Grid-based Finite Elements System for Solving Laplace-Beltrami Equations on 2-Manifolds

by

Ming Chuang

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2013

Abstract

Solving the Poisson equation has numerous important applications. On a Riemannian 2-manifold, the task is most often formulated in terms of finite elements and two challenges commonly arise: discretizing the space of functions and solving the resulting system of equations.

In this thesis, we describe a finite elements system that simultaneously addresses both aspects. The idea is to define a space of functions in 3D and then restrict the 3D functions to the mesh. Unlike traditional approaches, the discretized function space is defined without having to depend on the tessellation of the surface. Also importantly, the regularity and nesting structure of the function space supports an efficient, parallel multigrid solver.

A straightforward implementation of our approach works effectively when the surface embedding remains static. However, it is not well-suited to evolving domains, as the system needs to be set up repeatedly when embeddings change. We show that by tracking the metric structure, a large portion of the computational effort can be amortized and reused, avoiding a costly initialization of multigrid hierarchies at the beginning of each time-step. This idea of decoupling the metric structure from the initial embedding not only enables our system to support efficient surface evolution, but also provides a flexible means for interactively adjusting the anisotropy of diffusion-like processes, supporting real-time anisotropic signal processing.

Perhaps the most distinguishing characteristic of our approach is the use of an *extrinsic* function space. While this is different from the traditional approach and imbues our system with the much desired regularity, it also raises the concern that it could lead to embedding-dependent artifacts. Indeed, by using 3D functions defined in Euclidean space instead of functions defined over the manifold, we inherently supplant geodesic distances with Euclidean ones. As a result, points adjacent in 3D will have similar function values even if they are geodesically distant. To address the concern, we propose an extension to our system that enriches the function space by splitting existing functions as necessary. The extension, together with the metric tracking for evolving surfaces, complements our framework by making the function space behave like an intrinsic one.

We conduct numerous experiments to evaluate our framework. These include spectral analyses revealing the embedding-invariant robustness of our discretization, and convergence/performance analyses revealing the competitiveness of our approach against state-of-the-art methods. We also apply our work to various geometry-processing applications. Using curvature flows, we demonstrate that we can support efficient surface evolution where embeddings change with time. Formulating surface filtering as a solution to the screened-Poisson equation, we demonstrate that we can support an anisotropic editing system for surface details that processes high resolution meshes in real time.

Primary Reader: Michael Kazhdan

Secondary Readers: Szymon Rusinkiewicz and Randal Burns

Acknowledgments

Thank you Misha; you are my mentor, my guide, and my inspiration; you are the role model whom I strive to be. Thank you Mom and Dad; you are my cheerleaders, my supporters, and my advocates; you are the reason that I have so much passion in my heart. Most importantly, thank you Huan; you are my light, my strength, and my song; you are my everything.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decade, the graphics community has developed a number of exciting applications relating to the Poisson equation. Important examples include Poisson Image Editing [5], Laplacian Surface Editing [6], and Poisson Surface Reconstruction [7]. In these cases, solving the Poisson equation provides a natural means for integrating *local* constraints into a *smooth, global* solution. Specifically, the system is often constrained by prescribed gradients (i.e. targeted local differences), and then the solution with best matching gradients in the least-square sense is computed.

Due to its broad utility in simulation and modeling, being able to solve the Poisson equation efficiently has become an important topic. The goal of this thesis is to develop an effective numerical framework for solving Poisson-type problems that addresses various issues with previous methods.

We will particularly focus on the context of Riemannian surfaces (e.g., triangle

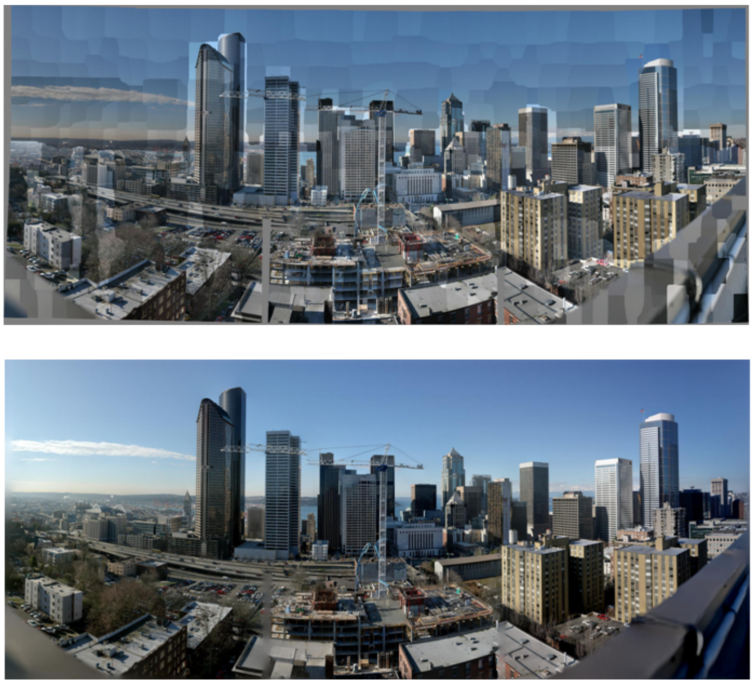


Figure 1.1: Panorama image stitching of 643 input photographs with differing exposures (image courtesy of [1]). While a direct composition of images results in visible discontinuities across image boundaries *(top)*, solving the Poisson equation for the gradient field gives a seamless result *(bottom)*.

meshes in 3D) where the inner product on tangent spaces is defined everywhere. Unlike in the context of regular grids (e.g., images), where the Fourier and multigrid techniques are readily available, solving the Poisson equation on surfaces is a non-trivial task: the discretization must take into account the metric and, since the resulting system of equations is often derived from an unstructured parameter domain, it is more difficult to develop an efficient numerical solver.

Having an effective framework for solving the Poisson equation on surfaces enables numerous applications. We can extend a large class of gradient-domain techniques designed for image processing to surfaces. For example, in the classical image stitching

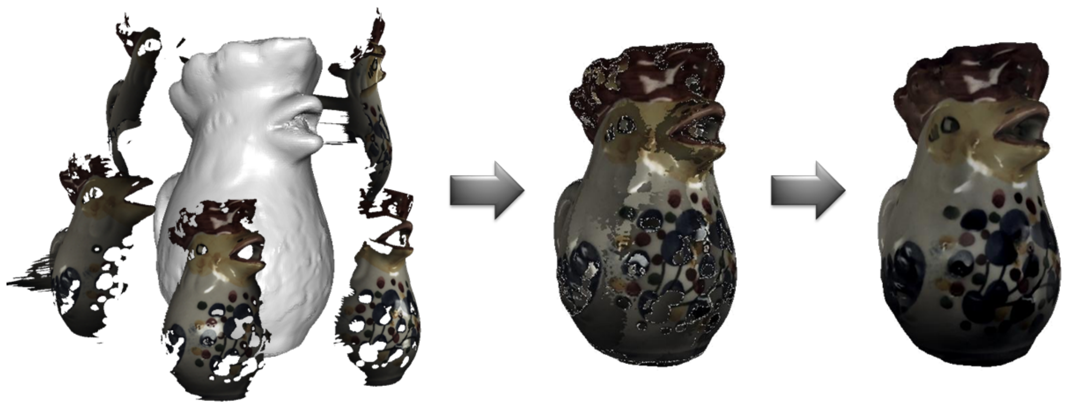


Figure 1.2: Color reconstruction of the rooster model from 3D scans *(left)*. Due to the lighting variation across scans, directly pulling color values from the closest scans results in visible discontinuities at scan transitions *(middle)*. Taking color gradients from the closest scans instead and solving the Poisson equation gives a seamless stitching result *(right)*.

problem (Figure 1.1), visible discontinuities across image boundaries can be removed by solving a Poisson system. Shifting the domain to surfaces, one can stitch color scans using a similar approach. As shown in Figure 1.2, due to lighting variation simply mosaicing scans over the surface results in artifacts at scan boundaries. Instead, pulling color gradients from scans, setting seam-crossing gradients to zero, and solving for the best-fitting color field, one obtains a seamless texture.

