Sapienza University of Rome

Department of Computer Science

PhD in Computer Science (XXXV cycle)

Scalable Geometry Processing for Computer Graphics Applications

Filippo Maggioli
ID number 1645300

Advisor
Prof. Emanuele Rodolà

Academic Year 2023-2024

Thesis defended on 11th September 2023
in front of a Board of Examiners composed by:

Prof. Marco Anisetti (chair)
Department of Computer Science
University of Milan

Prof. Simone Melzi
Department of Informatics, Systems and Communication
University of Milano-Bicocca

Prof. Luca Cosmo
Department of Environmental Sciences, Computer Science and Statistics
Ca' Foscari University of Venice

External reviewers:

Prof. Marco Tarini
Department of Computer Science
University of Milan

Bruno Lévy
National Institute for Research in Digital Science and Technology (INRIA)

Scalable Geometry Processing for Computer Graphics Applications
Ph.D. thesis. Sapienza University of Rome

© 2023 Filippo Maggioli. All rights reserved

This thesis has been typeset by LaTeX.

Version: October 2023

Author's email: maggioli@di.uniroma1.it

Acknowledgments

My most sincere and profound gratitude goes to my supervisor, prof. Emanuele Rodolà, for his invaluable support, expert mentoring, and genuine amicability. I owe him my personal growth as a researcher and human, and I'm beholden to him for sharing his knowledge with me and guiding me throughout these years.

I would like to extend my thanks to prof. Dominik L. Michels, for his knowledgeable advice during my visiting at the King Abdullah University of Science and Technology.

A genuine sign of gratefulness goes to the external reviewers, prof. Marco Tarini and Bruno Lèvy, for their meticulous comments and adept suggestions that navigated me toward the improvement of my thesis work.

I profoundly thank my family for their love and patience. No words could ever express how grateful I am to them for their support over all these years.

I thoroughly thank my collaborators, my colleagues, and my friends. All the laughs and cries, all the cheerful and melancholic moments, all the emotions they shared with me have been precious, warmed my heart, and fed my humanity.

Abstract

This thesis explores and investigates scalable solutions, grounded on geometry processing and differential geometry concepts, to different computer graphics tasks. My Ph.D. path gave me the opportunity to probe many research topics in the field of computer graphics, as well as delve into mathematical and computational problems. As a summary of my research activity, this thesis echoes my exploration, collecting results from different areas of computer graphics and computational geometry. From novel unified frameworks in spectral geometry to procedural texturing techniques, simulations, and matrix multiplication algorithms, all the discussed topics find their communion in the idea of providing geometry processing solutions made to scale for large volumes of data.

Contents

Introduction to the Thesis

This thesis is divided into three parts, each representing a research field that I spanned through my Ph.D. path. The common ground for my research has been the area of computational and Riemannian geometry in computer graphics applications, for which an introduction is provided in the following chapters.

Part I is devoted to the field of spectral geometry processing, which denoted the beginning of my Ph.D. career. When I started studying the notion of *Laplacian eigenproducts* [154] and their application in geometry processing [193], I immediately recognized the potential for speeding up many tasks in applied spectral geometry, like approximation and transfer of functions. Extending an eigenbasis with point-wise products of its elements is dramatically more efficient than computing a larger basis, and in Chapter 1 we present theoretical studies that show how they also provide a massive increase in the descriptive power. However, working with a non-orthogonal basis introduces numerical stability issues, and makes it difficult to extend the method and integrating it into other pipelines that intrinsically require an orthogonal setting (*e.g.*, ZoomOut [176]). So, in the chapter we also provide a groundwork for the orthogonalization of the eigenproducts and the extension of the functional map framework [202] to integrate this new basis, proving its efficiency and effectiveness in multiple tasks, from the approximation and transfer of functions, to the filtering of details [161]. Working in the setting of non-rigid and non-isometric correspondences deepened my interest in spectral geometry, and led me supporting my colleagues in related projects. We started searching for spectral correlations between manifolds and submanifolds, with the idea that non-rigid puzzles [145] could have a solution in a spectral fashion. The results presented in Chapter 2 prove that, despite being hidden behind a neural network, this correlation does exist, and that it can be exploited to compute non-rigid and non-isometric set operations between partial shapes [185]. Since the method rely on a data-driven optimization, I collaborated to the generation of a large dataset of partial shapes labeled with spectral data, and to the strengthening of the theoretical foundations by defining a mathematical background and a symmetry between the problem in the spectral setting and the problem in the latent space.

When studying Riemannian geometry, I started having an interest in generalizing problems to non-Euclidean metric spaces. Part II of this thesis is centered around this idea. Discovering the complexity and visually interesting animations arising from the simulation of slime mold organisms [114] led me to the idea of bringing this problem on surfaces for procedural texturing tasks. In Chapter 3 we show how, by using the notion of motion on surfaces and carefully adapting the movement to the metric space of a mesh, this kind of simulation can be generalized to produce complex animated textures in real-time [160]. My work on the area of simulations got me in touch with researchers from the *King Abdullah University of Science and Technology* (KAUST), where I spent a three months visiting period working on a diffusion model on graphs to simulate the water flow inside plants. The results from Chapter 4 show how this model can be integrated with solvers for the dynamics of non-rigid bodies to produce interactive and visually convincing simulations of the wilting process of crop plants [159]. My studies on non-Euclidean geometries also led to an interest in the idea of curves and connectivities in curved domains. The seminal results presented in Chapter 5 show that it is possible to obtain an high-quality remeshing by computing a geodesic farthest point sampling and the corresponding geodesic Delaunay triangulation. In the same chapter we also prove that this result can be obtained efficiently by avoiding multiple traversals of the entire mesh [158]. Without the convenient structure of the Euclidean metric spaces, even problems with well-established solutions could become very though, and this is the case for the task of curve registration. After discovering the recent community interest in designing curves on surfaces [162], I started working on an algorithm for reconstructing curves from sparse samples in non-Euclidean metric spaces. In Chapter 6 we discuss an algorithm that solves this task by adapting known techniques for the traveling salesman problem to triangular mesh domains.

Searching for efficient solutions often brought me to questioning about parallel solutions for usually though problems. Part III collects my research results in this direction. Procedural texturing tasks are historically addressed by defining multidimensional noise functions and projecting them onto a surface [101], a result that can be achieved with real-time performance by taking advantage of GPU computing. In Chapter 7 we present a new technique in this fashion, that exploits pixel shaders to generate fractal patterns on surfaces. Another direction that I investigated is the improvement of matrix multiplication algorithms. Many applications in computational and spectral geometry deals with large collections of shapes [228, 241], and handling of large matrices is often involved, eventually in the setting of a distributed system. In Chapter 8 we discuss a Strassen-like algorithm that efficiently deals with the multiplication of a matrix by its transpose, and that can be easily parallelized in both shared and distributed memory environments.

Overview of Published Results

This thesis is organized so that every chapter collects independent results and defines its own context. Each chapter is dedicated to present the results of a single research project.

Chapter 1: the results presented in this chapter have been published in the proceedings of the *42nd Annual Conference of the European Association for Computer Graphics* (*EUROGRAPHICS 2021*) [161].

Chapter 2: the results presented in this chapter have been published in the proceedings of the *43rd Annual Conference of the European Association for Computer Graphics* (*EUROGRAPHICS 2022*) [185].

Chapter 3: the results presented in this chapter have been published in the proceedings of the *30th Pacific Conference on Computer Graphics and Applications* (*Pacific Graphics 2022*) [160].

Chapter 4: the results presented in this chapter have been published in the proceedings of the *16th ACM SIGGRAPH Conference and Exhibition on Computer Graphics and Interactive Techniques in Asia* (*SIGGRAPH Asia 2023*) [159].

Chapter 5: the results presented in this chapter have been published as a pre-print on *arXiv* [158]. The seminal results about the preservation of spectral properties are not yet part of the pre-print.

Chapter 6: the seminal results presented in this chapter have not yet been published.

Chapter 7: the results presented in this chapter have been published in the poster proceedings of the *49th Special Interest Group on Computer Graphics and Interactive Techniques* (*SIGGRAPH 2022*) [157].

Chapter 8: the results presented in this chapter have been published at the *50th International Conference on Parallel Processing* (*ICPP 2021*) [12].

Motivations

Since its advent in the ‘50s, modern computer graphics has asserted its importance in major research fields and applications, such as scientific visualization, computer-aided engineering, and entertainment. Over the years, the research community in the area vastly increased, alongside the number of applications, now ranging from graphics design to computational biology, covering simulation, photography, and virtual reality.

In this massive research area, geometry processing takes a large spot. Geometry processing is a field where concepts from mathematics and geometry are used to efficiently deal with 3D data, addressing tasks such as analysis, deformation, and reconstruction, among others. Many successful geometry processing algorithms exploit notions and solutions from differential geometry to handle and process complex surfaces in 3D, taking advantage of the formulation via their intrinsic properties, like metric, curvature, and differential operators.

For instance, simulations and optimization problems on the surface domain can produce complex textures and patterns programmatically [124, 253, 269]. Again, intrinsic properties like the curvature, or features of differential operators like the Laplacian, can be used as powerful descriptors for finding non-rigid correspondences between shapes [179, 202]. Moreover, the metric of the surface induces the definition of distances over a mesh, allowing for the design of curves, patterns, and boolean operations directly on the shape [162, 188, 230].

The noticeable achievements of differential geometry algorithms are quickly downsized when it comes to industrial applications, especially in the entertainment industry. Movies usually deal with meshes composed of millions of polygons; in video games, moderately sized shapes must be processed in a matter of milliseconds and with low memory consumption to achieve real-time performance on most home systems. The differential geometry component of the algorithm does not scale to this volume of data: an optimization problem with 20k variables is unsuitable for real-time applications, and the spectral decomposition of a matrix with 3M$\times$3M entries is an unfeasible task.

This thesis spans a variety of applications in computer graphics, ranging from shape matching and procedural texturing to simulations and curve design. The goal is to provide efficient and easily parallelizable algorithms that, while still relying on concepts of differential geometry, explore new directions and ideas and deep dive into the mathematical framework to produce accurate or visually convincing solutions that can scale to industry-standard data volumes.

Background in Riemannian Geometry

In this thesis, we extensively make use of notions about differential geometry, spectral geometry and metric spaces. This chapter serves as an introductory background for all the mathematical tools we refer and use throughout the thesis.

We introduce the notion of Riemannian manifolds, local parametrizations and tangent spaces, defining the notion of curves and distances on smooth surfaces. Then, we provide an introductory presentation on the generalization of differential operator and the properties of the Laplacian eigendecomposition, with an overview of the functional maps framework and its use in correspondence and shape matching applications.

Riemannian Manifolds

Informally speaking, a d-dimensional smooth manifold embedded in $\mathbb{R}^n$ is composed by gluing together deformed slices of d-dimensional hyper-planes in a n-dimensional space.

The basic building block for defining the structure of a Riemannian manifold is the notion of *chart*.

Definition 1. *Given a topological space $\mathcal{M} \subset \mathbb{R}^n$, a* chart *$(\varphi, U)$ of $\mathcal{M}$ is a couple formed by an open subset $U \subset \mathcal{M}$ and an homeomorphism $\varphi : U \to \mathbb{R}^d$.*

Linking back to our informal definition, a chart determines a slice of the d-dimensional hyper-plane and the rule for deforming it. The collection of all the slices is called *atlas*.

Definition 2. *Given a topological space $\mathcal{M} \subset \mathbb{R}^n$, an* atlas *is a collection of charts $\mathfrak{A}_\mathcal{M} = \{(\varphi_\alpha, U_\alpha) \ : \ \alpha \in A\}$, such that $\bigcup_{\alpha \in A} U_\alpha = \mathcal{M}$.*

For determining how to glue the pieces together, we need to capture the idea of consistently overlapping the charts.

Definition 3. *Given a topological space $\mathcal{M} \subset \mathbb{R}^n$ and two charts $(\varphi_\alpha, U_\alpha)$ and (φ_β, U_β) of $\mathcal{M}$ such that $U_\alpha \cap U_\beta \neq \emptyset$, the composition $\tau_{\alpha,\beta} = \varphi_\beta \circ \varphi_\alpha^{-1}$ is called* transition map*.*

Informally speaking, the transition map is the operation of deforming a slice of the hyper-plane according to a chart, taking the piece of that slice belonging

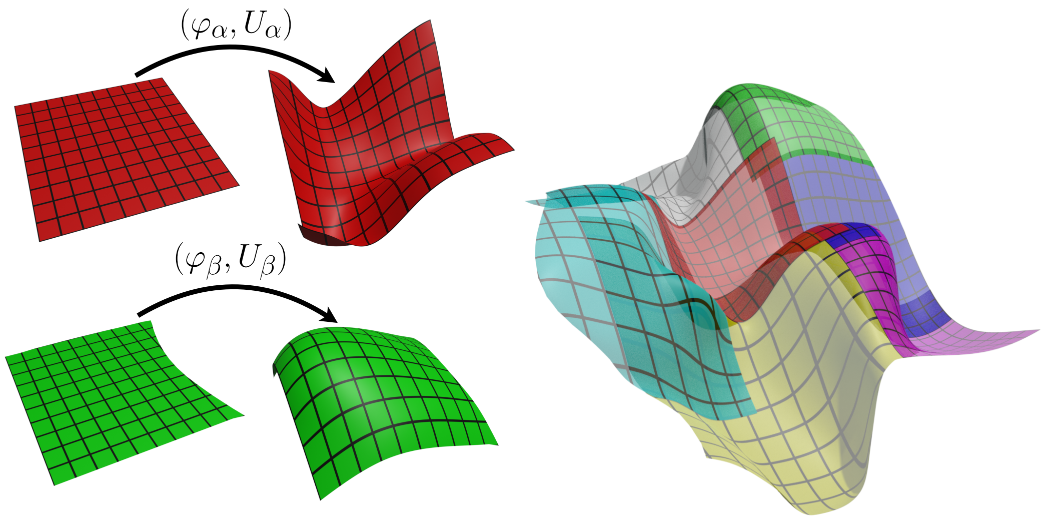


Figure i. Regions of the plane (left panel) are deformed according to different charts (middle plane). The deformation of multiple regions of the plane according to an atlas with smooth transition maps produces a smooth manifold (right panel). In this example, the manifold is 2-dimensional (*i.e.*, $d = 2$) and the embedding is 3-dimensional (*i.e.*, $n = 3$).

to second chart, and relaxing it back to the original hyper-plane according to the second chart.

Definition 4. *Given a topological space $\mathcal{M} \subset \mathbb{R}^n$ and an atlas $\mathfrak{A}_\mathcal{M} = \{(\varphi_\alpha : U_\alpha \to \mathbb{R}^d, U_\alpha) \ : \ \alpha \in A\}$, we call $\mathcal{M}$ a* smooth manifold *if, for every $\alpha, \beta \in A$ such that $U_\alpha \cap U_\beta \neq \emptyset$, the transition map $\tau_{\alpha,\beta}$ is a smooth map.*

The example in Figure i shows how an atlas defines a smooth surface. Each piece of the plane is deformed according to a corresponding chart. Then, if the charts are compatible in the overlapping regions, the resulting manifold is smooth.

While there would be much more to cover for properly defining a manifold, describing all the advanced theory goes beyond the scope of this thesis, and for further details we refer to the books from Do Carmo and Morita [73, 184].

At every point $x \in \mathcal{M}$, we can define a linear approximation of the surface around x, using the d-dimensional hyper-plane tangent to $\mathcal{M}$ at point x. We denote this tangent plane as $T_x(\mathcal{M})$. The disjoint union of all the tangent spaces is called *tangent bundle of* $\mathcal{M}$, and is denoted as $T(\mathcal{M}) = \bigcup_{x \in \mathcal{M}} T_x(\mathcal{M})$. Introducing the notion of tangent space allows us to define an inner product between vectors on the surface $\mathcal{M}$. Given a local chart (φ, U) such that for some $p \in \varphi(U)$ it holds $\varphi^{-1}(p) = x$, and by recalling that the tangent space $T_x(\mathcal{M})$ lies in $\mathbb{R}^n$, we can map d-dimensional vectors to $T_x(\mathcal{M})$ by using the Jacobian matrix $\mathcal{J}_{\varphi^{-1}}$. Thus, given two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d$, we can define their inner product on $T_x(\mathcal{M})$ as $g_x(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^\top \mathcal{J}_{\varphi^{-1}}^\top \mathcal{J}_{\varphi^{-1}} \boldsymbol{v}$. The products of the Jacobian matrices is usually called *metric tensor*, and it is denoted as the $d \times d$ symmetric matrix $\mathbf{g}_x = \mathcal{J}_{\varphi^{-1}}^\top \mathcal{J}_{\varphi^{-1}}$. So, the metric tensor encodes the distortion induced by mapping the vectors to the surface $\mathcal{M}$. Given a vector $\boldsymbol{v} \in \mathbb{R}^d$, we can define its length in $T_x(\mathcal{M})$ just as

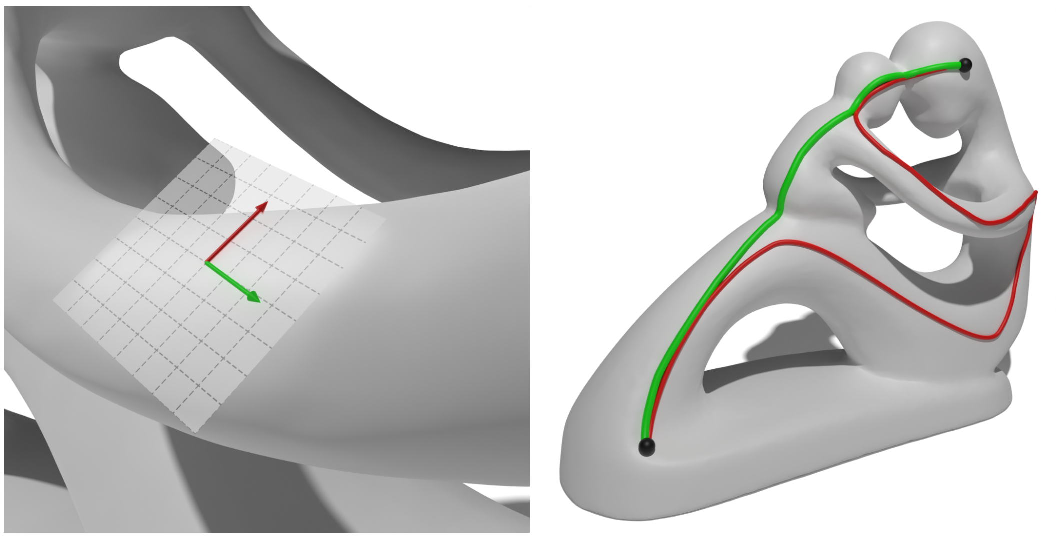


Figure ii. Left: the tangent plane to a point on the surface. Right: an arbitrary curve on the surface connecting two points (red) and the corresponding geodesic path (green).

$\|\boldsymbol{v}\|_{\mathbf{g}_x} = \sqrt{g_x(\boldsymbol{v},\boldsymbol{v})} = \sqrt{\boldsymbol{v}^\top \mathbf{g}\boldsymbol{v}}$. Defining a metric g_x at every point $x \in \mathcal{M}$, allows us to define an overall metric g over the entire manifold $\mathcal{M}$.

Definition 5. *A d-dimensional* Riemannian manifold *embedded in $\mathbb{R}^n$ is a couple $(\mathcal{M}, g)$, where $\mathcal{M}$ is a d-dimensional smooth manifold embedded in $\mathbb{R}^n$, and g is a metric over $\mathcal{M}$. The manifold $\mathcal{M}$ is said to be* equipped *with the metric g.*

Figure ii depicts an example of tangent plane to a point of the surface, with a corresponding local coordinate system.

Curves and Distances

A curve on a manifold $\mathcal{M}$ consists of a diffeomorphism $\gamma : (a,b) \subset \mathbb{R} \to \mathcal{M}$, and it is itself a 1-dimensional manifold embedded in the same space as $\mathcal{M}$. Given a local chart φ for $\mathcal{M}$ at the point x, we can define the differential of the curve at $\gamma(t) = x$ as $\frac{\mathrm{d}}{\mathrm{d}t}(\varphi \circ \gamma)$. This differential represents the infinitesimal movement of a point along the curve, and the directional vector lies in the tangent space $T_x(\mathcal{M})$. It can be proved that, independently of the chosen parametrization for the local chart, the resulting embedding of the tangent vector is the same [73, 184]. If $\mathcal{M}$ is a Riemannian manifold and g is the equipped metric, then we can use the tangent vectors to compute the length of the curve γ as

$$\ell(\gamma) = \int_a^b \sqrt{g_{\gamma(t)}(\gamma'(t), \gamma'(t))}\, \mathrm{d}t\,. \tag{i}$$

Determining the length of a curve allows us to define the notion of shortest path, and hence to generalize the idea of distance over surfaces.

Definition 6. *Let $(\mathcal{M}, g)$ be a d-dimensional Riemannian manifold embedded in $\mathbb{R}^n$, and let $x, y \in \mathcal{M}$ be two points over $\mathcal{M}$. Said $\Gamma_{x,y} = \{\gamma : (a,b) \to \mathcal{M} \ : \ \gamma(a) =$*

$x, \gamma(b) = y\}$ the set of curves on $\mathcal{M}$ having x and y as end-points, the geodesic path *connecting x to y is a curve $\gamma^*_{x,y} = \arg\min_{\gamma\in\Gamma_{x,y}} \ell(\gamma)$. The* geodesic distance *between x and y is the function $\delta_{\mathcal{M}}(x,y) = \min_{\gamma\in\Gamma_{x,y}} \ell(\gamma)$.*

So, intuitively, the geodesic path is the shortest walk on the surface connecting x and y, whereas the geodesic distance is the length of the geodesic path.

The example in Figure ii shows the difference between an arbitrary curve (highlighted in red) on the surface and the geodesic path (in green) connecting two points.

Differential and Spectral Geometry

Defining a metric induces the definition of distances over surfaces, but it also comes in handy for generalizing differential calculus.

Gradient Given some scalar field $f : \mathcal{M} \to \mathbb{R}$ over the surface $\mathcal{M}$, we want to generalize the notion of gradient of f. At any point $p \in \mathcal{M}$, we consider the basis $\{\partial_i = \frac{\partial\varphi^{-1}}{\partial x_i}\}$ for the tangent space $T_p(\mathcal{M})$ induced by the local chart φ. The directional derivative $\mathrm{d}f_p(\partial_i)$ of f at point p along the basis vector ∂_i is given by the chain rule $\mathrm{d}f_p(\partial_i) = \mathrm{d}f_p(\frac{\partial\varphi^{-1}}{\partial x_i}) = \frac{\partial}{\partial x_i}(f \circ \varphi^{-1}) = \frac{\partial \tilde{f}}{\partial x_i}$, where $\tilde{f} = f \circ \varphi^{-1}$ is the local parametrization of f according to the chart φ. Said $\nabla\tilde{f}$ the Euclidean gradient of $\tilde{f}$, the directional derivative $\mathrm{d}f_p(v)$ of f at point p along the direction vector $v \in T_p(\mathcal{M})$ can be obtained by linearity as $\mathrm{d}f_p(v) = \boldsymbol{v}^\top\nabla\tilde{f}$. By recalling that the directional derivative can also be expressed as the inner product between the gradient and v, and since the gradient of f must lie in the tangent space $T_p(\mathcal{M})$, the conversion to the Riemannian metric gives $\mathrm{d}f_p(v) = g_p(\mathrm{grad}(f), v)$, which we can use to solve for the gradient

$$\begin{gathered} g_p(\mathrm{grad}(f), v) = \boldsymbol{v}^\top \mathbf{g}_p \mathrm{grad}(f) = \mathrm{d}f_p(v) = \boldsymbol{v}^\top\nabla\tilde{f}\,, \qquad \forall v \in T_p(\mathcal{M})\,, \\ \mathrm{grad}(f) = \mathbf{g}^{-1}\nabla\tilde{f}\,. \end{gathered} \tag{ii}$$

Divergence For the generalization of the divergence, we rely on one of its alternate definitions. For any vector field $V : T(\mathcal{M}) \to \mathbb{R}^d$, and for any scalar field $f : \mathcal{M} \to \mathbb{R}$ with compact support, the divergence is the adjoint of the gradient. That is

$$\begin{gathered} \langle f,\ \mathrm{div}(V)\rangle_{\mathcal{M}} = \langle \mathrm{grad}(f),\ V\rangle_{T(\mathcal{M})}\,, \\ \int_{\mathcal{M}} f\mathrm{div}(V)\mathrm{d}S = \int_{\mathcal{M}} g_x(\mathrm{grad}(f), V)\mathrm{d}S = \int_{\mathcal{M}} \mathrm{grad}(f)^\top \mathbf{g}_x V \mathrm{d}S\,. \end{gathered} \tag{iii}$$

Here, $\mathrm{d}S = \sqrt{\det \mathbf{g}}\mathrm{d}x$ is the surface differential. By recalling that $\mathrm{grad}(f) = \mathbf{g}_p^{-1}\nabla\tilde{f}$, that $\mathbf{g}_x$ is symmetric and that Equation (iii) must hold for all f, an integration by parts leaves us with

$$\mathrm{div}(V) = -\frac{1}{\sqrt{\det \mathbf{g}_x}}\nabla\cdot(\sqrt{\det \mathbf{g}_x}V)\,, \tag{iv}$$

where $\nabla\cdot$ is the divergence operator in Euclidean space.

Laplace-Beltrami Operator Generalizing the Laplacian operator to surfaces comes straightforwardly from its definition as the divergence of the gradient. Given some scalar field $f : \mathcal{M} \to \mathbb{R}$, the Laplace-Beltrami operator on f is defined by means of Equations (ii) and (iv) as

$$\Delta f = \operatorname{div}(\operatorname{grad}(f)) = -\frac{1}{\sqrt{\det \mathbf{g}_x}} \nabla \cdot (\sqrt{\det \mathbf{g}_x} \mathbf{g}_x^{-1} \nabla \tilde{f}) \,, \tag{v}$$

being $\tilde{f} = f \circ \varphi^{-1}$ the local parametrization of f.

The Laplacian is a linear operator from the space $\mathcal{F}(\mathcal{M}, \mathbb{R})$ of scalar fields over $\mathcal{M}$ into itself, and hence it does make sense searching for eigenvalues and eigenvectors. Namely, we want to find all the non-trivial solutions ϕ to the problem

$$\begin{cases} \Delta\phi(x) = \lambda_\phi \phi(x) \\ \phi(\partial\mathcal{M}) = 0 \end{cases} \tag{vi}$$

where λ_ϕ is a scalar constant depending only on ϕ and $\partial\mathcal{M}$ denotes the boundary of the manifold $\mathcal{M}$. This specific eigenproblem imposes Dirichlet boundary conditions, constraining the function to be zero-valued at the boundary, if there is one. There are other possibilities for constraining ϕ, but in this thesis we will only refer to the Dirichlet eigenproblem.

For an extensive dissertation on differential geometry and specral properties of differential operators on manifolds we remand to the books from Morita and Chavel about this topic [184, 50, 51]. Here we summarize the properties we will mostly use throughout this thesis:

- the solution to the Laplacian eigenproblem is a countably infinite family (ϕ_k, λ_k), where ϕ_k is a real function and λ_k is a non-negative real value (thus forming a non-decreasing sequence);
- the Laplacian eigenfunctions forms an orthogonal basis for the set of square integrable scalar fields on $\mathcal{M}$;
- if $\mathcal{M}$ has no boundary, there is exactly a constant eigenfunction ϕ_0, associated with a null eigenvalue $\lambda_0 = 0$;
- if $\mathcal{M}$ is composed by multiple connected components $\mathcal{M}_1, \cdots, \mathcal{M}_h$, the solution to the eigenproblem on $\mathcal{M}$ is the union of the solutions to the eigenproblems on every component $\mathcal{M}_i$.

The Laplace-Beltrami eigenfunctions are a generalization to manifold domains of the Fourier basis in Euclidean space, meaning that the Laplacian eigenvalues generalizes the notion of base frequencies. When studying functions on Riemannian surfaces, an important quantity that encodes information about frequency is the *Dirichlet energy.*

Definition 7. *Given a Riemannian manifold* $(\mathcal{M}, g)$ *and a real function* $f : \mathcal{M} \to \mathbb{R}$ *on* $\mathcal{M}$*, said* $u = {}^{f}/_{\|f\|_{\mathcal{M}}}$ *the normalized version of* f*, the* Dirichlet energy *of* f *is*

$$\mathcal{E}\,(f) = \langle \operatorname{grad}(u),\ \operatorname{grad}(u) \rangle_{T(\mathcal{M})} = \langle u,\ \Delta u \rangle_{\mathcal{M}}\ . \tag{vii}$$

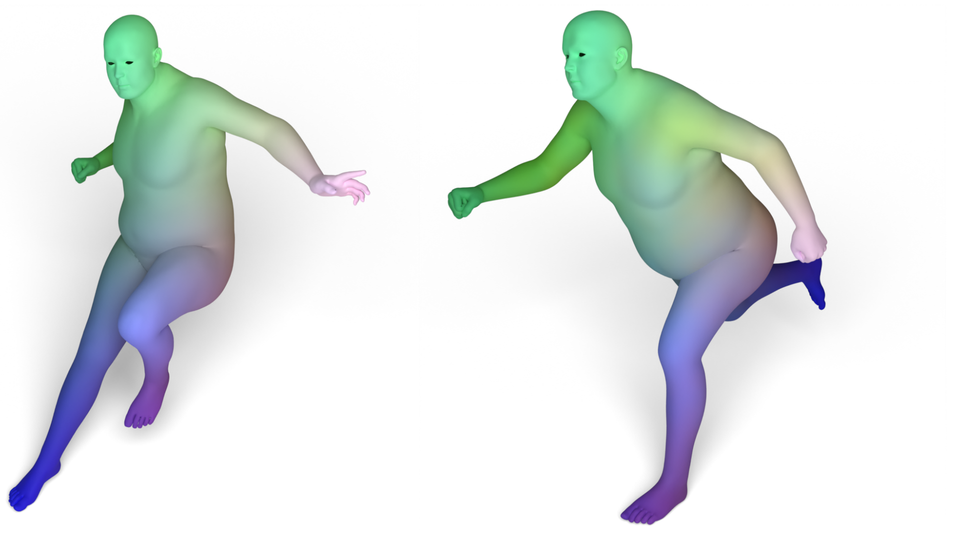


Figure iii. Example of isometric correspondence. Pairs of corresponding points are associated by the same color.

Correspondences and Functional Maps

Given two Riemannian manifolds $(\mathcal{M}, g_{\mathcal{M}})$ and $(\mathcal{N}, g_{\mathcal{N}})$ with induced geodesic distances, respectively, $\delta_{\mathcal{M}}$ and $\delta_{\mathcal{N}}$, we can relate $\mathcal{M}$ and $\mathcal{N}$ by defining a *correspondence*, which is a bijective function $c : \mathcal{M} \rightarrow \mathcal{N}$. Clearly, there exist infinitely many correspondences between the two manifolds, but we are usually interested in a particular type of correspondence, which we call *isometric*.

Definition 8. *Let $\mathcal{M}$ and $\mathcal{N}$ be Riemannian manifolds, respectively equipped with metrics $g_{\mathcal{M}}, g_{\mathcal{N}}$ and induced geodesic distances $\delta_{\mathcal{M}}, \delta_{\mathcal{N}}$, and let $c\mathcal{M} \rightarrow \mathcal{N}$ be a correspondence. c is said* isometric correspondence *(or* isometry*) if it holds*

$$\forall x, y \in \mathcal{M}, \quad \delta_{\mathcal{M}}(x, y) = \delta_{\mathcal{N}}(c(x), c(y))\,. \tag{viii}$$

If there exists an isometric correspondence, then $\mathcal{M}$ and $\mathcal{N}$ are said isometric manifolds.

Intuitively, an isometric correspondence encodes the idea of having the same shape in different poses, as in the example depicted in Figure iii. Sometimes, we are interested to be invariant to the scale of the manifolds, thus we can relax the condition by saying that there exists some constant α such that

$$\forall x, y \in \mathcal{M}, \quad \delta_{\mathcal{M}}(x, y) = \alpha\delta_{\mathcal{N}}(c(x), c(y))\,. \tag{ix}$$

While we generally refer to this as an isometry, it is sometimes useful to distinguish it from strict isometries and refer to it as α-isometry.

It is worth to mention that it is not always guaranteed that an isometry does exist. If that is the case, we could be anyway interested in finding some kind of meaningful correspondence, by relaxing the condition from Equation (ix) and asking

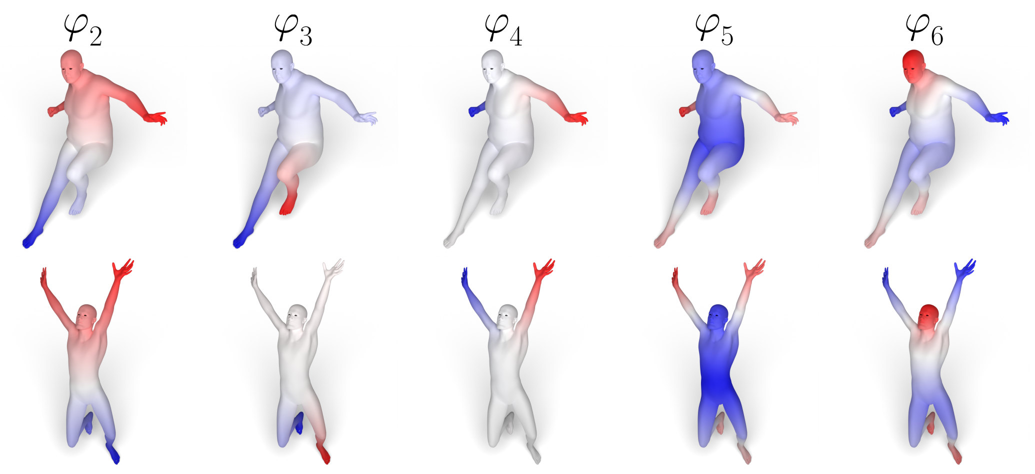


Figure iv. Two non-isometric, but semantically similar shapes and their Laplacian spectra.

for an approximated isometry or for a correspondence that minimizes the overall error

$$\int_{\mathcal{M}}\int_{\mathcal{M}} (\delta_{\mathcal{M}}(x,y) - \delta_{\mathcal{N}}(c(x),c(y)))^2 \mathrm{d}x\mathrm{d}y\,. \tag{x}$$

Since the notion of distance is induced by the metric tensor, which also determines the Laplacian, two isometric manifolds have the same Laplacian eigenvalues. Specifically, said (ϕ_k, λ_k) the family of solutions to the Laplacian eigenproblem on a manifold $\mathcal{M}$ and (ψ_k, μ_k) their analogue on a manifold $\mathcal{N}$, if $\mathcal{M}$ and $\mathcal{N}$ are α-isometric under a correspondence $c : \mathcal{M} \to \mathcal{N}$, then

$$\begin{aligned} \forall x \in \mathcal{M}, k \in \mathbb{N}, \quad \phi_k(x) = \psi_k(c(x))\,, \\ \lambda_k = \alpha^2 \mu_k\,. \end{aligned} \tag{xi}$$

The example in Figure iv shows that the Laplacian spectrum is robust even when the isometry does not exist. If the shapes are similar enough (*e.g.*, two humans, in the example), the Laplacian eigenfunctions concentrate their energy on areas that have the same semantics.

This relation between isometries and Laplacian leads to the idea of *functional maps* [202]. Given a correspondence $c : \mathcal{M} \to \mathcal{N}$, a functional map is a linear operator $T_F : L^2(\mathcal{M}) \to L^2(\mathcal{N})$ that maps functions on $\mathcal{M}$ to functions on $\mathcal{N}$, defined via the composition $T_F(f) = f \circ c$. Given the Laplacian eigenbases $\{\phi_k\}$ for $L^2(\mathcal{M})$ and $\{\psi_k\}$ for $L^2(\mathcal{N})$ (this works for any choice of bases, and it is not restricted to the Laplacian eigenfunctions), we can represent the mapping of any eigenfunction ϕ_i to $\mathcal{N}$ as

$$T_F(\phi_i) = \sum_j \underbrace{\langle \phi_i \circ c,\ \psi_j \rangle_{\mathcal{N}}}_{C_{i,j}} \psi_j\,. \tag{xii}$$

Since any function $f \in L^2(\mathcal{M})$ can be expressed as a linear combination of $\{\phi_k\}$, and by recalling that T_F is linear, the mapping of f onto $\mathcal{N}$ can be computed as

$$T_F(f) = \sum_i \langle \phi_i,\ f \rangle_{\mathcal{M}} T_F(\phi_i) = \sum_{i,j} \langle \phi_i,\ f \rangle_{\mathcal{M}} C_{i,j} \psi_j\,, \tag{xiii}$$

and since the delta indicator functions (*i.e.*, Dirac deltas)

$$I_p(x) = \begin{cases} \infty & x = p \\ 0 & x \neq p \end{cases} \tag{xiv}$$

belong to $L^2(\mathcal{M})$, we can retrieve the correspondence from the functional map.

By encoding the coefficients in a matrix $\mathbf{C}$, we can then reduce the problem of finding a correspondence from a continuous setting to a discrete setting. Moreover, we can approximate the mapping by truncating the bases to some index N, meaning that the problem of finding an approximated correspondence is finite.

Discretization

When working with manifolds in a discrete setting, like it's usually needed for computational applications, there are different choices to represent shapes. Throughout this thesis, we will stick to the standard triangular mesh representation.

Triangle Meshes A triangle mesh $\hat{\mathcal{M}} = (V, E, T)$ is a triple where:

- $V \subset \mathbb{R}^3$ is a set of vertices in 3-dimensional space;
- E is a set of unordered edges between vertices in V;
- T is a set of oriented triangles formed by the edges in E.

Usually, we indicize the vertices and represent the edges (reps. triangles) as pairs (res. triplets) of indices. However, it is sometimes convenient to represent them mathematically as pairs (resp. triplets) of actual vertices.

The ordering of the vertices in a triangle t is invariant under even permutations (*i.e.*, we don't care about the ordering as long as we just cycle the indices), and the ordering defines an outward normal vector $\hat{\boldsymbol{n}}_t$ according to the right-hand rule. The side of t facing the direction of $\hat{\boldsymbol{n}}_t$ is called *outer*, while the opposite side is called *inner*.

We assume the mesh structure to be consistent with the continuous definition of manifolds. We define a mesh to be a *manifold mesh* if the following holds:

- triangles and edges must not be degenerate (*i.e.*, no null-length edges and no null-area triangles);
- edges must be incident on exactly two triangles;
- a vertex can only be surrounded by a single fan of triangles, which must be closed.

Sometimes we allow manifolds and meshes to have boundaries, leading to a relaxation of some conditions. A vertex lying on the boundary has its fan of triangles open, whereas a boundary edge is incident to exactly one triangle.

Some manifolds, like the Möbius strip, are said to be non-orientable. These manifolds have only one side, meaning that chiral figures can be moved around on the surfaces and be brought back to its original position, but in the form of its mirror image. Throughout this thesis we require our manifolds (and hence our manifold meshes) to be orientable. To enforce this property, we ask for adjacent

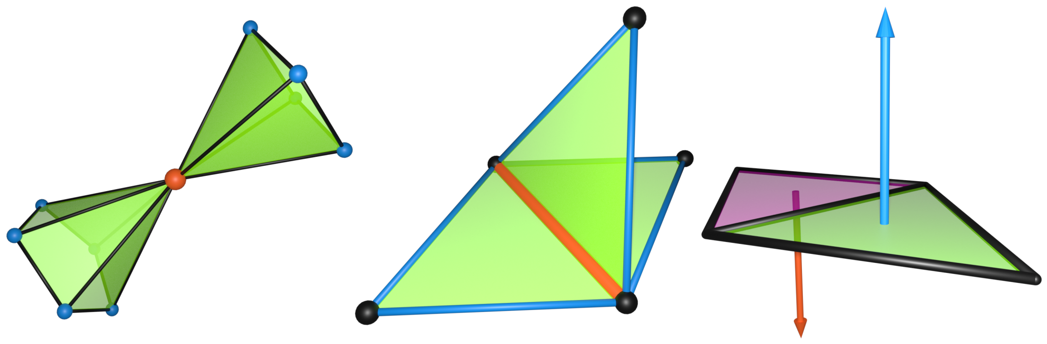


Figure v. Examples of non-manifold and non-orientable meshes. Left: the orange vertex is non-manifold because it is surrounded by two fans of triangles. Middle: The orange edge is non-manifold because it is incident on three faces. Right: the surface is non-orientable, because the two faces have normals pointing in opposite directions.

faces to have their normals pointing in compatible directions. Alternatively, this property can be stated as asking for edge incident on two faces to be traversed in opposite directions.

Figure v provides visual examples of non-manifold meshes with multiple triangle fans on the same vertex and edges incident on three triangles, as well as an example of non-orientable triangle mesh.

Differential Geometry We represents a function $f : \mathcal{M} \to \mathbb{R}$ as a vector $\boldsymbol{f} \in \mathbb{R}^{|V|}$ assuming real values on vertices. Vector fields $V : T(\mathcal{M}) \to \mathbb{R}^3$ can be represented either as vectors $\boldsymbol{v} \in \mathbb{R}^{|E|}$ assuming real values on edges or as vectors $\boldsymbol{v} \in \mathbb{R}^{|T| \times 3}$ assuming real vector values on triangles. Finally, we represent the Laplace-Beltrami operator as a $|V| \times |V|$ real valued sparse matrix $\mathbf{L} = \mathbf{A}^{-1}\mathbf{W}$, where $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$ contains the area elements of the vertices and $\mathbf{W} \in \mathbb{R}^{|V| \times |V|}$ is defined according to the local geometry [213]. The area matrix is also useful for computing the inner product between functions on the surface. Given any two functions $f, g : \mathcal{M} \to \mathbb{R}$ and their corresponding discrete vectors $\boldsymbol{f}, \boldsymbol{g} \in \mathbb{R}^{|V|}$, the weighted inner product $\boldsymbol{f}^\top \mathbf{A} \boldsymbol{g}$ realizes

$$\langle f,\ g \rangle_{\mathcal{M}} = \int_{\mathcal{M}} f(x) g(x) \mathrm{d}x \,. \tag{xv}$$

By realizing that $g(x) = 1$ is discretized as the vector $\mathbf{1} \in \mathbb{R}^{|V|}$, the computation of the integral of some function f over $\mathcal{M}$ can be computed as the inner product $\langle f,\ 1 \rangle_{\mathcal{M}}$, which reduces to the computation of the weighted inner product $\boldsymbol{f}^\top \mathbf{A} \mathbf{1}$.

For further details on the discretization of differential operators on surfaces, we refer to the Ph.D. dissertation from Hirani on discrete exterior calculus [103].

2D Parametrization Finally, we introduce the notion of discrete parametrization, which represents the discretized atlas of a manifold $\mathcal{M}$. Each triangle t represents a finite flat portion of the surface, meaning we can always find an affine transformation $\mathbf{J}_t$ that maps the 3D triangle into a 2-dimensional space. The transformation $\mathbf{J}_t$ effectively represents the discrete Jacobian matrix of the local chart,

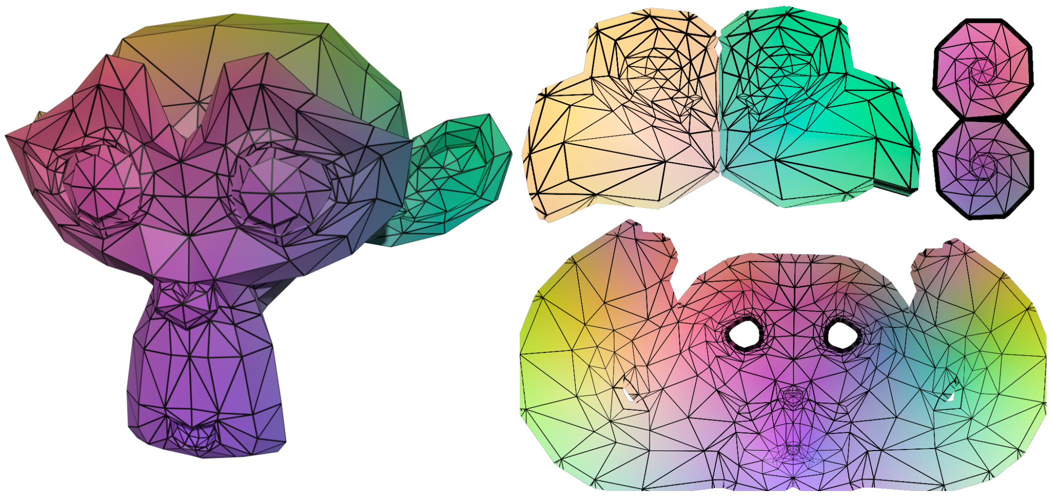


Figure vi. Example of parametrization of a triangle mesh. The 3D coordinates of the model are color-coded to better show the mapping to the unit square.

hence inducing a discrete metric $\mathbf{g}_t = \mathbf{J}_t^\top \mathbf{J}_t$. Depending on the applications, the local charts could map every triangle to the canonical triangle (*i.e.*, the triangle having vertices at the origin, at the point $(1, 0)$, and at the point $(0, 1)$), or could define a piecewise affine transformation of the entire mesh into some bounded region of $\mathbb{R}^2$, trying to minimize the distortion, the discontinuities and the number of connected components. The example depicted in Figure vi shows a parametrization of a triangular mesh to a square. The mesh is split into five connected components, which are cut and flatten onto the plane. The color-coding of the coordinates ease the parsing of the parametrization, showing where each triangle of the 3D mesh is mapped in the plane.

Part I

Spectral Geometry

Chapter 1

Orthogonalized Fourier Polynomials for Signal Approximation and Transfer

In this chapter, we propose a novel approach for the approximation and transfer of signals across 3D shapes. The proposed solution is based on taking pointwise polynomials of the Fourier-like Laplacian eigenbasis, which provides a compact and expressive representation for general signals defined on the surface. Key to our approach is the construction of a new orthonormal basis upon the set of these linearly dependent polynomials. We analyze the properties of this representation, and further provide a complete analysis of the involved parameters. Our technique results in accurate approximation and transfer of various families of signals between near-isometric and non-isometric shapes, even under poor initialization. Our experiments, showcased on a selection of downstream tasks such as filtering and detail transfer, show that our method is more robust to discretization artifacts, deformation and noise as compared to alternative approaches. The work presented in this chapter has been realized in collaboration with prof. Simone Melzi, which largely helped in the implementation and evaluation of the detail transfer and the spectral filter applications, the design of the experimental section, and the overall presentation of the text, together with prof. Emanuele Rodolà, prof. Maksim Ovsjanikov and prof. Michael Bronstein. The results presented in this chapter have been published in the proceedings of the *Annual Conference of the European Association for Computer Graphics* (*EUROGRAPHICS*) [161].

1.1 Introduction and Related Work

Approximation and transfer of signals between shapes are among the most widely explored tasks in computer vision and graphics, and are at the basis of numerous applications. Common to most approaches is the idea to encode the given surface signal in a basis that allows to represent and transfer it efficiently; among these, approaches based upon the construction of a Fourier-like basis (or rather its surface counterpart [265]) play the lion's share [134, 240, 272]. The key idea is to project the signal onto a low-dimensional function space, e.g., corresponding to the lowest

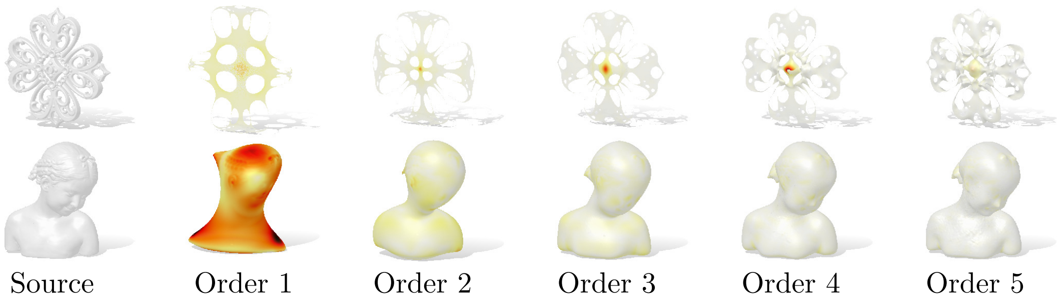


Figure 1.1. Surface approximation using orthogonalized polynomials of increasing order, where order 1 corresponds to the plain Laplacian eigenbasis. All of these approximations are obtained starting from just 5 eigenfunctions. Reconstruction error is encoded by color, growing from white to dark red.

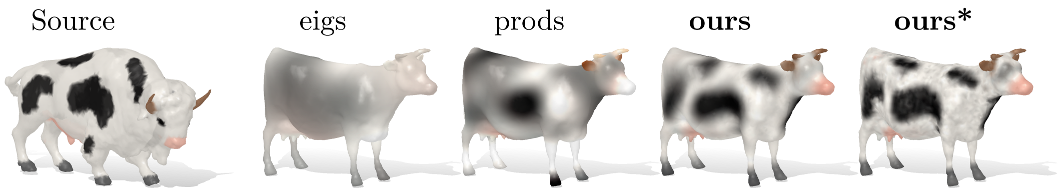


Figure 1.2. RGB signal transfer among two non-isometric shapes with different mesh topology. Our approach (*ours*, *ours**) better transfers the surface signal from bison to cow, while using the same amount of information as other existing approaches (*eigs* [202] and *prods* [193]). We refer to the experimental section for more details.

portion of the frequency band. This yields a well known trade-off between the compactness of the representation and its ability to capture higher frequencies of the signal. Efforts have been devoted to strike a balance between these two factors, by resorting to alternative bases or via costly post-processing steps; still, the search for a *compact* basis for representing high level of detail, is an unsolved problem to date.

A closely related problem to that of signal representation is the need to *transfer* these signals from a source to a target domain. This can often be cast as correspondence problem, where the objective is to find a transformation that acts as a bridge between source and target. This was shown in [202] to be equivalent to seeking a coherent set of basis functions for the given pair of shapes; the search for a correspondence is then phrased as the search for a linear map (called functional map) that aligns the basis functions on the source to those on the target. Follow-up works have embraced this view by introducing more stable ways to compute the functional map [194, 203, 83, 227], by extending the framework to the partial setting [233, 61], or by constructing new coherent bases explicitly as linear transformations of the Laplacian eigenfunctions [127, 17, 144].

The choice of Laplacian eigenfunctions as a reduced basis for representing surface signals is due to their optimality for continuous functions with bounded variation [5]. However, in many real applications such as texture transfer and shape interpolation, this band-limited representation may not provide the necessary accuracy for capturing fine details. To overcome these limitations, two main solutions have been proposed: (i) to design an ad-hoc basis for fixed sets of signals; (ii) to define algorithms for recovering the residual information that is lost in the representation. The former includes wavelets as a localized alternative to the Fourier basis [302, 55, 153, 100, 209, 122]. Other local constructions include those based on sparse regularization and Hamiltonian operators with step potentials [189, 128, 54, 178]. Specialized bases for piecewise-constant signals and vertex coordinates have been proposed in [172] and [173] respectively, but these do not generalize well to different function classes. Point (ii) is a more recent trend [176, 78]. The idea is to iteratively seek for bases of increasing dimension starting from an initial alignment between few Laplacian eigenfunctions. The iterative procedure preserves the alignment of the two bases as they increase in dimension, and sidesteps the need for further optimization to get an optimal alignment.

More closely related to ours is the work of Nogneng et al. [193], where the authors consider the set of pointwise products of the Laplacian eigenfunctions, in addition to the eigenfunctions alone, for representing surface signals more accurately. The main property of these *eigenproducts* is that their alignment can be explicitly and directly derived from the functional map between the standard eigenfunctions; this way, a correct alignment between a few eigenfunctions is automatically extended to the larger set, which includes their products.

Contribution. Our work addresses a key issue of the latter representation, namely that the set containing Laplacian eigenfunctions and eigenproducts is *not* linearly independent in general; as we show in the sequel, it is linearly independent only when very few eigenfunctions are involved. Thus, this set does not provide a unique representation for surface signals. Further, the linear dependence gives rise to instability in the transfer task, which must be handled through additional constraints and pre-processing as shown in [193]; see Figure 1.2 for an example. Here we follow a similar idea and use eigenproducts to increase the dimensionality of the basis, and in turn, the quality of the resulting representation. However, differently from [193], we do not limit our analysis to products of order 2, but we effectively exploit the entire set of "Fourier polynomials" with arbitrary order.

The work presented in this chapter fills the gaps left by [193] in several ways:

- For the first time, we provide a theoretical analysis on the space spanned by the eigenproducts, including a discussion on the frequency range that they capture;
- We propose the construction of an orthonormal basis on top of the linearly *dependent* set of polynomials, yielding a simpler, more accurate, stable and computationally efficient technique;
- We extend the discussion and empirical evaluation to eigenproducts of order greater than 2.

Our basis applies to several applications, such as detail transfer and spectral filtering, that are impossible to target through the representation proposed in [193] as we show in the experiments.

Figure 1.3. Shapes used for our theoretical results: human ($\sim$ 7k vertices), bunny ($\sim$ 10k), cat ($\sim$ 10k), and donut ($\sim$ 20k).

1.2 Background

We model a shape as a 2-dimensional Riemannian manifolds $\mathcal{M}$, equipped with the metric tensor g.

The Laplace-Beltrami operator obeys the Leibniz product rule with a correction term involving gradients [51]. Namely,

$$\Delta f(x)g(x) = f(x)\Delta g(x) + g(x)\Delta f(x) - 2\left\langle \nabla f(x),\ \nabla g(x)\right\rangle \,, \tag{1.1}$$

which, in the case of eigenfunctions, leads to[1]:

$$\Delta \phi_i(x)\phi_j(x) = (\lambda_i + \lambda_j)\phi_i(x)\phi_j(x) - 2\left\langle \nabla \phi_i(x),\ \nabla \phi_j(x)\right\rangle \,. \tag{1.2}$$

A rescaled version of the correction term was empirically used in [249] as a descriptor field for shape matching.

To properly introduce the mathematical background needed in this chapter, we formalize the notion of *eigenproduct*.

Definition 1.1. *Let $I = \{i_1, \cdots, i_n\} \in \mathbb{N}$ be a finite set of indices, possibly containing repeated elements. We define the* eigenproduct *$\phi_I : \mathcal{M} \to \mathbb{R}$ to be the scalar function defined as*

$$\phi_I(x) = \prod_{i \in I} \phi_i(x)\,, \tag{1.3}$$

where the multiplication is to be taken pointwise. A special case of eigenproduct is when $I = \{i, \cdots, i\}$ is a set containing n times the same index. In this case, we define the function $\phi_I = \phi_i^n$ an eigenpower*. Finally, an N-th order* Fourier polynomial *of K eigenfunctions is a linear combination of eigenproducts up to order N involving the first K eigenfunctions (excluding the constant one).*

1.3 Theoretical Results

In this section, we present some theoretical results on eigenproducts, together with some interesting implications of their properties. We first examine their frequency distribution, and compare it to the frequency distribution of the eigenfunctions (i.e. their associated eigenvalues). We then discuss a result on the approximation of eigenproducts in the space spanned by the eigenfunctions, providing possible interpretations and implications. We use different shapes to show the generality of our results (see Figure 1.3).

[1]Strictly speaking, functions in $L^2(\mathcal{M})$ do not admit a pointwise product; we keep the notation for the sake of simplicity, with the understanding that it remains valid in the proper Sobolev space.

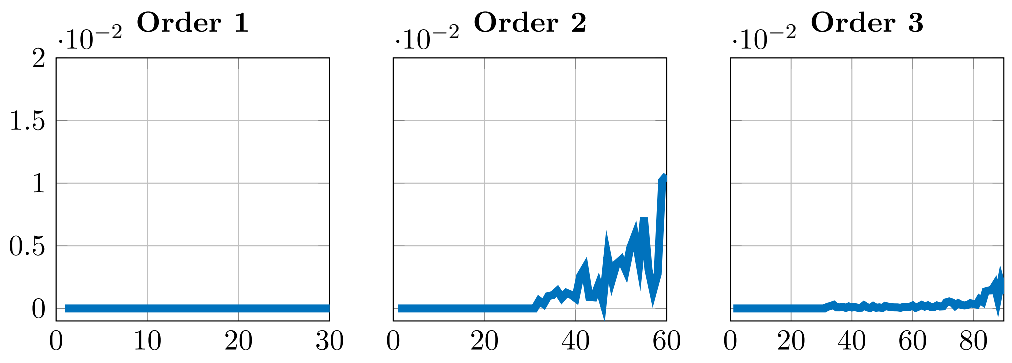


Figure 1.4. Each plot shows the relative error $\min\{(\lambda_i - \mathcal{E}(\phi_I))/\lambda_i\}$ between the i-th Laplacian eigenvalue (where i ranges on the x axis) and the closest frequency of a N-th order eigenproduct. Here NK Laplacian eigenvalues are considered, with $K = 30$ and product order $N = \{1, 2, 3\}$ (left to right). At order 1 the products correspond to the standard eigenfunctions, hence yielding exactly zero error. At increasing order the error stays close to zero, showing that each eigenvalue is approximated by the frequency of a product with $> 99.9\%$ accuracy. Results are averaged on four shapes.

1.3.1 Frequency Distribution

We now present a result about the distribution of frequencies (i.e. the Dirichlet energies) of eigenproducts. Proofs are grouped in Section 1.6.

Theorem 1.1. *Let $\mathcal{M}$ be a d-dimensional Riemannian manifold, and Δ be the associated Laplace-Beltrami operator. Then, let I be a set of indices of size N and let ϕ_I be an eigenproduct. The following relation holds:*

$$\mathcal{E}(\phi_I) \geq \frac{1}{2}\sum_{i\in I}\lambda_i \,. \tag{1.4}$$

Furthermore, in the special case of an eigenpower $\phi_I = \phi_i^N$, it holds

$$\mathcal{E}\left(\phi_i^N\right) = \frac{N^2}{2N-1}\lambda_i \,. \tag{1.5}$$

Corollary 1.1. *Let $\mathcal{M}$ be a 2-dimensional manifold, and Δ be the associated Laplace-Beltrami operator. Fixed $N, K \in \mathbb{N}$, and being $\tilde{\mathbf{\Phi}}$ the set of N-th order eigenproducts between the first K Laplacian eigenfunctions, $\max_{\phi_I \in \tilde{\mathbf{\Phi}}}\{\mathcal{E}(\phi_I)\} \in \Omega(NK)$.*

From Theorem 1.1 and Corollary 1.1, we can deduce the following. Since, by Weyl's law, for 2-dimensional manifolds the (NK)-th eigenvalue is $\Theta(NK)$, then using the Dirichlet energies of N-th order products between K eigenfunctions allows to express the same frequencies as if we use NK eigenfunctions. Further, within the band of the first NK eigenvalues, each eigenvalue is matched to high accuracy by the Dirichlet energy of an eigenproduct, as we empirically demonstrate in Figure 1.4.

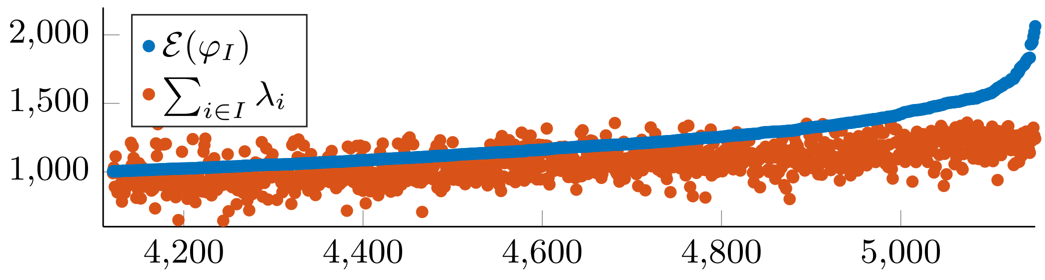
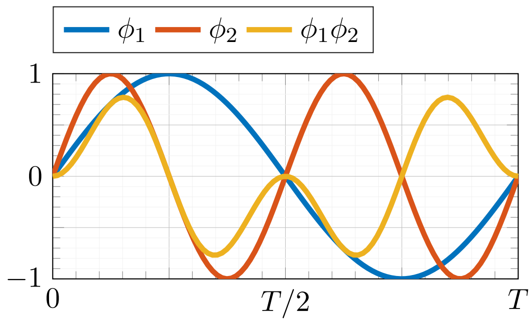


Figure 1.5. Dirichlet energies of order-2 products against the sum of the eigenvalues. The former grow much faster in the high portion of the spectrum.

Intuitively, this means that the eigenproducts up to order N have the same expressive power as the first NK eigenfunctions. Therefore, eigenproducts can be used to represent well band-limited functions within the band of the first NK eigenvalues.

Example. On the real line $\mathcal{M} = [0, T]$, consider the second-order eigenproduct $\tilde{\phi}(x) = \sin(2\pi x/T)\sin(4\pi x/T)$. Its Dirichlet energy is $\mathcal{E}\left(\tilde{\phi}\right) = 1/\|\tilde{\phi}\|^2 \int_0^T (\partial/\partial x\tilde{\phi}(x))^2 \mathrm{d}x$, resulting in $\mathcal{E}\left(\tilde{\phi}\right) = (1/\|\tilde{\phi}\|^2)5\pi^2/T$. Since the squared norm of $\tilde{\phi}$ is $T/4$, we get $\mathcal{E}\left(\tilde{\phi}\right) = 20\pi^2/T^2 = 4\pi^2/T^2 + 16\pi^2/T^2 = \lambda_1 + \lambda_2$. See the inset figure for an illustration.



Outside of the band of the first NK eigenvalues, the Dirichlet energy of eigenproducts grows more rapidly than the sum of the eigenvalues; we illustrate this behavior in Figure 1.5. In fact, the Dirichlet energy of an eigenproduct ϕ_I can be expressed as (see the proof of Theorem 1.1 for the derivations):

$$\mathcal{E}\left(\phi_I\right) = \frac{1}{2}\sum_{i\in I}\lambda_i + \sum_{i\in I}\int_{\mathcal{M}} \| \prod_{\substack{j\in I\\ j\neq i}} \phi_j(x)\nabla\phi_i(x)\|^2 \mathrm{d}x\,. \tag{1.6}$$

From Equation (1.6), we can see how as we add more factors to the eigenproduct, the number of terms grows and, since these terms are all non-negative, the whole energy is increased.

1.3.2 Approximating Eigenproducts

We now investigate the following question, and draw some interesting conclusions that were missing in previous work: Can a given eigenproduct be represented well in the standard eigenbasis?

Let $\mathcal{M}$ be a 2-dimensional manifold and $\{\phi_i\}$ be the set of its Laplacian eigenfunctions. For any function f, let $E_v(f)$ be the projection of f onto the first v

Laplacian eigenfunctions and let $R_\upsilon(f)$ be the L^2 norm of $f - E_\upsilon(f)$. Namely, the residual:

$$R_\upsilon(f) = \| f - \sum_{i=0}^{\upsilon} \langle \phi_i,\ f \rangle_{\mathcal{M}} \, \phi_i \|_{L^2} \,. \tag{1.7}$$

Aflalo et al. [5] proved the upper bound:

$$R_\upsilon^2(f) \leq \frac{\|\nabla f\|^2}{\lambda_{\upsilon+1}} \quad \forall f \,, \tag{1.8}$$

further showing that the bound can not be tightened by any other sequence of linearly independent functions $\{\phi_i\} \in L^2(\mathcal{M})$. This yields the optimality of the Laplacian eigenfunctions for representing any function with bounded gradient magnitude. For the special case of f being an eigenproduct ϕ_I, however, the bound is not very informative. We instead appeal to the following:

Theorem 1.2. *[154] Fixed $K, N \in \mathbb{N}$, with $K \gg 1$, for any set of indices $I = \{i_1, \cdots, i_N\}$, where each $i_j \leq K$, for any $\upsilon > K$ and for any $\chi \in \mathbb{N}$, it holds:*

$$R_\upsilon(\phi_I) \lesssim K^{\frac{N\sigma(2N,2)}{2}} \left(\frac{K}{\upsilon}\right)^{\frac{\chi}{2}} \,, \tag{1.9}$$

where

$$\sigma(p,d) = \max\left\{ \frac{d-1}{2}\left(\frac{1}{2} - \frac{1}{p}\right),\ d\left(\frac{1}{2} - \frac{1}{p}\right) - \frac{1}{2} \right\} . \tag{1.10}$$

This result can be read as follows. For large values of K, the products tend to add less and less information to the spanned space, until, eventually, all the products are spanned by a basis of υ eigenfunctions, for every $\upsilon > K$.

Example. Consider the product $\tilde{\phi}(x) = \sin(2\pi x)\sin(4\pi x)\sin(6\pi x)$ between the first three non-constant eigenfunctions on $\mathcal{M} = [0,1]$. If we represent this product in the basis of the first $\upsilon = 4$ non-constant eigenfunctions, we get the residual $R_4(\tilde{\phi}) = 3/64$; this residual is already quite small, if we consider that we are using only 1 more eigenfunction for representing the product.

Nevertheless, for small values of K and N it is very rare that an eigenproduct can be expressed exactly as a linear combination of eigenfunctions. Extending the basis with eigenproducts introduces a wealth of additional information. Even if they are not all linearly independent, they could expand the dimension of the spanned space by orders of magnitude. In Figure 1.6 we show examples where, using only 20 eigenfunctions and admitting products up to the 4-th order, we generate bases spanning functional spaces with up to 4000 dimensions. This kind of rank analysis was also missing in [193], while the study about the loss of the eigenproducts basis' full-rank property, reported in Figure 1.6, gives a heuristic for selecting parameters N and K. According to our theoretical and experimental analysis, we advocate using $K \leq 50, 15, 10$ for, respectively, $N = 2, 3, 4$, since this parameter setup maximizes the size of the basis with respect to the computed products.

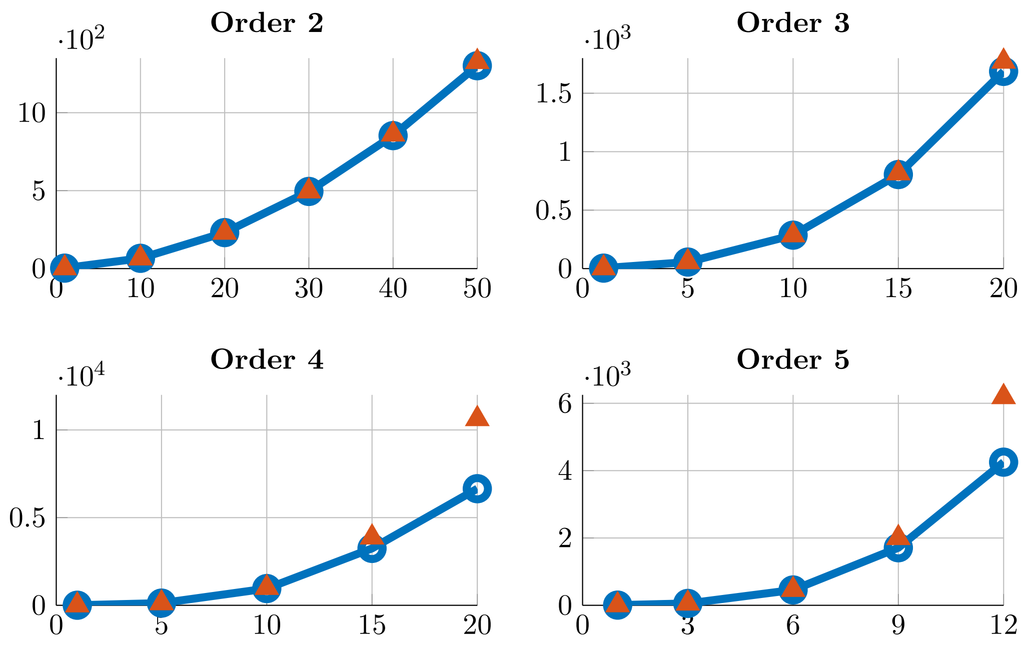


Figure 1.6. Number of linearly independent eigenproducts (in blue) against the number of eigenfunctions involved; each plot is for a different maximum product order. The total number of eigenproducts (in orange) is plotted for reference. These results are averaged on four different shapes.

1.3.3 Orthogonalized Eigenproducts

In the work of Nogneng et al. [193], resorting to eigenproducts for representing surface signals also involved solving an optimization problem prone to numerical instability. Here we propose a much simpler alternative that is less empirical, more stable, and provides consistently better results than [193]. Specifically, consider the set of order-N products of the first K eigenfunctions, spanning a function space $\mathcal{F}(\mathcal{M})$ with some dimension Q. We orthogonalize this set via the Gram-Schmidt (GS) algorithm, and obtain an orthonormal basis for the same space with exactly Q basis functions; see Figures 1.7 and 1.8 for examples. Although straightforward, this process offers advantages both in terms of computational effort and numerical stability.

Complexity. In [193], computing a representation for a given signal in the set of eigenproducts requires computing the SVD decomposition of a $n \times L$ matrix containing all eigenproducts as its columns, where n is the number of vertices and $L = \binom{K+N}{N}$ is the number of N-th order products between the first K eigenfunctions. This SVD decomposition has complexity $\mathcal{O}(n^2L + L^3)$ [93], which for $L \ll n$ (our case) reduces to $\mathcal{O}(n^2L)$. On the other hand, the computational complexity of GS is $\mathcal{O}(nL^2)$, which for $L \ll n$ is much more sustainable than SVD decomposition.

Stability. We demonstrate empirically that our orthogonal basis produces more stable results than [193]. For the latter method, numerical inaccuracies occurring

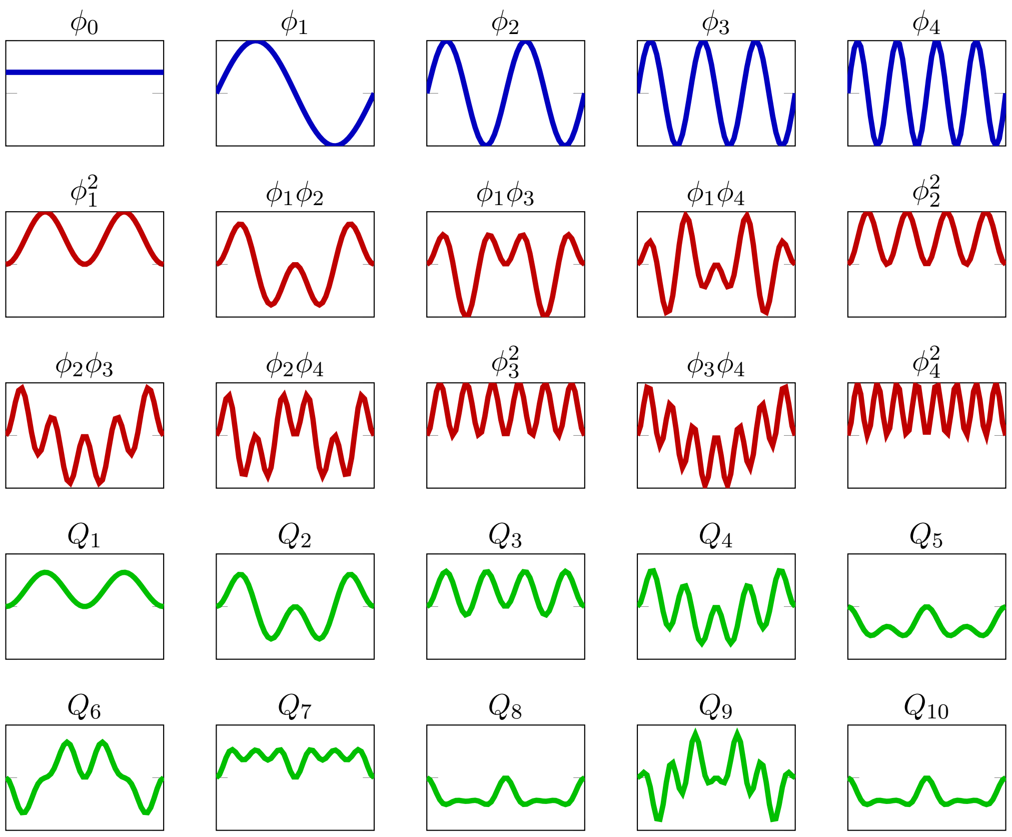


Figure 1.7. Standard Laplacian eigenfunctions (blue), eigenproducts (red, in lexicographic order) and orthonormalized eigenproducts (green) on the real line $[0, T]$.

in the computation of the representation tend to propagate when the function is mapped to a different domain, producing local scale errors; see Figure 1.9 and the experimental section for examples.

