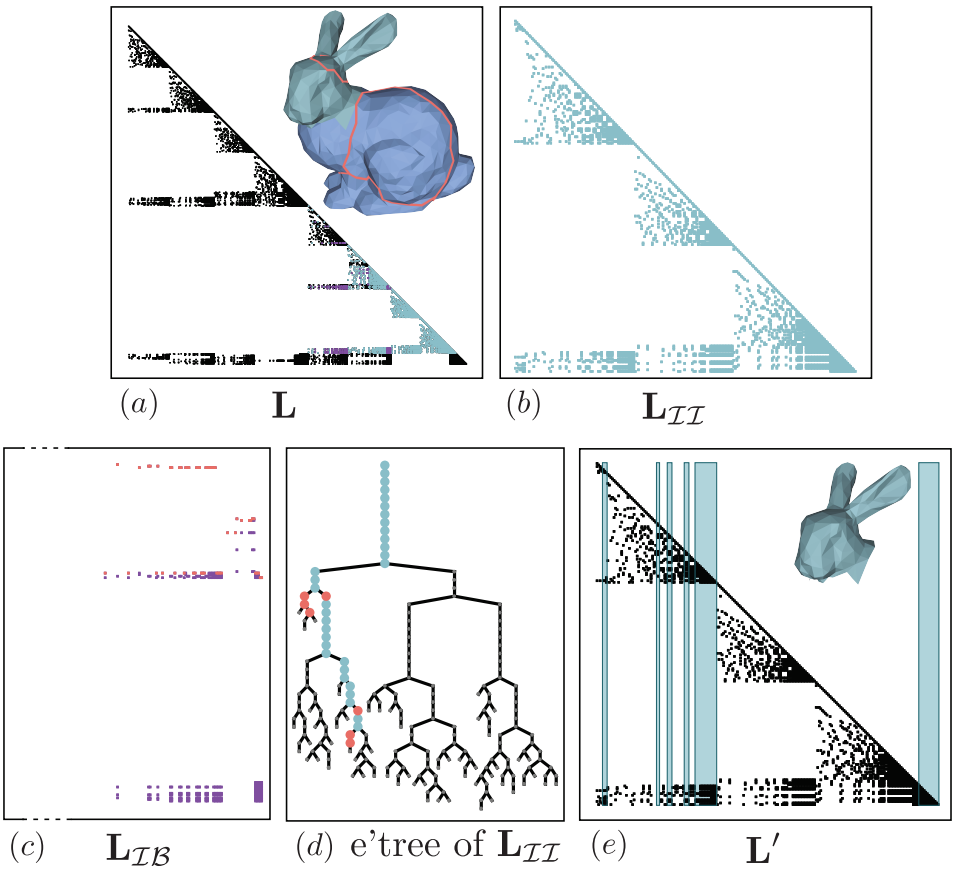


Figure 5.5: *(a) The turquoise section of the bunny model, consisting of vertices indexed by $\mathcal{I}$, is selected and a Cholesky factor $\mathbf{L}'$ for this submesh is to be computed. The Cholesky factor of the full mesh $\mathbf{L}$ induces an elimination tree, the first three levels of which are highlighted on the mesh. (b) The submatrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$, highlighted in (a), is extracted. (c) The matrix $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ can be used to determine the columns in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ that need to be updated. To do this, the index of the first entry in each column (highlighted in red) is identified and the elimination tree is traversed upwards, collecting the indices of columns that need to be updated (highlighted in (e)). To obtain the numerical values for the updated Cholesky factor $\mathbf{L}'$ the left-looking Cholesky factorization algorithm is run only on the highlighted columns while copying data from $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ for all other columns.*

of these columns can be determined very efficiently. Furthermore the update leves the sparsity structure of the sub factor unaffected, allowing us to just copy the data and refactor only those columns that need to be updated.

Sparsity structure Changing the nonzero structure of a sparse matrix can be computationally challenging since inserting values in the continuous row index and value arrays potentially involves reallocation and shifting of a lot of entries. It is therefore beneficial in our situation that the nonzero structure of the factor matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ does not change after it has been extracted from $\mathbf{L}$. Note that a nonzero in a sparse matrix always refers to a structural nonzero, that is, a value that is explicitly represented in the matrix – it may contain the numerical value 0. To see that the structure of the matrix $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ does not have to be changed when performing the update we have to show that the same structure can represent the Cholesky factorization of $\mathbf{A}_{\mathcal{I}\mathcal{I}}$. Consider an element a_{ij} of $\mathbf{A}$ with $i, j \in \mathcal{I}$. In order to represent this element the value

$$\left(\mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{I}\mathcal{I}}^{\mathsf{T}}\right)_{ij} = \mathbf{l}_{i\mathcal{I}}\mathbf{l}_{j\mathcal{I}}^{\mathsf{T}} \tag{5.13}$$

can not be structurally zero where $\mathbf{l}_{i\mathcal{I}}$ and $\mathbf{l}_{j\mathcal{I}}$ are rows in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. Assume $i \geq j$ without loss of generality. Since the nonzero structure of Cholesky factors only contain additional nonzeros as compared to the lower triangular part of the system matrix $\mathbf{A}$, we know that all nonzeros in $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ are also nonzero in $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. This means that $l_{i,j}$ and $l_{j,j}$ are both nonzero and it follows that the inner product in (5.13) does not vanish in general.

Due to this fact, $\mathbf{A}_{\mathcal{I}\mathcal{I}}$ can indeed be represented by a Cholesky factor with the same sparsity structure as $\mathbf{L}_{\mathcal{I}\mathcal{I}}$. This saves us the need to analyze the sparsity structure of $\mathbf{L}'$ based on its elimination tree. However, it is possible that the updated Cholesky factors explicitly store elements that are structurally zero as an explicit zero. In Section 5.7 we show that these values are actually quite rare.

Update Algorithm Our algorithm proceeds as depicted in Figure 5.5. Once the submesh consisting of vertices indexed by $\mathcal{I}$ is selected, the submatrix consisting

of rows and columns in $\mathcal{I}$ is extracted from $\mathbf{L}$. The update

$$\mathbf{A}_{\mathcal{I}\mathcal{I}} = \mathbf{L}'\mathbf{L}'^{\mathsf{T}} = \mathbf{L}_{\mathcal{I}\mathcal{I}}\mathbf{L}_{\mathcal{I}\mathcal{I}}^{\mathsf{T}} + \mathbf{L}_{\mathcal{I}\mathcal{B}}\mathbf{L}_{\mathcal{I}\mathcal{B}}^{\mathsf{T}} \tag{5.14}$$

can be thought of as a series of rank one updates of the form

$$\mathbf{L}'_k\mathbf{L}'^{\mathsf{T}}_k = \mathbf{L}'_{k-1}\mathbf{L}'^{\mathsf{T}}_{k-1} + \boldsymbol{\ell}_{\mathcal{I}b_k}\boldsymbol{\ell}^{\mathsf{T}}_{\mathcal{I}b_k} \tag{5.15}$$

where $k > 0$ and $\boldsymbol{\ell}_{\mathcal{I}b_k}$ is a column in $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ and $\mathbf{L}'_0 = \mathbf{L}_{\mathcal{I}\mathcal{I}}$. The columns in which $\mathbf{L}'_k$ and $\mathbf{L}'_{k-1}$ differ can be determined efficiently by finding the first off-diagonal nonzero in $\boldsymbol{\ell}_{\mathcal{I}b_k}$. The row index of this value is then used to traverse the elimination tree of $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ upwards to the root. The set of nodes that are discovered along the way represent the set of columns that will change due to the update. The columns that are modified across all rank one updates are found by traversing the tree starting from the first entries in every column of $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ (depicted red in Figure 5.5c,d). What contributes to the efficiency of the traversal is that it is stopped as soon as a node is found that has been discovered and treated previously. At this point it is guaranteed that all nodes up to the root have already been discovered while traversing from a different start node before.

The performance of the algorithm depends critically on the number of columns we have to update. Luckily the nested dissection structure ensures that in most practical cases this number is low compared to the recomputation of all columns – as would be needed for the full refactorization of the submesh.

To see this, consider the selected surface patch (turquoise region in Figure 5.5a). As the surface patch only covers a very localized part of the mesh, the matrix $\mathbf{L}_{\mathcal{I}\mathcal{B}}$ will be very sparse with most of its columns containing only zeros. This is despite its large dimensions, covering all columns of vertices not contained in $\mathcal{I}$. Moreover, considering the position of the first nonzero per column in the elimination tree of $\mathbf{L}_{\mathcal{I}\mathcal{I}}$ (red in Figure 5.5c,d), it becomes clear that they will only induce a change in a small subset of columns. Note that there are only seven distinct row indices highlighted in Figure 5.5c. The columns corresponding to discovered nodes are highlighted in Figure 5.5e. Values in not highlighted columns

are directly copied from $\mathbf{L}$ and will not be touched by the update procedure. All sub-trees that are completely contained in $\mathcal{I}$ are skipped when updating $\mathbf{L}_{\mathcal{II}}$ since $\mathbf{L}_{\mathcal{IB}}$ cannot have a row index contained in these sub-trees. Again the block structure given by the nested dissection ordering limits the amount of computation. After the columns that are affected by the update are determined, the left-looking Cholesky algorithm is run on the sub-factor, skipping all columns that remain unchanged. Our algorithm can be summarized as follows. Given the vertex indices of the submesh vertices $\mathcal{I}$

1. Compute the elimination tree of $\mathbf{L}'$.
2. Find the first nonzero in all columns of $\mathbf{L}_{\mathcal{IB}}$.
3. Traverse the elimination tree to its root starting at these indices and mark all columns corresponding to discovered nodes.
4. Initialize the Cholesky factor $\mathbf{L}'$ of $\mathbf{A}_{\mathcal{II}}$ based on its elimination tree.
5. Fill all unmarked columns of $\mathbf{L}'$ with corresponding values from $\mathbf{L}_{\mathcal{II}}$ and set all marked columns to zero.
6. Run the left-looking Cholesky factorization of $\mathbf{A}_{\mathcal{II}}$ using the initialized factor while skipping unmarked columns.

Row indices As discussed in Section 5.3, we store information about all columns in a row explicitly. As we are skipping most columns during factorization we cannot use techniques that rely on the fact that rows are accessed one after the other in order to access all values in a row. The row information we maintain has to be recomputed when extracting the sub-factor $\mathbf{L}_{\mathcal{II}}$, producing only a small overhead, however, it still doubles the amount of information necessary to store the nonzero structure of $\mathbf{L}_{\mathcal{II}}$. Since the algorithm is implemented in terms of supernodes instead of columns, the amount of structural information necessary is reduced compared to a regular sparse format. In Section 5.7 we analyze this overhead which amounts to below 15% of the total memory used to store the factorization in all our experiments.

5.6 Implementation

We implemented our algorithm from scratch in C++ inspired by Cholmod [14]. Since supernodal Cholesky factorization algorithms can leverage fast BLAS kernels we use the Intel MKL [47] with openMP support. To ensure a fair comparison, we link Cholmod with the same library for our performance evaluation. Besides the need for a BLAS/LAPACK compatible library, the code we provide is self-contained, however, if desired, it can be used in conjunction with Eigen [43]. For nested dissection reordering we employ Metis [53]. All timings have been conducted on a computer with a 3.49 GHz Intel Core i7 processor (four physical cores) and 32 GB of RAM.

5.7 Evaluation

To evaluate performance we compare our approach to Cholmod [14]. For a set of meshes of different sizes (shown in Figure 5.6), patches are generated by randomly selecting a vertex and then collecting 10%, 25% or 50% of all vertices closest to that vertex. For each such patch, the sub-factor is extracted and updated according to our algorithm.

The traditional approach consists in building the operator for the surface patch and refactoring it from scratch using Cholmod. For a fair comparison, we do not include setup operations like forming a submesh and setting up the operator matrix. Instead the submatrix $\mathbf{A}_{\mathcal{II}}$ is extracted directly from $\mathbf{A}$, which is much faster. We observe some variance in performance – our approach depends on how the patch appears in the elimination tree. When selecting a well separated feature, like in Figure 5.5, it is very likely that the sub-factor will contain large sub-trees of the elimination tree. Any sub-tree that is completely contained in the patch can be copied without modification. Selecting patches at random is therefore a slight disadvantage for our method.

Because the nested dissection ordering is spatially adaptive, it makes sense to ex-

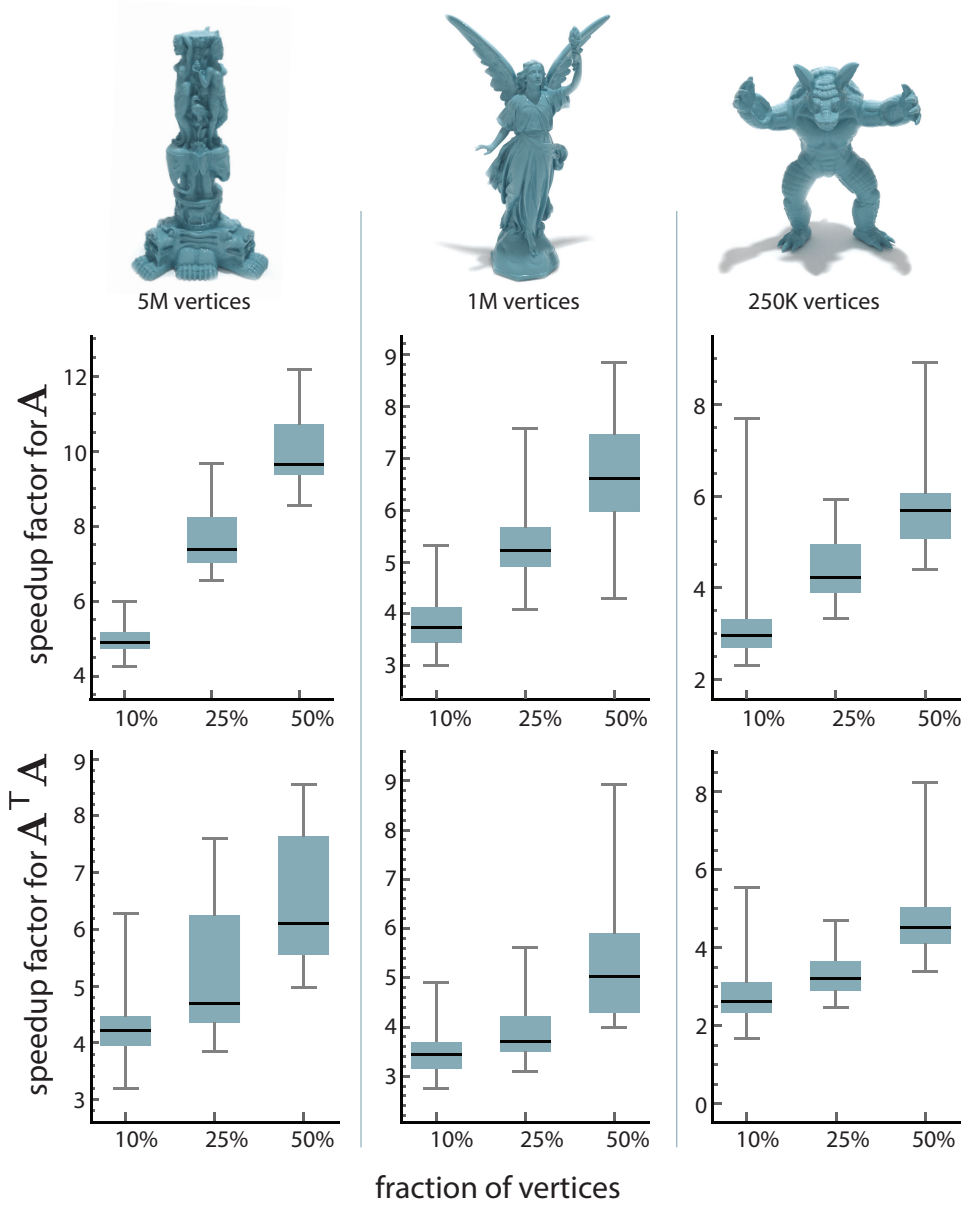


Figure 5.6: *Speedup factor of our update method compared to refactorization using Cholmod. The plots show the total range and quartiles for updates consisting of 10%, 25% and 50% of all vertices on three different meshes. Each experiment has been repeated 50 times for the mesh Laplacian* $\mathbf{A}$ *(upper row) and* $\mathbf{A}^\mathsf{T}\mathbf{A}$ *(lower row).*