Perhaps more interestingly, these gradient-domain techniques can further be extended to edit the geometry itself. While this is not interesting in the context of images (where the domain is planar), in the context of surfaces it provides a versatile tool for surface editing. As an example, using the embedding itself as the signal to be processed, we can smooth or sharpen the geometry as shown in Figure 1.3.

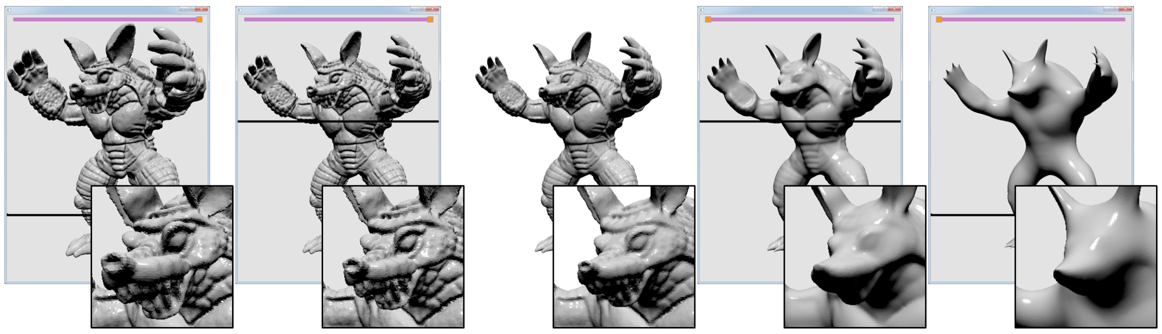


Figure 1.3: Gradient-domain geometry filtering. Setting the surface embedding as the signal to be processed *(center)* and solving a Poisson equation, we can perform operations like geometry smoothing *(right)* and sharpening *(left)*.

1.1 Problem Statement

In solving partial differential equations, the finite elements method (FEM) has emerged as a popular numerical technique over the past half century. FEM discretizes a continuous system by using a finite-dimensional function space. In particular, this is done by first choosing a finite set of test functions and then searching the best solution within the span of these functions.

On a tessellated surface, the common strategy for choosing test functions is to use shifts of a kernel (e.g., hat functions) centered at mesh vertices. The resulting linear system is most often solved by a "black-box" solver, such as the (iterative) conjugate-gradients solver [8] and the (direct) Cholesky-factorization solver [9]. Unfortunately, there are several drawbacks in this configuration:

1. **Tessellation dependence.** As the discretization of the problem depends on the tessellation, so does the numerical stability. Desirable properties of tessellation include uniform distribution of vertices and well-shaped elements (e.g.,

equilateral triangles). Poor tessellation quality often leads to an ill-conditioned system that is difficult to solve robustly and efficiently.

2. **No control of complexity.** The size of the system is determined by the number of mesh vertices. As a result, we have no choice but to solve an expensive system when the mesh is high resolution, even if the function of interest only has low frequency content.

3. **No multi-resolution structure.** A priori, such a function space is not equipped with a multi-resolution structure. As a result, it is difficult to implement a multi-level solver for fast approximation of the solution. In practice, non-hierarchical iterative solvers are too slow on their own. Direct solvers can be faster, but they do not support real-time applications on large scale problems. Additionally, they have a superlinear complexity in both time and space [10].

4. **Not parallelization/streaming friendly.** The lack of regularity in unstructured tessellation makes streaming/parallel implementation difficult. This makes it hard to take full advantages of many-core/multi-core hardware and also makes it challenging to solve out-of-core problems.

The goal of this work is to develop a numerical framework addressing these issues.

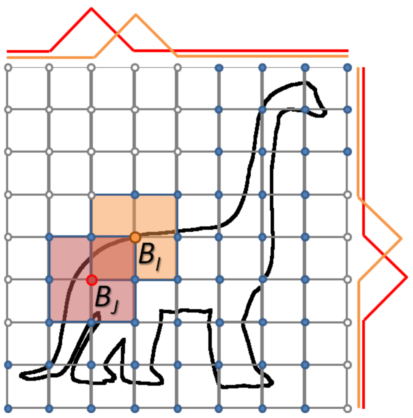


Figure 1.4: A mesh embedded in a regular voxel grid within which first-order B-splines are uniformly centered at grid corners.

1.2 General Approach

Looking at the list in the previous section, we observe that the main cause of the problems comes from the fact that the choice of test functions is coupled to the tessellation. Hence, our key idea is to decouple this relationship.

In particular, instead of defining an intrinsic function space based on the tessellation of the mesh, we define an *extrinsic* function space over the embedded manifold. We start by embedding the mesh within a regular voxel grid where test functions are centered at grid corners. We then restrict these 3D functions to the mesh. Figure 1.4 demonstrates the idea using a 2D example.

There are several advantages to this formulation: The function space defined in this way does not depend on tessellation; We have control over the system complexity by changing grid resolution; Using refinable functions as explained in later chapters, we obtain a multi-resolution structure supporting multi-level solvers; And finally, the regularity of the grid structure makes parallel/streaming implementations of our system feasible.

Extending to Evolving Embeddings

In applications like surface flow and mesh editing, where the embedding evolves with time, using our approach may seem impractical. A straightforward implementation of our approach would require re-embedding the surface within a voxel grid, setting up a new set of test functions and computing a new multi-resolution hierarchy at each time-step. Nevertheless, as we will show in this thesis, by tracking the evolving metric structure, we can evolve the test functions *with* the surface. As a result, the same multi-resolution hierarchy can be reused over time, significantly improving the performance of our system.

Extending to Anisotropy

The idea of decoupling the metric structure from the embedding is powerful: it allows us to anisotropically (and inhomogeneously) re-scale a gradient field by changing metric tensors, adjusting the system and/or constraints as needed. In particular, for gradient-domain signal processing techniques, this allows us to adaptively weight the fit of a candidate solution. As an example, in the application of surface filtering, scaling up distances across the feature lines of the mesh makes the filtering become edge-aware.

Extending to connectivity awareness

By using 3D test functions, we supplant geodesic distances with Euclidean ones. As a result, points adjacent in 3D will have similar function values even if they are geodesically distant. The problem becomes even more prominent when the mesh self-intersects, as by construction, points mapping to the same 3D location have to take on the same function values. To address this concern, we propose another extension to our approach that splits functions as necessary, so that the resulting function space becomes *connectivity-aware*. This final extension consolidates our framework and, to a large degree, makes our approach robust to the choice of embedding.

1.3 Organization

The rest of the thesis is organized as follows. We start with a brief literature survey in Chapter 2. We then review the relevant background for the finite elements method, B-splines, and multigrid in Chapter 3. We formally introduce our grid-based finite elements system and describe the implementation in Chapter 4. We then describe how modifying the metric enables efficient surface evolution and anisotropic signal processing in Chapter 5. We describe the extension of our approach that makes the function space connectivity-aware in Chapter 6. And finally, we conclude by summarizing our work and discussing directions for future research in Chapter 7.