Transfer. In a similar spirit as [193], where the authors extend the notion of functional map to eigenproducts, we also provide a way to compute a transfer matrix for our orthogonal bases.

Let us be given a functional map matrix $\mathbf{C}$ between two shapes $\mathcal{M}$ and $\mathcal{N}$, and let us assume a manifold-independent ordering of the eigenproducts (e.g., a lexicographic ordering on the indices of the factors). We then have an ordered set of P eigenproducts $\tilde{\mathbf{\Phi}} = \{\phi_{I_1}, \cdots, \phi_{I_P}\}$ on $\mathcal{M}$, and similarly $\tilde{\mathbf{\Psi}}$ on $\mathcal{N}$. After orthonormalization, we get a new set of functions in the form:

$$\zeta_i(x) = \phi_{I_i}(x) - \sum_{j=1}^{i-1} \langle \phi_{I_i},\ \zeta_j \rangle_{\mathcal{M}} \zeta_j(x)\,. \tag{1.11}$$

Matrix $\mathbf{C}$ maps each eigenfunction ϕ_i on $\mathcal{M}$ to a linear combination $\sum_{j=1}^{K} c_{j,i}\psi_j$ of eigenfunctions on $\mathcal{N}$. Assume for now that $T_F(\mathbf{\Phi}) = \mathbf{\Psi}\mathbf{C}$ is a strict equality. Further, for the sake of simplicity, we limit the exposition to second-order products,

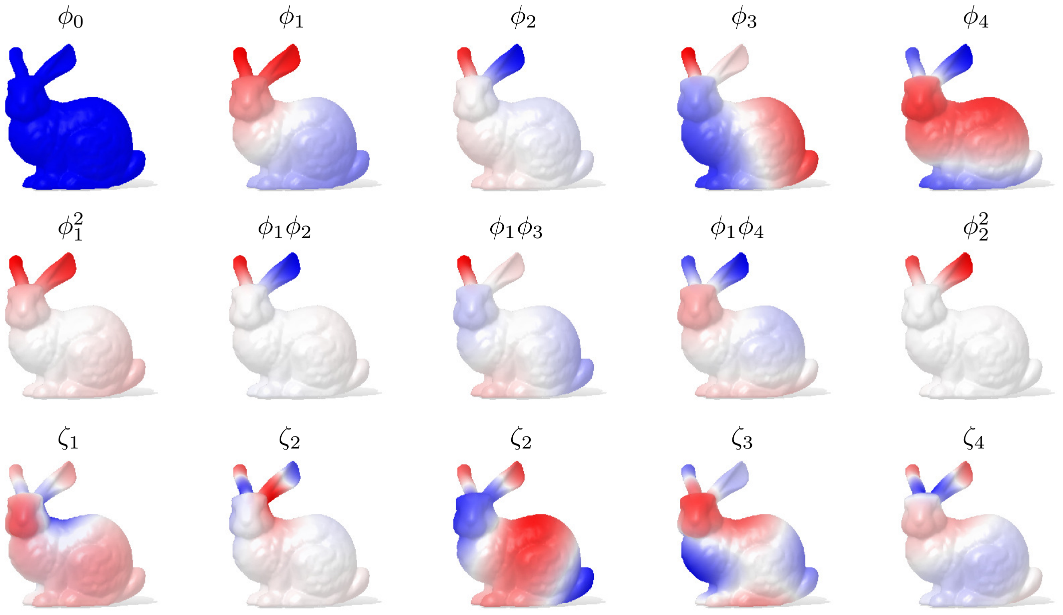


Figure 1.8. Top to bottom: first five Laplacian eigenfunctions, first five eigenproducts, and the orthogonalized eigenproducts.

but the process can be iterated and generalized to any higher order. Each eigenproduct ϕ_I on $\mathcal{M}$ is the product between two eigenfunctions ϕ_i and ϕ_j. When we consider the mapping induced by $\mathbf{C}$, we get:

$$\begin{aligned}\phi_I(x) = \phi_i(x)\phi_j(x) &= \left(\sum_{h=1}^{K} c_{h,i}\psi_h(x)\right)\left(\sum_{p=1}^{K} c_{p,j}\psi_p(x)\right) = \\ &= \sum_{h,p=1}^{K} c_{h,i}c_{p,j}\psi_h(x)\psi_p(x)\,.\end{aligned} \tag{1.12}$$

Thus, the coefficients for transferring products are fully determined by the coefficients for transferring eigenfunctions, as also shown in [193]. Hence, one can compute a matrix $\tilde{\mathbf{C}}(\mathbf{C})$, depending only on $\mathbf{C}$, such that $T_F(\tilde{\mathbf{\Phi}}) = \tilde{\mathbf{\Psi}}\tilde{\mathbf{C}}$.

Going one step further, we extend matrix $\tilde{\mathbf{C}}$ to a new matrix $\mathbf{O}$ that can correctly transfer the orthonormalized basis. We proceed as follows. In general, given a set of vectors $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_n)$ with n elements and rank $1 \le r \le n$, GS produces an orthogonal basis $\mathbf{Q} = (\mathbf{q}_1, \dots, \mathbf{q}_r)$ with r elements, spanning the same space as $\mathbf{B}$. A side-product is the $r \times n$ upper triangular matrix $\mathbf{R} = (r_{i,j})$:

$$\mathbf{R} = (r_{i,j}) = \begin{pmatrix} \langle \mathbf{q}_1,\ \mathbf{b}_1\rangle & \langle \mathbf{q}_1,\ \mathbf{b}_2\rangle & \cdots & \langle \mathbf{q}_1,\ \mathbf{b}_n\rangle \\ 0 & \langle \mathbf{q}_2,\ \mathbf{b}_2\rangle & \cdots & \langle \mathbf{q}_2,\ \mathbf{b}_n\rangle \\ \vdots & \ddots & \cdots & \vdots \\ 0 & 0 & \cdots & \langle \mathbf{q}_r,\ \mathbf{b}_n\rangle \end{pmatrix}, \tag{1.13}$$

such that $\mathbf{QR} = \mathbf{B}$.

Applied to the sets of eigenproducts $\tilde{\mathbf{\Phi}}$ and $\tilde{\mathbf{\Psi}}$, we get the factorizations $\tilde{\mathbf{\Phi}} = \mathbf{Q}_\Phi\mathbf{R}_\Phi$ and $\tilde{\mathbf{\Psi}} = \mathbf{Q}_\Psi\mathbf{R}_\Psi$.

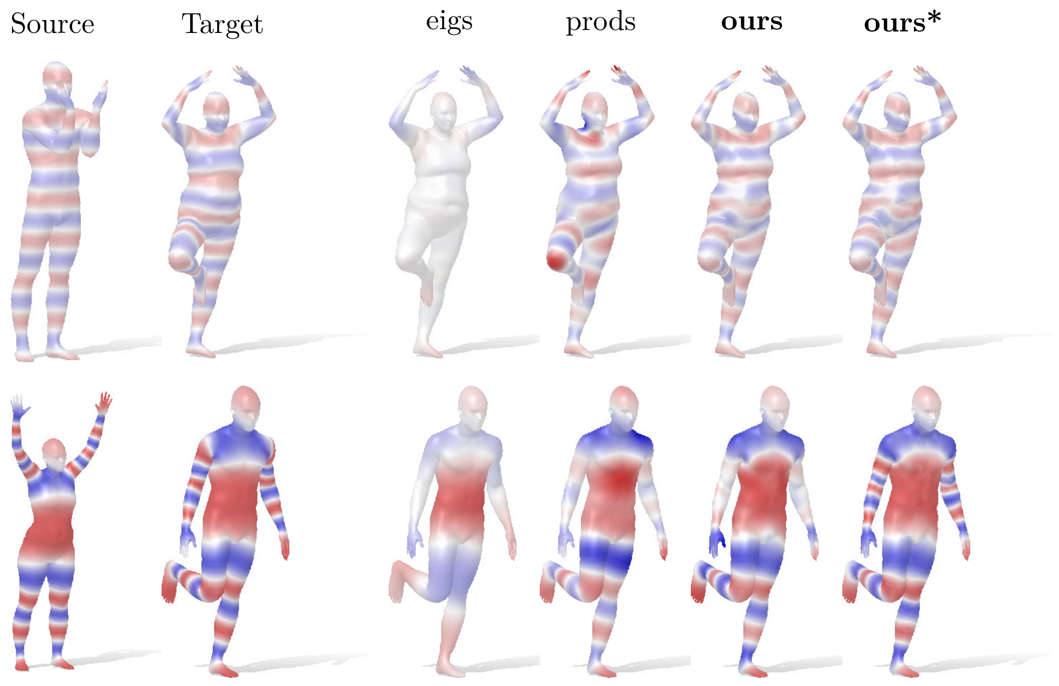


Figure 1.9. Function transfer example on two pairs from the FAUST[30] dataset. The adoption of eigenproducts as in [193] (denoted as *prods*) yields local scale artifacts (knee and neck in the first row) or loss of high frequency details (arms in the second row). With the standard eigenbasis (*eigs*), K eigenfunctions are simply not enough to capture the frequency content of the transferred signal. In the bottom row, the second variant of our method (*ours**) achieves almost perfect reconstruction.

Since $\mathbf{Q}_\Phi$ and $\mathbf{Q}_\Psi$ span the same space as $\tilde{\mathbf{\Phi}}$ and $\tilde{\mathbf{\Psi}}$, assuming there exists a meaningful mapping between $\mathbf{\Phi}$ and $\mathbf{\Psi}$, it makes sense to search for a mapping between $\mathbf{Q}_\Phi$ and $\mathbf{Q}_\Psi$. Thus, we seek for a matrix $\mathbf{O}$ such that:

$$T_F(\mathbf{Q}_\Phi) = \mathbf{Q}_\Psi \mathbf{O} \,. \tag{1.14}$$

By the factorization above, we can equivalently rewrite the mapping $T_F(\tilde{\mathbf{\Phi}}) = \tilde{\mathbf{\Psi}}\tilde{\mathbf{C}}$ as $T_F(\mathbf{Q}_\Phi)\mathbf{R}_\Phi = \mathbf{Q}_\Psi \mathbf{R}_\Psi \tilde{\mathbf{C}}$; plugging in Equation (1.14) and simplifying, we get:

$$\mathbf{O}\mathbf{R}_\Phi = \mathbf{R}_\Psi \tilde{\mathbf{C}} \,. \tag{1.15}$$

Since $\mathbf{R}_\Phi$ is full row-rank, it has a right inverse $\mathbf{R}_\Phi^\top \left(\mathbf{R}_\Phi \mathbf{R}_\Phi^\top\right)^{-1}$. Thus, if we have access to the bases on both manifolds, we can directly compute $\mathbf{O}$ as

$$\mathbf{O} = \mathbf{R}_\Psi \tilde{\mathbf{C}} \mathbf{R}_\Phi^\top \left(\mathbf{R}_\Phi \mathbf{R}_\Phi^\top\right)^{-1} \,. \tag{1.16}$$

For small enough order N and number of eigenfunctions K, the set of eigenproducts is likely to be full-rank (see Figure 1.6 for an empirical assessment); in this case, $\mathbf{R}_\Psi$ is a square invertible matrix, leading to:

$$\mathbf{O} = \mathbf{R}_\Psi \tilde{\mathbf{C}} \mathbf{R}_\Phi^{-1} \,, \tag{1.17}$$

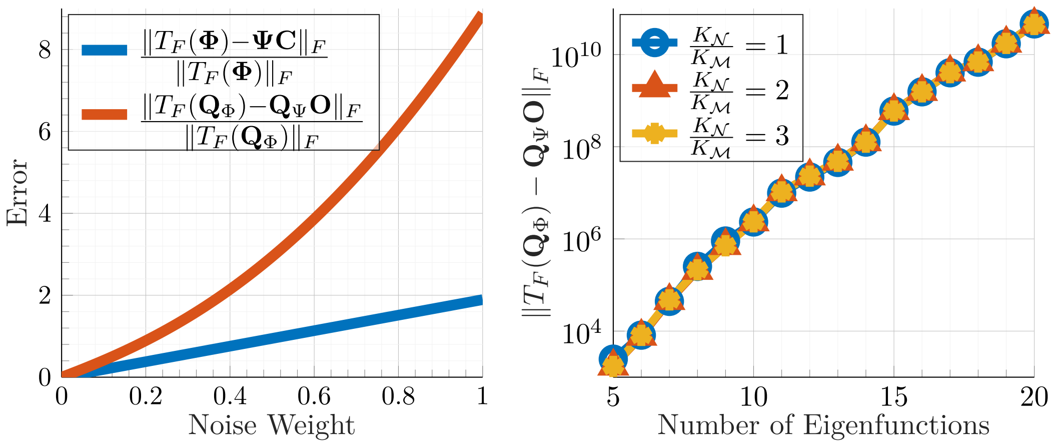


Figure 1.10. Left: Error generated by Equation (1.16) (red curve) at increasing random noise on the matrix $\mathbf{C}$. We show the error for the functional map as a reference (blue curve). Right: Error induced by the analytic expression of $\mathbf{O}$ in transferring the orthogonal basis onto the target shape at increasing number of eigenfunctions. Each curve represents a different ratio between the number $K_\mathcal{N}$ of eigenfunctions on the target and the number $K_\mathcal{M}$ on the source.

and since $\mathbf{R}_\Psi$ is upper triangular, its inverse can be computed efficiently.

We also derive the following alternative formula to compute iteratively the columns of $\mathbf{O}$ depending only on the matrices $\tilde{\mathbf{C}}$, $\mathbf{R}_\Phi$, and $\mathbf{R}_\Psi$:

$$T_F(\zeta_i) = \frac{1}{(\mathbf{R}_\Phi)_{i,i}} \Big(\sum_j \underbrace{\big(\sum_{l=1} (\mathbf{R}_\Psi)_{j,l} \tilde{c}_{l,i} - \sum_{h=1}^{i-1} (\mathbf{R}_\Phi)_{i,h} o_{j,h}\big)}_{o_{j,i}} \xi_j \Big) . \tag{1.18}$$

We report the full derivation of this formula Section 1.6. In both Equations (1.16) and (1.18), we have an explicit formula to compute $\mathbf{O}$ directly from $\tilde{\mathbf{C}}$, and thus, from $\mathbf{C}$.

1.3.4 Implementation Details

Despite their simplicity, the analytic solutions we introduced above can be unstable as they rely on the hypothesis that $\mathbf{C}$ maps perfectly the source eigenbasis onto the target, $T_F(\mathbf{\Phi}) = \mathbf{\Psi C}$. If the equality is not exact, we observed a quick degradation in accuracy. Moreover, in the iterative process of Equation (1.18), the error accumulates at each iteration. We highlight this issue in Figure 1.10 (left), by evaluating the error obtained in the estimation of $\mathbf{O}$ using the formula (1.16) while we increasingly add white noise to the coefficients stored in the map $\mathbf{C}$. We compute the relative error as the Frobenius norm of the difference between $T_F(\mathbf{\Phi})$ and $\mathbf{\Psi C}$ and the difference between $T_F(\mathbf{Q}_\Phi)$ and $\mathbf{Q}_\Psi \mathbf{O}$. The latter difference increases quickly when the former grows. Moreover, the accuracy of the transfer matrix $\mathbf{O}$ decreases as we increase K; we show this in Figure 1.10 (right), by plotting the error for different dimensions of the source and target bases. To solve these issues, we propose a different strategy to recover the transfer matrix $\mathbf{O}$. Being $\mathbf{O}$ a function

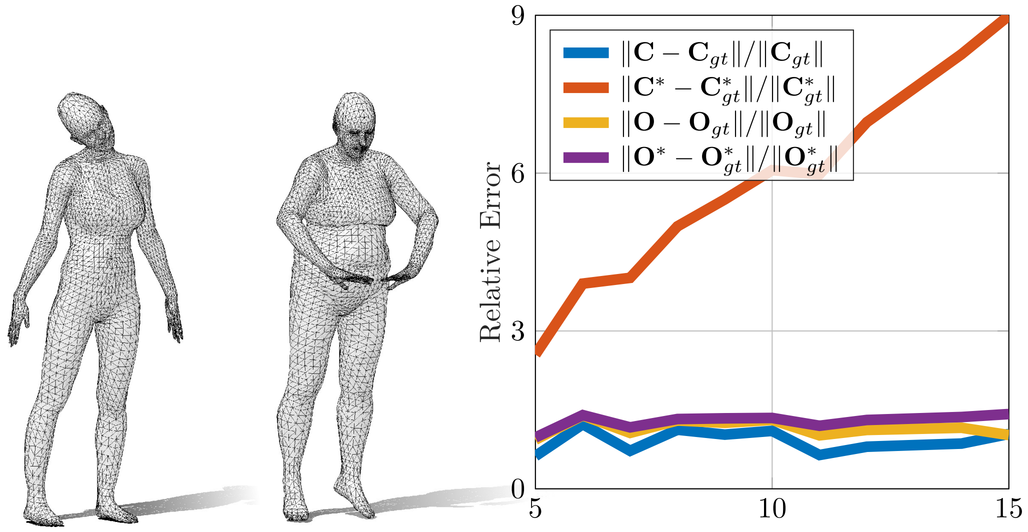


Figure 1.11. Comparison in the estimation of the transfer matrix directly from the point-to-point map extracted from $\tilde{\mathbf{C}}$. Left: an example pair with different connectivity. Right: error curves vs. size of the initial matrix $\mathbf{C}$, averaged over 20 random FAUST pairs.

of $\tilde{\mathbf{C}}$ (and so of $\mathbf{C}$), we make the educated guess that it is enough to align the first K eigenfunctions to obtain a sufficiently precise transfer matrix. To implement this idea, we proceed as follows:

1. We solve for the best permutation Π as the solution of a nearest neighbor assignment problem between $\tilde{\Psi}$ and $\tilde{\Phi}\tilde{\mathbf{C}}^\top$;

2. We estimate $\mathbf{O}$ as the solution in the least-squares sense of $\mathbf{Q}_\Psi \mathbf{O} = \Pi \mathbf{Q}_\Phi$.

This process is equivalent to extracting a point-to-point map from a given functional map as in [202], and then converting the point-to-point map to a new matrix with respect to different bases. Since our basis can be completely derived from the first K eigenfunctions (with respect to which the matrix $\mathbf{C}$ is encoded), we expect the estimated Π to contain enough information for estimating an accurate $\mathbf{O}$. By virtue of our analysis on the space spanned by the set of eigenproducts, we only need a small number of eigenfunctions $K \approx 12$, which ensures an easier optimization for the functional maps. It is worth to stress that this process is more feasible than extracting a point-to-point map directly from the orthogonalized basis, since we extract the map from just K eigenfunctions, rather than searching for a match between two sets of $\mathcal{O}(K^N)$ basis functions. This is even more efficient than fast refinement methods like ZoomOut [176], since we do a one-shot computation, rather than iterating on large bases one function at a time.

Dependency on the first K eigenfunctions. The i-th (for $i > K$) orthogonalized eigenproduct can be explicitly formulated in terms of the first K eigenfunctions. This is not true when we consider a basis of $K^* > K$ Laplacian eigenfunctions. Thus, it is reasonable to think that, starting from a functional map $\mathbf{C}$ for the first

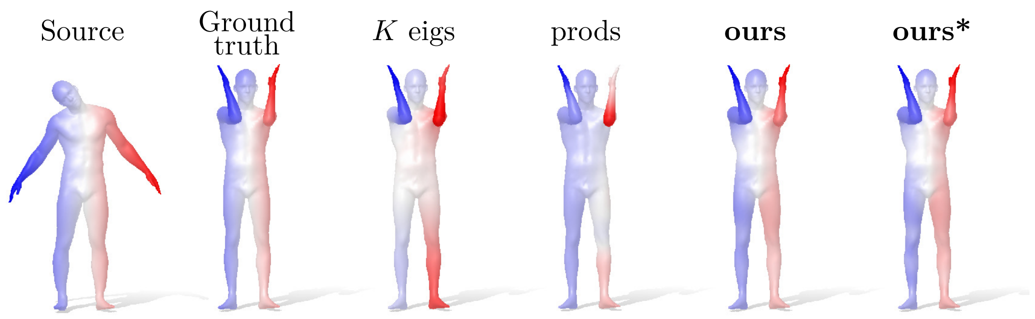
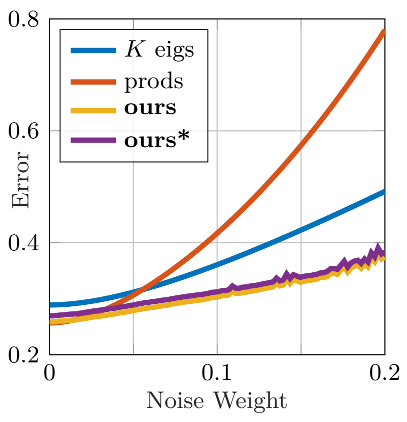


Figure 1.12. Comparisons on function transfer with noise.

K eigenfunctions, we can handle the orthonormalized products basis better than the standard Laplacian eigenbasis.

To test this claim, we apply the same strategy of computing our transfer matrix $\mathbf{O}$ to estimate a functional map $\mathbf{C}^*$ between eigenfunctions with the same cardinality as our bases. In Figure 1.11, for products of order $N = 3$ we compare the estimation of the two matrices varying the size K of the initial functional map $\mathbf{C}$ (x-axis). The error (y-axis) is defined as the relative Frobenius norm between the estimated and the ground-truth matrix. We report the error for the initial functional map $\mathbf{C}$ for comparison. In this test, the shapes do not share the same connectivity, as we remesh one of the two to 5k vertices. The errors for $\mathbf{O}$ and $\mathbf{O}^*$ are stable and comparable to the error of $\mathbf{C}$, while the error for the functional map $\mathbf{C}^*$ increases when K grows. This shows that it is possible to directly estimate $\mathbf{O}$ accurately from the point-to-point map extracted from $\mathbf{C}$.

Robustness to noise. We add an increasing amount of white noise (x-axis) to an input ground-truth functional map $\mathbf{C}$, and extract the transfer matrix $\mathbf{O}$. Then we compare the error in the transfer of the coordinate functions for K eigs, prods, ours and ours*. The curves in the inset figure show that our method is less sensitive to noise. These results are averaged on 20 random FAUST pairs, remeshed as in Figure 1.11. In Figure 1.12, we show a qualitative result under maximum noise. The corrupted map generates visible distortion in the transfer for both K eigs and prods, while the proposed method remains stable.



1.4 Experimental Results

We evaluate the performance of our basis in tasks of surface filtering, signal approximation and transfer. For all the tasks we compare with [193], and also report the

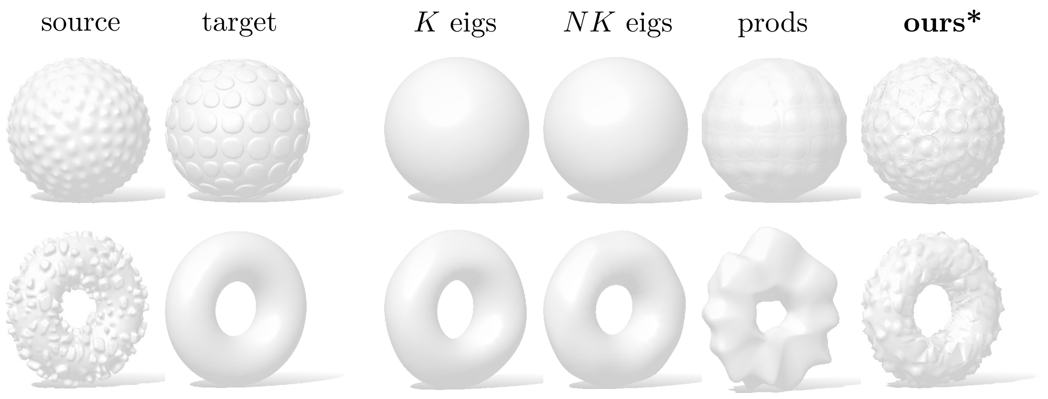


Figure 1.13. Detail transfer with our orthogonalized eigenproducts from a textured shape to another. The geometric details on the target are replaced correctly with those of the source shape.

results of taking a linear combination of NK eigenfunctions and of $\binom{K+N}{N}$ eigenfunctions.

We also investigate the numerical stability of GS, which may produce non-orthogonal vectors with large, non full-rank input bases, since linearly dependent functions must be discarded from the final set. This is done by thresholding small inner products. We consider (i) a direct implementation with a permissive threshold (10^{-2}) for the identification of linearly dependent vectors (denoted by **Ours**), and (ii) a reiterated variant, with a strict threshold (10^{-9}), which enforces orthogonality (**Ours***). The two approaches may output very different bases, which are shown by experiments to be suitable in different applications. In particular, **Ours** is more numerically stable, therefore more suitable for transferring signals in general (Tables 1.2 to 1.4). On the other hand, **Ours*** sacrifices numerical stability for more descriptive power, which can be useful for spectral filtering (Figures 1.14 and 1.15), reconstruction (Figure 1.17), detail transfer (Figure 1.13), or transfer between very different subjects (Table 1.6). The code is available at a public repository on GitHub[2].

1.4.1 Detail Transfer

In Figure 1.13, we highlight the representation power of the proposed basis by transferring fine geometric details (encoded as vertex coordinates) from a source shape to a target. For the first pair of shapes, we use products of order $N = 3$ of the first $K_{\mathcal{M}} = 30$ eigenfunctions on the source shape and $K_{\mathcal{N}} = 40$ on the target. For the second pair, we still use products of order $N = 3$, but we limited the number of eigenfunctions to $K_{\mathcal{M}} = 27$ on the source shape and $K_{\mathcal{N}} = 30$ on the target to show the result with a different set of parameters. The standard eigenfunctions cannot represent the details even with NK eigenfunctions, while prods [193] only approximates the details. Instead, our basis fulfils the task with good accuracy.

[2]https://github.com/filthynobleman/orthogonalized-fourier-polynomial

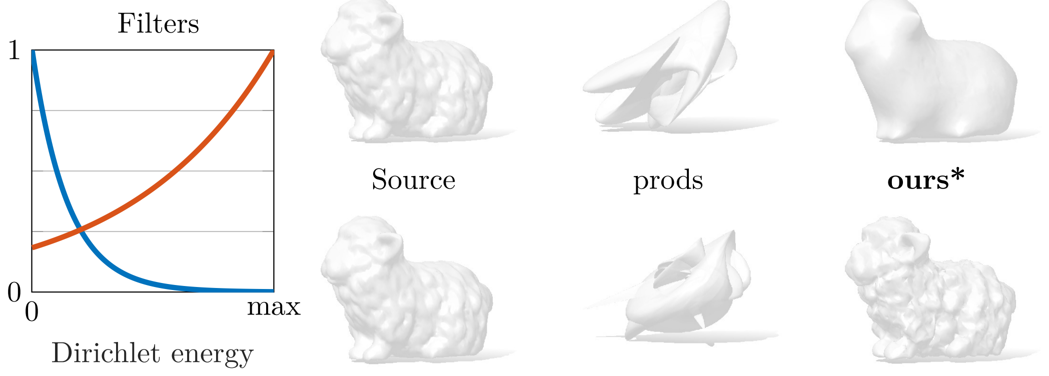


Figure 1.14. Filtering the coefficients of the vertex coordinates represented in the set of eigenproducts (middle) and in our orthogonal basis (right). We use a smoothing filter (blue curve, top row) and a sharpening filter (red curve, bottom row). Eigenproducts are not easy to control and do not provide meaningful variations, while our representation leads to the expected results.

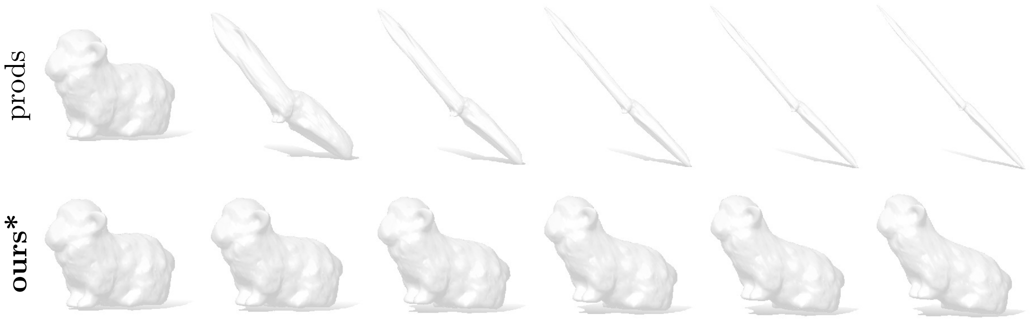


Figure 1.15. Stability of spectral filtering. From left to right, we gradually increase by ϵ only one coefficient in the spectral representation of the vertex coordinates; we use the same ϵ for both rows. While the reconstruction using eigenproducts degenerates quickly, ours yields a more stable and meaningful deformation.

1.4.2 Spectral Filtering

Since the eigenproducts of [193] are not linearly independent and do not provide a unique representation for a given signal, it is hard to design operations on the signal coefficients along the lines of spectral filtering approaches such as [272]. To demonstrate this, we run a test where we process vertex coordinates with simple filters (Figure 1.14): a smoothing filter that suppresses the high frequencies, and an enhancement filter that suppresses the lower portion of the spectrum. As a second test, in Figure 1.15 we continuously change a *single* coefficient in the spectral representation of the vertex coordinates, and then reconstruct the resulting geometry. While our basis can recover a meaningful smooth deformation of the original geometry, the plain eigenproducts rapidly degenerate to a flat shape. All these experiments are performed using products of order 3 on the first 15 eigenfunctions.

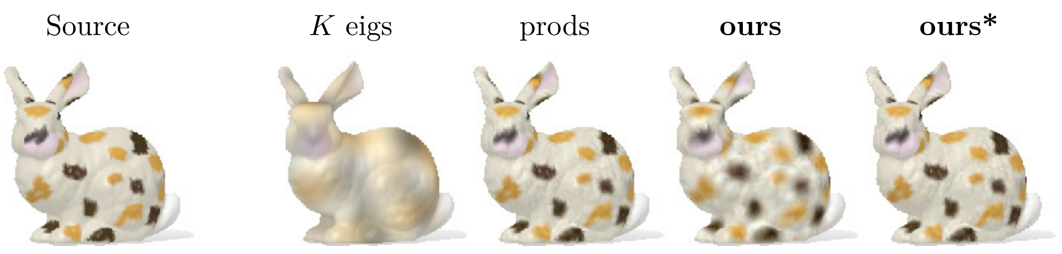


Figure 1.16. Reconstruction of a RGB signal using $K = 100$ eigenfunctions and order-2 products. We compare the reconstruction quality using K eigenfunctions, the eigenproducts of [193], and our two methods based on the orthogonalized basis.

	K	prods	**ours**	NK	$\binom{K+N}{N}$
HK k	**0.0%**	**0.0%**	**0.0%**	0.0%	0.0%
HK K	75.8%	18.1%	**13.7%**	61.1%	0.0%
HKS	1.1%	**0.0%**	**0.0%**	0.4%	0.0%
WKS	14.1%	**0.0%**	**0.0%**	5.7%	0.1%
Rand	37.5%	33.8%	**33.5%**	37.2%	33.4%
XYZ	15.2%	2.0%	**1.7%**	7.6%	0.8%
Ind	29.2%	12.5%	**12.2%**	23.6%	11.4%
SHOT	66.8%	35.4%	**33.8%**	58.8%	28.9%
AWFT	12.5%	5.9%	**5.8%**	10.5%	4.2%

Table 1.1. Reconstruction error of our method compared to the approach from [193] (prods) and just taking linear combinations of K eigenfunctions (K). As ideal references, we also report the results obtained via linear combinations of NK and $\binom{K+N}{N}$ eigenfunctions, which are prohibitive to compute on large shapes. Here, we used $K = 30$ and $N = 2$.

1.4.3 Function Approximation

In a discrete setting, computing K Laplacian eigenfunctions boils down to solving an eigenproblem for a sparse symmetric $n \times n$ matrix, which has complexity $\mathcal{O}(Kn^2)$ [255]. This is prohibitive for high resolution meshes and for large K, which is required to capture fine details. Using eigenproducts for the representation as in [193] can lead to big accuracy improvements, at the cost of increased numerical instability as discussed above. By contrast, our orthonormal basis leads to more stable and accurate results.

In Table 1.1 we compare with the standard eigenfunctions and the eigenproducts of [193] in terms of reconstruction error, measured as $(\int_{\mathcal{M}}(f-\tilde{f})^2)^{1/2}/(\int_{\mathcal{M}} f^2)^{1/2}$, where f is the original signal and $\tilde{f}$ is its reconstruction. We consider the same families of functions as in [193]. Namely, HK k, HK K: the heat kernel between a random point and the rest of the shape approximated using 200 and K eigenfunctions respectively. HKS, WKS: the heat and wave kernel signatures [260, 14]. Rand: random functions. XYZ: vertex coordinates. Ind: the binary indicator function of a random region. SHOT, AWFT: local descriptors [268, 177].

Moving to an orthonormal basis brings additional regularization, leading in turn to an increase in quality. Both our orthogonal basis and the plain eigenproducts

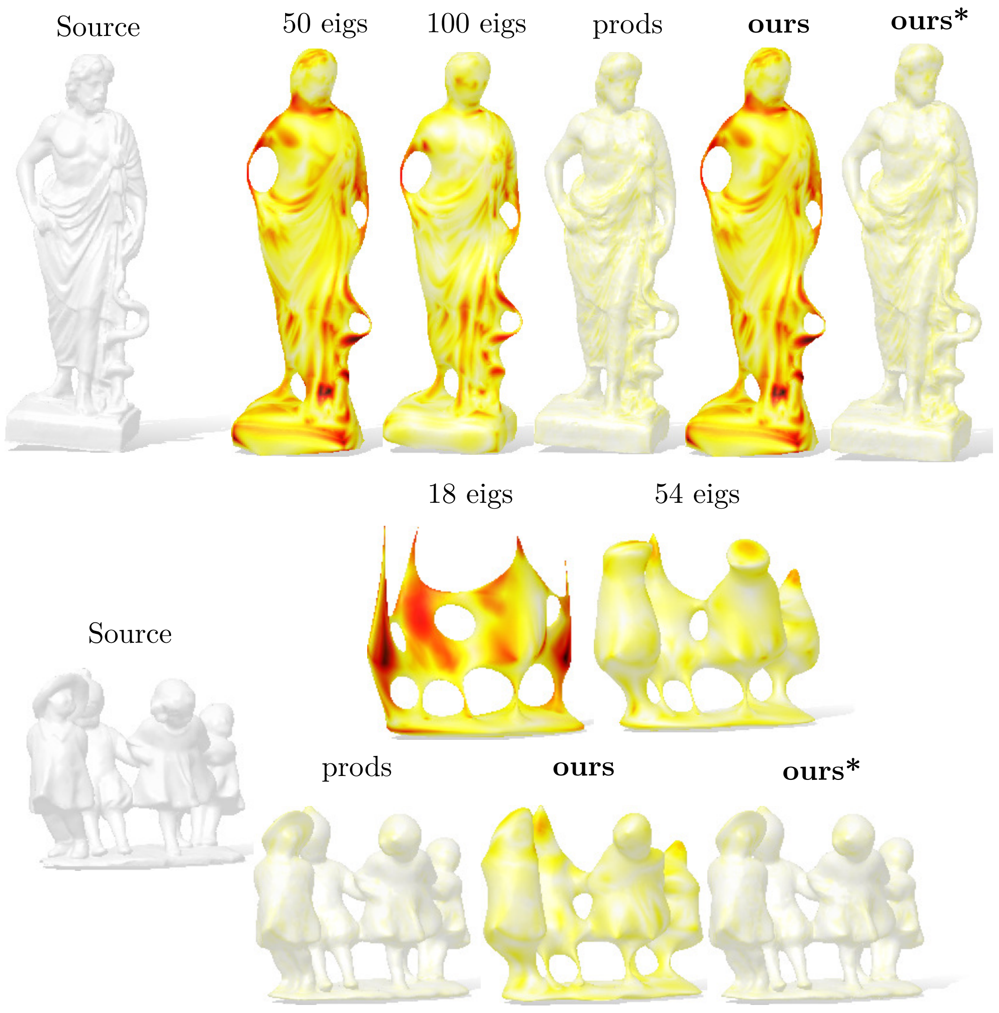


Figure 1.17. Coordinate reconstruction of two shapes. With N-th order products of K eigenfunctions, the eigenproducts and the orthogonal basis get better results than NK eigenfunctions, and start to catch some details from the surface. A lower threshold in the orthogonalization process produces a more descriptive basis, in these cases. Here we use $N = 2, K = 50$ for the statue and $N = 3, K = 18$ for the dancing children.

	K	prods	**ours**	**ours***	NK
HK k	70.3%	64.4%	**50.9%**	52.0%	17.7%
HK K	78.4%	61.3%	**52.8%**	55.9%	56.4%
HKS	12.0%	10.4%	**5.5%**	5.6%	5.9%
WKS	29.3%	22.0%	**10.6%**	10.8%	17.7%
Rand	50.7%	54.8%	**49.9%**	50.0%	50.8%
XYZ	40.6%	44.5%	**32.4%**	33.2%	21.3%
Ind	47.7%	51.2%	27.2%	**24.8%**	35.9%
SHOT	74.6%	73.3%	65.4%	**65.3%**	67.0%
AWFT	31.4%	35.4%	**23.8%**	24.2%	22.9%

Table 1.2. Transfer error comparison on 10 isometric pairs from TOSCA. The parameters are $N = 3$ and $K_{\mathcal{M}} = K_{\mathcal{N}} = 12$.

produce much better results than those obtained with K or even NK eigenfunctions. The two methods can, in most cases, compete with the approach of using $\binom{K+N}{N}$ eigenfunctions, without incurring in the prohibitive cost of computing a massive number of eigenfunctions (in the order of $\mathcal{O}(K^N)$). Qualitative results on this task are shown in Figures 1.1 and 1.17 for the coordinate functions (XYZ), and in Figure 1.16 for a RGB signal.

1.4.4 Function Transfer

We evaluate the transfer task on several datasets, differing in terms of mesh quality, resolution, regularity, and deformation type (isometry or lack thereof). We consider the same set of functions used for this experiment in [193]. As a reference, we also report the results obtained with linear combinations of NK standard eigenfunctions. For each method, we compute the functional map matrix $\mathbf{C}$ with the method of [194]. As an error measure, we report the normalized error $(\int_{\mathcal{N}}(f_{gt} - \tilde{f})^2)^{1/2}/(\int_{\mathcal{N}} f_{gt}^2)^{1/2}$, where f_{gt} is the ground truth signal on the target $\mathcal{N}$ and $\tilde{f}$ is the transferred counterpart.

Near-isometries (synthetic). For synthetic near-isometric meshes we use 10 random pairs from TOSCA [38]; Table 1.2 summarizes the comparisons, showing a 5–25% improvement over [193]. Using third-order products allows to successfully transfer the 46-th eigenfunction by using only 15 eigenfunctions on the source and 18 on the target shape, as shown in Figure 1.18.

Non-isometries (real humans). Comparisons are favorable also when considering more realistic non-isometric shapes, where we use 20 random pairs from the FAUST [30] dataset of real scans. Each pair is simultaneously inter-class and inter-pose. Table 1.3 summarizes the results. On this dataset, we select K and N as in [193] for a direct comparison.

Near-isometries (different meshing). We further evaluate the setting where the shapes have a wildly different vertex count and connectivity. For these tests, we

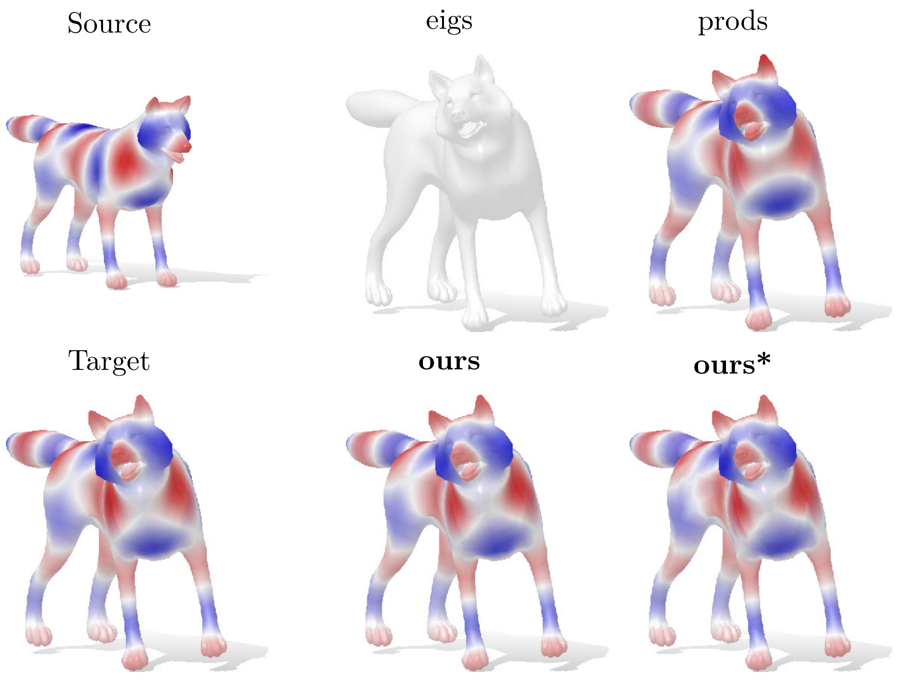


Figure 1.18. Transferring a high-frequency eigenfunction that is out of the span of the truncated Laplacian eigenbasis. Projecting onto the latter yields a zero signal due to orthogonality, while eigenproducts and our orthogonal basis can transfer it more precisely. Here we used $K_{\mathcal{M}} = 15$ eigenfunctions on the source shape, $K_{\mathcal{N}} = 18$ on the target shape, and $N = 3$ (third-order products).

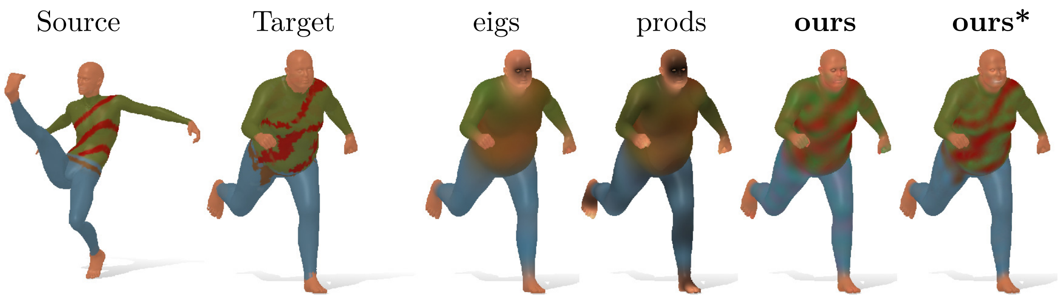


Figure 1.19. RGB signal transfer comparison on a pair from the SHREC'19 dataset. Our approach better transfers the fine details. In this example, the noise in the ground truth correspondence due to wildly different meshings is enough to produce local distortion in the function transfer, but our approach remains stable and produces a smooth correspondence.

take 10 random pairs from SHREC'19 [174]; the results are reported in Table 1.4 and highlight how, under a different connectivity, even using a large eigenbasis incurs into issues (NK column), whereas our method still gives stable results. Note that when a mesh in a pair is undersampled with respect to the other, the ground-truth correspondence is patchy and low-quality. Figure 1.19 shows how our approach can still produce a higher quality transfer.

	K	prods	**ours**	**ours***	NK
HK k	28.0%	27.4%	**26.6%**	27.1%	22.5%
HK K	66.7%	51.3%	**46.4%**	**46.4%**	56.3%
HKS	8.1%	12.3%	**6.6%**	6.7%	8.2%
WKS	16.7%	15.5%	**11.4%**	11.6%	12.3%
Rand	49.9%	51.0%	**49.6%**	50.5%	50.0%
XYZ	21.3%	21.9%	**19.5%**	20.0%	19.5%
Ind	36.4%	33.7%	**30.1%**	30.2%	31.4%
SHOT	76.6%	75.1%	**73.7%**	76.3%	70.7%
AWFT	15.1%	18.3%	**13.4%**	13.9%	14.7%

Table 1.3. Transfer error comparison on 20 inter-class pairs from FAUST. Here $N = 2$, $K_{\mathcal{M}} = 30$ and $K_{\mathcal{N}} = 40$.

	K	prods	**ours**	**ours***	NK
HK k	27.9%	29.3%	**26.2%**	26.4%	30.4%
HK K	79.9%	67.4%	**62.7%**	62.8%	75.8%
HKS	12.6%	17.5%	**10.7%**	10.8%	13.7%
WKS	25.4%	21.6%	**18.5%**	18.6%	22.4%
Rand	50.8%	52.8%	**50.2%**	50.4%	51.5%
XYZ	30.5%	30.2%	**29.2%**	29.6%	30.8%
Ind	41.0%	38.2%	**34.8%**	35.7%	36.6%
SHOT	66.3%	66.0%	**62.8%**	64.7%	64.8%
AWFT	24.6%	28.0%	**21.4%**	21.6%	25.6%

Table 1.4. Transfer error comparison on 10 non-isometric pairs (with very different connectivity) from SHREC'19, with $N = 2$, $K_{\mathcal{M}} = 30$ and $K_{\mathcal{N}} = 40$.

Non-isometries (different animals). We extend our comparison to the recent dataset SHREC'20[76], composed by 30 pairs divided in 5 different test-sets (1 partial-to-full and 4 full-to-full shape with 4 different levels of isometry: highest, high, low, lowest). In total, 14 different shapes are involved with different connectivity, topological errors and missing parts. A set of ∼50 sparse ground truth correspondences are provided for the evaluation. Due to the impossibility of retrieving a dense ground truth, here we measure the average absolute error $|f_{gt} - \tilde{f}|$ on the sparse landmarks only. In Table 1.5, we show that the proposed method transfers functions between these animals better than the competitors. A qualitative visualization is given by the RGB transfer results in Figure 1.2.

Non-isometries (different semantic class). Finally, we address the difficult setting where the shapes belong to different classes. We use the MISC dataset [175] with 10 random pairs in random poses for a woman, a man and a gorilla; a sparse correspondence between ∼10% of the vertices is given. The results reported in Table 1.6 show that, with orthogonalized third-order eigenproducts, we are still able to transfer functions with high accuracy. Having only a sparse ground truth correspondence, we use the same error measure of SHREC'20. We want to stress that,

	K	prods	**ours**	**ours***	NK
HK k	**12.2%**	15.8%	15.2%	13.0%	14.8%
HK K	4.4%	5.9%	5.3%	**3.2%**	4.8%
HKS	26.6%	30.9%	**23.6%**	**23.6%**	26.7%
WKS	19.2%	25.1%	**17.7%**	**17.7%**	19.9%
Rand	26.8%	40.4%	27.2%	**25.2%**	26.9%
XYZ	18.1%	51.5%	11.9%	**7.4%**	20.4%
Ind	21.0%	26.1%	20.4%	**18.6%**	22.1%
SHOT	**10.4%**	15.8%	12.6%	11.3%	10.7%
AWFT	18.8%	30.8%	20.4%	**18.3%**	19.6%

Table 1.5. Transfer error comparison for SHREC'20, averaged on the 5 test-sets (30 pairs with different levels of non-isometry, topological errors and missing parts), with $N = 3$, $K_{\mathcal{M}} = 12$, $K_{\mathcal{N}} = 12$. The values are multiplied by 10^2 to help the comparison.

	K	prods	**ours**	**ours***	NK
HK k	11.0%	10.6%	12.0%	**9.2%**	8.7%
HK K	3.1%	3.7%	2.9%	**2.1%**	3.2%
HKS	5.8%	6.2%	**5.4%**	**5.4%**	9.6%
WKS	11.0%	8.2%	**7.3%**	**7.3%**	9.3%
Rand	25.4%	28.6%	**25.2%**	25.4%	25.9%
XYZ	20.1%	21.1%	19.5%	**19.2%**	18.1%
Ind	10.9%	10.1%	8.0%	**6.9%**	11.3%
SHOT	**0.2%**	**0.2%**	**0.2%**	**0.2%**	0.2%
AWFT	**5.9%**	7.8%	6.5%	6.3%	5.6%

Table 1.6. Transfer error comparison on 10 strongly non-isometric pairs from MISC. In this setting, $N = 3$ and $K_{\mathcal{M}} = K_{\mathcal{N}} = 12$.

by using this measure, the values shown in Tables 1.5 and 1.6 must be interpreted differently from those shown in Tables 1.2 to 1.4. The error, here, is not relative, and depending on the availability of landmarks, it can have higher of lower values. However, this does not affect the comparison, since all the methods are compared under the same conditions. In Figure 1.20, we show a qualitative result for a pair of non-isometric shapes.

1.5 Conclusions

In this chapter, we proposed a new orthonormal basis based on the pointwise products of the eigenfunctions of the Laplace-Beltrami operator. We provided a theoretical analysis of the properties carried by this basis, and assessed its practical value in different settings and applications. In particular, orthogonalization yields a notable improvement over prior work [193] in terms of computational complexity and expressive power.

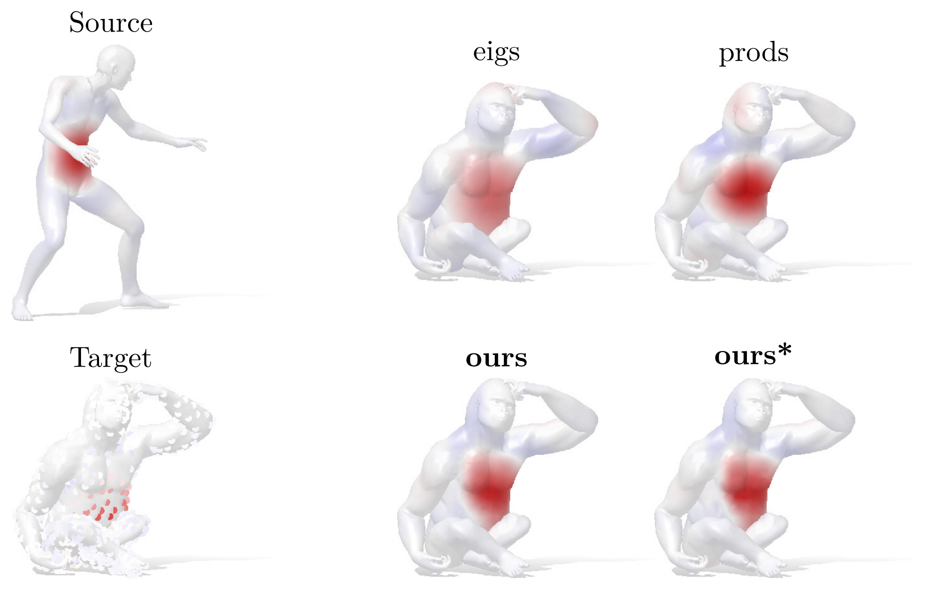


Figure 1.20. Example of function transfer on the MISC dataset. The ground truth on the target is only shown for the given landmarks.

Limitations and future work. Perhaps the main limitation of our approach lies in the pipeline for function transfer, where we strongly depend on the quality of the given functional map. If the method for building this map is not stable, the behavior of the orthogonal basis can be unpredictable. Furthermore, currently we need to rely on a point-to-point map in order to compute the transfer matrix, since our analytical derivations are not stable. Directly estimating this matrix *without* resorting to point-to-point conversion, is an important direction that we intend to explore. Another possible avenue for future research is to extend our construction to accommodate different definitions of products, for a more efficient and accurate representation for surface signals. Finally, we will explore the adoption of our basis in existing pipelines, such as ZoomOut [176], which are basis-agnostic and allow to extend this work to a number of other tasks.

1.6 Proofs

1.6.1 Laplacian of a Product

We want to show that, given a function $f(x) = \prod_{i=1}^{n} f_i(x)$, the application of the Laplace-Beltrami operator results in

$$\Delta f(x) = \sum_{i=1}^{n} \prod_{\substack{j=1 \\ j\neq i}}^{n} f_j(x)\Delta f_i(x) - \sum_{\substack{i,j=1 \\ i\neq j}}^{n} \prod_{\substack{h=1 \\ h\neq i,j}}^{n} f_h(x)\langle \nabla f_i(x), \nabla f_j(x)\rangle \tag{1.19}$$

We show it by induction, with the base case given by Equation (1.1) for $n = 2$. Assuming this true up to $n-1$, for $g = \prod_{i=1}^{n-1} f_i$ we have

$$\Delta f(x) = f_n(x)\Delta g(x) + g(x)\Delta f_n(x) - 2\,\langle \nabla f_n(x),\ \nabla g(x)\rangle \tag{1.20}$$

The first term expands to

$$\sum_{i=1}^{n-1} \prod_{\substack{j=1 \\ j\neq i}}^{n} f_j(x)\Delta f_i(x) - \sum_{\substack{i,j=1 \\ i\neq j}}^{n-1} \prod_{\substack{h=1 \\ h\neq i,j}}^{n} f_h(x) \left\langle \nabla f_i(x),\ \nabla f_j(x) \right\rangle \,, \tag{1.21}$$

while the second term is $\prod_{i=1}^{n-1} f_i(x)\Delta f_n(x)$, and the third term is

$$-2\sum_{j=1}^{n-1} \prod_{\substack{h=1 \\ h\neq j,n}}^{n} f_h(x) \left\langle \nabla f_j(x),\ \nabla f_n(x) \right\rangle \tag{1.22}$$

By adding these all together, we finally get back Equation (1.19).

1.6.2 Proof of Theorem 1.1

Proof. For simplicity, we assume $\|\phi_I\|^2 = 1$. Also, to keep the equations more readable, we will abuse of our notation. When we iterate over multiple indices and we indicate $i \neq j$ we do not mean that index i must be necessarily different from index j (since eigenproducts can involve the same eigenfunction multiple times), but that we cannot pick the i-th index two times.

By observing that

$$\Delta\phi_I(x) = \sum_{i_i\in I} \lambda_{i_i}\phi_I(x) - \sum_{\substack{i,j\in I \\ i\neq j}} \prod_{\substack{h\in I \\ h\neq i,j}} \phi_h(x) \left\langle \nabla\phi_i(x),\ \nabla\phi_j(x) \right\rangle \,, \tag{1.23}$$

and by knowing $\langle \nabla f,\ \nabla g\rangle_{T(\mathcal{M})} = \langle f,\ \Delta g\rangle_{\mathcal{M}}$, we easily get

$$\mathcal{E}\left(\phi_I\right) = \sum_{i\in I} \lambda_i - \sum_{\substack{i,j\in I \\ i\neq j}} \int_{\mathcal{M}} \prod_{\substack{h\in I \\ h\neq i}} \phi_h(x) \prod_{\substack{p\in I \\ p\neq j}} \phi_p(x) \left\langle \nabla\phi_i(x),\ \nabla\phi_j(x) \right\rangle \mathrm{d}x \,. \tag{1.24}$$

Now, let us consider the following terms:

$$\sum_{i\in I} \int_{\mathcal{M}} \prod_{\substack{j\in I \\ i\neq j}} \phi_j^2(x) \left\langle \nabla\phi_i(x),\ \nabla\phi_i(x) \right\rangle \mathrm{d}x \,. \tag{1.25}$$

Since the integrands are always non-negative, the whole sum is non-negative. Hence, we can give a lower bound to Equation (1.24):

$$\mathcal{E}\left(\phi_I\right) \geq \sum_{i\in I} \lambda_i - \sum_{i,j\in I} \int_{\mathcal{M}} \prod_{\substack{h\in I \\ h\neq i}} \phi_h(x) \prod_{\substack{p\in I \\ p\neq j}} \phi_p(x) \left\langle \nabla\phi_i(x),\ \nabla\phi_j(x) \right\rangle \mathrm{d}x \,. \tag{1.26}$$

The second term is the Dirichlet energy of ϕ_I. In fact, we have:

$$\begin{aligned} \mathcal{E}\left(\phi_I\right) &= \left\langle \nabla\phi_I,\ \nabla\phi_I \right\rangle_{T(\mathcal{M})} = \\ &= \sum_{i,j\in I} \left\langle \prod_{\substack{h\in I \\ h\neq i}} \phi_h\nabla\phi_i,\ \prod_{\substack{p\in I \\ p\neq j}} \phi_p\nabla\phi_j \right\rangle_{T(\mathcal{M})} = \\ &= \sum_{i,j\in I} \int_{\mathcal{M}} \prod_{\substack{h\in I \\ h\neq i}} \phi_h(x) \prod_{\substack{p\in I \\ p\neq j}} \phi_p(x) \left\langle \nabla\phi_i(x),\ \nabla\phi_j(x) \right\rangle \mathrm{d}x \,. \end{aligned} \tag{1.27}$$

By plugging it into Equation (1.26), we get:

$$\mathcal{E}\left(\phi_I\right) \geq \sum_{i \in I} \lambda_i - \mathcal{E}\left(\phi_I\right) \,, \tag{1.28}$$

which results in the claim.

Consider now the special case of $\phi_I = \phi_i^n$. In this case, all the integrals in Equation (1.24) are equal and can be written as

$$\int_{\mathcal{M}} \left\langle \phi_i^{n-1}(x) \nabla \phi_i(x),\ \phi_i^{n-1}(x) \nabla \phi_i(x) \right\rangle \mathrm{d}x \,, \tag{1.29}$$

which, recalling $\frac{1}{\alpha} \nabla f^{\alpha}(x) = f^{\alpha-1}(x) \nabla f(x)$, gives

$$\int_{\mathcal{M}} \left\| \phi_i^{n-1}(x) \nabla \phi_i(x) \right\|^2 \mathrm{d}x = \frac{1}{n^2} \int_{\mathcal{M}} \left\| \nabla \phi_i^n(x) \right\|^2 \mathrm{d}x = \frac{1}{n^2} \mathcal{E}\left(\phi_i^n\right) \,. \tag{1.30}$$

Since in Equation (1.24) there is a sum over $i \neq j$, we have a total of $n(n-1)$ of these terms. Hence, we are left with

$$\mathcal{E}\left(\phi_i^n\right) = n \lambda_i - \frac{n-1}{n} \mathcal{E}\left(\phi_i^n\right) \,, \tag{1.31}$$

which, with simple algebraic manipulations, leads to

$$\mathcal{E}\left(\phi_i^n\right) = \frac{n^2}{2n-1} \lambda_i \,. \tag{1.32}$$

□

1.6.3 Proof of Corollary 1.1

Proof. We know that the set of eigenproducts contains all the eigenpowers, and in particular $\phi_K^N \in \tilde{\mathbf{\Phi}}$. Thus, the following holds:

$$\max_{\phi_I \in \tilde{\mathbf{\Phi}}} \{\mathcal{E}\left(\phi_I\right)\} \geq \mathcal{E}\left(\phi_K^N\right) = \frac{N^2}{2N-1} \lambda_K \,. \tag{1.33}$$

By Weyl's asymptotic law for 2-dimensional manifolds we have $\lambda_K \in \Theta(K)$, therefore $\mathcal{E}\left(\phi_K^N\right) \in \Theta(NK)$. Since the maximum Dirichlet energy is lower bounded by this value, it must be that $\max_{\phi_I \in \tilde{\mathbf{\Phi}}} \{\mathcal{E}\left(\phi_I\right)\} \in \Omega(NK)$. □

1.6.4 Iterative Formula for the Transform O

We consider $\Pi : \mathcal{N} \to \mathcal{M}$ the point-to-point correspondence between two shapes $\mathcal{N}$ and $\mathcal{M}$. $T_F : L^2(\mathcal{M}) \to L^2(\mathcal{N})$ is the functional map associated to this correspondence defined via pull-back $T_F(f) = f \circ \pi$, $\forall f \in L^2(\mathcal{M})$. $\mathbf{C}$ is the matrix representation of T_F in a truncated pair of bases $\mathbf{\Phi}$ and $\mathbf{\Psi}$ for $L^2(\mathcal{M})$ and $L^2(\mathcal{N})$ respectively. For simplicity we consider both the finite bases with the same dimension K and the general case directly arise from our analysis. $\tilde{\mathbf{\Phi}}$ and $\tilde{\mathbf{\Psi}}$ are the matrices the columns of which are the eigenproducts of order N of the basis functions contained in $\mathbf{\Phi}$ and $\mathbf{\Psi}$. We represent each of these eigenproducts as $\tilde{\phi}_h$ and $\tilde{\psi}_\ell$ $\tilde{\mathbf{C}}$ is the funxtional maps extended to the eigenproducts following the formula proposed in [193].

As we describe in the main document, our bases are obtained by applying the Gram-Schmidt algorithm to $\tilde{\mathbf{\Phi}}$ and $\tilde{\mathbf{\Psi}}$ obtaining the two couples $\tilde{\mathbf{\Phi}} = \mathbf{Q}^{\Phi}\mathbf{R}^{\Phi}$ and $\tilde{\mathbf{\Psi}} = \mathbf{Q}^{\Psi}\mathbf{R}^{\Psi}$. we are therefore interested in estimating a transform $\mathbf{O}$ such that $T_F(\mathbf{Q}^{\Phi}) = \mathbf{Q}^{\Psi}\mathbf{O}$.

First of all due to the role of $\mathbf{C}$ we can write $\mathbf{\Psi C} = T_F(\mathbf{\Phi})$ and $\tilde{\mathbf{\Psi}}\tilde{\mathbf{C}} = T_F(\tilde{\mathbf{\Phi}})$. These equality could be only approximation depending on the quality of the maps $\mathbf{C}$ and $\tilde{\mathbf{C}}$, and in the alignment of the bases involved. For this reason we can already set the first K columns of $\mathbf{O} = \begin{bmatrix}\mathbf{C}\\ \mathbf{0}\end{bmatrix}$, where $\mathbf{0}$ is a matrix of zeros with K columns and the number of rows equal to the number of functions in $\mathbf{Q}^{\Psi}$ minus K. This allows us to consider the first K columns of $\mathbf{O}$ already computed and only estimated the remaining ones. For this reason we look for an iterative construction of $\mathbf{O}$ that estimate at each iteration a new column of $\mathbf{O}$ from left to right. This procedure is related with the iterative construction of the bases $\mathbf{Q}^{\Phi}$ and $\mathbf{Q}^{\Psi}$.

Let us start writing the explicit formula for the column ζ_i of $\mathbf{Q}^{\Phi}$:

$$\zeta_i = \frac{1}{\mathbf{R}^{\Phi}_{i,i}}(\tilde{\phi}_i - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}\zeta_h), \tag{1.34}$$

and in the same way:

$$\xi_j = \frac{1}{\mathbf{R}^{\Psi}_{j,j}}(\tilde{\psi}_j - \sum_{\ell=1}^{j-1}\mathbf{R}^{\Psi}_{i,\ell}\xi_\ell). \tag{1.35}$$

Now we want to compute the image via T_F of each function ζ_i (column) of $\mathbf{Q}^{\Phi}$, for $i > K$, thanks to the linearity of T_F we have:

$$\begin{aligned} T_F(\zeta_i) &= T_F\Big(\frac{1}{\mathbf{R}^{\Phi}_{i,i}}(\tilde{\phi}_i - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}\zeta_h)\Big) = \\ &= \frac{1}{\mathbf{R}^{\Phi}_{i,i}}(T_F(\tilde{\phi}_i) - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}T_F(\zeta_h)). \end{aligned} \tag{1.36}$$

Now we can consider that:

$$T_F(\tilde{\phi}_i) = \tilde{\mathbf{\Psi}}\tilde{\mathbf{C}}_{:,i}\,, \tag{1.37}$$

$$T_F(\zeta_h) = \sum_j \mathbf{O}_{j,h}\xi_j, \tag{1.38}$$

where j goes from 1 to the number of functions in $\mathbf{Q}^{\Psi}$ while $\tilde{\mathbf{C}}_{:,i}$ is the i-th column of $\tilde{\mathbf{C}}$. Equation (1.38) can be written only because we already know all the elements of $\mathbf{O}$, $\forall j$ and $\forall h \leq i-1$. Then we can substitute the equivalences Equations (1.37) and (1.38) in Equation (1.36):

$$T_F(\zeta_i) = \frac{1}{\mathbf{R}^{\Phi}_{i,i}}\Big(\tilde{\mathbf{\Psi}}\tilde{\mathbf{C}}_{:,i} - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}(\sum_j \mathbf{O}_{j,h}\xi_j)\Big). \tag{1.39}$$

Now we know that $\tilde{\mathbf{\Psi}} = \mathbf{Q}^{\Psi}\mathbf{R}^{\Psi}$ so we can write:

$$\begin{aligned} T_F(\zeta_i) &= \frac{1}{\mathbf{R}^{\Phi}_{i,i}}\Big(\mathbf{Q}^{\Psi}\mathbf{R}^{\Psi}\tilde{\mathbf{C}}_{:,i} - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}(\sum_j \mathbf{O}_{j,h}\xi_j)\Big) = \\ &= \frac{1}{\mathbf{R}^{\Phi}_{i,i}}\Big(\sum_j\sum_{l=1}\mathbf{R}^{\Psi}_{j,l}\tilde{\mathbf{C}}_{l,i}\xi_j - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}(\sum_j \mathbf{O}_{j,h}\xi_j)\Big), \end{aligned} \tag{1.40}$$

where the last equation comes from the definition of the matrix product. Now we can collect and reorder the element in the last equation with respect to the sum over j:

$$T_F(\zeta_i) = \frac{1}{\mathbf{R}^{\Phi}_{i,i}}\Big(\sum_j \underbrace{(\sum_{l=1}\mathbf{R}^{\Psi}_{j,l}\tilde{\mathbf{C}}_{l,i} - \sum_{h=1}^{i-1}\mathbf{R}^{\Phi}_{i,h}\mathbf{O}_{j,h})}_{\mathbf{O}_{j,i}}\,\xi_j\Big), \tag{1.41}$$

where $\mathbf{O}_{j,i}$ only depends on $\mathbf{R}^{\Phi}, \tilde{\mathbf{C}}, \mathbf{R}^{\Psi}$ and form the first $i-1$ columns of $\mathbf{O}$. This result proves that $\mathbf{O}$ can be iteratively computed on its columns as a function of known variables. Moreover, being $\tilde{\mathbf{C}}$ a function of $\mathbf{C}$, this result clarify once again that it is possible to fully recover $\mathbf{O}$ from $\mathbf{C}$ justifying the proposed procedure.

Chapter 2

Learning Spectral Unions of Partial Deformable 3D Shapes

Spectral geometric methods have brought revolutionary changes to the field of geometry processing. Of particular interest is the study of the Laplacian spectrum as a compact, isometry and permutation-invariant representation of a shape. Some recent works show how the intrinsic geometry of a full shape can be recovered from its spectrum, but there are approaches that consider the more challenging problem of recovering the geometry from the spectral information of partial shapes. In this chapter, we propose a possible way to fill this gap. We introduce a learning-based method to estimate the Laplacian spectrum of the union of partial non-rigid 3D shapes, without actually computing the 3D geometry of the union or any correspondence between those partial shapes. We do so by operating purely in the spectral domain and by defining the union operation between short sequences of eigenvalues. We show that the approximated union spectrum can be used as-is to reconstruct the complete geometry [168], perform region localization on a template [224] and retrieve shapes from a database, generalizing ShapeDNA [228] to work with partialities. Working with eigenvalues allows us to deal with unknown correspondence, different sampling, and different discretizations (point clouds and meshes alike), making this operation especially robust and general. Our approach is data-driven and can generalize to isometric and non-isometric deformations of the surface, as long as these stay within the same semantic class (e.g., human bodies or horses), as well as to partiality artifacts not seen at training time. This work was made in collaboration with Luca Moschella (MSc.), to whom goes the credit for carrying out the design and implementation of the network architecture and the experiments, to prof. Simone Melzi and prof. Luca Cosmo, both of which largely helped in improving the technical soundness, strengthening the theoretical foundations, and designing the experimental setup, to Or Litany (Ph.D.), prof. Maksim Ovsjanikov and prof. Leonidas Guibas, who contributed in improving the overall presentation, and to prof. Emanuele Rodolà, whose supervision over the entire research greatly helped in reaching and improving the achieved results. The results presented in this chapter have been published in the proceedings of the *Annual Conference of the European Association for Computer Graphics* (*EUROGRAPHICS*) [185].

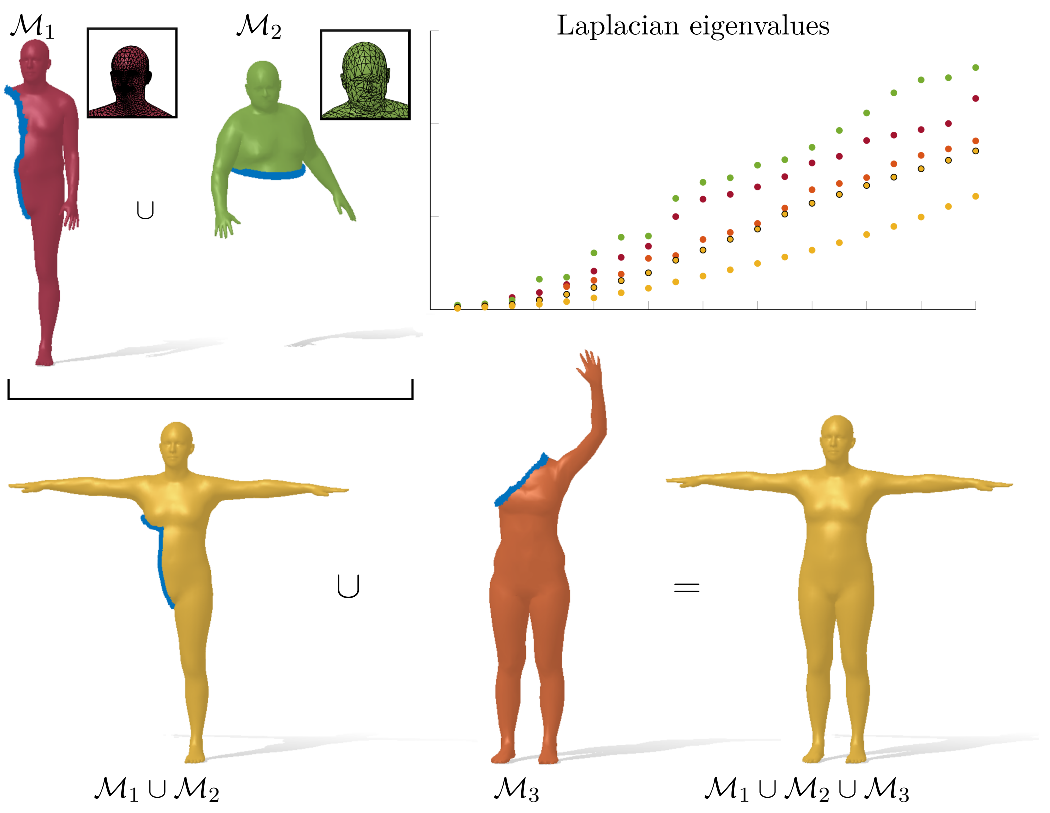

$\mathcal{M}_1$
$\cup$
$\mathcal{M}_2$
Laplacian eigenvalues
$\mathcal{M}_1 \cup \mathcal{M}_2$
$\cup$
$\mathcal{M}_3$
$=$
$\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3$

Figure 2.1. Given a collection of partial deformable shapes $\{\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3\}$ as input, our method predicts the Laplacian eigenvalues of their union without first having to compute a correspondence or a transformation between the input shapes. The resulting eigenvalues (top right plots, colors correspond to each surface) can be used to reconstruct the final shape if needed, up to isometry/pose (bottom right). In this example, the input shapes have different poses, varying overlap, and different mesh connectivity.

2.1 Introduction

Recent progress in spectral geometry processing has brought to significant qualitative leaps that lead to better results in a range of challenging tasks such as deformable shape matching [202, 144], retrieval [228, 39], style [168] and pose transfer [127, 296] among others.

More recently, a great deal of attention has been put on the study of the eigenvalues of the Laplace-Beltrami operator (i.e., the *Laplacian spectrum*) as a compact, isometry and permutation-invariant representation of the input shape. It has been shown that, with the appropriate knowledge on the input domain, this representation contains enough information for localize shape's regions [224] and even to reconstruct the geometry of the shape [60, 165]. However, these methods typically operate in a controlled scenario requiring to have access to the *full* geometry of the shape, ignoring the fact that real-world data are riddled with partiality artifacts.