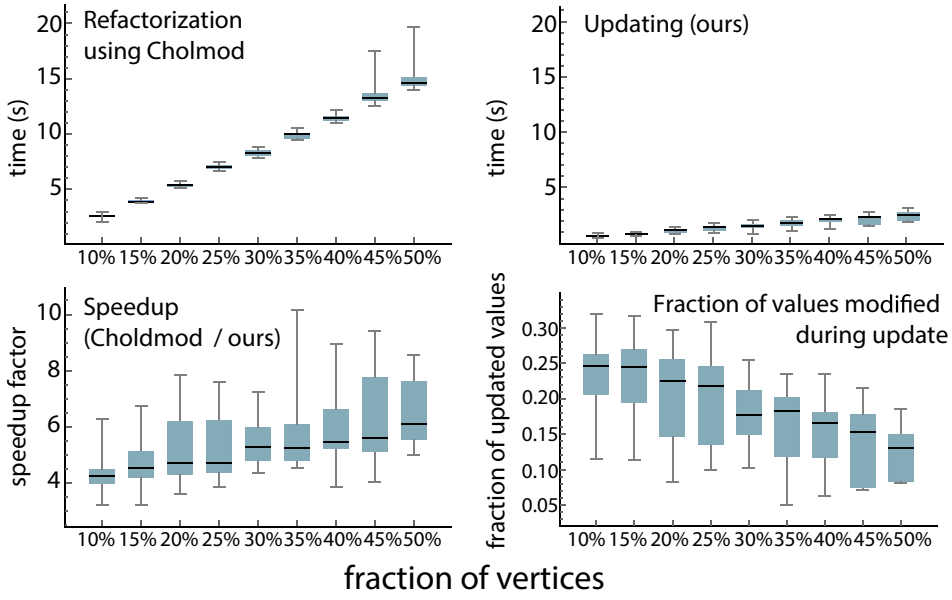


Figure 5.7: *Detailed statistics for the Cholesky factor of* $\mathbf{A}^\mathsf{T}\mathbf{A}$ *on a mesh with 5 million vertices. Absolute updating and factorization times across 50 experiments are illustrated (top row) along with the distribution of speedup factors and the fraction of values that need to be updated in* $\mathbf{L}_{\mathcal{II}}$.

tract sub-factors and submatrices in ascending order. In other words, the set of patch indices $\mathcal{I}$ is sorted. For a compact, localized patch, a restriction of the ordering to that patch will yield a good block structure which in turn guarantees a sparse factorization and fast updates. This is illustrated in Figure 5.5b. Selecting the submatrix from $\mathbf{L}$ while keeping the relative ordering of columns and rows produces a well ordered matrix $\mathbf{L}_{\mathcal{II}}$.

However, it might be possible to improve sparsity by reordering the matrix $\mathbf{A}_{\mathcal{II}}$, which is not an option for our update procedure. To test if Cholmod can profit from sorting the matrix, we ran our experiments with Cholmod twice, turning resorting on and off respectively. Across all instances Cholmod could not amortize resorting times during the factorization phase. It appears that the matrices are already sorted in a way that allows efficient computation, i.e. induce a well balanced elimination tree so the benefit of resorting is negligible compared to the necessary computation.

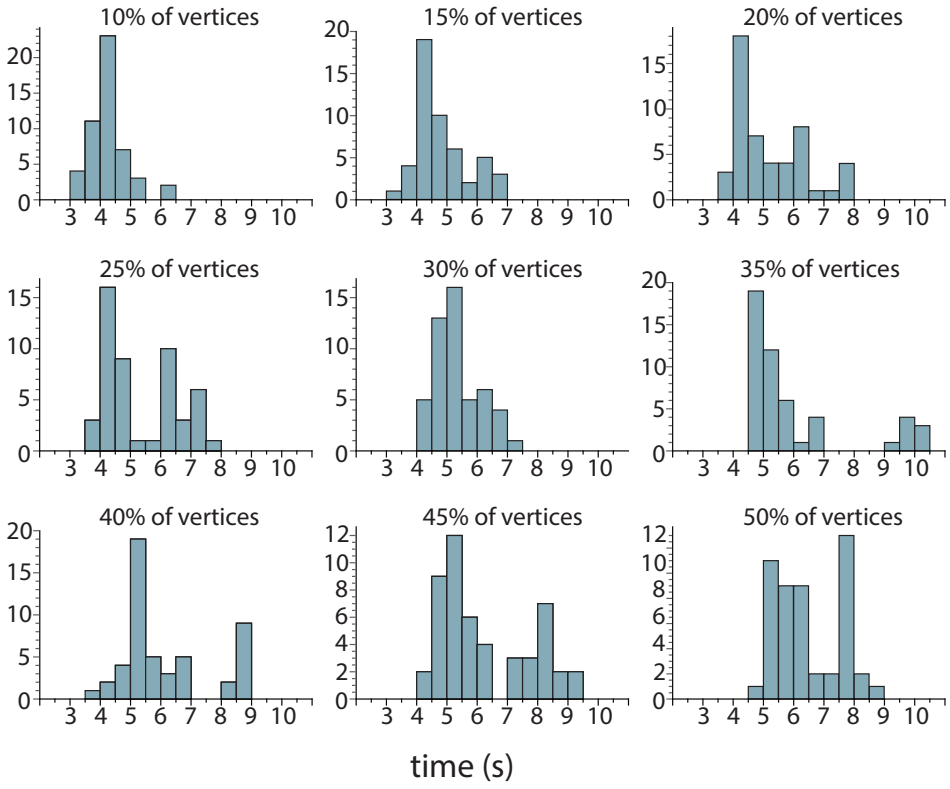


Figure 5.8: *Histograms of speedup for updates of the Cholesky factor of* $\mathbf{A}^\mathsf{T}\mathbf{A}$ *on a mesh with 5 million vertices. The diagrams show the frequency of occurrence in 50 random updates producing Cholesky factorizations for submeshes of the given size.*

Across all experiments, our method outperformed the costly refactorization step by a factor of 4 to 6 on average. Figure 5.6 highlights how total mesh size affects our algorithm. The plots show the mean (black), quartiles (blue) and the range (gray) of speedup factors across 50 experiments. We present experiments on three meshes of different sizes updating the factorization of the mesh Laplacian $\mathbf{A}$ and $\mathbf{A}^\mathsf{T}\mathbf{A}$ used in least squares systems.

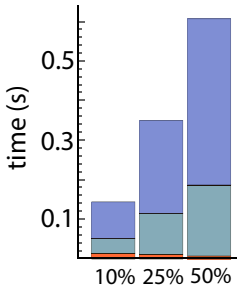
Our method performs better for large patches on large meshes as in these cases a lot of data can be reused, which would otherwise have to be recomputed. Best performance is to be expected if a majority of vertices is selected forming a locally separated patch as shown in Figure 5.1. This also explains the high variance to-

wards larger speedups whenever the randomly chosen patch happened to select a well separated region.

The performance of Cholmod is measured separately and shown in Figure 5.7. The data shows almost linear growth in time with system size. The performance of our method depends critically on the fraction of values that have to be updated in the Cholesky factor (Figure 5.7, right). We can see, again, that updating becomes more beneficial for larger patch sizes.

Figure 5.8 shows histograms of performance data for a mesh with 5 million vertices across 50 random experiments for each submesh size (Figure 5.6, left). The bimodal distribution for larger neighborhoods can be explained by patches that cover the complete upper or lower half of the mesh, and patches that cut the mesh in half, therefore having a larger interface region between vertices of the patch and the rest of the mesh.

Performance breakdown To get a better understanding of how much each step of our algorithm contributes to the overall runtime, we break down the total runtime by setup costs such as determining the columns that need to be updated (red), copying data from the global Cholesky factor (green) and running the refactorization (blue). Again, we averaged values for 50 experiments on a mesh of one million vertices, selecting patches of 10^5, 2.5×10^5 and 5×10^5 vertices respectively. Generally 30% of the time is used to copy the unchanged values to the new factor. The refactorization accounts for around 68% and the rest of the time is spend for graph traversal to find the columns that need to be updated. These numbers are consistent across mesh and submesh sizes.



Memory cost Compared to refactoring from scratch our algorithm needs to maintain the matrix structure in compressed row form as well. This amounts to some extra memory that we measure as percentage of the memory for this data

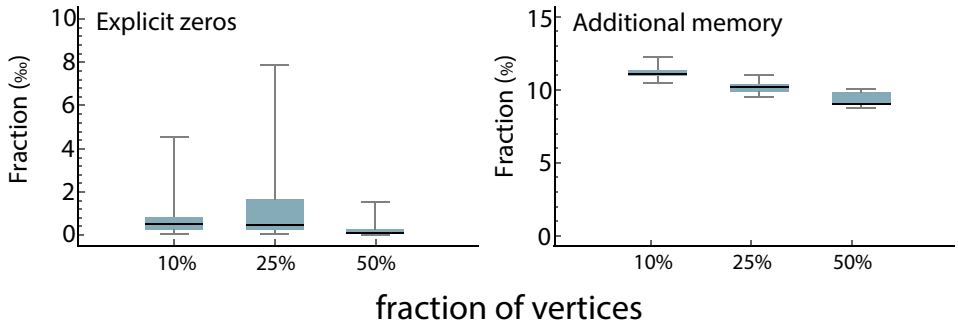


Figure 5.9: *Left: The fraction of matrix values that are explicitly stored in our updated factorization although they are actually zero. Right: The amount of extra memory used by our factorization to store additional row information.*

mem (MB)		submesh vertices		
		10%	25%	50%
vertices	250k	19 (17)	45 (40)	111 (100)
	1M	114 (113)	324 (294)	697 (637)
	5M	599 (539)	1626 (1476)	3395 (3096)

Table 5.7.1: *Memory consumptions (in MB) for factors computed using our updating algorithm and using refactorization (in brackets).*

with respect to the total memory occupied by the Cholesky factor (Figure 5.9, right). Again we used a mesh with one million vertices and different submesh sizes. The fraction of additional memory decreases with submesh size and stays below 15% in all our experiments. The results are also consistent across mesh sizes. Table 5.7.1 shows absolute memory consumption in MB for factors computed by updating a global factorization compared to factorizations computed from scratch. In Section 5.5 we have shown that the nonzero structure of the extracted sub-factor $\mathbf{L}_{\mathcal{II}}$ is compatible with the updated factor. However, there are possibly nonzeros present in $\mathbf{L}_{\mathcal{II}}$ that are zero in $\mathbf{L}'$. This might be another source for additional memory. In Figure 5.9 (left) we show that these structural nonzeros that have the numerical value zero amount for less than 0.1% of all values and therefore occupy only a negligible amount of extra memory.

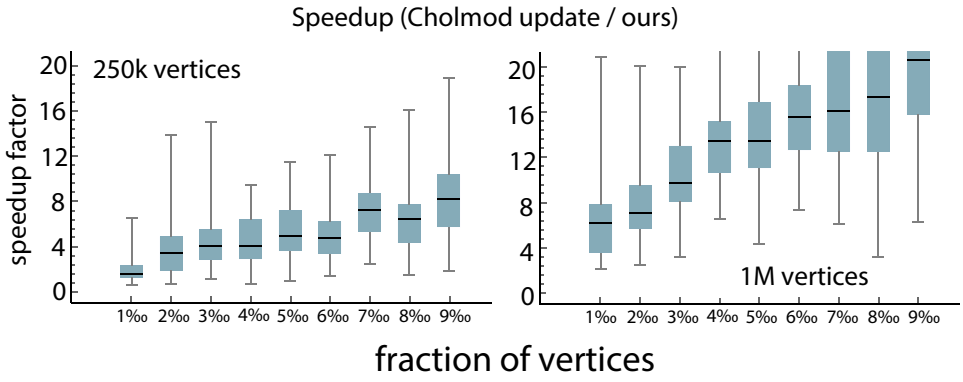


Figure 5.10: *Performance comparison of our algorithm and the update procedure implemented in Cholmod. Even for very small meshes and submeshes our method outperforms Cholmod significantly.*

Comparison to low-rank updates Cholmod provides a method to update Cholesky factors when the original matrix has been updated by a low rank modification of the form $\mathbf{A} + \mathbf{V}\mathbf{V}^\mathsf{T}$ with $\mathbf{V} \in \mathbb{R}^{n \times k}$ [22]. This method is quite efficient when the update has low rank (i.e. k is very small). We could use this method to perform the update in Equation (5.14), which is the central step in our algorithm. Even for very small submeshes with less than 0.1% of the vertices our algorithm outperforms Cholmod's update procedure (Figure 5.10). The speedup of our method compared to Cholmod becomes more extreme for larger meshes and we conclude that our algorithm is to be preferred for almost all practical usage scenarios where submeshes of reasonable size are considered. However, when updating only a few rows of a least squares system to constrain single vertices, which might for example be necessary when constructing Least Squares Meshes [95], Cholmod's algorithm will be faster.

5.8 Discussion & conclusion

We proposed a method to quickly build a sparse Cholesky factorization for a linear operator defined on a localized submesh using a factorization of an operator

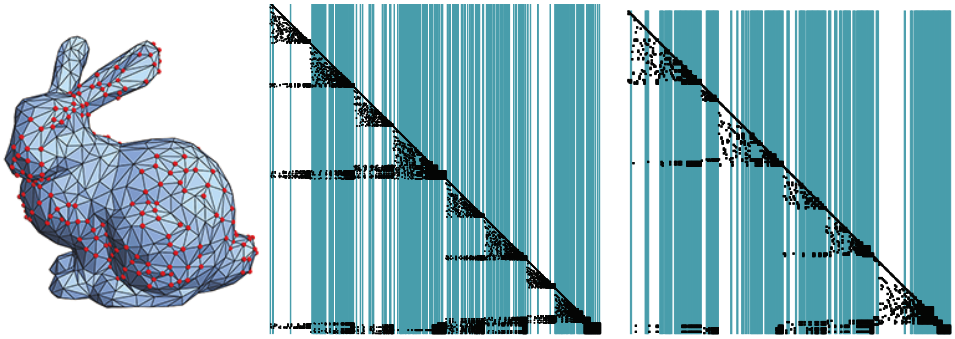


Figure 5.11: *Non local patches (left) will induce sub-factors* $\mathbf{L}_{\mathcal{II}}$ *collecting data from many different sub-trees of the elimination tree of* $\mathbf{L}$ *(center, columns in* $\mathcal{I}$ *colored) and consequently require costly updates of* $\mathbf{L}_{\mathcal{II}}$ *(right, columns requiring an update are highlighted).*

defined on the full mesh. This operation is more efficient than constructing a new factorization for that submesh from scratch. Depending on the shape and size of the submesh we observe speedup factors of 4 to 6 on average, however, this factor gets higher for submeshes covering a significant number of vertices on large meshes. These are the situations where performance is most critical and applications will benefit from our algorithm the most. Moreover, our approach was never slower than a complete refactorization in our experiments because it just performs a partial refactorization, reducing to a full refactorization in the worst case, which is very unlikely for sensible regions of interest. An example of an update that is not very efficient is depicted in Figure 5.11. Selected vertices (in red) are not localized and the columns and rows that are selected by $\mathbf{L}_{\mathcal{II}}$ are scattered all over $\mathbf{L}$. This also means that many sub-trees of the elimination tree are covered and a large number of columns have to be updated as depicted in the right image. However even in this extreme example not all columns of the factor need to be updated and we can slightly outperform a complete refactorization.