Chapter 2

Related Work

Solving the Poisson equation has widespread applications in scientific and engineering fields. In this chapter, we focus on a survey of its uses in image processing and geometry processing (Section 2.1), and numerical strategies for solving the resulting systems (Section 2.2).

2.1 The Poisson Equation in Graphics

Many gradient-domain techniques require solving the energy minimization problem:

$$\min_u \int ||\nabla u - \vec{s}||^2 \tag{2.1}$$

That is, given the vector-valued function, $\vec{s}$, describing a gradient field, we want to find the scalar field, u, whose gradients best fit $\vec{s}$. Using the Euler-Lagrange formulation, one can show that the minimizer to 2.1 needs to satisfy the Poisson equation:

$$\Delta u = \nabla \cdot \vec{s}$$

In many applications, solving the Poisson equation provides a convenient way for integrating desired local differences into a smooth global solution.

Image Processing

Recently, gradient-domain techniques have been extensively used for various image processing purposes. The common workflow consists of extracting gradients from images(s), compositing/processing extracted gradients, and then solving the Poisson equation for the image whose gradients best fit the prescribed field.

Based on this idea, numerous applications have been developed. Shadows and lighting in images can be removed by dampening or selecting appropriate gradients [11–13]. High dynamic range (HDR) images can be tone-mapped to low dynamic range representations by attenuating strong gradients [14]. Visible discontinuities in panoramic image stitching can be removed by zeroing or blending color gradients across patch boundaries [1, 15], and similar approaches have been used for object cloning [5], texture transferring [5], photomontaging [16], and image matting [17].

Apart from synthesizing plausible images, the Poisson equation has also been used for stylization and content creation. Color images can be converted into grayscale images in a saliency-preserving fashion by fusing chrominance and luminance gradients [18]. Colorization and tone adjustment can be done by propagating sparse user edits [19,20]. Non-photorealistic and abstract rendition of images can also be achieved by edge-aware gradient flattening [20, 21]. Interactive image painting systems that operate directly on the gradient domain have also been developed [22, 23].

Geometry Processing

Due to its close relationship with the heat equation, earlier applications of the Poisson equation to geometry-processing focused on surface fairing [24–26]. As shown in [26], geometry smoothing can be formulated as a process of heat diffusion. The idea is extended in later works which make this process more feature-aware. These include techniques for anisotropic diffusion that redefine inner-products on tangent spaces using curvature information [27–30], and techniques based on optimization frameworks leveraging feature points from the original geometry to "anchor" the smoothing process [31–33].

As in image processing, the Poisson equation has also become a key component in mesh editing systems. Geometric details can be transferred between meshes by blending differential coordinates [34]. Sparse local edits of geometry can be adapted and propagated smoothly to the whole surface by solving the Poisson equation for new coordinate functions [6, 35–39]. We refer readers to [40] for an extensive survey of the subject.

The Laplace-Beltrami operator has been widely used in shape analysis. Using the analogy between the eigenvectors of Laplacian and the Fourier basis, one can realize frequency-based signal processing on meshes [24,41]. Leveraging the eigenvalues and/or eigenvectors of the operator, isometry-invariant shape descriptors can be defined [42,43]. Embedding the mesh into a high-dimensional space using the spectrum of the operator [43], one can identify intrinsic symmetries of a shape [44]. Drawing on the relationship with heat diffusion, point-based signatures of geometry [45] and various shape-aware distances [46,47] have also been formulated.

2.2 Numerical Methods

As mentioned previously, solving the Poisson equation on meshes requires a discrete/discretized definition of the Laplace-Beltrami operator. The topic has attracted a great deal of research in the computer graphics community. Graph-based, combinatorial operators have the advantages of simplicity and efficiency [24, 48]. Geometry-driven operators taking angles and areas into account give rise the ubiquitous cotangent-weight Laplacian for triangle meshes [49, 50] and its extension to general polygon tessellations [51]. Recently, an operator based on intrinsic Delaunay triangulation has been proposed that addresses the problem of non-convex weighting [52, 53]. We refer readers to [54] for a study of the tradeoffs between different operators.

Apart from a robust numerical discretization, efficiently solving the arising system of equations is also of essential importance for many applications. As Poisson-like systems are usually sparse, direct sparse solvers have been a popular choice [55–57]. In particular, when the system is symmetric and positive-definite, Cholesky factorization is often favored due to its numerical stability and efficiency [58]. Furthermore, when the system is fixed but needs to be solved repeatedly for a constraint that varies over the course of an application, the factorization cost can be amortized, making these sparse factorization techniques an appealing option [6].

Multigrid methods have also been used [59]. Their low memory usage and ability to dampen low-frequency errors efficiently make them preferable to direct solvers when the problem size is exceedingly large and/or the exact solution is not necessary. To define the hierarchical structure on unstructured domains, "black-box" algebraic multigrid methods [3] that rely solely on information encoded in matrices have been studied [60–62]. Additionally, more geometry-driven approaches that rely on the design of mesh hierarchies have also appeared in geometry-processing applications [2, 27, 38, 63–66].

Chapter 3

Finite-Elements, B-splines, and Multigrid

Given a Riemannian 2-manifold $(\mathcal{M}, \mathsf{g})$, consider a generic case where we want to fit a function u such that the following energy is minimized:

$$\int_{\mathcal{M}} \lambda(u-f)^2 + \|\nabla_{\mathsf{g}} u - \vec{s}\|_{\mathsf{g}}^2 \, d\mu_{\mathsf{g}} \tag{3.1}$$

Here, $\vec{s}$ is a (tangent) vector field guiding the gradient of u, and f is a real valued function constraining the values of u. The constant, λ, controls the tradeoff between the gradient and value constraints. We use the metric tensor, g, to define the gradient operator ∇_{g} in the tangent space, the norm $\|\cdot\|_{\mathsf{g}}$ of tangent vectors, and the area measure $d\mu_{\mathsf{g}}$ on $\mathcal{M}$.

We will assume that either $\mathcal{M}$ is closed or u satisfies appropriate boundary conditions (see below). Applying Stokes' theorem, one obtains the Euler-Lagrange formulation giving the minimizer of Equation 3.1, known as the solution to the screened-Poisson equation:

$$(\lambda - \Delta_{\mathtt{g}})u = f - \nabla_{\mathtt{g}} \cdot \vec{s} \tag{3.2}$$

where $\nabla_{\mathtt{g}}\cdot$ is the divergence operator on $(\mathcal{M}, \mathtt{g})$, and $\Delta_{\mathtt{g}} := \nabla_{\mathtt{g}} \cdot \nabla_{\mathtt{g}}$ is the *Laplace-Beltrami* operator: the generalization of the Laplace operator $\Delta := \nabla \cdot \nabla$ to $(\mathcal{M}, \mathtt{g})$.

Our goal is to develop a numerical framework for solving Equation 3.2 efficiently. Note that Equation 3.2 has been central to numerous geometry-processing applications, as it can be used for general-purpose gradient fitting (as in mesh editing, texture stitching, and parameterization) and, due to the close relationship between the Laplace operator and the heat equation, it can be used to perform time-integration in diffusion processes (as in curvature flow and mesh fairing).