In this chapter, we propose a learning-based framework to predict the Laplacian spectrum of the union of two shapes directly from the spectrum of the individual parts. This enables the aforementioned spectral methods to be applied directly in case of partial views of the same shape, without resorting to methods that explicitly fuse the 3D geometry of partial shapes. Indeed, a typical pipeline to combine partial shapes can be very cumbersome, and requires to match the corresponding regions, extract a set of (non-rigid) transformations from the matches, and merge the partial views into a consistent discretization. Each of these steps can be error-prone and difficult to solve, as testified by a wealth of literature on non-rigid shape matching and reconstruction, especially in the case of partial shapes. For example, the mere presence of inconsistent surface sampling can cause problems in most matching pipelines [174].

Motivated by the excellent results achieved by the methods that exploit the Laplacian spectrum representation, we propose a different perspective. We claim that, in many cases, it is not necessary to have the extrinsic geometry of a target full shape and propose to directly estimate the intrinsic properties of the sought full shape *without* having to materialize its surface geometry. This is done by translating the objective of merging partial shapes from the spatial to a purely *spectral* domain. For each partial surface, our method takes as input the truncated sequence of its Laplacian eigenvalues, which act as a surrogate of the shape geometry, and predicts as output the eigenvalue sequence of the 3D model obtained from the union of the partial surfaces (or an isometric deformation thereof) – but not the 3D model itself, as visualized in Figure 2.1. This eigenvalues prediction task is in general ill-posed, but can be resolved by means of a data prior, namely by training a deep net on a few hundred examples. The advantages of this approach are numerous, and include associativity (i.e. $A \cup (B \cup C) = (A \cup B) \cup C$), invariance to deformations and sampling, and generalization to different discrete representations for the input geometry. In a way, this recalls the notion of "homomorphic encryption" in secure computation [231], where the task is to perform calculations on encrypted data without decrypting it first.

Contribution In this chapter, we introduce a learning-based method to estimate the Laplacian spectrum of the union of partial non-rigid 3D shapes, without actually computing the 3D geometry of the union. Sidestepping the reconstruction means that we do not have to commit to one specific 3D embedding in the output (e.g. a specific pose for a human body), but leave this choice to task-specific blocks. Moreover, our method takes advantage of the geometric insight that the spectra can be used not only for single shape recovery and processing (as done in prior works) but also to enable multi-shape operations such as unions. Once a spectrum is predicted, it can be fed *as-is* to any existing spectral pipeline operating with eigenvalues. For example, we can reconstruct the full 3D geometry by using the method in [168] as an output module. If the geometry is not needed, e.g., for tasks of shape retrieval [228] and region localization [224], we achieve the same accuracy that can be obtained in the case where the full shape is given.

2.2 Related Work

We discuss two lines of research that are most closely related to our spectral aggregation task: partial non-rigid aggregation of shapes in their extrinsic form (e.g, mesh or point cloud), and spectral analysis of partial shapes.

2.2.1 Nonrigid Shape Aggregation

Recovering deformable 3D shapes from partial scans has numerous applications in AR/VR, manufacturing, and robot manipulation. A common setting for this problem is non-rigid registration, where the scans are captured sequentially and exhibit mild inter-frame deformations, and significant overlap. In such cases, template-less methods have been shown to perform well by using general deformation models such as thin-plate splines [40, 41] or as-rigid-as-possible energies [137]. Wand et al. [277, 276] used dynamic "surfels" to represent the input surfaces, and proposed a statistical model to recover the underlying template shape. Temporal coherence has been used in [266] to generate dense correspondences from robust landmarks, and in [181] to reconstruct a space-time surface embedded in 4D. Sharf et al. [244] incorporated a mass conservation prior to control the plausibility of the reconstructed surface. The "Dynamic Fusion" method of Newcomb et al. [190] and follow-up work [250], demonstrated real-time, template-free non-rigid reconstruction allowing both the object and the camera to move.

More related to our setting are cases where the input set is sparser, and the deformations between the scans can change significantly. In fact, we do *not* assume temporal coherence or an initial alignment. Similar to us, methods designed for these settings usually assume a strong prior on the shape category or even rely on a parametric model. In [299], a generic human template was used for building a personalized parametric human body model similar to SCAPE [9]. Chang and Zwicker [46, 47] assumed an articulated model and solved for joints and skinning weights. More recently, advances in geometric deep learning for processing point clouds and meshes were used to leverage data-driven priors (e.g. from [31]) for deformable shape completion and fusion [142, 98].

2.2.2 Eigenvalues and Partiality

Spectral representations based on the Laplacian are widely used in the analysis of deformable shapes, mainly due to their isometric invariance. Much less attention has been given to the effect of partiality on the spectrum.

Shape Correspondence A first attempt at utilizing the Laplacian eigenfunctions to recover dense correspondences between a partial and a full shape was shown in [233], building upon the seminal functional maps framework [202]. This was further extended to matching shapes in the presence of clutter [61], and to a more efficient fully spectral variant in [144]. In the context of partial shape aggregation, most relevant is an extension to the multi-part matching algorithm (a.k.a "non-rigid puzzle") proposed in [145]. Recently, deep learning techniques have also been utilizing Laplacian eigenfunctions for matching [143, 99, 92, 16, 237, 13]. Replacing eigenfunctions with a basis learned from data was recently proven more robust and therefore applicable to challenging settings including point clouds and partiality [167].

Reconstruction Aside from matching, other works have investigated spectral methods for non-rigid completion and registration. In FARM [166] and its high-resolution variant [165], a functional maps representation is incorporated into a parametric model-based regression pipeline.

Shape from Spectrum Most closely related to ours are works that aim directly to recover the shape from its underlying spectrum, also known as the problem of "hearing the shape of a drum" [115]. This procedure was recently studied by Cosmo et al. [60] in practical rather than purely theoretical settings. Their pipeline, dubbed "isospectralization", was proven useful in multiple application scenarios, and extended in [168] by replacing the regularizers of [60] with a data-driven prior.

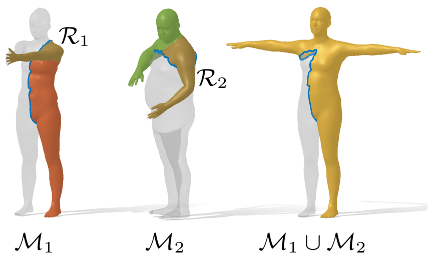
In this work, we aim to perform the union of partial deformable shapes from their spectral representation. Our method differs from the ones listed above for two main reasons: i) we consider the shape from spectrum problem in the more challenging setting of partiality, and ii) rather than recovering the geometry of the full shape, we aim to recover its Laplacian spectrum, given the spectrum of two parts. In other words, we introduce the problem of spectral unions of partial shapes and propose an effective solution. In this light our work is related to [262], which devised a framework to learn fuzzy representations that enable set operations on man-made objects.

2.3 Proposed Method

Let us be given two partial shapes $\mathcal{M}_1$ and $\mathcal{M}_2$, and let $\mathcal{M}_1 \cup \mathcal{M}_2$ denote their non-rigid alignment, as depicted on the left side of Figure 2.1. We seek an answer to the following question: what can we say about $\mathcal{M}_1 \cup \mathcal{M}_2$, without actually computing this union? More specifically: can we predict the spectrum of $\mathcal{M}_1 \cup \mathcal{M}_2$ without having to solve for the point-to-point correspondence between them?

With no additional priors, the question is ill-posed; for example, there are infinitely many ways in which two sheets of paper can be glued together. In the sequel, we claim that the spectrum of the union can be predicted by coupling Laplacian eigenvalues with a data prior without solving a correspondence or reconstruction problem in the process.

Mathematical Preliminaries We model a shape as a Riemannian manifold $\mathcal{M}$ with boundary $\partial\mathcal{M}$. Each manifold identifies an equivalence class of isometries, and thus has infinitely many embeddings in $\mathbb{R}^3$ (e.g. changes in pose). Let us be given two manifolds $\mathcal{M}_1$ and $\mathcal{M}_2$, together with a diffeomorphism $\pi : \mathcal{R}_1 \to \mathcal{R}_2$ between regions $\mathcal{R}_1 \subseteq \mathcal{M}_1$ and $\mathcal{R}_2 \subseteq \mathcal{M}_2$. A third manifold $\mathcal{M}_1 \cup \mathcal{M}_2$ can be obtained by attaching $\mathcal{M}_1$ to $\mathcal{M}_2$ over the common region via the map π (as depicted in the inset figure). We refer to $\mathcal{M}_1 \cup \mathcal{M}_2$ as the *union shape*[1].



[1]We keep the mathematical description simple for the sake of clarity. Formally, this operation is called *connected sum*, denoted by $\mathcal{M}_1 \# \mathcal{M}_2$, and is part of the surgery theory of manifolds, see, e.g., [235].

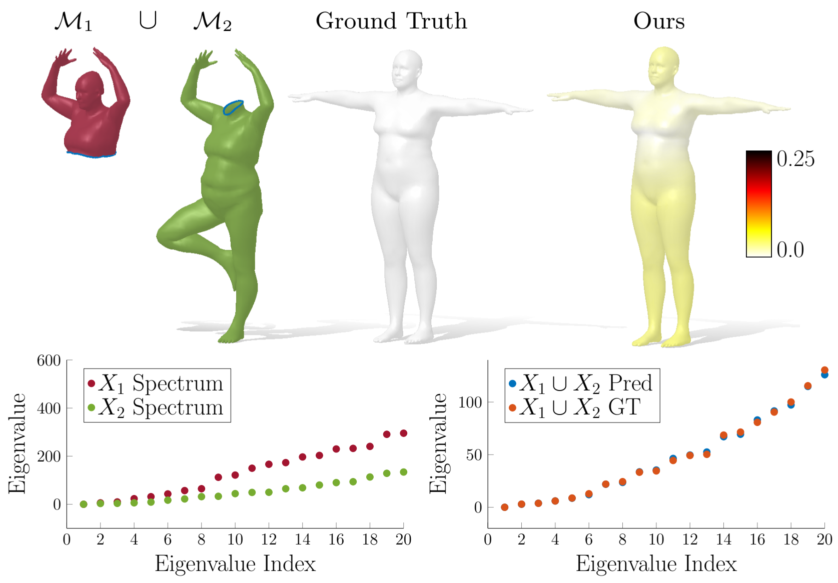


Figure 2.2. Example of partial shapes whose union entirely covers the full shape. This is the simplest setting that we consider in this chapter. Given the spectra of the partial shapes (red and green), we recover the spectrum of their union, and from the spectrum we recover the geometry in standard T-pose using a shape-from-spectrum reconstruction method [168]. The white shape is recovered from ground-truth eigenvalues; ours is colored with a heatmap, which encodes reconstruction error.

On each $\mathcal{M}$ we consider the Laplace-Beltrami operator Δ, extending the notion of Laplace operator from Euclidean geometry to surfaces. This operator admits a spectral decomposition:

$$\Delta\phi_i(x) = \lambda_i\phi_i(x) \qquad x \in \mathcal{M} \setminus \partial\mathcal{M} \tag{2.1}$$

$$\phi_i(x) = 0 \qquad x \in \partial\mathcal{M} \tag{2.2}$$

into eigenvalues $\lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \cdots$ and associated eigenfunctions $\phi_1, \phi_2, \phi_3, \dots$; we adopt homogeneous Dirichlet boundary conditions (2.2). The set of eigenvalues forms a discrete *spectrum*, which we assume to be ordered non-decreasingly. In this chapter, we consider truncated spectra of length k, and introduce the vector-valued function:

$$\boldsymbol{\lambda} : \mathcal{M} \mapsto (\lambda_1, \dots, \lambda_k) \,. \tag{2.3}$$

In particular, we completely discard the eigenfunctions $\phi_1(x), \phi_2(x), \dots$, which are point-based quantities and thus highly dependent on shape discretization.

Remark. Since the Laplacian Δ is invariant to isometries, so is its truncated spectrum encoded in $\boldsymbol{\lambda}$. This means that eigenvalues capture shape information *up to pose*, a fundamental property that is at the basis of our method.

Problem Statement In non-rigid alignment problems, one is given 3D embeddings (e.g. point clouds) for $\mathcal{M}_1$ and $\mathcal{M}_2$, and must recover a 3D embedding of their union $\mathcal{M}_1 \cup \mathcal{M}_2$. Since, in this setting, $\mathcal{M}_1$ and $\mathcal{M}_2$ may undergo wildly different deformations, there is no guarantee that they have the same 3D coordinates on the common region. Therefore, it is not clear how a 3D embedding for $\mathcal{M}_1 \cup \mathcal{M}_2$ should look like.

In our work, we propose to mitigate this problem by switching from a discrete representation of the 3D embedding of $\mathcal{M}_1 \cup \mathcal{M}_2$ to a discrete representation of the entire isometry class, given by $\boldsymbol{\lambda}(\mathcal{M}_1 \cup \mathcal{M}_2)$. Then, we translate the problem of recovering an alignment between 3D embeddings to the estimation of a parametric nonlinear operator $\mathcal{U}_\Theta : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^k$, such that:

$$\boldsymbol{\lambda}(\mathcal{M}_1 \cup \mathcal{M}_2) = \mathcal{U}_\Theta(\boldsymbol{\lambda}(\mathcal{M}_1), \boldsymbol{\lambda}(\mathcal{M}_2))\,. \tag{2.4}$$

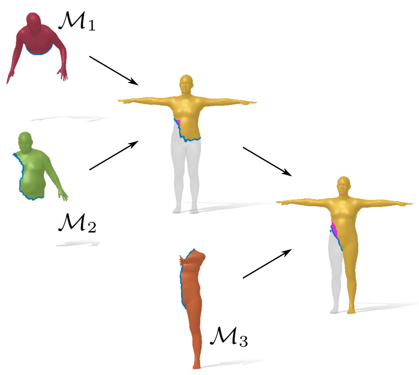
We call $\mathcal{U}_\Theta$ the *spectral union* operator, and model it as a deep neural network with learnable parameters Θ. A specific definition for the architecture and the loss are given in Section 2.4.

Remark. In general, the spectrum of the union shape $\mathcal{M}_1 \cup \mathcal{M}_2$ is *not* simply the union of the spectra of $\mathcal{M}_1, \mathcal{M}_2$. This is only true if $\mathcal{M}_1$ and $\mathcal{M}_2$ correspond to disjoint regions of the complete shape (see e.g. [233, Sec. 3.1]), while in this chapter we consider the case in which $\mathcal{M}_1$ and $\mathcal{M}_2$ partially overlap, thus making the interactions between the two spectra much more complex.

Difficulty Settings and Associativity Estimating an operator $\mathcal{U}_\Theta$ that makes Equation (2.4) hold for many different pairs $(\mathcal{M}_1, \mathcal{M}_2)$ is not a simple problem, even with short sequences (in this chapter we use $k = 20$). In fact, it is known that Laplacian spectra can vary wildly under partiality perturbations [85], and predicting these variations can be difficult.

Based on these observations, we consider two different scenarios with different characteristics:

1. $\mathcal{M}_1 \cup \mathcal{M}_2$ is a complete, watertight shape;
2. $\mathcal{M}_1 \cup \mathcal{M}_2$ is a partial shape itself.



As we demonstrate below, Scenario 1 is simple enough to be solved with a feed-forward network, and generalizes well to unseen data, as shown in Figure 2.2. Scenario 2 is more difficult, since allowing partiality on the union shape introduces another dimension of variability, as well as more ambiguity on the possible output; see Figure 2.3 for examples.

Despite being more difficult to solve, the latter scenario lends itself to modeling more complex interactions. In particular, exploiting the associative property of the union, we can compose $m > 2$ partial shapes simply by aggregating pairwise unions recursively:

$$\boldsymbol{\lambda}(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \cdots \cup \mathcal{M}_m) = \mathcal{U}_\Theta(\boldsymbol{\lambda}(\mathcal{M}_1 \cup \mathcal{M}_2 \cup \cdots \cup \mathcal{M}_{m-1}), \boldsymbol{\lambda}(\mathcal{M}_m))\,. \tag{2.5}$$

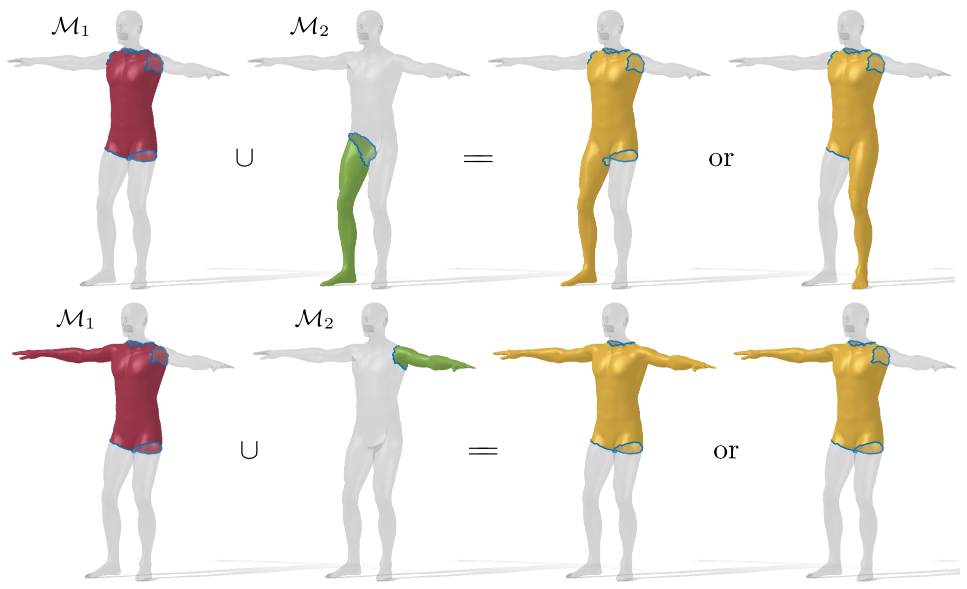


Figure 2.3. Spectra capture isometry classes, thus there exist ambiguous cases where unions have multiple valid solutions. *Top*: The two solutions are isometric, hence intrinsically equivalent. *Bottom*: Since each part is isospectral to its symmetric version, the union of the two spectra can result in three possible solutions (we only show two for simplicity). The semi-transparent full shape is for reference.

See Figure 2.1 and the inset for an illustration. Note that composing m partial shapes resembles the 'non-rigid puzzle' setting seen in [145], although with a crucial difference: the method of [145] has access to the complete shape, which is instead unknown to us.

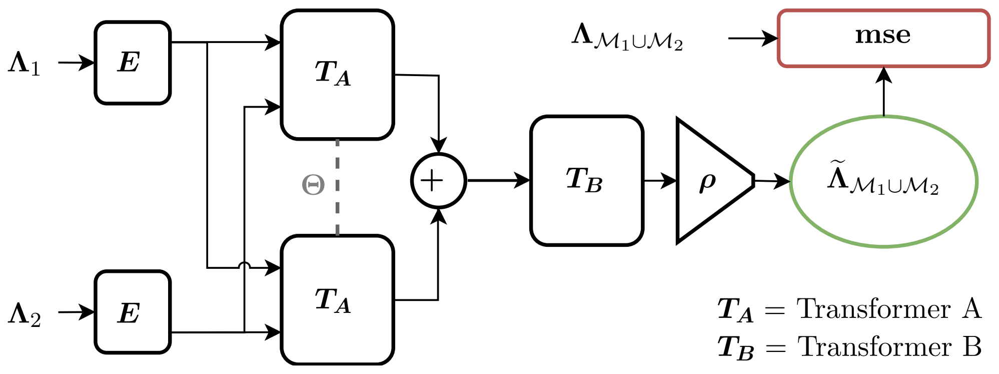
2.4 Network Architecture

Our network takes as input two sequences of k eigenvalues, each associated with a partial shape, and outputs a sequence of k eigenvalues, as a prediction of the spectral union. Figure 2.4 illustrates the neural architecture. It is composed of three main blocks: (1) the projection of the input eigenvalues into a high dimensional space; (2) two transformers, forced to be commutative, to learn the union operation; (3) a dimensionality reduction to decode the spectral union.

Eigenvalue Embeddings The Laplacian eigenvalues of surfaces form a non-decreasing sequence that approximately grows linearly with rate inversely proportional to surface area, a behavior described by Weyl's asymptotic law [286]. This results in the input eigenvalues hugely varying depending on the area of the partial shape. To guard against network instability we encode the spectra via the offset representation:

$$\text{off}(\lambda_i) = \lambda_i - \lambda_{i-1}\,,$$

with $\text{off}(\lambda_1) = \lambda_1$. This representation has the further advantage of imposing the increasing order constraint on the predicted eigenvalues, by just requiring the non-negativity of the predicted offset sequence.



$\mathbf{\Lambda}_1$
$\mathbf{\Lambda}_2$
$\boldsymbol{E}$
$\boldsymbol{E}$
$\boldsymbol{T_A}$
$\boldsymbol{T_A}$
Θ
$+$
$\boldsymbol{T_B}$
$\boldsymbol{\rho}$
$\widetilde{\mathbf{\Lambda}}_{\mathcal{M}_1\cup\mathcal{M}_2}$
$\mathbf{\Lambda}_{\mathcal{M}_1\cup\mathcal{M}_2}$
mse
$\boldsymbol{T_A}$ = Transformer A
$\boldsymbol{T_B}$ = Transformer B

Figure 2.4. Our neural architecture. $\mathbf{\Lambda}_1$ and $\mathbf{\Lambda}_2$ are the input eigenvalues of the partialities, $\boldsymbol{E}$ embeds the eigenvalues into a high dimensional space, the transformer $\boldsymbol{T_A}$ produces a latent representation of the inputs that are summed up to obtain a commutative latent representation of the union, the transformer $\boldsymbol{T_B}$ plus the linear dimensionality reduction $\boldsymbol{\rho}$ decodes this latent representation to obtain the predicted eigenvalues $\widetilde{\mathbf{\Lambda}}_{\mathcal{M}_1\cup\mathcal{M}_2}$.

In practice, the network sees each spectrum as a sequence of length k offsets $\mathbf{\Lambda} = (\text{off}(\lambda_1), \ldots, \text{off}(\lambda_k)) \in \mathbb{R}^k$, each one is then embedded into a higher-dimensional representation of length $2\ell + 1$, constructed as follows:

$$\text{off}(\lambda_i) \mapsto \left(\vec{\theta}_a^{\,i}, \quad \text{off}(\lambda_i)\vec{\theta}_b, \quad \text{off}(\lambda_i)\right) ,$$

where $\vec{\theta}_a^{\,i}$ is a ℓ-dimensional vector acting as a positional encoding for the i-th offset, and $\vec{\theta}_b$ is a linear mapping of the offset to a ℓ-dimensional space. The learnable vectors $\vec{\theta}_a^{\,i}$ and $\vec{\theta}_b$, once learned, are independent from the input shapes and eigenvalues.

This representation encodes both the eigenvalue quantity and its position in the sequence, which is a fundamental information for recovering the geometry area.

Symmetric Architecture Given the eigenvalue sequences of $\mathbf{\Lambda}_1$ and $\mathbf{\Lambda}_2$ (associated to $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively), our neural architecture learns how to perform their union without ever leaving the spectral domain. We further require our model to be commutative, i.e., the result should *not* depend on which pair between $(\mathbf{\Lambda}_1, \mathbf{\Lambda}_2)$ or $(\mathbf{\Lambda}_2, \mathbf{\Lambda}_1)$ is given as input.

We gain this invariance by using a single transformer $\boldsymbol{T_A}$ on the embedded eigenvalues, performing two symmetric operations to obtain a representation of $\mathbf{\Lambda}_1$ informed about $\mathbf{\Lambda}_2$ and vice-versa. The two transformed representations are summed together to obtain a commutative representation of the union. We then feed the result to the second transformer $\boldsymbol{T_B}$, whose task is to decode the union into a representation that can be easily reduced, via a simple linear layer $\boldsymbol{\rho}$, from the high-dimensional representation back to a sequence of eigenvalues. The whole architecture is illustrated in Figure 2.4.

The transformers are position-aware neural networks, where the output for each eigenvalue depends on its value and position together with all the other eigenvalues and their positions. It employs an *attention* mechanism to learn relation among eigenvalues.

In the network, the dimensionality of each representation is 32, $\boldsymbol{T_A}$ has 8 heads and 6 layers meanwhile $\boldsymbol{T_B}$ has 8 heads and 3 layers. Thus, $\boldsymbol{\rho}$ reduces the representation dimensionality from 32 to 1.

Training Our model is trained with a mean squared error loss between the predicted and ground truth spectra. Before entering the loss, the offset representation for the eigenvalues is decoded with a cumulative sum. Experimental results show that penalizing the loss according to the linear increase of the eigenvalues does not yield significant improvements. In the training phase, we augmented the partial regions with small random changes in their surface area. The optimizer used is Adam with a learning rate of $2 \cdot 10^{-4}$ and weight decay of 10^{-5}. We use a learning rate scheduler to escape local minima and stabilize the training, in particular the cosine annealing with warm restarts scheduler [152], doubling at each restart the number of epochs between restarts. We trained the model for 6741 epochs for a total of 1d 13h 46m on a GeForce RTX 2080 Ti, tracking the experiments with [29].

2.5 Data and Evaluation

In our experiments we use 3D data from the FAUST [30] and SURREAL [275] datasets of deformable human shapes with different identities. This provides us with a total of 50 different identities, each in 10 different poses. To produce partial data, we first extract surface patches of various sizes from the full shapes, and then combine the patches randomly to form two datasets:

- A dataset of ∼150 partial pairs, where each union covers the entire surface. We test in three different settings depending on the information given at training time: (i) known identity, unknown partiality; (ii) unknown identity, known partiality; (iii) both identity and partiality are unknown. We define an identity as *known* if the training set contains any partiality in any pose of the same shape, and we consider a partiality *known* if the two input partiality types together with their corresponding union are in the training dataset in any shape identity or pose.
- A dataset of ∼100 partial pairs, whose union does not cover the entire surface. For training, the partial shapes are augmented by enlarging/shrinking the patches randomly. We consider the same three settings as above.

As shown in Figure 2.3, there are cases in which more than one region on the template is a valid solution to the union problem. Two different ambiguities arise: (a) symmetric counterparts of one or both input partial regions may produce different union regions with different spectra; (b) symmetric union regions are described by the same spectra even though they are localized in different parts of the shape. We remove these ambiguities in the training data by following a minimum union area principle and privileging "left-sided" symmetries exploiting a ground truth symmetry map and labels of the template left side. By this choice, associativity is promoted as we show empirically in the results.

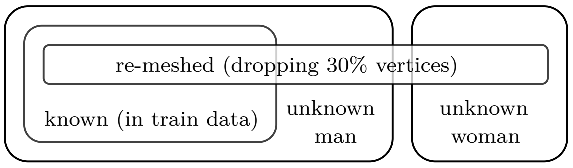
We define two test sets. In Test A, the pose or the type of partiality have never been seen, but the predicted union may be seen in a different pose or identity

		mse	mae
TEST A	known man	**11.14**	**2.09**
	unknown man	13.25	2.59
	unknown woman	36.92	3.93
	known man re-meshed	29.61	3.31
	unknown man re-meshed	32.67	3.60
	unknown woman re-meshed	62.33	5.23
TEST B	known man	15.41	2.59
	unknown man	24.05	3.60
	unknown woman	64.47	4.99
	known man re-meshed	51.20	4.54
	unknown man re-meshed	75.91	5.90
	unknown woman re-meshed	**110.17**	**6.78**

Table 2.1. Error between the predicted and ground truth eigenvalues in different experimental settings. In each row, "known" denotes an identity included in the training set, "unknown" one not included, and "re-meshed" indicates that the shapes were re-meshed by removing 30% of their vertices before computing their spectrum.

at training time. TEST B is more challenging, since the union of the two parts has never been seen at training (neither in a different pose or identity, nor as a union of different partialities). The number of samples in the test datasets is about 15% of all the data available, the remaining data is used for training.

We analyze both TEST A and TEST B scenarios in different settings summarized in the inset Venn diagram.



In Table 2.1 we report a quantitative analysis of the predictive power of our learning model, according to the mean squared error (mse) and mean absolute error (mae) metrics.

Further, we perform qualitative experiments on different classes, on horses from TOSCA [38] and earphones from PartNet [182], some examples are in Figure 2.7, 2.8 and 2.9. Through these experiments we prove our method generalization ability to any shape category. Additionally, we successfully trained the neural network on humans and fine-tuned it to work with horses, where the data is scarce, demonstrating that it is possible to perform transfer learning between different shape classes.

2.6 Applications

We can easily plug our method into existing pipelines that take as input Laplacian eigenvalues. Unique to our approach is that it addresses the scenario in which only partial views of the complete shape are available.

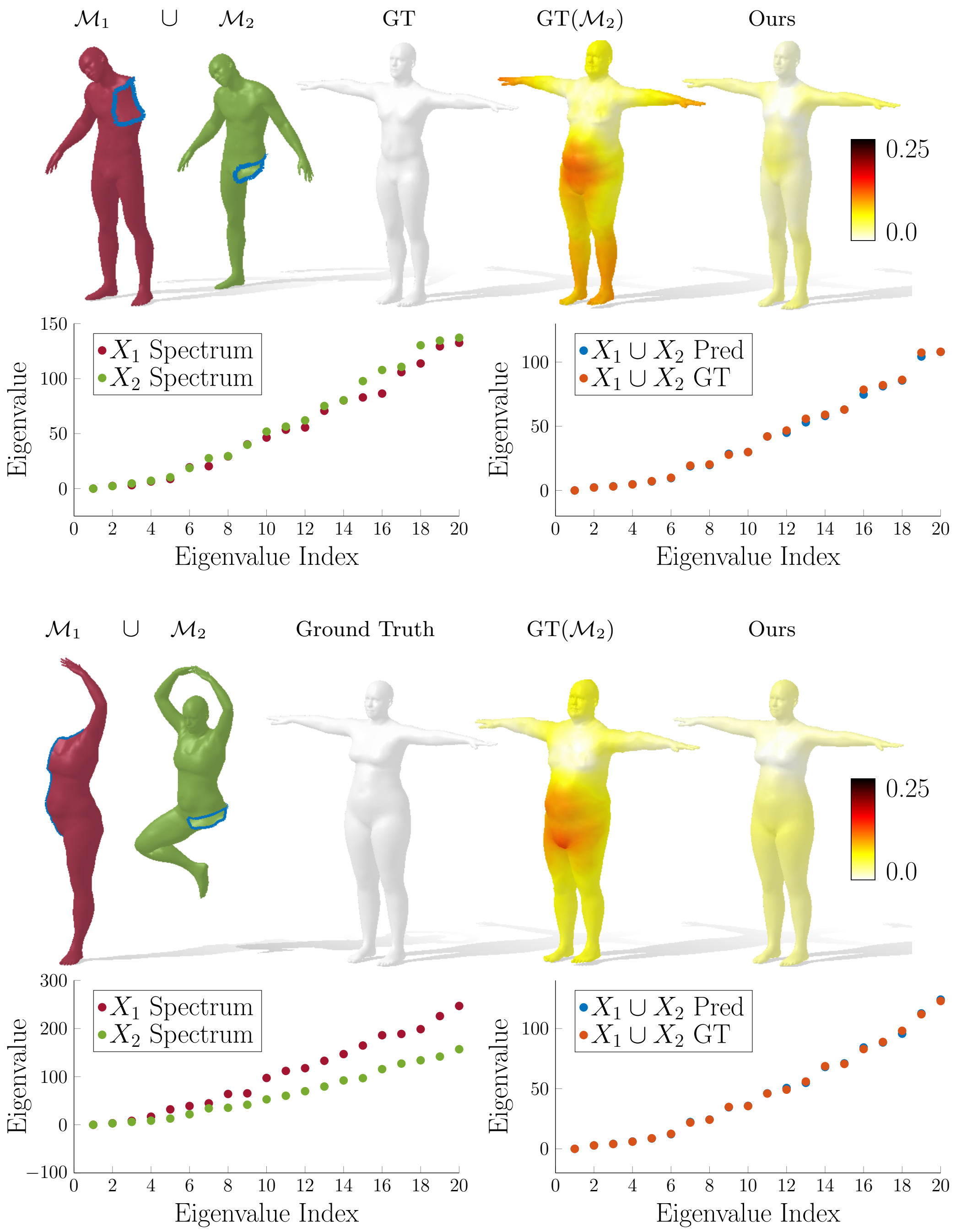


Figure 2.5. Given two partial shapes as input, we compare the reconstruction obtained by running the method of [168] only on a partial input (the green shape), yielding the fourth shape, with the reconstruction obtained from our predicted full spectrum, yielding the last shape.

		IoU	Acc.
Test A	known man	**99.28%**	**99.61%**
	unknown man	93.78%	95.83%
	unknown woman	94.19%	96.32%
	known man re-meshed	98.54%	99.06%
	unknown man re-meshed	91.44%	94.08%
	unknown woman re-meshed	93.47%	95.56%
Test B	known man	97.96%	98.55%
	unknown man	87.58%	92.52%
	unknown woman	96.05%	98.46%
	known man re-meshed	93.04%	97.33%
	unknown man re-meshed	**83.69%**	**91.08%**
	unknown woman re-meshed	95.59%	98.43%

Table 2.2. Intersection over union (IoU) and accuracy in the region localization task, in different experimental settings. Model trained on a single identity, to show generalization.

2.6.1 Geometry Reconstruction

To recover the shape geometry from its predicted union eigenvalues, we use the data-driven method of [168], which takes eigenvalues as input and directly yields a 3D mesh embedding as output. An example is given in Figure 2.2, where we compare the geometry recovered from our estimated spectra with the one obtained from the ground truth eigenvalues. For human meshes where the correspondence between their T-Pose and the connectivity adopted in [168] is known, we can compute the point-wise reconstruction error as the L2 distances between correspondent points. We plot this Euclidean error on the reconstructed surface. White color corresponds to zero error and dark red encodes a larger error. Our spectrum prediction is accurate enough to retain the core geometric information of the original partial shapes, as it can be seen in these examples. For these experiments we used the pre-trained network provided by the authors of [168] and [169]. We sampled the test shapes outside the training set adopted in these papers. Thus the network is not specifically trained to handle spectra predicted by our pipeline. Moreover, since the spectrum encodes just intrinsic properties (i.e. appearance) of the shape, all the reconstructions of [168] are in the T-pose.

To emphasize the importance of having an aggregated spectrum, as predicted by our model, in Figure 2.5 we show the reconstructions obtained with the method of [168] when using the spectrum of just one of the two partial shapes as input. The result in this case is quite different from what is expected, showing that existing state-of-the-art pipelines are not able to handle partial shapes correctly.

2.6.2 Region Localization

This task, introduced in [224], consists in locating, on a fixed template, the region corresponding to a given partial shape. To solve this problem we combine the spectral union model introduced in Section 2.4 with a simple MLP, described in detail

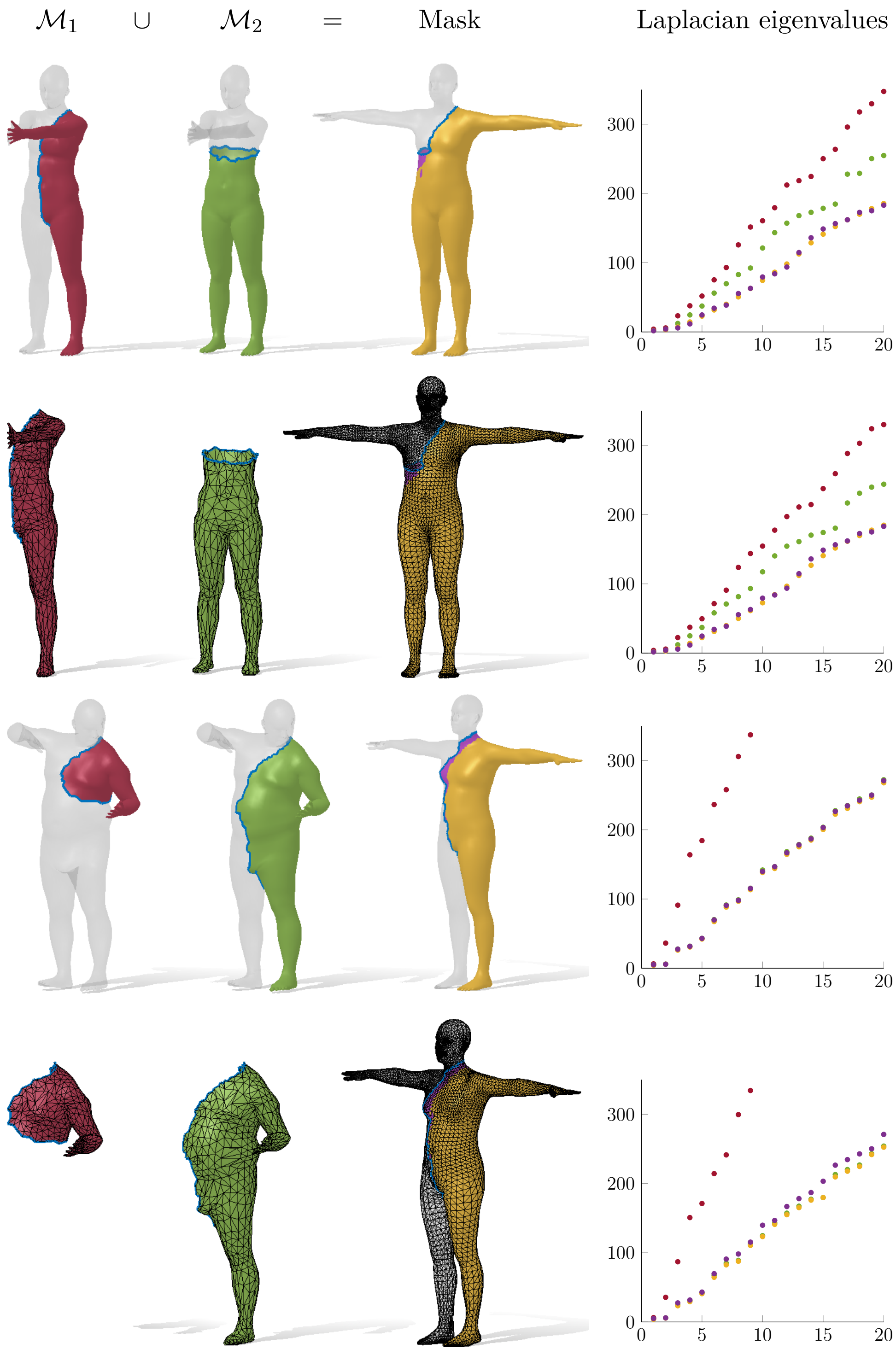


Figure 2.6. Region localization task, under the effect of different mesh connectivity. Given the eigenvalues of two partial shapes, we correctly predict an indicator function that represents the union of the two over a fixed template.

		IoU	Acc.
Test A	known man	**98.24%**	**99.09%**
	unknown man	96.26%	97.64%
	unknown woman	96.17%	98.04%
	known man re-meshed	97.70%	98.74%
	unknown man re-meshed	95.88 %	97.78%
	unknown woman re-meshed	96.04%	97.66%
Test B	known man	97.43%	99.14%
	unknown man	93.31%	98.23%
	unknown woman	95.74%	98.59%
	known man re-meshed	97.61%	99.11%
	unknown man re-meshed	**90.85 %**	**97.63%**
	unknown woman re-meshed	96.81%	98.98%

Table 2.3. Performance when training on six different identities instead of a single identity (compare with Table 2.2).

in the supplementary materials. The MLP takes as input the predicted eigenvalues of the union, and outputs an indicator function over the vertices of the template. The spectral union operator is not trained to solve the region localization but is used as-is with frozen weights.

In principle, substituting $\boldsymbol{T_B}$ in Figure 2.4 with the region localization MLP would work if the whole system is trained end-to-end. However, the goal of this work is to perform the union operation in the spectral space. Moreover, if we do not impose the union to be a spectrum, we would not be able to compose the predicted union spectra with another partiality.

In the loss definition, one must take care of the potential ambiguities exemplified in Figure 2.3; we do so by implementing a symmetry-invariant loss, that does not penalize symmetric solutions. The MLP is trained using the train/test splits described in Section 2.4, with the difference that we used just 6 different identities in the training phase.

To analyze the prediction quality on this task we adopt two metrics: intersection over union (IoU) of the predicted mask with the ground truth mask, and accuracy, i.e. the ratio of correctly predicted vertices over the full template. We show several qualitative results in Figure 2.6 and attach an interactive demo in the supplementary materials.

Robustness to Remeshing One key aspect of Laplacian eigenvalues is that they are robust to shape discretization and mesh connectivity. Our model inherits this robustness; see Figure 2.6, where we highlight the re-meshed inputs by visualizing their surface triangulation. This is supported also by Tables 2.2 and 2.3, where the performance on the re-meshed shapes is comparable with the original ones. In these experiments, we test our network with the eigenvalues computed from noisy, re-meshed partial shapes obtained by removing 30% of their vertices with an edge collapse algorithm [90].

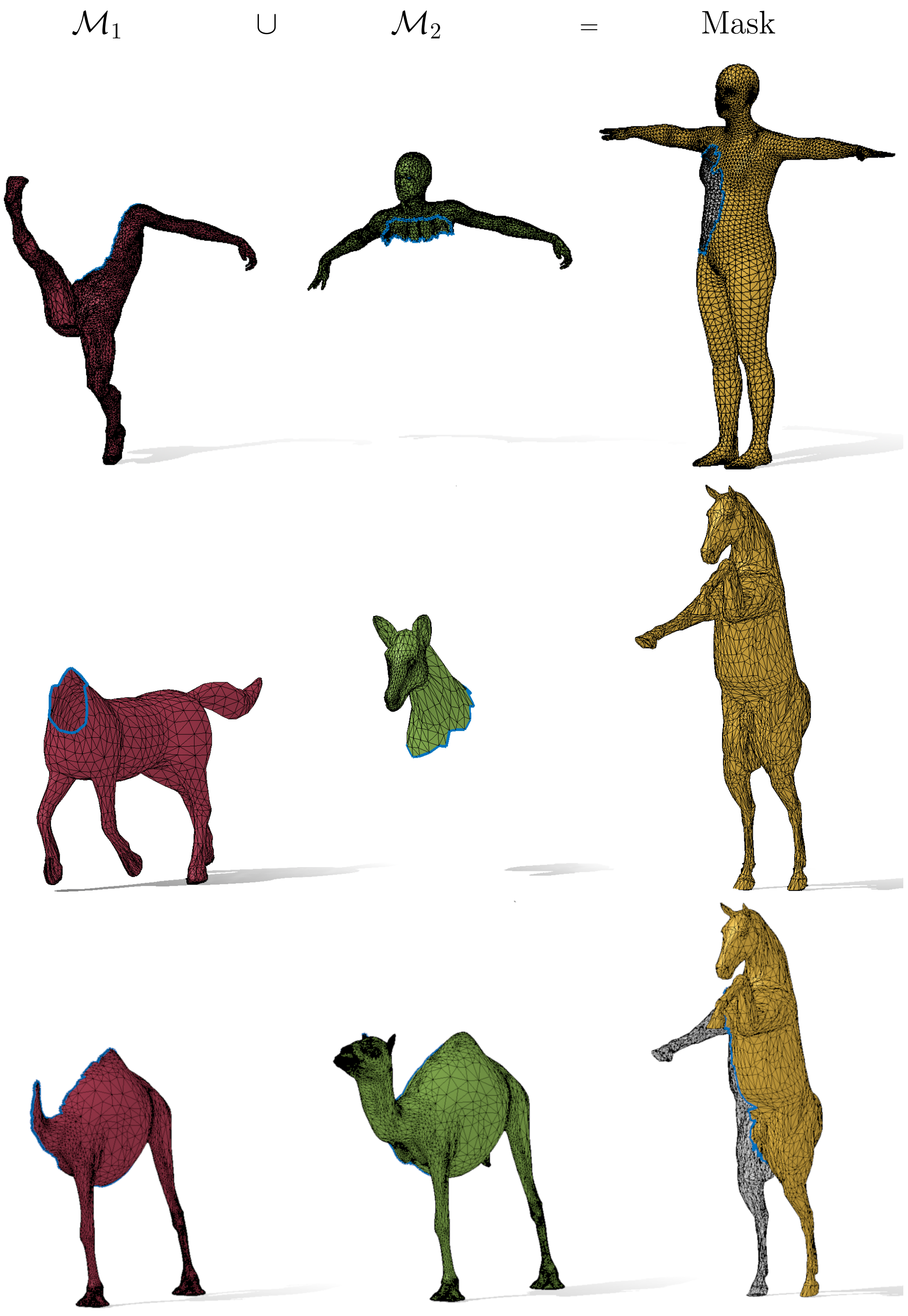


Figure 2.7. Region localization across different datasets. Partial shapes come from datasets not involved in the training.

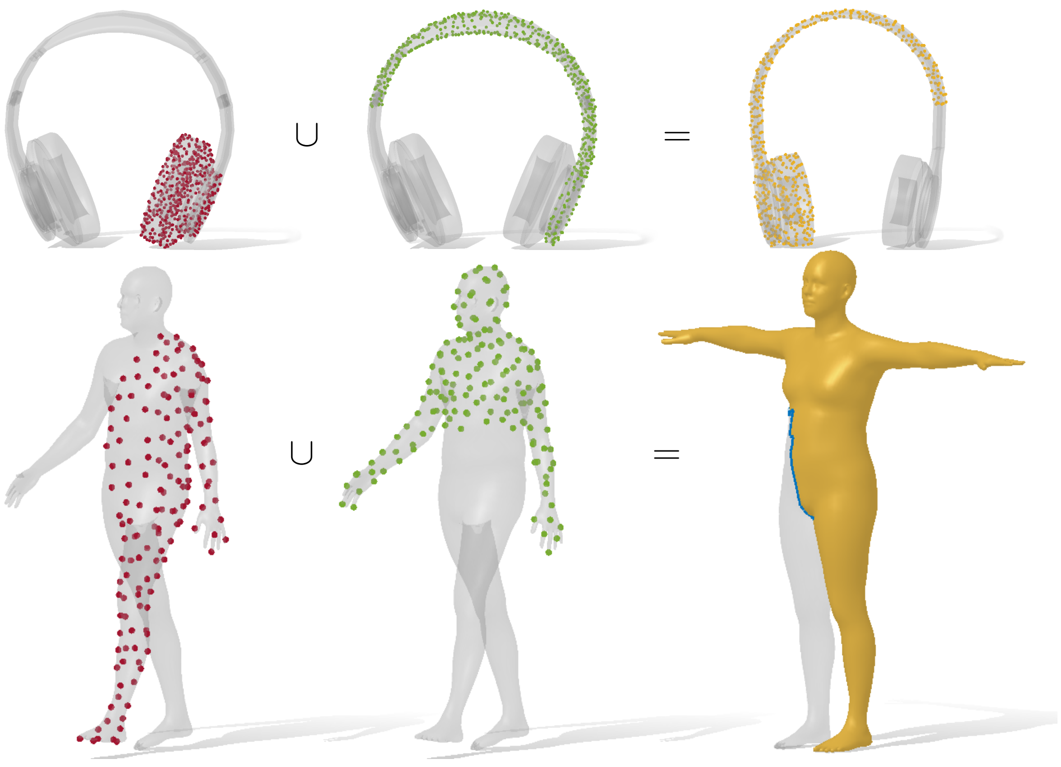


Figure 2.8. Region localization from partial point cloud spectra. The white mesh is just shown as a visual reference.

Generalization to New Identities Our approach generalizes to identities unseen at training time as can be noted in Table 2.3. To further stress this aspect, we devised an experimental setup in which we used as training set just a single identity. The results of this setup are shown in Table 2.2.

Generalization to Different Datasets In Figure 2.7 we use partial shapes from other datasets to localize regions on the fixed template. These shapes have different triangulation, vertex density and style, confirming generalization across datasets. More specifically: a shape from TOSCA [38] for humans (first row), one from SMAL [304] for the horses (second row) and a camel shape that has a different triangulation and comes from a different class (third row).

Generalization to Point Clouds We obtain good results also on point clouds, as shown in Figure 2.8. For earphones, in the top row, we perform both training and testing on point clouds. In the bottom row, we show that our model trained on human meshes generalizes to point clouds. We compute the Laplacian for point clouds with the method of [245].

Associativity We can compute spectral unions of > 2 partial shapes iteratively as described in Section 2.3. In Figure 2.9 we show qualitative results over three parts.

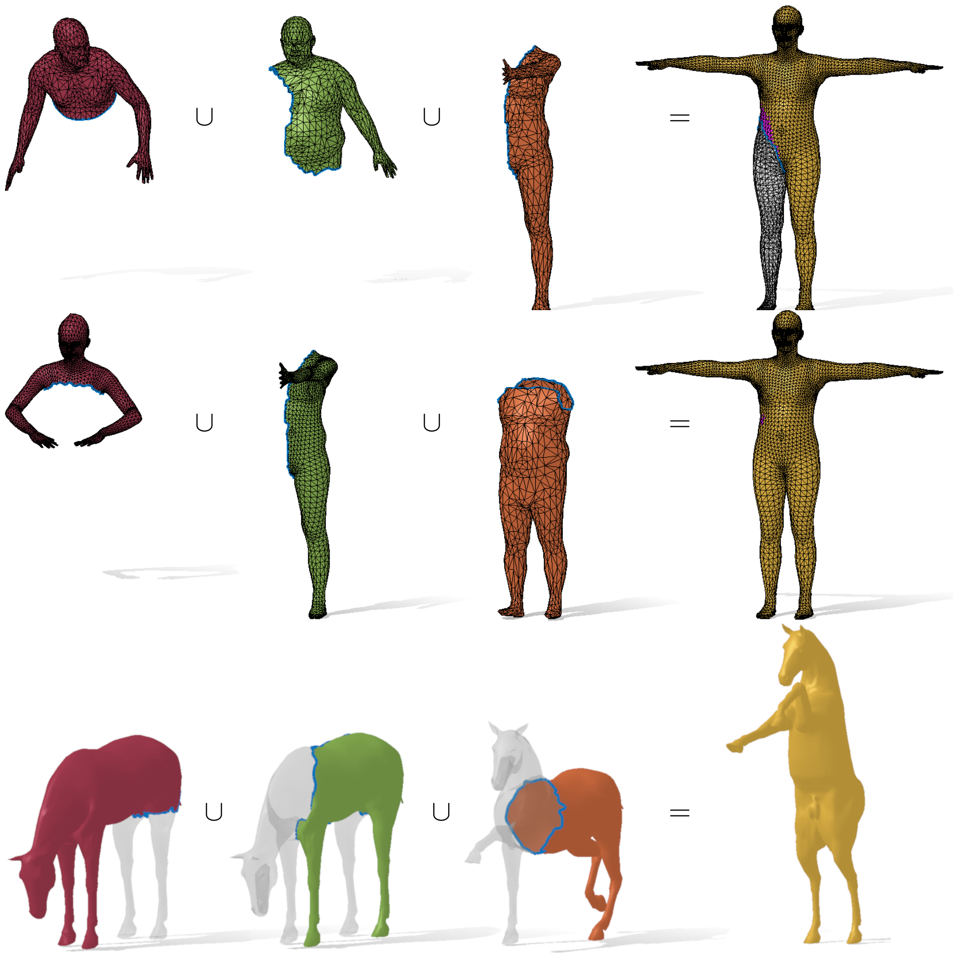


Figure 2.9. Example of associativity. Note that all the human meshes involved have different connectivity.

Interpolation Finally, in Figure 2.10 we first interpolate the spectra of two partial shapes (in green), and then compute the union of the interpolated spectra with the spectrum of a fixed shape (in red). From each of these unions, we predict a mask on the given template (in yellow). We can see how in the first example (top row) the mask changes smoothly. On the other hand, in the second example it is less obvious how to interpolate the completely missing leg, resulting in an abrupt discontinuity in the predicted mask.

	top-1	top-5	top-10
Ours	86.14%	**97.75%**	**99.20%**
ShapeDNA	**86.59%**	96.81%	97.72%

Table 2.4. Comparisons on the shape retrieval task.

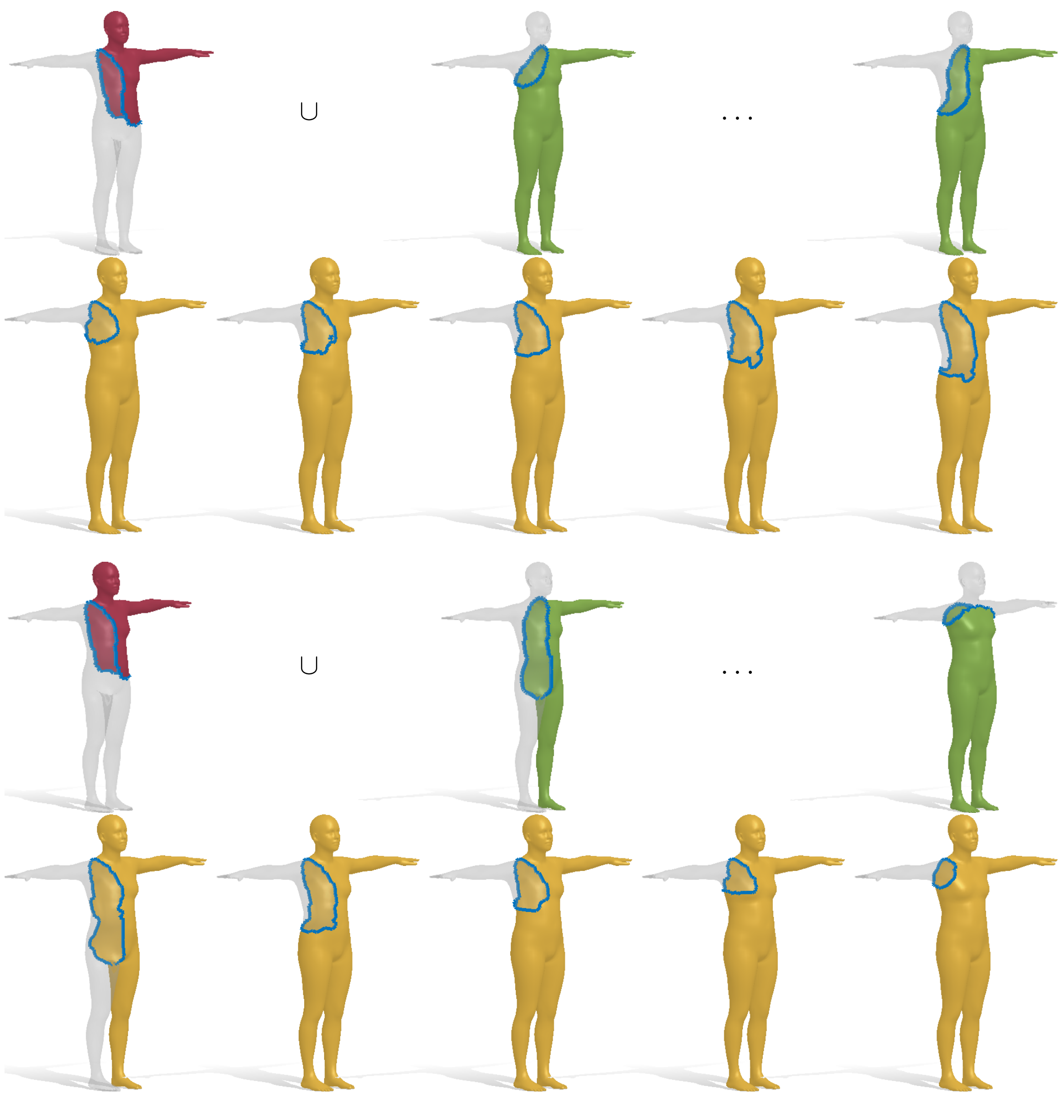


Figure 2.10. Two examples of linear interpolation of eigenvalues (green shapes), and the resulting predicted masks (in yellow). Please refer to the main text for details.

2.6.3 Shape Retrieval

This task consists in retrieving a query from a database of shapes that could undergo several deformations. A well-known spectral method to tackle this problem, ShapeDNA [228], adopts the Laplacian spectrum as a shape signature. In the space of these signatures, nearest-neighbor search yields the desired result. However, in order to work correctly, ShapeDNA needs the spectrum of a complete shape; extensions of this signature to the partial case have proven unsuccessful to date [234]. Our method applies directly to this case, since we can estimate the eigenvalues of the unknown complete shape whenever the input query is just a collection of its partial views.

We run our tests on a dataset of 440 complete shapes (44 identities in 10 poses each). For our method, we evaluate 4400 pairs of partial shapes; for each pair we predict the ShapeDNA signature of their union and use it to query the database. We compare it with the accuracy obtained by standard ShapeDNA on each of the 440 complete shapes in the database, which assumes exact knowledge of the union spectra; nevertheless, the identity it retrieves may be wrong due to spectra variations caused by deformations between different poses. We measure the performance using top-k metrics, which count the number of times a shape with the correct identity is in the first k retrieved shapes; we use $k = 1, 5, 10$. The results are reported in Table 2.4, and show that our predicted eigenvalues are accurate enough to compete with, and even surpass, ShapeDNA for this task. The better performance is due to the robustness of our method to the noise induced by the pose change.

2.7 Conclusion

We introduced a method to recover the aggregated Laplacian spectrum of a collection of partial deformable shapes, while avoiding the computational burden of computing correspondences or extrinsic alignments. Our method involves a deep net that, given two eigenvalue sequences as input, simply produces another eigenvalue sequence as output. In spite of its apparent simplicity, this method allows to address a number of applications that traditionally require solving for a correspondence, and retains a comparable quality (in some cases, even higher) to methods that have direct access to the 3D geometry of the full shape.

Limitations and Future Directions Perhaps the main limitation of our method lies in the missing mathematical guarantee that our predicted sequences are actual Laplacian eigenvalues, despite our positive empirical results. We consider enforcing this constraint as an interesting direction of further research. Another interesting area for improvement is the region localization generalization capability, where our current model seems to struggle with out-of-distribution union partialities. We are optimistic that a more diverse and extensive training set would boost the generalization performance. Moreover, we did not consider unprocessed partial single-view or depth scans of physical objects. We expect a drop in performance on such data comparable to other spectral methods. We consider improving the robustness of spectral methods on natural non-pre-processed data as an essential and challenging research direction.

Part II

Non-Euclidean Metric Spaces

Chapter 3

Mold Manifold Simulation for Real-Time Procedural Texturing

The slime mold algorithm has recently been under the spotlight thanks to its compelling properties studied across many disciplines like biology, computation theory, and artificial intelligence. However, existing implementations act only on planar surfaces, and no adaptation to arbitrary surfaces is available. Inspired by this gap, we propose a novel characterization of the mold algorithm to work on arbitrary curved surfaces. Our algorithm is easily parallelizable on GPUs and allows to model the evolution of millions of agents in real-time over surface meshes with several thousand triangles, while keeping the simplicity proper of the slime paradigm. We perform a comprehensive set of experiments, providing insights on stability, behavior, and sensibility to various design choices. We characterize a broad collection of behaviors with a limited set of controllable and interpretable parameters, enabling a novel family of heterogeneous and high-quality procedural textures. The appearance and complexity of these patterns are well-suited to diverse materials and scopes, and we add another layer of generalization by allowing different mold species to compete and interact in parallel. The work presented in this chapter has been realized in collaboration with Riccardo Marin (Ph.D.), who helped in designing the experimental validation and the presentation of the results, prof. Simone Melzi and prof. Emanuele Rodolà, both of which invaluably helped in the improvement of the overall presentation, with a particular focus on the methodological discussion. The results presented in this chapter have been published in the proceedings of the *Pacific Conference on Computer Graphics and Applications* (*Pacific Graphics*) [160].

3.1 Introduction

In different disciplines, the interest in biological evolutive systems has grown in recent years. Biology and chemistry scholars and, more recently also bioinformatics, artificial intelligence, and computation theory researchers are focusing on such complex systems. More specifically, systems that can produce a consistent global behavior by a few local rules are compelling due to their simplicity but powerful

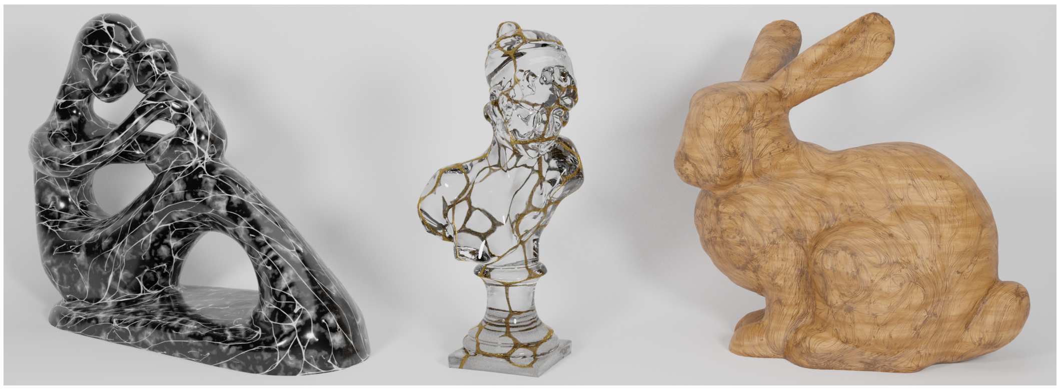


Figure 3.1. Different materials generated from the simulation of our slime approach at a certain frame: marble and wooden grain (left and right respectively), or golden web on a glass structure (middle).

expressive capabilities. Slime mold systems are included in this family [114]. Biological mold organisms propagate following pheromones attraction – from a computer scientist's perspective, perfectly fitting the divide-and-conquer paradigm. These properties have motivated researchers to simulate their evolution, exploiting modern computational capabilities to explore and investigate their properties. Among the available solutions, no method offers an implementation on arbitrary surfaces to the best of our knowledge. We believe this is not due to a lack of interest in non-Euclidean domains, but rather because these dynamic simulations on curved surfaces require a thorough background in differential geometry, not common in the biological community.

From a Computer Graphics perspective, producing complex patterns over a surface belongs to the procedural texturing domain. Despite the increasing availability of computational power and advances in geometry processing, texture design is still a human-centered and time-consuming task for the most. The growing amount and quality of geometrical assets urge the generation of realistic synthetic textures, while guaranteeing the quality and efficiency required by the entertainment industry.

In this chapter, we present a high-quality, real-time implementation of a slime mold algorithm for arbitrary surfaces with potentially millions of agents running in parallel. We design it led by simplicity, efficiency, and generality principles. We introduce only the necessary technicalities, accepting approximations that keep the method simple but are not visually harmful to the final quality. Our method entirely takes place over the surface and offers a set of parameters that artists can easily interpret, so as to produce various distinguishable patterns. These can be used for modeling different materials, as shown in Figure 3.1. We further show how to constrain the mold evolution to specific regions by defining repulsion / attraction areas.

Our contribution can be summarized as follows:

- we provide the first slime mold algorithm for surfaces, with an analysis of its behavior; we show that it is predictable, and respects the expected properties of this kind of organisms;

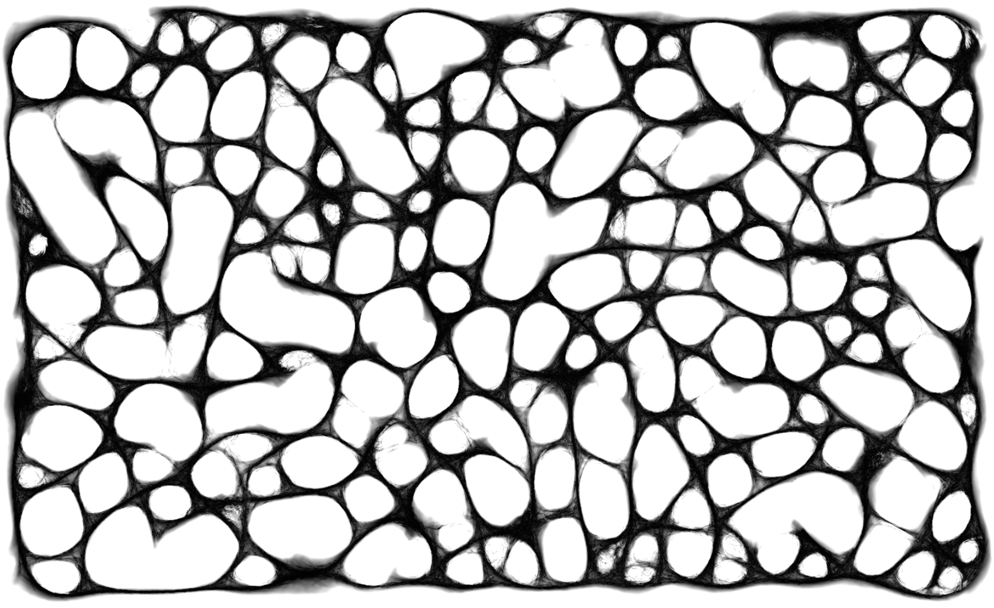


Figure 3.2. Visualization of the pheromone trace in a slime mold simulation. The simulation involves 1M agents and takes place on a bounded flat region.

- we define a new family of patterns for procedural texturing that is interpretable, controllable and admitting path constraints in the pattern evolution;
- we release a light-speed implementation that scales well at different texture resolutions, number of agents, and mesh resolution, opening to real-time video applications and massive texture generation.

3.1.1 Related Work and Background

Slime Mold Simulation The evolution of biological patterns is a vast research area in biology, and it often produces interesting visualizations that can also be used in movies and generative art. A famous example of pattern formation is the evolution of the *Physarum Polycephalum*, also known as slime mold. The algorithm first presented in [114] produces complex patterns like the one shown in Figure 3.2. In recent years, the slime mold simulation has become popular because of its large area of application, which covers cognition, optimization, computation and machine learning [273, 42, 79, 139, 2]. The slime mold algorithm is also starting to be used in generative art [171] and visualization techniques [80]. The algorithm for simulating slime molds has proven to be so valuable that some researchers are starting to integrate it in larger pipelines to achieve compelling results [300, 1].

Slime simulation falls under the category of systems that can produce complex behaviors from a set of simple rules, like cellular automata simulation [53] and reaction-diffusion systems [291].

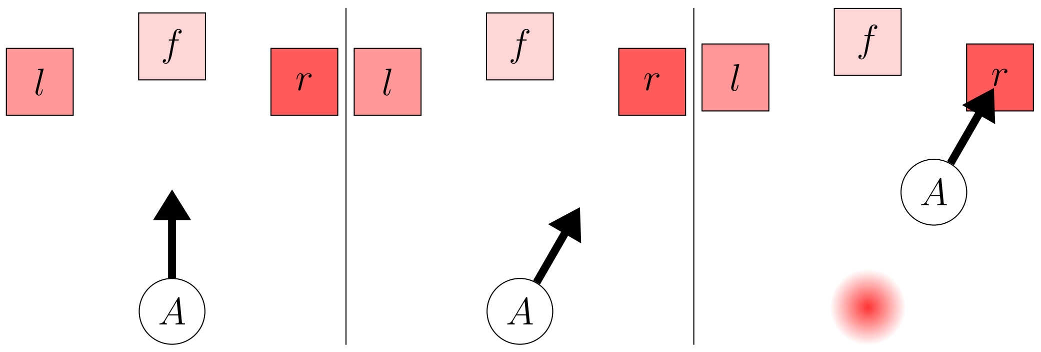


Figure 3.3. The behavior of a slime agent A. The agent samples from three sensors in front of it (left l, right r, and forward f) and determines the next direction by choosing the one with highest pheromone concentration (darker red). When leaving a location, the agent releases pheromone.

The rules of the system can be summarized in three steps:

- A set of agents lives in a closed space where they can move freely;
- Each agent releases a pheromone trace that diffuses and evaporates over time;
- Each agent tries to follow the pheromone trace in its field of view.

Figure 3.3 summarizes how agents act during an iteration. Each agent samples three regions in front of it at an angle (forward, left, or right sensor). The region with the most pheromone traces determines the steering direction of the agent. While moving, the agent releases more pheromone, which in turn diffuses locally and evaporates over time. The simulation can be tuned according to several parameters affecting the agents and the ambient.

Despite the increasing interest in this argument, the research in this direction has been limited to tuning and optimizing the slime mold algorithm, or to its application in different areas.

Procedural Texturing The main works in this context are devoted to procedural generation of height maps for landscapes [132, 200, 208], patterns for planar surfaces [291], 3D noise functions [58, 101] or mixes of other textures [77]. Other works have proposed to use procedural texturing for vector field visualization [22] or the representation of complex repetitive geometries [191], but still, these works are limited to flat domains. Texture synthesis on surfaces has been first addressed in [285], and during the years the research greatly advanced; see [287] for an extensive survey. However, the works in this area mainly exploit example-based methods, adapting them to curved domains [131], or they are limited to simple patterns [124]. The only few exceptions are simulation based approaches [269, 253], which makes them similar in spirit to our method. Only recently, some works have shown procedural generation of complex patterns directly on surfaces [188, 84], but these works are limited to recursive structures or repetitive patterns and do not explore other possibilities. Another work following a similar fashion is [162], where the authors propose a method for drawing Bézier curves on manifold meshes. Still, such curves require a set of input points from the user, which is unfeasible for creating large and complex patterns.

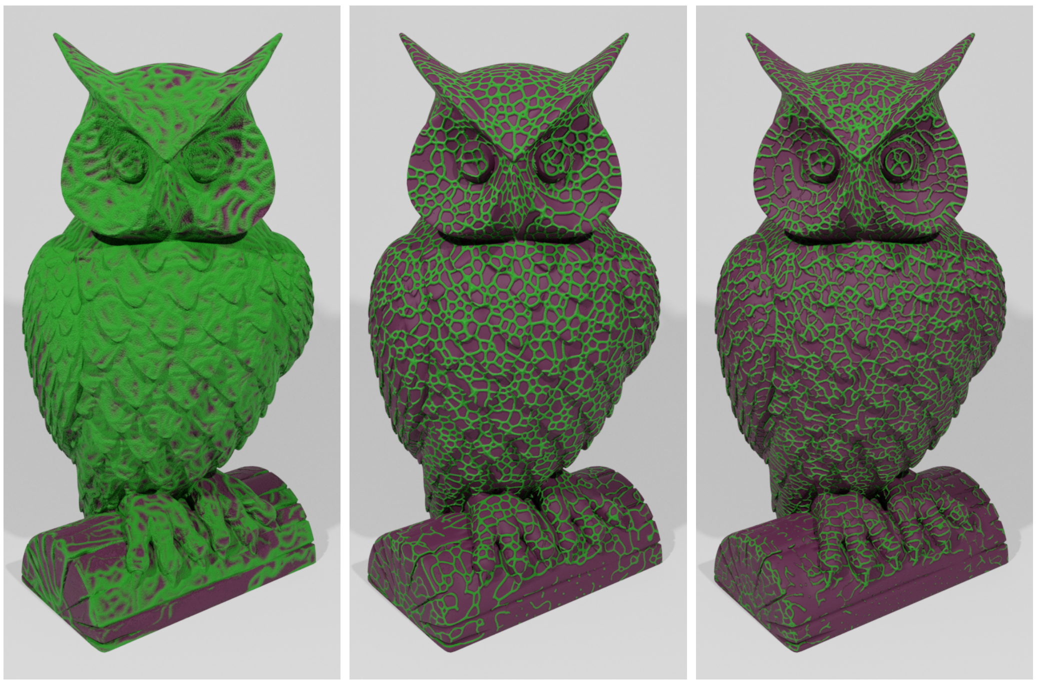
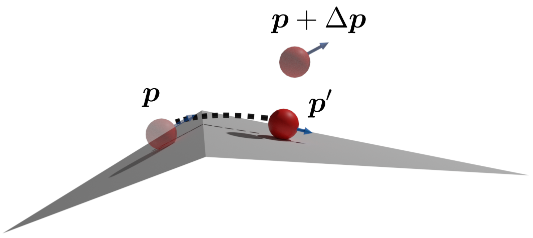


Figure 3.4. Effect of sub-step subdivision on the slime mold simulation. We use the same simulation parameters in all three cases, but vary the number of sub-steps. Left to right: single step, 10 sub-steps, 100 sub-steps.

3.2 Method

In our setting, an agent **A** is a particle on $\mathcal{M}$ that moves over the surface following slime simulation rules. We require that our method: i) works on general meshes and topologies; ii) allows **A** to move on arbitrary paths in real time; iii) guarantees numerical stability. This section provides a detailed description of how we define our method to achieve such desiderata.