Compared to other methods for fast factorization updates, like [111], our method is not approximate. We construct the unique Cholesky factor of the constrained matrix, moreover, since we are just reusing information that has been already computed and compute updated columns from scratch, our algorithm is numerically

as stable as a complete refactorization. This also enables cascading updates: After computing the Cholesky factor with respect to a submesh by our update procedure it can be used to compute a Cholesky factor on a new submesh contained in the first one. This is a reasonable scenario when interactively editing a mesh.

As the solution of linear systems is a fundamental building block in geometry processing and simulation we expect a variety of methods to benefit from our algorithm. We provide ready to use C++ code that only depends on a BLAS/LAPACK implementation to facilitate further applications. The code can be conveniently used with Eigen [43] sparse matrices.

6

Efficient computation of smoothed exponential maps

6.1 Introduction

Many interactive geometry processing applications require local parameterizations around a point of interest, e.g. for local texture mapping (decals) or editing [85, 100]. Desirable properties for these maps are:

- their computation should be as fast as possible, enabling operations at interactive rates for moderately sized regions, and
- the isometric distortion of the parameterization should be small around the point of interest and then possibly degrade with distance.

Let **p** be the desired origin of the local parameterization. Fast local parameterization is commonly based on the idea of tracing geodesics emanating from **p**. Each point can then be parameterized based on its geodesic distance to **p** and the angle of this geodesic at **p** relative to a fixed reference direction in the tangent frame. In the plane, using straight lines as paths, this approach leads to polar coordinates. The concept analogous to straight lines on a surface are geodesic paths leading to geodesic polar coordinates. The shortest path between two points on a surface is a geodesic. In contrast to the planar setting, there might be more than one shortest path. On the sphere, for example, there are infinitely many shortest path connecting a point to its antipode. Local parameterizations based on geodesics therefore only make sense for regions where there is a unique shortest geodesic for every point connecting it to the central point **p**. In this case the polar coordinates induced by the geodesics are refereed to as *log map* or, arguably more common, by its inverse, the *exponential map* (see Figure 6.3).

While tracing geodesics may be natural and the concept of exponential maps may be well established there are fundamental and practical problems: distinct geodesic curves emanating from the origin **p** may intersect, leading to singularities in the parameterization. Also, computing exact geodesic curves on discrete surfaces is expensive. This suggests that one rather wants to approximate the exponential map, focusing on fast computation rather than generating exact geodesic distances and angles [86].

For discrete surfaces Crane et al. [17] show how distances can be computed by solving sparse linear systems. In particular they observe that gradients of the discrete heat kernel at a specific vertex closely approximate gradients of the distance field. Our technical contribution is an extension of this method that also determines the required angular information at the center vertex (see Figure 6.1). Since our method is based on evaluating angles of gradients formed with a fixed reference direction it inherits the robustness of the original method. As a result, we can efficiently compute both, distances as well as angles, by solving sparse linear systems.

The heat kernel is the result of simulating heat diffusion for a short time starting

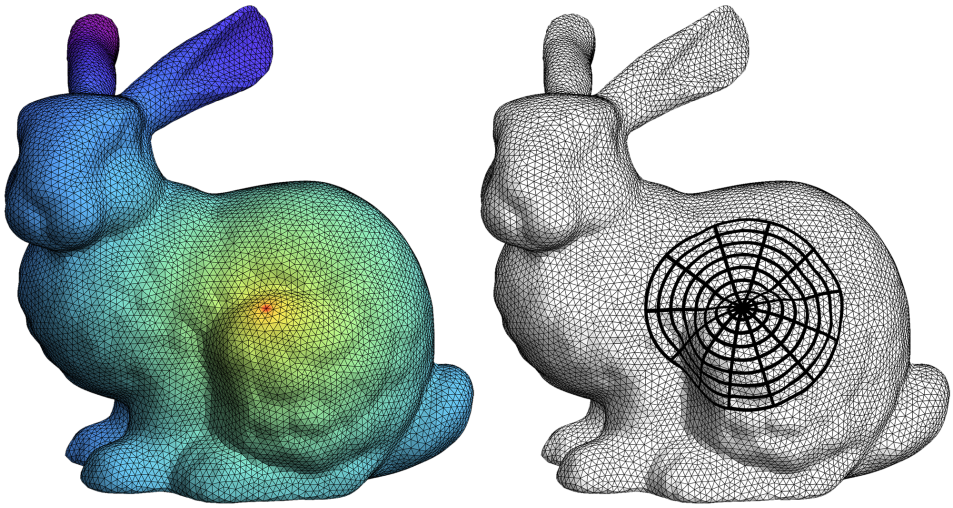


Figure 6.1: *Computing the heat kernel centered at a small set of vertices (left) can be used to efficiently compute exponential maps (right).*

with a singular heat source. For larger times, however, one gets curves that resemble geodesics but are smoother. In other words, the diffusion time provides a parameter that allows to control the smoothness of the map. This property is useful in presence of surface features that would cause the exact exponential map to be discontinuous or exhibit large distortion.

Defining the linear systems on the local surface patch and then solving them using either direct or iterative methods would be significantly slower than forward traversals from the center vertex. However, the localization technique presented in Chapter 4 demonstrates how to reuse the factorization of global systems for local diffusion problems, resulting in a significant speedup for the type of problem we consider. Our central observation is that the computation of distances *and angles* is in fact faster than methods based on Dijkstra's algorithm. We believe this is a counter-intuitive and very useful result. It can be explained by the fact that the factorization of the global system is effectively a precomputation step, which is exploited in the solution of a particular local problem.

We demonstrate how to exploit the symmetry of the diffusion problem to obtain angular information for all vertices in a specific region using only a few local solves

of the prefactored system.

This allows us to compute the angles at almost negligible extra cost compared to the distances along the paths. We demonstrate that our method is not only overall faster than existing techniques but also that the smoothness resulting from using diffusion for the definition of paths avoids fold-over and distortions occurring in exact exponential maps or Dijsktra-based approximations.

6.2 Related work

Our work aims at the fast generation of local parameterizations, suitable for use in interactive applications. We focus our description of related work on the subset of local parameterization techniques that are capable of generating parameterizations at interactive rates.

Schmidt et al. [86] (DEM) approximate exponential maps at a vertex $\mathbf{v}_i$ by augmenting Dijkstra's algorithm. For each vertex $\mathbf{v}_j$ an arbitrary tangent frame F_j orthogonal to the vertex normal is determined. Next, local exponential maps for each vertex are computed by rotating each adjacent edge of that vertex into the tangent frame. To express vectors in a tangent frame F_j with respect to F_i a rotation aligning both frames is computed. To find the final exponential map, the mesh graph is traversed using Dijkstra's algorithm starting at vertex $\mathbf{v}_i$. When traversing an edge its local exponential map representation is rotated to the reference frame and added up. The robustness of this approach can be improved by averaging local exponential maps from all already visited neighbors in each Dijkstra step [84]. We use this robust version of the algorithm in our comparisons.

The algorithm merely computes an approximation of the exponential map because it only uses linear isotropic mappings between coordinate systems and approximates the real geodesic by an edge path. Nevertheless, the mappings produced resemble exact exponential maps nicely in many practical cases.

Malvær et al. [71] (DGPC) also propagate information using a Dijkstra-like algorithm. They proceed by inferring information for a vertex in a triangle when distance information to both other vertices is known. To this end a virtual source

point in the triangle plane is computed that has the correct distance to both vertices. The authors report increased accuracy as compared to Schmidt et al. [86], assuming exact exponential maps as the reference.

Exact exponential maps can be defined using exact polyhedral geodesics, i.e. the shortest paths connecting two vertices on the mesh geometry. There are several algorithms computing this piecewise linear path. While earlier approaches where rather slow [13, 73] more sophisticated implementations could improve performance significantly [99, 106, 108]. Some of these algorithms can sacrifice accuracy for performance. Ying et al. [112] present an algorithm with tunable accuracy that also propagates angular information thus producing all information needed for a exponential map. The authors report that they can outperform Crane et al. [17] when the accuracy is relatively low but fail to match their performance when comparable accuracy is requested. Precomputation times are several orders of magnitude higher than for the method of Crane et al. in that case. Since our algorithm has essentially the same precomputation step as Crane et al. [17], this comparison carries over to our method. Algorithms that compute polyhedral geodesics, either exact or approximate, have the advantage that they are usually not affected by the triangulation quality in contrast to heat based methods that can exhibit artifacts for anisotropic, irregular and non-Delaunay meshes. This comes usually at a higher performance cost. Moreover, we have found exponential maps based on polyhedral algorithms to lack smoothness because angles are strongly influenced by the shape of triangles adjacent to the source vertex. Furthermore, even the smallest cavity on a smooth mesh will lead to artifacts due to non-unique geodesics. We argue that smoothed versions of the exponential map, as computed by our heat based method, are preferable over exact geodesic exponential maps in many scenarios.

Sun et al. [98] compute local stroke parameterizations enabling texture manipulation on a mesh. They employ exact polyhedral geodesics based on on Xin et al.'s work [108, 109]. Each point close to a stroke is parameterized by its distance to the stroke and the arclength at the closest point.

Exponential maps on triangular meshes have recently be used to define local charts for convolutional neural networks [69]. In these applications very fast approxima-

tions of the exponential map for small neighborhoods are required. This emphasis on performance, however, can introduce significant inaccuracies that seem to be tolerable for this specific task.

In concurrent work Sharp et al. [88] introduce a heat based method to perform parallel transport of vector valued data on meshes. The basic idea of their method is similar to ours, yet they approach the development from the perspective of generalizing the heat method. This naturally leads to stronger theoretical foundation. They also provide a wide range of different instances for specific problems, including how to quickly compute exponential maps.

In contrast, we focus on the specific problem and provide a detailed analysis of the preservation of length and angles as well as the behavior of singularities with respect to the time step, and contrast this behavior with a range of possibly competing methods.

6.3 Preliminaries

We introduce and recall some notation and quantities associated with the triangulated surface. In particular, we need the area vector per face and the mean curvature normal derived from the discrete Laplace-Beltrami operator.

For discretizing the heat equation we need a discrete Laplace-Beltrami operator. We suggest to use the cotan operator, which we denote as $\mathbf{L}_C \in \mathbb{R}^{n\times n}$ and a lumped mass matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$ 3.5. Using this operator we define normal directions at the vertices as the direction of the area gradient (or, equivalently, the mean curvature)

$$\hat{\mathbf{n}}_i = \left(\mathbf{L}_C(\mathbf{v}_0, \mathbf{v}_1, \ldots)^\mathsf{T}\right)_i . \tag{6.1}$$

This allows us to assign a unit normal direction as

$$\mathbf{n}_i = \begin{cases} \hat{\mathbf{n}}_i / \|\hat{\mathbf{n}}_i\| & \hat{\mathbf{n}}_i \neq 0 \\ \sum_{ijk} \mathbf{a}_{ijk} / \| \sum_{ijk} \mathbf{a}_{ijk}\| & \text{else,} \end{cases} \tag{6.2}$$

where we derive the vertex normal from the area vectors if the mean curvature is

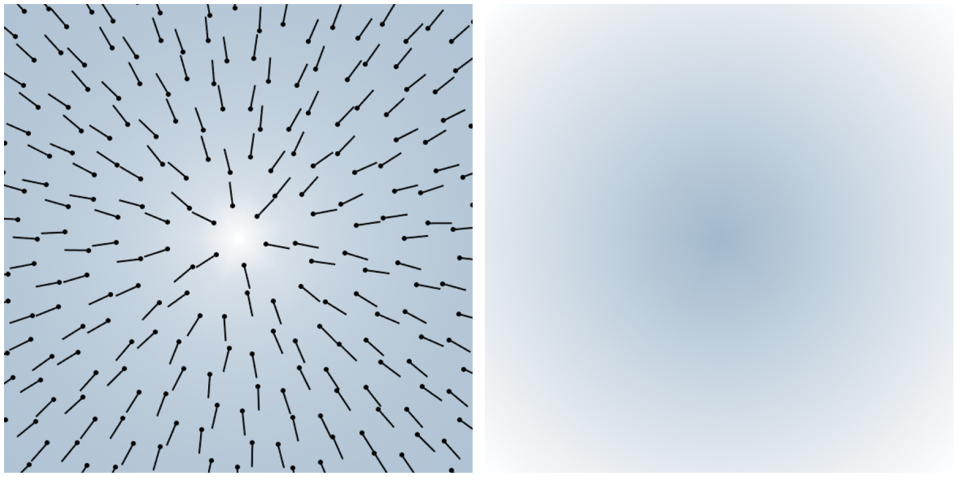


Figure 6.2: *Computing geodesic distance using heat diffusion. First the heat kernel is evaluated at a vertex and the piecewise linear gradient is computed (left). Integrating the normalized gradients yields geodesic distances (right).*

zero.

6.4 Background

6.4.1 Distances from diffusion

Crane et al. [17] compute geodesic distances by employing a result by Varadhan [103] stating that geodesic distance can be computed as a point wise transformation of the heat kernel. The heat kernel is the fundamental solution of the heat equation and describes how much heat is distributed from a point $\mathbf{p}$ to a point $\mathbf{q}$ on the surface in time t.