In this chapter, we review the mathematical background relevant to the development of our framework. This helps set up the context facilitating our discussion in the following chapters. In particular, we review the finite elements discretization for Poisson-like problems (Section 3.1), refinable B-splines for constructing multiresolution function spaces (Section 3.2), and the multigrid method for solving linear systems (Section 3.3).

3.1 Finite-Elements

The finite elements method (FEM) is a commonly used technique for solving partial differential equations. In this section, we review the FEM discretization in the context of solving Poisson-like systems on meshes (i.e., Equation 3.2).

In general, we assume that the solution to the screened-Poisson equation resides in the space of twice differentiable functions on $\mathcal{M}$. Since $C^2(\mathcal{M})$ is infinite-dimensional, the problem is made tractable by first choosing a finite-dimensional subspace $\mathcal{F} \subset C^2(\mathcal{M})$ and then searching for the closest solution within $\mathcal{F}$. Under this setting, Equation 3.2 needs to be *projected* onto $\mathcal{F}$. To do this, we first define the projection operator, $\pi : C^2(\mathcal{M}) \to \mathcal{F}$, such that for $f \in C^2(\mathcal{M})$

$$\Big\langle\ \pi(f)\ ,\ b\ \Big\rangle = \Big\langle\ f\ ,\ b\ \Big\rangle\ ,\ \forall b \in \mathcal{F}$$

where the inner-product, $\langle \cdot, \cdot \rangle : C^2(\mathcal{M}) \times C^2(\mathcal{M}) \to \mathbb{R}$, on a compact Riemannian manifold $(\mathcal{M}, g)$ is most naturally defined by

$$\Big\langle\ a\ ,\ b\ \Big\rangle = \int_{\mathcal{M}} a \cdot b\ d\mu_{\mathrm{g}}.$$

Applying π to Equation 3.2, one obtains $\pi\Big((\lambda - \Delta_{\mathrm{g}})u\Big) = \pi\Big(f - \nabla_{\mathrm{g}} \cdot \vec{s}\Big)$, or equivalently

$$\Big\langle\ (\lambda - \Delta_{\mathrm{g}})u\ ,\ b\ \Big\rangle = \Big\langle\ f - \nabla_{\mathrm{g}} \cdot \vec{s}\ ,\ b\ \Big\rangle\ ,\ \forall b \in \mathcal{F} \tag{3.3}$$

When $\mathcal{F}$ is the span of functions $\{b_1, \ldots, b_n\}$, in order to satisfy Equation 3.3 it is *sufficient* to have:

$$\Big\langle\ (\lambda - \Delta_{\mathrm{g}})u\ ,\ b_i\ \Big\rangle = \Big\langle\ f - \nabla_{\mathrm{g}} \cdot \vec{s}\ ,\ b_i\ \Big\rangle\ ,\ \forall i \in \{1, \ldots, n\}$$

At this point, the challenge of solving Poisson-like systems has been reduced to solving an $n \times n$ linear system for the coefficient vector, $\mathbf{u} = [\mathrm{u}_1, \ldots, \mathrm{u}_n]^\dagger$, such that

$$(\lambda M + L)\mathbf{u} = \mathbf{f} + \mathbf{s} \tag{3.4}$$

with M the *mass* matrix

$$M_{ij} = \int_{\mathcal{M}} b_i \cdot b_j \; d\mu_{\mathtt{g}} \;, \tag{3.5}$$

L the *stiffness* matrix (where $\mathtt{g}(\cdot,\cdot)$ is the inner-product defined by $\mathtt{g}$)

$$L_{ij} = \; -\int_{\mathcal{M}} \Delta_{\mathtt{g}} b_i \cdot b_j \; d\mu_{\mathtt{g}} \; = \int_{\mathcal{M}} \mathtt{g}\Big(\nabla_{\mathtt{g}} b_i, \nabla_{\mathtt{g}} b_j\Big) \; d\mu_{\mathtt{g}} \;, \tag{3.6}$$

$\mathbf{f}$ the projected value constraint

$$\mathbf{f}_i = \int_{\mathcal{M}} f \cdot b_i \; d\mu_{\mathtt{g}} \;, \tag{3.7}$$

and $\mathbf{s}$ the projected gradient constraint

$$\mathbf{s}_i = \; -\int_{\mathcal{M}} (\nabla_{\mathtt{g}} \cdot \vec{s}) \cdot b_i \; d\mu_{\mathtt{g}} \; = \int_{\mathcal{M}} \mathtt{g}\Big(\vec{s}, \nabla_{\mathtt{g}} b_i\Big) \; d\mu_{\mathtt{g}} \tag{3.8}$$

Here we make a few remarks about the mass and stiffness matrices:

- The second equality in Equation 3.6 (and similarly Equation 3.8) is commonly referred to as the *weak formulation* of the Laplacian and only requires square integrable first-order derivatives. It is derived from Stokes' theorem and holds when $\mathcal{M}$ is closed, or when either the test functions or their normal derivatives vanish at boundaries. Interestingly though, for many gradient-domain applications, we do not really require these conditions for obtaining the minimizer of Equation 3.1. [1]

- When the functions in the spanning set $\{b_1, \ldots, b_n\}$ are linearly independent, the mass matrix is positive definite and the stiffness matrix is positive semi-definite. To see this, first note that any coefficient vector $\mathbf{u}$ represents a function $u \in \mathcal{F}$ by $u = \sum \mathrm{u}_i b_i$. If $\mathbf{u}$ is not a zero vector, then $\mathbf{u}^\dagger M \mathbf{u} = \int_M (u)^2 d\mu_{\mathrm{g}}$ must be positive and $\mathbf{u}^\dagger L \mathbf{u} = -\int_M \|\nabla_{\mathrm{g}} u\|_{\mathrm{g}}^2 d\mu_{\mathrm{g}}$ must be non-negative (it is zero when u is constant on each connected component of $\mathcal{M}$).

- Since the matrices are both symmetric, when the mass matrix is positive-definite, solving the generalized eigenvalue problem, $L\mathbf{v} = \alpha M \mathbf{v}$, gives real eigenvalue/eigenvector pairs $(\alpha_i, \mathbf{v}_i)$, with the eigenvectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ forming an orthonormal basis w.r.t. the inner product defined by the mass matrix. This basis provides a "natural" frequency decomposition of functions defined over meshes [24], analogous to the Fourier basis defined over regular domains.

Example

In the case of triangle meshes, the common choice for test functions b_i are "tent" functions centered at the mesh vertices and supported within each vertex's incident triangles [49]. This configuration results in the area-weight formula for the mass

[1]The minimizer of Equation 3.1 was obtained by integrating the inner-product of gradients, rather than the product of functions and Laplacians. The condition was later turned into the screened-Poisson equation using Stokes' theorem. Here the weak formulation effectively "reverts" the application of Stokes' theorem.

matrix

$$M_{ij} = \begin{cases} \frac{|T_{ij}^1|+|T_{ij}^2|}{12} & \text{if } j \in N(i) \\ \sum_{k\in N(i)} M_{ik} & \text{if } j = i \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

and the cotangent-weight formula for the stiffness matrix

$$L_{ij} = \begin{cases} -\frac{cot\beta_{ij}^1+cot\beta_{ij}^2}{2} & \text{if } j \in N(i) \\ -\sum_{k\in N(i)} L_{ik} & \text{if } j = i \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

where $N(i)$ are the vertices adjacent to vertex i, $|T_{ij}^1|$ and $|T_{ij}^2|$ are the areas of the two triangles sharing edge (i,j), and β_{ij}^1 and β_{ij}^2 are the two angles opposite edge (i,j).