3.2.1 Movement Over the Surface



The motion of a particle over a surface is a well-known problem. A widely used solution is triangle unfolding [248, 247], but to keep the implementation simple we decide to approximate the path by moving the particle in small steps and reprojecting it on the surface.

Given the initial position $\boldsymbol{p}$ of **A**, let t be the triangle that contains $\boldsymbol{p}$, and let $\Delta\boldsymbol{p}$ be the vector encoding a forward step. We split $\Delta\boldsymbol{p}$ into smaller sub-steps; at each sub-step, we re-project **A**'s position onto the surface. To ease the projection computation, we consider only the triangle t' that is the nearest to $\boldsymbol{p} + \Delta\boldsymbol{p}$ and incident to t (see inset). This movement parcellation limits inconsistency produced by edge crossing and guarantees we cannot cover more than one triangle at a time. Alternatively, one could compute exactly when an edge has been crossed and split the movement perfectly between the incident triangles. However, we argue that our approach does not require any check, and the approximation produced is negligible. Our sub-step strategy is crucial, as can be seen in Figure 3.4; the leftmost owl does not use any sub-step subdivision, resulting in an unrecognizable pattern. While increasing the number of sub-steps produces a more precise approximation of the motion, we stress that the resulting path does not converge to a geodesic. In fact, slime agents move along a constant direction in the parametric space, which does not necessarily coincide as moving along a geodesic.

Algorithm 3.1 summarizes the entire process. The procedure TangentSubStep is given a point $\boldsymbol{p}$ in texture space and the triangle t where $\boldsymbol{p}$ lies. It computes the movement over the tangent space of the mesh along some vector $\Delta\boldsymbol{p}$. The vector is still defined in texture space, and thus must be rescaled according to the local metric tensor $\mathbf{g}(t)$ (Line 3). The final position after the movement is converted to barycentric coordinates to determine if the border of the triangle has been crossed (Lines 5-10). The adjacent triangle w closest to the final point is selected and the point is projected on it (Lines 12 and 13). Finally, the coordinates are converted to 2D via barycentric coordinates (Lines 15 and 16). To handle meshes with boundaries, it is sufficient to identify if the agent crossed a boundary edge (*i.e.*, there is no adjacent triangle on that edge). If this is the case, we make the agent bounce on the edge.

The vectors $\boldsymbol{n}_\tau$ and $\boldsymbol{c}_\tau$ are, respectively, the normal and the barycenter of a triangle τ, while $\mathrm{Adj}(\tau)$ is the set of its adjacent triangles. The matrices $\mathbf{T}_\tau$ and $\mathbf{L}_\tau$ are defined for each triangle τ as

$$\begin{aligned}\mathbf{T}_\tau &= \begin{pmatrix}\boldsymbol{r}_{\tau,1}-\boldsymbol{r}_{\tau,3} & \boldsymbol{r}_{\tau,2}-\boldsymbol{r}_{\tau,3}\end{pmatrix} \in \mathbb{R}^{2\times 2}\\ \mathbf{L}_\tau &= \begin{pmatrix}\boldsymbol{v}_{\tau,1} & \boldsymbol{v}_{\tau,2} & \boldsymbol{v}_{\tau,3}\end{pmatrix} \in \mathbb{R}^{3\times 3},\end{aligned} \tag{3.1}$$

where $\boldsymbol{r}_{\tau,i}$ are the coordinates of the vertices of τ in texture space and $\boldsymbol{v}_{\tau,i}$ are the coordinates of the vertices of τ in 3D space. The matrix $\mathbf{T}_\tau$ is the conversion matrix from barycentric to edge coordinates and is such that $\mathbf{T}_\tau\boldsymbol{\lambda}' = \boldsymbol{p}-\boldsymbol{r}_{\tau,3}$, given that $\boldsymbol{\lambda}'$ are the first two components of the barycentric coordinates of $\boldsymbol{p}$. Matrix $\mathbf{L}_\tau$ is the conversion matrix from barycentric to Cartesian coordinates. Usually, it is defined as a rectangular matrix in $\mathbb{R}^{4\times 3}$, since it must have a row of ones in order to ensure the point is inside the triangle. In our case, we need to allow points to move outside triangles, and thus we remove that row. As a side effect, we can move back and forth between Cartesian and barycentric coordinates by inverting $\mathbf{L}_\tau$, which saves much computation with respect to the classic area-based method.

The procedure TangentStep iterates the process multiple times to continuously determine the new position and direction in texture space, as well as the current triangle. The number of sub-steps N is some fixed constant depending on the mesh resolution and it is used to determine the length of the sub-step (Line 3). At each iteration, the algorithm moves the agent by a small step (Line 6) and then uses a slightly larger step to determine the next direction (Line 8). The direction is then normalized and rescaled to a proper length (Line 9). The call to TangentSubStep also has the job of determining the next direction and the triangle of the next position to pass them to the next iteration.

Algorithm 3.1 Step in the tangent space of a mesh.

```
1: procedure TangentSubStep($\boldsymbol{p}$, $\Delta\boldsymbol{p}$, $t$)
2:     // Rescale the displacement vector
3:     $\Delta\boldsymbol{p} \leftarrow \frac{\Delta\boldsymbol{p}}{\sqrt{(\Delta\boldsymbol{p})^\top \mathbf{g}(t)\Delta\boldsymbol{p}}}$
4:     // Convert to barycentric coordinates
5:     $\boldsymbol{q} \leftarrow \boldsymbol{p} + \Delta\boldsymbol{p}$
6:     $\boldsymbol{\lambda} \leftarrow \mathbf{T}_t^{-1}(\boldsymbol{q} - \boldsymbol{r}_{t,3})$
7:     $\lambda_3 \leftarrow 1 - \lambda_1 - \lambda_2$
8:     if $\boldsymbol{\lambda}$ is inside $t$ then
9:         return $(t,\ \boldsymbol{p} + \Delta\boldsymbol{p})$
10:    end if
11:    // Search for the nearest adjacent triangle
12:    $w \leftarrow \arg\min_{\tau \in \mathrm{Adj}(t)} \{|\langle \boldsymbol{n}_\tau,\ \boldsymbol{q} - \boldsymbol{c}_\tau \rangle|\}$
13:    $\boldsymbol{q} \leftarrow \boldsymbol{q} - \langle \boldsymbol{n}_w,\ \boldsymbol{q} - \boldsymbol{c}_w \rangle\, \boldsymbol{n}_w$
14:    // Return final texture coordinates and the new triangle
15:    $\boldsymbol{\lambda} \leftarrow \mathbf{L}_w^{-1}\boldsymbol{q}$
16:    return $(w,\ \lambda_1\boldsymbol{r}_{w,1} + \lambda_2\boldsymbol{r}_{w,2} + \lambda_3\boldsymbol{r}_{w,3})$
17: end procedure
1: procedure TangentStep($\boldsymbol{p}$, $\Delta\boldsymbol{p}$, $t$)
2:     $\boldsymbol{p}' \leftarrow \boldsymbol{p}$
3:     $\Delta\boldsymbol{p}' \leftarrow \frac{\Delta\boldsymbol{p}}{N}$
4:     for $i \leftarrow 1$ to $N$ do
5:         // Compute the final position and the new direction
6:         $(w,\ \boldsymbol{p}'') \leftarrow$ TangentSubStep($\boldsymbol{p}'$, $\Delta\boldsymbol{p}'$, $t$)
7:         $(-,\ \Delta\boldsymbol{p}'') \leftarrow$ TangentSubStep($\boldsymbol{p}'$, $\Delta\boldsymbol{p}'$, $t$)
8:         $\Delta\boldsymbol{p}' \leftarrow \|\Delta\boldsymbol{p}'\|\ \frac{\Delta\boldsymbol{p}'' - \boldsymbol{p}''}{\|\Delta\boldsymbol{p}'' - \boldsymbol{p}''\|}$
9:         $\boldsymbol{p}' \leftarrow \boldsymbol{p}''$
10:        $t \leftarrow w$
11:    end for
12:    return $(t,\ \boldsymbol{p}',\ \mathrm{atan2}(\Delta\boldsymbol{p}'))$
13: end procedure
```

3.2.2 Length Rescaling

In practice, we operate in texture space and represent the position of each agent by two coordinates, and its direction as a single scalar (i.e., the steering angle). The pheromone is encoded as color. If the agent does not cross the border of a triangle during a step, the change in position is easily computed by summing two 2-dimensional vectors. However, this choice also requires taking into account the metric distortion induced by the UV mapping.

To guarantee that a vector $\boldsymbol{a}$ in texture space has uniform 3D length on the mesh domain, we divide it by the norm $\|\boldsymbol{a}\|_{\mathbf{g}} = \sqrt{\boldsymbol{a}^\top \mathbf{g}\boldsymbol{a}}$, where $\mathbf{g}(t)$ is the discrete metric tensor at triangle t. Since the latter depends entirely on the triangle t, it can be pre-computed for all triangles at initialization. In Figure 3.5, we depict an example of this correction in texture space and 3D space.

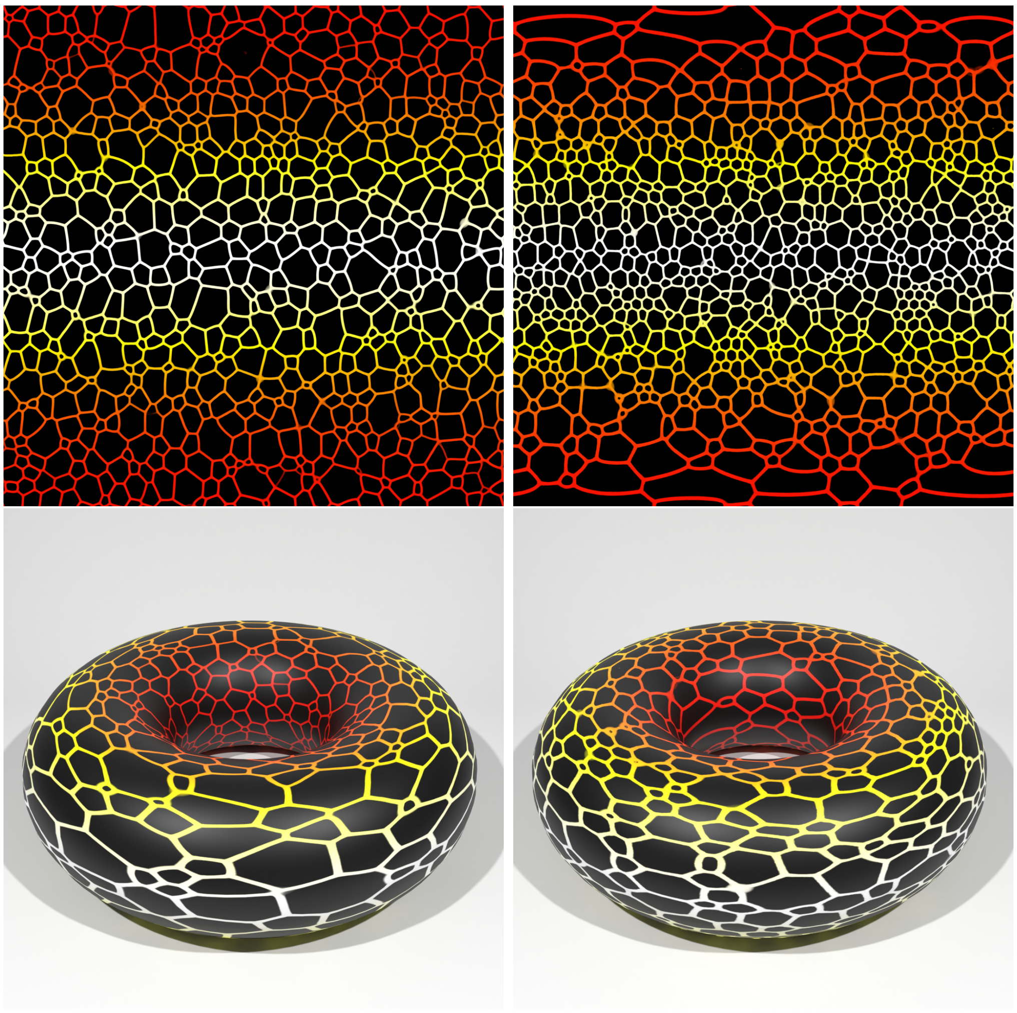


Figure 3.5. Comparison between two pheromone traces on a torus with and without length rescaling. On the left, the agent movements in texture space are of the same size, producing a pattern with inconsistent lengths in 3D. On the right, lengths in texture space are rescaled according to the metric, generating a more uniform pattern in 3D. We represent the metric $\|\cdot\|_{\mathbf{g}}$ as a colormap growing from dark red to white.

3.2.3 Mold Evolution

Our mold evolution process follows the spirit of [114]; we summarize it in Algorithm 3.2. We pre-compute the set of adjacent triangles and the metric tensor at each triangle (Lines 2-6). Then, we simulate a certain number of steps. At each step, all the agents sense three positions in front of them by sampling the texture Pheromone (Lines 11-15). The sensor placement depends on two parameters: the sensor distance δ_s and the sensor angle ϑ_s. The location with the highest quantity of pheromone attracts the agent (Line 17). The agent then turns to that direction with turn speed ϑ_a and moves along the new direction with velocity δ_a (Lines 18 to 20). Some randomness can be added to the agent's steering for adding extra dynamism and random changes to the pattern. We stress that Algorithm 3.1 is only a possible implementation of TangentStep, which we choose for keeping the overall

Algorithm 3.2 Slime mold on surfaces.

```
1: procedure MoMaS($\mathcal{M}$, $h_f$, $\Delta h$)
2:     for all triangle $t$ do
3:         // Pre-compute adjacency and metric tensor
4:         Compute Adj($t$), $\mathbf{g}(t)$
5:     end for
6:     $h \leftarrow 0$
7:     while $h < h_f$ do
8:         for all agent $a$ do
9:             // Sense the pheromone trace
10:            for $\theta \in \{-\vartheta_s,\ 0,\ \vartheta_s\}$ do
11:                $\Delta\boldsymbol{s} \leftarrow (\cos(a.\theta + \theta),\ \sin(a.\theta + \theta))$
12:                $(\boldsymbol{s}_\theta, -, -) \leftarrow$ TangentStep($a.\boldsymbol{p}$, $\delta_s\Delta\boldsymbol{s}$, $a.t$)
13:                $P_\theta \leftarrow$ Pheromone($\boldsymbol{s}_\theta$)
14:            end for
15:            // Determine the next direction and move the agent
16:            $\theta^* \leftarrow \arg\max_{\theta\in\{-\vartheta_s,\ 0,\ \vartheta_s\}} \{P_\theta\}$
17:            $a.\theta \leftarrow a.\theta + \vartheta_a\theta^*$
18:            $\Delta\boldsymbol{p} \leftarrow (\cos(a.\theta),\ \sin(a.\theta))$
19:            $(a.\boldsymbol{p}, a.t, a.\theta) \leftarrow$ TangentStep($a.\boldsymbol{p}$, $\delta_a\Delta\boldsymbol{p}$, $a.t$)
20:            // Release pheromone
21:            Pheromone($a.\boldsymbol{p}$) $\leftarrow 1$
22:        end for
23:        // Apply global pheromone diffusion and evaporation
24:        Blur the texture Pheromone
25:        Pheromone $\leftarrow \max(\text{Pheromone} - \varepsilon_d, 0)$
26:        $h \leftarrow h + \Delta h$
27:    end while
28: end procedure
```

method simple and efficient, but it could be any procedure that moves a point over the tangent space and returns the final position, the triangle containing the final position, and the final direction aligned with the movement. An implementation of Algorithm 3.2 is available at a public repository on GitHub.[1]. Once the agents finish releasing their pheromone in their new position on the texture Pheromone (Line 22), we apply a blurring algorithm to diffuse the pheromone and remove a small quantity that evaporates with velocity ε_d (Lines 25 and 26). The parameters introduced in the last lines characterize the behavior and the obtained pattern, and we analyze them in detail in Section 3.3.3.

As a final note, to keep our implementation simple and GPU-friendly, we accomplish the blurring step (Line 25) with a standard Gaussian blur directly on the texture. While this can in principle generate visible seams, it allows us to handle very high-resolution textures in real time, and the artifacts on texture seams are not visually noticeable as we demonstrate in our experiments.

[1]https://github.com/filthynobleman/slime-manifold

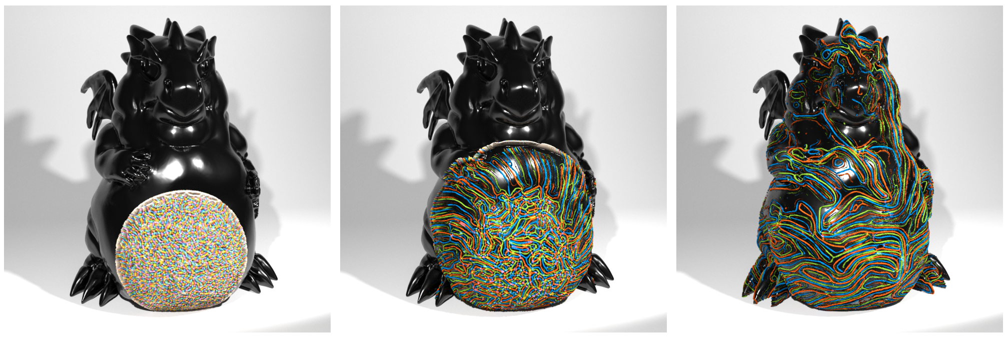


Figure 3.6. Evolution of a 3-species slime mold over a surface. Agents start from an initial region and diffuse over the entire mesh.

3.2.4 Agent Implementation

Agents are represented by a data structure with the following fields:

- $\boldsymbol{p}$: position of the agent in texture space;
- θ: direction angle in texture space;
- t: triangle containing point $\boldsymbol{p}$.

Each agent contains only four values, making the representation memory-efficient. Despite the triangle being a piece of redundant information (it could be inferred by $\boldsymbol{p}$), computing it at each step becomes unfeasible in terms of performance.

Since the agents act independently, we implement their behavior on GPU, achieving real-time performance on the simulation.

3.2.5 Multiple Species, Obstacles and Attractors

The slime mold algorithm allows for a further generalization, including multiple species of agents; see Figure 3.6. Each different species releases a different type of pheromone, attracting agents of the same species and repelling others. To implement this mechanics, we need to distinguish between pheromone types and define 'attractive' and 'repelling' pheromone. Since we encode the pheromone trace in a texture, we assign a channel of the texture to each species. When sampling for pheromone at a given cell, an agent adds the pheromone from the channel of its species and subtracts the pheromone from the channel of other species. This way, agents from the same species will tend to aggregate and isolate from other species. In Figure 3.7, we visualize an example of this behavior.

Similarly, we exploit attractive/repelling pheromone to model obstacles and attractors. An obstacle is a region R_O that agents must avoid, whereas an attractor is a region R_A that agents must reach and never leave. With the expressive power of our system, we can easily define these regions as particular pheromones that never diffuse and evaporate. We assign to the obstacle region R_O an infinitely large amount of pheromone that repels all the species. This trick guarantees that an agent prefers any other direction rather than entering R_O. For the attractors, we define an amount of attractive pheromone superior to the number of species but not infinite. Since the pheromone of species has values in $[0, 1]$, this always guarantees attraction, but still makes the agents able to produce patterns inside R_A.

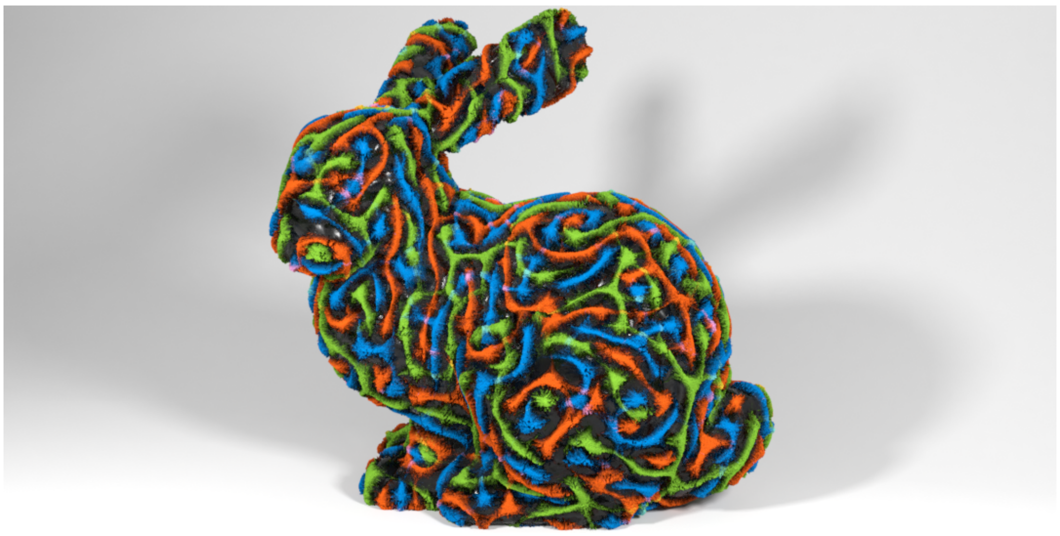


Figure 3.7. An example of simulation with multiple species, repelling each other. The pheromone trace of each species is stored in a different channel of the texture and is represented with a different color.

Parameter		**Property**		**Tested Range**
Movement Speed	δ_a	Pattern Scale	↑	$[1.0, 2.0]$
Turn Speed	ϑ_a	Stabilization	↓	$[10.0, 50.0]$
Vision Distance	δ_s	Cell Formation	↑	$[0.4, 2.0]$
Vision Angle	ϑ_s	Thickness	↑	$[10°, 50°]$
Evaporation Rate	ε_d	Clustering	↓	$[0.2, 1.0]$

Table 3.1. Each parameter is associated with a specific property of the generated pattern most affected by its variation. The arrow indicates whether the parameter affects the property positively (green up arrow) or negatively (red down arrow). The last column shows the range we tested for the parameter.

3.3 Results

Before passing to a quantitative and qualitative evaluation, we provide a description of the key parameters of our algorithm.

Parameters Among the parameters, five of them have a higher impact on the produced pattern in our experiments. More in detail:

δ_a: movement speed (*i.e.* how fast agents move);
ϑ_a: turning speed (*i.e.* how fast agents change direction);
δ_s: distance of vision (*i.e.* how far the sensor is placed);
ϑ_s: angle of vision (*i.e.* how widely sensors are placed);
ε_d: evaporation rate (*i.e.* how fast the pheromone decays).

In Table 3.1, for each parameter, we show the tested range of values and the property that is mainly affected by each of them.

3.3.1 Mold Simulation

The generic evolution of the pattern follows these steps:

1. the agents are initialized randomly;
2. the agents start to move freely, covering large portions of the surface;
3. the agents rapidly aggregate, and the properties of the pattern family show up;
4. the pattern changes slowly but continuously, without changing family.

These steps are common to all configurations and are not affected by the parameters, even if changing the parameters can produce small changes in the timing.

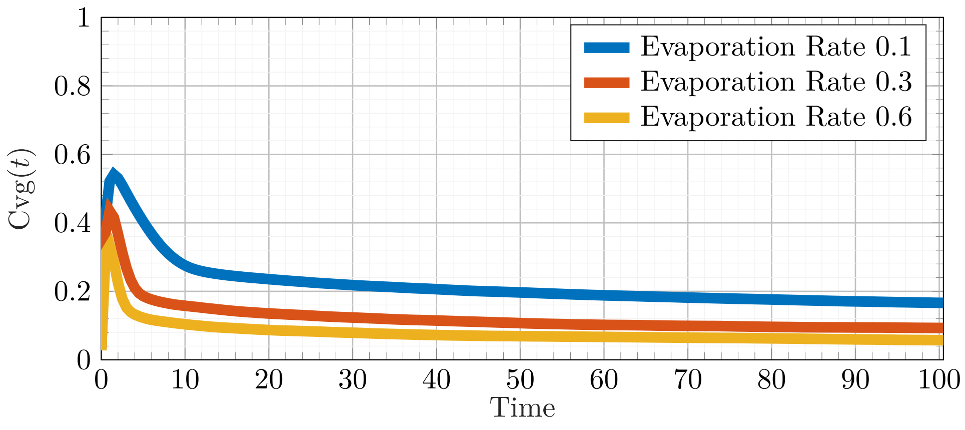
To give a quantitative idea of the pattern distribution and identify when the process reaches each step, we studied how much surface is covered by pheromone over time. More formally, we define the *pheromone coverage* $\mathrm{Cvg}(t)$ and the *pheromone presence* $\mathrm{Pnc}(t)$, as:

$$\begin{aligned} \mathrm{Cvg}(t) &= \frac{1}{\mathrm{Full}} \int_{\mathcal{M}} \mathrm{Pheromone}_t(x)\ \mathrm{d}x\,, \\ \mathrm{Pnc}(t) &= \frac{1}{\mathrm{Full}} \int_{\mathcal{M}} \lceil \mathrm{Pheromone}_t(x) \rceil\ \mathrm{d}x\,, \end{aligned} \tag{3.2}$$

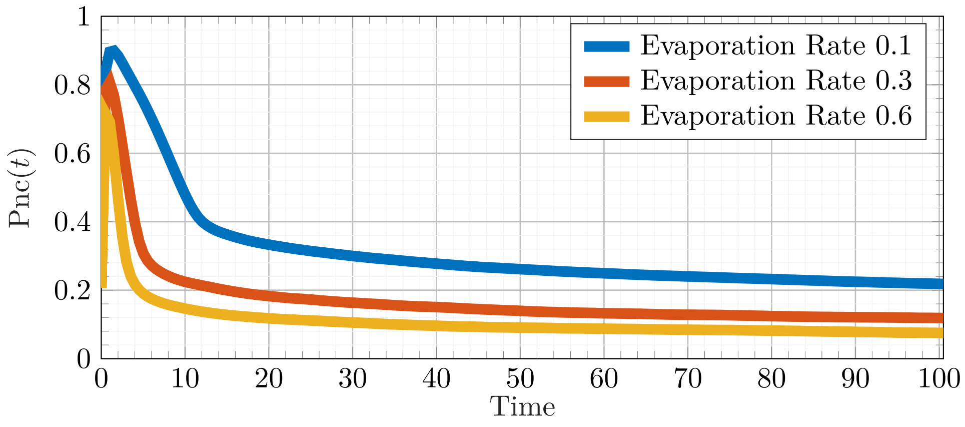
where $\mathrm{Pheromone}_t(x)$ is the quantity of pheromone at point x and at time t and takes values in $[0, 1]$, and $\lceil\cdot\rceil$ is the ceiling operator. The values are normalized with respect to the total coverage $\mathrm{Full} = \int_{\mathcal{M}} \mathrm{d}x$.

Figures 3.8a and 3.8b show the typical evolution of pheromone coverage and presence during a simulation, with different values of evaporation rate. After the random initialization, the agents start to move freely, and they cover large portions of the mesh, reaching a peak. As the pattern is formed, the coverage decreases. The stable line after the peak represents the slow evolution of the pattern. Biological simulations also exhibit a similar behavior, showing our method acts as expected [210, 163]. The evaporation rate affects pheromone coverage and presence, as shown in Figure 3.8c. The other parameters do not affect the pheromone coverage and have a marginal impact on the pheromone presence.

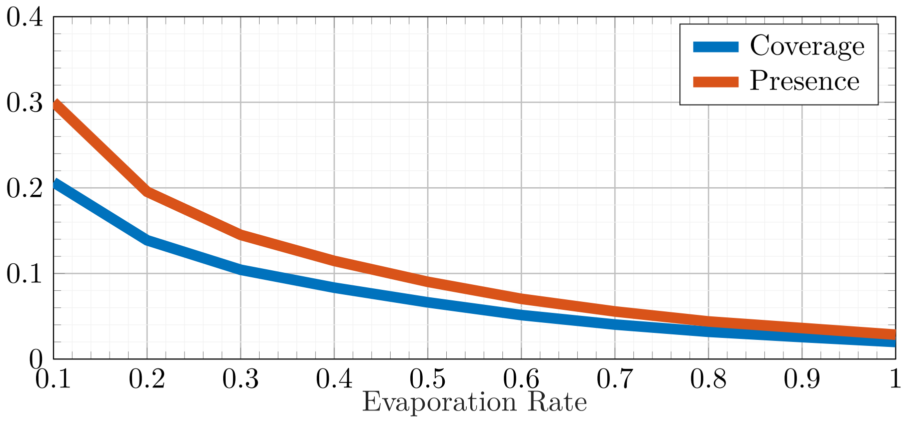
Attractors and Obstacles. In Figure 3.9, we show a simulation with attractors and obstacles. The ability to solve mazes is a well-studied property of organisms like the *Physarum Polycephalum* [3, 186], and we tested our method in this context. The agents are initialized in the red region and expand toward the attractive region (in green), avoiding the walls. In our implementation, we use a three-channel texture to define the starting, repelling, and attractive regions, each associated with a different channel. In contrast, in Figure 3.6 we show a 3-species mold evolution over a surface with no obstacles or attractors, thus leaving the agents completely free.



(a)



(b)



(c)

Figure 3.8. Evolution of the pheromone coverage (a) and presence (b) during a simulation of 100 seconds. After the initial peak, the agents stabilize on the pattern family and slowly vary the resulting pattern. Different decay rates shift the curve, without changing the evolution. The pheromone coverage and presence after 70 seconds of simulation vary as the decay rate changes (c). The slower the evaporation, the higher the coverage and presence. We performed these experiments on the shape on the left of Figure 3.1.

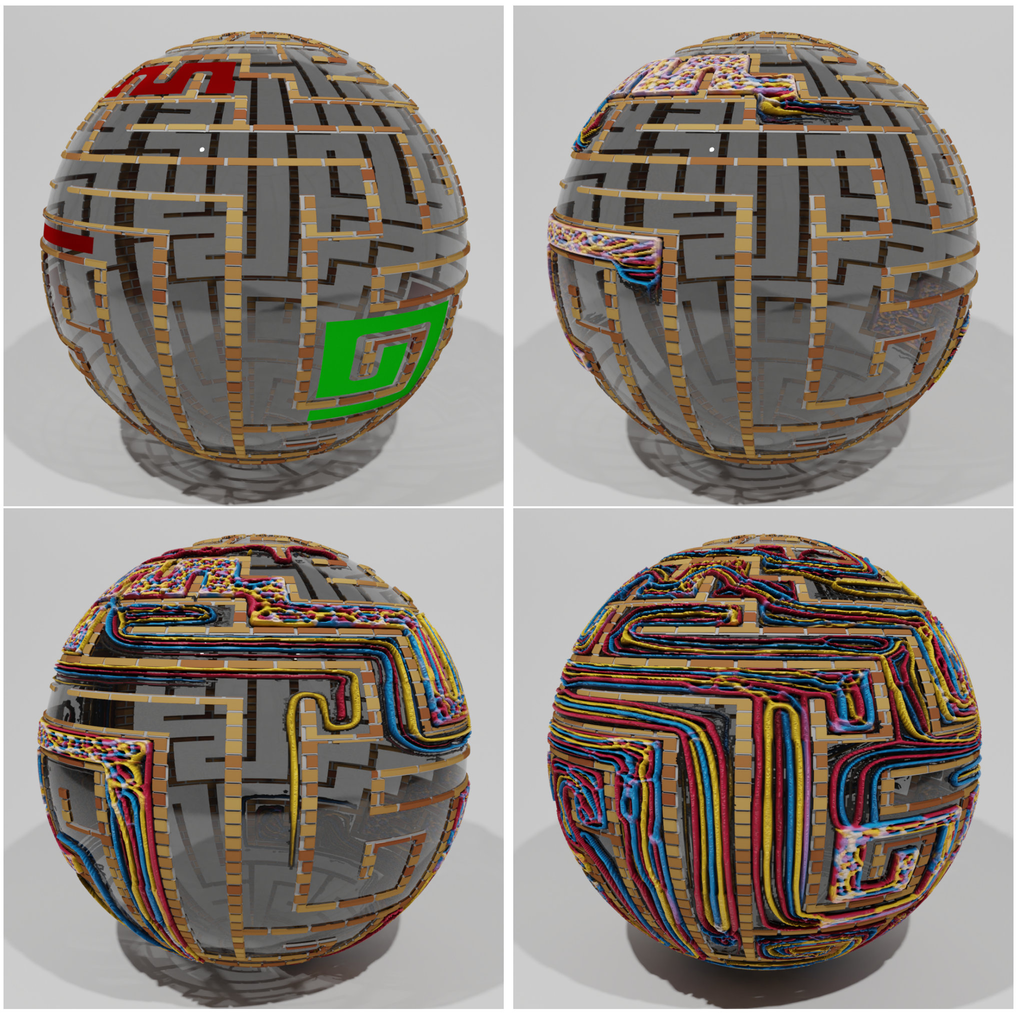


Figure 3.9. A sphere is decorated with a texture representing a maze (top-left frame). The red region is the starting zone, and the green region is the attractor. Three species of slime are initialized in the starting region (top-right) and evolve over the maze (bottom-left). When the agents reach the attractor, they start forming patterns inside and never leave the attracting region (bottom-right).

3.3.2 Performance

We run our simulations on a fixed set of ten meshes and under different conditions. All the experiments have been carried out on a NVidia RTX 2080 Ti (11GB VRAM).

Texture Resolution. Table 3.2 summarizes the average, maximum, and minimum time per frame using textures at different resolutions. We run the simulations for 900 frames with the same settings. In particular, we use ten sub-steps in the tangent space. Despite the increase in time per frame, results show that our algorithm achieves real-time performance at very high resolutions.

texture size	avg. tpf	max. tpf	min. tpf
2048	8.473	37.277	7.362
4096	9.503	37.693	8.268
8192	12.120	39.039	10.263
16384	23.715	43.552	16.220

Table 3.2. Time per frame (in milliseconds) at different texture resolutions. Simulations are ran using 10 sub-steps and involve 1M agents.

num. steps	avg. tpf	max. tpf	min. tpf
1	12.070	37.204	10.020
10	13.428	38.632	11.087
100	47.828	61.519	44.756
1000	423.136	488.078	411.836

Table 3.3. Time per frame (in milliseconds) at different steps per frame. Simulations are ran on a 8192×8192 texture and involve 1M agents.

num. agents	avg. tpf	max. tpf	min. tpf
1M	12.053	39.522	10.214
2M	19.112	37.205	17.306
3M	26.303	41.857	24.006
4M	33.510	53.710	31.114

Table 3.4. Time per frame (in milliseconds) at different number of agents. Simulations are ran on a 8192×8192 texture and use 10 sub-steps.

Number of Sub-steps. To show the impact of our movement parcellation policy, in Table 3.3, we show the average, maximum and minimum time per frame at different number of sub-steps. We set the texture resolution to 8192×8192. As for Table 3.2, simulations are done for 900 frames, fixing the other simulation parameters. These results show that one or ten steps perform similarly. Subdividing by 100 or 1000 steps, the frame rate significantly drops. We conjecture that up to 100 steps, the GPU threads warm-up cover a significant portion of the frame time. This dependency on the number of sub-steps relates the execution time to the mesh resolution. For denser meshes, the number of triangles crossed by a single step increases, together with the number of sub-steps required to keep the result consistent. This, in turn, increases the execution time.

Number of Agents. Finally, we are interested in studying the efficiency of our implementation while varying the number of agents. Table 3.4 summarizes our results over 900 frames. The increase in the average time per frame is almost linear (about 7 ms at each increment of one million agents), showing our approach scales well with the number of agents.

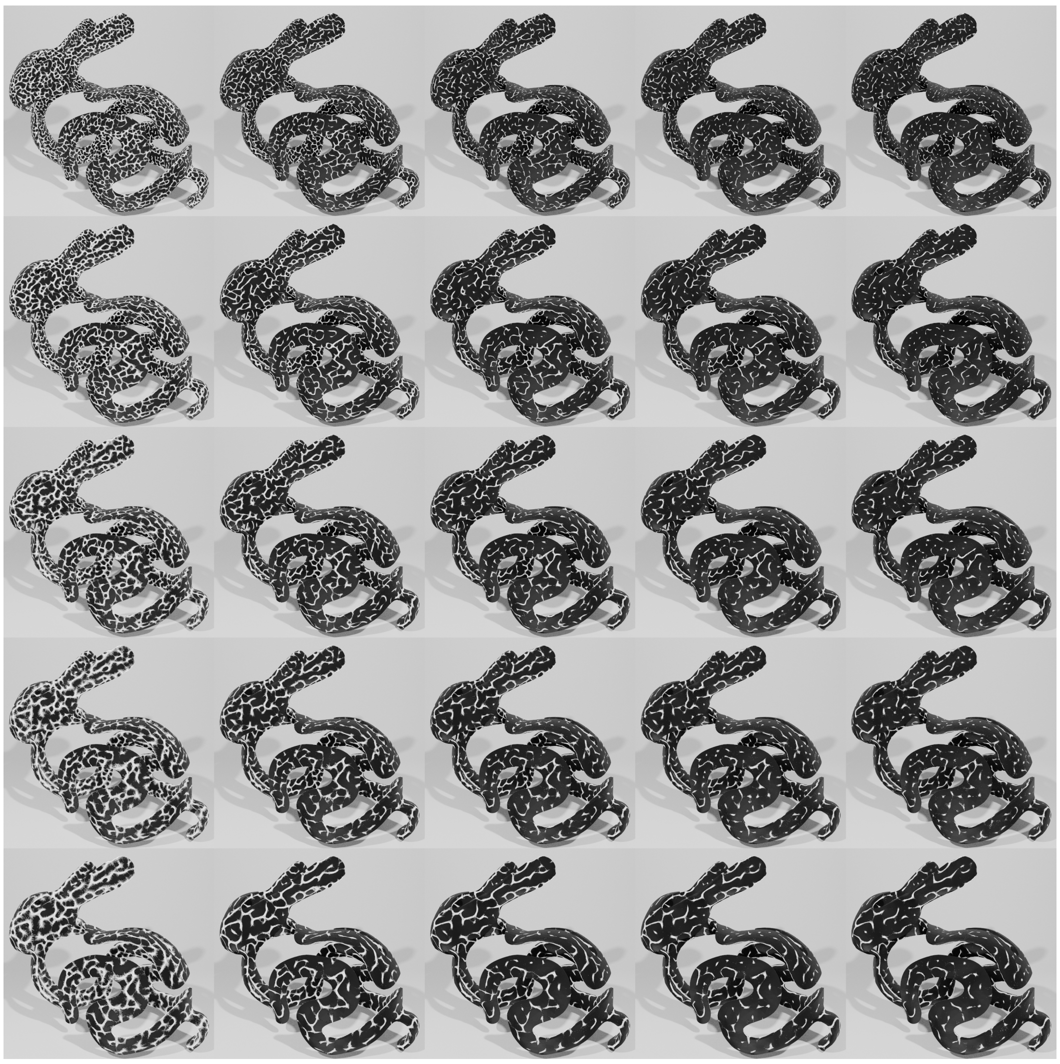


Figure 3.10. Changes in the resulting pattern when varying the movement speed and the evaporation rate. The change in movement speed scales up the pattern along each column, making it wider. As the evaporation rate increases along each row, the agents tend to form shorter chains and collapse into local clusters.

3.3.3 Families of Patterns

Our parameters allow to generate different types of pattern. The movement speed controls the pattern scale; as we increase the velocity of the agents, the pattern becomes uniform, and local details tend to disappear. Figure 3.10 shows this variation together with the variation in the evaporation rate. Here the values of the parameters are linearly interpolated inside the ranges shown in Table 3.1. This parameter weighs on the tendency of agents to form clusters. When the evaporation rate is low, the pheromone stays on for longer, and agents can travel by longer distances following it. As the evaporation rate increases, the agents tend to aggregate only with other agents already near them, and thus they tend to form small clusters.

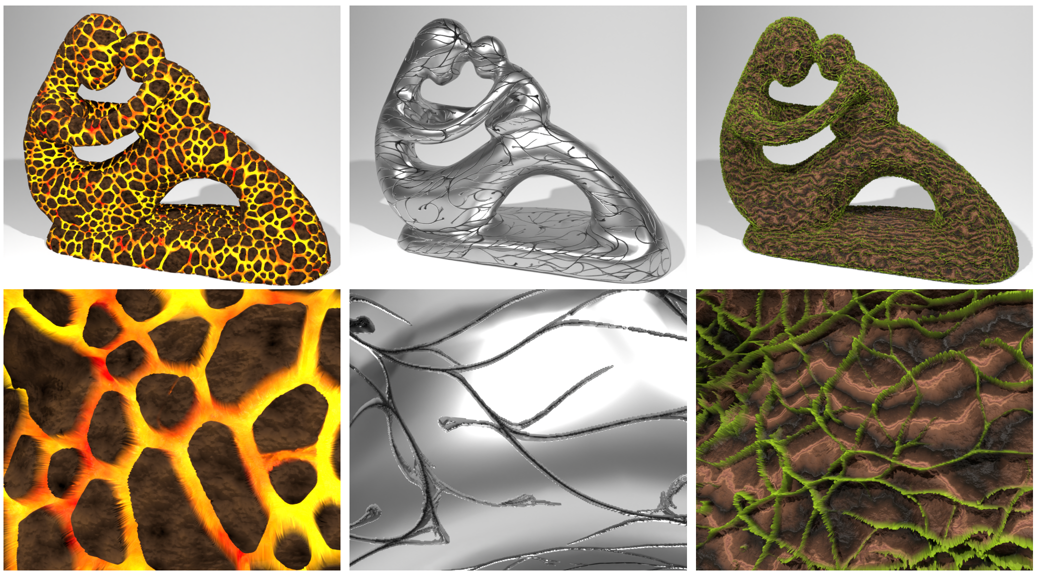


Figure 3.11. Frames from simulations with the same mesh and initial conditions but with different simulation parameters. The produced patterns have been used as composition masks and details maps for more complex materials. The second row shows a close-up detail of the surface.

3.3.4 Evolutive Procedural Texturing

The creation of complex patterns on meshes is a time-consuming task for texture artists. For many applications, such as landscape generation or tile synthesis, procedural texturing has become very useful and effective [170, 200, 132, 208]. Nevertheless, most of the literature is devoted to the generation of complex textures on planar domains, and only a few works deal with general manifold meshes [188, 162, 84]. The ability to generate a broader range of complex patterns that follow the shape of a mesh is of fundamental importance in generating a large variety of contents. Moreover, [114] shows that it is possible to obtain a variety of really different patterns by simply tuning the simulation parameters, as shown in Figure 3.11.

Visualizing the evolution of patterns and effects using textures has been extensively used for time-dependent vector fields and scalar functions in 2D domains [192, 22]. However, animating materials and textures over surfaces is currently limited to very simple shaders and texture changes. The evolution of slime mold organisms on a surface effectively animates a coloration of the mesh at the texture level. In Figures 3.1 and 3.11 we show how to exploit our method to simulate properties like wood grain, metal incisions, or lava rivers.

3.3.5 Limitations

Our method inherits the main limitations of texture-based approaches. For example, dealing with meshes with many small triangles may cause a performance drop due to the high resolution required to model pixels and the step subdivision. However, this does not impact the correctness of our method, and we expect this situation to occur only for contexts where real-time performance is less relevant.

Moreover, if adjacent triangles in a mesh generate an angle smaller than 90 degrees can produce inconsistencies in reprojecting the agents. In this case, triangle unfolding can be a more desirable option for moving the agent over the surface. For this reason, we implemented the algorithm so that TangentStep can be viewed as a black-box routine and replaced according to the user needs.

3.4 Conclusions

We presented a generalization of the slime mold algorithm to surface meshes and an analysis of its behavior and controllability with simulation parameters. The algorithm produces complex patterns on triangle meshes, behaving similarly to real biological entities. We discussed the applicability of our method to computer graphics tasks like procedural texturing. Finally, we discussed our GPU implementation and its applicability to real-time texture animation.

We consider studying how molds evolve on surfaces with different geometrical properties as an interesting future direction. This analysis could open to other applications, *e.g.*, extrapolating geometry descriptors or intrinsic properties of the surface from the mold's distribution.

Chapter 4

A Physically-Inspired Approach to the Simulation of Plant Wilting

Plants are among the most complex objects to be modeled in computer graphics. While a large body of work is concerned with structural modeling and the dynamic reaction to external forces, our work focuses on the dynamic deformation caused by plant internal wilting processes. To this end, we motivate the simulation of water transport inside the plant which is a key driver of the wilting process. We then map the change of water content in individual plant parts to branch stiffness values and obtain the wilted plant shape through a *position based dynamics* simulation. We show, that our approach can recreated measured wilting processes and does so with a higher fidelity than approaches ignoring the internal water flow. Realistic plant wilting is not only important in a computer graphics context but can also aid the development of machine learning algorithms in agricultural applications through the generation of synthetic training data. This work was realized in collaboration with Jonathan Klein (Ph.D.) and Torsten Hädrich (Ph.D.), to whom goes the credit for realizing the models of the plants and designing the experiments, to Sören Pirk (Ph.D.) and prof. Wojtek Pałubicki, both of which largely helped in defining the positioning of this work in the state-of-the-art and in finding references from literature in biology, and to prof. Dominik L. Michels and prof. Emanuele Rodolà, whose supervision in refining and organizing the work greatly improved the presentation and the results. The results presented in this chapter have been published in the proceedings of the *ACM SIGGRAPH Conference and Exhibition on Computer Graphics and Interactive Techniques in Asia* (*SIGGRAPH Asia*) [159].

4.1 Introduction

High quality plant models are required as content in games and movies as well as for applications in architecture, urban planning, and forestry [70]. Moreover, synthetic data generated from detailed crop models is nowadays applied to train smart systems carrying out computer vision tasks in agriculture [123]. For instance, *SWEEPER* – a sweet pepper harvesting robot – has been successfully trained using synthetic

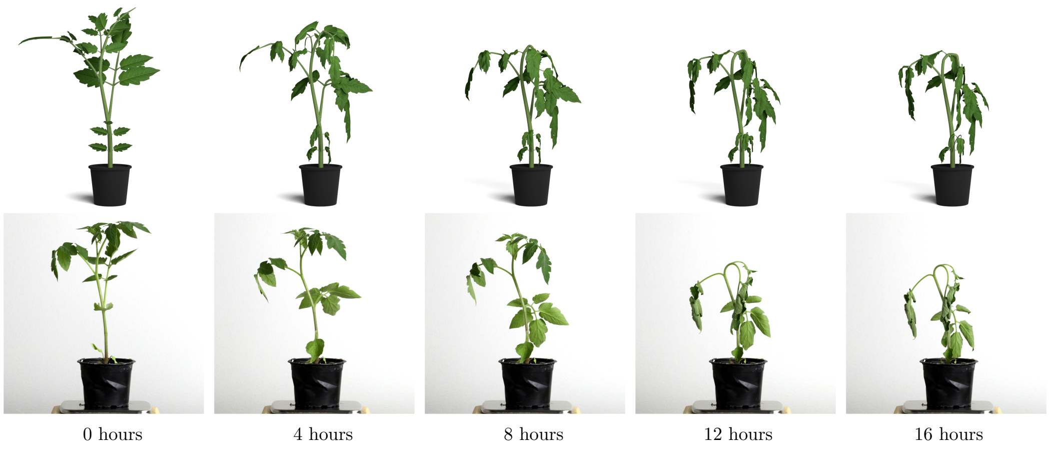


Figure 4.1. Temporal evolution (left to right) of the wilting process of a tomato plant (*Solanum lycopersicum*) simulated with our approach (top) and captured using an experimental setup in the laboratory (bottom).

data to estimate the exact location of the individual peppers [293, 21].

While numerous studies in computer graphics and agricultural research have provided solutions to dynamically simulate the 3D development of plants, these works usually focus on the growth process itself [261]. Processes such as plant reaction to stresses, e.g. disease or drought, are rarely simulated in a way that are useable for rendering. The main reasons for this are twofold: on the one hand, the detailed understanding of developmental processes controlling stress reactions of plants is still an active topic of research in biology, and on the other hand, the efficient integration of plant and mechanical signaling in a computer simulation poses non-trivial numerical challenges.

Specifically, overcoming the challenge of comprehending how plants react to drought is becoming an increasingly important task for humanity. More frequent drought conditions due to climate change are widely believed to cause die-back of forests in Europe, and unpredictable climatic events can have a detrimental impact on yield production in agriculture [20]. Consequently, the phenomenon of plant wilting is becoming more common in many regions.

In our method, we integrate a physical model of water distribution through the plant's vascular system which is essential for plant growth. We focus on the description of the wilting process as a result of insufficient water supply of the plant. Our model is based on a partial differential equation (PDE) description of plant hydraulics and a simplistic model of evapotranspiration for leaves. This model allows us to calculate the continuous water flow dynamics mapped to a discrete graph representation of the plant structure. In contrast to other models of plant hydraulics presented in the literature, our approach is based on efficient approximations of analytical solutions using an optimized backward difference formula integration scheme.

In summary: (i) we identify the importance of simulating the hidden water flow inside the plant to faithfully recreate its dynamic behavior, (ii) we propose a simple but physically inspired model to simulate said water flow, and (iii) we connect the water flow to the shape change during wilting through a stiffness mapping, and successfully recreate the wilting via a physical simulation.

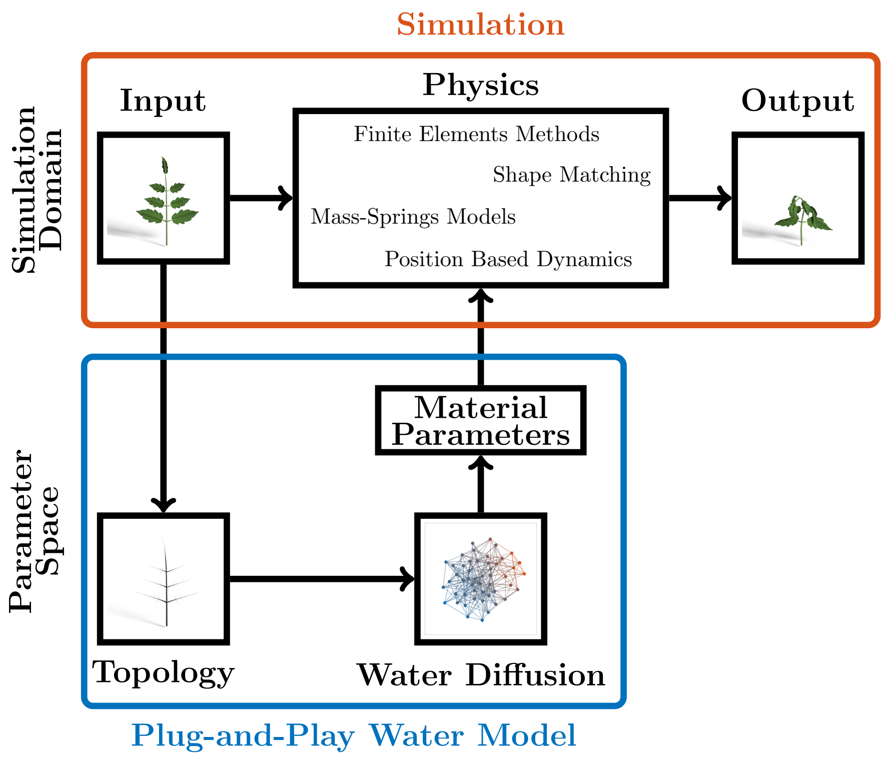


Figure 4.2. The water diffusion model we propose is designed to be a plug-and-play extension to different kinds of physical simulators. By using the topology of the simulated plant, it can compute the complex behavior of the water diffusion inside the plant itself. The water distribution is then used by the physical simulation to compute wilting deformations more realistically than those obtained by the manual tuning of the plant's stiffness.

4.2 Related Work

Faithfully modeling trees and plants is a long standing goal in computer graphics research. Many of the early approaches focus on modeling the morphology of branching structures. Methods exist to generate branching patterns based on fractals [10], grammars [10], repetitive patterns [201], cellular automata [95], and even particle systems [226]. L(indenmayer)-systems [220] enable generating plants based on production rules that can be used to express branching patterns for a wide range of plants. Combined with geometric modeling and user interaction, rule-based modeling [141] allows generating highly complex tree models – even for less experienced users. More recent procedural modelling approaches aim to express tree growth in a phenomenological or self-organizing manner [205, 238, 254].

Several approach focus on reconstructing trees and plants based on sensor data. Popular methods for tree reconstruction span from leveraging images [35, 135, 225, 264] and point clouds [150, 146], to videos [136], segmentation masks [11], silhouettes [290], and envelope shapes [24]. Li et al. [140] reconstruct plants from sequences of point cloud data to identify branching patterns to faithfully model budding and bifurcation events for leafy plants.

On a different trajectory researchers have explored to use sketch-based interfaces to model trees [199], plants [8], and even flowers [109]. Compared to other approaches, sketch-based plant modeling provides more fine-grained control, which often is of high relevance for content creation. Moreover, it has been shown that sketch-based modeling can be combined with evolved procedural approaches to generate complex branching patterns with lightweight user intervention [151].

More recently, a number of methods focus on the dynamics and physically-plausible modeling of plants. This ranges from interactively modeling and animating the growth of plants [151, 215], the response of plants to physical phenomena, such as wind [216, 96, 222] or fire [214], the simulation of the cambium of trees [129], to the interaction of plants with their environment [216, 292]. Zhao and Barbic [301] as well as Wang et al. [279] use finite element methods (FEM) solvers to simulate biophyical and biomechanical deformations of plants to capture their plasticity and to generate plant motion. Owens et al. [204] propose algorithms for modeling and animating the development of inflorescences, while Ringham et al. [229] aim at increasing the realism of flower models based on a combination of mathematical models for pigmentation patterns. The simulation of biological and physical phenomena for plants is often realized through efficient representations, such as plant modules [206], particles [97], or discretization schemes that specifically consider the morphology of plants, such as veins [111], or branches [214].

Closest to our work are the approaches of Hädrich et al. [97] and Shao et al. [243] who also employ position-based dynamics [23] for simulating the growth of climbing plants and the rod-dynamics of branching structures. Finally, our approach is similar to the objective of Jeong et al. [111] who also aim to simulate the morphological changes of drying leaves. However, unlike these methods we propose a physically-plausible diffusion model that allows us to simulate the water flow in the vascular system and – consequently – the wilting of an entire plant.

4.3 Methodology

In this section, we analyze the process of water transport and evapotranspiration inside plants. From that, we derive a computational model for simulating the water diffusion, and we determine the system's initial conditions.

4.3.1 Water Uptake and Loss Process

Plants lose a significant amount of water via evaporation and transpiration. About 95% of water loss happens on leaves and about 5% on the stem [232]. The loss of water depends on the surface area and varies from the range $[15, 250]\mathrm{g}/(\mathrm{h} \cdot \mathrm{m}^2)$ during daytime down to $[1, 20]\mathrm{g}/(\mathrm{h} \cdot \mathrm{m}^2)$ during night. Since transpiration happens via wetting the surface of the plant's cells, the water loss rate reduces as the plant dries out [232].

Water uptake happens via a water potential gradient from the soil to the air, passing through the roots, the stem, and the leaves. Water moves from areas with higher potential to areas with lower potential, and according to Molz [183] one of the simplest models for water transport is to relate the rate of water flow in direct proportion with the water potential difference, *i.e.*

$$\frac{\partial \theta}{\partial t} = -\frac{\Delta \Psi}{R}\,, \tag{4.1}$$

where θ is the amount of water, t is time, Ψ is the water potential, and R is the resistance to the water flow.

Water potential depends on various properties like osmosis, gravity, and humidity [263]. However, water pressure inside the cells of plants is usually the most significant term; hence, we can approximate the potential by considering only the pressure.

As the water transpires and evaporates, the pressure at leaves decreases, and water starts moving upward more rapidly. However, if the water intake from the soil is reduced (or stopped), the water leaving the lower parts of the plant is never replaced, and the pressure slowly decreases. Eventually, the water transport becomes slower and slower until the plant runs out of water and dies completely.

4.3.2 Computational Model

We model a plant as a graph (*i.e.*, as a pair of nodes and edges) that includes the main structure (stem and branches) as well as the leaf veins. This allows for a unified handling of all plant parts: Nodes in the main structure have a loss rate $\ell_v = 0$, whereas nodes representing the venation of a leaf have a loss rate $\ell_v = \delta\ A_v$. Here, δ is the water loss rate per surface area, and it is constant for the entire plant, and A_v is the leaf surface area covered by that node, which we approximate as $A_v = r_v\ h_v$. We approximate the node segments with a cylinder of radius r_v and height h_v, from which we can compute the volume V_v. We associate to each node the water amount θ_v contained in that segment, and we compute the water pressure at the segment as $P_v = \frac{\theta_v}{V_v}$.

Finally, at each node, we also compute the water flow resistance through the segment using the equation for circular pipes [45]: $R_v = \frac{\pi\ r_v^4}{8\ \mu\ h_v}$, where μ is the dynamic viscosity of the water. For the resistance to the water flow along an edge $e = (u, v)$, we average the resistances at the nodes $R_e = \frac{R_u + R_v}{2}$.

With all these quantities defined, we can set up a system of differential equations from (4.1) and integrate the water loss induced by the transpiration of the leaves. For each node v, said $\mathcal{N}_v$ the set of nodes adjacent to v, we have

$$\frac{\partial \theta_v}{\partial t} = -\sum_{u \in \mathcal{N}_v} \frac{P_v - P_u}{R_{(u,v)}} - \ell_v \theta_v = -\sum_{u \in \mathcal{N}_v} \frac{1}{R_{(u,v)}} \left(\frac{\theta_v}{V_v} - \frac{\theta_u}{V_u} \right) - \ell_v \theta_v\,. \tag{4.2}$$

We now arrange all the water content values inside a vector $\boldsymbol{\theta} = (\theta_1,\ \cdots\ \theta_n)$. We also define two diagonal matrices $\mathbf{D}_V$ and $\mathbf{D}_\ell$, respectively filled with the volumes and the water loss rates of the nodes. By doing so, we can rewrite the system in (4.2) as follows:

$$\frac{\partial \boldsymbol{\theta}}{\partial t} = \mathbf{R}\ \mathbf{D}_V^{-1}\ \boldsymbol{\theta} - \mathbf{D}_\ell\ \boldsymbol{\theta} = \left(\mathbf{R}\ \mathbf{D}_V^{-1} - \mathbf{D}_\ell \right) \boldsymbol{\theta}\,, \tag{4.3}$$

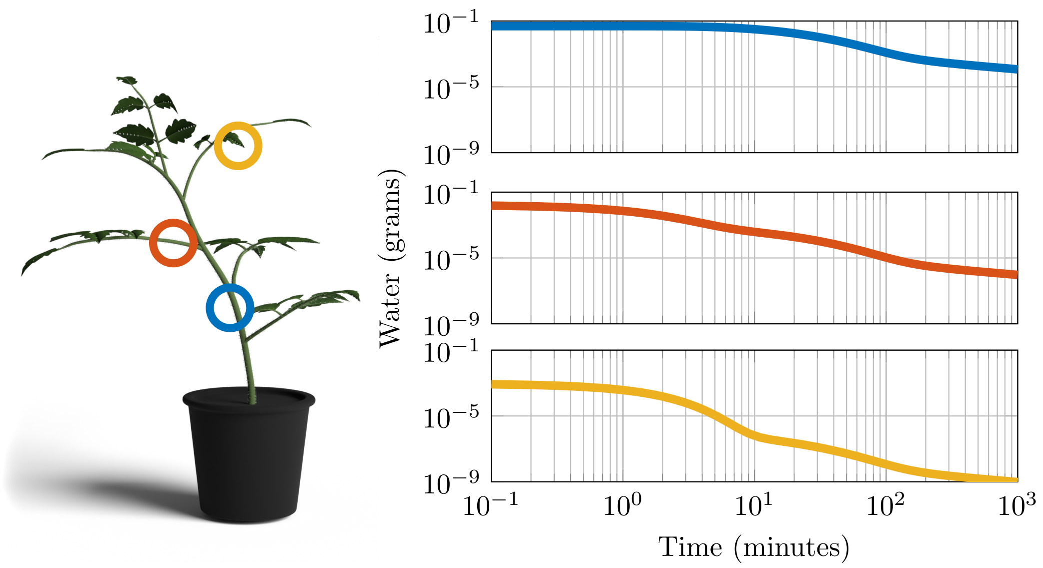


Figure 4.3. Amount of water inside different parts of the plant at varying of time. The thickness of the section determines the initial amount of water. The farthest the section from the leaf, the more it retains water over time. Our water diffusion model is able to capture the complexity of the water distribution, as the rate of water loss is not constant over time.

where $\mathbf{R}$ is a symmetric matrix defined as

$$\mathbf{R} = (r_{v,u}) = (r_{u,v}) = \begin{cases} \dfrac{1}{R_{(u,v)}}\,, & u \in \mathcal{N}_v\,, \\ -\displaystyle\sum_{\substack{u=1\\u\neq v}}^{n} r_{u,v}\,, & u = v\,, \\ 0\,, & \text{otherwise}\,. \end{cases} \tag{4.4}$$

The matrix $\mathbf{S} = \mathbf{R}\,\mathbf{D}_V^{-1} - \mathbf{D}_\ell$ is a linear application that does not depend on the water content and does not change over time. This means the system of differential equations in (4.3) has a solution that can be computed analytically as

$$\boldsymbol{\theta}(t) = \mathsf{exp}\,(\mathbf{S}\;t)\,\boldsymbol{\theta}(0)\,, \tag{4.5}$$

which gives us a formulation for evaluating the water content of the plant at each time instant t, assuming the initial water distribution $\boldsymbol{\theta}(0)$ is known.

In Figure 4.3, we show that the solution to these differential equations is able to capture the complexity of the water distribution inside the plant. The water evolution follows non-trivial curves and moves around, trying to equalize the pressure everywhere, changing its local flow according to how fast the water transpires and evaporates from the leaves.

4.3.3 Initial Conditions

For determining the system's initial condition, we notice the following: a healthy plant can replace all the water it loses via transpiration and evaporation. From the point of view of the water distribution, the plant does not lose water at all, and hence, the water loss is 0 everywhere. By zeroing out the entire matrix $\mathbf{D}_\ell$, we see that the system in (4.3) tends to equalize the pressure everywhere. Since we assume the wilting process to start from a healthy plant in stable conditions, we initialize the water distribution so that the pressure is constant through the entire plant, namely $\theta_v(0) = V_v$.

4.4 Algorithmics

In this section, we provide an efficient method for computing the analytical solution of our water model. We then describe an alternative technique for efficiently computing a close approximation of the solution when the system becomes too large. Finally, we show how our water model can be integrated with a physical solver to simulate the dynamics of a wilting plant.

4.4.1 Efficient Evaluation

The analytical solution provided in (4.5) guarantees numerical stability and accuracy when computing the water diffusion, and allows for the evaluation of the water distribution at any point in time without the need to simulate the entire diffusion process or worrying about the size of the time steps. However, the matrix exponential is notoriously difficult to compute, and evaluating it at every frame can easily become computationally unfeasible.

We propose a two-step solution that moves all the heavy lifting to the initialization step and allows a real-time evaluation at every simulated frame. During the initialization of the water model, we pre-compute the spectral decomposition of the system matrix $\mathbf{S} = \mathbf{\Phi\Lambda\Phi}^{-1}$. We obtain the matrix exponential as

$$\mathsf{exp}\left(\mathbf{S}t\right) = \mathbf{\Phi}\mathsf{exp}\left(\mathbf{\Lambda}t\right)\mathbf{\Phi}^{-1}\,, \tag{4.6}$$

where $\mathsf{exp}\left(\mathbf{\Lambda}t\right)$ is computed by taking the component-wise exponential of the diagonal entries of $\mathbf{\Lambda}t$.

Note, that is is not required to explicitly compute the matrix exponential since we are only interested in the matrix-vector product $\mathsf{exp}\left(\mathbf{S}\ t\right)\boldsymbol{\theta}(0)$. We can rearrange the computation order as

$$\mathbf{\Phi}\left(\mathsf{exp}\left(\mathbf{\Lambda}t\right)\left(\mathbf{\Phi}^{-1}\boldsymbol{\theta}(0)\right)\right), \tag{4.7}$$

so that we only evaluate matrix-vector products. Furthermore, we do not require the inverse of the eigenvectors $\mathbf{\Phi}^{-1}$. We solve the linear system $\mathbf{\Phi}\boldsymbol{\xi}_0 = \boldsymbol{\theta}(0)$ once during the initialization and then compute the products $\mathbf{\Phi}\left(\mathsf{exp}\left(\mathbf{\Lambda}t\right)\boldsymbol{\xi}_0\right)$, where the product $\mathsf{exp}\left(\mathbf{\Lambda}t\right)\boldsymbol{\xi}_0$ is just a scaling of the entries of $\boldsymbol{\xi}_0$ by the entries of the diagonal matrix. Said n the size of the system (*i.e.*, the number of nodes in the tree structure), at each step of the simulation, we only have to evaluate a single $\mathcal{O}(n^2)$ operation, which can be easily parallelized either on CPU or GPU for even more efficiency.

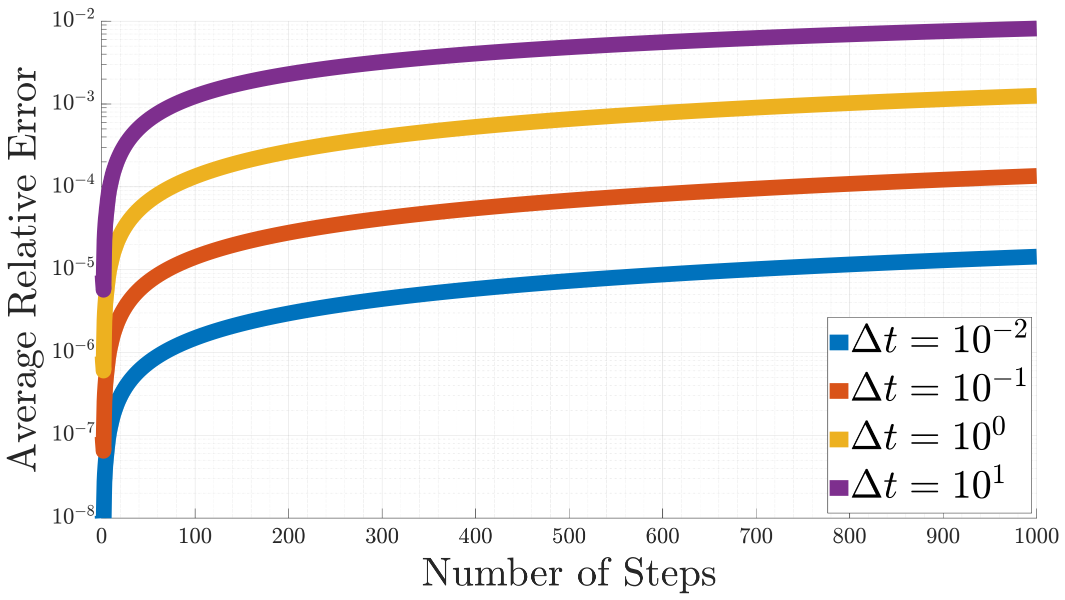


Figure 4.4. Relative mean squared error of the solution obtained with BDF6 over time, compared with the analytic solution and averaged across plants with different resolutions. Each curve is obtained by selecting a different time step for the integration.

4.4.2 Handling of Large Plants

When dealing with small plants, the numerical stability of the analytic solution comes in very handy. However, when dealing with larger plants with thousands of nodes, the spectral decomposition becomes unfeasible. For huge models, even the $\mathcal{O}(n^2)$ matrix-vector multiplication becomes too costly.

To overcome the problem, we propose an alternative evaluation leveraging numerical integrators. The stiffness of the problem leads us to rely on implicit stable methods, and we choose to use a sixth-order backward difference formula (BDF6), which is a notoriously stiffly stable algorithm [259].

The problem with this type of algorithms is that their solution can be computationally costly to evaluate for large systems due to the implicit formulation. However, by noticing that the system is linear and time-invariant, we can use the efficient approach described by Cellier *et al.* [44]. We keep track of the last six water evaluations in a $n \times 6$ matrix $\boldsymbol{\Theta}(t)$ and we store the coefficients of the algorithm as $\alpha = \frac{60}{147}$ and $\boldsymbol{\beta} = \frac{1}{147}(-10,\ 72,\ -225,\ 400,\ -450,\ 360)^\top$. By calling Δt the integration time step, and denoting by $\mathbf{I}_n$ the $n \times n$ identity matrix, we can obtain the water distribution $\boldsymbol{\theta}(t + \Delta t)$ at time $t + \Delta t$ by solving the sparse linear system

$$(\mathbf{I}_n - \alpha\ \Delta t\ \mathbf{S})\,\boldsymbol{\theta}(t + \Delta t) = \boldsymbol{\Theta}(t)\boldsymbol{\beta}\,. \tag{4.8}$$

We notice that the matrix $\mathbf{I}_n - \alpha\ \Delta t\ \mathbf{S}$ is sparse, each row generally containing no more than five or six entries. Furthermore, as the system is a constant across the entire simulation which allows us to apply an efficient sparse LU pre-factorization.

To verify the efficacy of this solution, we select a sample of different plants ranging from 300 to 2k nodes and simulate 1000 steps. We compare the results from the simulation with those obtained with the analytic solution and average

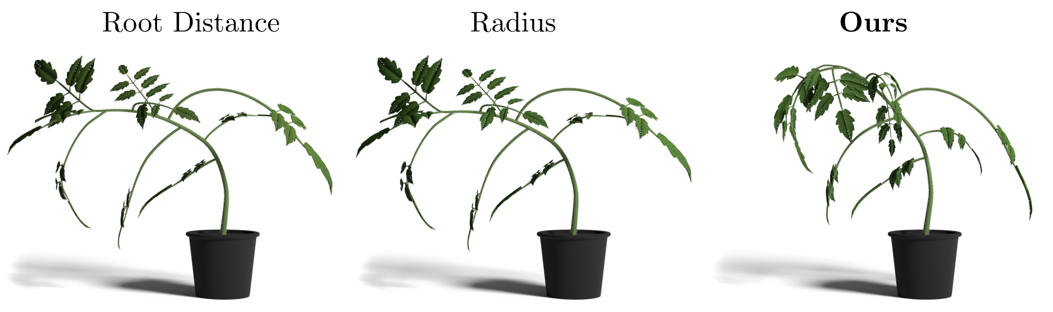


Figure 4.5. Comparison of two different approaches for computing the elasticity modulus of a plant against our method based on the water diffusion. In all three cases, the elasticity modulus at the root node is the same. Left: the elasticity modulus is computed as $1/(1+d)$ where d is the topological distance of the node from the root node. Middle: the elasticity modulus is computed as a linear function of the section's radius. Right: the elasticity modulus is computed from the amount of water in the section obtained from the simulation of the water diffusion.

the relative error across all the plants. We also consider the error at varying the time step, as the step size heavily affects the quality of the simulation. Figure 4.4 shows the results of our experiments, proving that our solution is stable over time and robust even to large time steps. The error accumulates linearly over time and increases linearly as the step size increases. However, the error always stays in an acceptable range, except for long-lasting simulations with very large time steps.

4.4.3 Integration with Physics Simulators

Our water model is independent from the physical simulation of the actual plant and can be easily integrated with different solvers. For our implementation, we rely on a position based dynamics (PBD) solver since it is a well-established method for the simulation of plants and trees within the graphics community [69, 214].

For the modeling of the plants, we use the representation proposed in Deul *et al.* [69], where each plant segment is a cylinder, connected to each of its neighbors via a zero stretch-bend-twist constraint. According to Kim *et al.* [121], the elasticity modulus of the cells of a plant increases with the relative water content. We approximate this effect through the following logistic function for computing the elasticity E from the water content:

$$E(\theta) = \frac{1}{1 + \mathsf{exp}\left(-\kappa\left(\theta - \chi_0\right)\right)} . \tag{4.9}$$

The function has two tunable parameters which are determined during calibration. χ_0 is the shift of the logistic curve along the axis of θ, while κ controls its steepness.

We find, that χ_0 should usually be set to zero to immediately start the wiling process while κ acts more as a material parameter that abstracts the complex cellular structure of plants. A deeper evaluation of this is found in the next section.