The heat kernel for a fixed point $\mathbf{p}$ and time t can be approximated on a surface mesh by solving a sparse linear system. For this the heat equation is discretized using the discrete Laplace-Beltrami operator $\mathbf{L}_C$ [17]. Solving the linear system

$$(\mathbf{M} - t\mathbf{L}_C)\mathbf{h}_i = \mathbf{A}\mathbf{h}_i = \mathbf{e}_i \tag{6.3}$$

where $\mathbf{e}_i$ is the $i-$th unit vector, representing a singular heat source at vertex i, yields values of the heat kernel for the fixed source point $\mathbf{v}_i$ and time step t, see Figure 6.2 (left). Instead of using these values directly the authors observe that accuracy can be significantly improved by exploiting the fact that the gradient of the heat kernel is parallel to the gradient of the distance function. The gradient of the piecewise linear function defined by $\mathbf{h}_i$ can be computed for triangle $k = (k_1 k_2 k_3)$ as

$$\mathbf{g}_i^k = \frac{1}{2\|\mathbf{a}_k\|} \sum_{s=1}^{3} (\mathbf{h}_i)_{k_s} (\mathbf{n}_k \times (\mathbf{v}_{k_{s+2}} - \mathbf{v}_{k_{s+1}})) \tag{6.4}$$

where indices are interpreted modulo 3. Because the gradient of the distance function has unit length everywhere, the correct and desired gradient field is obtained by normalizing the gradient of the heat values on each triangle

$$\hat{\mathbf{g}}_i^k = \mathbf{g}_i^k / \|\mathbf{g}_i^k\|. \tag{6.5}$$

Integrating this gradient field by solving the Poisson equation

$$\mathbf{L_C d}_i = \mathbf{D}(\hat{\mathbf{g}}_i^0, \hat{\mathbf{g}}_i^1, \ldots)^\mathsf{T}, \tag{6.6}$$

where $\mathbf{D}$ is the discrete divergence operator, results in the distance values $\mathbf{d}_i$.

6.4.2 Localized linear solves

The cost for the computation of geodesic distances using heat diffusion is dominated by the solution of two linear systems. Since both of them are sparse, symmetric and positive definite (when regularized) the systems can be efficiently solved by constructing the sparse Cholesky factorization.

Both linear systems can be prefactored and reused to efficiently solve for geodesic distances between an arbitrary vertex and all others in a small local patch.

Since exponential maps are in many cases only desired in a local neighborhood of a central vertex, both localized solution techniques, the approximate one intro-

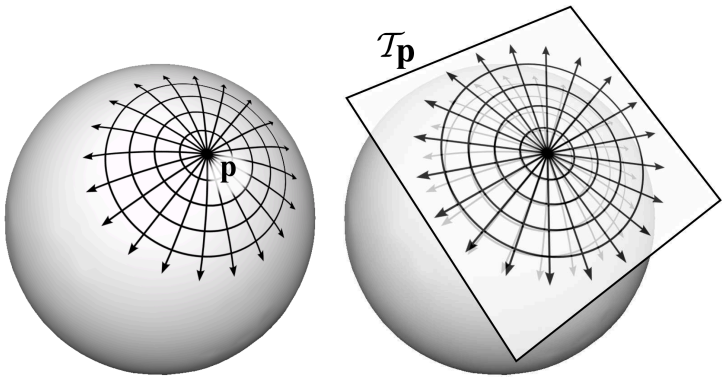


Figure 6.3: *Illustration of the exponential map centered at* $\mathbf{p}$. *Image courtesy of Ryan Schmidt.*

duced in Section 3.6 and the exact one described in Chapter 4, can be employed to yield an efficient algorithm. Because heat values decay exponentially but their gradients are renormalized (Equation (6.5)) the algorithm is sensitive to accuracy issues. For that reason we adopt the exact sparse solution technique, the approximate method might also be applicable in certain scenarios.

6.5 Algorithm

Given a center vertex $\mathbf{v}_i$ we wish to compute the log map for a set of vertices $\{\mathbf{v}_j\}_{j\in\mathcal{I}}$. Each vertex in this set is mapped to the geodesic distance to $\mathbf{v}_i$ and the the angle of the tangent vector defined by that geodesic at $\mathbf{v}_i$ with a fixed reference direction. We can use the methods laid out in the previous section to compute distances based on solving prefactored sparse linear systems. The remaining problem is to compute the angular part of the log map. Our first observation is similar to Crane et al. [17]: the tangents of geodesics are parallel to the gradients of the solution of heat diffusion for short times t.

An essential property of our solution is symmetry: the heat diffused in time t from vertex i to vertex j is identical to the heat diffused from j to i in the same time. This property is exactly captured in the discrete setting – at least if the discrete Laplace operator matrix $\mathbf{L}$ and mass matrix $\mathbf{M}$ are symmetric. Then the heat diffused from

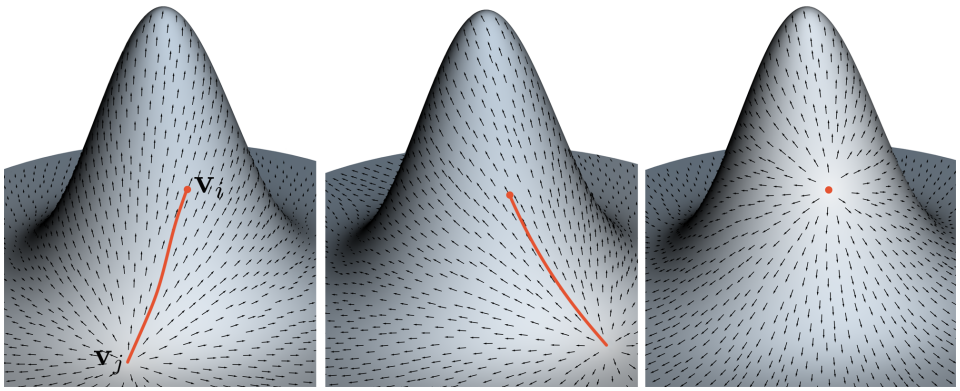


Figure 6.4: *Left: Simulating heat diffusion starting at vertex $\mathbf{v}_j$ we obtain tangent vectors to geodesics passing through $\mathbf{v}_j$. Evaluating these vectors at $\mathbf{v}_i$ defines the angular component of the log map centered at $\mathbf{v}_i$. Right: We show that in order to evaluate tangent vectors at $\mathbf{v}_i$ for all vertices in a region it is enough to solve only a few diffusion problems close to the center of the map.*

i to j is $(\mathbf{h}_i)_j$ where $\mathbf{h}$ is the solution to Equation 6.3. This is just the ij entry of $(\mathbf{M} - t\mathbf{L_C})^{-1} = \mathbf{A}^{-1}$. Since the matrix $\mathbf{A}$ is symmetric, its inverse is also symmetric and we have $(\mathbf{h}_i)_j = (\mathbf{h}_j)_i$.

Computing the angle for one vertex To compute the tangent to the geodesic connecting vertices i and j we diffuse heat starting at vertex j and evaluate the gradient in vertex i. To this end we solve Equation 6.3 with the right-hand side $\mathbf{e}_j$. Based on the heat values $\mathbf{h}_j$ we can compute the heat gradient per triangle using Equation (6.4). To define the gradient of $\mathbf{h}_j$ at vertex i we use gradients $\mathbf{g}_j^k$ in the triangles adjacent to it. We represent the discrete tangent space at vertex i by the vector space orthogonal to the (normalized) mean curvature normal $\mathbf{n}_i$ (6.2). Projecting the vertex star of i along $\mathbf{n}_i$ results in a conformal flattening, meaning the relative angles incident at vertex i are being approximately preserved. Each gradient $\mathbf{g}_j^k$ is projected onto this discrete tangent space, as illustrated in Figure 6.5, leading to

$$\bar{\mathbf{g}}_j^k = \mathbf{g}_j^k - \mathbf{n}_i \mathbf{n}_i^\mathsf{T} \mathbf{g}_j^k. \tag{6.7}$$

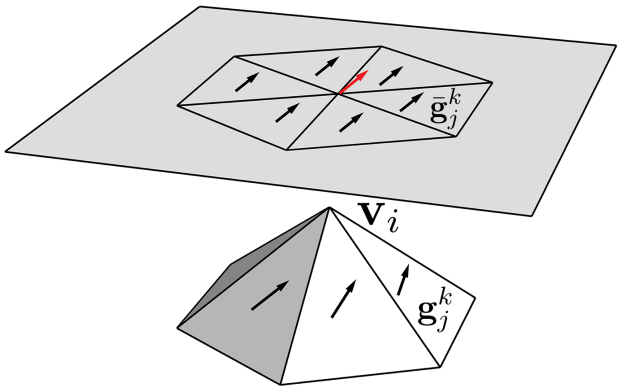


Figure 6.5: *Evaluating the gradient of the heat kernel centered at $\mathbf{v}_j$ in all triangles adjacent to $\mathbf{v}_i$ and averaging in the tangent plane gives the angular component of the map.*

The final direction is then computed by taking the area weighted average of the projected gradients:

$$\mathbf{g}_j^i = \sum_k A_k \bar{\mathbf{g}}_j^k. \tag{6.8}$$

The angle of this vector to a fixed reference direction in the discrete tangent space represents the angular component of the log map and forms, together with the distance information, geodesic polar coordinates for vertex $\mathbf{v}_j$ with respect to the center $\mathbf{v}_i$. Figure 6.4 (left) illustrates the gradient vectors for many vertices, however, we are only interested in the vector at the center vertex of the log map. It would also be possible, and arguably more elegant, to compute the gradient using an intrinsic parameterization, i.e. renormalizing the angles at the central vertex to 2π. However, we found the naive approach to consistently yield better results. Any other method for computing the gradient at a vertex could also be used.

Computing directions for all vertices in the region Suppose we want to evaluate the log map with respect to $\mathbf{v}_i$ for all vertices in a specific region. Using the technique described in the previous paragraph would require solving the heat diffusion problem

$$(\mathbf{M} - t\mathbf{L}_C)\mathbf{h}_j = \mathbf{e}_j \tag{6.9}$$

for each vertex $\mathbf{v}_j$ in that region. To compute the gradient we are actually only interested in a few elements of $\mathbf{h}_j$, namely those associated with the vertex $\mathbf{v}_i$ and its immediate neighbors $\mathcal{N}(i)$. While solving for these few values is very efficient using the localization technique described before, doing so for a large number of vertices still leads to an overall inefficient approach.

Yet, we observe that symmetry allows us to reverse the roles of source and destination. This means, instead of evaluating $(\mathbf{h}_j)_k$ for every j and $k \in \mathcal{N}(i) \cup \{i\}$ we can evaluate $(\mathbf{h}_k)_j$, giving us the same values. This way we only need to perform $d = |\mathcal{N}(i) \cup \{i\}|$ forward and back substitutions. We can easily compute these solutions in parallel exploiting vectorization which allows us to traverse the sparse matrix data structures only once during forward and back substitution. This means that we can compute d solutions in time sublinear in d in practice. Moreover, we need the values $\mathbf{h}_i$ anyway to compute geodesic distances.

Efficient computation Even though we are interested in the exponential map only in a small neighborhood around the source vertex the prefactored linear systems give values for all vertices of the mesh. Therefore performance depends on the size of the full mesh which is undesirable in most applications. A possible remedy would be to extract a local neighborhood of the mesh and solve the linear systems locally. This requires factoring two linear systems for each instance of the problem. Moreover, the result will differ since the heat kernel using the full operator considers heat diffusing over the complete mesh. Using a sufficiently large neighborhood, however, would mitigate this effect close to the surface vertex, see 3.6 for a discussion of this approach.

Using the approach described im Chapter 4 offers a way to sidestep the repeated factorization of the local problems by reusing a global factorization. This technique computes the *exact* same numerical values as traditional forward and back substitutions. The runtime of this approach depends only very mildly on the size of the full mesh.

6.6 Implementation

The implementation of the proposed approach is straightforward given an implementation of Cholesky factorization and local back substitution. Input for the algorithm is the index of the source vertex i and a set of vertex indices $\mathcal{I}$ representing the neighborhood for which the exponential map shall be computed. In a preprocess Cholesky factorizations of the heat operator $\mathbf{A}$ and the mesh Laplacian $\mathbf{L}_C$ have been computed for the full mesh. The algorithm proceeds as follows:

1. Locally solve the heat equation (6.3) for i and all vertices connected to i using the precomputed Cholesky factor resulting in vectors $\mathbf{h}_k$.
2. Compute geodesic distances using $\mathbf{h}_i$ following Crane et al. [17], integration of the gradient field is again computed by locally solving against a prefactored system.
3. For each vertex $j \in \mathcal{I}$ compute the gradient of the values $(\mathbf{h}_k)_j$ in all triangles connected to i, project them onto the tangent plane at i, build the area weighted average and compute the angle between the gradient and a fixed reference direction in the tangent plane.
4. Optionally compute coordinates by converting the polar coordinates to Cartesian ones.

6.7 Evaluation

We evaluated our algorithm on a set of meshes with different characteristic features as well as standard geometries. It is well known that the heat method can produce distorted results if the mesh is not Delaunay [66]. We therefore restrict our analysis to Delaunay meshes where possible. For meshes that do not possess this property a Laplacian based on the intrinsic Delaunay triangulation should be used [32, 66].

We compare our algorithm to the Dijkstra based approaches DEM and DGPC. For DEM we use our carefully optimized custom implementation including improvements regarding robustness introduced in [84]. We have empirically found

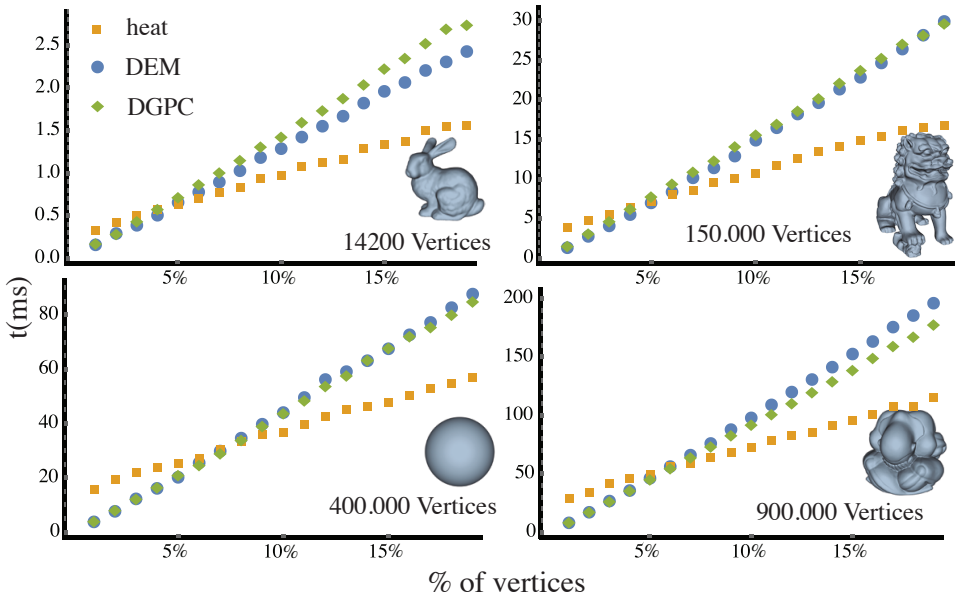


Figure 6.6: *Performance for exponential maps using our method (heat), discrete exponential maps (DEM) [86] and discrete geodesic polar coordinates (DGPC) [71]. We compare timings for fixed meshes and varying patch size (measured in fraction of total vertices). Each measurement is averaged across 50 random patches.*

that a Dijkstra implementation based on Fibonacci heaps for dynamically updating the priority queue provides the best performance. For DGPC we use publicly available code provided by the authors.

6.7.1 Performance

To evaluate performance we conducted two basic experiments. First, we vary the patch size for different meshes and compute exponential maps using our method, DEM and DGPC (see Figure 6.6). Even though our algorithm exploits the sparse local evaluation method there is still a dependency of runtime on the size of the full mesh which is not the case for the alternative approaches. Across meshes of different sizes we find that our method outperforms its competitors when the patch covers at least 5% of the mesh's vertices.