3.2 B-splines

In computer graphics, B-splines have been important for approximation, modeling, and signal analysis. In this section, we focus our review on the *uniform* B-splines in Euclidean space, as their *refinability* will be a cornerstone of the construction of our multi-resolution function spaces.

Definition

In 1D, a B-spline $b^n(x)$ of degree n is defined by convolving a characteristic function $b^0(x)$ with itself n times [67]:

$$b^n(x) = \int_{x-1}^{x} b^{n-1}(p)dp$$

where

$$b^0(x) = \begin{cases} 1 & \text{if } x = [0,1) \\ 0 & \text{otherwise} \end{cases}$$

Using translated and scaled copies of $b^n(x)$, we can define a family of vector spaces, indexed by width h and spanned by:

$$b_i^{n,h}(x) = b^n\left(\frac{x}{h} - i\right) \tag{3.11}$$

In this setting, h can be thought of as the width of a regular 1D grid partitioning the parameter domain into uniform intervals, and i is the offset. We will denote by $V^{n,h}$ the space of functions that are representable using a grid of resolution h:

$$V^{n,h} := \left\{ f(x) \mid f(x) = \sum_{i\in\mathbb{Z}} \mathrm{u}_i \cdot b_i^{n,h}(x) \right\} \tag{3.12}$$

Note that each basis function here has local support and that the formulation guarantees C^{n-1} continuity. Moreover, the n-th order derivatives exist almost everywhere and are *square integrable*, meaning that $V^{n,h}$ is in the n-th order Sobolev space.

Nesting Structure

Apart from being smooth and having compact support, an important property of B-splines is that they are *refinable* . That is,

$$b^n(x) = \sum_{k\in\mathbb{Z}} P^n(k) \cdot b^n(2x - k) \tag{3.13}$$

with the prolongation stencil

$$P^n(k) = \begin{cases} 2^{-n}\begin{pmatrix} n+1 \\ k \end{pmatrix} & \text{if } 0 \le k \le n+1, \\ 0 & \text{otherwise} \end{cases} \tag{3.14}$$

Plugging this back to Equation 3.11, we observe that the spaces of functions defined earlier are *nested*: $V^{n,2h} \subset V^{n,h}$. Hence, any function in $V^{n,2h}$ can be expressed as a linear combination of functions in $V^{n,h}$.

Extending to Higher Dimensions

B-splines can be extended to N dimensions by considering the product of N univariate B-splines (here we overload $b^n : \mathbb{R}^N \to \mathbb{R}$):

$$b^n\Big(x_1, \cdots, x_N\Big) = \prod_{i=0}^{N} b^n(x_i)$$

Similarly, the function spaces can be extended to higher dimensions:

$$V_N^{n,h} := \Big\{ f(x) \mid f(x) = \sum_{i \in \mathbb{Z}^N} \mathrm{u}_i \cdot b_i^{n,h}(x) \Big\}$$

Finally, using Equation 3.13 and 3.14 the refinability of N-dimensional B-splines can be expressed by:

$$b^n(x) = \sum_{k \in \mathbb{Z}^N} P_N^n(k) \cdot b^n\Big(2x - k\Big) \tag{3.15}$$

where k is now a coordinate offset in N dimensions, and the prolongation stencil is obtained by taking a tensor-product of 1D stencils:

$$P_N^n(k) = \prod_{i=0}^{N} P^n(k_i) \tag{3.16}$$

As with the 1D case, this results in nesting $V_N^{n,2h} \subset V_N^{n,h}$

3.3 Multigrid

Multigrid methods are an important family of numerical methods for solving both linear and nonlinear problems. In this chapter, we review the multigrid method for solving linear systems resulting from elliptic PDEs.

In solving linear systems, iterative methods are known to be memory efficient and easy to implement. Unfortunately, without proper preconditioning, the methods converge too slowly on their own. Via a spectral analysis, one can show that many iterative methods such as Jacobi and Gauss-Seidel relaxations dampen high frequency errors *faster* than they dampen low frequency errors [59]. Hence, the idea of the multigrid method is to downsample the residual to a lower resolution domain without losing too much information, so that the errors appear as higher frequency content in the new system (*restriction*). The solution obtained from the lower resolution system is then upsampled to "correct" low frequency errors of the solution at the original resolution (*prolongation*).

Formally, given a system $A^h\mathbf{u}^h = \mathbf{f}^h$ defined on a grid Ω^h of resolution h, the standard 2-level (*V-cycle*) multigrid is implemented as follows:

- Relax the current solution $\mathbf{u}^h$ (using, e.g., Gauss-Seidel iterations).
- Compute the residual: $\mathbf{r}^h = \mathbf{f}^h - A\mathbf{u}^h$.
- Downsample $\mathbf{r}^h$ to the coarser Ω^{2h} using the *restriction* stencil: $\mathbf{r}^{2h} = R_h^{2h}\mathbf{r}^h$.
- Solve for the correction, $\tilde{\mathbf{u}}^{2h}$, such that $A^{2h}\tilde{\mathbf{u}}^{2h} = \mathbf{r}^{2h}$.

- Upsample $\tilde{\mathbf{u}}^{2h}$ back to Ω^h using the *prolongation* stencil: $\tilde{\mathbf{u}}^h = P^h_{2h}\tilde{\mathbf{u}}^{2h}$.
- Update the solution: $\mathbf{u}^h \leftarrow \mathbf{u}^h + \tilde{\mathbf{u}}^h$.
- Relax the solution $\mathbf{u}^h$ again.

Notice that the above algorithm is general. The main challenge in using it is defining the lower resolution system, A^{2h}, and the operators for performing restriction/prolongation, R^{2h}_h/ P^h_{2h}.

Restriction and Prolongation

While there is no specific procedure for defining a lower resolution system, in general the two variational properties should be satisfied, to ensure good convergence rates of the solver [59]:

- $A^{2h} = R^{2h}_h \cdot A^h \cdot P^h_{2h}$
- $R^{2h}_h = (P^h_{2h})^\dagger$

The first property is commonly referred to as the *Galerkin Condition* that constrains the definition of coarser systems, while the second property states that the restriction and prolongation operators should be transposes of each other.

Example

On a regular domain (e.g., images), a common practice for solving the Poisson equation is to use uniform B-splines to define a multi-resolution function space,

$\{V_N^{n,h}\}$, and then solve the resulting linear system (Equation 3.4) using the multigrid method. Under this setting, the prolongation matrix is defined in terms of the coefficient stencil (Equation 3.16) and the restriction matrix is its transpose. In particular, one can show that the lower resolution system defined using the coarser grid automatically satisfies the Galerkin Condition.

Chapter 4

Grid-based Finite Elements System

In this chapter, we introduce our grid-based finite elements system for solving Poisson-like problems on meshes using a multigrid solver. We will describe the general approach (Section 4.1), present implementation details (Section 4.2), and show the results of several experiments (Section 4.3).

4.1 Approach

Using finite elements to formulate a Poisson-like problem requires choosing a finite-dimensional function space (Chapter 3). For triangle meshes, this is often done by defining the spanning functions to be the tent functions over the mesh vertices. These functions are piecewise linear and are supported within the one-ring of each vertex. Using Equation 3.5 and 3.6, one obtains the area-weight formula for the mass matrix and the cotangent-weight formula for the stiffness matrix [49]. Although the functions adapt to the sizes of the triangles, the linear system is tied to the tessellation of the mesh and the function space does not come equipped with a multigrid structure.