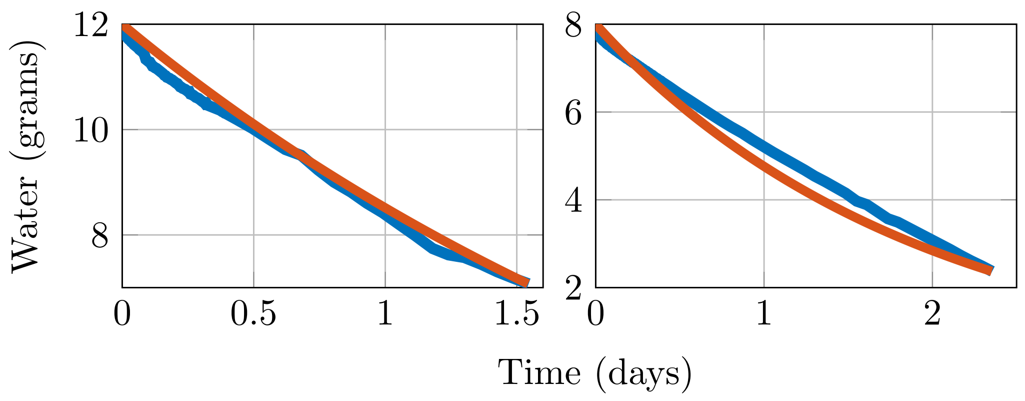


Figure 4.6. Comparison of the water curves obtained from real data (blue curves) and simulation (orange curves) on two different tomato plants. The initial water content of the simulation is matched to the real plant, and we used the same water loss rate for both plants.

Note, that the dynamic simulation of the plant is entirely rod based in our approach. Since leaves are represented through their veins in the plant graph, their deformation naturally emerges. While we find that this gives convincing results in practice, methods with a more detailed focus on leaf deformation have been presented as well [111].

4.5 Results

In this section, we provide an evaluation of our water model against the actual wilting processes of real plants. We then show how our model can be calibrated to simulate various types of plants and model different types of diseases.

4.5.1 Water Model Evaluation

To evaluate our water diffusion model, we record wilting processes of real plants and compare the resulting data with those coming from our simulation of the water transport. This evaluation is however fundamentally limited by the practical difficulties in measuring the water content in individual plant parts. These measurements can be acquired in two different ways: For highly accurate results, the plant part is separated by cutting it off and then immediately weighted on a highly sensitive scale. It is then slowly but thoroughly dried (e.g. 15 hours at 80°C [121]) and weighted again. The destructive nature of this method prohibits obtaining a time series of water measurements.

Alternatively, different types of measurement devices for sap flow can be attached to plant stems [251]. Absolute water values can be obtained through integration of the flow, but these devices are rigid and comparably heavy and thus significantly interfere with the dynamic deformation through wilting. They furthermore require a minimum steam diameter making them not applicable for most of our examples (especially not for the thin branches that start wilting first).

We can, however, measure the total water content of a plant over time. To this end, the plant is carefully removed from the surrounding soil and placed in a pot filled with dry sand on top of a centigram precise scale. Since all water inside the setup is now situated inside the plant, a weight loss of the whole system during the wilting process directly corresponds to a water loss of the plant.

The initial water content of the plant is computed from the weight difference of the plant at the beginning and end of the wilting process. The virtual plant is modeled after the real plant by using photographs from multiple directions. The radii of each branch and the leaf sizes are adjust to fit the real plant. For all our experiments, we modeled plants with 500 to 2000 nodes. We can then simulate the wilting of the virtual plant and compare its water loss curve directly to the real plant. Figure 4.6 shows the results of this experiments on two different young tomato plants. We find, that both the real and simulated plant exhibit the same exponential water loss over time.

To evaluate not just the total amount of water inside the plant but also its distribution we resort to a qualitative analysis due to the aforementioned reasons. In Figure 4.5, we compare the plant shape resulting from our wilting model to two different naive ones. We find, that our approach is superior in catching the variations in plant stiffness which gives credit to our water distribution model.

4.5.2 Environment and Material Calibration

The two main parameters used for calibration are the water loss rate δ (expressed in g s^{-1} cm^{-1}) and the elasticity mapping slope κ (expressed in g^{-1}). While δ is a combination of the environment (e.g. temperature, humidity, irradiation, etc.) and plant material (different plants can evaporate a different amount of water per leaf area and time), κ is a pure material parameter the describes how sensitive a plant is to water loss. In our experiments we keep the environmental conditions constant by performing them inside a room with constant temperature, humidity and illumination. We explore how different combinations of these values affect the wilting process in Figure 4.7.

4.5.3 Descriptive Power

We now show a variety of different results that can be achieved by our method.

By tuning the model's parameters, we can simulate various types of crop plants, and we are not constrained to plants growing up from the soil. Figure 4.8 shows an example of the simulation of a hanging ivy plant. Ivy is notoriously resilient to water absence, so we tuned down the water loss rate and the slope of the mapping from water to elasticity. At the beginning of the simulation, the main stems are strong enough to keep their shape even if the plant is growing from top to bottom. After some time without water, the plant loses its stiffness and starts to fall down following gravity.

Our model is also capable of handle larger plants, such as a found in industrial greenhouse environments. These are often forced into a specific and more efficient shape (in terms of growth and harvest) by attaching them to a supporting structure such as grids or wires. We implement this through positional constraints on specific

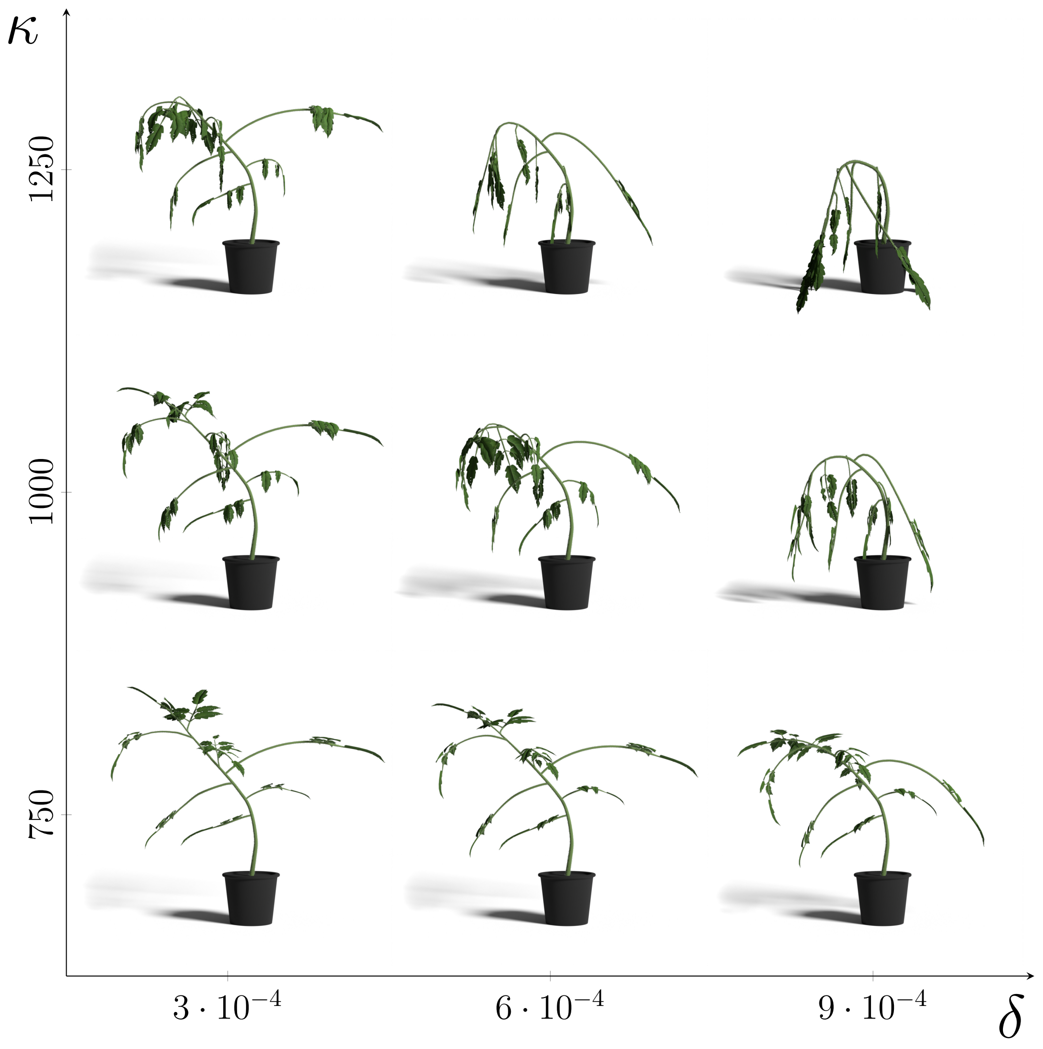


Figure 4.7. The tunable parameters determine the wilting behavior of the plant. As the water loss rate (δ) of the plant increases, the plant loses water more rapidly and wilts faster. In contrast, increasing the steep (κ) in the mapping from the water content to the elasticity modulus determines the difference in stiffness between parts retaining the water differently.

plant nodes in the physical solver. An example of a 2000 node tomato plant is shown in Figure 4.9. Scenarios like this can be used to train AI tasks in industrial farming applications [123].

Being derived from the connections on a tree structure, our water diffusion model can also be used to model certain common diseases and physical damages that alter the plants ability to transport water to parts of it. To represent this kind of damage, we increase the water flow resistance of a connection (or set it to an infinite value to completely disabled transport) to reduce the amount of water that can reach the top parts of the plant. We can also tune down (or zero out) the loss rate of the bottom part of the plant if we want it to be in a healthy state. In Figure 4.10, we show an example of disease modeling where a point of the main stem is damaged

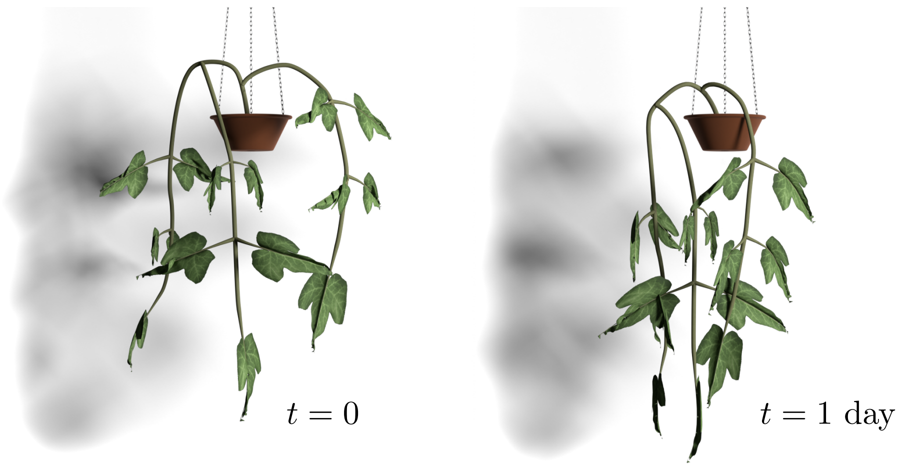


Figure 4.8. Our technique generalizes well to different types of plants. Even if hanging plants grow downward, their branches could be stiff and resistant. As the water amount decreases, they tend to fall down, only affected by gravity.

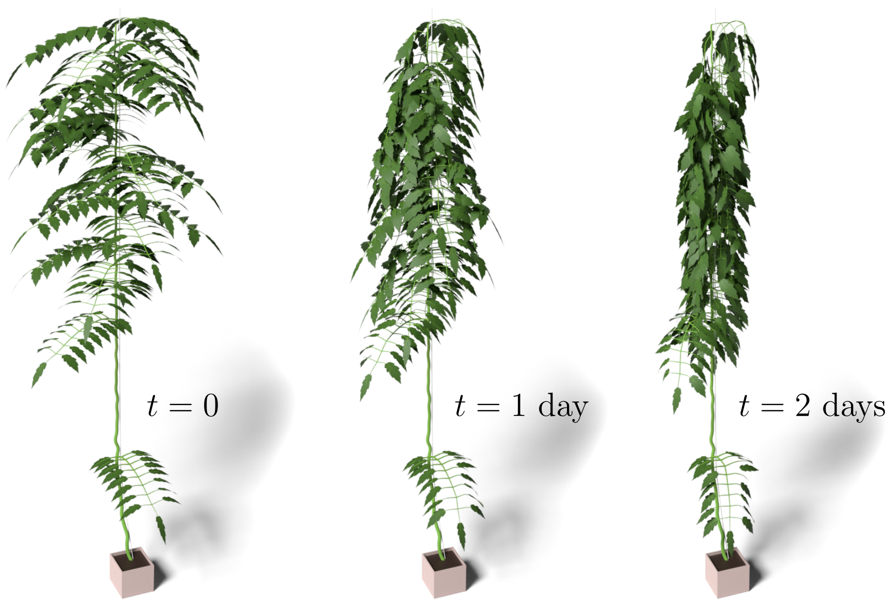


Figure 4.9. Simulation of a large plant with 1954 nodes. As it is common in industrial greenhouses, the main stem is fixed to a supporting structure at several places, which prevents the plant from entirely collapsing during wilting.

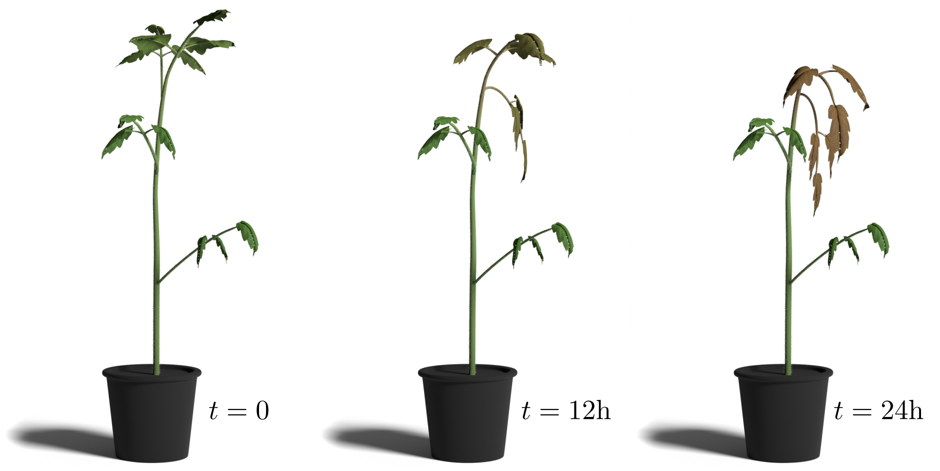


Figure 4.10. Plants can be affected by diseases that prevent water from diffusing correctly. With our model, we can simulate the entire plant and still obtain this behavior by simply cutting or weakening a connection at some node. The cut prevents water from flowing in some parts of the plant, which will wilt without affecting the sections connected to the root.

and cannot bring water to the top branches and leaves. We see that the top part of the plant dries and wilts, whereas the rest of the plant keeps its shape.

4.5.4 Performance

As discussed in Section 4.4, our method consists of simulating the water diffusion inside the plant, and then mapping the water content to the stiffness, leveraging on the PBD solver to simulate the dynamics of the plant. Since our method relies on external solvers for simulating the plant dynamics, we evaluate the performance by coupling our method with a state-of-the-art PBD simulator, comparing at each frame the time required by solving for the physics of the system with the execution time of our water model.

In Figure 4.11 we show how the execution time for a single simulated frame is distributed between the evaluation of the water model (blue) and the simulation of the physics dynamics with the PBD solver (orange). The time for the water model evaluation also accounts for the time required by the mapping of the water to the stiffness. We notice that the physics solver computationally dominates the simulation step across different plants with varying number of nodes. As both methods scales linearly, the time ratio remains constant around 1%.

The amount of simulated frames required is situational dependent. The dynamic solver must converge to an equilibrium state, which depends on the plant geometry. Since the water model is evaluated much faster, the stiffness is adjusted after each PBD iteration. Smooth animations, such as shown in the supplemental video, generally require more frames for convincing visual quality, e.g. the teaser scene was produced with 100 k simulator frames. On the other extreme, producing static images of a few wilting states require far less frames, e.g. the complex plant in Figure 4.9 was computed with just 2000 frames.

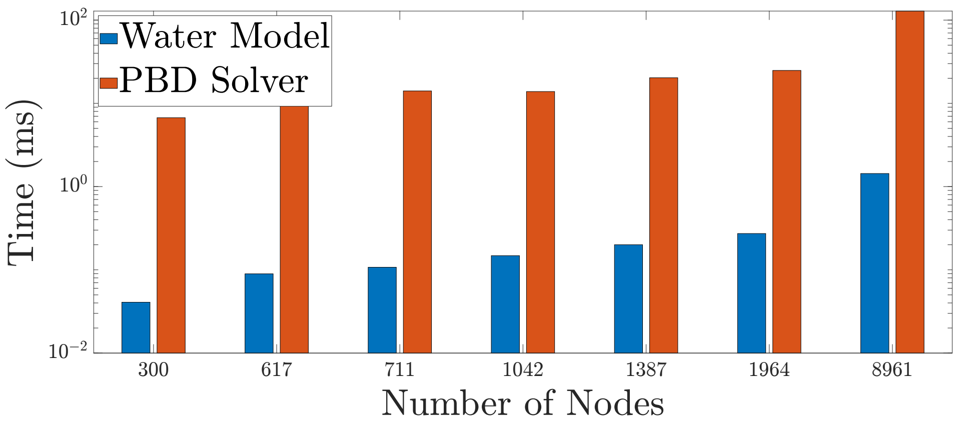


Figure 4.11. Distribution of the average execution time between the water model (blue) and the PBD solver (orange) at every time step for different plants. The PBD solver is computationally dominant independently on the number of nodes. The experiments have been carried out on a machine equipped with an Intel i7-10700K CPU and 32 GB of main memory.

4.6 Conclusion, Limitations, and Future Work

We propose a physically-inspired diffusion model for simulating water distribution inside a plant. Our model follows a computational scheme that can be computed efficiently, and we also propose an approximation that introduces a negligible error and scales well to problems with large sizes. The water diffusion model is defined to work independently in a parametric space and can be integrated with any physical simulator for simulating the dynamics of a wilting plant, other than existing approaches for simulating wilting leaves. In contrast to hand-crafted wilting animations and ad-hoc simulations, our pipeline exposes only two parameters that can be adjusted to simulate a variety of different plants.

The definition of the problem can be adjusted with further intuitive tuning to match plants affected by certain diseases and physical damages. Finally, we evaluate the model showing that its results match experimental data and that the simulated wilting process is visually convincing when compared to real wilting plants.

Our approach can be extended in several ways: Ligneous (*woody*) plant parts are more resistant to deformation through water loss. This could be modeled by allowing varying κ values for different plant parts. The current diffusion model also neglects the capability of certain plants to react to external stress, such as closing the leaf cells to reduce water evaporation. However, these complex processes are still not fully researched, which makes them very hard to model in a computer graphics contexts. It would furthermore be interesting to directly couple more advanced models for leaves, fruits, and flowers [111, 120] with our water model to increase realism of plant parts that are not modeled well by elastic rods.

Chapter 5

Massive Uniform Mesh Decimation via a Fast Intrinsic Delaunay Triangulation

Triangular meshes are still today the data structure at the main foundations of many computer graphics applications. With the increasing demand in content variety, a lot of effort has been and is being put into developing new algorithms to automatically generate and edit geometric assets, with a particular focus on 3D scans. However, this kind of content is often generated with a dramatically high resolution, making it impractical for a large variety of tasks. Furthermore, procedural assets and 3D scans largely suffer from poor geometry quality, which makes them unsuitable in various applications. We propose a new efficient technique for massively decimating dense meshes with high vertex count very quickly. The proposed method relies on a fast algorithm for computing geodesic farthest point sampling and Voronoi partitioning, and generates simplified meshes with high-quality uniform triangulations. The work presented in this chapter has been realized in collaboration with Daniele Baieri (MSc.), who largely helped in finding a relevant literature and deeply studied the pre-processing step to produce an accurate complexity analysis, and prof. Emanuele Rodolà, to whom goes the credit for improving the presentation and for being an invaluable guide throughout the relevant related literature. The seminal results presented in this chapter have been published as a pre-print on *arXiv* [158].

5.1 Introduction

In recent years, the request for content variety in many computer graphics applications has increased dramatically. Especially in the entertainment industry (*e.g.*, videogames and movies), the demand for a wide range of different assets is becoming overwhelming. Artists and researchers work continuously to design new methods for automating and simplifying content production and editing.

Technologies such as 3D scanning have been game-changing in this regard, since a wide variety of assets can be produced in virtually no time from real-world objects. However, these types of technologies notoriously suffer from high-frequency noise

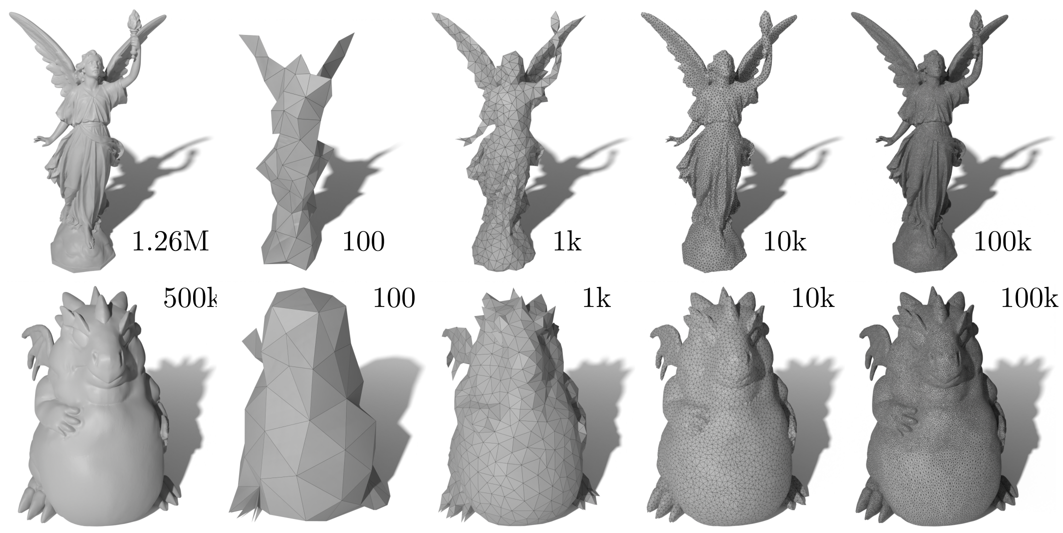


Figure 5.1. Examples of our method applied to dense shapes. From left to right: the original high-density shape, followed by the corresponding remeshed variants, with an increasing number of vertices. The sampled points are evenly spaced, and the triangulation is uniform, without sharp angles or degenerate triangles.

and poor quality of generated connectivity, other than the output being extremely dense in terms of vertex count [26].

This is where classical geometry processing comes into play, providing a whole range of mesh simplification and remeshing tools from its vast literature, in order to reduce the complexity of the geometry and increase the overall quality. Mesh processing is still today a leading topic in computer graphics, and a lot of research in this area focuses on simplifying complex geometries or increasing their quality. However, the simplification task always focused on deleting a limited amount of geometric elements to preserve the details of the original geometry as much as possible [90, 125]. On the other hand, remeshing algorithms completely reorganize and redesign the mesh in order to obtain a high-quality model that fits the original shape well, but without caring much about the density of the output geometry [119].

Dense meshes and high vertex count are very effective for representing high-frequency details of the surface, which can be useful in many applications, like 3D printing. However, there is a variety of other fields where a low polygon count is mandatory. For example, in real-time rendering a small number of faces is vital to guarantee efficiency [289, 284], and in spectral geometry processing a lot of research is devoted to design techniques for speeding up computation on meshes with many vertices [187, 161], which would otherwise be unfeasible. Our goal is to provide a novel approach to the mesh simplification task for addressing the problem of massively decimating large meshes, while still retaining high-quality connectivity.

To summarize, our contribution is a fast algorithm based on the geodesic Delaunay triangulation, which can efficiently simplify very large meshes, generating a low-resolution model which approximates the underlying geometry and provides robust connectivity without poor quality elements such as sharp angles or degenerate triangles (see Figure 5.1). To achieve this task, we also provide an efficient algorithm to approximate farthest point sampling based on the geodesic metric and geodesic Voronoi partitioning, and we show how to improve the robustness of our algorithm through a smart surface resampling strategy.

5.2 Related Work

Our method falls into the category of mesh simplification algorithms, as its goal is to massively reduces the vertex and face count of the input shape. However, rather than a subtractive method which iteratively removes geometric elements or clusters vertices [33], we propose an algorithm that samples the surface and computes a new connectivity. For this reason we consider it appropriate relating it to remeshing techniques as well.

5.2.1 Remeshing

Remeshing is a classic topic in geometry processing which has been vastly studied for decades. In general, the goal of remeshing algorithms is to alter geometry and connectivity of an input mesh in order to improve some measure of quality, while providing a good approximation for the input.

Part of the theory behind our proposal is grounded in traditional remeshing techniques. The seminal studies for isotropic (*i.e.*, guaranteeing well-shaped elements) remeshing [28, 52, 239] leverage the notion of restricted Delaunay triangulation, denoted $\mathrm{Del}_{\mathcal{S}}(\mathcal{P})$ for a surface $\mathcal{S}$ and a point set $\mathcal{P}$ (defined as the sub-complex of the Delaunay triangulation of $\mathcal{P}$ whose dual Voronoi faces intersect $\mathcal{S}$), whereas others impose constraints on the shape of geometric elements and solve optimization problems [32].

A significant fraction of remeshing algorithms are based on Centroidal Voronoi Tessellations [74], which are Voronoi decompositions such that for each texel the generating point is also the center of mass. Particularly relevant to our work are the extensions of CVT where the Voronoi decomposition is computed based on the geodesic metric, also known as geodesic-based remeshing methods: Wang *et al.* [281, 280] showed two algorithms to compute intrinsic (*i.e.* geodesic) CVT on triangle meshes. Liu *et al.* [149] improved this previous proposal, by showing how to avoid local optimal solutions and proposing an algorithm with provable probablistic convergence to the global optimum. Subsequently, a method to compute intrinsic Delaunay triangulations as the dual of intrinsic CVT was developed [147].

Yi *et al.* [295] and Liu *et al.* [148] proposed simplification-based remeshing approach which provide satisfying approximations (as they both explicitly optimize for the Hausdorff metric) and allow for control over output vertex count. However, both these methods exhibit super-linear time complexity ($\mathcal{O}(n\log(n))$ and $\mathcal{O}(nK\log(K))$, respectively) both in theory and in practice.

For a complete review of the remeshing literature, we refer to the survey from Khan *et al.* [119] and the book from Botsch *et al.* [33].

5.2.2 Mesh Decimation and Simplification

Reducing the size of a mesh, in terms of number of vertices and polygons, is a task akin to remeshing which is mainly motivated by the necessity for lightweight 3D models in a wide range of graphics applications. A very popular algorithm, which can often be found implemented in 3D graphics tools, is due to Garland *et al.* [90]: their method iteratively contracts arbitrary pairs of vertices (thus also handling disconnected components, allowing to join them), maintaining suface approximation error via quadric metrices. This greedy scheme, which was formalized by Kobbelt *et al.* [125], is actually common to several previously proposed simplification methods [106, 104, 242], which only differ in the chosen heuristics for pair selection, error metric, and simplification operation (*e.g.* vertex contraction, edge collapse).

Other works focused on recomputing a decimated triangulation of a surface trying to optimize different quality criteria, as examined by the survey from Bern *et al.* [27]. More similar in spirit to our work is the idea proposed by Li *et al.* [138], which tries to condense remeshing and simplification in a single algorithm that produces a non-obtuse decimated triangulation of a given surface. However, the method offers no guarantees on the decimation task, as the resulting meshes could even be more dense than the input.

Mesh decimation has also been studied in recent literature, under novel points of view or exploiting new geometric optimization frameworks. DeCoro *et al.* [66] proposed a scheme for GPU implementations of quadric metric decimation, showing that the efficiency gains are sufficient to run the algorithm in real time. The spectral geometry processing framework can be applied to mesh decimation as showed in [133], by requiring the simplified mesh to preserve spectral features of the original. Finally, Potamias *et al.* [219] leveraged geometric deep learning in order to optimize a data-driven decimation model which has the advantages of being differentiable and also considerably faster than traditional algorithms.

5.3 Method

We provide a detailed explanation of our method. First, we give a high-level overview of the algorithm, highlighting its properties. Then, we provide an in-depth description of the single steps, focusing on our design choices aimed at improving the performance and the quality of the output.

5.3.1 Delaunay Remeshing

We are given a triangular mesh $\mathcal{M} = (V, E, T)$, where

- $V \subset \mathbb{R}^3$ is a set of vertices;
- $E \subset V^2$ is a set of edges between vertices;
- $T \subset V^3$ is a set of triangles composing the surface.

and we denote by $\delta_{\mathcal{M}} : V \times V \to \mathbb{R}$ the geodesic distance between pair of vertices on $\mathcal{M}$.

Our approach consists in sampling the surface $\mathcal{M}$ evenly, and using the geodesic distance to define a non-Euclidean Delaunay triangulation between these vertices in the mesh domain. The process is summarized in Algorithm 5.1.

Algorithm 5.1 Delaunay remeshing algorithm.

```
procedure DelaunayRemeshing($\mathcal{M} = (V, E, T)$, $n$)
    FPS ← FarthestPointSampling($\mathcal{M}$, $n$, $\delta_{\mathcal{M}}$)
    VP ← VoronoiPartitioning($\mathcal{M}$, FPS, $\delta_{\mathcal{M}}$)
    DT ← Voronoi2Delaunay($\mathcal{M}$, VP)
    return (FPS, DT)
end procedure
```

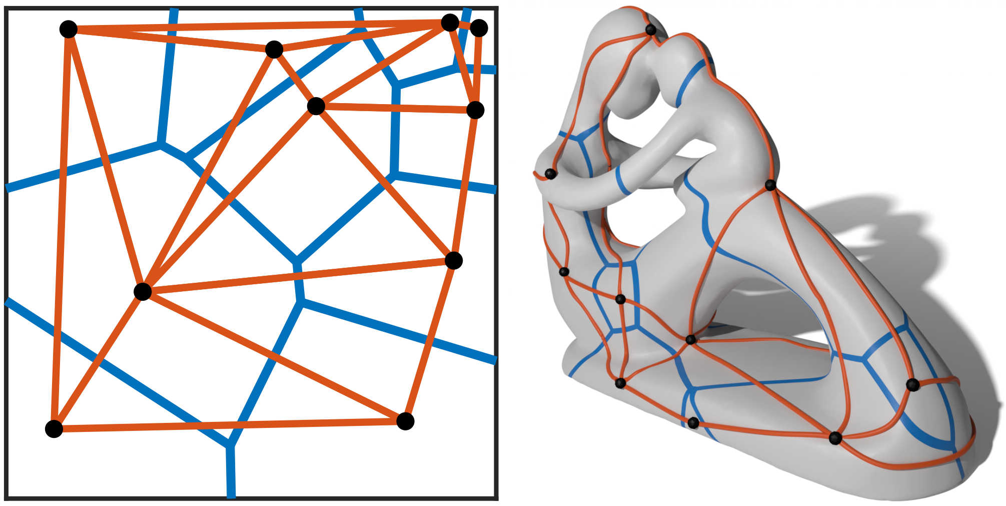


Figure 5.2. Examples of duality between the Voronoi partitioning and Delaunay triangulation problems. The centroids of adjacent Voronoi texels (separated by the blue lines) are connected together, forming a triangular Delaunay connectivity (orange lines).

We first compute a farthest point sampling of $\mathcal{M}$, using the geodesic distance $\delta_{\mathcal{M}}$ as metric. This ensures that $\mathcal{M}$ is uniformly sampled, and that the sample points are approximately evenly spaced according to the metric of the mesh.

Since the mesh domain is equipped with a non-Euclidean metric, the Delaunay triangulation cannot be computed with well established approached like the Bowyer-Watson algorithm [34, 283]. Instead, we rely on the duality between the Delaunay triangulation and the Voronoi partitioning [15], which has already been generalized, exploited and successfully used on manifold domains [280, 147]. The example in Figure 5.2 shows that this duality is independent on the choice of metric, and holds both in planar and non-Euclidean domains.

Using the geodesic metric, we partition the mesh into Voronoi texels, each having as a centroid a vertex from the farthest point sampling. The partitioning is then used to compute the geodesic Delaunay triangulation, producing a triangular connectivity of the sample points. Finally, we move back to the Euclidean domain, and we connect the points using straight edges in $\mathbb{R}^3$, following the connectivity computed by the algorithm.

By construction, the sampled vertices are uniformly sampled from the surface with approximately even spacing. And, since the triangulation is performed directly on the mesh domain, the triangles are guaranteed to intersect the original

Algorithm 5.2 Voronoi Farthest Point Sampling algorithm.

```
procedure VoronoiFPS($\mathcal{M} = (V, E, T)$, $n$)
    $i \leftarrow$ random index in $[1, |V|]$
    $S \leftarrow \{i\}$
    $D \leftarrow \delta_{\mathcal{M}}(V, V_i)$
    $P \leftarrow$ a vector of length $|V|$ initialized to $i$
    for $h \leftarrow 2$ to $n$ do
        // Find the next sample
        $p \leftarrow \arg\max_{1 \leq j \leq |V|}(D_j)$
        $D_p \leftarrow 0$
        $S \leftarrow S \cup \{p\}$
        $Q \leftarrow \{p\}$
        // Expand from $p$, updating distances and partitions
        while $Q \neq \emptyset$ do
            $q \leftarrow \arg\max_{j \in Q}(D_j)$
            $Q \leftarrow Q \setminus \{q\}$
            for $r \in \text{adj}(q)$ do
                if $D_r > \delta_{\mathcal{M}}(V_p, V_r)$ then
                    $D_r \leftarrow \delta_{\mathcal{M}}(V_p, V_r)$
                    $Q \leftarrow Q \cup \{r\}$
                    $P_r \leftarrow p$
                end if
            end for
        end while
    end for
end procedure
```

surface. The algorithm produces a triangulation which follows the original shape and whose triangles have a roughly uniform area. Moreover, since the triangulation is Delaunay, it provides robustness guarantees on the shape of the triangles.

5.3.2 Voronoi FPS

The major bottleneck of the procedure we described so far is the computation of the farthest point sampling, followed by Voronoi partitioning, both occurring in the mesh domain. To overcome this problem, we decide to combine the two steps in a single procedure, exploiting the properties of both to design an algorithm which is faster by orders of magnitude.

By approximating the geodesic distance with a shortest path on the mesh-graph, we setup a modified Dijkstra algorithm with early-stops, as shown in Algorithm 5.2. Here, we denote the i-th vertex of the mesh as V_i and, with an abuse of notation, we write $\delta_{\mathcal{M}}(V, x)$ to identify the vector of geodesic distances from all the vertices in V to the vertex x.

Theorem 5.1. *Given a triangular mesh $\mathcal{M}$ and a number of vertices n, Algorithm 5.2 computes a geodesic farthest point sampling S of size n and partitions $\mathcal{M}$ with a geodesic Voronoi decomposition whose centroids are the points in S.*

Proof. To prove the correctness of the algorithm, we first show that the vector D always stores the shortest distance from the set of sample points. Before the first iteration, any point can be selected and the vector D stores the shortest distances from that point. When a new point q is selected, the mesh-graph is visited starting from q. For any visited vertex r, the value D_r is updated with the distance $\delta_{\mathcal{M}}(V_p, V_r)$ if and only if $\delta_{\mathcal{M}}(V_p, V_r)$ is smaller than the current distance from the sample set.

Since D stores the shortest distance from the current sample set, taking the point that maximizes that distance at each iteration guarantees that at the end of the algorithm S contains a valid geodesic farthest point sampling. Moreover, since the partition of any vertex r is updated with the current sample point p if and only if the distance $\delta_{\mathcal{M}}(V_p, V_r)$ is smaller than the distance to any other sample point in S, at the end of the algorithm the vector P contains the correct geodesic Voronoi partitioning of $\mathcal{M}$ which uses the sample points S as centroids. □

To study the asymptotic complexity of the algorithm, we need to focus our attention on the fact that, at each iteration, we visit an increasingly smaller subset of the mesh. Since the vertices are added to the queue only if the distance from the current sample p is smaller than the distance from any other sample, the visit only covers the current Voronoi partition of p. By assuming a uniform distribution of the vertices, at the iteration h the visited sub-graph has size approximately $|V|/h$. By using the appropriate data structures (*i.e.*, priority queues, adjacency graphs), the complexity of a single iteration is $\mathcal{O}\left(\frac{|V|+|E|}{h}\log\left(\frac{|V|}{h}\right)\right)$. Taking out the maths, the total complexity of the algorithm is

$$\mathcal{O}\left(\left(|V|+|E|\right)\log\left(|V|\right)\mathbf{H}_n\right) \approx \mathcal{O}\left(\left(|V|+|E|\right)\log\left(|V|\right)\log(n)\right) , \tag{5.1}$$

where the harmonic number $\mathbf{H}_n$ is approximately the natural logarithm [57].

Theoretically, the loop is dominated by the $\mathcal{O}(|V|)$ $\arg\max$ function, which would keep the total complexity to $\mathcal{O}(n|V|)$. However, finding the $\arg\max$ in a vector is, in practice, a fast operation which extensively takes advantage of the cache locality, and thus does not have a noticeable impact on the algorithm's performance.

5.3.3 Dual Triangulation

Once we obtained the Voronoi partitioning of a farthest point sampling, computing the Delaunay triangulation is quite straightforward. We start from an empty set of edges E' and we iterate over all the edges $e \in E$ of the mesh $\mathcal{M}$. As long as the edge $e = (V_i, V_j)$ connects two nodes belonging to different partitions $P_i \neq P_j$, we create a link between V_{P_i} and V_{P_j} and we add it to E'.

This process leaves us with a graph $G = (S, E')$, which represents the mesh-graph of the decimated mesh. To obtain the decimated mesh $\mathcal{M}'$, we visit the graph G and find all the three-cycles, which define the faces of $\mathcal{M}'$. Figure 5.3 depicts an example summarizing the entire procedure.

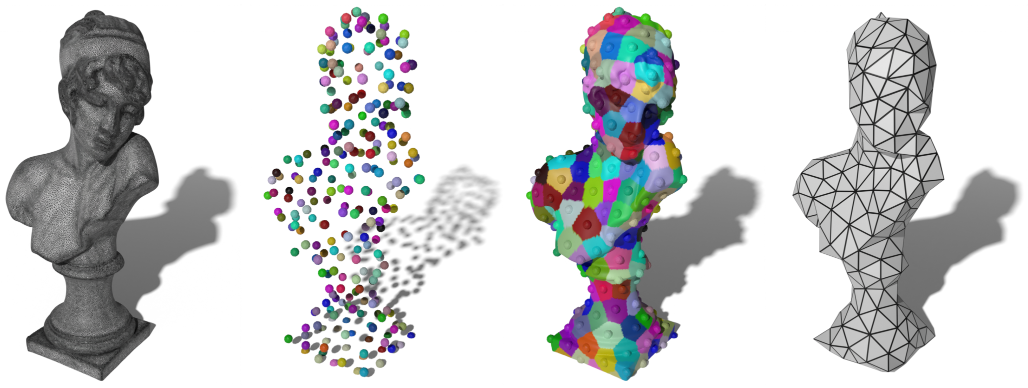


Figure 5.3. An example of the behavior of our algorithm. The input geometry (first) is sampled and partitioned using Algorithm 5.2 (second). The connectivity of the sampled points is then computed using the underlying original geometry (third) and the resulting triangulation is given in output as a mesh (fourth).

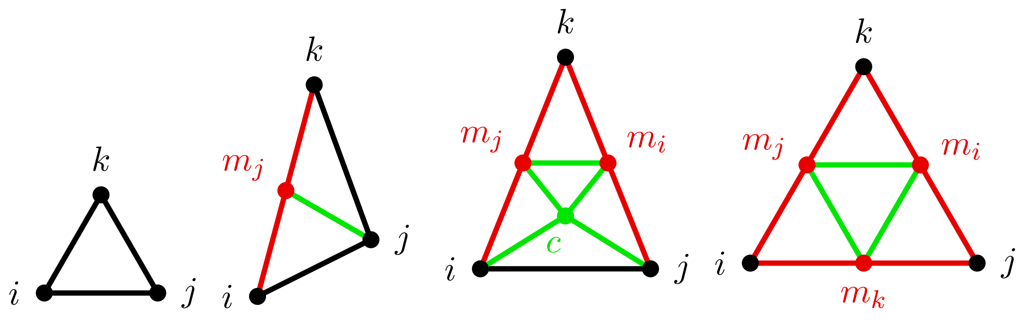


Figure 5.4. Resampling scheme of the triangles, depending on the number of split edges. Oversize edges are identified in red, and the added connectivity in green.

5.3.4 High-Density Sampling

Sometimes triangles in the input mesh may be highly non-homogeneous, and very large triangles can have a negative impact on the uniformity of the remeshed shape. Generalizing Algorithm 5.1 to include face sampling is certainly possible; yet, it would be very expensive in terms of computational costs, as computing exact geodesics which include face sampling is rather costly and would not allow to take advantage of the benefits of Algorithm 5.2.

We propose an alternative method based on resampling the mesh as a pre-processing step. Statistically, triangular meshes have triangle count ~3 times the number of vertices. Given the total area $A_{\mathcal{M}}$ of the original shape $\mathcal{M}$, we can estimate the average triangle area for the output mesh as $A_E = {}^{A_{\mathcal{M}}}/{}_{3n}$, being n the number of vertices of the remeshing. An equilateral triangle of area A_E has each side of length $\ell = \sqrt{{}^{2A_E}/{}_{\sqrt{3}}}$, so we split in half every edge with length greater than ℓ and we insert a new vertex in the midpoint. We then split the mesh triangles depending on the number of split edges incident on that triangle, and iterate the process until no edges are oversize.

We provide the complete pseudocode for the resampling algorithm in Algo-

Algorithm 5.3 Mesh resampling algorithm.

```
procedure Resampling($\mathcal{M} = (V, E, T)$, $\ell_{\max}$)
    $\xi \leftarrow +\infty$
    while $\xi > \ell_{\max}$ do
        // Mark all long edges
        for $e \in E$ do
            $\xi \leftarrow \max(\|e\|_2, \xi)$
            if $\|e\|_2 > \ell_{\max}$ then
                Mark $e$ for splitting
            end if
        end for
        // Split according to scheme in Figure 5.4
        for $t \in T$ do
            $\omega_t \leftarrow$ number of marked edges incident on $t$
            $v_0, v_1, v_2 \leftarrow$ vertices opposite to edges $e_0, e_1, e_2$
            $m_0, m_1, m_2 \leftarrow$ midpoints of edges $e_0, e_1, e_2$
            if $\omega_t = 0$ then
                continue
            else if $\omega_t = 1$ then
                $e_i \leftarrow$ marked edge
                $T \leftarrow T \cup \{(v_i,\ m_i,\ v_j),\ \ (v_i,\ m_i,\ v_k)\}$
            else if $\omega_t = 2$ then
                $e_i, e_j \leftarrow$ marked edges
                $c \leftarrow \frac{1}{4}(v_i + v_j + m_i + m_j)$
                $T \leftarrow T \cup \{(m_i,\ m_j,\ v_k)\}$
                $T \leftarrow T \cup \{(m_i,\ m_j,\ c),\ \ (v_i,\ v_j,\ c)\}$
                $T \leftarrow T \cup \{(m_i,\ v_j,\ c),\ \ (v_i,\ m_j,\ c)\}$
            else if $\omega_t = 3$ then
                $T \leftarrow T \cup \{(m_0,\ m_1,\ m_2),\ \ (v_0,\ m_1,\ m_2)\}$
                $T \leftarrow T \cup \{(m_0,\ v_1,\ m_2),\ \ (m_0,\ m_1,\ v_2)\}$
            end if
            $T \leftarrow T \setminus \{t\}$
        end for
        Recalculate $E$ from $T$
    end while
end procedure
```

rithm 5.3. The algorithm recursively split the edges in half and recompute the connectivity of the mesh, until no edge is longer than the given threshold.

At every recursion, all the edges longer than the threshold $\ell_{\max}$ are marked for splitting and the length ξ of the longest edge is computed.

Then, we perform an iteration over all the triangles, searching for the marked edges incident on every triangle t. The splitting scheme is summarized in Figure 5.4.

- If the edges incident on t are not marked, than t can be kept as is, and no new triangles or edges are added.
- If t has only one marked incident edge, we split that edge in half and create two new triangles. However, since the splitted edge cannot be longer than the sum of the other two edges, the two halves created are shorter than the threshold. Moreover, the newly added edge is bounded by the two short edges. Hence, the new edges and triangles will not be splitted at the next iteration.
- If t has two marked incident edges, we split them in half and connect the midpoints and create a new triangle and a quad. This new connection is half the shorter edge, hence the triangle has at least one short edge. Even if the quad could be splitted in two triangles, we decide to add a new vertex at the barycenter of the quad and create four triangles, to enforce a symmetry in the split and mitigate the formation of new long edges. We have no guarantees for these triangles, but at least two of them must have at least one short edge.
- If the edges of t are all marked, we split them and connect all the midpoints, creating four new triangles. For these triangles, we cannot make any assumption.

Finally, if t has at least one marked edge, we remove it from the old connectivity, as it has already been replaced.

Once all the triangles have been split, we recompute all the edges using the triangles connectivity and we start a new iteration.

By using the appropriate data structures (*i.e.*, hash sets and hash maps), we can achieve the operations for each edge and triangle in constant time, computing each iteration in $\mathcal{O}(|E| + |T|)$ time, and we avoid an expensive recomputation of the triangles adjacency map at every loop cycle. Furthermore, expanding the splits from the edges allows us to avoid unnecessary splits of triangles that are already small enough.

Theorem 5.2. *Let $\mathcal{M} = (V, E, T)$ be a triangular mesh and n the number of output vertices for our decimation algorithm. Given the definition for $\ell_{max} = \sqrt{2A_{\mathcal{M}}/3n\sqrt{3}}$, $\hat{e} = \arg\max_{e \in E} \|e\|$, $q = \|\hat{e}\|/A_{\mathcal{M}}$, the time complexity of our resampling procedure is*

$$\mathcal{O}\left(qn \cdot \log(qn) \cdot |T|\right) . \tag{5.2}$$

Proof. We start by observing that the outer loop of the algorithm runs for exactly $d = \lceil \log_2 (\|\hat{e}\|/\ell_{\max}) \rceil$ iterations, which is the minimum number of subdivisions which would make the longest edge in $\mathcal{M}$ shorter than $\ell_{\max}$. Thus, the worst-case complexity of the outer loop is $\mathcal{O}(\log(\|\hat{e}\|/\ell_{\max}))$. Plugging the definition for $\ell_{\max}$ and removing constant factors, we may write the outer loop complexity as

$$\begin{aligned}\mathcal{O}\left(\log\left(\frac{\|\hat{e}\|}{\ell_{\max}}\right)\right) &= \mathcal{O}\left(\log\left(\frac{\|\hat{e}\|}{\sqrt{2A_{\mathcal{M}}}/\sqrt{3n\sqrt{3}}}\right)\right) \\ &= \mathcal{O}\left(\frac{1}{2}\log\left(\frac{n\|\hat{e}\|}{A_{\mathcal{M}}}\right)\right) = \mathcal{O}\left(\log(qn)\right) .\end{aligned}$$

The inner loop complexity analysis, on the other hand, is complicated by the variations in the size of the mesh. We perform the analysis by only considering the triangle loop, since it is often assumed that for triangle meshes $|E|, |T| = \mathcal{O}(|V|)$. Let $\{T_i\}_{i=0}^{d}$ be the sequence of triangle sets generated through the execution of the algorithm, where $T_0 = T$. The sequence is non-decreasing in size (since if there are no splits, the algorithm terminates), thus the last element will be the largest.

In order to evaluate $|T_d|$, we establish the following recurrence equation following the splitting schemes from Figure 5.4, where $T_i^k = \{t \in T_i \ : \ t \text{ was split } k\text{-ways}\}$:

$$|T_{i+1}| = |T_i^0| + 6|T_i^1| + 15|T_i^2| + 4|T_i^3| \,. \tag{5.3}$$

To estimate $|T_i^k|$, we have to consider the probability of any edge being oversize at the i-th step p_i. With a reasonable assumption of independence, one can find for any triangle $t \in T_i$:

$$P(t \in T_i^k) = \begin{cases} (1-p_i)^3 & k = 0\,, \\ 3p_i(1-p_i)^2 & k = 1\,, \\ 3p_i^2(1-p_i) & k = 2\,, \\ p_i^3 & k = 3\,. \end{cases} \tag{5.4}$$

And conclude that $\mathbb{E}\left[|T_i^k|\right] = P(t \in T_i^k)|T_i|$. Substituting back into Equation (5.3) and carrying out the math, you get

$$|T_{i+1}| = (-6p_i^3 + 6p_i^2 + 3p_i + 1)|T_i| = c(p_i)|T_i| \,. \tag{5.5}$$

We consider the worst case for increase in number of triangles, that is, p_i constant and set to $\max_{p\in[0;1]} c(p)$. The result would be $p = 0.8604$ and $|T_{i+1}| = c(0.8604)|T_i| = 4.201|T_i|$. In order to solve for T_d, we apply the definition:

$$\begin{aligned} |T_i| &= \sum_{j=0}^{i-1} 4.201|T_j| = 4.201|T_{i-1}| + \sum_{j=0}^{i-2} 4.201|T_j| \\ &= 4.201|T_{i-1}| + |T_{i-1}| = 5.201|T_{i-1}| \\ &= 5.201 \cdot 5.201|T_{i-2}| = \cdots = 5.201^{i-1} \cdot 4.201|T_0| \,. \end{aligned} \tag{5.6}$$

And finally, $|T_d| = 5.201^{d-1} \cdot 4.201|T|$. Recalling that $d = \mathcal{O}(\log(qn))$, the asymptotic complexity of our resampling algorithm is:

$$\mathcal{O}\left(\log(qn) \cdot 2^{\mathcal{O}(\log(qn))} \cdot |T|\right) = \mathcal{O}\left(qn \cdot \log(qn) \cdot |T|\right) \,. \tag{5.7}$$

□

This procedure certainly introduces a significant computational overhead, as it increases the size of the input mesh, on top of its already significant cost. However, it is very helpful with respect to output quality when dealing with highly non-homogeneous meshes, as shown in the example from Figure 5.5.

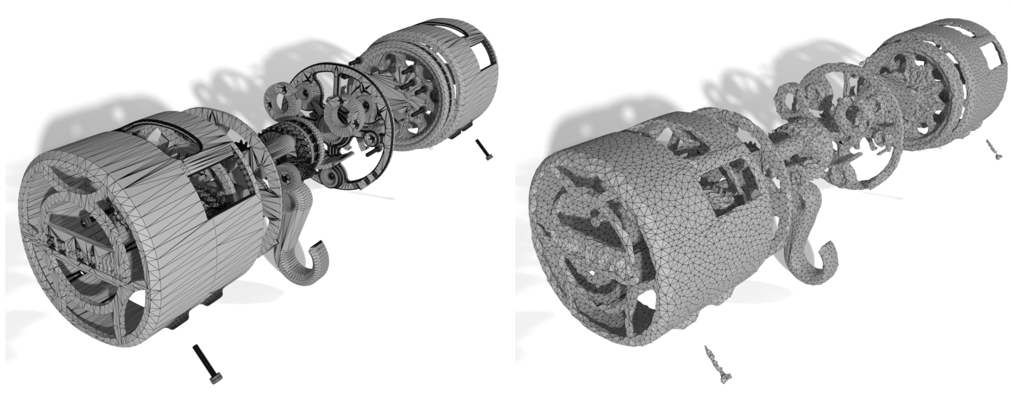


Figure 5.5. An example of decimation with surface resampling. The long and skinny triangles in the original shape disappear in the remeshed output. The triangles are uniform, while still preserving the original complex and disconnected geometry.

5.4 Results

We experimentally validate our algorithm and discuss the results of our tests. We first discuss the quality of the output meshes of our algorithm, providing both quantitative and qualitative results. Then, we examine the execution time and the performance of our implementation. Finally, we briefly discuss the quality improvement given by the resampling of large triangles.

All the experiments have been performed on a machine equipped with an Intel i7-10700K CPU (16 cores and 3.80 GHz clock frequency) and 32 GB of main memory at 3600 MHz frequency. The quantitative results are averaged on a set of 20 shapes from the Thingi10K dataset [303].

5.4.1 Remeshing Quality

To validate our algorithm, we run a batch of experiments on a small subset of Thingi10K for several output resolution values. Since our goal is to provide a uniform remeshing of the input shape, we experimentally measure: (i) the triangle areas across the shape; (ii) the triangle quality across the shape; (iii) the surface matching to the original shape.

Since all our metrics are averaged across different shapes, we ensure that the quantitative measurements are consistent by centering the meshes at the origin and rescaling all the coordinates in $[-1, 1]$.

In Figure 5.6 we show a comparison between a high-resolution mesh decimated to 1% of its original vertices using the Blender's `decimate` modifier [56] and the `edge-collapse` algorithm from the CGAL library [43], in contrast to the remeshing obtained with our method. This example highlights the main difference between our approach and the other techniques: the other approaches try to approximate as much as possible the original shape, sacrificing the quality of triangles and connectiv-

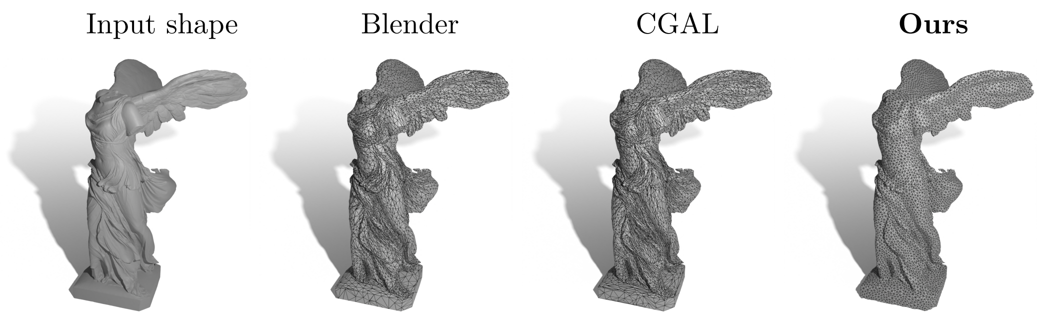


Figure 5.6. An high-resolution mesh (first) decimated to 1% of its original vertices using different methods. The `decimate` algorithm from Blender (second) and `edge-collapse` from CGAL (third) better preserve the high-frequency features of the mesh, but fail in generating a uniform triangulation, and often generate triangles with sharp angles. Our method (fourth) yields a more stable output, where triangles have roughly the same area and without sharp angles.

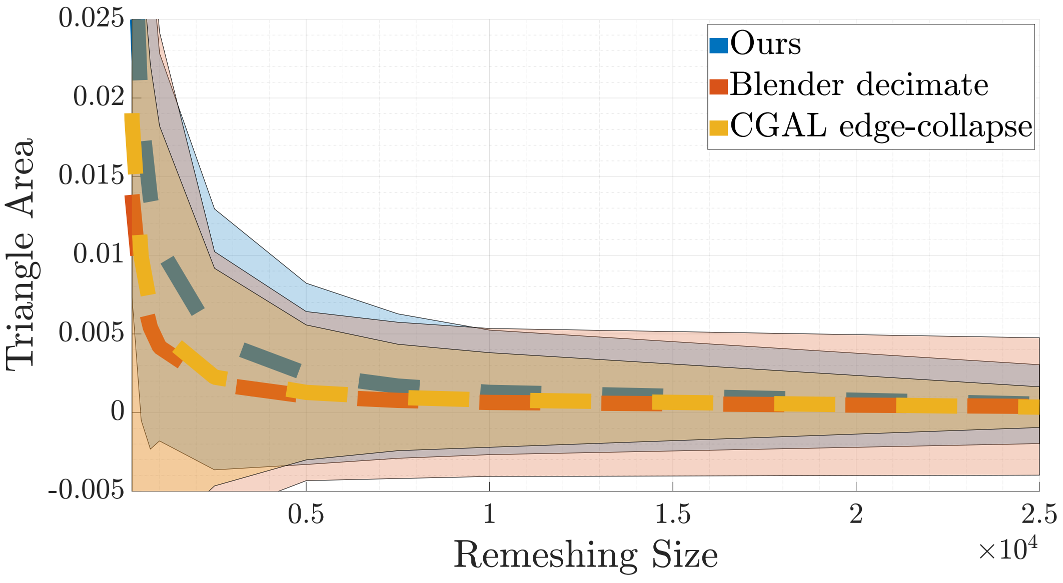


Figure 5.7. Average triangle area and standard deviation with respect to the number of sampled vertices. In comparison to Blender, our algorithm generates more uniform triangles. In contrast, CGAL obtains a comparable uniformity, and tends to create smaller areas.

ity, whereas our algorithm is much more stable in generating a uniform high-quality triangulation.

To evaluate the triangle uniformity, we compute the area of every triangle of each mesh. From these values, we obtain the mean and the standard deviation. Figure 5.7 summarizes the results of this experiment for our algorithm, in comparison to other well-established methods. As expected, the mean area decreases as the number of sample points increase. Since we experimentally verified that the number of triangles in the output of our remeshing algorithm is about 3 times the number of vertices, the number of triangles increases linearly with the remeshing size n, and the mean area decreases with $1/n$. The standard deviation is relatively small,

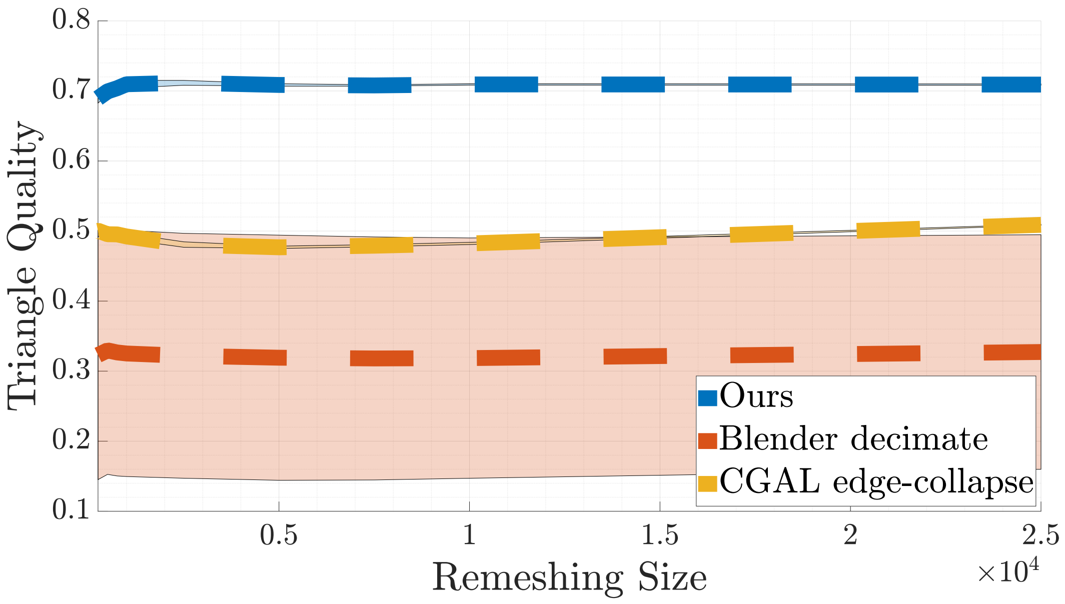


Figure 5.8. Average triangle quality and standard deviation with respect to the number of sampled vertices. Compared to other methods, our method generates higher quality triangles in a more stable manner.

and we notice that it decreases with the size of the mean area. Thus, the triangles are roughly uniform across the shape, and they tend to be more stable with larger samples. We expect this to depend on the behavior of the farthest point sampling algorithm, as it tends to not always distribute the points evenly for small sample sizes.

Compared to the other approaches, our algorithm yields more uniform triangle areas, as can be seen from the distribution of the standard deviation. The `edge-collapse` algorithm from the CGAL library achieves comparable results to our algorithm, with a very low standard deviation, but the average area decreases very rapidly. Similarly, the Blender's `decimate` modifier tends to produce small triangles, but with consistently larger standard deviation over time.

Triangle quality evaluation has been extensively discussed in the literature [119]. Between the possible approaches, we use the metric $Q(t) = 2\sqrt{3} A_t / S_t \ell_t^*$ proposed by Frey *et al.* [87], where A_t is the triangle area, S_t is the semi-perimeter and ℓ_t^* is the longest edge. This metric spans from 0 (for degenerate triangles with null area) to 1 (for equilateral triangles). Similarly to the area case, we measure the triangle quality for each triangle of each shape, and then we compute the average and standard deviation. The results summarized in Figure 5.8 show that our method is very stable in producing high quality triangles, with negligible standard deviation. The triangles produced by the CGAL's `edge-collapse` algorithm are also stable, but the overall quality is significantly lower than ours. By looking at the definition of quality, this means that the algorithm tends to produce "skinnier" triangles, as the ratio between the area and the edge lengths is smaller. Finally, Blender's `decimate` generates the lowest quality triangles, and the high standard deviation proves that the method does not try to generate triangles with specific shape or properties.

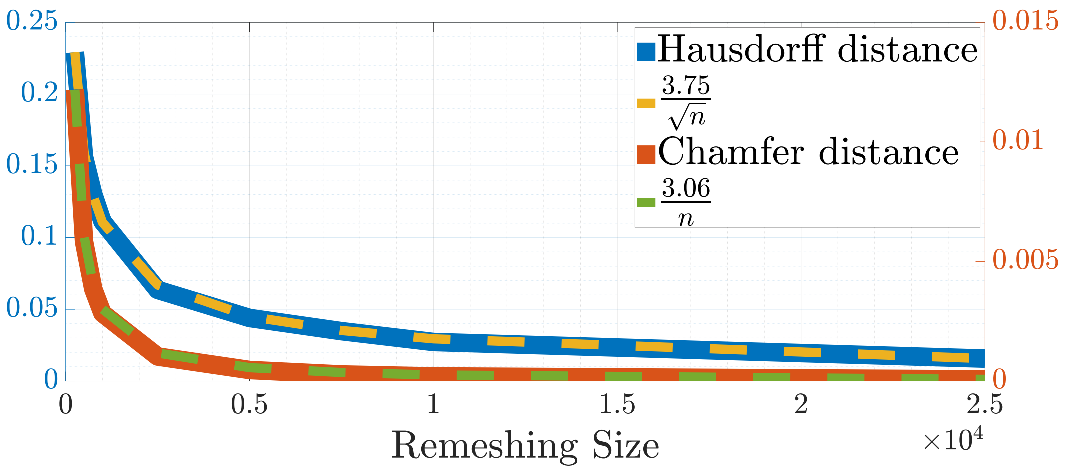


Figure 5.9. Hausdorff and Chamfer distances between the original and remeshed shapes with respect to the remeshing size. In both cases, the reconstruction error decreases with the number of vertices. The Hausdorff distance decreases proportionally to $\sqrt{n}$, whereas the Chamfer distance decreases proportionally to n.

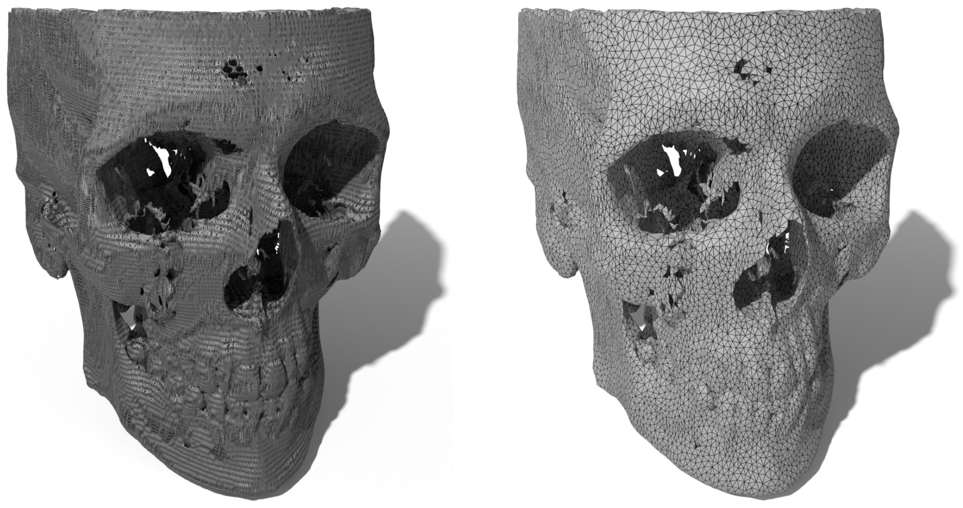


Figure 5.10. An example of drastic decimation obtained with our algorithm. The original mesh has ~1.4M vertices, whereas with our algorithm we are able to obtain a uniform decimated shape with 50k vertices which still preserves the original underlying geometry.

Finally, we evaluate the reconstruction accuracy of our method by evaluating the Hausdorff and the Chamfer distances between the original and the remeshed shape. Given the original mesh $\mathcal{M} = (V, E, T)$ and the remeshed one $\mathcal{M}' = (V', E', T')$, Hausdorff distance $H(\mathcal{M}, \mathcal{M}')$ and the Chamfer distance $C(\mathcal{M}, \mathcal{M}')$ are computed as

$$\begin{aligned} H(\mathcal{M}, \mathcal{M}') &= \max_{x \in V} \min_{y \in V'} \|x - y\|_2 \,, \\ C(\mathcal{M}, \mathcal{M}') &= \frac{1}{|V|} \sum_{x \in V} \min_{y \in V'} \|x - y\|_2^2 + \frac{1}{|V'|} \sum_{y \in V'} \min_{x \in V} \|x - y\|_2^2 \,. \end{aligned} \tag{5.8}$$

We show the Hausdorff and Chamfer distances in Figure 5.9, plotted against

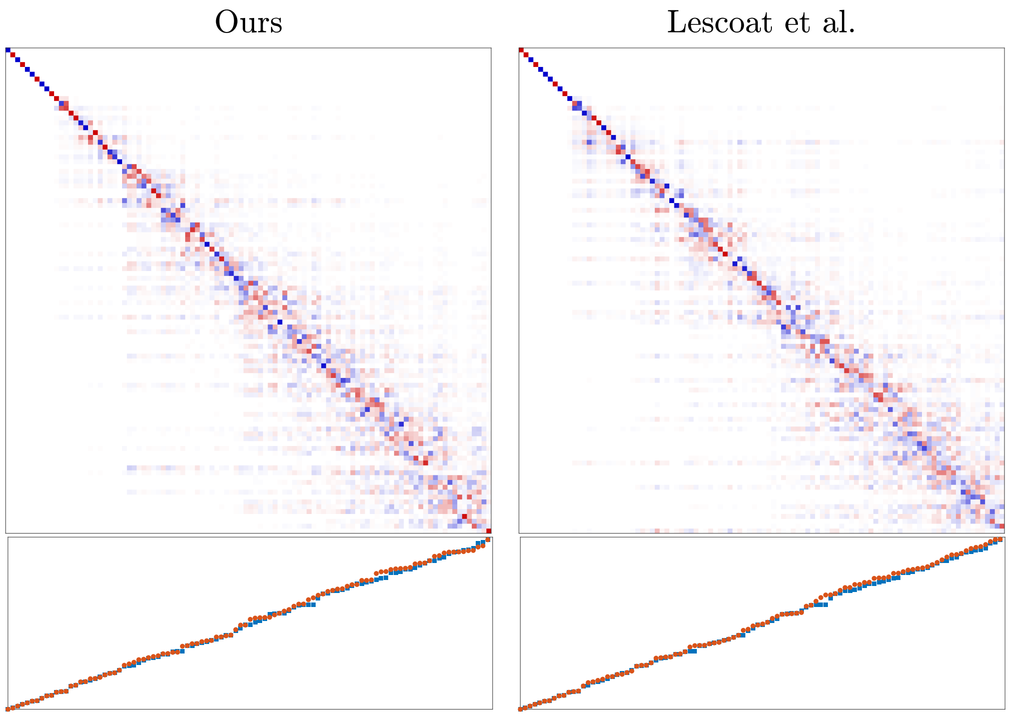


Figure 5.11. The top row shows the ground truth functional map between the original `fertility` mesh and a remeshing obtained with our algorithm using 50k vertices, compared with the result obtained with the algorithm from Lescoat *et al.* [133]. The bottom row shows the comparison of the eigenvalues.

the remeshing size and averaged across the dataset. As we expect, the distances decrease as the number of sampled vertices increases. The quick drop is also not very surprising: since we are sampling evenly from the vertices of the original shape, we expect to have a lot of null differences, and the biggest difference to be bounded by the largest Voronoi partition. As we increase the number of samples n, the area of the partitions decreases as $1/n$, and the radius of the partition decreases as $1/\sqrt{n}$. So, it is reasonable that the Hausdorff distance decreases proportionally to $\sqrt{n}$, and the Chamfer distance proportionally to n.

In Figure 5.10 we show an example of how our algorithm can reduce the amount of vertices in the input shape by orders of magnitude and produce a uniform remeshing, while still preserving the overall underlying geometry.

As a final note, since our algorithm produces a triangulation that is geodesically uniform, according to the underlying original geometry, it is reasonable to guess the remeshed output would encode some of the spectral information from the original mesh. As a test case, we use the mesh `fertility` (~250k vertices) and decimate it to 50k vertices using Algorithm 5.1 and the method from Lescoat *et al.* [133], which is specific for simplifying meshes while preserving the spectral decomposition. We obtain a ground truth correspondence between the original mesh and the output of the two algorithms by mapping nearest neighbouring vertices, and from that we build the corresponding functional map. The results are shown in the top row of Figure 5.11, while the bottom row of the figure shows the comparison between

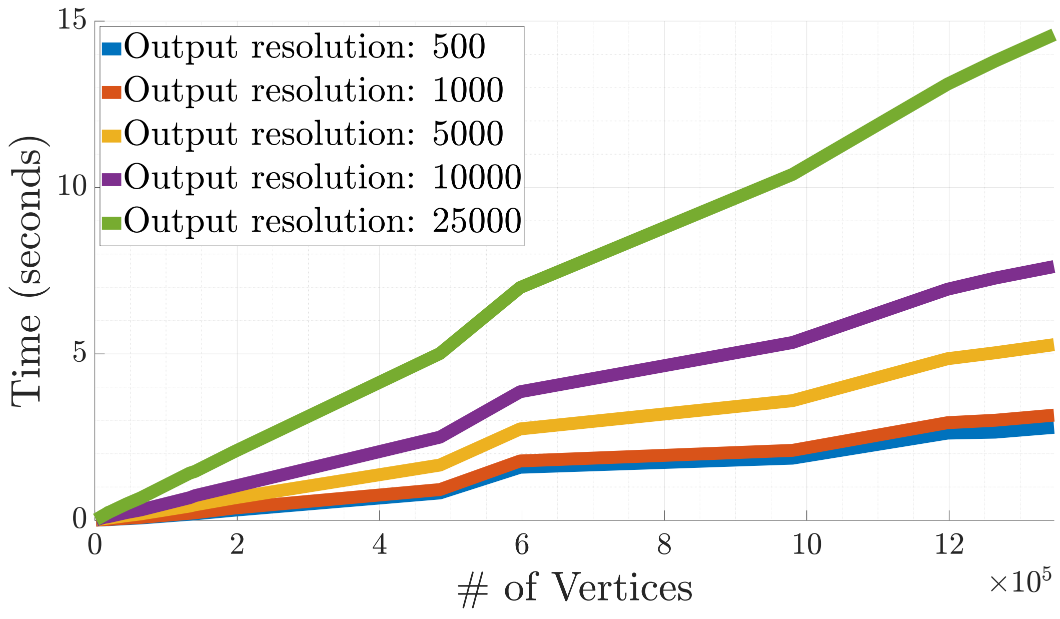


Figure 5.12. Execution time of the remeshing algorithm at varying of the input mesh density. Different curves represent different numbers of output vertices.

the original eigenvalues and the eigenvalues of the remeshed shape. We see that the difference in the quality of the functional map and the eigenvalues matching is negligible, but while our method took about 5 seconds, the technique from Lescoat *et al.*, which requires the spectral decomposition of the original shape, ran for about 1h 30m before producing its output.

5.4.2 Performance Analysis

As anticipated by the study of computational complexity in Section 5.3.2, the performance of our algorithm mainly depends on the number of vertices of the input shape and the number of desired output vertices. To test the real performance of our implementation, we used a variety of different combinations of input and output number of vertices.