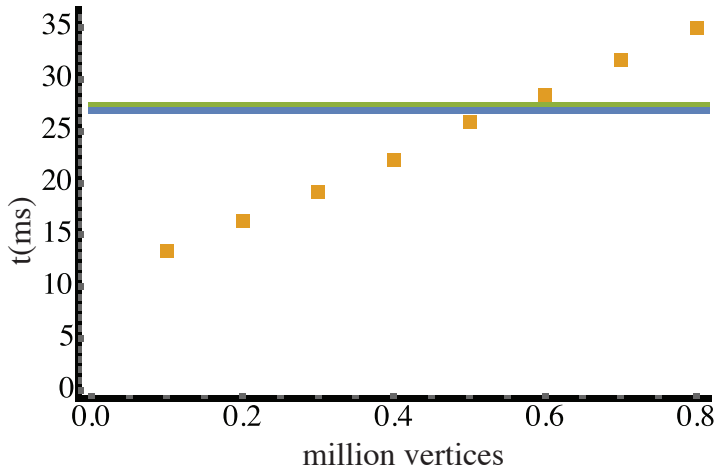


Figure 6.7: *Keeping the patch size fixed at 25000 vertices we vary the resolution of the full mesh. Left we see the performance of the heat method (yellow) which can beat DEM (blue) and DGPC (green) if the patch size is below 5% of the full mesh. Consequently we see our method outperform its competitors for meshes with fewer than* 5×10^5 *vertices.*

In a second experiment (Figure 6.7) we keep the shape of the mesh and the patch size fixed and vary the tessellation density. The performance of DEM and DGPC is unaffected because their runtime only depends on a local traversal that is not influenced by the size of the full mesh. As seen in the first experiment our algorithm is only more efficient than the other two if the size of the exponential map is more than 5% of the full mesh in terms of vertex count. We see this result confirmed in the second experiment where we choose a patch size of 2.5×10^4 vertices and consequently see better performance only for tessellations with fewer than 5×10^5 vertices.

That our approach is faster than simple traversal algorithms can be explained by the fact that the factorization of the system matrices is an effective precomputation step. This precomputation step may also be considered the downside of our method. In order for the factorization to be reasonably sparse and allow for fast local back solves, the matrices have to be reordered using nested dissection (see Section 4.2).

6.7.2 Quality

The main goal of our algorithm is to generate smooth exponential maps even close to the cut locus or areas of high curvature where competing methods produce artifacts. However, we also evaluate accuracy compared to analytic results, even though this is not our main goal and might even contradict smoothness. We can show that our method still produces quite accurate results for different standard geometries.

Smoothness We compare our algorithm to the approaches by Schmidt et al. [86] (DEM) and Malvær et al. [71] (DGPC) in terms of smoothness of the exponential map. Figure 6.8 shows some rather challenging cases of regions of high curvature that might also include parts of the cut locus (e.g. second row, close to the eyes). In all cases our algorithm produces smooth injective maps whereas other methods exhibit fold over and discontinuities in the map. These problems are usually more pronounced for DEM. For regions of low curvature all results are very close (last row). Our algorithm also performs well on very large meshes (Figure 6.9, 3.5M vertices) and does not only produce smooth maps but also outperforms competing methods. In this example we also compare to discrete conformal maps [26] which might introduce severe area distortion and is therefore not considered in the following experiments.

Exponential maps computed using our method are smooth even far away from the central vertex (see Figure 6.9). However, they might exhibit distortions at elongated features, like the exact exponential map would. Unlike competing approaches our technique does not produce noise for these regions.

Our method can not guarantee injectivity of the map. Figure 6.13 (left) shows an extreme case. Even though our algorithm does not produce an injective map it is still smooth in contrast to other methods. Note that even exact polyhedral geodesics as implemented by Surazhsky et al. [99] will introduce artifacts at the cut locus because the exponential map is not defined at these points. By tweaking the time step parameter t in equation (6.3) we can also compute injective maps

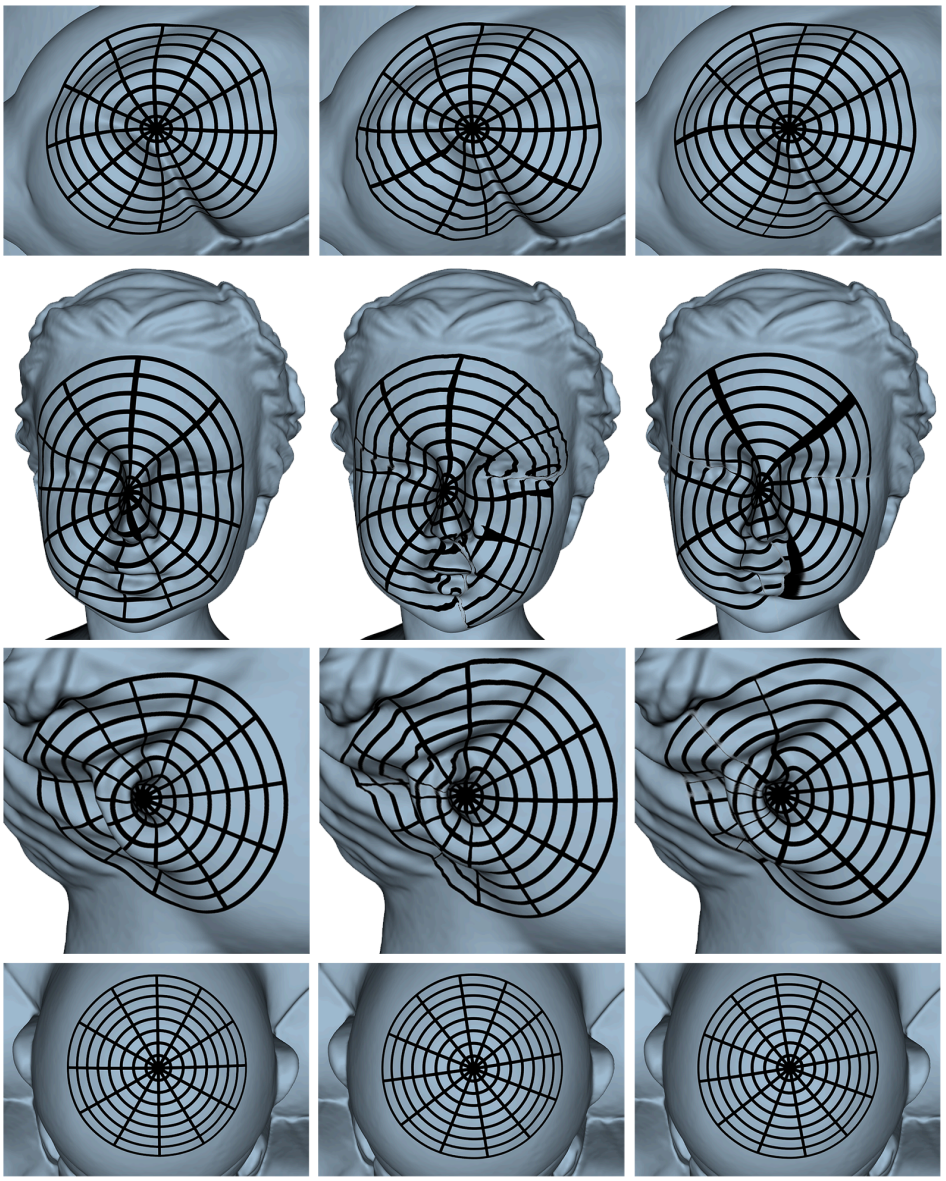


Figure 6.8: *Quality comparisons for our method (left), DEM (center) and DGPC (right). Our method yields smoother results for regions of high curvature. For smooth regions all methods perform similar.*

for these extreme cases. A parameter of 10^2h^2 produces an injective map that is considerable smoothed and therefore not as close to the ground truth exponential map. Choosing smaller values for t yields exponential maps that are more accurate

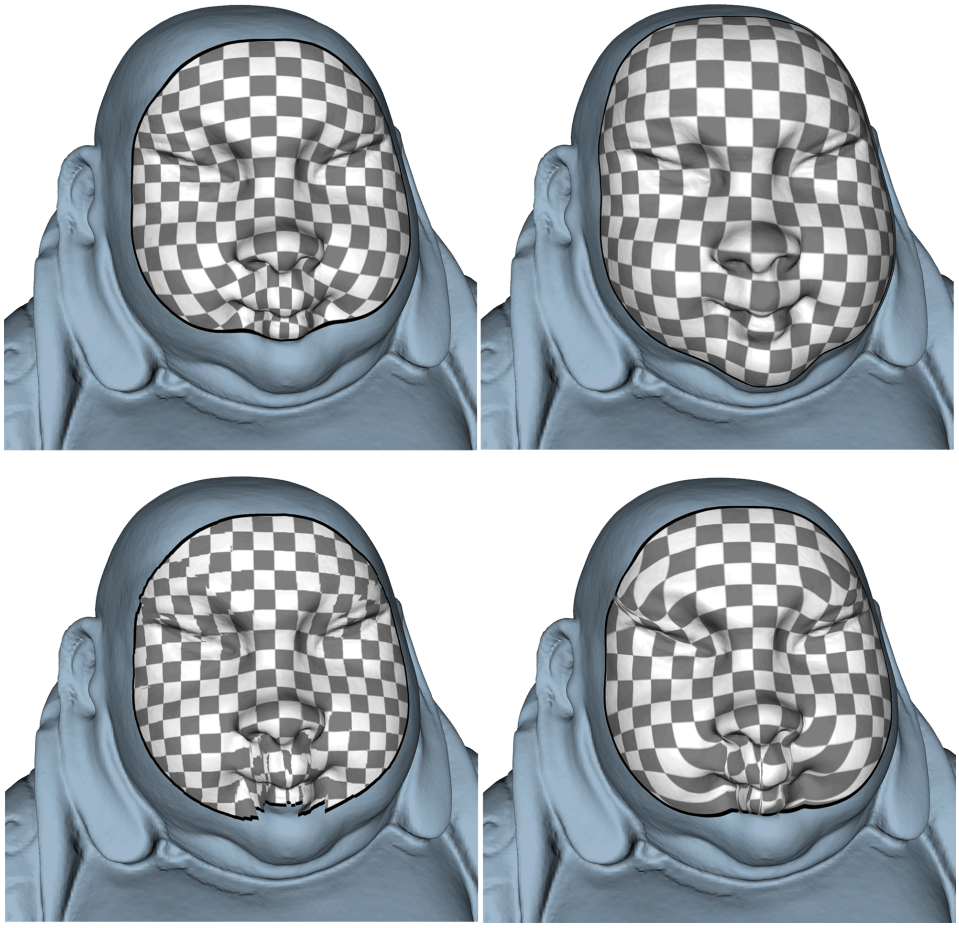


Figure 6.9: *Comparison of our method (top left), conformal map (top right), DEM (lower left) and DGPC (lower right).*

but exhibit artifacts. The parameter is therefore useful in order to tune the trade off between accuracy and smoothness. Unless noted otherwise, we use the squared mean edge length h^2 for this parameter, as suggested by Crane et al. [17].

Both Dijkstra based algorithms propagate information along one path and introduce error due to the inherent assumption that mappings between tangent spaces are isotropic, which is not true in general. Moreover, this implies that in cases where geodesics are not unique the algorithm decides arbitrarily on one specific path. Heat diffusion, in contrast, can be interpreted as tracing a large number of particles moving randomly across the mesh [56]. Our approach therefore propa-

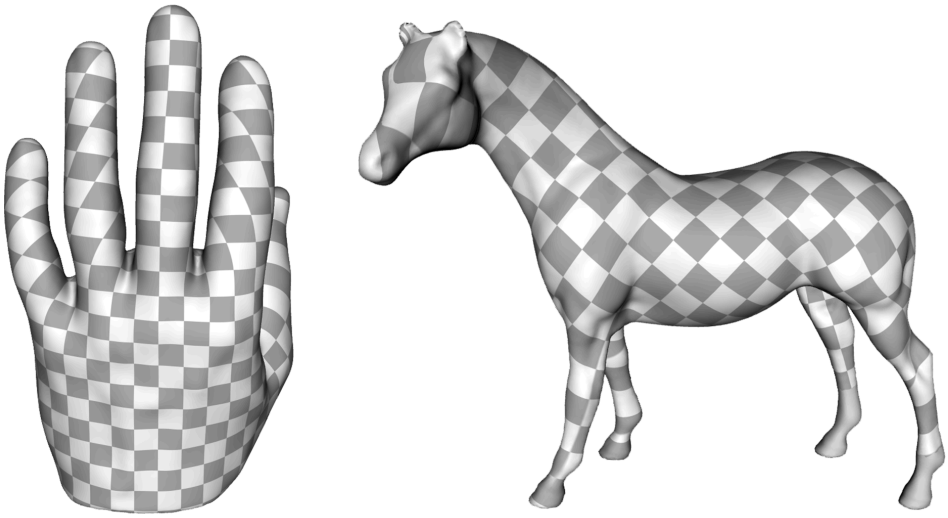


Figure 6.10: *Our exponential maps extend smoothly across the whole surface except for the cut locus.*

gates information along all possible paths simultaneously and consequently yields smoother maps. For planar meshes the exponential maps by Schmidt et al. [86] and Malvær et al. [71] both produce exact results by construction.

Accuracy To evaluate accuracy we compare results on three analytically defined geometries for which we can can explicitly compute values of the exponential map. We do not expect our method to outperform the other two in every case because these geometries are very smooth and therefore represent cases where the inherent assumptions of the Dijkstra based methods are very well fulfilled.

For the sphere we can easily determine the correct value at every vertex. For the two dimensional Gauss function we use the fact that it is rotationally symmetric and therefore directly gives the angular part of the exponential map when using the maximum as origin. The distance can be computed by numerical integration along a one dimensional vertical slice of the object. The saddle mesh is defined in terms of sine functions. By solving a differential equation with appropriate boundary conditions we can trace geodesics across the surface. For these geodesics the an-

gle is constant and the distance can be easily determined by measuring arc length. We numerically solve the differential equations using Mathematica and achieve a residual error below 10^{-6}. We project points sampled from a set of geodesics, starting at the map origin, onto the mesh and compare values by linear interpolation across the mesh triangles.

To investigate the influence of mesh resolution we compare results for different mean edge lengths. For each method and mesh we measure the error in the distance and angle component (E_d and E_a, respectively) of the exponential map and report averaged values in Table 6.8.1. Figure 6.14 shows error plots for the sphere and Gauss meshes as well as the resulting map. For the saddle mesh only the maps are shown because the reference solution is only defined along a discrete set of geodesics.

Our method tends to yield better results if the tessellation is relatively fine. While the average error is in most cases smaller for our method compared to DEM, both in terms of distance and angle, the DGPC still produces better results in many cases. In terms of visual quality our result matches those of DGPC on the test meshes.

The influence of tessellation quality on accuracy is illustrated in Figure 6.12. The first mesh is a very regular triangulation of the plane and serves as reference solution. For the second mesh 50% of the edges have been randomly flipped resulting in non-regular and nearly degenerate triangles. This introduces artifacts in the exponential map but the overall quality is preserved, even at some distance from the center. The errors in distance are 0.72×10^{-2} and 0.205×10^{-2} in angle as compared to 0.91×10^{-3} and 0.9×10^{-3} respectively for the regular mesh. For an anisotropic mesh, generated by anisotropic scaling, we see significant distortion close to the center, however, at some distance accuracy can be recovered. The errors for distance and angle are 0.77×10^{-2} and 0.23×10^{-2}. Since these meshes are planar most method based on exact polyhedral geodesics or Dijkstra based propagation will yield exact result. Our algorithm is useful whenever performance and smoothness is more important than robustness against very irregular meshes.