In this work, we pursue an alternate approach in which 3D test functions are chosen independent of the mesh. In particular, we start with the multi-resolution function space, $V_3^{n,h}$, described in Chapter 3, and then consider its restriction to the mesh to obtain the function space $V_{3,\mathcal{M}}^{n,h}$. That is, a function f is within $V_{3,\mathcal{M}}^{n,h}$ *if and only if* there exists a function g within $V_3^{n,h}$, such that the two functions take identical values on $\mathcal{M}$:

$$f \in V_{3,\mathcal{M}}^{n,h} \iff \exists g \in V_3^{n,h} \text{ s.t. } g(p) = f(p)\ \forall p \in \mathcal{M}.$$

Note that our function space $V_{3,\mathcal{M}}^{n,h}$ (defined on meshes) has a nesting structure directly inherited from the function space $V_3^{n,h}$ (defined on $\mathbb{R}^3$). To see this, consider a function $f \in V_{3,\mathcal{M}}^{n,2h}$: There exists a function $g \in V_3^{n,2h}$ that is equal to f on $\mathcal{M}$. Since $V_3^{n,2h} \subset V_3^{n,h}$, g must also be in $V_3^{n,h}$. It follows that f is in $V_{3,\mathcal{M}}^{n,h}$ as well.

As a result, we obtain a family of function spaces on meshes that do not depend on mesh tessellation and are equipped with a multi-resolution structure.

Restriction and Prolongation

Let $\{b_1^{n,h}, \ldots, b_m^{n,h}\}$ be a basis for $V_3^{n,h}$ and let $\phi : V_3^{n,h} \to V_{3,\mathcal{M}}^{n,h}$ be the linear operator taking functions from $V_3^{n,h}$ to functions in $V_{3,\mathcal{M}}^{n,h}$, such that $\phi(g) = f$ if and only if $g(p) = f(p)\ \forall p \in \mathcal{M}$. Thus, $\{\phi(b_1^{n,h}), \ldots, \phi(b_m^{n,h})\}$ is a set of functions spanning $V_{3,\mathcal{M}}^{n,h}$. Note that this is *not* a basis of $V_{3,\mathcal{M}}^{n,h}$ since the functions may be linearly dependent (e.g., when the support of $b_i^{n,h}$ does not overlap $\mathcal{M}$ then $\phi(b_i^{n,h}) = 0$ as a function on $\mathcal{M}$).

We observe that the same prolongation/restriction stencil used for performing the refinement/coarsening of $\{b_1^{n,h}, \dots, b_m^{n,h}\}$ in the regular 3D case can still be used for performing the refinement/coarsening of $\{\phi(b_1^{n,h}), \dots, \phi(b_m^{n,h})\}$. To see this, consider a function $f \in V_{3,\mathcal{M}}^{n,2h}$: It can be expressed (possibly not uniquely) as a linear combination of $\{\phi(b_1^{n,2h}), \dots, \phi(b_m^{n,2h})\}$ and each $\phi(b_i^{n,2h})$ can be prolonged to $V_{3,\mathcal{M}}^{n,h}$ by

$$\phi(b_i^{n,2h}) = \phi\left(\sum_k P_i(k) \cdot b_i^{n,h}\right) = \sum_k P_i(k) \cdot \phi\left(b_i^{n,h}\right)$$

where P_i is the ordinary stencil for prolonging $b_i^{n,2h}$.

4.2 Implementation

In this section, we describe how the approach is implemented in practice. We will explain the setup of our test functions, the approach for computing system integrals, the approach for efficiently downsampling/upsampling the system, and a parallel implementation of the multigrid solver.

Choosing Test Functions

Independent of the input mesh, we first set up a regular voxel grid partitioning the Euclidean 3-space into voxels of width $h = 1/2^d$. (We will refer to such grid as *a* *voxel grid of depth d.*) First-order, tensor-product, 3D B-splines are then centered at each grid corner and are scaled so that they are supported within the eight adjacent voxels.

Next, we *embed* the mesh within the voxel grid by translating and scaling the mesh so that it fits into the unit cube $[0,1] \times [0,1] \times [0,1] \subset \mathbb{R}^3$ (the resolution of the function space is therefore made independent of the scale of the mesh, though we do maintain the translation and scales so that subsequent computation can be adjusted to be in the original coordinate frame.). Using grids of successively finer resolutions and restricting B-splines as described in Section 4.1, we obtain a multi-resolution hierarchy of function spaces on the mesh.

Note that, though we could use all of the grid corners to define test functions, only the B-splines whose support overlaps the mesh contribute to the system. Thus, to reduce the linear dependence (and the dimension) of our spanning set, we discard the grid corners defining B-splines not supported on the mesh (see Figure 4.1). To this end, we build an *adaptive* octree around the mesh and only consider the corners of the octree cells.

Finally, for building and solving the linear system in the later stages, we need a way to track and index the B-splines. We do this by leveraging the octree data structure. As an octree indexes grid cells rather than corners, we associate each B-spline with the octree cell in the front, top, right of its support (Figure 4.2, left). Since such cells do not necessarily exist (because the octree cells were only created

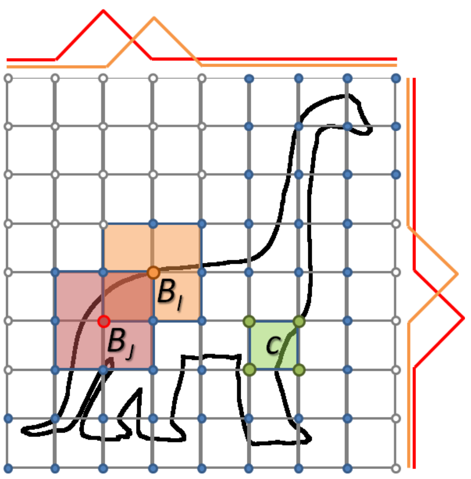


Figure 4.1: Each B-spline is supported within eight adjacent cells (orange and red regions). Each cell is in the support of the B-splines located at its corners (green region). B-splines whose support does not overlap the geometry do not contribute to the system and therefore are discarded (white dots).

when they intersected the mesh), we pad the missing cells into the current octree so that all supported B-splines can be properly indexed.

Here we make two observations. First, if a node c^l at depth l has to exist, so does its parent node c^{l-1} at depth $l-1$. This is because the support of the B-spline indexed by c^l *must be a subset* of the support of the B-spline indexed by c^{l-1} (see Figure 4.2). Second, within a cell c all points $p \in c \cap \mathcal{M}$ are contained within the support of exactly *eight* B-splines. This regularity facilitates vectorizing/parallelizing the computation.