The results of our experiments are shown in Figure 5.12, where we plotted the execution time at various output resolutions against the number of vertices of the input meshes. Accordingly to the theoretical analysis, the execution time is always linear in the number of vertices of the input shape. We also notice that the increase in the slope of the curve is not logarithmic in the output resolution, but still sublinear: the execution time for an output resolution of 25k vertices is not 5 times the time required by a 5k vertices remeshing, and 1k vertices does not require double the time of 500. This is in agreement with our analysis, since, as we stated in Section 5.3.2, searching the maximum at every iteration is still a linear operation, but it is way more efficient than other complex sections of the algorithm due to cache locality and vectorization of the operations.

A great advantage of our approach is that it scales linearly with the size of the output mesh, and not with the number of removed vertices. This makes it

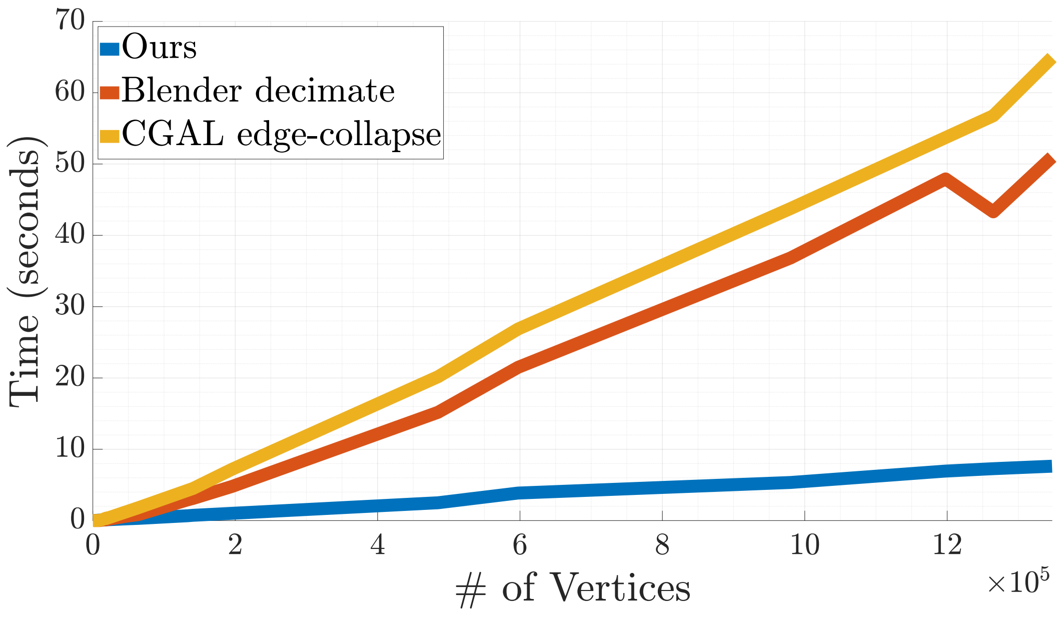


Figure 5.13. Comparison between the execution time of our algorithm against other methods on decimation of variably sized input meshes to obtain a remeshed shape with 10k vertices. The fast FPS Voronoi decomposition allow us to achieve competitive performance.

suitable in applications where a drastic reduction of vertices is needed. To prove the effectiveness of the method, we decimated the shapes in the dataset using the Blender's `decimate` modifier and the CGAL's `edge-collapse` algorithm, to obtain an output shape with 10k vertices. We measured the execution times, and we compared them with those obtained with Algorithm 5.1. From Figure 5.13 we can see that our method outperforms the other algorithms, and the performance gap increases linearly with the size of the input shape. This is because Algorithm 5.1 incrementally builds the decimated shape, whereas the other approaches iteratively remove mesh elements. Of course, our approach would be outperformed when the size of the input and output are close, but for drastic reductions in resolution it performs much better.

5.4.3 Resampling of Large Triangles

As anticipated in Section 5.3.4, the input shape offers no guarantees on the distribution and shape of the triangles. Since Algorithm 5.1 samples the vertices from the original mesh, if some very large triangles existed in the input, those triangles are very likely to appear also in the output.

With our resampling strategy, the algorithm can overcome this issue by picking extra points from the longest edges and from the largest faces. Depending on the amount of large faces and their size, the resampling step can be costly, and could produce a massive amount of extra vertices and edges. Figure 5.14 shows the dependency between the output resolution and the execution time of the resampling step. In particular, we notice how the resampling requires logarithmic time for low output resolutions, but as the resolution increases the number of split edges becomes

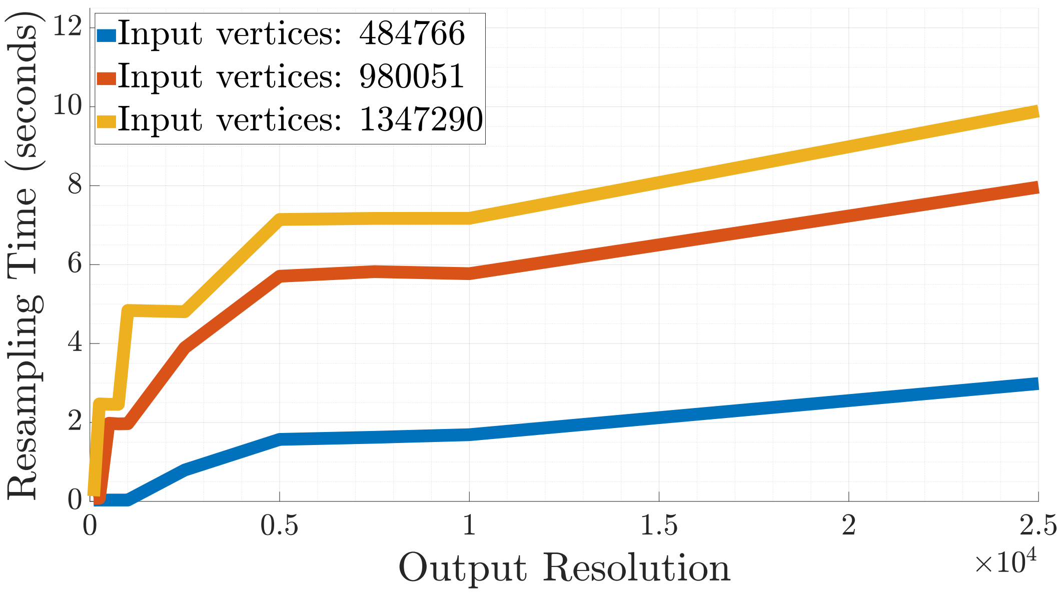


Figure 5.14. Execution time for the resampling algorithm on various input sizes and output resolution. The time for the preprocessing step increases with the output resolution, as the edge length threshold decreases. However, even if the execution time increases with the size of the original shape, it is affected more by the connectivity of the mesh than by the input size.

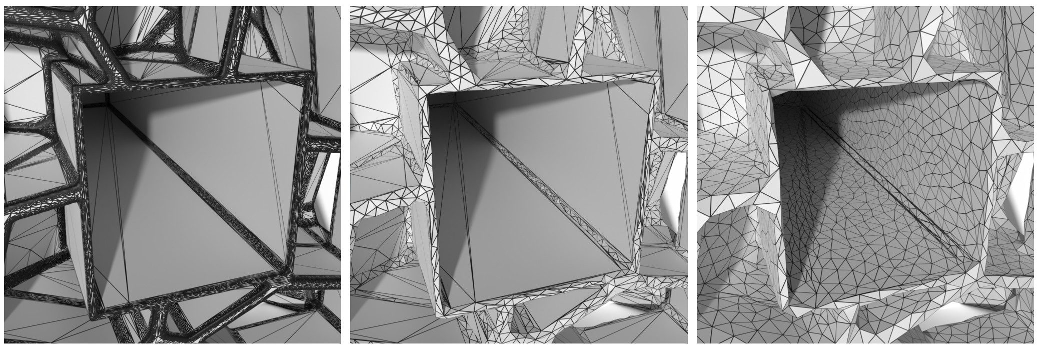


Figure 5.15. An example of simplification with and without resampling. The original mesh (left) is highly non-homogeneous and contains large faces. Without preprocessing (middle) our algorithm fails to generate a uniform distribution of triangles. Performing the resampling step (right) gives better results with more uniform triangle areas.

very high and we approach the worst case. However, results like Figure 5.15 show that it is really game-changing for the output quality, when the input is largely non-homogeneous. In Figure 5.16 we show quantitatively how the algorithm benefits from this preprocessing step. The standard deviation from the average triangle area becomes negligible; almost all the triangles are very close to the average, meaning that the remeshing is much more uniform. Furthermore, the average triangle quality slightly increases, proving that the triangles tend to be more regular across the shape.

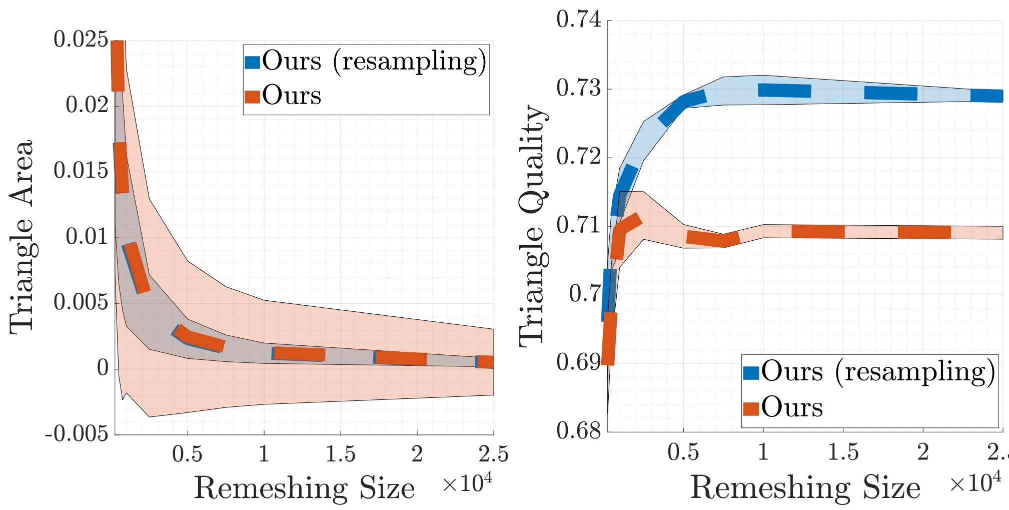


Figure 5.16. Comparison of the mean and standard deviation in the distribution of triangle areas and qualities, with and without using the resampling preporcessing step. Being able to work with a finer sampling of the input mesh results in a more uniform distribution of triangle areas and an higher triangle quality in the decimated mesh.

5.5 Limitations and Conclusions

We presented a new algorithm which exploits non-Euclidean Delaunay triangulation to compute a fast drastic mesh simplification. The statistical uniformity guarantees on triangles shapes and areas which our algorithm provides can be strengthened with a mesh resampling step performed as preprocessing. We compared our algorithm with other well-established decimation method, and we showed that our technique is faster and yields more stable results for massive simplifications. Furthermore, we designed an efficient algorithm for computing geodesic farthest point sampling and geodesic Voronoi partitioning which scales well on highly dense triangular meshes.

As an outlook for future directions, we intend to focus on strengthening the uniformity guarantees on the output mesh, in order to provide tight bounds and more strict constraints on the shapes and dimensions of the triangles. Moreover, we plan to focus on extending the algorithm to handle face sampling, avoiding the preprocessing step on non-homogeneous shapes, and to deal with subtractive simplifications for cases where the required decimation is less significant. Finally, the preliminary results shown in this chapter show that the algorithm has potential for producing remeshed shapes that preserve spectral properties. We intend to explore this direction further, investigate the reasons behind this behavior, and apply the algorithm to pipelines relying on functional maps for other tasks, like shape matching and signal transfer.

Chapter 6

Reconstructing Curves in Non-Euclidean Domains

Reconstructing 2D curves from sample points is a fundamental problem in computer graphics, with consolidated utility in vector graphics applications. Despite the amount of research devoted to this task, there is still room for improvement and many directions are yet to be explored. Vector graphics on surfaces is becoming an important topic nowadays, especially for its utility in design applications. However, the task of designing and editing curves on surfaces has been only barely explored recently. A big step has been done with the definition of Bézier splines in non-Euclidean geometries and discrete manifolds, but the process still mostly relies on human intervention. We provide an algorithm that generates a curve directly on a (possibly bounded) surface from a set of sample points. Being able to solve this problem automatically and without human input will be beneficial in reducing the workload for artists, as well as help in industrial applications where surface elements need to be cut, edited, or connected at boundaries, or measured. Moreover, an algorithmic solution permits addressing this task in cases where the amount of input points is too large to be handled interactively by a user, such as when they are generated programmatically or from 3D scans. The work presented in this chapter has been realized in collaboration with Diana Marin (MSc.), to whom goes the credit for the algorithmic idea, the generation of the dataset, and the invaluable help in the analysis and presentation of the results, Stefan Ohrhallinger (Ph.D.), whose help in clarifying the positioning of this research and its applicability largely improved the quality of the work, prof. Michael Wimmer, and prof. Emanuele Rodolà, both of which helped in improving the overall presentation and defining the structure of the task.

6.1 Introduction

Vector graphics have always been an important field in computer graphics, and it is widely applied in many fields, spanning from design and art to engineering. One of the important reasons for its success is the ability to generate infinite resolution smooth complex visualizations with relative ease while requiring only little input geometry.

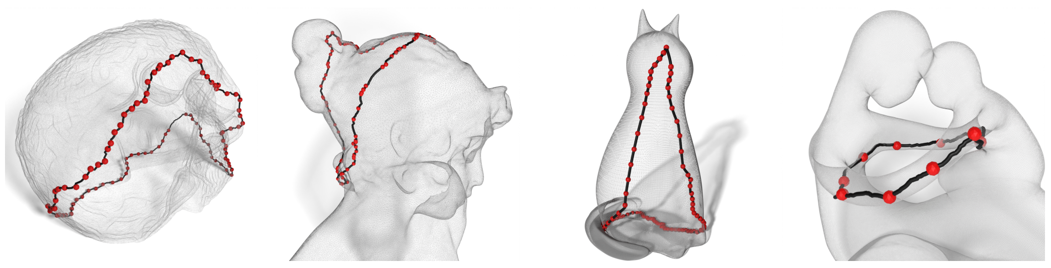


Figure 6.1. Curves reconstructed on various surfaces. Given a set of sample vertices of the mesh, our method generates a geodesic closed curve that traverses all the samples.

Recently, an increasing interest has been devoted to moving vector graphics onto surfaces, trying to address certain issues stemming from texturing methods. Texturing is a well-established approach for defining patterns and decorations on surfaces, but it generally relies on finite-resolution images and parameterization. The latter is not always available or could be difficult or expensive to define. Procedural textures try to overcome these problems, defining patterns via mathematical functions and algorithms. However, they generally rely on multi-dimensional noise functions that are then projected onto the surface [77, 101, 157], which are usually agnostic of the underlying geometric properties and can incur high computation times.

Besides some works generalizing sample-based texture synthesis to triangular meshes [285, 270], it has been only in recent years that some solutions have been proposed which try to leverage properties of non-Euclidean metric spaces and define patterns directly on surfaces, either via recursive structures [188] or simulated behaviors [160]. Another avenue of development for texture synthesis is represented by neural networks that generate a textured mesh in the style of an input image [180].

Despite the existence of various solutions for decorating surfaces, the problem of constructing lines and curves on discrete manifolds has not been addressed satisfactorily yet. Defining curves and shapes directly on surfaces is innovative for design applications [217], and has a relevant impact in the processing of archaeological data [126, 91] by extracting specific decorations from the models. However, little research has been devoted to improving the definition and the reconstruction of curves on discrete surfaces, besides efforts to generalize Bézier curves [162]. Furthermore, the existing solutions are generally centered around human interaction, as they are designed to be tools for artists and end users. The same dependability on user interaction is faced by the field of generating cuts in models [223], where our method's results can also be used to automatize this process.

With this work, we present a novel algorithm for reconstructing closed curves directly over discrete surfaces that may contain boundaries. Our method generalizes existing techniques for 2D curve reconstruction, leveraging the non-Euclidean metrics to solve the problem on manifold domains. We propose an efficient solution, allowing for the reconstruction of curves made from given samples over high-resolution arbitrary meshes (*e.g.*, independently of the genus or the presence of boundaries) with an interactive frame rate. Such an algorithm would allow for reducing the need for inputs from an external user, speeding up the process of designing constrained curves. The applications of such a method are manifold, ranging from design usage for decorating meshes, cutting, editing, connecting, or measuring object parts in industrial settings, to selecting closed subsets for boolean operations on the surface such as intersection or removal. Input as unstructured points on surfaces that require such automatic connection include salient points from texture, or from geometry such as sharp features, or may be generated by simulations such as collisions or fracturing.

6.2 Related Work

Reconstructing curves from samples on a non-Euclidean domain represents an untapped domain existing at the intersection of curve reconstruction, surface design, and texturing using on-surface elements. Our method is also related to heat diffusion approaches through the way we have designed the reconstruction. This section will provide an overview of various techniques that deal with each of these fields individually and explain how they relate to our work.

Planar curve reconstruction (*i.e.*, in the case where samples and their respective reconstructed curve live in $\mathbb{R}^2$) is dealt with by numerous methods. They are usually divided into two main categories: implicit (methods that approximate the inside/outside of the shape and recreate the boundary based on the division between the two) [105, 118] and explicit (interpolatory methods that create some ordering among the sample points). We will focus on the explicit reconstruction methods, as they are the most similar to our work. However, most of these methods are limited to planar curves, even if some of them have been extended to surface reconstruction, and those which are able to reconstruct curves in $\mathbb{R}^3$, do not use a manifold to construct the curve on.

In order to interpolate the input sample points, most of the methods compute a graph on the input and use a subset of the edges as the final reconstruction. One of the most commonly used types of graphs is the Delaunay triangulation, due to its geometrical properties and theoretical guarantees of including the reconstruction subject to sampling density [6]. Starting from the Delaunay triangulation, Amenta et al. [6] filter the edges whose proximity is empty of samples and of Voronoi vertices. This approach has been improved to take into account Voronoi poles - Voronoi vertices corresponding to long, skinny Voronoi cells [7].

Using the same Delaunay starting base, a greedy procedure is used to pick a seed vertex and find the nearest neighbors until the end points are close enough to be connected or all points have been connected [207]. This is similar to our proposed work, however, we are not using the Delaunay triangulation as a base, and their method can be only used for planar curves. Using a similar idea, various criteria can be applied for creating a connection between two points: the new neighbor has to be situated in the half-plane (defined using the normal at the current sample) opposite to the previous connection [72], or in the opposite half-plane defined using the bisector of the previous edge [196]. Another set of criteria used to filter the triangulation is based on leveraging the Voronoi poles to approximate the normals for the half-plane computation, considering the angle and the ratio between the current edge and its Voronoi counterpart [71]. Various other methods build on the Delaunay triangulation as a starting base for curve reconstruction [164, 197], and a comprehensive survey on multiple implicit methods can be found here [198].

Our method lifts the reconstruction of curves from planar surfaces to non-manifold domains.

Vector graphics on planar surfaces have been thoroughly researched and are being used in multiple tools[110, 4]. Recently, editing and importing curves on surfaces have received interest in the graphics field, due in part to the improvement in computing geodesic distances efficiently [246]. Users are able to interact with designs directly on the mesh, by either editing splines on a 2D local projection of the surface [217], which is usually prone to artifacts due to the projection procedure, as explored in [297]. Editing splines directly on the surfaces is also possible, bypassing the projection artefacts, by using geodesic metrics on the surface [188, 162]. However, these editing methods require user input or a predefined ordering of the samples in order to construct the curves on the surface, which is the missing link we are providing here. Hence, our method also provides a building block for further editing of splines on surfaces.

The heat equation describes the intensity with which heat would be distributed across a surface from an initial starting point. On surfaces, the isolines corresponding to various intensities relate to the geodesic lines of the same seed point through Varadhan's equation [274]. This relation has been explored in order to obtain improved results for geodesic computation [62] and the procedure is similar to ours in the sense of greedy expansion across the surface. However, we do not make use of the heat kernel directly in our computation, it just inspires our method.

6.3 Method

We are given a 2-dimensional manifold $\mathcal{M}$ and a set of points P sampled from a 1-dimensional closed curve $\mathcal{C} \subset \mathcal{M}$ (*i.e.*, a curve lying on the surface that is $\mathcal{M}$). Using the points in P, we want to infer the curve $\mathcal{C}$, or at least some curve $\mathcal{C}^* \approx \mathcal{C}$ that approximates the original curve.

The manifold $\mathcal{M}$ is discretized by means of a triangular mesh $\hat{\mathcal{M}} = (V, E, T)$, where: (i) V is a set of vertices sampled from $\mathcal{M}$; (ii) $E \subset V^2$ is a set of edges; (iii) $T \subset V^3$ is a set of triangles. To simplify the formulation of the problem, we assume that the given points P are vertices of $\hat{\mathcal{M}}$ and that we can safely approximate the target curve $\mathcal{C}$ with a 3D polygon $P \subset E$ entirely made of edges of $\hat{\mathcal{M}}$.

6.3.1 TSP on Surfaces

Our approach finds its foundations in classical theory for solving the traveling salesman problem (TSP). Clearly, the two problems are different, but there are common parts. For instance, we do not allow for passing through the same sample point more than once, and even if we are not explicitly searching for the shortest path, we highly penalize long walks going back and forth between distant points in favor of short and smooth paths connecting near vertices. Also, the TSP is not restricted to Euclidean domains which is a prerequisite for our use case. We use the simple yet effective nearest-neighbor algorithm for the TSP as a base (from here on, referred to as TSP-NN) [113]. This choice is dictated by the necessity of heavily modifying the solution to account for the generalization to non-Euclidean domains and the application of other restrictions that the pure TSP solver does not impose, like

Algorithm 6.1 Curve reconstruction on surfaces.

```
1: **procedure** CurveRecon($\hat{\mathcal{M}} = (V, E, T),\ P = \{p_1, \cdots, p_k\}$)
2:     $p_T \leftarrow p_1$
3:     $P \leftarrow P \setminus \{p_1\}$
4:     $\hat{\mathcal{C}} \leftarrow [p_1]$
5:     **while** $P \neq \emptyset$ **do**
6:         $p_N \leftarrow$ NNAvoid($\hat{\mathcal{M}}, p_T, P, \hat{\mathcal{C}}$)
7:         $\mathcal{P} \leftarrow$ ShortestPathAvoid($\hat{\mathcal{M}}, p_T, p_N, \hat{\mathcal{C}}$)
8:         $\hat{\mathcal{C}} \leftarrow [\hat{\mathcal{C}}, \mathcal{P}]$
9:         $P \leftarrow P \setminus \{p_N\}$
10:        $p_T \leftarrow p_N$
11:    **end while**
12:    $\mathcal{P} \leftarrow$ ShortestPathAvoid($p_T, p_1, \hat{\mathcal{C}}$)
13:    $\hat{\mathcal{C}} \leftarrow [\hat{\mathcal{C}}, \mathcal{P}]$
14:    **return** $\hat{\mathcal{C}}$
15: **end procedure**

16: **procedure** NNAvoid($\hat{\mathcal{M}} = (V, E, T),\ p \in V,\ P \subset V,\ \hat{\mathcal{C}} \subset V$)
17:     $Q \leftarrow \{p\}$
18:     $\boldsymbol{d} \leftarrow \infty \in \mathbb{R}^{|V|}$
19:     $\boldsymbol{d}[p] \leftarrow 0$
20:     **while** $Q \neq \emptyset$ **do**
21:         $q^* \leftarrow \arg\min_{q \in Q}(\boldsymbol{d}[q])$
22:         **if** $q^* \in P$ **then**
23:             **return** $q^*$
24:         **end if**
25:         $Q \leftarrow Q \setminus \{q^*\}$
26:         **for** $r \in \mathrm{adj}(q^*)$ **do**
27:             **if** $r \in \hat{\mathcal{C}}$ **then**
28:                 **continue**
29:             **end if**
30:             **if** $\boldsymbol{d}[r] < \boldsymbol{d}[q^*] + \mathrm{dist}(r, q^*)$ **then**
31:                 **continue**
32:             **end if**
33:             $\boldsymbol{d}[r] \leftarrow \boldsymbol{d}[q^*] + \mathrm{dist}(r, q^*)$
34:             $Q \leftarrow Q \cup \{r\}$
35:         **end for**
36:     **end while**
37: **end procedure**
```

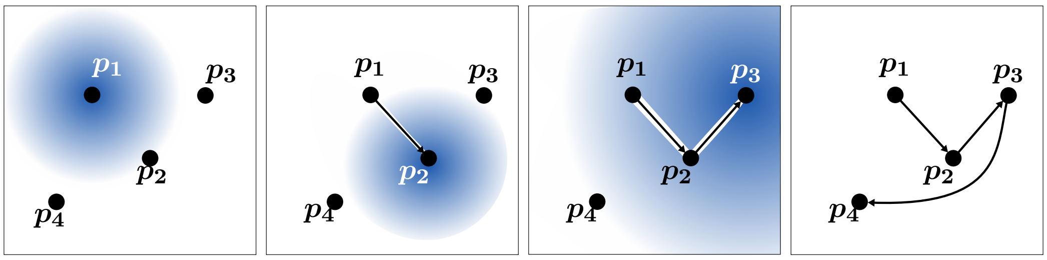


Figure 6.2. A simple example of our modified nearest neighbor search in 2D. The isotropic expansion from each sample point allows for finding the closest vertex without the need of checking the entire mesh. The path is required not to have self-intersections, resulting in the avoidance of the center vertex p_2, which is already connected.

non-intersecting edges. Hence, a simple, versatile, and easily tunable algorithm is needed.

Working in a non-Euclidean domain is usually an additional challenge, but we can take advantage of the underlying graph structure to simplify the computation and reduce the time complexity. TSP-NN usually needs to compute the Euclidean nearest neighbor for each sample point, resulting in an $\mathcal{O}\left(n^2\right)$ complexity. Since searching for the nearest neighbor forces us to traverse the entire graph, we rather start from the current point and expand isotropically on the surface along edges until we reach the first unvisited sample. This optimization prevents us from visiting the entire triangular mesh for every sample point and searching for the nearest neighbor since we stop the visit as soon as we find it.

As pointed out earlier in the section, in the TSP, the path is not required to prevent self-intersections,whereas our solution produces a non-self-intersecting curve. To address this issue, we slightly modify the shortest path search on the mesh graph to ignore all the vertices already included in the walk and all their incident edges. This includes the isotropic expansion in search for the nearest neighbor, meaning that this search must take into account the avoidance of the already constructed curve. If the surface is bounded, this does not affect our algorithm since it only considers the mesh graph and does not have to treat edges on the boundary differently.

Since the resulting curve must be closed, the last visited sample is connected back to the starting point once all the points have been covered. Since we do not allow for self-intersections, we close the curve only at the end of the process, meaning that before the completion of the curve, the surface is never partitioned, allowing for the path between the last and the first samples to avoid intersections with the previously computed curve.

The entire process is summarized in Algorithm 6.1, where ShortestPathAvoid is a routine for searching for the shortest path avoiding the input set of vertices, and NNAvoid (Lines 16-37) is an algorithm for isotropically expanding from a given vertex along edges until the nearest sample is found, while avoiding the input set of vertices. Figure 6.2 provides a basic example of how the algorithm behaves in the simplified case of a planar domain.

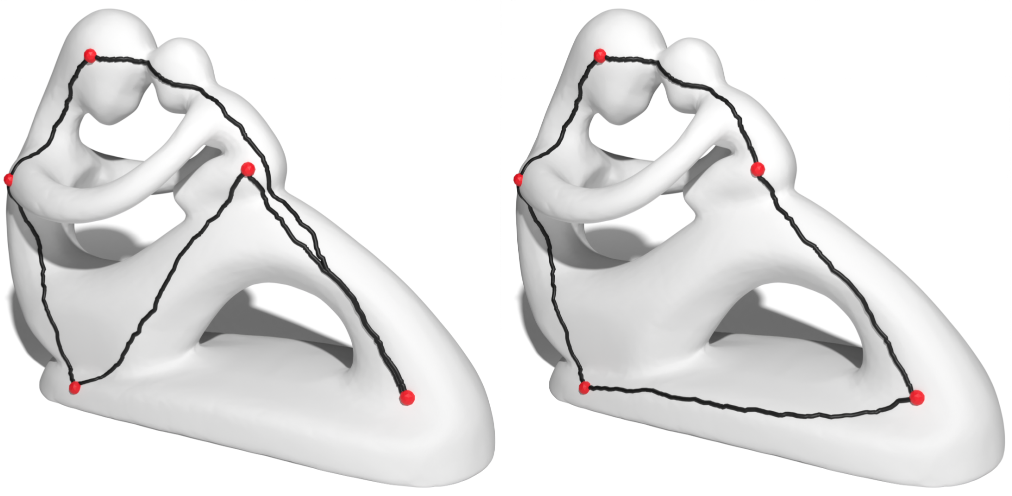


Figure 6.3. A surface curve computed using just the basic nearest neighbor algorithm (left), compared with the approach augmented by the maximum angle heuristic (right). With the same input, the heuristic is able to find a better-shaped path with negligible added computational cost.

6.3.2 Refinements

The example in Figure 6.2 exposes an issue with this approach. Despite the algorithm behaving nicely in most cases, it is sensitive to the choice of the starting point, producing ill-shaped contours when the placement of the points is adversarial (*e.g.*, very sparse samplings of thin shapes).

A possible approach could be to select multiple starting points and choose the best curve by some metric. However, this approach would be highly inefficient in terms of computation expense, and it also does not provide any structural guarantee. For instance, in the planar example from Figure 6.2, we would like the curve to be $\boldsymbol{p}_1\boldsymbol{p}_3\boldsymbol{p}_2\boldsymbol{p}_4$, and only starting from $\boldsymbol{p}_4$ allows for finding that shape.

Instead, we again take inspiration from the existing literature on the TSP. When dealing with the TSP in a 2-dimensional space, it is possible to find a close approximation by taking the convex hull of all the vertices and refining it iteratively (we will refer to this algorithm as TSP-CH) [195]. For choosing the next vertex to visit, TSP-CH sculpts edges trying to maximize the angles of the path. Similar approaches for angle heuristics have been explored in the field of curve reconstruction as well [72, 196].

Using this approach directly would be infeasible again since, despite the notion of convexity being defined on manifolds [271], the computation of a geodesic convex hull on arbitrary surfaces still represents a difficult open problem. We, therefore, leverage the idea of picking the largest possible angles and apply it as a heuristic to refine the choice of the nearest neighbor. Instead of isotropically expanding along edges until we reach the closest sample, we continue the expansion to cover the ℓ closest points. Between them, we pick the sample p_N that maximizes θ/δ^2, where θ is the surface angle formed by $P_N p_T p_L$ (being p_T the current sample and p_L the previous one), and δ is the geodesic distance from the current sample. The distance at the denominator acts as a corrector when dealing with dense sampling. When multiple samples are very close, we would ideally connect them in a straight line, but just maximizing the angle would produce a back-and-forth connection. Penalizing points that, despite producing large angles, are too distant, solves this issue. Actually computing the surface angle would require exact geodesic paths on the surface and an angle computation close to the vertex. However, in our experiments, approximating the surface angle with the angle in $\mathbb{R}^3$ produced good results, so we use this approach for efficiency. Figure 6.3 shows how by adding this simple heuristic we can find a much better-shaped path compared to the baseline nearest neighbor solution.

The heuristic, however, needs to be defined at the starting vertex since there are no previous samples. To address this issue, we still search for the closest ℓ samples, and between them, we pick the pair that produces the maximum angle. We then choose the closest point in this pair and continue with the algorithm. This ensures that our algorithm is deterministic.

6.3.3 Complexity Analysis

Since our algorithm uses triangular meshes as input, its complexity is not only related to the number of samples of the curve but also to the density of the mesh. We refer to the number of vertices of the mesh as $n = |V|$ and to the number of sample points as $k = |P|$. We recall that the number of edges and triangles of a triangular mesh is linear in n, so the spatial complexity of $\hat{\mathcal{M}}$ is $\mathcal{O}(n)$. However, for the sake of simplicity, we will refer to it simply as n. We use the Dijkstra algorithm for the isotropic expansion from a vertex, which has a well-known complexity of $\mathcal{O}(n \log n)$. We decide to use this algorithm because of its high efficiency and a high degree of approximation for most practical applications, especially when it comes to highly dense meshes. For studying the overall complexity of our approach, we distinguish between the case where the sample points are uniformly spaced and the case where they are not.

Theorem 6.1. *Given a triangular mesh $\hat{\mathcal{M}} = (V, E, T)$ with n vertices and a set $P \subset V$ of k samples, Algorithm 6.1 computes a discrete curve $\hat{\mathcal{C}}$ on the surface of $\hat{\mathcal{M}}$ passing through the points in P in $\mathcal{O}(n \log n \log k)$ time.*

Proof. If the sample points are uniformly spaced, our isotropic expansion to the closest neighbor covers approximately $1/k$ of the mesh for each sample, meaning that each visit takes $\mathcal{O}(n/k \log n/k)$. Multiplying it by k visits, the overall complexity becomes $\mathcal{O}(n \log n/k)$.

If the sample points are not uniformly spaced, we again distinguish between different cases. If all the samples are clustered together, then the isotropic expansions are all very short and cover a small part of the mesh, meaning the dominant term is $\mathcal{O}(k) \ll \mathcal{O}(n)$. If they are not, then since they cannot be uniformly spaced, we can group them into h clusters $Q_1, \cdots, Q_h$. Covering each cluster Q_i would require $\mathcal{O}(n_i \log n_i)$, where n_i is the number of vertices of the mesh surrounding the points in the cluster Q_i. Summing all these contributions results in a $\mathcal{O}(n \log n)$ time complexity. On top of that, we need to consider the complexity of expanding from each cluster Q_i to its closest cluster Q_j. This grouping into clusters reduces the problem

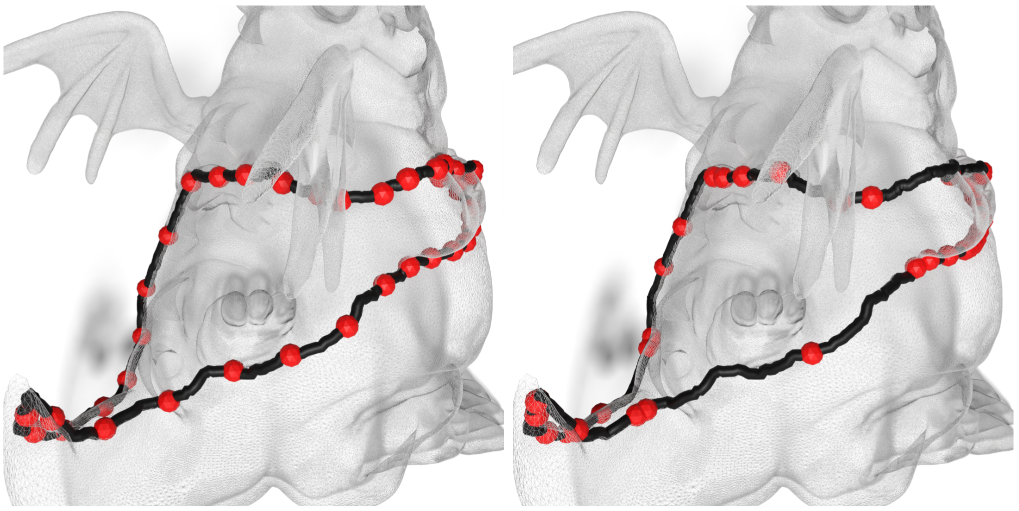


Figure 6.4. From the original curve, which is a dense vertex path, with any two consecutive vertices connected by an edge, we extract a subset of fixed length. Under the assumption of a uniform triangulation, we expect the number of vertices on a path to approximate the distance between them. Hence, for uniform sampling (left), we aim to have the same number of steps between any two consecutive samples. For random sampling (right), we use a uniform distribution on the complete curve to extract the required number of unique vertices.

to the original formulation, where we have h points instead of k. In the worst-case scenario, we iteratively reduce the problem by a factor of r, which results in $\log_r k$ recursions. Since each recursive step is at most $\mathcal{O}\left(n \log n\right)$, the total complexity results to be $\mathcal{O}\left(n \log n \log k\right)$. □

6.4 Results

In order to evaluate our method, we programmatically generate a dataset with 7k curves over 10 shapes extracted from Thingi10K [303] with diverse properties: varying numbers of vertices and triangle sizes, different genus values, and open boundaries. For a consistent evaluation, we ensure all the shapes have unitary volume (for shapes with open boundaries, we close them to compute the volume). We compare the resulting curves obtained using our algorithm with the ground truth curves, and measure the performance to validate the complexity analysis.

The algorithm is implemented in C++, and all the experiments are carried out on a machine equipped with an Intel i7-10700K CPU (16 cores and 3.80 GHz clock frequency) and 32 GB of main memory at 3600 MHz frequency.

6.4.1 Dataset Generation

Starting from a random seed vertex, we mimic a random walk on the surface, along the edges, with some constraints and store all the visited vertices as the ground truth curve. We move away from the initial vertex towards a local geodesic distance maximum. Once this has been reached, we move away from the local maximum, while trying to keep a constant distance to the seed vertex for a randomly chosen number of steps. Once this sideways crawl has been completed, we try to reach the original vertex. We use this procedure to generate 50 curves for each of the 10 meshes in our collection. Furthermore, since the generated curves are densely sampled, as they include all the vertices traversed in our walk, we extract a subset of predefined size as our input for the reconstruction algorithm. We employ both uniformly-spaced sampling and random sampling on that walk using a uniform distribution as illustrated in Figure 6.4. By uniformly-spaced sampling we mean a uniform distance between consecutive samples. We define this distance as the path length (number of vertices on the path), which, assuming a uniformly-sized triangulation, should approximate the distance between samples. We use 7 different values for our input sample size (10, 25, 40, 55, 70, 85, 100) to test the method's reliability in the presence of sparse sampling.

6.4.2 Evaluation

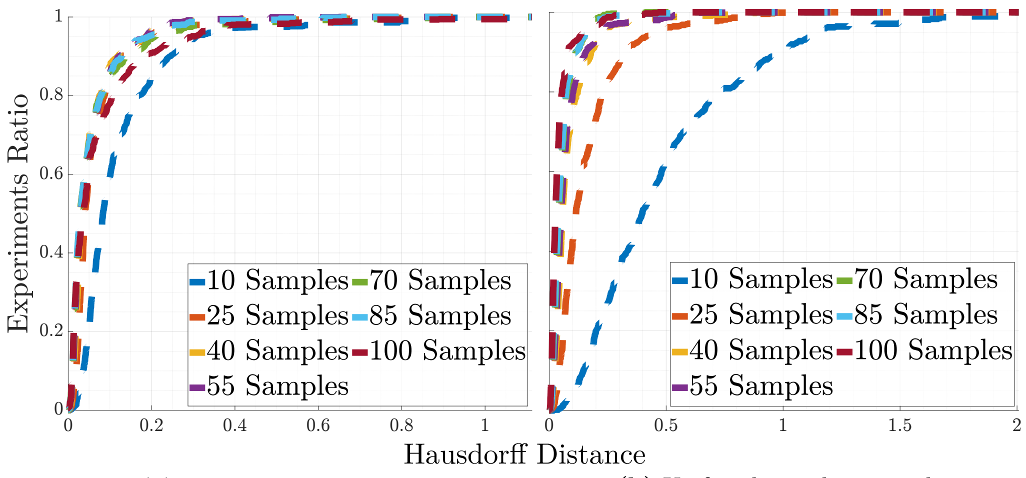
In order to evaluate our method, we run the algorithm on the entire dataset. For extensive testing, we run all the experiments with three different values for the size of the nearest neighbors set to use with the heuristics, bringing the total number of experiments to 21k. The sizes of the nearest neighbors set that we choose for the experiments are: 1 (meaning we only select the closest neighbor and we do not use the heuristic), 3, and 5.

The algorithm manages to output a closed, non-intersecting curve for 83.2% of the experiments, and for the remaining 16.8% it fails to find a closed non-intersecting path that passes through all the samples. In Section 6.4.4 we discuss the conditions of failure in detail.

For the successful experiments, we produce a twofold evaluation. We first analyze the accuracy of the reconstructed curve, comparing how close the reconstruction is to the original curve under different error measures and properties of the input data. Considering the increasing variety of methods for defining curves on surfaces [62, 162], we notice that in some applications it could be more relevant to compute the ordered sequence of sample points, and then leaving the task of defining the actual curve to other methods, especially if they do not depend on the original triangulation. With this setting in mind, we also analyze the correctness of the ordered sequence under varying conditions.

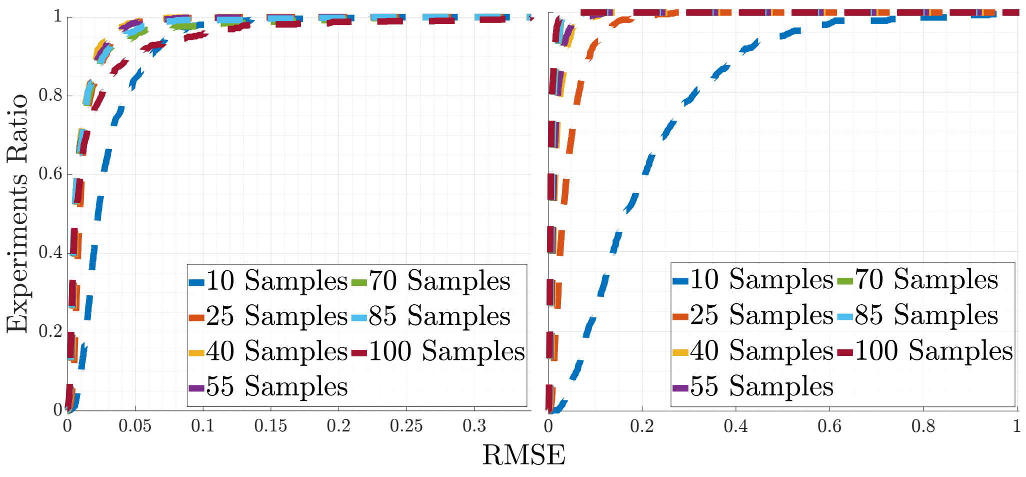
Distance Metrics For evaluating the quality of reconstruction, given the ground truth curve $\hat{\mathcal{C}}^*$ and the reconstructed curve $\hat{\mathcal{C}}$, we compute the Hausdorff distance $H(\hat{\mathcal{C}}^*, \hat{\mathcal{C}})$ and the root mean squared error ($RMSE$) $R(\hat{\mathcal{C}}^*, \hat{\mathcal{C}})$, which are well-established metrics in curve reconstruction literature [198]. The metrics are defined in the Euclidean space as

$$\begin{aligned} H(\hat{\mathcal{C}}^*, \hat{\mathcal{C}}) &= \max\left(\max_{\boldsymbol{x}\in\hat{\mathcal{C}}^*}\min_{\boldsymbol{y}\in\hat{\mathcal{C}}} \|\boldsymbol{x}-\boldsymbol{y}\|_2,\ \max_{\boldsymbol{y}\in\hat{\mathcal{C}}}\min_{\boldsymbol{x}\in\hat{\mathcal{C}}^*} \|\boldsymbol{x}-\boldsymbol{y}\|_2\right), \\ R(\hat{\mathcal{C}}^*, \hat{\mathcal{C}}) &= \sqrt{\frac{1}{|\hat{\mathcal{C}}^*|}\sum_{\boldsymbol{x}\in\hat{\mathcal{C}}^*}\min_{\boldsymbol{y}\in\hat{\mathcal{C}}} \|\boldsymbol{x}-\boldsymbol{y}\|_2}\,. \end{aligned} \tag{6.1}$$



(a) Evenly spaced sampling. **(b)** Uniformly random sampling.

Figure 6.5. Hausdorff distance AUC (Area Under the Curve) between the ground truth input and the reconstructed curve at a varying number of samples of the input curve.



(a) Evenly spaced sampling. **(b)** Uniformly random sampling.

Figure 6.6. RMSE AUC between the ground truth and the reconstructed curve at varying of the number of samples.

We partition the dataset along the number of input samples and the type of sampling (*i.e.*, uniformly spaced sampling or randomly uniform sampling). For each partition, and for each metric, we compute the corresponding AUC curve, summarizing the results in Figures 6.5 and 6.6. Overall, for about 90% of the reconstructed curves, we achieve $H(\hat{\mathcal{C}}^*, \hat{\mathcal{C}}) \leq 0.25$ and $R(\hat{\mathcal{C}}^*, \hat{\mathcal{C}}) \leq 0.07$, and examples like the ones presented in Figures 6.1 and 6.7 show how our method can achieve high fidelity to the original curve. However, by investigating the results on the different dataset partitions we can gather more information on how the algorithm behaves under various conditions. The most evident result is highlighted by the comparison between the values of the metrics on uniformly spaced input samples (Figure 6.5a and 6.6a) against uniformly random input samples (Figure 6.5b and 6.6b). We can see how, in both cases, the algorithm behaves much better when the input samples are uniformly spaced. Random sampling can generate an increasingly sparse input, and when the sampling

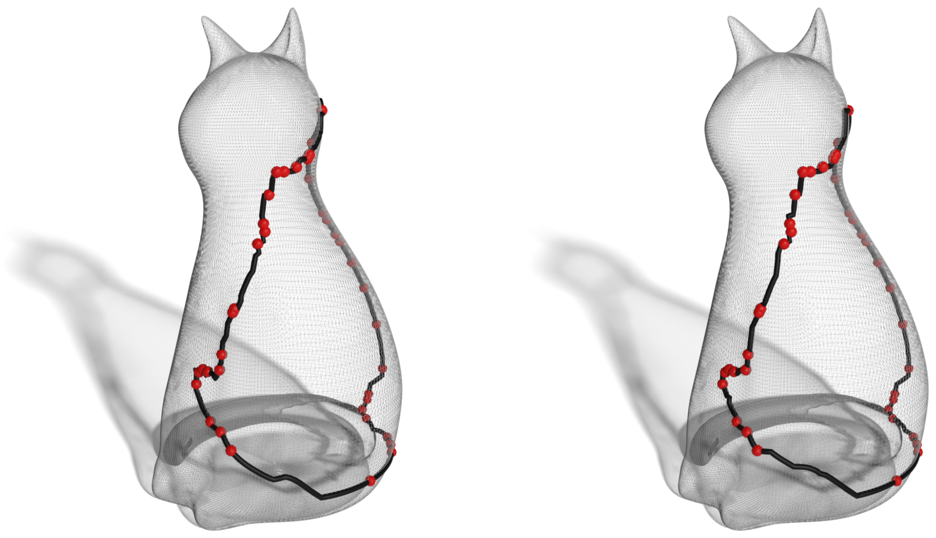


Figure 6.7. The original curve is randomly sampled (left). Our method is fed with the samples and reconstructs a curve that faithfully traces the original path (right).

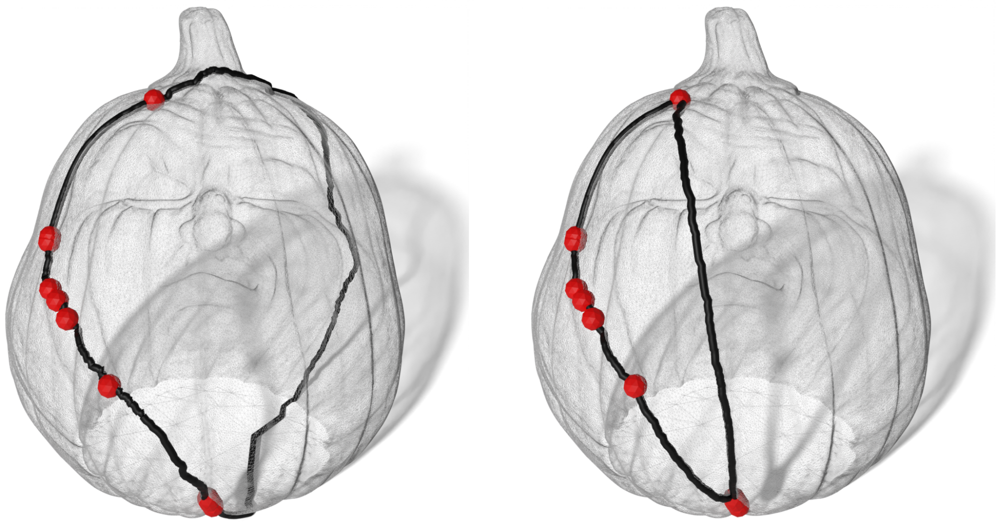


Figure 6.8. The original curve is sampled across its length (left). Our method reconstructs a curve that correctly orders the sequence of samples, but fails to match the original curve (right).

is very sparse this also increases the algorithm's chances of wrongly choosing a very close sample, which is not the next in the ground truth sequence of the original curve. This also explains the performance increase when varying the number of samples. This is somehow expected, as increasing the number of input samples also increases the information on the original curve and allows for less freedom on which sample to choose next. However, the performance gap shown in Figure 6.5a and 6.6a between 10 and 25 samples proves how with a random sampling the reconstruction task is more difficult to address.

Samples **NN**	10	25	40	55	70	85	100
1	33.54%	37.63%	43.15%	40.59%	45.88%	40.53%	41.90%
3	41.86%	45.16%	45.54%	42.79%	44.50%	41.78%	39.94%
5	42.83%	46.09%	46.62%	42.69%	45.34%	43.22%	39.71%

Table 6.1. Percentage of curves for which the algorithm produces a correct ordering of the input samples, when the sampling is evenly spaced. The overall trend is increasing with the increase in the number of samples and the number of nearest neighbor candidates for the heuristic.

Samples **NN**	10	25	40	55	70	85	100
1	60.69%	14.13%	34.34%	28.90%	34.20%	28.84%	22.57%
3	63.43%	24.88%	39.81%	36.60%	37.88%	41.35%	29.22%
5	63.11%	32.12%	41.72%	39.33%	39.61%	41.62%	30.82%

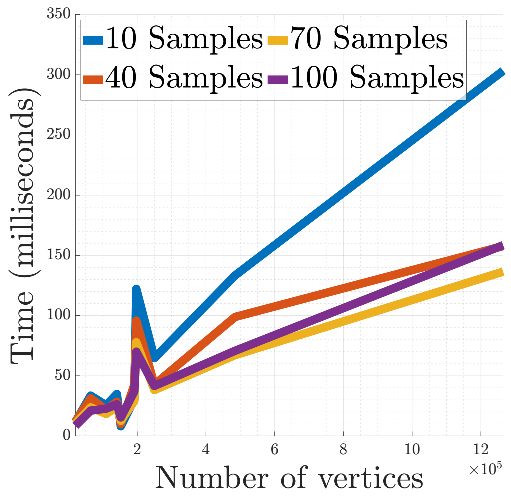
Table 6.2. Percentage of curves for which the algorithm produces a correct ordering of the input samples when the sampling is randomly uniform. The overall trend is increasing with the increase in the number of nearest neighbor candidates for the heuristic but decreases as the number of samples increases.

Ordering Both the Hausdorff distance and the RMSE take into account the shape of the curve, and since we deal with vertex paths, the quality of our reconstruction is strictly bound to the quality of the mesh. Since there are other methods that define geodesic paths between two consecutive samples, we believe that is of interest to study our algorithm's ability to infer the right ordering of the samples, without considering the actual reconstruction path between samples. Figure 6.8 depicts an example where the right ordering of the samples does not guarantee fidelity to the original curve between samples, and hence the reason why this analysis is carried out separately.

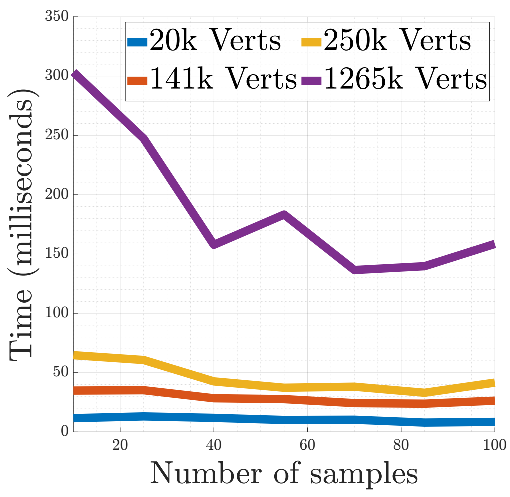
We examined whether our output connects the sample points in the same order as the ground truth, studying how the ability to compute the correct ordering depends on the number of samples and the number of candidates used by the heuristic for choosing the next vertex.

Once again, we partition the results separating the uniformly spaced input samplings and the uniformly random samplings. Tables 6.1 and 6.2 summarize the results of our experiments, showing the percentage of correctly ordered paths at varying of the number of samples and the number of nearest neighbor used in the heuristic.

The number of nearest neighbors included in the heuristic resulted to have a negligible effect on the overall evaluation of the curve using the Hausdorff distance and the RMSE. However, the effects of the heuristic start to be evident when we restrict our evaluation to the ordering of the input samples. Table 6.2 shows this result clearly, especially when dealing with more than 10 samples. Increasing the number of candidates should increase the percentage of correctly ordered curves as the correct next sample might not necessarily be the closest one. Our heuristic has



(a) Average execution time at varying the number of vertices of the mesh.



(b) Average execution time at varying the number of input samples.

Figure 6.9. Average execution time of the algorithm at varying the number of vertices in the input mesh and the number of input curve samples. Despite the outlier spike explained by the mesh quality, in some instances, the overall trend depending on the number of vertices is in agreement with the theoretical analysis. Contrarily to the complexity study, denser samplings result in a more efficient computation.

more chances of picking the correct one if the pool of candidates is larger. This trend can be noticed in Table 6.1 as well, even if the impact is less appreciable and there are few exceptions.

Intuitively, increasing the number of input samples should heavily affects the ability to infer the correct ordering, as the combinatorial complexity of the problem increases exponentially with the number of input samples, and a greedy algorithm like ours is more likely to take wrong decisions. Table 6.2 shows this result, presenting a drastic drop from curves with 10 samples to curves with 100 samples. However, the results presented in Table 6.1 seem to contradict our intuition. Since the samples are uniformly spaced, for a single candidate, we increase the chance of first finding the correct one with our geodesic expansion by increasing the number of samples and implicitly moving the samples closer together. If we increase the number of candidates, the data follows the expected trend for small numbers of samples since samples are still far enough to not always have the correct candidate as the closest encountered sample. For a high enough number of samples, however, the decrease in distance between samples seems to confuse our heuristic in picking the correct candidate.

6.4.3 Performance

To analyze the time performance of our algorithm, we time it over the entire dataset. From the complexity analysis discussed in Section 6.3.3 it emerges that the algorithm depends on the number of vertices of the mesh and the number of input samples, so we relate the runtime to these values. Overall, more than 95% of the runs took less than 0.25 seconds, and the longest experiment lasted for 2.2 seconds on a mesh with 1.25M vertices and 10 input samples, selecting among 5 nearest neighbors for the heuristic.

Figure 6.9a shows how the execution time is related to the number of vertices of the mesh, fixing the number of input samples. Despite the sparsity of the samples, the overall trend is about linear, accordingly to our theoretical analysis. The spike at 200k is caused by the `pumpkin` model (see Figure 6.8). This model exhibits a very heterogeneous distribution of vertices, alternating sparse patches to dense clusters. This variety increases the overall complexity, as traversing some areas of the mesh results in a high number of edge traversals.

From the plot we can also notice that the algorithm seems to be slower when there are less input samples. This fact is confirmed by the results presented in Figure 6.9b, where we show how increasing the number of samples of the input curve, the overall execution time decreases. This happens because increasing the number of samples decreases the spacing between samples, and hence, our geodesic expansion decreases in size as well. The candidates for the closest sample are found faster and with fewer queries, since samples are closer to each other on the mesh. While not invalidating the time complexity analysis, these results show that when dealing with real curves, the algorithm achieves good performance independently of the number of input samples, and hence that it is suitable for both sparse and dense reconstructions.

6.4.4 Limitations

Since our computation is based on the given triangulation and we do not change the initial mesh in any way, the success and its degree depend on the mesh quality. This is the reason why most of the failure cases are on the `egg`, `frog`, and `octocat` meshes, which suffer from poor and non-uniform triangulations, exhibiting low-quality areas with few vertices and long, skinny triangles. However, our method still manages to reconstruct most of the curves even for these difficult meshes, as can be seen in Figure 6.10.

Since the curve reconstruction problem has been only addressed for the planar case until now, there is no dataset to use for reconstructing curves on surfaces. In order to obtain quantitative results, we need multiple and variate curves on multiple models, which can only be obtained programmatically. However, the poor mesh quality influences the ground truth generation as well, since the original curve is also defined as a vertex path on the mesh. The generator has to obtain a high number of samples on a low-poly triangulation which implies a chaotic walk on the surface, as can be visible in Figure 6.12.

Furthermore, increasing the number of samples affects the reconstruction as well, since more samples imply more constraints for our method which, due to the greedy approach, result in cutting off sections of the mesh too early. This trend can be observed in the aggregated results over all the sample values, visible in Figure 6.11.

Inheriting limitations from the related issue of curve reconstruction in the plane, our method can only deal with closed non-intersecting curves and it is sensitive to non-uniform and sparse sampling. Moreover, being an interpolatory method, curves with noise or outliers would incorporate the noise in the reconstruction instead of eliminating or modifying the incorrect samples.

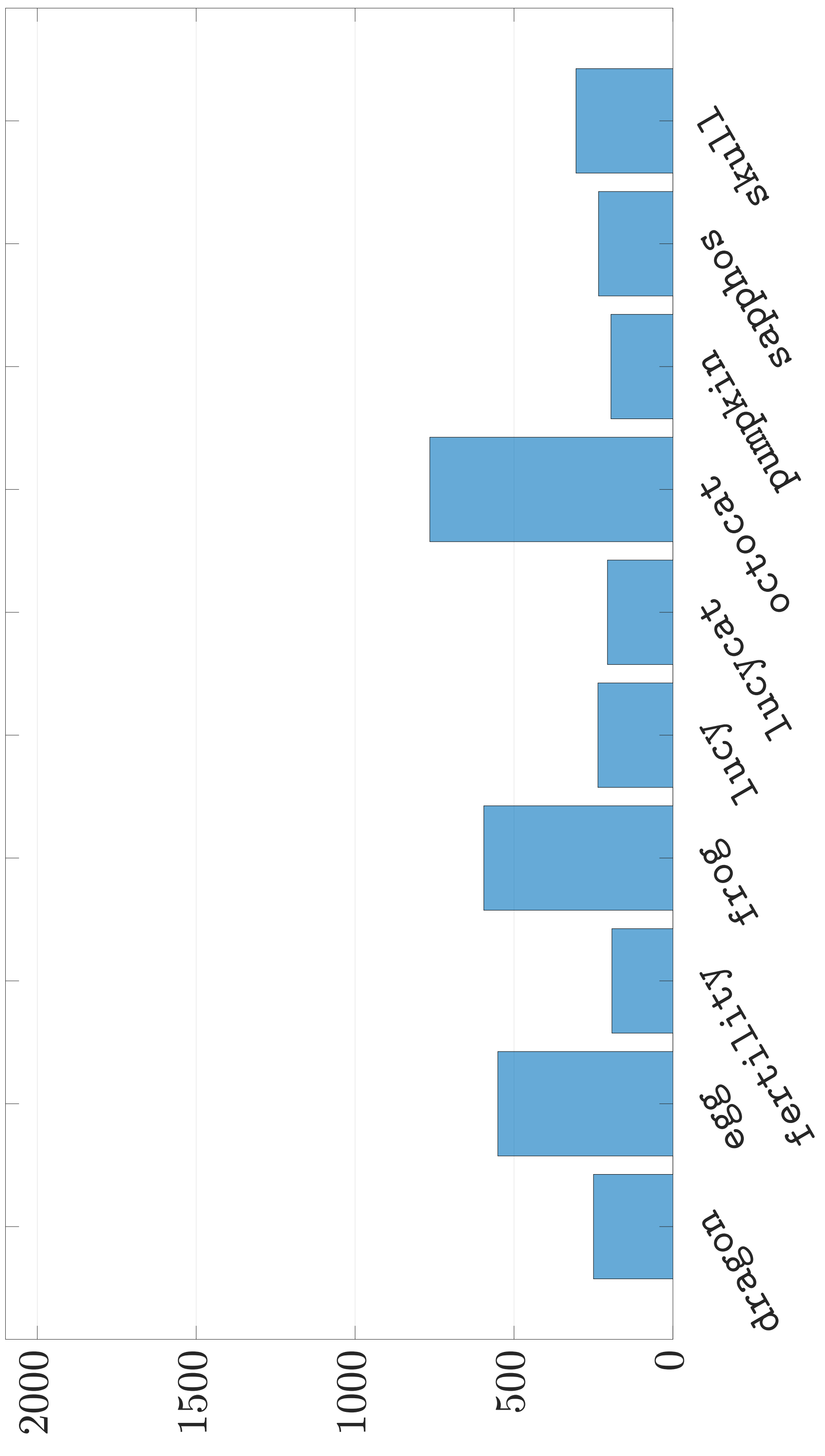


Figure 6.10. Distribution of failure cases among the meshes in the dataset. Each mesh is tested on 2.1k curves, and the meshes that fail most are those with lower resolution or with low-quality triangulation.

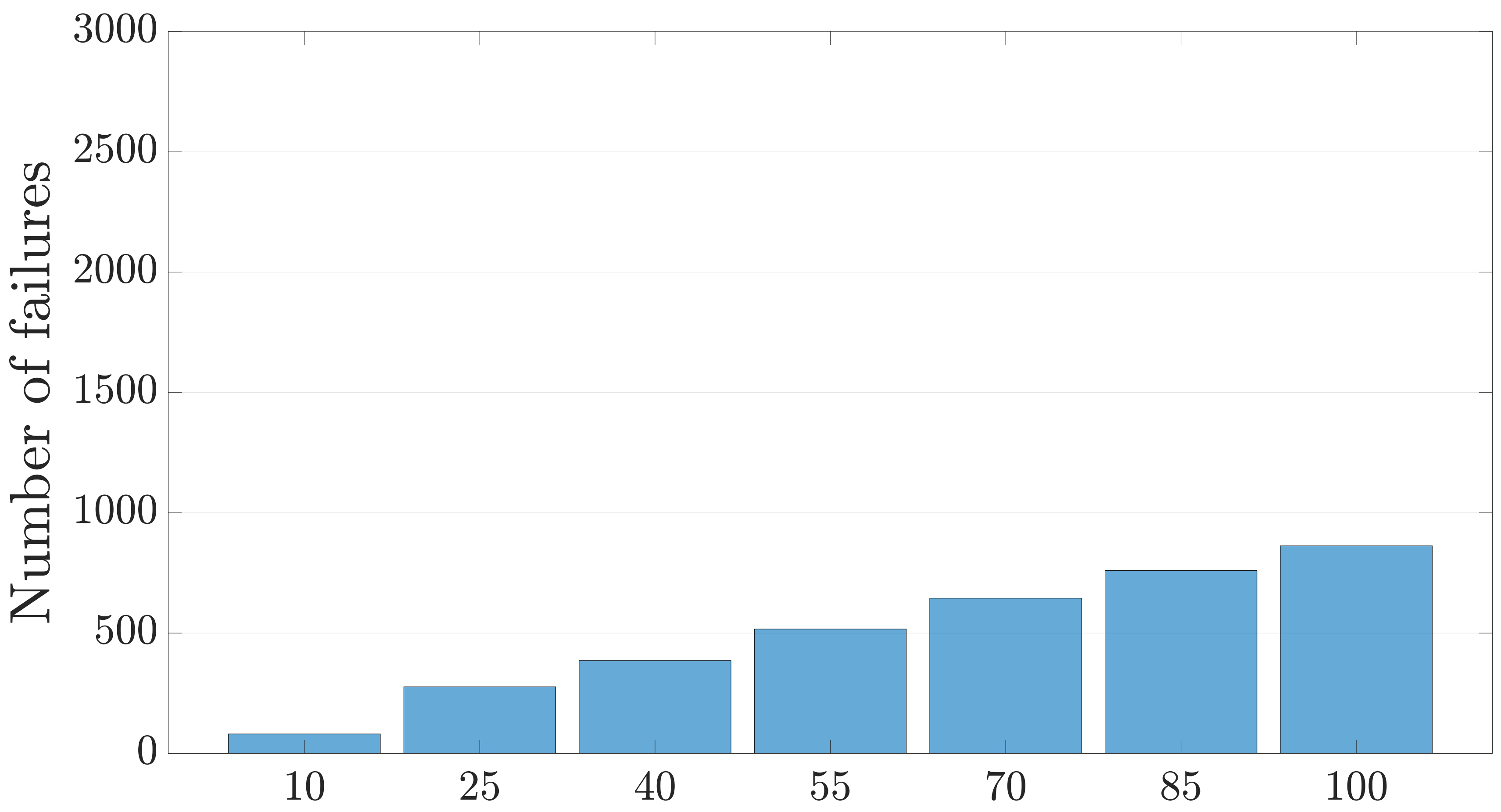


Figure 6.11. Distribution of failure cases among runs with different sampling resolutions. Each of the samples count is tested on 3k curves. Increasing the number of samples provides additional constraints, inducing the greedy behavior of the algorithm to take the wrong decision and partition the mesh before closing the path.

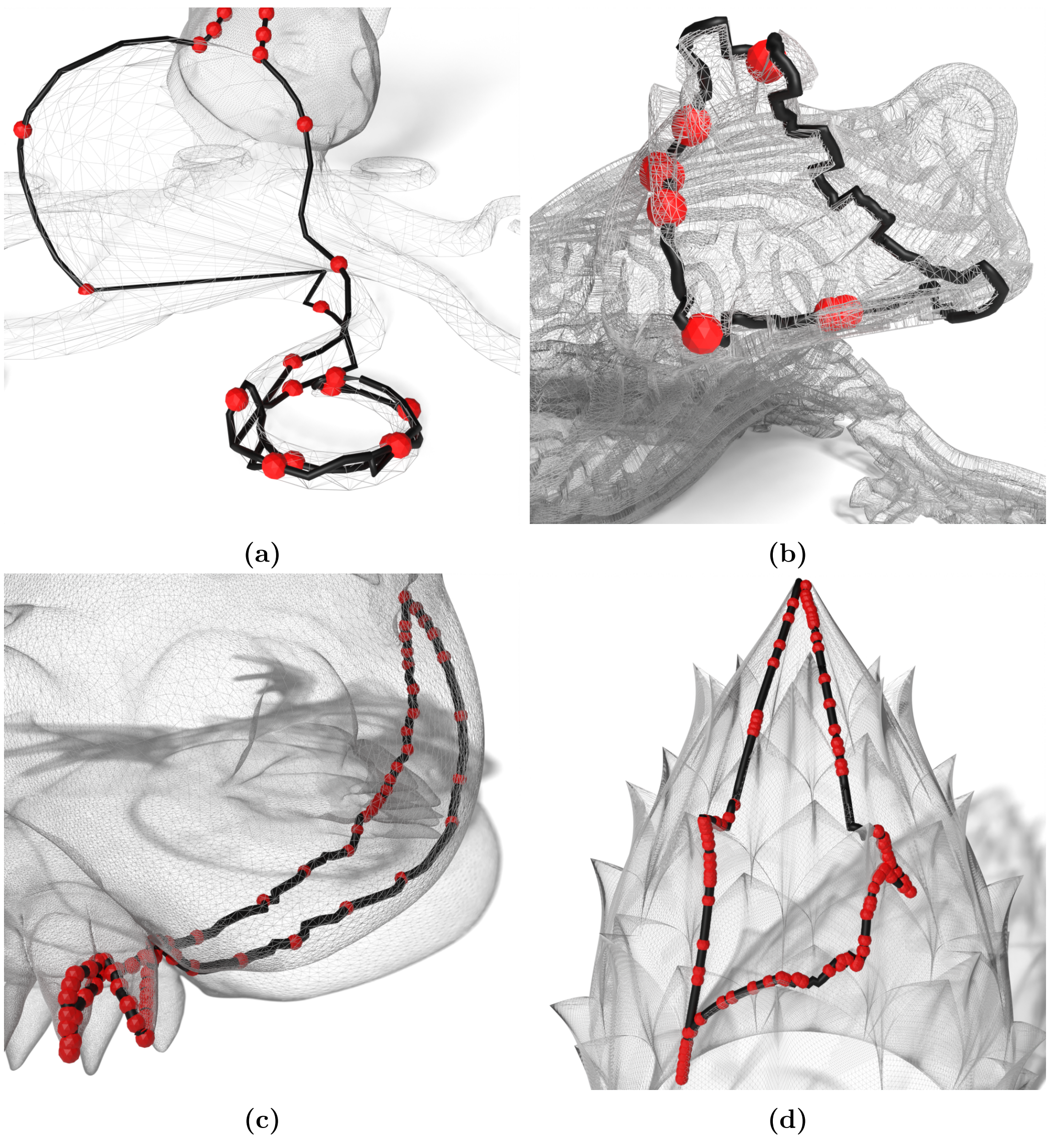


Figure 6.12. Examples of failure cases - samples, illustrated in red, are a subset of the ground truth black curve on the surface. The poor quality of the mesh, explained by the non-uniformity of triangles (Subfigures 6.12a, 6.12d) together with the unconstrained generation of the curve dataset (Subfigure 6.12c), result in ill-defined paths that our algorithm cannot reconstruct. Sampling density is another factor that prevents our algorithm from generating a reconstruction - Subfigure 6.12b.

6.5 Conclusions

In this work, we introduced a generalization of the curve reconstruction task on triangular mesh domains, showing its relevance in different research fields and real-world applications. We presented an efficient greedy algorithm for solving this problem, adapting existing solutions for the traveling salesman problem in $\mathbb{R}^2$ to non-Euclidean domains.

For extensive testing, we generated a dataset of curves on surfaces, on shapes with variable density, genus, connectivity, and topology. The analysis of our results showed that our method is suitable for addressing curve reconstruction on surfaces, producing good-looking curves that fit the ground truth well while efficiently dealing with meshes at very high resolutions. We discussed the ability of our method to produce the correct ordering of the input samples, which enables our method to be integrated into more complex pipelines of curve generation and design that are not bound to the original triangulation, such as splines approximately fitting the samples.

As a future direction, we intend to explore this task deeper and improve our method. Curve reconstruction is a well-established field, counting a large variety of approaches and solutions. Adapting state-of-the-art methods for planar reconstruction to non-Euclidean geometries could result in more effective and efficient approaches. Furthermore, we recognize the absence of a dataset as a major limitation in our testing pipeline, leading to extra difficulties in analyzing the quality of our solution. Finally, dealing with vertex paths bounds us to the mesh quality and resolution. Addressing the problem on the entire domain of the surface would be a first step towards producing curves of higher quality.

Part III

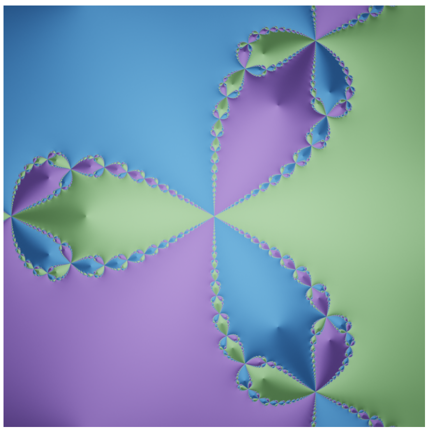
Parallelization & Optimization

Chapter 7

Newton's Fractals on Surfaces via Bicomplex Algebra

In this chapter we propose a novel algorithm for decorating surfaces with Newton's fractal patterns only relying on a pixel shader. We exploit the properties of the algebraic field of bicomplex numbers to generalize Newton's fractals to 4 dimensions and efficiently evaluate the shader with an iterative solver. This work was realized in collaboration with Daniele Baieri (MSc.), to whom goes the credit for improving the smooth gradient formula and producing some of the rendered scenes, prof. Simone Melzi and prof. Emanuele Rodolà, both of which largely helped in improving the overall presentation. The results presented in this chapter have been published in the poster proceedings of the *Special Interest Group on Computer Graphics and Interactive Techniques* (*SIGGRAPH*) [157].