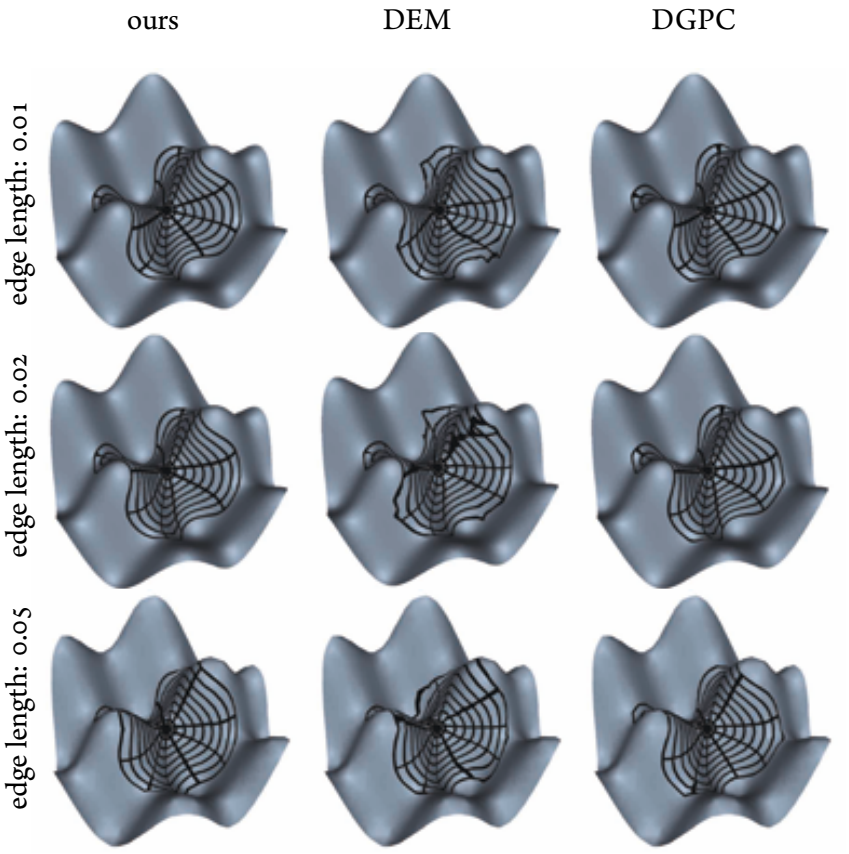


Figure 6.11: *Exponential maps at a saddle point for meshes of different resolution.*

6.8 Conclusion

We presented an algorithm to generate smoothed exponential maps on discrete surfaces. The method is based on discrete heat diffusion and localized solutions of linear systems. The most important observation is that exploiting linear solvers that provide localization and using system factorization as a precomputation step, it is possible to improve computation speed over simple traversal algorithms. As computing angles of the paths is similar to solving a transport problem along the path this observation could be very useful in a wide range of geometry processing algorithms.

While we believe the local parameterizations our approach provides are improving

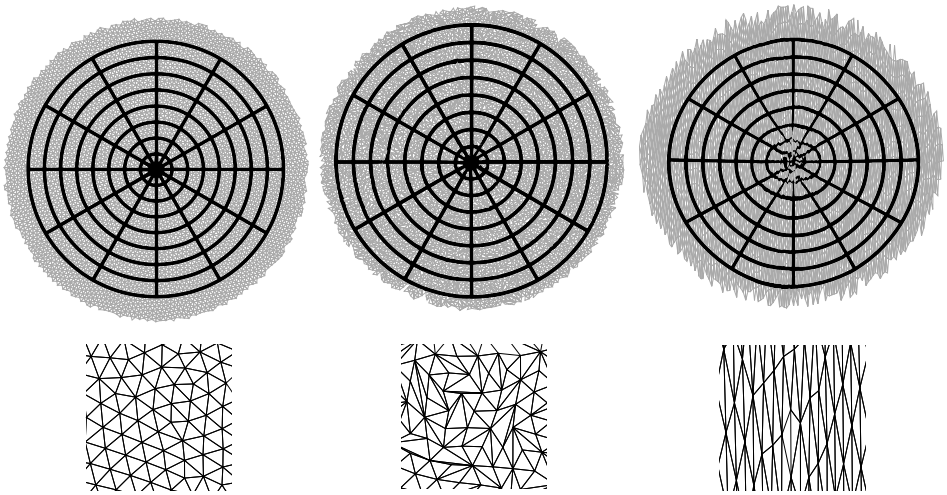


Figure 6.12: *Influence of mesh quality on accuracy. Our method performs well also for irregular (center) and anisotropically scaled meshes (right).*

on earlier approaches also in terms of quality, it would be great to provide some guarantees in terms of the quality of the mapping. Given the complexity of current algorithms that provide such guarantees it may be doubtful that it is possible to obtain mappings with guarantees at interactive rates. The option to increase the diffusion time seems to be a good compromise in this direction.

		Sphere			Gauss			Saddle		
		0.001	0.002	0.005	0.001	0.002	0.005	0.001	0.002	0.005
distance	heat	$0.612 \cdot 10^{-3}$	$0.211 \cdot 10^{-2}$	$0.174 \cdot 10^{-2}$	$0.593 \cdot 10^{-3}$	$0.584 \cdot 10^{-3}$	$0.142 \cdot 10^{-2}$	$0.324 \cdot 10^{-3}$	$0.928 \cdot 10^{-3}$	$0.173 \cdot 10^{-2}$
	DEM	$0.125 \cdot 10^{-2}$	$0.148 \cdot 10^{-2}$	$0.912 \cdot 10^{-3}$	$0.453 \cdot 10^{-2}$	$0.364 \cdot 10^{-2}$	$0.863 \cdot 10^{-2}$	$0.287 \cdot 10^{-3}$	$0.963 \cdot 10^{-3}$	$0.197 \cdot 10^{-2}$
	DGPC	$0.9 \cdot 10^{-4}$	$0.174 \cdot 10^{-3}$	$0.279 \cdot 10^{-3}$	$0.523 \cdot 10^{-3}$	$0.221 \cdot 10^{-3}$	$0.138 \cdot 10^{-2}$	$0.524 \cdot 10^{-4}$	$0.135 \cdot 10^{-3}$	$0.440 \cdot 10^{-3}$
angle	heat	$0.186 \cdot 10^{-3}$	$0.255 \cdot 10^{-3}$	$0.107 \cdot 10^{-2}$	$0.212 \cdot 10^{-3}$	$0.629 \cdot 10^{-3}$	$0.461 \cdot 10^{-2}$	$0.384 \cdot 10^{-2}$	$0.848 \cdot 10^{-2}$	$0.187 \cdot 10^{-1}$
	DEM	$0.325 \cdot 10^{-3}$	$0.538 \cdot 10^{-3}$	$0.138 \cdot 10^{-3}$	$0.129 \cdot 10^{-2}$	$0.168 \cdot 10^{-2}$	$0.829 \cdot 10^{-2}$	$0.323 \cdot 10^{-1}$	$0.376 \cdot 10^{-1}$	$0.402 \cdot 10^{-1}$
	DGPC	$0.409 \cdot 10^{-4}$	$0.871 \cdot 10^{-4}$	$0.21 \cdot 10^{-3}$	$0.179 \cdot 10^{-3}$	$0.408 \cdot 10^{-4}$	$0.168 \cdot 10^{-2}$	$0.111 \cdot 10^{-2}$	$0.314 \cdot 10^{-2}$	$0.106 \cdot 10^{-1}$

Table 6.8.1: *Accuracy measurements for the experiments illustrated in Figures 6.14 and 6.11. Average errors in the angle and distance component with respect to reference solutions are reported for different mesh resolutions.*

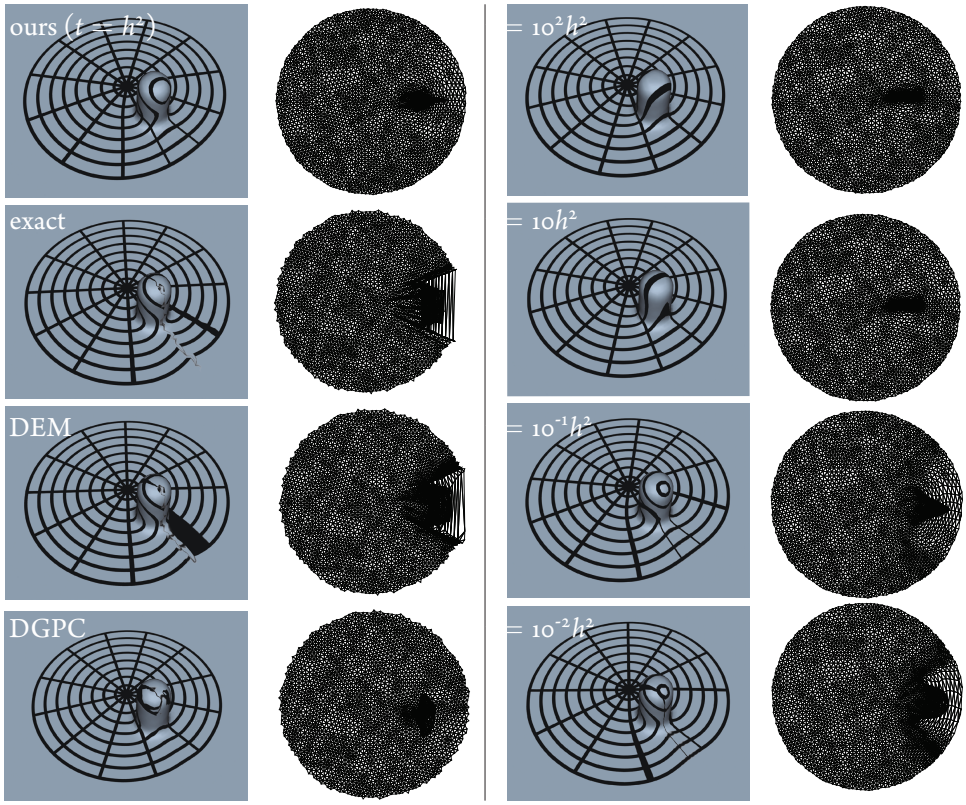


Figure 6.13: *Exponential maps in a challenging case including parts of the cut locus. Left: Our method compared to exact polyhedral geodesics, DEM and DGPC. Right: By varying the time step t of the diffusion problem the smoothness of the map can be influenced. As default choice we use the squared mean edge length h^2 as suggested by Crane et al. [17]. For smaller time steps the exponential maps become similar to the result using exact polyhedral geodesics but are smoother.*

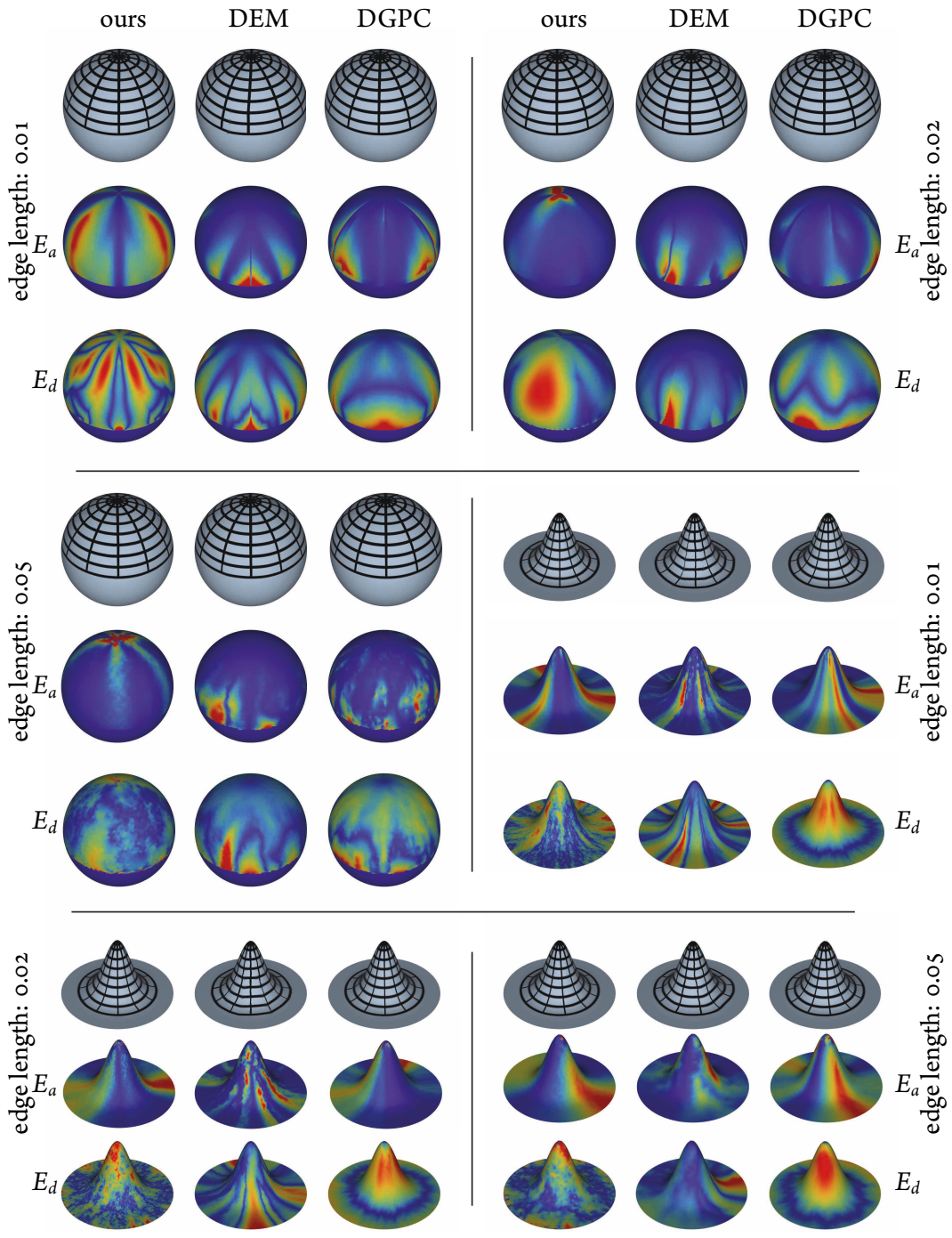


Figure 6.14: *Comparing our method, exact polyhedral geodesics, DGPC and discrete exponential maps on basic geometries. The map and errors in angle (E_a) and distance (E_d) are visualized for different mesh resolutions.*

7

Conclusion

In this thesis I demonstrated how problems in geometry processing can be efficiently solved by jointly employing insights from discrete differential geometry and numerical linear algebra. I argue that many challenging tasks can be tackled by looking at the tight interplay between both fields rather than treating them as two distinct entities. Commonly numerical tools are used as black boxes, and for good reasons. Using well established libraries leverages decades of expert knowledge. However, the price we pay for this convenience lies in the generality of the libraries interfaces. Direct sparse linear solvers, for example, accept arbitrary sparse matrices that are subsequently factorized, only imposing global constraints such as symmetry. In this thesis, and in geometry processing in general, the sparse matrices are most of the time related to the structure of a (triangle) mesh graph. Consequently, the matrix has a very distinct structure. The solver is oblivious to the source of the sparse matrix and can not exploit any of these specific properties. Moreover, when

approximate solutions are sought, it is very hard to communicate what kind of approximation is tolerable. The approximate sparse solve in Section 3.6, for example, requires knowledge about the spatial decay properties of this particular linear problem. Opening the black box and exploiting the geometric nature of the input data can reveal opportunities to find elegant and efficient solutions to common problems. These optimized algorithms are not merely improvements that lead to a performance increase by a fixed constant, they can fundamentally change the algorithms complexity. The exact sparse solve in Chapter 4 allows to solve for a single value of the solution in $\mathcal{O}(n \log(n))$ instead of $\mathcal{O}(n^2)$, if the meshes allow for a balanced elimination tree which is usually the case. Even though the exact sparse solve gives the exact results, its overhead is negligible and it can always be employed whenever the solution is only relevant in a subset of the variables.