Computing Integrals

Having set up the test functions, we need to compute the coefficients defining the Poisson system (Equations 3.5 through 3.8). To this end, we use a quadrature

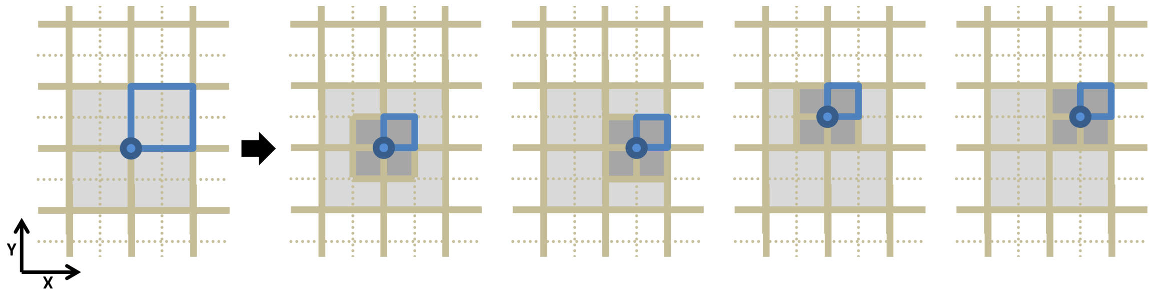


Figure 4.2: Indexing B-splines by octree nodes. Each octree node (blue outlined) indexes the B-spline (whose support is colored light gray) centered at its back, bottom, left corner (*left*). Note that this support covers the supports of the B-splines (colored dark gray) indexed by the child nodes (*middle to right*). Thus, if a child node indexes a B-spline whose support intersects the mesh, so do its ancestors.

approach summing up integral contributions from appropriate sampling positions:

$$\int_{\mathcal{M}} f(p)\, dp \;\approx\; \sum_{p\in P} f(p)\; w(p)$$

where $P \subset \mathcal{M}$ is the set of quadrature points and $w : P \to \mathbb{R}$ gives the associated quadrature weights.

To obtain P and ω, we observe that the B-splines are strictly *polynomial* within each octree cell. Thus, we *clip* the triangles of the embedded mesh by the faces of the octree cells, so that each triangle of the clipped mesh is fully contained within one (finest resolution) octree cell. This ensures that the B-splines are polynomial on each triangle and hence a simple quadrature formula can be used for defining P and ω.

Using this approach, we compute the system coefficients by summing up integral contribution from grid cells:

$$M_{ij} = \sum_{c\in C} \sum_{t\in T(c)} \sum_{p\in P(t)} b_i(p) \cdot b_j(p) \cdot \omega(p) \tag{4.1}$$

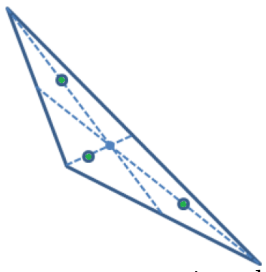
$$L_{ij} = \sum_{c\in C} \sum_{t\in T(c)} \sum_{p\in P(t)} \mathtt{g}\Big(\nabla_{\mathtt{g}} b_i(p), \nabla_{\mathtt{g}} b_j(p)\Big) \cdot \omega(p) \tag{4.2}$$

$$\mathbf{f}_i = \sum_{c\in C} \sum_{t\in T(c)} \sum_{p\in P(t)} f(p) \cdot b_i(p) \cdot \omega(p) \tag{4.3}$$

$$\mathbf{s}_i = \sum_{c\in C} \sum_{t\in T(c)} \sum_{p\in P(t)} \mathtt{g}\Big(\vec{s}(p), \nabla_{\mathtt{g}} b_i(p)\Big) \cdot \omega(p) \tag{4.4}$$

where C stands for the collection of all grid cells, $T(c)$ returns the triangles contained within cell c, and $P(t)$ is the set of quadrature points associated with triangle t. (Note that, in practice, the summation only needs to be taken over the subset of grid cells contained in the support of b_i and b_j.) We summarize the algorithm in Algorithm 4.1 and 4.2, where NeighborCorner(p) returns the eight octree cells whose associated B-splines contain p within their support. Here, $\nabla_{\mathtt{g}} b_i(p)$ is computed by projecting $\nabla b_i(p)$ onto the subspace of $\mathbb{R}^3$ that coincides with the tangent space of the embedded $\mathcal{M}$. That is, $\nabla_{\mathtt{g}} b_i(p) \equiv \nabla b_i(p) - \langle n(p), \nabla b_i(p)\rangle \cdot n(p)$, where $n(p)$ is the surface normal at p.

In theory, we would need to use quadrature formulae designed for sixth degree polynomials (since each entry in the mass matrix is obtained by multiplying two trilinear functions). This, however, requires using at least 11 cubature points per triangle [68], posing a significant cost for high resolution meshes.



In practice, we have found that using an approximate formula is sufficient and does not lead to perceptible errors. In our applications, we use the 3-point formula [69] that places three equally weighted samples on the mid-points of the line segments connecting the vertices of a triangle to its centroid, as visualized by the green dots in the inset.

```
**Algorithm I**: SetSystemMatrices()
$M$ , $L \leftarrow 0$
for $c \in C$ , $t \in T(c)$ , $p \in P(t)$
    for $i \in \text{NeighborCorner}(p)$
        for $j \in \text{NeighborCorner}(p)$
            $M_{ij} \leftarrow M_{ij} + b_i(p) \cdot b_j(p) \cdot w(p)$
            $L_{ij} \leftarrow L_{ij} + \mathsf{g}(\nabla_{\mathsf{g}} b_i(p), \nabla_{\mathsf{g}} b_j(p)) \cdot w(p)$
return ( $M$ , $L$ )
```

Algorithm 4.1: Algorithm for computing the system matrices.

```
**Algorithm II**: SetConstraintVectors()
$\mathbf{f}$ , $\mathbf{s} \leftarrow 0$
for $c \in C$ , $t \in T(c)$ , $p \in P(t)$
    for $i \in \text{NeighborCorner}(p)$
        $\mathbf{f}_i \leftarrow \mathbf{f}_i + f(p) \cdot b_i(p) \cdot w(p)$
        $\mathbf{s}_i \leftarrow \mathbf{s}_i + \mathsf{g}(\vec{s}(p), \nabla_{\mathsf{g}} b_i(p)) \cdot w(p)$
return ( $\mathbf{f}$ , $\mathbf{s}$ )
```

Algorithm 4.2: Algorithm for computing the constraint vectors.

Downsampling the System

To obtain coarser systems, one can carry out integration repeatedly at different grid resolutions. However, this can be inefficient because the cost of integration depends only on the number of quadrature points assigned (since each quadrature point is in the support of eight basis functions and hence contributes to exactly 64 matrix coefficients, regardless of the resolution of the function space).

Instead, one could leverage the fact that the system satisfies the Galerkin condition, $L^{l-1} = (P^l_{l-1})^\dagger \cdot L^l \cdot (P^l_{l-1})$, with P^l_{l-1} the prolongation matrix defined using the prolongation stencil (Chapter 3). As a result, downsampling the linear system can be implemented using two matrix-matrix multiplications. Although the matrices are sparse, the number of non-zero entries in each row is not uniform, ranging between 8 and 27 (depending on how the surface passes through the nearby voxels). In practice, we have found that such an approach is neither cache nor parallelization friendly.

For better efficiency, we leverage the regularity of the grid structure in our implementation, using the fact that the system coefficients can be computed by summing up integral contribution from voxels (a process commonly referred to as *finite elements assembly*). In particular, we iteratively downsample the per-voxel integrals to the coarser levels and then assemble the systems from the downsampled integrals. The advantage of the approach, as also observed by [70], is that each integral at coarse levels can be computed (in parallel) with a cache friendly memory access pattern, because the computation within each parent cell can be performed by regularly accessing integrals from $2 \times 2 \times 2$ child voxels.