7.1 Introduction



The Newton-Raphson method is a well-known iterative method used to find roots of any function $f : \mathbb{C} \to \mathbb{C}$. Nearby points usually converge to the same solution. Hence one can identify regions associated with each solution, whose boundaries describe fractal patterns [212]. This type of pattern is known as Newton's fractal and is typically used to generate interesting visualizations and effects like the one shown in the inset figure. The usual approach is to define some polynomial $p(z)$ with roots $\{\xi_i\}_{i=1}^{n}$ and apply the Newton-Raphson method to all points on the plane. Each point is associated with an index i corresponding to the solution ξ_i it converged to and then colored with some coloration $\mathcal{C}(i)$. Despite the beautiful visualizations that can arise, Newton's fractals can only be used to generate image patterns.

Bicomplex numbers are a generalization of complex numbers that define a closed and commutative algebra in four dimensions [64, 65]. The bicomplex field $\mathbb{BC}$ is

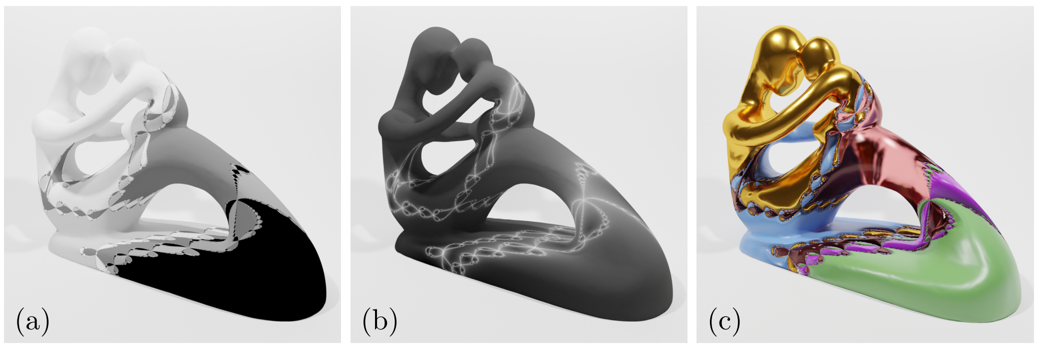


Figure 7.1. A bicomplex Newton's fractal on a surface (a), the map of its convergence speed (b) and a material built over these two maps (c).

isomorphic to the field of 2×2 matrices over $\mathbb{C}$ spanned by the following basis:

$$\mathbf{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \boldsymbol{i} = \begin{pmatrix} i & 0 \\ 0 & i \end{pmatrix} \quad \boldsymbol{j} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \boldsymbol{k} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} . \tag{7.1}$$

Any bicomplex number can always be described as $z = x\mathbf{1} + y\boldsymbol{j}$, being $x, y \in \mathbb{C}$. Differently from the complex field, bicomplex algebra is isomorphic to a direct sum of two algebras over $\mathbb{C}$ [65]. In fact, one can construct two idempotent, zero divisors and orthogonal elements $\boldsymbol{e} = (1+\boldsymbol{k})/2$ and $\boldsymbol{e}^\dagger = (1-\boldsymbol{k})/2$ such that every bicomplex number z can be uniquely decomposed into $z = \alpha\boldsymbol{e} + \beta\boldsymbol{e}^\dagger$, making effectively $\{\boldsymbol{e}, \boldsymbol{e}^\dagger\}$ a basis for $\mathbb{BC}$ over the scalar field $\mathbb{C}$.

Most of the properties of complex numbers still hold in bicomplex algebra [236]. Bicomplex numbers are commutative and invertible, and the elementary functions can be easily extended [155]. The derivative of a bicomplex function $f : \mathbb{BC} \to \mathbb{BC}$ is well defined, and most of the differentiation rules (*e.g.*, derivatives of elementary functions, chain rule, product rule, linearity) still apply.

Bicomplex numbers have already been used in abstract mathematics and computer graphics applications for describing fractals [282]. However, previous work is limited to considering classic escape-time fractals, like the Mandelbrot and Julia sets. In this work, we show that it is possible to use bicomplex numbers to generalize even the well-known Newton's fractal. Moreover, we also provide insights for possible application in procedural texturing and volumetric rendering.

7.2 Method

Since bicomplex functions in the form $f : \mathbb{BC} \to \mathbb{BC}$ can be expanded with Taylor [236], and the Taylor series in $\mathbb{BC}$ converges, we can use the classical proof of the Newton-Raphson method to show that Newton's iteration converges to the root of a function: this allows us to generalize Newton's fractal to bicomplex numbers, and use it to generate 4-dimensional patterns.

In the complex plane, the roots of the polynomial $c^n - 1 = 0$ lie on the unit circle and are equispaced, and the interesting patterns arising from this particular family of polynomials are shown in most visualizations of Newton's fractal. This still holds in $\mathbb{BC}$, where the roots of $z^n - 1 = 0$ are equispaced on the unitary 4-dimensional

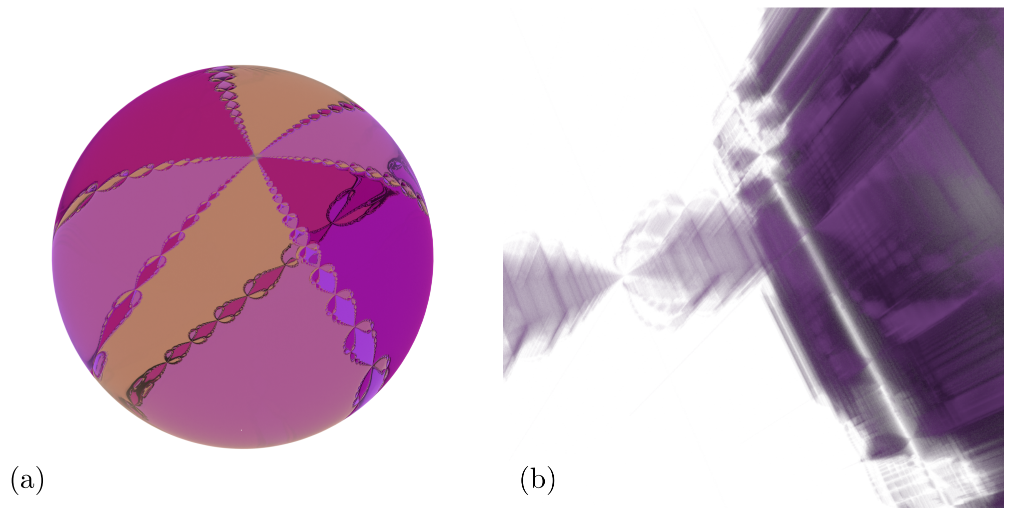

(a) (b)

Figure 7.2. An example of bicomplex Newton's fractal used as decoration (a). A volumetric rendering of a region that identifies a solution of $z^3 - 1 = 0$ (b).

hyper-sphere [218]. This polynomial has n^2 closed form solutions [218, 155] $\beta_{ij} = \alpha_i \boldsymbol{e} + \alpha_j \boldsymbol{e}^\dagger$, with $\alpha_k = \cos(2\frac{k-1}{n}\pi) + i\sin(2\frac{k-1}{n}\pi)$.

Applying the Newton-Raphson method to solve bicomplex polynomials in the form $z^n - 1 = 0$ produces the same patterns of the complex plane, but it extends to 4 dimensions. By using the 3D coordinates of the visualized points, we can generate interesting and complex patterns, as we show in Figures 7.1a and 7.2a, while using the fourth coordinate either as a tunable parameter or as a time dependency, allowing for an additional degree of freedom.

7.3 Implementation

We implement the algorithm as a pixel shader, where each thread computes Newton's iteration on the given point. This gives us the possibility to fully exploit GPU computing.[1]

We optimize our iteration scheme using ad-hoc algebraic manipulations for the specific polynomial in use:

$$z_{k+1} = z_k - \frac{f(z_k)}{\frac{\partial f(x_k)}{\partial z}} = z_k - \frac{z_k^n - 1}{n\ z_k^{n-1}} = \frac{1}{n}\left((n-1)z_k + z_k^{1-n}\right). \tag{7.2}$$

This way we avoid bicomplex division, which usually requires 64 FLOPs. We also take advantage of the decomposition in the basis $\{\boldsymbol{e}, \boldsymbol{e}^\dagger\}$ to efficiently compute powers: in fact, since both $\boldsymbol{e}$ and $\boldsymbol{e}^\dagger$ are idempotent, and since $\boldsymbol{e}\boldsymbol{e}^\dagger = 0$, then we have $z^n = (\alpha\boldsymbol{e} + \beta\boldsymbol{e}^\dagger)^n = \alpha^n\boldsymbol{e} + \beta^n\boldsymbol{e}^\dagger$ (for any $n \in \mathbb{Z}$), and complex powers can be computed efficiently using polar coordinates.

We exploit the decomposition to represent bicomplex numbers in the whole shader. Other than simplifying and speeding up the computation, this representation also makes it easier to compute which bicomplex solution a number converged

[1] A sample implementation with Unity surface shaders that achieves real-time performance is available at https://github.com/filthynobleman/newton-fractals.

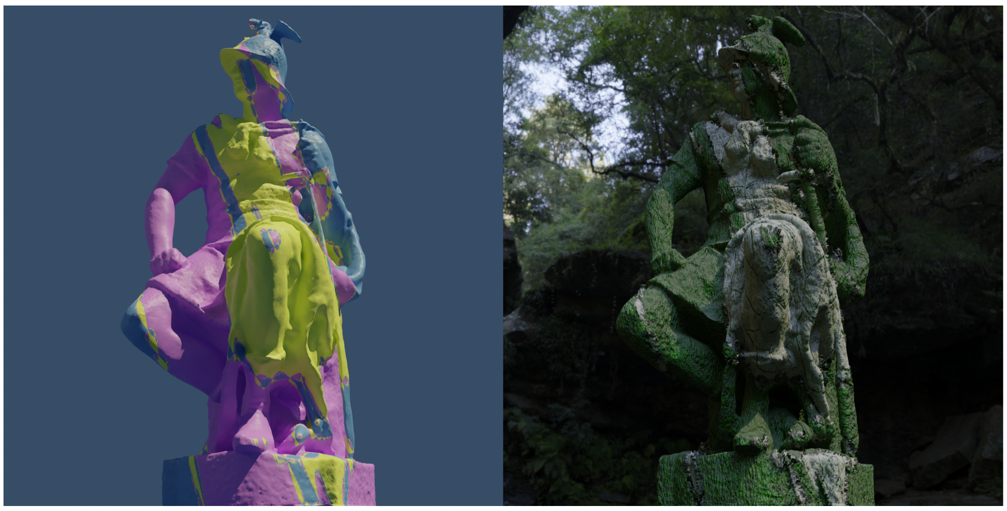


Figure 7.3. An example of bicomplex Newton's fractal used as mask for mixing different materials.

to. If a number converged to some $z = \alpha \boldsymbol{e} + \beta \boldsymbol{e}^\dagger$, we can reduce the problem to find to which complex solution α and β converged.

A smooth gradient representing the speed of convergence typically enriches Newton's fractals using it, for example, as a heightmap for additional detail in the texture. Among the existing approaches, we compute this gradient as

$$t = P - \sum_{k=0}^{P} \frac{1}{1 + \exp(\delta_k + \theta) - \exp(\theta)}\,, \tag{7.3}$$

where $\delta_k = |z_{k+1} - z_k|$ is the magnitude of each step, P is the number of Newton iterations and θ is a tunable parameter. The same approach can be easily generalized to a bicomplex setting, as we show in Figure 7.1b.

7.4 Results and Conclusions

Bicomplex numbers offer an instrument for generalizing Newton's fractal to 4 dimensions. This type of fractal offers a new possibility for procedural generation of 4D textures. We have shown that bicomplex Newton's fractals generate interesting and complex patterns, similar to their complex counterpart. Our results prove that these patterns can fit all the most common applications, from decorating surfaces (Figure 7.1c) and masking materials (Figure 7.3), to volumetric rendering of fractal regions (Figure 7.2b).

Chapter 8

Efficiently Parallelizable Strassen-Based Multiplication of a Matrix by its Transpose

The multiplication of a matrix by its transpose, $\mathbf{A}^T\mathbf{A}$, appears as an intermediate operation in the solution of a wide set of problems. In this chapter, we propose a new cache-oblivious algorithm (AtA) for computing this product, based upon the classical Strassen algorithm as a sub-routine. In particular, we decrease the computational cost to $2/3$ the time required by Strassen's algorithm, amounting to $\frac{14}{3}n^{\log_2 7}$ floating point operations. AtA works for generic rectangular matrices, and exploits the peculiar symmetry of the resulting product matrix for saving memory. In addition, we provide an extensive implementation study of AtA in a shared memory system, and extend its applicability to a distributed environment. To support our findings, we compare our algorithm with state-of-the-art solutions specialized in the computation of $\mathbf{A}^T\mathbf{A}$. Our experiments highlight good scalability with respect to both the matrix size and the number of involved processes, as well as favorable performance for both the parallel paradigms and the sequential implementation, when compared with other methods in the literature. The work presented in this chapter has been realized in collaboration with Viviana Arrigoni (Ph.D.), who designed the overall algorithmic scheme and with whom I worked on the algorithmic analysis and the design details of the distributed algorithm, prof. Annalisa Massini and prof. Emanuele Rodolà. The results presented in this chapter have been published at the *International Conference on Parallel Processing* (*ICPP*) [12].

8.1 Introduction

Matrix multiplication is a fundamental operation in Linear Algebra and HPC, as it appears as an intermediate step in a wide set of problems. Many researchers have devoted their efforts to the algorithmic aspects of matrix multiplication, with the aim of improving the computational cost of existing algorithms and to devise and implement new solutions for parallel architectures. Designing a distributed algorithm for matrix multiplication is a challenging task, due to the inherent dependence of the data scattered in the system's distributed memory, and due to the overhead due to the communication cost of assembling the resulting product matrix.

The product of a matrix by its transpose, $\mathbf{A}^T\mathbf{A}$ (as well as $\mathbf{A}\mathbf{A}^T$), is a particular matrix multiplication involved in several applications. For example, computing $\mathbf{A}\mathbf{A}^T$ is a straightforward, yet effective, method to check for orthogonality or to project vectors onto the space spanned by the columns of $\mathbf{A}$. This product, in fact, is repeatedly computed in the Gram-Schmidt algorithm for vector basis orthogonalization, where $\mathbf{A}$ is the progressively built projection matrix. One way to solve the least squares problem of under and over determined linear systems $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, is to solve the associated system of normal equations, obtained by left-hand multiplying the original system by $\mathbf{A}^T$, thus obtaining a square linear system $\mathbf{A}^T\mathbf{A}\boldsymbol{x} = \mathbf{A}^T\boldsymbol{b}$. Also, the Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$ can be computed by studying the eigenproblem for $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$. Furthermore, the product of a matrix by its transpose commonly arises in discrete exterior calculus and discrete differential geometry. One example is given by the discrete heat kernel $\mathbf{K}(t) = \mathbf{\Phi}\mathbf{E}(t)\mathbf{\Phi}^T$, with $\mathbf{E}(t) = \exp(-\mathbf{\Lambda}t)$ being a diagonal matrix, so that $\mathbf{K}(t) = (\mathbf{\Phi}\mathbf{E}(t)^{1/2})(\mathbf{\Phi}\mathbf{E}(t)^{1/2})^T$ can be efficiently computed [298]. Many other applications of the product $\mathbf{A}^T\mathbf{A}$ are described in [257], together with its properties such as positive semi-definiteness. In this work, we consider the multiplication between $\mathbf{A}^T$ and $\mathbf{A}$, where $\mathbf{A}$ may have any size and shape. We rely on a recursive approach that, as described in [211], is virtually tuning free and avoids the significant tuning efforts that are required by iterative blocked algorithms to achieve near-optimal performance. Our contribution is threefold.

- First, we introduce AtA (Section 8.3), a cache-oblivious algorithm for computing $\mathbf{A}^T\mathbf{A}$ that requires $2/3n^{(\log_2 7)} + 1/3n^2$ multiplications. We exploit the self-similarity of the $\mathbf{A}^T\mathbf{A}$ product with its sub-problems and the Strassen's algorithm, that is recursively applied to possibly rectangular matrices, without introducing additional computational and space cost, deriving from dynamic peeling and padding, as in [108, 267]. In contrast to [75], our algorithm works on any algebraic field. We prove that AtA exhibits high efficiency for both memory and time, and show that it is efficiently implementable, as it does not hide large constant factors. We also describe our implementation of Strassen's algorithm, and compare its performance with that of the Intel MKL BLAS `gemm` routine for matrix multiplication.

- Second, we describe AtA-S, our multi-threaded implementation of AtA for a shared memory system, relying on OpenMP (Section 8.4.2). A well-engineered scheduler that assigns different tasks to each thread in such a way that computations can be carried out in perfect parallelism by preventing memory collisions. Performance evaluation shows that our implementation outperforms the multi-threaded Intel MKL BLAS routines (e.g. `syrk` for symmetric rank-K update) on large matrices, even on Intel processors.

- Finally, we extend our approach to distributed systems, leveraging the standard message-passing paradigm MPI. Our distributed algorithm AtA-D allows the distribution of the computational effort among a larger number of processes (Section 8.4.3). This is particularly convenient on very large matrices.

To validate the effectiveness of our algorithms, we study their performance by running a set of tests on dense matrices of variable size (Section 8.5). We analyse different metrics for evaluating the scalability of our parallel implementation, and compare our results with benchmark solutions for distributed systems. We run tests on a cluster of multi-core nodes endowed with 2×8 core Intel Xeon E5-2630v3 processors, 2.4 Ghz, 4 GB RAM/core.

8.2 Related Work

Nowadays, matrix multiplication is still a hot topic in HPC and numerical algorithmics. In 1969, Strassen [258] was the first to reduce the computational complexity of the standard matrix multiplication from $O(n^3)$ to $O(n^{\log_2 7})$. More recently, Coppersmith and Winograd [59] devised an algorithm for matrix multiplication running in $\sim O(n^{2.38})$ time. In the last decade, many have devoted their efforts to improve this limit ([256, 288, 89]). These works make use of algebraic tensors that, despite the elegance of the resulting method, are still hardly used in practice as they come at the cost of very large hidden constants and frequent memory access.

Several authors have designed hybrid algorithms, deploying Strassen's multiplication in conjunction with conventional matrix multiplication, to overcome the overhead of Strassen's algorithm on small matrices, see, e.g., [36, 37, 102, 108, 25]. Huss-Lederman *et al.* [108] propose two techniques, known as dynamic peeling and static padding, in order to apply Strassen's algorithm to odd-sized matrices. Thottethodi *et al.* [267] propose two strategies to optimize memory efficiency in Strassen by minimizing padding and peeling operations. Many researchers have proposed a parallel implementation of Strassen's algorithm. In [156], Luo and Drake explored Strassen-based parallel algorithms that use the communication patterns known for classical matrix multiplication. They considered using a classical 2D parallel algorithm and using Strassen locally and at the highest level. This approach is improved in [94] by using a more efficient parallel matrix-multiplication algorithm running on a more communication-efficient machine. In [63], Strassen's algorithm is extended to deal with rectangular and arbitrary-size matrices. Their approach leverages on a suitable combination of Strassen's with ATLAS and GotoBLAS. Other parallel approaches [68, 107, 252] have used more complex parallel schemes and communication patterns, and consider at most two steps of Strassen. In [18], a parallel algorithm based on Strassen's fast matrix multiplication, Communication - Avoiding Parallel Strassen (CAPS), is described. The authors show that its complexity matches the communication lower bounds described in [19]. This work is extended in [67] to handle rectangular matrices (CARMA). More recently, Kwasniewski *et al.* [130] proposed a near optimal algorithm for matrix multiplication that models the matrix multiplication dependencies by the red-blue pebble game [112] to derive an I/O optimal schedule, improving the performance of previous works.

Both Strassen's algorithm and AtA fall into the class of recursive blocked algorithms. The work in [82, 117] proves the effectiveness of this kind of algorithms for dense Linear Algebra. The work in [81] introduces FRPA, an interface for implementing recursive problems in parallel that gets as an input the recursive problem, and handles parallelization and auto-tuning automatically. Similarly to our approach, Charara *et al.* [49] propose block recursive matrix multiplication and linear solver algorithms. They show how recursion enhances data reuse and concurrency in GPUs. Differently from the work presented in this chapter, they specialize on triangular matrices. In [48], the authors also adapt this blocking strategy to handle batched operations on small matrix sizes (up to 256) to stress the register usage and maintain data locality. In [211], Elmar and Bientinesi introduce ReLAPACK, a collection of recursive algorithms for dense Linear Algebra. While this work corroborates the recursive approach that we implement in our algorithms, it does not provide a routine specialized in the $\mathbf{A}^T\mathbf{A}$ product for general matrices. Instead, they propose a routine for the same multiplication only on triangular matrices. We highlight that the solutions proposed for the multiplication of a matrix by its transpose on triangular matrices (TRSYRK) is useful for many applications but cannot be applied on general matrices.

Although much research has been devoted to optimizing the implementation of parallel matrix multiplication, very few solutions have been proposed for the $\mathbf{A}^T\mathbf{A}$ multiplication. In [75], Dumas *et al.* propose an algorithm for the product $\mathbf{A}\mathbf{A}^T$ whose computational complexity is improved by a constant factor with respect to previously known reductions. This approach is applicable only to matrices lying in fields where skew-orthogonal matrices exist (e.g., $\mathbb{C}$ and finite fields of prime characteristics), which is not the case for $\mathbb{R}$ and $\mathbb{Q}$, that instead are important in many applications, such as the study of embedded systems, computational geometry and system simulations.

Except for some sporadic attempts to implement a method for distributing in a balanced way the workload for matrix multiplication among processes with the MapReduce programming model [221, 116], the approach that we implement here for the distributed parallel model has barely been investigated.

8.3 AtA

In this section, we describe our sequential recursive algorithm for the matrix multiplication $\mathbf{A}^T\mathbf{A}$, dubbed AtA, and we provide implementation details. We remark that our solution also works for the product $\mathbf{A}\mathbf{A}^T$. Yet, when row-major order is the default layout for array storage, the $\mathbf{A}^T\mathbf{A}$ multiplication is in practice harder to perform, as memory access is inherently column-wise, hence not cache friendly. Since AtA includes calls to Strassen for generic matrix multiplications, we also outline a time and space efficient implementation for this algorithm.

8.3.1 AtA in Detail

Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ be a rectangular matrix. The idea behind AtA is the following: at each recursive step, matrix $\mathbf{A}$ is divided into four sub-matrices as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \qquad \begin{aligned} \mathbf{A}_{1,1} &= \mathbf{A}_{0:m_1,0:n_1} \in \mathbb{R}^{m_1\times n_1} \\ \mathbf{A}_{1,2} &= \mathbf{A}_{0:m_1,n_1:n} \in \mathbb{R}^{m_1\times n_2} \\ \mathbf{A}_{2,1} &= \mathbf{A}_{m_1:m,0:n_1} \in \mathbb{R}^{m_2\times n_1} \\ \mathbf{A}_{2,1} &= \mathbf{A}_{m_1:m,n_1:n} \in \mathbb{R}^{m_2\times n_2} \end{aligned} \tag{8.1}$$

being $m_1 := \lfloor \frac{m}{2} \rfloor$, $m_2 := \lceil \frac{m}{2} \rceil$, $n_1 := \lfloor \frac{n}{2} \rfloor$, $n_2 := \lceil \frac{n}{2} \rceil$. We address to sub-matrices of a matrix $\mathbf{A}$ as to indexed sub-blocks ($\mathbf{A}_{i,j}$) or with line and column intervals ($\mathbf{A}_{r_1:r_2,c_1:c_2}$). The product matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ is also split into four sub-matrices, resulting in the following:

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}^T\mathbf{A}_{1,1} + \mathbf{A}_{2,1}^T\mathbf{A}_{2,1} \in \mathbb{R}^{n_1 \times n_1},\\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}^T\mathbf{A}_{1,2} + \mathbf{A}_{2,1}^T\mathbf{A}_{2,2} \in \mathbb{R}^{n_1 \times n_2},\\
\mathbf{C}_{2,1} &= \mathbf{A}_{1,2}^T\mathbf{A}_{1,1} + \mathbf{A}_{2,2}^T\mathbf{A}_{2,1} \in \mathbb{R}^{n_2 \times n_1},\\
\mathbf{C}_{2,2} &= \mathbf{A}_{1,2}^T\mathbf{A}_{1,2} + \mathbf{A}_{2,2}^T\mathbf{A}_{2,2} \in \mathbb{R}^{n_2 \times n_2}.
\end{aligned} \tag{8.2}$$

Both $\mathbf{C}_{1,1}$ and $\mathbf{C}_{2,2}$ consist of two addends that are, in turn, the left hand product of a matrix by its transpose. Hence, four recursive calls are employed to compute the sub-products $\mathbf{A}_{1,1}^T\mathbf{A}_{1,1}$ and $\mathbf{A}_{2,1}^T\mathbf{A}_{2,1}$ to obtain $\mathbf{C}_{1,1}$, and $\mathbf{A}_{1,2}^T\mathbf{A}_{1,2}$ and $\mathbf{A}_{2,2}^T\mathbf{A}_{2,2}$ to obtain $\mathbf{C}_{2,2}$.

Since for any matrix $\mathbf{A}$ the product $\mathbf{A}^T\mathbf{A}$ is symmetric, at each recursive step only the lower triangular part of the product matrix is computed, $\text{low}(\mathbf{C}_{i,i})$, $i = 1, 2$. As for component $\mathbf{C}_{2,1}$, in order to compute its two terms in the sum, we implement the generalized Strassen's algorithm for non-square matrices. The sub-matrix $\mathbf{C}_{1,2}$ is equal to $\mathbf{C}_{2,1}^T$, and therefore must not be explicitly computed. In Algorithm 8.1 we provide the pseudo-code of AtA. The base case occurs when the number of entries of the sub-matrix fits in the cache. In that case, the multiplication is performed by the BLAS function for $\mathbf{A}^T\mathbf{A}$, `?syrk`, where the character `?` represents a generic data type in accordance with standard notation used in manuals, [86]. In Algorithm 8.1, we also sketch our implementation of Strassen: before the actual recursive Strassen algorithm is called (Strassen), in FastStrassen we conveniently prepare an environment for memory efficiency by pre-allocating the memory for Strassen's algorithm, as explained in Section 8.3.3. The reduced number of multiplications in Strassen's algorithm is achieved by computing more matrix additions. In our implementation of Strassen, matrix additions are performed by calling the BLAS routine `?axpy` (for the vector addition $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$). The base-case condition in Strassen is analogous to the one of AtA. When the base-case condition holds, we call the BLAS routine `?gemm` for the generic $\mathbf{A}^T\mathbf{B}$ multiplication. To handle odd-sized matrices, we do not implement well-known strategies such as peeling or padding, since these are known for introducing computational and memory overhead. Instead, we manage sums between matrices of discordant size by conveniently applying the BLAS routine `?axpy` for array sums, so that it simulates padding of an extra 0 column or row, by excluding the last row and/or column of a sub-matrix from the sum.

AtA and FastStrassen are designed to be efficient alternatives to the BLAS routines `?gemm` and `?syrk`. Thus, they perform the same operations, respectively $\mathbf{C} = \alpha\mathbf{A}^T\mathbf{B} + \beta\mathbf{C}$ and $\mathbf{C} = \alpha\mathbf{A}^T\mathbf{A} + \beta\mathbf{C}$. However, we avoid introducing the scaling factor β from our algorithms for clarity of exposure, since $\mathbf{C}$ can be simply scaled before applying the algorithms.

Algorithm 8.1 AtA- Serial

```
 1: procedure AtA($\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{C} \in \mathbb{R}^{n\times n}, \alpha \in \mathbb{R}$)
 2:     if $m \times n \leq cache\_size$ then
 3:         blas_?syrk($\mathbf{A}, \mathbf{C}, \alpha$)
 4:         return
 5:     end if
 6:     Initialize pointers to $\mathbf{A}_{i,j}$, $\mathbf{C}_{i,j}$, $i, j = 1, 2$
 7:     AtA($\mathbf{A}_{1,1}, \mathbf{C}_{1,1}, \alpha$)
 8:     AtA($\mathbf{A}_{2,1}, \mathbf{C}_{1,1}, \alpha$)
 9:     AtA($\mathbf{A}_{1,2}, \mathbf{C}_{2,2}, \alpha$)
10:     AtA($\mathbf{A}_{2,2}, \mathbf{C}_{2,2}, \alpha$)
11:     FastStrassen($\mathbf{A}_{1,2}, \mathbf{A}_{1,1}, \mathbf{C}_{2,1}, \alpha$)
12:     FastStrassen($\mathbf{A}_{2,2}, \mathbf{A}_{2,1}, \mathbf{C}_{2,1}, \alpha$)
13: end procedure

14: procedure FastStrassen($\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{m\times k}, \mathbf{C} \in \mathbb{R}^{n\times k}, \alpha \in \mathbb{R}$)
15:     Allocate $\mathbf{M} = \mathbf{0}^{n\times k/2}$
16:     Allocate $\mathbf{P} = \mathbf{0}^{m\times n/2}$
17:     Allocate $\mathbf{Q} = \mathbf{0}^{m\times k/2}$
18:     Strassen($\mathbf{M}$, $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\alpha$)
19: end procedure
```

8.3.2 Computational Complexity

The idea behind Strassen's algorithm is to perform a 2×2 matrix multiplication using 7 multiplications instead of 8, as required by naive matrix multiplication [258]. Nevertheless, Strassen's algorithm involves 18 sums between sub-matrices, thus leading to a computational complexity $T_S(n) \approx 7n^{\log_2 7}$. In Algorithm 8.1, there are four recursive calls to AtA on basically halved dimensions, two calls to FastStrassen and 3 sums. Thus, we can derive the recurrence function for AtA runtime depending on the input size n as follows:

$$T(n) = 4T\left(\frac{n}{2}\right) + 2T_S\left(\frac{n}{2}\right) + 3\left(\frac{n}{2}\right)^2 \approx \frac{2}{3}T_S(n). \tag{8.3}$$

The overall computational complexity of AtA reduces the one of the general matrix multiplication $\mathbf{A}^T\mathbf{A}$, amounting to $n^2(n+1)$, and of Strassen's algorithm naively applied for computing $\mathbf{A}^T\mathbf{A}$, that would require the same number of products as for the general matrix multiplication, and only 16 sums instead of the 18 matrix additions in the original Strassen's formulation.

8.3.3 Space Complexity

In AtA, at each recursive step, pointers to the current portions of $\mathbf{A}$ and $\mathbf{C}$ are initialized so that, when the condition for the base-case occurs, the matrix multiplications are carried out on the correct sub-matrices of $\mathbf{A}$, and stored in the corresponding locations in $\mathbf{C}$.

Algorithm 8.2 RecursiveGEMM

```
1:  procedure RecursiveGEMM($\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{m \times k}, \mathbf{C} = \mathbf{0}^{n \times k}$)
2:      if $m \times n + m \times k \leq cache\_size$ then
3:          blas_?gemm($\mathbf{A}^T, \mathbf{B}, \mathbf{C}$)
4:      end if
5:      for $i = 1, 2$ do
6:          for $j = 1, 2$ do
7:              for $k = 1, 2$ do
8:                  RecursiveGEMM($\mathbf{A}_{k,i}$, $\mathbf{B}_{k,j}$, $\mathbf{C}_{i,j}$)
9:              end for
10:         end for
11:     end for
12: end procedure
```

Strassen's algorithm for general matrix multiplication is called twice. One drawback of the naive Strassen implementation is the great amount of memory allocated at each recursive step to store the results of the intermediate matrix additions required by the algorithm. In order to avoid frequent memory allocations and releases, we call recursive Strassen (Strassen) on pre-allocated matrices, **M**, **P** and **Q** (FastStrassen). The size of such matrices is sufficiently large to store all intermediate matrix operation results throughout the recursive calls. In fact, given an $n \times n$ matrix, at each recursive step we halve both the dimensions, rounding up the result to the nearest integer when matrices have odd sizes. By doing so, the amount of memory used by the algorithm when the base case is reached if

$$\sum_{i=1}^{\log_2 n} \frac{(n + \log_2 n)^2}{4^i} = (n + \log_2 n)^2 \left(\frac{1}{3} - \frac{4}{3n^2} \right) \leq \frac{n^2}{2} \tag{8.4}$$

which, multiplied by the three supporting matrices **M**, **P** and **Q**, results in a total of $\frac{3}{2}n^2$. Although the overall space complexity of Strassen does not change, we are able to save time for memory allocation at each recursive step. Consequently, the space complexity of AtA is $S(n) = \frac{3}{2}n^2$.

In Section 8.5, we show that Strassen's algorithm benefits from the described strategy for memory allocation.

8.3.4 Cache Complexity

In this section, we show the cache complexity of AtA. We assume the ideal cache model and we denote with M the cache size, and with b the size of the cache line.

Proposition 8.1. *The cache complexity of AtA, $C_{\text{AtA}}(n; M, b)$, is the same as the cache complexity of Strassen, $C_S(n; M, b) = \Theta(1 + n^2/b + n^{\log_2(7)}/b\sqrt{M})$, [88].*

Proof. We prove the thesis by induction. First, we observe that $C_{\text{AtA}}(2; M, b) = 6C_S(1; M, b) \leq 7C_S(1; M, b) = C_S(2; M, b)$. Assuming as inductive hypothesis that

$C_{\text{AtA}}(n/2; M, b) \le C_S(n/2; M, b)$, it holds that:

$$\begin{aligned} C_{\text{AtA}}(n; M, b) &= 4C_{\text{AtA}}(n/2; M, b) + 2C_S(n/2; M, b) \\ &\le 6C_S(n/2; M, b) \le 7C_S(n/2; M, b) = C_S(n; M, b). \end{aligned}$$

Furthermore, notice that: $C_S(n/2; M, b) \le C_{\text{AtA}}(n; M, b) \le C_S(n; M, b)$. Hence, the thesis holds. □

8.4 Parallel AtA

Our algorithm for the $\mathbf{A}^T\mathbf{A}$ product, AtA, can be conveniently parallelized to work on both shared and distributed-memory systems. We will refer to our shared and distributed-memory algorithms for $\mathbf{A}^T\mathbf{A}$ as AtA-S and AtA-D, respectively. Our parallel implementations of AtA take advantage of the recursive nature of AtA to distribute tasks (and possibly data) to different processes in an efficient way. To do so, an initial phase that implements a scheduler covering the recursion tree of AtA is integrated in both parallel algorithms. In this way, we assign a task to each different parallel process, as we explain in Section 8.4.1. After this preliminary phase, each process knows which sub-problem it has to solve.

8.4.1 Preliminary Phase: Task Assignment

Usually, recursive algorithms are parallelized with a fork-join paradigm, according to their natural behaviour: at each recursive call, a new thread is created to accomplish that call. However, repeatedly creating and killing threads introduces a significant overhead, especially when it happens as a nested procedure. A parallelized `for` loop approach can usually improve this thread start-up overhead. For this reason, rather than addressing the problem by distributing recursive calls between newly created threads, we simulate the behaviour of a fork-join algorithm to determine, for each thread, on which sub-matrices it must work. This is particularly useful to generalize our approach to both shared memory and distributed settings.

Building the Task Tree

To conveniently distribute tasks among P parallel processes collaborating to compute $\mathbf{A}^T\mathbf{A}$, in the first phase of our algorithms, each process builds the recursion tree of a modified version of AtA, that we shall call AtANaive, and explores a part of it with a breadth-first search (BFS), see Figure 8.1. AtANaive considers classic recursive general matrix multiplication instead of Strassen, and can be easily implemented by modifying Algorithm 8.1 to call RecursiveGEMM instead of Strassen. RecursiveGEMM, summarized in Algorithm 8.2, is a recursive algorithm for the naive general matrix multiplication. The reasons behind this choice will be explained in Section 8.4.1. We define the *task tree*, denoted with $\mathcal{T}$, to be the sub-tree of the recursion tree of AtA, obtained by spanning the latter with a BFS, that is interrupted as soon as $\mathcal{T}$ counts P leaves, labeled from 0 to $P-1$. Both AtA-S and AtA-D implement the task tree, but with some differences concerning data and task division. In AtA-D, each p-th leaf corresponds to the task

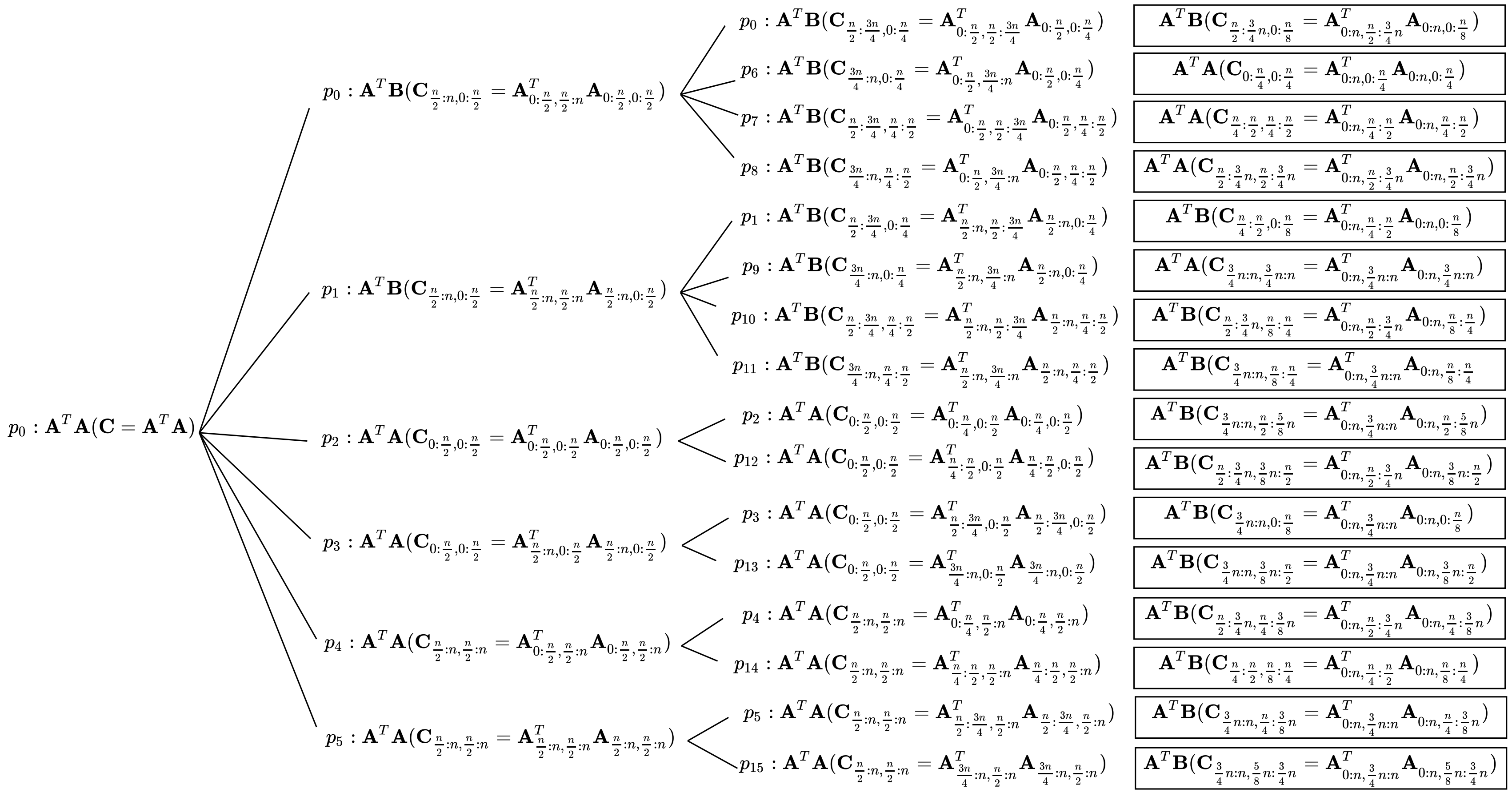

$p_0 : \mathbf{A}^T\mathbf{A}(\mathbf{C} = \mathbf{A}^T\mathbf{A})$
$p_0 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:n,0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}})$
$p_1 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:n,0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:n}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{2}})$
$p_2 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},0:\frac{n}{2}}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{2}})$
$p_3 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,0:\frac{n}{2}}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{2}})$
$p_4 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{2},\frac{n}{2}:n})$
$p_5 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:n}\mathbf{A}_{\frac{n}{2}:n,\frac{n}{2}:n})$
$p_0 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{4}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{4}})$
$p_6 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,0:\frac{n}{4}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{3n}{4}:n}\mathbf{A}_{0:\frac{n}{2},0:\frac{n}{4}})$
$p_7 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{0:\frac{n}{2},\frac{n}{4}:\frac{n}{2}})$
$p_8 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{2},\frac{3n}{4}:n}\mathbf{A}_{0:\frac{n}{2},\frac{n}{4}:\frac{n}{2}})$
$p_1 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{4}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{4}})$
$p_9 : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,0:\frac{n}{4}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{3n}{4}:n}\mathbf{A}_{\frac{n}{2}:n,0:\frac{n}{4}})$
$p_{10} : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3n}{4},\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{n}{2}:\frac{3n}{4}}\mathbf{A}_{\frac{n}{2}:n,\frac{n}{4}:\frac{n}{2}})$
$p_{11} : \mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3n}{4}:n,\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:n,\frac{3n}{4}:n}\mathbf{A}_{\frac{n}{2}:n,\frac{n}{4}:\frac{n}{2}})$
$p_2 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{0:\frac{n}{4},0:\frac{n}{2}}\mathbf{A}_{0:\frac{n}{4},0:\frac{n}{2}})$
$p_{12} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{4}:\frac{n}{2},0:\frac{n}{2}}\mathbf{A}_{\frac{n}{4}:\frac{n}{2},0:\frac{n}{2}})$
$p_3 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{2}}\mathbf{A}_{\frac{n}{2}:\frac{3n}{4},0:\frac{n}{2}})$
$p_{13} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{2},0:\frac{n}{2}} = \mathbf{A}^T_{\frac{3n}{4}:n,0:\frac{n}{2}}\mathbf{A}_{\frac{3n}{4}:n,0:\frac{n}{2}})$
$p_4 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{0:\frac{n}{4},\frac{n}{2}:n}\mathbf{A}_{0:\frac{n}{4},\frac{n}{2}:n})$
$p_{14} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{n}{4}:\frac{n}{2},\frac{n}{2}:n}\mathbf{A}_{\frac{n}{4}:\frac{n}{2},\frac{n}{2}:n})$
$p_5 : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{n}{2}:\frac{3n}{4},\frac{n}{2}:n}\mathbf{A}_{\frac{n}{2}:\frac{3n}{4},\frac{n}{2}:n})$
$p_{15} : \mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:n,\frac{n}{2}:n} = \mathbf{A}^T_{\frac{3n}{4}:n,\frac{n}{2}:n}\mathbf{A}_{\frac{3n}{4}:n,\frac{n}{2}:n})$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,0:\frac{n}{8}} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,0:\frac{n}{8}})}$
$\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{0:\frac{n}{4},0:\frac{n}{4}} = \mathbf{A}^T_{0:n,0:\frac{n}{4}}\mathbf{A}_{0:n,0:\frac{n}{4}})}$
$\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{4}:\frac{n}{2},\frac{n}{4}:\frac{n}{2}} = \mathbf{A}^T_{0:n,\frac{n}{4}:\frac{n}{2}}\mathbf{A}_{0:n,\frac{n}{4}:\frac{n}{2}})}$
$\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{2}:\frac{3}{4}n} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{2}:\frac{3}{4}n})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{4}:\frac{n}{2},0:\frac{n}{8}} = \mathbf{A}^T_{0:n,\frac{n}{4}:\frac{n}{2}}\mathbf{A}_{0:n,0:\frac{n}{8}})}$
$\boxed{\mathbf{A}^T\mathbf{A}(\mathbf{C}_{\frac{3}{4}n:n,\frac{3}{4}n:n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{3}{4}n:n})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{8}:\frac{n}{4}} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{4}})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{8}:\frac{n}{4}} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{4}}}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{2}:\frac{5}{8}n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{2}:\frac{5}{8}n})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{3}{8}n:\frac{n}{2}} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{3}{8}n:\frac{n}{2}})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,0:\frac{n}{8}} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,0:\frac{n}{8}})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{3}{8}n:\frac{n}{2}} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{3}{8}n:\frac{n}{2}})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{2}:\frac{3}{4}n,\frac{n}{4}:\frac{3}{8}n} = \mathbf{A}^T_{0:n,\frac{n}{2}:\frac{3}{4}n}\mathbf{A}_{0:n,\frac{n}{4}:\frac{3}{8}n})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{n}{4}:\frac{n}{2},\frac{n}{8}:\frac{n}{4}} = \mathbf{A}^T_{0:n,\frac{n}{4}:\frac{n}{2}}\mathbf{A}_{0:n,\frac{n}{8}:\frac{n}{4}})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{n}{4}:\frac{3}{8}n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{n}{4}:\frac{3}{8}n})}$
$\boxed{\mathbf{A}^T\mathbf{B}(\mathbf{C}_{\frac{3}{4}n:n,\frac{5}{8}n:\frac{3}{4}n} = \mathbf{A}^T_{0:n,\frac{3}{4}n:n}\mathbf{A}_{0:n,\frac{5}{8}n:\frac{3}{4}n})}$

Figure 8.1. A tree of 16 processes distributing $A \in \mathbb{R}^{n\times n}$. Boxed labels on the right-hand side are the leaf nodes of the tree generated by AtA-S, corresponding to computation tasks assigned to corresponding processes in the left-hand side leaf labels.

that process p has to fulfil, and contains directives on both the computational and communication activity that is due to the corresponding process. Specifically, a leaf task t provides the following information:

1. $t.computationType$: Which type of computation process p has to carry out. It can be either a $\mathbf{A}^T\mathbf{A}$ or a $\mathbf{A}^T\mathbf{B}$ multiplication;
2. $t.\mathbf{X}.offset$ and $t.\mathbf{X}.q$, with $\mathbf{X} \in \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, $q \in \{m, n\}$: The row and column offsets as well as the size of the sub-matrices of $\mathbf{A}$ and $\mathbf{C}$ process p has to work on;
3. $t.parent$: The parent process that sends sub-matrices of $\mathbf{A}$ to its children (during the distribution phase), and to which process p has to send the result of the task that was assigned to it or, if p is the parent, the information on its children' tasks (during result retrieval).

Inner nodes of $\mathcal{T}$ instead, represent tasks concerning data distribution and retrieval, possibly involving sums of sub-matrices of $\mathbf{C} = \mathbf{A}^T\mathbf{A}$, and consequent communication (point 3 of the previous list), and are executed by a subset of processes. In contrast, in AtA-S only leaf nodes of $\mathcal{T}$ correspond to a task, whereas inner nodes are ignored, as no communication is involved. For the same reason, leaf tasks only include information about what kind of computation the corresponding threads have to carry out and on the sub-matrices they have to work on (points 1 and 2 of the previous list).

Load Balancing

The task tree of AtA-D is created so that, at each level, given P available processes, $\alpha \cdot P$ processes compute a general $\mathbf{A}^T\mathbf{B}$ matrix multiplication; for the remaining $(1-\alpha) \cdot P$ processes, a task for a $\mathbf{A}^T\mathbf{A}$ multiplication is assigned to them. Here, $\alpha \in (0, 1)$ is a parameter for balancing the workload among distributed processes, as the computational complexity of a $\mathbf{A}^T\mathbf{A}$ product is lower than the one of $\mathbf{A}^T\mathbf{B}$. The task tree $\mathcal{T}$ is built by calling RecursiveGEMM (whose computational complexity is roughly twice the one of AtA, $T(n)$). Therefore the number of multiplications carried out in $\mathcal{T}$ to perform $\mathbf{A}^T\mathbf{B}$ is twice the one needed to compute $\mathbf{A}^T\mathbf{A}$. The load balancing parameter must be such that $4 \cdot T(n)/(1-\alpha)P = 2 \cdot 2T(n)/\alpha P$. In accordance, we set $\alpha = 1/2$. This task division is repeated recursively at each level, by progressively decreasing the number of available processes, P. The number of recursive parallel steps depends on P and α. In particular, for $\alpha = 0.5$, the number of parallel levels in the task tree, ℓ is given by the following expression:

$$\begin{aligned} &\ell(P=1) = 0, \quad \ell(2 \leq P \leq 6) = 1 \\ &\ell(P > 6) = 1 + k + \operatorname{sign}\left(\frac{P}{4} \mod 8^{\max\{k;1\}}\right), \end{aligned} \tag{8.5}$$

where $k = \max\left\{k \in \mathbb{N} : \frac{P/4}{8^k} \geq 1\right\}$ and $\operatorname{sign}(x)$ is the sign function, returning 0 for $x = 0$ and 1 for $x > 0$. Indeed, when AtA-D is called on P processes, $P/2$ of them are going to compute $\mathbf{C}_{2,1}$; out of them, $P/4$ processes compute $\mathbf{A}_{1,2}^T\mathbf{A}_{1,1}$, whereas the remaining $P/4$ are in charge for $\mathbf{A}_{2,2}^T\mathbf{A}_{2,1}$ (see Equation 8.2). These tasks are in turn distributed among 8 processes each, recursively (corresponding to the eight recursive calls of RecursiveGEMM). This splitting is repeated as long as it possible (i.e., until $P/4/8^k \geq 1$). If by doing so, all $P/4$ processes are used (i.e., $P/4$ is a multiple of 8^k, for some k), all processes work on equally sized matrices. Otherwise, some processes will further split their tasks to smaller matrices, resulting in an additional parallel level. We say that the last parallel level is *complete* when all leaves corresponding to $\mathbf{A}^T\mathbf{A}$ tasks are grouped in bunches of 6 siblings, and when all leaves corresponding to $\mathbf{A}^T\mathbf{B}$ tasks are grouped in bunches of 8 siblings.

The task tree for AtA-S is quite different. In order to avoid concurrent overlapping writes, input matrices are tiled in horizontal and vertical blocks, as depicted in Figure 8.2. This way, we ensure that each thread computes a different $\mathbf{C}_{i,j}$. With this new scheme, we make three recursive calls to AtA (instead of 6) and four recursive calls to FastStrassen (instead of 8). Therefore, the number of parallel levels in AtA-S, given P threads, is the following:

$$\begin{aligned}
&\ell(P=1)=0, \quad \ell(P=2,3)=1, \\
&\ell(P>3)=1+k+\operatorname{sign}\left(\frac{P}{2} \mod 4^{\max\{k;1\}}\right),
\end{aligned} \tag{8.6}$$

with $k=\max\left\{k\in\mathbb{N}:\frac{P/2}{4^k}\geq 1\right\}$. In Figure 8.1, we show an example of the task tree with 16 processes for AtA-D, and the leaf nodes of the task tree for AtA-S (boxed).

Naive Matrix Multiplication over Strassen

In our parallel algorithms, we do not rely on Strassen for general $\mathbf{A}^T\mathbf{B}$ matrix multiplication when building the recursion tree, that instead is created by simulating AtANaive. This is done with the goal of optimizing the resources of distributed architectures, as the naive general matrix-multiplication algorithm does not allocate the additional memory required by Strassen, resulting in a faster memory management. Furthermore, Strassen's algorithm would possibly cause a hardly manageable workload unbalance between processes implementing an $\mathbf{A}^T\mathbf{A}$ multiplication, and those that would be in charge of computing the intermediate matrix sums appearing in Strassen's algorithm. However, Strassen's algorithm can still be used at leaf-level computation.

8.4.2 Shared Memory AtA

AtA can be implemented with a shared-memory parallel paradigm on multi-core machines. We rely on OpenMP to efficiently distribute the workload between threads. Each thread simulates the recursion of AtANaive as described in Section 8.4.1. The workload is distributed so that each thread writes in a different memory location, hence there is no need of handling data collisions of any kind. Instead, the problem is divided in a fashion that makes it embarrassingly parallel. We call AtA-S our multi-threaded algorithm for $\mathbf{A}^T\mathbf{A}$.

Algorithm 8.3 AtA-S- Shared

```
1: procedure AtAS($\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{C} \in \mathbb{R}^{n\times n}$)
2:     Generate tree $\mathcal{T}$
3: end procedure
4: parfor each leaf-node $v$ of $\mathcal{T}$ do
5:     Get task $t$ from node $v$
6:     if $t.computationType = \mathbf{A}^T\mathbf{A}$ then
7:         AtA($\mathbf{A}_{t.\mathbf{A}.offset}$, $\mathbf{C}_{t.\mathbf{C}.offset}$, 1)
8:     else if $t.computationType = \mathbf{A}^T\mathbf{B}$ then
9:         FastStrassen($\mathbf{A}_{t.\mathbf{A}.offset}$, $\mathbf{A}_{t.\mathbf{B}.offset}$, $\mathbf{C}_{t.\mathbf{C}.offset}$, 1)
10:    end if
11: end parfor
```

AtA-S in Detail

Let us denote with P the number of available threads. Our algorithm for multi-threaded machines, AtA-S, can be divided into two phases. During the first phase, one task is assigned to each thread by simulating the recursion of AtANaive, as described in Section 8.4.1. In order to prevent memory collisions and to achieve embarrassing parallelism, tasks are organized so that each thread writes on a different and disjoint memory location. This is done by dividing the resulting matrix $\mathbf{C}$ into four blocks, as shown in Equation 8.2, whereas $\mathbf{A}$ is tiled vertically or horizontally, instead of in 2×2 blocks (see Figure 8.2). This procedure avoids concurrent writing management, it guarantees data and thread reuse and relies on the equality:

$$\mathbf{C}_{i,j} = \mathbf{A}_{i,1}\mathbf{B}_{1,j} + \mathbf{A}_{i,2}\mathbf{B}_{2,j} = \mathbf{A}_{i,*}\mathbf{B}_{*,j}, \tag{8.7}$$

for $i, j = 1, 2$. Such instruction and data assignment allows for a faster execution, since threads never need to synchronize.

During the second phase of AtA-S, each thread retrieves its task from the tree $\mathcal{T}$, specifying which routine (either AtA or FastStrassen) the corresponding thread must call, and on which sub-matrices of $\mathbf{A}$ and $\mathbf{C}$ it must operate. On multicore systems, this means that data reuse in both L1 and L2 cache is optimized, since each thread operates on the same data throughout its entire lifespan. Since the tasks correspond to disjoint sub-problems, at the end of the computation each thread only needs to synchronize with the others, then the algorithm stops. In Algorithm 8.3 we provide the pseudo-code of AtA-S.

Computational Complexity of AtA-S

We study the time complexity $T(n, P)$ of AtA-S to perform the multiplication $\mathbf{A}^T\mathbf{A}$ on an $n \times n$ matrix $\mathbf{A}$ and distributing the workload between P processes.

At first, the algorithm needs to generate the task tree and each process has to retrieve its task. These procedures have the same complexity as a BFS visit on a tree with P leaves, hence $O(P)$.

The time complexity of the second step corresponds to the one of the most expensive leaf task, which appears at the end of a path of RecursiveGEMM

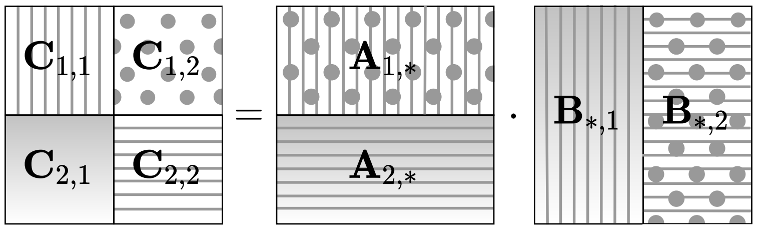


Figure 8.2. Multiplication with vertical/horizontal tiling.

calls. At level l, the size of the product matrix $\mathbf{C}$ is reduced to a block of size ${}^{n}\!/_{2^l} \times {}^{n}\!/_{2^l}$, resulting from a multiplication between ${}^{n}\!/_{2^l} \times n$ and $n \times {}^{n}\!/_{2^l}$ matrices. Thus, the total complexity is reduced by $4^{\ell(P)}$, being $\ell(P)$ the number of levels in the task tree. Hence the total complexity of the algorithm is:

$$T(n, P) = O(P) + O\left(\frac{1}{4^{\ell(P)}} n^{\log_2 7}\right). \tag{8.8}$$

Notice that $\ell(P)$ is a discrete, non-injective function. Hence, especially with few processes, the speed-up behaves like a step function. Despite this behaviour, $\ell(P) \approx \log_4 P$, meaning with large numbers of processes we achieve a theoretical linear speed-up.

8.4.3 Distributed Memory AtA

Modern computers are equipped with an ever-increasing number of cores inside CPU chips. However, when it comes to massive volumes of data, computationally intensive tasks such as matrix multiplication are simply prohibitive, even for the most recent 16- or 32-cores chipsets, and even with hyper-threading capabilities. Distributed parallelism plays a crucial role in this setting, as it allows to distribute the workload between multiple machines. In such an environment, providing fast distributed algorithms for operations in Linear Algebra, including $\mathbf{A}^T\mathbf{A}$ multiplication, is a key task to limit bottlenecks.

In this section, we describe a distributed algorithm for $\mathbf{A}^T\mathbf{A}$, that works for any matrix size and with arbitrarily many processes and cores. We shall refer to this algorithm as AtA-D. AtA-D follows a distribute-compute-retrieve paradigm, as initially the input matrix $\mathbf{A}$ is stored on the root process only, and distributed to other processes according to their tasks. Finally, the resulting matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ is retrieved back by the root process. We implement a parallel communication scheme to limit data transfer overhead.

AtA-D in Detail

Let P be the number of distributed processes. In AtA-D, each process p first builds the task tree $\mathcal{T}$ as described in Section 8.4.1. To understand in detail how $\mathcal{T}$ is used in AtA-D, we shall refer to the example of Figure 8.1. As we said, each node represents a task, but only tasks contained in leaf nodes correspond to an actual matrix multiplication. Inner nodes instead represent tasks assigned only to the parents of the nodes branching out of them, and they are necessary to retrieve and

Algorithm 8.4 AtA-D- Distributed

```
1:  procedure AtAD($\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{C} \in \mathbb{R}^{n \times n}$)
2:      Generate tree $\mathcal{T}$
3:      for each $v$ of $\mathcal{T}$ in the path from my leaf to the root do
4:          Get my task $t$ from node $v$
5:          if $v$ is a leaf then
6:              if $t.computationType = \mathbf{A}^T\mathbf{A}$ then
7:                  $\mathbf{C}_{t.\mathbf{C}.offset} = \mathbf{A}^T\mathbf{A}(\mathbf{A}_{t.\mathbf{A}.offset})$
8:              else if $t.computationType = \mathbf{A}^T\mathbf{B}$ then
9:                  $\mathbf{C}_{t.\mathbf{C}.offset} = \mathbf{A}^T\mathbf{B}(\mathbf{A}_{t.\mathbf{A}.offset}, \mathbf{A}_{t.\mathbf{B}.offset})$
10:             end if
11:         end if
12:         if $t.parent \neq$ my ID then
13:             Send $\mathbf{C}_{t.\mathbf{C}.offset}$ to $t.parent$
14:         else
15:             Receive $\mathbf{C}_{children.t.\mathbf{C}.offset}$ from my children
16:             Sum over the sub-matrices and store result in $\mathbf{C}$
17:         end if
18:     end for
19: end procedure
```

combine the portions of the result matrix scattered among different processes, and eventually to send them, level by level, up to the root process, p_0. In the example of Figure 8.1, $\mathcal{T}$ is the task tree for $P = 16$ processes on a square matrix. Leaf nodes are generated so that processes p_0, p_1 and $p_6 \ldots, p_{11}$ share the workload to compute $\mathbf{C}_{2,1}$. The remaining half of the processes is devoted to compute $\mathbf{C}_{1,1}$ and $\mathbf{C}_{2,2}$. If the number of distributed processes is not enough to make a complete level, as in this example, instead of calling multiple tasks on different tiles of the matrices, processes perform either an $\mathbf{A}^T\mathbf{A}$ or a $\mathbf{A}^T\mathbf{B}$ operation on vertically and horizontally tiled sub-matrices at the leaf-level. For instance, observe the first batch of sibling-leaves in Figure 8.1. To compute $\mathbf{C}_{n/2:n,0:n/2} = \mathbf{A}^T_{0:n/2,n/2:n}\mathbf{A}_{0:n/2,0:n/2}$, AtANaive would perform 8 recursive calls to $\mathbf{A}^T\mathbf{B}$; in AtA-D, each of these calls is served by one distributed process, if available. When this is not the case, as in the example that we are considering, processes p_0, p_6, p_7, p_8 divide $\mathbf{A}_{0:n/2,n/2:n}$ and $\mathbf{A}_{0:n/2,0:n/2}$ in vertical tiles so as to compute the related portions of $\mathbf{C}$ as depicted in Figure 8.2. When the computation is over, partial results are collected by the parents of each group of siblings (processes p_i, $i = 0, \ldots, 5$). This operation is iterated by traversing the tree up to its root, p_0, and allows for a convenient parallel communication reducing data transfer overhead. In order to optimize the communication and to reduce the exchanged data volume, we encode the sub-matrices resulting from $\mathbf{A}^T\mathbf{A}$ operations as packed lower triangular matrices. Nevertheless, the entire operation, once it returns to the root process, still produces a standard square matrix. In Algorithm 8.4, we provide the pseudocode of AtA-D. In line 13, if the process has to fulfill a $\mathbf{A}^T\mathbf{B}$ task, it sends to its parent the entire sub-matrix $\mathbf{C}_{t.\mathbf{C}.offset}$; otherwise, it only sends low($\mathbf{C}_{t.\mathbf{C}.offset}$). In lines 7 and 9, $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{B}$ may refer to AtA or `blas_?syrk`, and to FastStrassen or `blas_?gemm`, respectively. As we shall see in Section 8.5, the real benefit of using our implementation of AtA and FastStrassen arises on matrices with larger size, therefore they are favourable when handling larger volumes of data.

Computational and Communication Complexity of AtA-D

In contrast to parallel algorithms for distributed matrices, AtA-D does not include any communication between processes at computation time, as the input matrix is scattered among distributed processes so that they own the exact portions of $\mathbf{A}$ on which they have to operate.

Proposition 8.2. *The computational cost of AtA-D (Algorithm 8.4) on a matrix of size n and with using P processes, $C(n, P)$ is:*

$$C(n,P) = O\left(\left(n/2^{\ell(P)} \right)^2 \cdot n/2^{\ell(P)-1} \right),$$

if the load balancing parameter α is set to 0.5*.*

Proof. $C(n, P)$ depends on the number of recursive levels that can be layered with the available resources and on α. For $\alpha = 0.5$, the computational complexity of AtA-D is given by the time for computing $\mathbf{A}^T\mathbf{B}$ on matrices of size at most $n/2^{\ell(P)} \times n/2^{\ell(P)-1}$, that is $O\left(\left(n/2^{\ell(P)} \right)^2 \cdot n/2^{\ell(P)-1} \right)$, where $\ell(P)$ is the number of parallel levels defined in Equation 8.5. □

We express the communication cost for matrix distribution and result retrieval in terms of latency and bandwidth costs of a distributed algorithm, denoted with $L(n, P)$ and $BW(n, P)$, respectively, using the same definitions introduced in [19] and adopted also in [130]. Latency cost is the communicated-message count, whereas bandwidth is expressed in terms of communicated-word count. Messages and words counts are computed along the critical path of the distributed algorithm, as defined in [294].

Proposition 8.3. *The latency of AtA-D on a matrix of size $n \times n$ and with P processes is $L(n, P) = O(2[7 \cdot (\ell(P) - 1) + 5])$. Its bandwidth is $BW(n, P) \leq 6(n/2)^2 + \frac{n(n+2)}{2} + 7/6 n^2(1 - 1/4^{\ell(P)-2})$.*

Proof. In AtA-D, the critical path corresponds to the sequence of communication operations carried out by the root process p_0. After the first parallel level, p_0 works on a $\mathbf{A}^T\mathbf{B}$ task and shares its workload with 7 other processes at each parallel level. When the compute phase is over, at each level $l \in \{2, \ldots, \ell(P)\}$ process p_0 collects partial results from its (at most) seven children; at level $l = 1$, it retrieves the entire matrix $\mathbf{C} = \mathbf{A}^T\mathbf{A}$ by combining together the results of its five siblings. This operation is carried out both for data distribution and result collection. Hence, $L(n, P) = O(2[7 \cdot (\ell(P) - 1) + 5])$.
During the data distribution phase, message sizes (i.e., portions of input matrix $\mathbf{A}$) decrease when descending from the root down to the leaves of $\mathcal{T}$. In the first level, p_0 distributes two matrices of size $n/2 \times n/2$ to the other process that is in charge to carry out $\mathbf{A}^T\mathbf{B}$ tasks, and one sub-matrix of the same size to each of its four siblings

that have to compute $\mathbf{A}^T\mathbf{A}$. For each level $l \in \{2, \ldots, \ell(P)\}$, the root process sends matrices of size $n/2^l$ to at most 7 other processes. Hence, during the distribution phase, $BW(n, P)$ is $O(5\,(n/2)^2 + 7 \cdot \sum_{l=2}^{\ell(P)} (n/2^l)^2) = O(5(n/2)^2 + 7/12 n^2(1 - 1/4^{\ell(P)-2}))$. With similar considerations and taking into account the fact that processes sending symmetric portions of $\mathbf{C}$ only store its lower triangular part (low($\mathbf{C}$)), it holds that the bandwidth during the result retrieval phase amounts to $O((n/2)^2 + 4(n(n+2)/8) + 7 \cdot \sum_{l=2}^{\ell(P)} (n/2^l)^2) = O((n/2)^2 + n(n+2)/2 + 7/12 n^2(1 - 1/4^{\ell(P)-2})$. The thesis follows by summing together the two components. □

From this analysis, we see that computation has the prominent role in time complexity $T(n, P) = C(n, P) + L(n, P) + BW(n, P)$. This fact will be confirmed by our experimental results, presented in Section 8.5, where we see how increasing the matrix sizes provides an always increasing benefit in using the distributed algorithm, proving that communication cost $L(n, P) + BW(n, P)$ is absorbed by the computational cost, $C(n, P)$, for growing values of n.

8.5 Performance Evaluation

We evaluate the performance of our algorithms with an extensive set of experiments over multiple benchmarks. Our code is available at https://github.com/filthynobleman/AtA.

8.5.1 Experimental Setup

All tests reported in this section were run on TeraStat[1], a cluster of 12 compute nodes, each equipped with 2 sockets of Intel Xeon E5-2630v3 8 cores, 2.4 Ghz, 4 GB RAM per core.

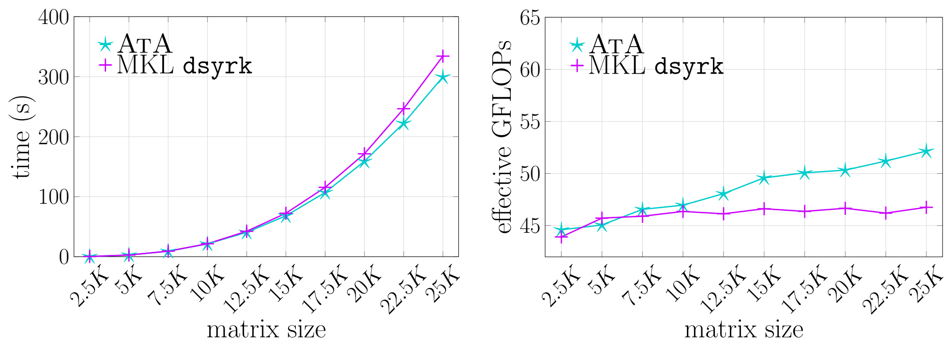
We test our algorithms and benchmark solutions on square and tall matrices, generated randomly. We carry out experiments in both single and double floating-point precision, to highlight the fact that our algorithm achieves good performance in both settings.

In the tests, we exploit the Intel Math Kernel Library (MKL) both by integrating BLAS routines for basic matrix operations, and for the validation of the proposed algorithms through performance comparisons with shared and distributed memory parallel benchmark solutions. MKL is a framework that includes routines and functions optimized for Intel and compatible processor-based computers, and provides C/C++ interfaces and the acceleration of libraries for Linear Algebra (including BLAS and ScaLapack) within several third-party math libraries. [278, 86].

8.5.2 Metrics

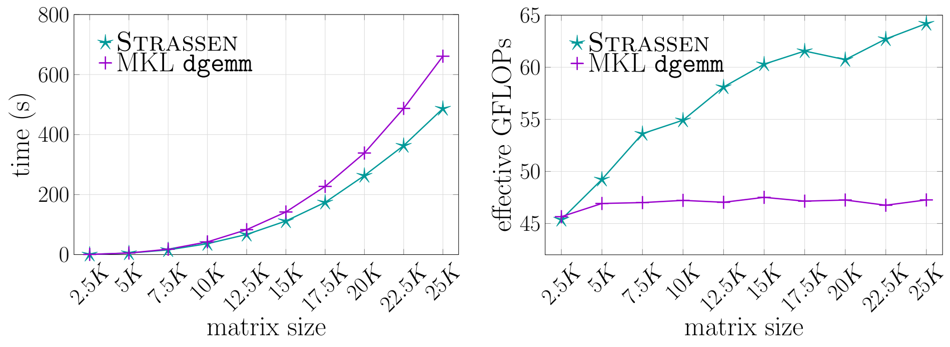
To compare the performance of our algorithms against benchmark methods, we use the average elapsed time in seconds and the effective GFLOPs. Effective GFLOPs is a measure for comparing classical and fast matrix-multiplication algorithms. For classical algorithms, which perform $2n^3$ floating point operations, Equation 8.9 gives

[1]https://www.dss.uniroma1.it/en/node/6554



(a) Elapsed time. **(b)** Effective GFLOPs.

Figure 8.3. ATA vs Intel MKL `dsyrk`



(a) Elapsed time. **(b)** Effective GFLOPs.

Figure 8.4. FastStrassen vs Intel MKL `dgemm`

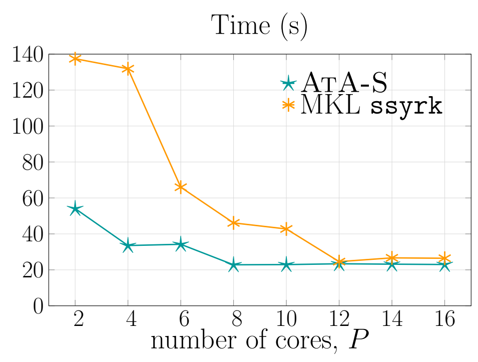
the actual GFLOPs; for fast matrix-multiplication algorithms, it gives the performance relative to classical algorithms, but does not accurately represent the number of floating point operations performed [67]. For fair comparisons, we calculate the metrics as:

$$\text{effective GFLOPs} = \frac{rn^3}{\text{execution time in seconds} \cdot 10^9} \tag{8.9}$$

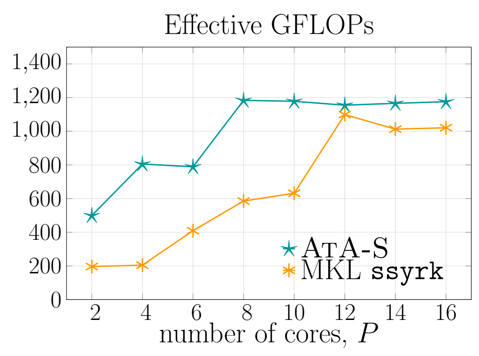
where $r = 1$ when we test algorithms specifically built for the $\mathbf{A}^T\mathbf{A}$ product, whereas $r = 2$ when algorithms for the general matrix multiplication are tested.

8.5.3 Sequential

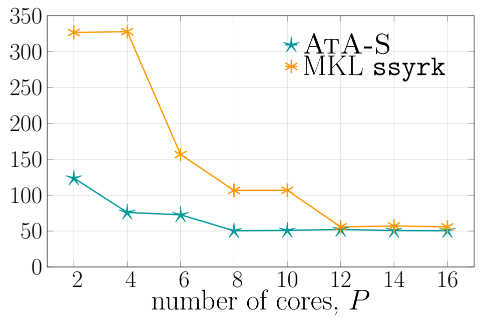
Figures 8.3 and 8.4 show the execution time and effective GFLOPs of the sequential AtA and FastStrassen routines, respectively. Their performance is compared to the Intel MKL counterparts: `dsyrk` and `dgemm`. The experiments are carried out on matrices of growing matrix size (from $2.5 \cdot 10^3$ to $2.5 \cdot 10^4$), and run on a single Intel core. The time difference between our solutions and the ones implemented by Intel MKL grows with the matrix size, reflecting the lower computational cost of our approach. Figure 8.4 proves how Strassen's algorithm benefits from the pre-memory-allocation strategy described in Section 8.3.3.