7.1 Contributions

In order to solve discrete geometric problems involving the Laplacian, knowledge about the discretization of the operator is paramount. In Chapter 2 I elaborate on a set of well established properties that one wants to maintain in the discretization of the Laplacian. As demonstrated by Wardetzky et al. [107] these properties are in general incompatible and a construction of a perfect Laplacian does not exist. I propose a change of perspective: instead of constructing a perfect discrete Laplacian for the given mesh one can find a mesh that is close to it but admits a perfect Laplacian. My contribution is an iterative algorithm that finds such a mesh and its associated Laplacian. Contrary to most other constructions, our algorithm naturally generalizes to polygonal meshes.

Using a discrete Laplacian we can approximate heat diffusion. Crane et al. [17] demonstrated that heat diffusion can be used to compute geodesic distances on the mesh surface. I take this concept and use it to compute centroidal Voronoi diagrams in Chapter 3. My contributions include the intrinsic approximation of centroids on surfaces, where previous approaches use the embedding of the mesh. Moreover, I demonstrate how local heat diffusion can be efficiently approximated by selective back substitution.

Motivated by the approximate local back substitution I present an exact sparse solution method in Chapter 4. The method also does not compute the solution to all variables but still yields the exact result for a subset of them. This can be achieved by analyzing the dependencies among variables with the help of the elimination tree.

While localized solves are efficient for diffusion type problems, we commonly want to solve local problems that only differ from a global one in the boundary conditions. Since this requires an update of the system matrix, the exact sparse solve can not be employed directly. Chapter 5 proposes a solution to this problem. Instead of refactoring the operator of the local problem, I demonstrate how the factorization of the local problem, including boundary conditions, can be computed faster by reusing information from the factorization of the global linear operator.

Chapter 6 demonstrates how to compute the exponential map with the help of heat diffusion. I propose an algorithm that extends the computation of geodesic distances, as introduced by Crane et al. [17], to also compute the angle of the geodesic in the tangent plane of the source vertex. This algorithm constitutes an example that benefits from the exact sparse solve as introduced in Chapter 4. The algorithm is competitive in terms of performance and produces smooth maps while still being accurate where competing approaches introduce artifacts.

7.2 Outlook

Exploiting the structure of Cholesky factorizations can have an impact in many areas of geometry processing and the presented algorithms just scratch the surface of potential applications. One possible direction is the approximate update of Cholesky factors. For non-linear deformation energies one commonly needs to factor system matrices at every step of a simulation. Being able to update the factors when geometry changes potentially improves performance significantly. Hecht et al. [46] propose a first step in this direction and achieve a speedup factor of about 2. Systematically analyzing the numeric effect of updating a specific column of the factor with the help of the elimination tree can potentially lead to

much faster algorithms.

While sparse matrix factorizations can be efficient and robust for medium size meshes up to a million vertices, they reach their limits for larger meshes in terms of runtime and memory. A common solution is to resort to iterative solvers like the conjugate gradient method. These solvers can be very efficient but, unlike direct methods, their convergence rate depends on the stiffness of the linear problem. Using insights into the structure of factorization problems can help to implement hybrid algorithms unifying the advantages of both approaches: limited memory consumption and robustness. Several approaches in this direction for fluid solvers based on the Schur complement exist, e.g. [61]. However, scalable general purpose solvers on triangle meshes are still an open problem. In light of the hierarchical structure induced by the nested dissection ordering it will also be interesting to combine factorizations with multigrid approaches.

The nested dissection algorithm usually constitutes another black box. Since reordering can, depending on the implementation, take about the same time as the factorization itself, this seems to be an interesting direction of research. Common implementations such as Metis [53] assume a general input graph. Again, we could exploit our knowledge about the mesh graph. Especially the fact that meshes are usually embedded could help to find close to optimal separators.

On a broader scale, many numerical algorithms like iterative liner solvers, nonlinear optimization methods and sparse eigensolvers commonly represent black boxes that can potentially be opened and optimized to leverage geometric a priori knowledge about the input data.

Bibliography

[1] Marc Alexa and Max Wardetzky. Discrete Laplacians on General Polygonal Meshes. *ACM Trans. Graph.*, 30(4):102:1–102:10, July 2011. ISSN 0730-0301. doi:10.1145/2010324.1964997.

[2] Pierre Alliez, Éric Colin de Verdière, Olivier Devillers, and Martin Isenburg. Centroidal Voronoi Diagrams for Isotropic Surface Remeshing. *Graph. Models*, 67(3):204–231, May 2005. ISSN 1524-0703. doi:10.1016/j.gmod.2004.06.007.

[3] Franz Aurenhammer. A Criterion for the Affine Equivalence of Cell Complexes in $\mathbb{R}^d$ and Convex Polyhedra in $\mathbb{R}^{d+1}$. *Discrete Comput. Geom.*, 2(1): 49–64, December 1987. ISSN 0179-5376. doi:10.1007/BF02187870.

[4] Alexander G. Belyaev and Pierre-Alain Fayolle. On Variational and PDE-Based Distance Function Approximations. *Comput. Graph. Forum*, 34(8): 104–118, December 2015. ISSN 0167-7055. doi:10.1111/cgf.12611.

[5] William Benjamin, Andrew Wood Polk, S.V.N. Vishwanathan, and Karthik Ramani. Heat Walk: Robust Salient Segmentation of Non-rigid Shapes. *Computer Graphics Forum*, 30(7):2097–2106, 2011. doi:10.1111/j.1467-8659.2011.02060.x.

[6] Alexander I. Bobenko and Boris A. Springborn. A Discrete Laplace-Beltrami Operator for Simplicial Surfaces. *Discrete Comput. Geom.*, 38(4): 740–756, December 2007. ISSN 0179-5376. doi:10.1007/s00454-007-9006-1.

[7] Mario Botsch and Olga Sorkine. On Linear Variational Surface Deformation Methods. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):213–230, January 2008. ISSN 1077-2626. doi:10.1109/TVCG.2007.1054.

[8] Sofien Bouaziz, Mario Deuss, Yuliy Schwartzburg, Thibaut Weise, and Mark Pauly. Shape-Up: Shaping Discrete Geometry with Projections. *Comput. Graph. Forum*, 31(5):1657–1667, August 2012. ISSN 0167-7055. doi:10.1111/j.1467-8659.2012.03171.x.

[9] Herm Jan Brascamp and Elliott H Lieb. On extensions of the Brunn-Minkowski and Prékopa-Leindler theorems, including inequalities for log concave functions, and with an application to the diffusion equation. *Journal of Functional Analysis*, 22(4):366 – 389, 1976. ISSN 0022-1236. doi:10.1016/0022-1236(76)90004-5.

[10] Alexander M. Bronstein, Michael M. Bronstein, Leonidas J. Guibas, and Maks Ovsjanikov. Shape Google: Geometric Words and Expressions for Invariant Shape Retrieval. *ACM Trans. Graph.*, 30(1):1:1–1:20, February 2011. ISSN 0730-0301. doi:10.1145/1899404.1899405.

[11] Guillermo D. Cañas and Steven J. Gortler. Surface Remeshing in Arbitrary Codimensions. *Vis. Comput.*, 22(9):885–895, September 2006. ISSN 0178-2789. doi:10.1007/s00371-006-0073-8.

[12] Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder. A Simple Geometric Model for Elastic Deformations. *ACM Trans. Graph.*, 29(4): 38:1–38:6, July 2010. ISSN 0730-0301. doi:10.1145/1778765.1778775.

[13] Jindong Chen and Yijie Han. Shortest Paths on a Polyhedron. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, SCG '90, pages 360–369, New York, NY, USA, 1990. ACM. ISBN 0-89791-362-0. doi:10.1145/98524.98601.

[14] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw.*, 35(3):22:1–22:14, October 2008. ISSN 0098-3500. doi:10.1145/1391989.1391995.

[15] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker. Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps. *Proceedings of the National Academy of Sciences*, 102(21):7426–7431, 2005. ISSN 0027-8424. doi:10.1073/pnas.0500334102.

[16] Keenan Crane, Ulrich Pinkall, and Peter Schröder. Spin Transformations of Discrete Surfaces. *ACM Trans. Graph.*, 30(4):104:1–104:10, July 2011. ISSN 0730-0301. doi:10.1145/2010324.1964999.

[17] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in Heat: A New Approach to Computing Distance Based on Heat Flow. *ACM Trans. Graph.*, 32(5):152:1–152:11, October 2013. ISSN 0730-0301. doi:10.1145/2516971.2516977.

[18] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. ISBN 0898716136. doi:10.1137/1.9780898718881.

[19] Timothy A Davis. User Guide for LDL, a concise sparse Cholesky package, 2011.

[20] Timothy A. Davis and William W. Hager. Modifying a Sparse Cholesky Factorization. *SIAM J. Matrix Anal. Appl.*, 20(3):606–627, May 1999. ISSN 0895-4798. doi:10.1137/S0895479897321076.

[21] Timothy A. Davis and William W. Hager. Multiple-Rank Modifications of a Sparse Cholesky Factorization. *SIAM J. Matrix Anal. Appl.*, 22(4):997–1013, July 2000. ISSN 0895-4798. doi:10.1137/S0895479899357346.

[22] Timothy A. Davis and William W. Hager. Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves. *ACM Trans. Math. Softw.*, 35(4):27:1–27:23, February 2009. ISSN 0098-3500. doi:10.1145/1462173.1462176.

[23] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2016. doi:10.1017/S0962492916000076.

[24] Fernando de Goes, Pierre Alliez, Houman Owhadi, and Mathieu Desbrun. On the Equilibrium of Simplicial Masonry Structures. *ACM Trans. Graph.*, 32(4):93:1–93:10, July 2013. ISSN 0730-0301. doi:10.1145/2461912.2461932.

[25] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 317–324, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-48560-5. doi:10.1145/311535.311576.

[26] Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic Parameterizations of Surface Meshes. *Computer Graphics Forum*, 21(3):209–218, 2002. doi:10.1111/1467-8659.00580.

[27] M.P. do Carmo. *Differential geometry of curves and surfaces*. Prentice-Hall, 1976.

[28] Gerhard Dziuk and Charles M. Elliott. Finite element methods for surface PDEs. *Acta Numerica*, 22:289–396, 2013. doi:10.1017/S0962492913000056.

[29] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary Meshes. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 173–182, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-4. doi:10.1145/218380.218440.

[30] Herbert Edelsbrunner, M. J. Ablowitz, S. H. Davis, E. J. Hinch, A. Iserles, J. Ockendon, and P. J. Olver. *Geometry and Topology for Mesh Generation (Cambridge Monographs on Applied and Computational Mathematics)*. Cambridge University Press, New York, NY, USA, 2006. ISBN 052168207X. doi:10.1017/CBO9780511530067.

[31] Sharif Elcott and Peter Schröder. Building Your Own DEC at Home. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. doi:10.1145/1198555.1198667.

[32] Matthew Fisher, Boris Springborn, Alexander I. Bobenko, and Peter Schroder. An Algorithm for the Construction of Intrinsic Delaunay Triangulations with Applications to Digital Geometry Processing. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 69–74, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi:10.1145/1185657.1185668.

[33] Michael S. Floater. Parametrization and Smooth Approximation of Surface Triangulations. *Comput. Aided Geom. Des.*, 14(3):231–250, April 1997. ISSN 0167-8396. doi:10.1016/S0167-8396(96)00031-3.

[34] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. Random-Walk Computation of Similarities Between Nodes of a Graph with Application to Collaborative Recommendation. *IEEE Trans. on Knowl. and Data Eng.*, 19(3):355–369, March 2007. ISSN 1041-4347. doi:10.1109/TKDE.2007.46.

[35] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. doi:10.1137/0710032.

[36] Alan George and Joseph W. H. Liu. An Automatic Nested Dissection Algorithm for Irregular Finite Element Problems. *SIAM Journal on Numerical Analysis*, 15(5):1053–1069, 1978. ISSN 00361429. URL http://www.jstor.org/stable/2156722.

[37] John R. Gilbert and Tim Peierls. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM J. Sci. Stat. Comput.*, 9(5):862–874, September 1988. ISSN 0196-5204. doi:10.1137/0909058.

[38] K. Gębal, J. A. Bærentzen, H. Aanæs, and R. Larsen. Shape Analysis Using the Auto Diffusion Function. In *Proceedings of the Symposium on Geometry Processing*, SGP '09, pages 1405–1413, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association. doi:10.1111/j.1467-8659.2009.01517.x.

[39] David Glickenstein. Geometric triangulations and discrete Laplacians on manifolds. *arXiv Mathematics e-prints*, Aug 2005. URL https://arxiv.org/abs/math/0508188.

[40] David Glickenstein. A Monotonicity Property for Weighted Delaunay Triangulations. *Discrete Comput. Geom.*, 38(4):651–664, December 2007. ISSN 0179-5376. doi:10.1007/s00454-007-9009-y.

[41] Fernando de Goes, Pooran Memari, Patrick Mullen, and Mathieu Desbrun. Weighted Triangulations for Geometry Processing. *ACM Trans. Graph.*, 33 (3):28:1–28:13, June 2014. ISSN 0730-0301. doi:10.1145/2602143.

[42] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. ISBN 0-8018-5414-8.

[43] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3, 2010. URL http://eigen.tuxfamily.org.

[44] Igor Guskov. Manifold-based Approach to Semi-regular Remeshing. *Graph. Models*, 69(1):1–18, January 2007. ISSN 1524-0703. doi:10.1016/j.gmod.2006.05.001.

[45] Richard S. Hamilton. A matrix Harnack estimate for the heat equation. *Communications in Analysis and Geometry*, 1(1):113–126, 1993. doi:10.4310/CAG.1993.v1.n1.a6.

[46] Florian Hecht, Yeon Jin Lee, Jonathan R. Shewchuk, and James F. O'Brien. Updated Sparse Cholesky Factors for Corotational Elastodynamics. *ACM*

Trans. Graph., 31(5):123:1–123:13, September 2012. ISSN 0730-0301. doi:10.1145/2231816.2231821.

[47] Intel. *Intel Math Kernel Library. Reference Manual.* Intel Corporation, 2009.

[48] Ilse C. F. Ipsen. Computing an Eigenvector with Inverse Iteration. *SIAM Rev.*, 39(2):254–291, June 1997. ISSN 0036-1445. doi:10.1137/S0036144596300773.

[49] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded Biharmonic Weights for Real-time Deformation. *ACM Trans. Graph.*, 30(4): 78:1–78:8, July 2011. ISSN 0730-0301. doi:10.1145/2010324.1964973.

[50] Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2018. URL http://libigl.github.io/libigl/.

[51] Rafel Jaume and Günter Rote. Recursively-Regular Subdivisions and Applications. *arXiv e-prints*, Oct 2013. URL https://arxiv.org/abs/1310.4372.

[52] Hermann Karcher. Riemannian Center of Mass and so called Karcher Mean. *arXiv e-prints*, Jul 2014. URL https://arxiv.org/abs/1407.2087.

[53] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998. ISSN 1064-8275. doi:10.1137/S1064827595287997.

[54] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with Dual Quaternions. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 39–46, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-628-8. doi:10.1145/1230100.1230107.

[55] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. *Proceedings of the National Academy of Sciences*, 95(15):8431–8435, 1998. ISSN 0027-8424. doi:10.1073/pnas.95.15.8431.

[56] G.F. Lawler. *Random Walk and the Heat Equation.* Student mathematical library. American Mathematical Society, 2010. doi:10.1090/stml/055.

[57] Yunjin Lee, Hyoung Seok Kim, and Seungyong Lee. Mesh parameterization with a virtual boundary. *Computers & Graphics*, 26(5):677 – 686, 2002. ISSN 0097-8493. doi:10.1016/S0097-8493(02)00123-1.

[58] Bruno Lévy and Hao (Richard) Zhang. Spectral Mesh Processing. In *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10, pages 8:1–8:312, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0395-8. doi:10.1145/1837101.1837109.

[59] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least Squares Conformal Maps for Automatic Texture Atlas Generation. *ACM Trans. Graph.*, 21(3):362–371, July 2002. ISSN 0730-0301. doi:10.1145/566654.566590.

[60] Yaron Lipman, Raif M. Rustamov, and Thomas A. Funkhouser. Biharmonic Distance. *ACM Trans. Graph.*, 29(3):27:1–27:11, July 2010. ISSN 0730-0301. doi:10.1145/1805964.1805971.

[61] Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. A Scalable Schur-complement Fluids Solver for Heterogeneous Compute Platforms. *ACM Trans. Graph.*, 35(6):201:1–201:12, November 2016. ISSN 0730-0301. doi:10.1145/2980179.2982430.

[62] Joseph W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, January 1990. ISSN 0895-4798. doi:10.1137/0611010.

[63] Ligang Liu, Lei Zhang, Yin Xu, Craig Gotsman, and Steven J. Gortler. A Local/Global Approach to Mesh Parameterization. In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pages 1495–1504, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. doi:10.1111/j.1467-8659.2008.01290.x.

[64] Yang Liu, Balakrishnan Prabhakaran, and Xiaohu Guo. Point-Based Manifold Harmonics. *IEEE Transactions on Visualization and Computer Graphics*, 18(10):1693–1703, October 2012. ISSN 1077-2626. doi:10.1109/TVCG.2011.152.

[65] Yang Liu, Hao Pan, John Snyder, Wenping Wang, and Baining Guo. Computing Self-supporting Surfaces by Regular Triangulation. *ACM Trans. Graph.*, 32(4):92:1–92:10, July 2013. ISSN 0730-0301. doi:10.1145/2461912.2461927.

[66] Yong-Jin Liu, Chun-Xu Xu, Dian Fan, and Ying He. Efficient Construction and Simplification of Delaunay Meshes. *ACM Trans. Graph.*, 34(6):174:1–174:13, October 2015. ISSN 0730-0301. doi:10.1145/2816795.2818076.

[67] Lin Lu, Bruno Lévy, and Wenping Wang. Centroidal Voronoi Tessellation of Line Segments and Graphs. *Comput. Graph. Forum*, 31(2pt4):775–784, May 2012. ISSN 0167-7055. doi:10.1111/j.1467-8659.2012.03058.x.

[68] Li Ma. Partial convexity to the heat equation. *arXiv Mathematics e-prints*, Apr 2006. URL https://arxiv.org/abs/math/0604059.

[69] Jonathan Masci, Davide Boscaini, Michael M. Bronstein, and Pierre Vandergheynst. Geodesic Convolutional Neural Networks on Riemannian Manifolds. In *Proceedings of the 2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, ICCVW '15, pages 832–840, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-9711-7. doi:10.1109/ICCVW.2015.112.

[70] J. Clerk Maxwell. On Reciprocal Figures, Frames, and Diagrams of Forces. *Transactions of the Royal Society of Edinburgh*, 26(1):1–40, 1870. doi:10.1017/S0080456800026351.

[71] Eivind Lyche Melvær and Martin Reimers. Geodesic Polar Coordinates on Polygonal Meshes. *Comput. Graph. Forum*, 31(8):2423–2435, December 2012. ISSN 0167-7055. doi:10.1111/j.1467-8659.2012.03187.x.

[72] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds. In Hans-Christian Hege and Konrad Polthier, editors, *Visualization and Mathematics III*, pages 35–57, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-662-05105-4. doi:10.1007/978-3-662-05105-4_2.

[73] Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. The Discrete Geodesic Problem. *SIAM J. Comput.*, 16(4):647–668, August 1987. ISSN 0097-5397. doi:10.1137/0216045.

[74] Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. Spectral Conformal Parameterization. In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pages 1487–1494, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. doi:10.1111/j.1467-8659.2008.01289.x.

[75] Patrick Mullen, Pooran Memari, Fernando de Goes, and Mathieu Desbrun. HOT: Hodge-optimized Triangulations. *ACM Trans. Graph.*, 30(4):103:1–103:12, July 2011. ISSN 0730-0301. doi:10.1145/2010324.1964998.

[76] Daniele Panozzo, Ilya Baran, Olga Diamanti, and Olga Sorkine-Hornung. Weighted Averages on Surfaces. *ACM Trans. Graph.*, 32(4):60:1–60:12, July 2013. ISSN 0730-0301. doi:10.1145/2461912.2461935.

[77] Daniele Panozzo, Philippe Block, and Olga Sorkine-Hornung. Designing Unreinforced Masonry Models. *ACM Trans. Graph.*, 32(4):91:1–91:12, July 2013. ISSN 0730-0301. doi:10.1145/2461912.2461958.

[78] Giuseppe Patané and Michela Spagnuolo. Heat Diffusion Kernel and Distance on Surface Meshes and Point Sets. *Computers & Graphics*, 37(6): 676–686, October 2013. ISSN 0097-8493. doi:10.1016/j.cag.2013.05.019.

[79] Ulrich Pinkall and Konrad Polthier. Computing discrete minimal surfaces and their conjugates. *Experim. Math.*, 2:15–36, 1993. doi:10.1080/10586458.1993.10504266.

[80] Yipeng Qin, Xiaoguang Han, Hongchuan Yu, Yizhou Yu, and Jianjun Zhang. Fast and Exact Discrete Geodesic Computation Based on Triangle-oriented Wavefront Propagation. *ACM Trans. Graph.*, 35(4):125:1–125:13, July 2016. ISSN 0730-0301. doi:10.1145/2897824.2925930.

[81] S. Rippa. Minimal Roughness Property of the Delaunay Triangulation. *Comput. Aided Geom. Des.*, 7(6):489–497, October 1990. ISSN 0167-8396. doi:10.1016/0167-8396(90)90011-F.

[82] Raif M. Rustamov. Barycentric Coordinates on Surfaces. *Computer Graphics Forum*, 29(5):1507–1516, 2010. doi:10.1111/j.1467-8659.2010.01759.x.

[83] Rohan Sawhney and Keenan Crane. Boundary First Flattening. *ACM Trans. Graph.*, 37(1):5:1–5:14, December 2017. ISSN 0730-0301. doi:10.1145/3132705.

[84] Ryan Schmidt. *Part-Based Representation and Editing of 3D Surface Models*. PhD thesis, University of Toronto, Canada, 2010. URL http://hdl.handle.net/1807/29859.

[85] Ryan Schmidt. Stroke Parameterization. *Computer Graphics Forum*, 32 (2pt2):255–263, 2013. doi:10.1111/cgf.12045.

[86] Ryan Schmidt, Cindy Grimm, and Brian Wyvill. Interactive Decal Compositing with Discrete Exponential Maps. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 605–613, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi:10.1145/1179352.1141930.

[87] James Albert Sethian. A Fast Marching Level Set Method for Monotonically Advancing Fronts. *Proceedings of the National Academy of Sciences of the United States of America*, 93(4):1591–1595, 1996. ISSN 00278424. doi:10.1073/pnas.93.4.1591.

[88] Nicholas Sharp, Yousuf Soliman, and Keenan Crane. The Vector Heat Method. *arXiv e-prints*, May 2018. URL https://arxiv.org/abs/1805.09170.

[89] Hanxiao Shen, Zhongshi Jiang, Denis Zorin, and Daniele Panozzo. Progressive embedding. *ACM Trans. Graph.*, 38(4):32:1–32:13, July 2019. ISSN 0730-0301. doi:10.1145/3306346.3323012.

[90] Jonathan Richard Shewchuk. Delaunay Refinement Algorithms for Triangular Mesh Generation. *Comput. Geom. Theory Appl.*, 22(1-3):21–74, May 2002. ISSN 0925-7721. doi:10.1016/S0925-7721(01)00047-5.

[91] Robin Sibson. A brief description of natural neighbour interpolation. *Interpreting multivariate data*, 21:21–36, 1981.

[92] Justin Solomon, Fernando de Goes, Gabriel Peyré, Marco Cuturi, Adrian Butscher, Andy Nguyen, Tao Du, and Leonidas Guibas. Convolutional Wasserstein Distances: Efficient Optimal Transportation on Geometric Domains. *ACM Trans. Graph.*, 34(4):66:1–66:11, July 2015. ISSN 0730-0301. doi:10.1145/2766963.

[93] Olga Sorkine. Differential Representations for Mesh Processing. *Computer Graphics Forum*, 2006. ISSN 1467-8659. doi:10.1111/j.1467-8659.2006.00999.x.

[94] Olga Sorkine and Marc Alexa. As-rigid-as-possible Surface Modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP '07, pages 109–116, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-46-3. doi:10.1145/344779.344859.

[95] Olga Sorkine and Daniel Cohen-Or. Least-Squares Meshes. In *Proceedings of the Shape Modeling International 2004*, SMI '04, pages 191–199, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2075-8. doi:10.1109/SMI.2004.38.

[96] Olga Sorkine, Daniel Cohen-Or, Yaron. Lipman, Marc. Alexa, Christian Rössl, and Hans-Peter Seidel. Laplacian Surface Editing. In *Proceedings*

of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing, SGP '04, pages 175–184, New York, NY, USA, 2004. ACM. ISBN 3-905673-13-4. doi:10.1145/1057432.1057456.

[97] Jian Sun, Maks Ovsjanikov, and Leonidas Guibas. A Concise and Provably Informative Multi-scale Signature Based on Heat Diffusion. In *Proceedings of the Symposium on Geometry Processing*, SGP '09, pages 1383–1392, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association. doi:10.1111/j.1467-8659.2009.01515.x.

[98] Qian Sun, Long Zhang, Minqi Zhang, Xiang Ying, Shi-Qing Xin, Jiazhi Xia, and Ying He. Texture Brush: An Interactive Surface Texturing Interface. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 153–160, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1956-0. doi:10.1145/2448196.2448221.

[99] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. Fast Exact and Approximate Geodesics on Meshes. *ACM Trans. Graph.*, 24(3):553–560, July 2005. ISSN 0730-0301. doi:10.1145/1073204.1073228.

[100] Kenshi Takayama, Ryan Schmidt, Karan Singh, Takeo Igarashi, Tamy Boubekeur, and Olga Sorkine. GeoBrush: Interactive Mesh Geometry Cloning. *Computer Graphics Forum*, 30(2):613–622, 2011. doi:10.1111/j.1467-8659.2011.01883.x.

[101] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015. URL https://doc.cgal.org/latest/Manual/.

[102] W. T. Tutte. How to Draw a Graph. *Proceedings of the London Mathematical Society*, s3-13(1):743–767, 1963. doi:10.1112/plms/s3-13.1.743.

[103] S. R. S. Varadhan. On the behavior of the fundamental solution of the heat equation with variable coefficients. *Communications on Pure and Applied Mathematics*, 20(2):431–455, 1967. doi:10.1002/cpa.3160200210.

[104] Etienne Vouga, Mathias Höbinger, Johannes Wallner, and Helmut Pottmann. Design of Self-supporting Surfaces. *ACM Trans. Graph.*, 31(4):87:1–87:11, July 2012. ISSN 0730-0301. doi:10.1145/2185520.2185583.

[105] Xiaoning Wang, Xiang Ying, Yong-Jin Liu, Shi-Qing Xin, Wenping Wang, Xianfeng Gu, Wolfgang Mueller-Wittig, and Ying He. Intrinsic Computation of Centroidal Voronoi Tessellation (CVT) on Meshes.

Comput. Aided Des., 58(C):51–61, January 2015. ISSN 0010-4485. doi:10.1016/j.cad.2014.08.023.

[106] Xiaoning Wang, Zheng Fang, Jiajun Wu, Shi-Qing Xin, and Ying He. Discrete Geodesic Graph (DGG) for Computing Geodesic Distances on Polyhedral Surfaces. *Comput. Aided Geom. Des.*, 52(C):262–284, March 2017. ISSN 0167-8396. doi:10.1016/j.cagd.2017.03.010.

[107] Max Wardetzky, Saurabh Mathur, Felix Kälberer, and Eitan Grinspun. Discrete Laplace Operators: No Free Lunch. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP '07, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-46-3. doi:10.2312/SGP/SGP07/033-037.

[108] Shi-Qing Xin and Guo-Jin Wang. Improving Chen and Han's Algorithm on the Discrete Geodesic Problem. *ACM Trans. Graph.*, 28(4):104:1–104:8, September 2009. ISSN 0730-0301. doi:10.1145/1559755.1559761.

[109] Shi-Qing Xin, Xiang Ying, and Ying He. Efficiently Computing Geodesic Offsets on Triangle Meshes by the Extended Xin-Wang Algorithm. *Comput. Aided Des.*, 43(11):1468–1476, November 2011. ISSN 0010-4485. doi:10.1016/j.cad.2011.08.027.

[110] Dong-Ming Yan, Bruno Lévy, Yang Liu, Feng Sun, and Wenping Wang. Isotropic Remeshing with Fast and Exact Computation of Restricted Voronoi Diagram. In *Proceedings of the Symposium on Geometry Processing*, SGP '09, pages 1445–1454, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association. doi:10.1111/j.1467-8659.2009.01521.x.

[111] Yu-Hong Yeung, Jessica Crouch, and Alex Pothen. Interactively Cutting and Constraining Vertices in Meshes Using Augmented Matrices. *ACM Trans. Graph.*, 35(2):18:1–18:17, February 2016. ISSN 0730-0301. doi:10.1145/2856317.

[112] Xiang Ying, Xiaoning Wang, and Ying He. Saddle Vertex Graph (SVG): A Novel Solution to the Discrete Geodesic Problem. *ACM Trans. Graph.*, 32(6):170:1–170:12, November 2013. ISSN 0730-0301. doi:10.1145/2508363.2508379.