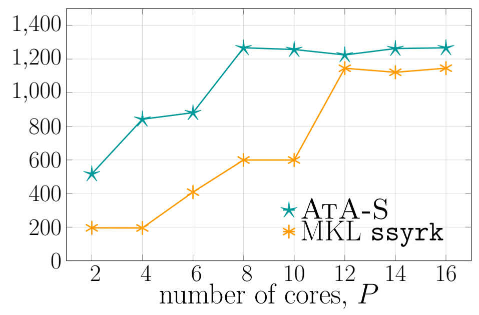
(a) Elapsed time, $\mathbf{A} \in \mathbb{R}^{30K \times 30K}$.



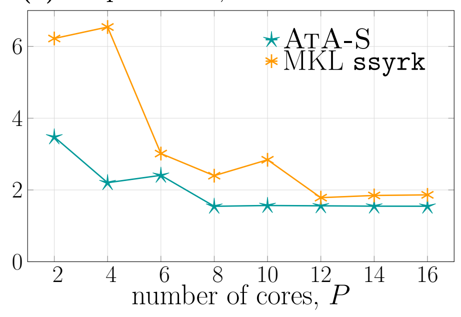
(b) EGs, $\mathbf{A} \in \mathbb{R}^{30K \times 30K}$.



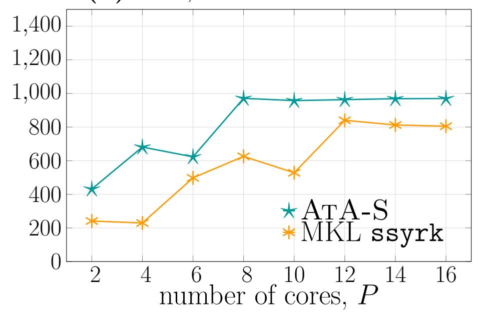
(c) Elapsed time, $\mathbf{A} \in \mathbb{R}^{40K \times 40K}$.



(d) EGs, $\mathbf{A} \in \mathbb{R}^{40K \times 40K}$.



(e) Elapsed time, $\mathbf{A} \in \mathbb{R}^{60K \times 5K}$.



(f) EGs, $\mathbf{A} \in \mathbb{R}^{60K \times 5K}$.

Figure 8.5. Experimental results of AtA-S and Intel MKL dsyrk in terms of elapsed time in seconds (left column) and effective GFLOPs (right column), varying the number of available cores P on fixed matrix sizes with a 16 threads configuration.

8.5.4 Shared Memory

For evaluating the shared memory parallel implementation of the $\mathbf{A}^T\mathbf{A}$ product, AtA-S, we compare it against the Intel MKL implementation of the BLAS routine `ssyrk`, for single precision symmetric rank-K update. For both methods, we always use a 16 thread setup, and we analyse the execution time and the effective GFLOPs (Equation 8.9 with $r = 1$) while varying the number of available cores. In light of the sequential experiments shown in Figures 8.3 and 8.4, we compare AtA-S and MKL `ssyrk` on larger matrices, where tests highlight more interesting results. In particular, we run experiments on square matrices of size $3 \cdot 10^4 \times 3 \cdot 10^4$, $4 \cdot 10^4 \times 4 \cdot 10^4$ and on tall matrices of size $6 \cdot 10^4 \times 5 \cdot 10^3$. Figure 8.5 summarizes our results. As anticipated by the study of the computational complexity, the execution time is reduced by 1/4 at each complete parallel level. Figures 8.5a, 8.5c and 8.5e show how our algorithm can compete with the MKL implementation when the core availability is large, and that significantly outperforms the Intel implementation in the $P \leq 10$ cores setup. Furthermore, we show in Figures 8.5b, 8.5d and 8.5f that AtA-S is capable not only of accomplishing a large amount of floating point operations per second, but also that its performance growth rate is consistent with the step-wise behaviour of the time complexity studied in Section 8.4.2. This justifies sporadic thinnings in performance gap between the two methods.

From Figure 8.5, we can observe that the performance of both methods stall when more than 8 cores are used. Indeed, multi-threaded MKL automatically chooses the optimal number of threads (in our architecture, this corresponds to 16 threads). For a fair comparison, we use the same setup in AtA-S. Performance scales with the number of available cores, but, when hyper-threading is enabled, 8 cores are enough to reach the 16-thread plateau. Therefore, performance cannot increase significantly for $P > 8$.

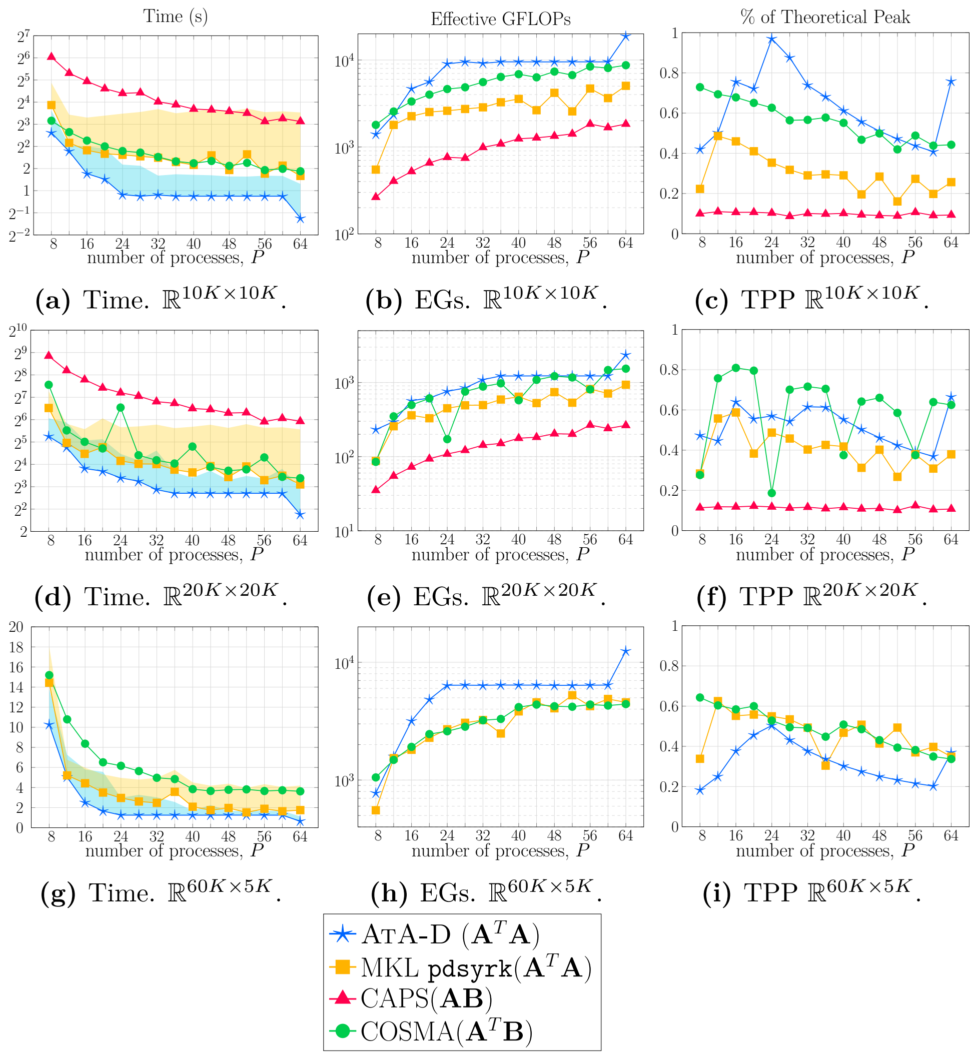
8.5.5 Distributed Memory

To complete our performance evaluation, we also compare our implementation for distributed architectures of AtA, AtA-D, with fast distributed algorithms for matrix multiplication. We recall that AtA-D differs from standard methods for distributed matrix multiplication, as it does not perform computations on distributed matrices. Instead, in AtA-D the input matrix $\mathbf{A}$ is only stored by the root process, p_0, that first distributes it among other processes cooperating to perform the $\mathbf{A}^T\mathbf{A}$ product, and then collects the partial result of each process to combine them. This approach makes our method unsuitable for distributed chains of operations, since for every operation, the matrix must be repeatedly scattered and gathered back, thus introducing communication overhead, but our results highlight that it is an efficient alternative for distributing single $\mathbf{A}^T\mathbf{A}$ operations. At the current state-of-the-art, there are a variety of methods for multiplying distributed matrices, but in the most recent literature there are three algorithms which stand out:

1. Intel MKL ScaLapack `p?syrk`: the Intel Math Kernel libraries (MKL) provide optimized implementation of ScaLapack routines for high-performance dense Linear Algebra operations on distributed clusters. In ScaLapack, distributed processes are organized in 2D grids of size $m_P \times n_P = P$. For each value of P, we set optimal m_P and n_P by calling `MPI_Dims_create`. We analyse the execution time required to perform the $\mathbf{A}^T\mathbf{A}$ matrix multiplication by the built-in ScaLapack function `pdsyrk`, and the time to retrieve the result of the operation.

2. CAPS[2]: the Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication [18] is a distributed algorithm for general square matrix multiplications $\mathbf{AB}$. Soon after CAPS, the same authors proposed CARMA [67], that also handles rectangular matrices. Nevertheless, it was not possible

[2]`https://github.com/lipshitz/CAPS/`



(a) Time. $\mathbb{R}^{10K\times10K}$. **(b)** EGs. $\mathbb{R}^{10K\times10K}$. **(c)** TPP $\mathbb{R}^{10K\times10K}$.

(d) Time. $\mathbb{R}^{20K\times20K}$. **(e)** EGs. $\mathbb{R}^{20K\times20K}$. **(f)** TPP $\mathbb{R}^{20K\times20K}$.

(g) Time. $\mathbb{R}^{60K\times5K}$. **(h)** EGs. $\mathbb{R}^{60K\times5K}$. **(i)** TPP $\mathbb{R}^{60K\times5K}$.

Figure 8.6. Experimental results of ATA-D, Intel MKL `pdsyrk`, CAPS and COSMA in terms of elapsed time in seconds (left column), effective GFLOPs (central column) and % of theoretical peak (right column) varying the number of distributed processes P on fixed matrix sizes.

to test this method as it relies on Cilk Plus, a tool for parallel computing now marked as deprecated[3].

3. COSMA[4]: differently from CAPS, this communication-optimal algorithm for general matrix multiplication does not rely on Strassen's algorithm, instead, it uses red-blue pebble game to precisely model the matrix-multiplication dependencies. In [130], the authors show that COSMA outperforms all previously proposed frameworks for general matrix multiplication. It also works for multiplication on transposed matrices, and therefore we test it to perform $\mathbf{A}^T\mathbf{B}$ products.

[3]https://www.cilkplus.org/, Last accessed 07-01-2021
[4]`https://github.com/eth-cscs/COSMA`

To simulate massively distributed architectures, in our experiments, we reserve only one core per distributed process. As a consequence, each process has small memory availability (4GB RAM/core). The results of our experiments for the distributed-memory solution are shown in Figure 8.6. In Figures 8.6a, 8.6d and 8.6g, marked lines represent the compute time of all considered methods. The shaded areas above the curves describing AtA-D and `pdsyrk` represent the additional time required for communication, i.e., for retrieving the resulting matrix to the root process. We consider two groups of square matrices, having size 10^4 and $2 \cdot 10^4$ (Figures 8.6a, 8.6b, 8.6c and 8.6d, 8.6e, 8.6f respectively), and one set of tall matrices of size $6 \cdot 10^4 \times 5 \cdot 10^3$ (Figures 8.6g, 8.6h, 8.6i). Because CAPS does not operate on rectangular matrices, we could not test it on the latter set of experimental configurations. As we can observe from Figure 8.6, scalability of AtA-D is nonlinear and it rather follows an almost-stepwise trend with respect to P. This is a consequence of Equation 8.5, that shows how some values of P allow for a more effective and balanced workload between processes. This is evident for small values of P (when a greater availability of processes weighs significantly on the workload of each process), as well as for $P = 64$. Despite the different nature of the parallelism implemented in AtA-D with respect to the benchmark methods analysed in this section, our experiments corroborate the efficiency of the task distribution implemented in AtA-D. In Figures 8.6c, 8.6f and 8.6i we show the percentage of theoretical peak performance (TPP) for all tested algorithms. We compute it as the effective GFLOPs over the theoretical performance peak of the nodes of our cluster. For all tested methods, the effective GFLOPs are computed as in Equation 8.9, (as those reported in Figures 8.6b, 8.6e, 8.6h), except for AtA-D, for which we now use the complexity of AtA (Equation 8.3). Regarding the percentage of theoretical peak, we can see how our algorithm has comparable behaviour with respect to the other solutions on square matrices, but it performs worse on the rectangular case. The high performance of our method relies on careful ordering and placement of highly optimized BLAS routines. However, especially when working on tall matrices, we need to perform several calls to BLAS Level 1 routines (i.e., to compute intermediate sums both in AtA and FastStrassen) and system calls (i.e., memory copies) on very short rows. This leads to more memory accesses and poorer vectorization capability than dealing with the same amount of data, distributed in fewer, longer rows, would entail. As a consequence, the overall performance with respect to the theoretical peak is worsened. Furthermore, in Figures 8.6c, 8.6f and 8.6i we see that AtA-D loses a bit of efficiency on larger matrices. This is due to the fact that each process calling the FastStrassen routine needs to allocate $3/2n^2$ space of memory, hence memory handling slows down the entire process. In addition, also in the last column of Figure 8.6, we can observe performance peaks after slow degradations as we did in the first two. We stress that this is a consequence of the fact that for some values of P, workload among processes is distributed more efficiently (see Equation 8.5). As a matter of fact, the computational complexity of AtA-D decreases exponentially at each level, but the number of levels increases logarithmically with the number of processes, P. Therefore for numbers of processes that result in the same number of parallel levels, the improvement is less appreciable.

Finally, in order to study the scalability of AtA-D with respect to AtA-S, and

n	SM (16 cores)	DM (96 cores)	Speed-up
30K	45.16 s	21.24 s	2.13
40K	106.19 s	43.96 s	2.42
50K	221.63 s	81.77 s	2.71
60K	863.32 s	129.08 s	6.69

Table 8.1. Shared memory (SM) implementation of $\mathbf{A}^T\mathbf{A}$ compared with the distributed memory (DM) on large square $n \times n$ matrices.

to validate the possibility of integrating the two methods, we compare AtA-S and AtA-D on very large matrices of increasing size and report results in Table 8.1. AtA-S works on 16 cores with 16 threads, whereas AtA-D works on 6 distributed nodes, each equipped with 16 cores, for a total of 96 cores. Each node executes a distributed process calling either AtA-S for $\mathbf{A}^T\mathbf{A}$-type products, or multi-threaded MKL `dgemm` for $\mathbf{A}^T\mathbf{B}$-type multiplications. The times reported in Table 8.1 for AtA-D also include communication time (for distributing data and collecting results). Speed-up is computed as T_{SM}/T_{DM}, where T_{SM} and T_{DM} are the execution times of the shared and the distributed-memory algorithms, respectively. In accordance with our computational and communication cost analysis (Section 8.4.3), the speed-up of AtA-D over AtA-S increases when the size of the input matrix increases, as the computation cost overwhelms the communication overhead. Furthermore, the shared-memory implementation suffers when considering larger matrices, since frequent memory access slows down execution (two $60K \times 60K$ matrices require 57 GB out of the 64 GB available on the test machine), consequently decreasing performance. This is highlighted by the results of Table 8.1, where we can observe that $60K \times 60K$ matrices require high computation time, dominated by the time for memory management.

8.6 Conclusions

We propose AtA, an algorithm for the $\mathbf{A}^T\mathbf{A}$ product, that reduces the computational complexity of commonly employed algorithms, and that is conveniently implementable in practice on matrices defined on arbitrary domains and of any size and aspect ratio. The computational cost of AtA benefits from the fast matrix multiplication introduced by Strassen's algorithm, and is cache-oblivious. We show that AtA can be efficiently implemented in shared and distributed memory environments. In the shared memory implementation of AtA, tasks are assigned to parallel threads so that they work in perfect parallelism. Our theoretical analysis is supported by experiments that prove the excellent performance of our implementations in comparison with state-of-the-art counterparts.

Conclusions

In this thesis, we covered a variety of computer graphics applications, spanning from procedural texturing and curve design to shape matching and simulations. The common ground for all these topics is addressing multiple different tasks from a geometric perspective and proposing efficient solutions based on the geometry processing background.

We have proposed a new framework, extending the well-known functional maps framework, that allows for addressing the reconstruction and transferring of signals between shapes at a reduced computational complexity, and we have shown that by adopting machine learning techniques, we can compute non-rigid set operations between partial manifolds by only knowing the Laplacian eigenvalues. These two novel solutions convey that spectral geometry techniques can be made to scale and that steps can be taken to produce more efficient algorithms.

In a procedural texturing setting, we dealt with solutions based on existing backgrounds and explored new directions. We proved that four-dimensional algebraic spaces could be exploited to generate fractal noises on surfaces in real time and that simulations on non-Euclidean domains produce complex dynamic patterns that can be used for animating textures. In the same spirit, we provided a solution for simulating water diffusion in graph-like structures, showing that it can fit complex pipelines to achieve visually convincing animations of wilting plants.

Furthermore, we explored walks, isotropic expansions, and triangulations under generalized Riemannian metrics, and we applied these concepts to design algorithms for fast computation of geodesically uniform remeshing and non-Euclidean curve reconstruction.

Finally, we explored the idea of improving the mathematical toolbox at the core of many differential geometry techniques, realizing an efficient and easily parallelizable algorithm for specific matrix multiplications.

In conclusion, this thesis spans different applications in the geometry processing field, providing fast, scalable, and accurate solutions to a variety of tasks, bringing them to a suitable state for dealing with industry-standard volumes of data.

Bibliography

[1] Abdel-Basset, M., Chang, V., and Mohamed, R. HSMA_WOA: A hybrid novel Slime mould algorithm with whale optimization algorithm for tackling the image segmentation problem of chest X-ray images. *Applied Soft Computing 95* (2020), 106642.

[2] Abdel-Basset, M., Mohamed, R., Chakrabortty, R. K., Ryan, M. J., and Mirjalili, S. An efficient binary slime mould algorithm integrated with a novel attacking-feeding strategy for feature selection. *Computers & Industrial Engineering 153* (2021), 107078.

[3] Adamatzky, A. Slime mold solves maze in one pass, assisted by gradient of chemo-attractants. *IEEE transactions on nanobioscience 11*, 2 (2012), 131–134.

[4] Adobe Inc. Adobe illustrator.

[5] Aflalo, Y., Brezis, H., and Kimmel, R. On the optimality of shape and data representation in the spectral domain. *SIAM Journal on Imaging Sciences 8*, 2 (2015), 1141–1160.

[6] Amenta, N., Bern, M., and Eppstein, D. The crust and the beta-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing 60* (01 1998), 125–135.

[7] Amenta, N., Choi, S., and Kolluri, R. K. The power crust. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications* (New York, NY, USA, 2001), SMA '01, Association for Computing Machinery, p. 249–266.

[8] Anastacio, F., Sousa, M. C., Samavati, F., and Jorge, J. A. Modeling plant structures using concept sketches. NPAR '06, ACM, p. 105–113.

[9] Anguelov, D., Srinivasan, P., Koller, D., Thrun, S., Rodgers, J., and Davis, J. Scape: shape completion and animation of people. In *ACM SIGGRAPH 2005 Papers*. 2005, pp. 408–416.

[10] Aono, M., and Kunii, T. Botanical tree image generation. *IEEE Comput. Graph. Appl. 4(5)* (1984), 10–34.

[11] Argudo, O., Chica, A., and Andujar, C. Single-picture reconstruction and rendering of trees for plausible vegetation synthesis. *Comput. Graph. 57*, C (2016), 55–67.

[12] Arrigoni, V., Maggioli, F., Massini, A., and Rodolà, E. Efficiently parallelizable strassen-based multiplication of a matrix by its transpose. In *50th International Conference on Parallel Processing* (2021), pp. 1–12.

[13] Attaiki, S., Pai, G., and Ovsjanikov, M. Dpfm: Deep partial functional maps. In *2021 International Conference on 3D Vision (3DV)* (2021), IEEE, pp. 175–185.

[14] Aubry, M., Schlickewei, U., and Cremers, D. The wave kernel signature: A quantum mechanical approach to shape analysis. In *Proc. ICCV Workshops* (2011), IEEE, pp. 1626–1633.

[15] Aurenhammer, F., Klein, R., and Lee, D. *Voronoi Diagrams And Delaunay Triangulations*. World Scientific Publishing Company, 2013.

[16] Aygün, M., Lähner, Z., and Cremers, D. Unsupervised dense shape correspondence using heat kernels. In *2020 International Conference on 3D Vision (3DV)* (2020), IEEE, pp. 573–582.

[17] Azencot, O., and Lai, R. Shape analysis via functional map construction and bases pursuit. *arXiv preprint arXiv:1909.13200* (2019).

[18] Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., and Schwartz, O. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proc. 24th ACM Symp. Parallelism in Algorithms and Architectures (SPAA)* (2012), pp. 193–204.

[19] Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM) 59*, 6 (2013), 1–23.

[20] Banerjee, S., Banerjee, D. P., and Mukhopadhyay, A. Implications of global warming on changing trends in crop productivity - a review. *International Letters of Natural Sciences 11* (02 2014), 16–29.

[21] Barth, R., IJsselmuiden, J., Hemming, J., and Henten, E. V. Data synthesis methods for semantic segmentation in agriculture: A capsicum annuum dataset. *Computers and Electronics in Agriculture 144* (2018), 284–296.

[22] Belcher, J., and Koleci, C. Using animated textures to visualize electromagnetic fields and energy flow. *arXiv preprint arXiv:0802.4034* (2008).

[23] Bender, J., Müller, M., and Macklin, M. A survey on position based dynamics, 2017. In *Proceedings of the European Association for Computer Graphics: Tutorials* (Goslar, DEU, 2017), EG '17, Eurographics Association.

[24] Benes, B., Andrysco, N., and Šť'ava, O. Interactive modeling of virtual ecosystems. In *Proceedings of the Fifth Eurographics Conference on Natural Phenomena* (Goslar, DEU, 2009), NPH'09, Eurographics Association, p. 9–16.

[25] Benson, A., and Ballard, G. A framework for practical parallel fast matrix multiplication. In *Proc. 20th ACM SIGPLAN PPoPP* (2015), pp. 42–53.

[26] Berger, M., Tagliasacchi, A., Seversky, L. M., Alliez, P., Guennebaud, G., Levine, J. A., Sharf, A., and Silva, C. T. A survey of surface reconstruction from point clouds. *Computer Graphics Forum 36*, 1 (2017), 301–329.

[27] Bern, M., and Eppstein, D. Mesh generation and optimal triangulation. In *Computing in Euclidean geometry.* World Scientific, 1995, pp. 47–123.

[28] Bern, M. W., Eppstein, D., and Gilbert, J. R. Provably good mesh generation. *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science* (1990), 231–241 vol.1.

[29] Biewald, L. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[30] Bogo, F., Romero, J., Loper, M., and Black, M. J. FAUST: Dataset and evaluation for 3D mesh registration. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (Piscataway, NJ, USA, June 2014), IEEE.

[31] Bogo, F., Romero, J., Pons-Moll, G., and Black, M. J. Dynamic faust: Registering human bodies in motion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (July 2017).

[32] Botsch, M., and Kobbelt, L. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (New York, NY, USA, 2004), SGP '04, Association for Computing Machinery, p. 185–192.

[33] Botsch, M., Kobbelt, L., Pauly, M., Alliez, P., and Lévy, B. *Polygon mesh processing.* CRC press, 2010.

[34] Bowyer, A. Computing Dirichlet tessellations*. *The Computer Journal 24*, 2 (01 1981), 162–166.

[35] Bradley, D., Nowrouzezahrai, D., and Beardsley, P. Image-based reconstruction and synthesis of dense foliage. *TOG 32*, 4 (2013), 74:1–74:10.

[36] Brent, R. P. Algorithms for matrix multiplication. Tech. Rep. TR-CS-70-157, Stanford University, 1970.

[37] Brent, R. P. Error analysis of algorithms for matrix multiplication and triangular decomposition using winograd's identity. *Numerische Mathematik 16* (1970), 145–156.

[38] Bronstein, A., Bronstein, M., and Kimmel, R. *Numerical Geometry of Non-Rigid Shapes.* Springer, New York, NY, 2008.

[39] Bronstein, A. M., Bronstein, M. M., Guibas, L. J., and Ovsjanikov, M. Shape google: Geometric words and expressions for invariant shape retrieval. *ACM Transactions on Graphics (TOG) 30*, 1 (2011), 1–20.

[40] Brown, B. J., and Rusinkiewicz, S. Non-rigid range-scan alignment using thin-plate splines. In *Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004.* (2004), IEEE, pp. 759–765.

[41] Brown, B. J., and Rusinkiewicz, S. Global non-rigid alignment of 3-d scans. In *ACM SIGGRAPH 2007 papers.* 2007, pp. 21–es.

[42] Burgin, M., and Adamatzky, A. Structural machines and slime mould computation. *International Journal of General Systems 46*, 3 (2017), 201–224.

[43] Cacciola, F., Rouxel-Labbé, M., Şenbaşlar, B., and Komaromy, J. Triangulated surface mesh simplification. In *CGAL User and Reference Manual*, 5.5.2 ed. CGAL Editorial Board, 2023.

[44] Cellier, F. E., and Kofman, E. *Continuous system simulation.* Springer Science & Business Media, 2006.

[45] Çengel, Y., and Cimbala, J. *Fluid Mechanics: Fundamentals and Applications.* McGraw-Hill Education, 2018.

[46] Chang, W., and Zwicker, M. Range scan registration using reduced deformable models. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 447–456.

[47] Chang, W., and Zwicker, M. Global registration of dynamic range scans for articulated model reconstruction. *ACM Transactions on Graphics (TOG) 30*, 3 (2011), 1–15.

[48] Charara, A., Keyes, D., and Ltaief, H. Batched triangular dense linear algebra kernels for very small matrix sizes on gpus. *ACM Trans Math. Softw. (TOMS) 45*, 2 (2019), 1–28.

[49] Charara, A., Ltaief, H., and Keyes, D. Redesigning triangular dense matrix computations on gpus. In *Euro-Par* (2016), Springer, pp. 477–489.

[50] Chavel, I. *Riemannian geometry: a modern introduction*, vol. 98. Cambridge university press, 2006.

[51] Chavel, I., Randol, B., and Dodziuk, J. *Eigenvalues in Riemannian Geometry.* ISSN. Elsevier Science, 1984.

[52] Chew, L. P. Guaranteed-quality mesh generation for curved surfaces. In *SCG '93* (1993).

[53] Chopard, B., and Droz, M. *Cellular automata*, vol. 1. Springer, 1998.

[54] Choukroun, Y., Shtern, A., Bronstein, A., and Kimmel, R. Hamiltonian operator for spectral shape analysis. *TVCG 26* (2017).

[55] Coifman, R. R., and Maggioni, M. Diffusion wavelets. *Applied and Computational Harmonic Analysis 21*, 1 (2006), 53–94.

[56] Community, B. O. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[57] Conway, J., and Guy, R. *The Book of Numbers*. Copernicus Series. Springer New York, 1998.

[58] Cook, R. L., and DeRose, T. Wavelet noise. *ACM Transactions on Graphics (TOG) 24*, 3 (2005), 803–811.

[59] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proc. 19th ACM Symp. Theory of Computing (STOC)* (1987), pp. 1–6.

[60] Cosmo, L., Panine, M., Rampini, A., Ovsjanikov, M., Bronstein, M. M., and Rodolà, E. Isospectralization, or how to hear shape, style, and correspondence. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), pp. 7529–7538.

[61] Cosmo, L., Rodolà, E., Masci, J., Torsello, A., and Bronstein, M. Matching deformable objects in clutter. In *Proc. 3D Vision (3DV)* (2016), pp. 1–10.

[62] Crane, K., Weischedel, C., and Wardetzky, M. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph. 32*, 5 (oct 2013).

[63] D'Alberto, P., and Nicolau, A. Adaptive strassen's matrix multiplication. In *Proc. 21st Int. Conf. on Supercomputing* (2007), ACM, pp. 284–292.

[64] Davenport, C. *A Commutative Hypercomplex Calculus with Applications to Special Relativity*. Eigenverl., 1991.

[65] Davenport, C. *A Commutative Hypercomplex Algebra with Associated Function Theory*. Birkhauser Boston Inc., USA, 1996, p. 213–227.

[66] DeCoro, C., and Tatarchuk, N. Real-time mesh simplification using the gpu. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), I3D '07, Association for Computing Machinery, p. 161–166.

[67] Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., O., O. S., and Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *IEEE 27th Int. Symp. on Parallel and Distributed Processing* (2013), IEEE, pp. 261–272.

[68] Desprez, F., and Suter, F. Impact of mixed-parallelism on parallel implementations of the strassen and winograd matrix multiplication algorithms. *Concurr. Comput.: Pract. Exper. 16*, 8 (2004), 771–797.

[69] Deul, C., Kugelstadt, T., Weiler, M., and Bender, J. Direct position-based solver for stiff rods. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 313–324.

[70] Deussen, O., and Lintermann, B. *Digital design of nature: computer generated plants and organics.* Springer Science & Business Media, 2005.

[71] Dey, T., and Wenger, R. Fast reconstruction of curves with sharp corners. *Int. J. Comput. Geometry Appl. 12* (10 2002), 353–400.

[72] Dey, T. K., and Kumar, P. A simple provable algorithm for curve reconstruction. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 1999), SODA '99, Society for Industrial and Applied Mathematics, p. 893–894.

[73] do Carmo, M. *Differential Geometry of Curves and Surfaces: Revised and Updated Second Edition.* Dover Books on Mathematics. Dover Publications, 2016.

[74] Du, Q., Faber, V., and Gunzburger, M. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Review 41*, 4 (1999), 637–676.

[75] Dumas, J., Pernet, C., and Sedoglavic, A. On fast multiplication of a matrix by its transpose. In *Proc. 45th Int. Symp. Symbolic and Algebraic Computation* (New York, NY, USA, 2020), ISSAC '20, Association for Computing Machinery, p. 162–169.

[76] Dyke, R. M., Lai, Y.-K., Rosin, P. L., Zappalà, S., Dykes, S., Guo, D., Li, K., Marin, R., Melzi, S., and Yang, J. SHREC'20: Shape correspondence with non-isometric deformations. *Computers & Graphics 92* (2020), 28–43.

[77] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. *Texturing & modeling: a procedural approach.* Morgan Kaufmann, 2003.

[78] Eisenberger, M., Lahner, Z., and Cremers, D. Smooth shells: Multi-scale shape registration with functional maps. In *Proc. CVPR* (June 2020).

[79] Ekinci, S., Izci, D., Zeynelgil, H. L., and Orenc, S. An application of slime mould algorithm for optimizing parameters of power system stabilizer. In *2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)* (2020), pp. 1–5.

[80] Elek, O., Burchett, J. N., Prochaska, J. X., and Forbes, A. G. Polyphorm: structural analysis of cosmological datasets via interactive physarum polycephalum visualization. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2020), 806–816.

[81] Eliahu, D., Spillinger, O., Fox, A., and Demmel, J. Frpa: A framework for recursive parallel algorithms. Tech. Rep. UCB/EECS-2015-28, EECS Department, University of California, Berkeley, 2015.

[82] Elmroth, E., Gustavson, F., Jonsson, I., and Kågström, B. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM review 46*, 1 (2004), 3–45.

[83] Ezuz, D., and Ben-Chen, M. Deblurring and denoising of maps between shapes. *Computer Graphics Forum 36*, 5 (2017), 165–174.

[84] Fanni, F. A., Pellacini, F., Scateni, R., and Giachetti, A. Pavel: Decorative patterns with packed volumetric elements. *ACM Trans. Graph. 41*, 2 (jan 2022).

[85] Filoche, M., and Mayboroda, S. Strong localization induced by one clamped point in thin plate vibrations. *Physical review letters 103*, 25 (2009), 254301.

[86] for Intel® Math Kernel Library C, D. R.

[87] Frey, P. J., and Borouchaki, H. Surface mesh quality evaluation. *International journal for numerical methods in engineering 45*, 1 (1999), 101–118.

[88] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. In *40th Symp. Foundations of Computer Science (FOCS)* (1999), IEEE, pp. 285–297.

[89] Gall, F. L. Powers of tensors and fast matrix multiplication. In *Proc. 39th Int. Symp. Symbolic and Algebraic Computation* (2014), pp. 296–303.

[90] Garland, M., and Heckbert, P. S. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 209–216.

[91] Gilboa, A., Tal, A., Shimshoni, I., and Kolomenkin, M. Computer-based, automatic recording and illustration of complex archaeological artifacts. *Journal of Archaeological Science 40*, 2 (2013), 1329–1339.

[92] Ginzburg, D., and Raviv, D. Cyclic functional mapping: Self-supervised correspondence between non-isometric deformable shapes. In *European Conference on Computer Vision* (2020), Springer, pp. 36–52.

[93] Golub, G., and Van Loan, C. *Matrix Computations.* Johns Hopkins Studies in the Mathematical Sciences. JHU Press, 2013.

[94] Grayson, B., Shah, A., and van de Geijn, R. A high performance parallel strassen implementation. *Parallel Processing Letters 6* (1995), 3–12.

[95] Greene, N. Detailing tree skeleton with voxel automata. *SIGGRAPH'91, Course Notes on Photorealistic Volume Modeling and Rendering Techniques* (1991).

[96] Habel, R., Kusternig, A., and Wimmer, M. Physically guided animation of trees. *CGF 28*, 2 (2009), 523–532.

[97] Hädrich, T., Benes, B., Deussen, O., and Pirk, S. Interactive modeling and authoring of climbing plants. *Comput. Graph. Forum 36*, 2 (May 2017), 49–61.

[98] Halimi, O., Imanuel, I., Litany, O., Trappolini, G., Rodolà, E., Guibas, L., and Kimmel, R. The whole is greater than the sum of its nonrigid parts, 2020.

[99] Halimi, O., Litany, O., Rodolà, E., Bronstein, A. M., and Kimmel, R. Unsupervised learning of dense shape correspondence. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), pp. 4370–4379.

[100] Hammond, D. K., Vandergheynst, P., and Gribonval, R. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis 30*, 2 (2011), 129–150.

[101] Hart, J. C. Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), pp. 87–94.

[102] Higham, N. Exploiting fast matrix multiplication within the level 3 blas. *ACM Trans. Math. Softw. 16*, 4 (1990), 352–368.

[103] Hirani, A. N. *Discrete exterior calculus.* California Institute of Technology, 2003.

[104] Hoppe, H. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, Association for Computing Machinery, p. 99–108.

[105] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. Surface reconstruction from unorganized points. *SIGGRAPH Comput. Graph. 26*, 2 (jul 1992), 71–78.

[106] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. Mesh optimization. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1993), SIGGRAPH '93, Association for Computing Machinery, p. 19–26.

[107] Hunold, S., Rauber, T., and Runger, G. Combining building blocks for parallel multi–level matrix multiplication. *Parallel Computing 34* (2008), 411–426.

[108] Huss-Lederman, S., Jacobson, E., Tsao, A., Turnbull, T., and Johnson, J. Implementation of strassen's algorithm for matrix multiplication. In *Proc. ACM/IEEE Conf. on Supercomputing* (1996).

[109] Ijiri, T., Owada, S., and Igarashi, T. Seamless integration of initial sketching and subsequent detail editing in flower modeling. *CGF 25*, 3 (2006), 617–624.

[110] Inkscape Project. Inkscape.

[111] Jeong, S., Park, S.-H., and Kim, C.-H. Simulation of morphology changes in drying leaves. In *Computer Graphics Forum* (2013), vol. 32, Wiley Online Library, pp. 204–215.

[112] Jia-Wei, H., and Kung, H. T. I/o complexity: The red-blue pebble game. In *Proc. ACM Symp. Theory of Computing (STOC)* (1981), ACM, p. 326–333.

[113] Johnson, D. S., and McGeoch, L. A. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization 1*, 1 (1997), 215–310.

[114] Jones, J. Characteristics of pattern formation and evolution in approximations of physarum transport networks. *Artificial Life 16* (2010), 127–153.

[115] Kac, M. Can one hear the shape of a drum? *The american mathematical monthly 73*, 4P2 (1966), 1–23.

[116] Kadhum, M., Qasem, M. H., Sleit, A., and Sharieh, A. Efficient mapreduce matrix multiplication with optimized mapper set. In *Computer Science On-line Conference* (2017), Springer, pp. 186–196.

[117] Kågström, B. Management of deep memory hierarchies–recursive blocked algorithms and hybrid data structures for dense matrix computations. In *Int. Workshop on Applied Parallel Computing* (2004), Springer, pp. 21–32.

[118] Kazhdan, M., Bolitho, M., and Hoppe, H. Poisson Surface Reconstruction. In *Symposium on Geometry Processing* (2006), A. Sheffer and K. Polthier, Eds., The Eurographics Association.

[119] Khan, D., Plopski, A., Fujimoto, Y., Kanbara, M., Jabeen, G., Zhang, Y. J., Zhang, X., and Kato, H. Surface remeshing: A systematic literature review of methods and research directions. *IEEE Transactions on Visualization and Computer Graphics 28*, 3 (2022), 1680–1713.

[120] Kider Jr, J. T., Raja, S., and Badler, N. I. Fruit senescence and decay simulation. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 257–266.

[121] Kim, J. H., and Lee-Stadelmann, O. Y. Water relations and cell wall elasticity quantities in phaseolus vulgaris leaves. *Journal of experimental botany 35*, 6 (1984), 841–858.

[122] Kirgo, M., Melzi, S., Patanè, G., Rodolà, E., and Ovsjanikov, M. Wavelet-based heat kernel derivatives: Towards informative localized shape analysis. *Computer Graphics Forum* (2020). first online Nov. 2020.

[123] Klein, J., Waller, R. E., Pirk, S., Pałubicki, W., Tester, M., and Michels, D. L. Synthetic data at scale: A paradigm to efficiently leverage machine learning in agriculture. *SSRN 4314564* (2023).

[124] Knöppel, F., Crane, K., Pinkall, U., and Schröder, P. Stripe patterns on surfaces. *ACM Trans. Graph. 34* (2015).

[125] Kobbelt, L. P., Campagna, S., and Seidel, H.-P. A general framework for mesh decimation. In *Graphics Interface* (1998).

[126] Kolomenkin, M., Shimshoni, I., and Tal, A. Demarcating curves for shape illustration. In *ACM SIGGRAPH Asia 2008 papers*. 2008, pp. 1–9.

[127] Kovnatsky, A., Bronstein, M., Bronstein, A., Glashoff, K., and Kimmel, R. Coupled quasi-harmonic bases. *Computer Graphics Forum 32*, 2pt4 (2013), 439–448.

[128] Kovnatsky, A., Glashoff, K., and Bronstein, M. M. Madmm: a generic algorithm for non-smooth optimization on manifolds. In *Proc. ECCV* (2016), Springer, pp. 680–696.

[129] Kratt, J., Spicker, M., Guayaquil, A., Fišer, M., Pirk, S., Deussen, O., Hart, J. C., and Benes, B. Woodification: User-controlled cambial growth modeling. *CGF 34*, 2 (2015), 361–372.

[130] Kwasniewski, G., Kabić, M., Besta, M., VandeVondele, J., Solcà, R., and Hoefler, T. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis* (2019), SC '19.

[131] Lefebvre, S., and Hoppe, H. Appearance-space texture synthesis. *ACM Trans. Graph. 25*, 3 (jul 2006), 541–548.

[132] Lefebvre, S., and Neyret, F. Pattern based procedural textures. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics* (New York, NY, USA, 2003), I3D '03, Association for Computing Machinery, p. 203–212.

[133] Lescoat, T., Liu, H.-T. D., Thiery, J.-M., Jacobson, A., Boubekeur, T., and Ovsjanikov, M. Spectral mesh simplification. *Computer Graphics Forum 39*, 2 (2020), 315–324.

[134] Lévy, B. Laplace-Beltrami eigenfunctions towards an algorithm that understands geometry. In *Proc. SMI* (2006), pp. 13–25.

[135] Li, B., Kałużny, J., Klein, J., Michels, D. L., Pałubicki, W., Benes, B., and Pirk, S. Learning to reconstruct botanical trees from single images. *ACM Transaction on Graphics 40*, 6 (12 2021).

[136] Li, C., Deussen, O., Song, Y.-Z., Willis, P., and Hall, P. Modeling and generating moving trees from video. *TOG 30*, 6 (2011), 127:1–127:12.

[137] Li, H., Sumner, R. W., and Pauly, M. Global correspondence optimization for non-rigid registration of depth scans. *Computer graphics forum 27*, 5 (2008), 1421–1430.

[138] Li, J. Y. S., and Zhang, H. Nonobtuse remeshing and mesh decimation. SGP '06, Eurographics Association, p. 235–238.

[139] Li, S., Chen, H., Wang, M., Heidari, A. A., and Mirjalili, S. Slime mould algorithm: A new method for stochastic optimization. *Future Generation Computer Systems 111* (2020), 300–323.

[140] Li, Y., Fan, X., Mitra, N. J., Chamovitz, D., Cohen-Or, D., and Chen, B. Analyzing growing plants from 4d point cloud data. *TOG 32*, 6 (2013).

[141] Lintermann, B., and Deussen, O. Interactive modeling of plants. *IEEE Comput. Graph. Appl. 19*, 1 (1999), 56–65.

[142] Litany, O., Bronstein, A., Bronstein, M., and Makadia, A. Deformable shape completion with graph convolutional autoencoders. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), pp. 1886–1895.

[143] Litany, O., Remez, T., Rodolà, E., Bronstein, A., and Bronstein, M. Deep functional maps: Structured prediction for dense shape correspondence. In *Proceedings of the IEEE International Conference on Computer Vision* (2017), pp. 5659–5667.

[144] Litany, O., Rodolà, E., Bronstein, A., and Bronstein, M. Fully spectral partial shape matching. *Computer Graphics Forum 36*, 2 (2017), 247–258.

[145] Litany, O., Rodolà, E., Bronstein, A. M., Bronstein, M. M., and Cremers, D. Non-rigid puzzles. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 135–143.

[146] Liu, Y., Guo, J., Benes, B., Deussen, O., Zhang, X., and Huang, H. Treepartnet: Neural decomposition of point clouds for 3d tree reconstruction. *ACM Transaction on Graphics 40*, 6 (Dec. 2021), 1–16.

[147] Liu, Y.-J., Fan, D., Xu, C.-X., and He, Y. Constructing intrinsic delaunay triangulations from the dual of geodesic voronoi diagrams. *ACM Trans. Graph. 36*, 2 (apr 2017).

[148] Liu, Y.-J., Xu, C.-X., Fan, D., and He, Y. Efficient construction and simplification of delaunay meshes. *ACM Trans. Graph. 34*, 6 (nov 2015).

[149] Liu, Y.-J., Xu, C.-X., Yi, R., Fan, D., and He, Y. Manifold differential evolution (mde): A global optimization method for geodesic centroidal voronoi tessellations on meshes. *ACM Trans. Graph. 35*, 6 (dec 2016).

[150] Livny, Y., Pirk, S., Cheng, Z., Yan, F., Deussen, O., Cohen-Or, D., and Chen, B. Texture-lobes for tree modelling. In *ACM SIGGRAPH 2011 Papers* (2011), SIGGRAPH '11, ACM.

[151] Longay, S., Runions, A., Boudon, F., and Prusinkiewicz, P. Treesketch: Interactive procedural modeling of trees on a tablet. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling* (2012), SBIM '12, p. 107–120.

[152] Loshchilov, I., and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[153] Lounsbery, M., DeRose, T. D., and Warren, J. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Trans. on Graphics 16*, 1 (1997), 34–73.

[154] Lu, J., Sogge, C. D., and Steinerberger, S. Approximating pointwise products of laplacian eigenfunctions. *Journal of Functional Analysis 277*, 9 (2019), 3271–3282.

[155] Luna-Elizarrarás, M., Saphiro, M., Struppa, D., and Vajiac, A. Bicomplex Numbers and their Elementary Functions. *Cubo (Temuco) 14* (00 2012), 61 – 80.

[156] Luo, Q., and Drake, J. A scalable parallel strassen's matrix multiplication algorithm for distributed-memory computers. In *Proc. ACM Symp. Applied Computing, SAC'95* (1995), pp. 221–226.

[157] Maggioli, F., Baieri, D., Melzi, S., and Rodolà, E. Newton's fractals on surfaces via bicomplex algebra. In *ACM SIGGRAPH 2022 Posters*. 2022, pp. 1–2.

[158] Maggioli, F., Baieri, D., and Rodolà, E. Massive uniform mesh decimation via a fast intrinsic delaunay triangulation. *arXiv preprint arXiv:2305.09274* (2023).

[159] Maggioli, F., Klein, J., Hädrich, T., Rodolà, E., Pałubicki, W., Pirk, S., and Michels, D. L. A physically-inspired approach to the simulation of plant wilting. In *SIGGRAPH Asia 2023 Conference Papers* (2023), pp. 1–8.

[160] Maggioli, F., Marin, R., Melzi, S., and Rodolà, E. MoMaS: Mold Manifold Simulation for Real-time Procedural Texturing. *Computer Graphics Forum* (2022).

[161] Maggioli, F., Melzi, S., Ovsjanikov, M., Bronstein, M. M., and Rodolà, E. Orthogonalized fourier polynomials for signal approximation and transfer. *Computer Graphics Forum 40*, 2 (2021), 435–447.

[162] Mancinelli, C., Nazzaro, G., Pellacini, F., and Puppo, E. b/surf: Interactive bezier splines on surface meshes. *IEEE Transactions on Visualization & Computer Graphics*, 01 (may 2021), 1–1.

[163] Marbach, S., Ziethen, N., Bastin, L., Baeuerle, F., and Alim, K. Network architecture determines vein fate during spontaneous reorganization, with a time delay. *bioRxiv* (2021).

[164] Marin, D., Ohrhallinger, S., and Wimmer, M. Sigdt: 2d curve reconstruction. *Computer Graphics Forum 41*, 7 (Oct. 2022), 25–36.

[165] Marin, R., Melzi, S., Rodolà, E., and Castellani, U. High-resolution augmentation for automatic template-based matching of human models. In *2019 International Conference on 3D Vision (3DV)* (2019), IEEE, pp. 230–239.

[166] Marin, R., Melzi, S., Rodolà, E., and Castellani, U. Farm: Functional automatic registration method for 3d human bodies. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 160–173.

[167] Marin, R., Rakotosaona, M.-J., Melzi, S., and Ovsjanikov, M. Correspondence learning via linearly-invariant embedding. *Advances in Neural Information Processing Systems 33* (2020).

[168] Marin, R., Rampini, A., Castellani, U., Rodolà, E., Ovsjanikov, M., and Melzi, S. Instant recovery of shape from spectrum via latent space connections. In *2019 International Conference on 3D Vision (3DV)* (2019), IEEE, pp. 37–46.

[169] Marin, R., Rampini, A., Castellani, U., Rodolà, E., Ovsjanikov, M., and Melzi, S. Spectral shape recovery and analysis via data-driven connections. *International Journal of Computer Vision 129*, 10 (2021), 2745–2760.

[170] Maung, D., Shi, Y., and Crawfis, R. Procedural textures using tilings with perlin noise. In *2012 17th International Conference on Computer Games (CGAMES)* (2012), IEEE, pp. 60–65.

[171] McGraw, T., and Ferdousi, B. Red versus blue: Slime mold civil war. In *SIGGRAPH Asia 2021 Posters* (New York, NY, USA, 2021), SA '21 Posters, Association for Computing Machinery.

[172] Melzi, S. Sparse representation of step functions on manifolds. *Computers & Graphics 82* (2019), 117 – 128.

[173] Melzi, S., Marin, R., Musoni, P., Bardon, F., Tarini, M., and Castellani, U. Intrinsic/extrinsic embedding for functional remeshing of 3D shapes. *CAG 88* (2020), 1 – 12.

[174] Melzi, S., Marin, R., Rodolà, E., Castellani, U., Ren, J., Poulenard, A., Wonka, P., and Ovsjanikov, M. SHREC 2019: Matching Humans with Different Connectivity. In *Eurographics Workshop on 3D Object Retrieval* (2019), The Eurographics Association.

[175] Melzi, S., Ovsjanikov, M., Roffo, G., Cristani, M., and Castellani, U. Discrete time evolution process descriptor for shape analysis and matching. *ACM Transactions on Graphics (TOG) 37*, 1 (Jan. 2018), 4:1–4:18.

[176] Melzi, S., Ren, J., Rodolà, E., Sharma, A., Wonka, P., and Ovsjanikov, M. Zoomout: Spectral upsampling for efficient shape correspondence. *ACM Transactions on Graphics (TOG) 38*, 6 (Nov. 2019), 155:1–155:14.

[177] Melzi, S., Rodolà, E., Castellani, U., and Bronstein, M. Shape analysis with anisotropic windowed fourier transform. In *International Conference on 3D Vision (3DV)* (2016).

[178] Melzi, S., Rodolà, E., Castellani, U., and Bronstein, M. Localized manifold harmonics for spectral shape analysis. *Computer Graphics Forum 37*, 6 (2018), 20–34.

[179] Milios, E. E. Shape matching using curvature processes. *Computer Vision, Graphics, and Image Processing 47*, 2 (1989), 203–226.

[180] Mishra, S., and Granskog, J. CLIP-based Neural Neighbor Style Transfer for 3D Assets. In *Eurographics 2023 - Short Papers* (2023), V. Babaei and M. Skouras, Eds., The Eurographics Association.

[181] Mitra, N. J., Flöry, S., Ovsjanikov, M., Gelfand, N., Guibas, L. J., and Pottmann, H. Dynamic geometry registration. In *Symposium on geometry processing* (2007), pp. 173–182.

[182] Mo, K., Zhu, S., Chang, A. X., Yi, L., Tripathi, S., Guibas, L. J., and Su, H. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[183] Molz, F. J. Models of water transport in the soil-plant system: A review. *Water resources research 17*, 5 (1981), 1245–1260.

[184] Morita, S., Society, A. M., Nagase, T., and Nomizu, K. *Geometry of Differential Forms*. Iwanami series in modern mathematics. American Mathematical Society, 2001.

[185] Moschella, L., Melzi, S., Cosmo, L., Maggioli, F., Litany, O., Ovsjanikov, M., Guibas, L., and Rodolà, E. Learning spectral unions of partial deformable 3d shapes. In *Computer Graphics Forum* (2022), vol. 41, Wiley Online Library, pp. 407–417.

[186] Nakagaki, T. Smart behavior of true slime mold in a labyrinth. *Research in Microbiology 152*, 9 (2001), 767–770.

[187] Nasikun, A., Brandt, C., and Hildebrandt, K. Fast approximation of laplace-beltrami eigenproblems. *Computer Graphics Forum 37*, 5 (2018), 121–134.

[188] Nazzaro, G., Puppo, E., and Pellacini, F. Geotangle: Interactive design of geodesic tangle patterns on surfaces. *ACM Trans. Graph. 41*, 2 (nov 2021).

[189] Neumann, T., Varanasi, K., Theobalt, C., Magnor, M., and Wacker, M. Compressed manifold modes for mesh processing. *Computer Graphics Forum 33*, 5 (2014), 35–44.

[190] Newcombe, R. A., Fox, D., and Seitz, S. M. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proc. CVPR* (2015), IEEE, pp. 343–352.

[191] Neyret, F. Animated texels. In *Computer Animation and Simulation '95* (Vienna, 1995), D. Terzopoulos and D. Thalmann, Eds., Springer Vienna, pp. 97–103.

[192] Neyret, F. Advected textures. In *ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), Eurographics Association, pp. 147–153.

[193] Nogneng, D., Melzi, S., Rodolà, E., Castellani, U., Bronstein, M., and Ovsjanikov, M. Improved functional mappings via product preservation. *Computer Graphics Forum 37*, 2 (2018), 179–190.

[194] Nogneng, D., and Ovsjanikov, M. Informative descriptor preservation via commutativity for shape matching. *Computer Graphics Forum 36*, 2 (2017), 259–267.

[195] Norback, J. P., and Love, R. F. Geometric approaches to solving the traveling salesman problem. *Management Science 23*, 11 (1977), 1208–1223.

[196] Ohrhallinger, S., Mitchell, S., and Wimmer, M. Curve reconstruction with many fewer samples. *Computer Graphics Forum 35* (08 2016), 167–176.

[197] Ohrhallinger, S., and Mudur, S. An Efficient Algorithm for Determining an Aesthetic Shape Connecting Unorganized 2D Points. *Computer Graphics Forum* (2013).

[198] Ohrhallinger, S., Peethambaran, J., Parakkat, A. D., Dey, T. K., and Muthuganapathy, R. 2d points curve reconstruction survey and benchmark. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 611–632.

[199] Okabe, M., Owada, S., and Igarashi, T. Interactive design of botanical trees using freehand sketches and example-based editing. In *ACM SIGGRAPH Courses* (2007), ACM.

[200] Olsen, J. Realtime procedural terrain generation.

[201] Oppenheimer, P. E. Real time design and animation of fractal plants and trees. *Proc. of SIGGRAPH 20*, 4 (1986), 55–64.

[202] Ovsjanikov, M., Ben-Chen, M., Solomon, J., Butscher, A., and Guibas, L. Functional maps: a flexible representation of maps between shapes. *ACM Transactions on Graphics (ToG) 31*, 4 (2012), 1–11.

[203] Ovsjanikov, M., Corman, E., Bronstein, M., Rodolà, E., Ben-Chen, M., Guibas, L., Chazal, F., and Bronstein, A. Computing and processing correspondences with functional maps. In *SIGGRAPH ASIA 2016 Courses* (New York, NY, USA, 2016), ACM, pp. 9:1–9:60.

[204] Owens, A., Cieslak, M., Hart, J., Classen-Bockhoff, R., and Prusinkiewicz, P. Modeling dense inflorescences. *ACM Trans. Graph. 35*, 4 (jul 2016).

[205] Palubicki, W., Horel, K., Longay, S., Runions, A., Lane, B., Měch, R., and Prusinkiewicz, P. Self-organizing tree models for image synthesis. *ACM Trans. Graph. 28*, 3 (2009), 58:1–58:10.

[206] Pałubicki, W., Makowski, M., Gajda, W., Hädrich, T., Michels, D. L., and Pirk, S. Ecoclimates: Climate-response modeling of vegetation. *ACM Trans. Graph. 41*, 4 (2022).

[207] Parakkat, A. D., and Muthuganapathy, R. Crawl through neighbors: A simple curve reconstruction algorithm. *Computer Graphics Forum 35*, 5 (2016), 177–186.

[208] Parberry, I. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques 3*, 1 (2014).

[209] Patanè, G. Laplacian spectral basis functions. *Computer-Aided Geometric Design 65* (2018), 31 – 47.

[210] Patino-Ramirez, F., Arson, C., and Dussutour, A. Substrate and cell fusion influence on slime mold network dynamics. *Scientific reports 11*, 1 (2021), 1–20.

[211] Peise, E., and Bientinesi, P. Algorithm 979: recursive algorithms for dense linear algebra—the relapack collection. *ACM Trans. Math. Softw. (TOMS) 44*, 2 (2017), 1–19.

[212] Peitgen, H., Saupe, D., Fisher, Y., Barnsley, M., for Computing Machinery Special Interest Group on Graphics, A., McGuire, M., Mandelbrot, B., Devaney, R., and Voss, R. *The Science of Fractal Images.* Springer New York, 1988.

[213] Pinkall, U., and Polthier, K. Computing Discrete Minimal Surfaces and their Conjugates. *Exp. Math. 2*, 1 (1993), 15–36.

[214] Pirk, S., Jarząbek, M., Hädrich, T., Michels, D. L., and Palubicki, W. Interactive wood combustion for botanical tree models. *ACM Trans. Graph. 36*, 6 (nov 2017), 197:1–197:12.

[215] Pirk, S., Niese, T., Deussen, O., and Neubert, B. Capturing and animating the morphogenesis of polygonal tree models. *TOG 31*, 6 (2012), 169:1–169:10.

[216] Pirk, S., Stava, O., Kratt, J., Said, M. A. M., Neubert, B., Měch, R., Benes, B., and Deussen, O. Plastic trees: Interactive self-adapting botanical tree models. *ACM Trans. Graph. 31*, 4 (July 2012), 50:1–50:10.

[217] Poerner, M., Suessmuth, J., Ohadi, D., and Amann, V. Adidas tape: 3-d footwear concept design. In *ACM SIGGRAPH 2018 Talks* (New York, NY, USA, 2018), SIGGRAPH '18, Association for Computing Machinery.

[218] Pogorui, A., and Rodríguez-Dagnino, R. On the set of zeros of bicomplex polynomials. *Complex Variables and Elliptic Equations 51*, 7 (2006), 725–730.

[219] Potamias, R. A., Ploumpis, S., and Zafeiriou, S. Neural mesh simplification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2022), pp. 18583–18592.

[220] Prusinkiewicz, P., and Lindenmayer, A. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., 1990.

[221] Qasem, M. H., Sarhan, A. A., Qaddoura, R., and Mahafzah, B. A. Matrix multiplication of big data using mapreduce: a review. In *2nd Int. Conf. Applications of Information Technology in Developing Renewable Energy Processes & Systems (IT-DREPS)* (2017), IEEE, pp. 1–6.

[222] Quigley, E., Yu, Y., Huang, J., Lin, W., and Fedkiw, R. Real-time interactive tree animation. *TVCG 24*, 5 (2018), 1717–1727.

[223] Radwan, M., Ohrhallinger, S., Eisemann, E., and Wimmer, M. Cut and paint: Occlusion-aware subset selection for surface processing. In *Proceedings of Graphics Interface 2017* (May 2017), Canadian Human-Computer Communications Society / Société canadienne du dialogue humain-machine, pp. 82–89.

[224] Rampini, A., Tallini, I., Ovsjanikov, M., Bronstein, A. M., and Rodolà, E. Correspondence-free region localization for partial shape similarity via hamiltonian spectrum alignment. In *2019 International Conference on 3D Vision (3DV)* (2019), IEEE, pp. 37–46.

[225] Reche-Martinez, A., Martin, I., and Drettakis, G. Volumetric reconstruction and interactive rendering of trees from photographs. *TOG 23*, 3 (2004), 720–727.

[226] Reeves, W. T., and Blau, R. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph. 19*, 3 (July 1985), 313–322.

[227] Ren, J., Poulenard, A., Wonka, P., and Ovsjanikov, M. Continuous and orientation-preserving correspondences via functional maps. *ACM Transactions on Graphics (TOG) 37*, 6 (2018).

[228] Reuter, M., Wolter, F.-E., and Peinecke, N. Laplace–beltrami spectra as 'shape-dna' of surfaces and solids. *Computer-Aided Design 38*, 4 (2006), 342 – 366. Symposium on Solid and Physical Modeling 2005.

[229] Ringham, L., Owens, A., Cieslak, M., Harder, L. D., and Prusinkiewicz, P. Modeling flower pigmentation patterns. *ACM Trans. Graph. 40*, 6 (dec 2021).

[230] Riso, M., Nazzaro, G., Puppo, E., Jacobson, A., Zhou, Q., and Pellacini, F. Boolsurf: Boolean operations on surfaces. *ACM Transactions on Graphics (TOG) 41*, 6 (2022), 1–13.

[231] Rivest, R. L., Adleman, L., and Dertouzos, M. L. On data banks and privacy homomorphisms. *FOUNDATIONS OF SECURE COMPUTATIONS* (1979), 171.

[232] Roberts, M. B. V. *Biology: a functional approach.* Nelson Thornes, 1986.

[233] Rodolà, E., Cosmo, L., Bronstein, M., Torsello, A., and Cremers, D. Partial functional correspondence. *Computer Graphics Forum 36*, 1 (2017), 222–236.

[234] Rodolà, E., Cosmo, L., Litany, O., Bronstein, M., Bronstein, A., Audebert, N., Hamza, A. B., Boulch, A., Castellani17, U., Do16, M., et al. Shrec'17: Deformable shape retrieval with missing parts.

[235] Rolfsen, D. *Knots and Links.* Mathematics lecture series. Publish or Perish, 1976.

[236] Rönn, S. Bicomplex algebra and function theory. *arXiv preprint math/0101200* (2001).

[237] Roufosse, J.-M., Sharma, A., and Ovsjanikov, M. Unsupervised deep learning for structured shape matching. In *Proceedings of the IEEE International Conference on Computer Vision* (2019), pp. 1617–1627.

[238] Runions, A., Lane, B., and Prusinkiewicz, P. Modeling trees with a space colonization algorithm. *EG Nat. Phenom.* (2007), 63–70.

[239] Ruppert, J. A new and simple algorithm for quality 2-dimensional mesh generation. In *ACM-SIAM Symposium on Discrete Algorithms* (1993).

[240] Rustamov, R. M. Laplace-beltrami eigenfunctions for deformation invariant shape representation. In *Proc. SGP* (2007), Eurographics Association, pp. 225–233.

[241] Rustamov, R. M., Ovsjanikov, M., Azencot, O., Ben-Chen, M., Chazal, F., and Guibas, L. Map-based exploration of intrinsic shape differences and variability. *ACM Transactions on Graphics (TOG) 32*, 4 (2013), 1–12.

[242] Schroeder, W. J., Zarge, J. A., and Lorensen, W. E. Decimation of triangle meshes. *SIGGRAPH Comput. Graph. 26*, 2 (jul 1992), 65–70.

[243] Shao, H., Kugelstadt, T., Hädrich, T., Pałubicki, W., Bender, J., Pirk, S., and Michels, D. L. Accurately solving rod dynamics with graph learning. In *Advances in Neural Information Processing Systems (NeurIPS)* (2021).

[244] Sharf, A., Alcantara, D. A., Lewiner, T., Greif, C., Sheffer, A., Amenta, N., and Cohen-Or, D. Space-time surface reconstruction using incompressible flow. *ACM Transactions on Graphics (TOG) 27*, 5 (2008), 1–10.

[245] Sharp, N., and Crane, K. A Laplacian for Nonmanifold Triangle Meshes. *Computer Graphics Forum (SGP) 39*, 5 (2020).

[246] Sharp, N., and Crane, K. You can find geodesic paths in triangle meshes by just flipping edges. *ACM Trans. Graph. 39*, 6 (nov 2020).

[247] Sharp, N., Gillespie, M., and Crane, K. Geometry processing with intrinsic triangulations. In *ACM SIGGRAPH 2021 Courses* (New York, NY, USA, 2021), SIGGRAPH '21, Association for Computing Machinery.

[248] Sharp, N., Soliman, Y., and Crane, K. Navigating intrinsic triangulations. *ACM Trans. Graph. 38*, 4 (jul 2019).

[249] Shtern, A., and Kimmel, R. Spectral gradient fields embedding for non-rigid shape matching. *CVIU 140* (2015), 21–29.

[250] Slavcheva, M., Baust, M., Cremers, D., and Ilic, S. Killingfusion: Non-rigid 3d reconstruction without correspondences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 1386–1395.

[251] Smith, D. M., and Allen, S. J. Measurement of sap flow in plant stems. *Journal of Experimental Botany 47*, 305 (1996).

[252] Song, F., Dongarra, J., and Moore, S. Experiments with strassen's algorithm: From sequential to parallel. In *Proc. Parallel and Distributed Computing and Systems, (PDCS)* (2006).

[253] Stam, J. Flows on surfaces of arbitrary topology. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, Association for Computing Machinery, p. 724–731.

[254] Stava, O., Pirk, S., Kratt, J., Chen, B., Měch, R., Deussen, O., and Benes, B. Inverse procedural modelling of trees. *Computer Graphics Forum* (2014), n/a–n/a.

[255] Stewart, G. W. A krylov–schur algorithm for large eigenproblems. *SIAM J. Matrix Anal. Appl. 23*, 3 (2002), 601–614.

[256] Stothers, A. On the complexity of matrix multiplication. *Journal of Complexity 19* (2003), 43–60.

[257] Strang, G. *Linear Algebra and Its Applications, Fourth Ed.* Thomson Brooks/Cole, 2006.

[258] Strassen, V. Gaussian elimination is not optimal. *Numerische mathematik 13*, 4 (1969), 354–356.

[259] Süli, E., and Mayers, D. F. *An introduction to numerical analysis.* Cambridge university press, 2003.

[260] Sun, J., Ovsjanikov, M., and Guibas, L. A concise and provably informative multi-scale signature based on heat diffusion. *Computer graphics forum 28*, 5 (2009), 1383–1392.

[261] Sun, Y., hu, K., Zhang, K., Jiang, L., and Xu, Y. Simulation of nitrogen fate for greenhouse cucumber grown under different water and fertilizer management using the eu-rotaten model. *Agricultural Water Management 112* (09 2012), 21–32.

[262] Sung, M., Dubrovina, A., Kim, V. G., and Guibas, L. Learning fuzzy set representations of partial shapes on dual embedding spaces. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 71–81.

[263] Taiz, L., Zeiger, E., Møller, I. M., Murphy, A., et al. *Plant physiology and development.* No. Ed. 6. Sinauer Associates Incorporated, 2015.

[264] Tan, P., Fang, T., Xiao, J., Zhao, P., and Quan, L. Single image tree modeling. *TOG 27*, 5 (2008), 108:1–108:7.

[265] Taubin, G. A signal processing approach to fair surface design. In *ACM SIGGRAPH* (1995), pp. 351–358.

[266] Tevs, A., Berner, A., Wand, M., Ihrke, I., Bokeloh, M., Kerber, J., and Seidel, H.-P. Animation cartography—intrinsic reconstruction of shape and motion. *ACM Transactions on Graphics (TOG) 31*, 2 (2012), 1–15.

[267] Thottethodi, M., Chatterjee, S., and Lebeck, A. Tuning strassen's matrix multiplication for memory efficiency. In *Proc. 1998 ACM/IEEE Conf. on Supercomputing (SC'98)* (1998), IEEE, pp. 36–36.

[268] Tombari, F., Salti, S., and Di Stefano, L. Unique signatures of histograms for local surface description. In *Proc. ECCV* (2010), Springer, pp. 356–369.

[269] Turk, G. Generating textures on arbitrary surfaces using reaction-diffusion. *SIGGRAPH Comput. Graph. 25*, 4 (jul 1991), 289–298.

[270] Turk, G. Texture synthesis on surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), pp. 347–354.

[271] Udriste, C. *Convex functions and optimization methods on Riemannian manifolds*, vol. 297. Springer Science & Business Media, 1994.

[272] Vallet, B., and Lévy, B. Spectral geometry processing with manifold harmonics. *Computer Graphics Forum 27*, 2 (2008), 251–260.

[273] Vallverdú, J., Castro, O., Mayne, R., Talanov, M., Levin, M., Baluška, F., Gunji, Y., Dussutour, A., Zenil, H., and Adamatzky, A. Slime mould: The fundamental mechanisms of biological cognition. *Biosystems 165* (2018), 57–70.

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

[275] Varol, G., Romero, J., Martin, X., Mahmood, N., Black, M. J., Laptev, I., and Schmid, C. Learning from synthetic humans. In *CVPR* (2017).

[276] Wand, M., Adams, B., Ovsjanikov, M., Berner, A., Bokeloh, M., Jenke, P., Guibas, L., Seidel, H.-P., and Schilling, A. Efficient reconstruction of nonrigid shape and motion from real-time 3d scanner data. *ACM Transactions on Graphics (TOG) 28*, 2 (2009), 1–15.

[277] Wand, M., Jenke, P., Huang, Q., Bokeloh, M., Guibas, L., and Schilling, A. Reconstruction of deforming geometry from time-varying point clouds. In *Symposium on Geometry processing* (2007), pp. 49–58.

[278] Wang, E., Zhang, Q., Zhang, B. S. G., Lu, X., Wu, Q., and Wang, Y. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[279] Wang, H. Y., Kang, M. Z., Hua, J., and Wang, X. J. Modeling plant plasticity from a biophysical model: Biomechanics. In *Proceedings of the 12th ACM SIGGRAPH Intl. Conf. on VRCAI* (2013), ACM, pp. 115–122.

[280] Wang, X. Intrinsic computation of voronoi diagrams on surfaces and its application, 2015.

[281] Wang, X., Ying, X., Liu, Y.-J., Xin, S.-Q., Wang, W., Gu, X., Mueller-Wittig, W., and He, Y. Intrinsic computation of centroidal voronoi tessellation (cvt) on meshes. *Comput. Aided Des. 58*, C (jan 2015), 51–61.

[282] Wang, X.-Y., and Song, W. The generalized m–j sets for bicomplex numbers. *Nonlinear Dynamics 72* (2013), 17–26.

[283] Watson, D. F. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*. *The Computer Journal 24*, 2 (01 1981), 167–172.

[284] Webster, N. L. High poly to low poly workflows for real-time rendering. *Journal of visual communication in medicine 40*, 1 (2017), 40–47.

[285] Wei, L.-Y., and Levoy, M. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), pp. 355–360.

[286] Weyl, H. Über die asymptotische verteilung der eigenwerte. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse 1911* (1911), 110–117.

[287] Wie, L.-Y., Lefebvre, S., Kwatra, V., and Turk, G. State of the Art in Example-based Texture Synthesis. In *Eurographics 2009 - State of the Art Reports* (2009), M. Pauly and G. Greiner, Eds., The Eurographics Association.

[288] Williams, V. Multiplying matrices faster than Coppersmith-Winograd. In *Proc. 44th ACM Symp. Theory of Computing (STOC)* (2012), p. 887–898.

[289] Wimmer, M., and Wonka, P. Rendering time estimation for real-time rendering. In *Rendering Techniques* (2003), pp. 118–129.

[290] Wither, J., Boudon, F., Cani, M.-P., and Godin, C. Structure from silhouettes: a new paradigm for fast sketch-based design of trees. *CGF 28*, 2 (2009), 541–550.

[291] Witkin, A., and Kass, M. Reaction-diffusion textures. In *Proceedings of the 18th annual conference on computer graphics and interactive techniques* (1991), pp. 299–308.

[292] Wong, S.-K., and Chen, K.-C. A procedural approach to modelling virtual climbing plants with tendrils. *Comput. Graph. Forum* (2015).

[293] Wouter Bac, C., Hemming, J., van Tuijl, B. A. J., Barth, R., Wais, E., and van Henten, E. J. Performance evaluation of a harvesting robot for sweet pepper. *Journal of Field Robotics 34*, 6 (2017), 1123–1139.

[294] Yang, C.-Q., and Miller, B. P. Critical path analysis for the execution of parallel and distributed programs. In *Proc. 8th Int. Conf. on Distributed Computing Systems (ICDCS)* (1988), IEEE, pp. 366–373.

[295] Yi, R., Liu, Y.-J., and He, Y. Delaunay mesh simplification with differential evolution. *ACM Trans. Graph. 37*, 6 (dec 2018).

[296] Yin, M., Li, G., Lu, H., Ouyang, Y., Zhang, Z., and Xian, C. Spectral pose transfer. *Computer Aided Geometric Design 35* (2015), 82–94.

[297] Yuksel, C., Lefebvre, S., and Tarini, M. Rethinking texture mapping. *Computer Graphics Forum 38*, 2 (2019), 535–551.

[298] Zeng, W., Guo, R., Luo, F., and Gu, X. Discrete heat kernel determines discrete riemannian metric. *Graphical Models 74*, 4 (2012), 121–129.

[299] Zhang, Q., Fu, B., Ye, M., and Yang, R. Quality dynamic human body modeling using a single low-cost depth camera. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2014), pp. 676–683.

[300] Zhao, J., Gao, Z.-M., and Sun, W. The improved slime mould algorithm with levy flight. *Journal of Physics: Conference Series 1617* (aug 2020), 012033.

[301] Zhao, Y., and Barbič, J. Interactive authoring of simulation-ready plants. *ACM Trans. Graph. 32*, 4 (2013), 84:1–84:12.

[302] Zhong, M. Harmonic shape analysis: From fourier to wavelets. Master's thesis, Stony Brook University, 2012.

[303] Zhou, Q., and Jacobson, A. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (2016).

[304] Zuffi, S., Kanazawa, A., Jacobs, D., and Black, M. J. 3D menagerie: Modeling the 3D shape and pose of animals. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (July 2017).