As an example, consider the computation of the mass matrix (Equation 4.1). Each entry is computed by summing up contribution from individual octree cells:

$$M_{ij} = \sum_{c} Int^M(c, i, j) \tag{4.5}$$

where $Int^M(c, i, j)$ is the contribution of cell c to the value of the integral M_{ij},

$$Int^M(c, i, j) = \sum_{t \in T(c)} \sum_{p \in P(t)} b_i(p) \cdot b_j(p) \cdot w(p)$$

Now, observe that $Int^M(c, i, j)$ can also be computed by downsampling the integral contribution from c's $2 \times 2 \times 2$ child voxels $\{c'\}$

$$\begin{aligned} Int^M(c, i, j) &= \sum_{c' \subset c} \sum_{t \in T(c')} \sum_{p \in P(t)} \left(\sum_{i'} P_i(i') \cdot b_{i'}(p) \right) \left(\sum_{j'} P_j(j') \cdot b_{j'}(p) \right) \omega(p) \\ &= \sum_{c' \subset c, i', j'} P_i(i') \cdot P_i(j') \cdot \left(\sum_{t \in T(c')} \sum_{p \in P(t)} b_{i'}(p) \cdot b_{j'}(p) \cdot \omega(p) \right) \\ &= \sum_{c' \subset c, i', j'} P_i(i') \cdot P_i(j') \cdot Int^M(c', i', j') \end{aligned} \tag{4.6}$$

As each cell is in the support of exactly eight test functions, in our implementation we first compute the *element matrix*

$$E_{mn} = \begin{cases} Int^M\Big(\lfloor m/8 \rfloor, idx(m), idx(n)\Big) & \text{if } \lfloor m/8 \rfloor = \lfloor n/8 \rfloor \\ 0 & \text{otherwise} \end{cases}$$

where $idx(m)$ returns the index of the B-spline centered at the (m mod 8)-th corner of cell $\lfloor m/8 \rfloor$. (Note that E is a block diagonal matrix where each block corresponds to one cell and consists of 8×8 non-zero entries.) Coarse element matrices are then computed by downsampling element matrices from finer levels using the stencil described by Equation 4.6. The computation of each coarse entry is performed *in parallel* as a value *gathering* process, regularly accessing entries from $2 \times 2 \times 2$ finer cells. The memory access pattern is optimized because A) entries from one cell are grouped into a block, and B) the blocks are sorted by the z-curve order of their corresponding cells when we construct the element matrix.

Finally, to assemble the system coefficients from the element matrix, another value gathering process is performed. This is implemented in parallel for all rows of the matrix. For each row i, we first pre-allocate a table of size $3 \times 3 \times 3$, equal to the maximum number of non-zero entries per row. Next, we walk through the voxels in the support of b_i (there are at most $2 \times 2 \times 2$ such voxels existing in the octree), and accumulate the entries associated with b_i into the pre-allocated table. The table is then collapsed into a compact list by getting rid of zero entries. [1]

Solver Parallelization

We now describe the parallel implementation of our multigrid solver. Looking at the standard V-cycle algorithm reviewed in Section 3.3, we observe that four kinds of operations are needed in order to implement a multigrid solver:

- Computing residuals (matrix-vector multiplication).
- Constructing coarser systems (previously discussed).

[1]We use a common "list of lists" data structure for storing sparse matrices, where each row is stored as a list of entries, and each entry consists a column index and a non-zero value.

- Restricting/prolonging residuals/coefficients (matrix-vector multiplication).
- Relaxing coefficients (?).

As matrix-vector multiplication is easily parallelizable and construction of lower resolution systems has already been discussed, we now focus on the parallelization of relaxation.

In this work, we implement the Gauss-Seidel iteration in parallel by leveraging the regularity of the grid structure. The idea is inspired by the work on domain decomposition for parallel multilevel methods [71].

We first partition the test functions into $2^d + 1$ horizontal *slices*, so that test functions in slice z are centered at grid corners of height $z/2^d$. Given the number of threads, k, we further decompose these slices into k groups $0 = z_0 < z_1 < \cdots < z_{k-1} < z_k = 2^d$ so that the number of test functions belonging to slices $[z_i, z_{i+1})$ is as uniform as possible.

Note that since we use first-order B-splines as test functions, as long as two threads are not simultaneously processing slices that are adjacent to each other, no memory conflict can occur. Thus, for a single Gauss-Seidel iteration, we let thread k relax coefficients of test functions belonging to slices $[z_k, z_{k+1} - 1)$ independently. Then, only after a call that synchronizes all threads, we let each thread proceed to relax coefficients of test functions belonging to slices z_k.

It is also possible to perform multiple Gauss-Seidel iterations in parallel. To do this, we expand the span of slices assigned to each thread (so the domains partially

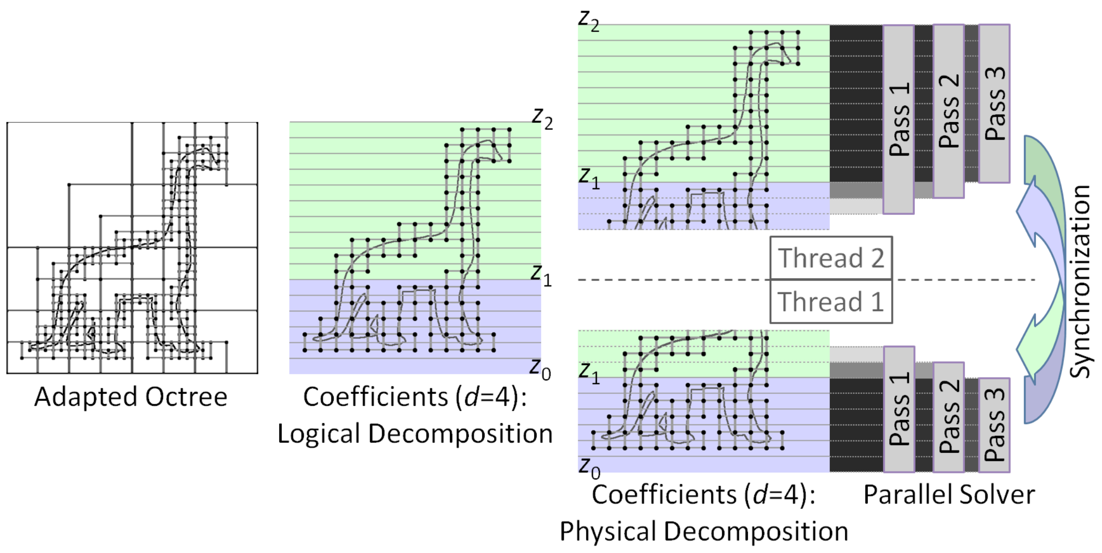


Figure 4.3: Parallelization of Gauss-Seidel Relaxation. By decomposing the solution coefficients into overlapping blocks and shrinking the vertical extent of relaxed coefficients on subsequent updates, threads can perform multiple updates in parallel, without having to synchronize coefficient values between passes.

overlap each other at boundaries) and shrink the span at the end of each iteration. Consider the case where we want to perform I Gauss-Seidel iterations: At iteration i, each thread is restricted to operate on slices $[z_k - I + i, z_{k+1} + I - i)$, while always keeping slices $[z_k - I, z_{k+1} + I)$ in the local memory of each thread. The arrangement ensures that when we want to update a coefficient for the i-th time, all its neighbors will have been updated at least $i - 1$ times, resulting in a valid implementation of Gauss-Seidel iterations. A visualization of the approach is shown in Figure 4.3, where three Gauss-Seidel iterations are performed in parallel without synchronization between iterations.

Function Evaluation

After solving for the coefficient vector, $\mathbf{u} = [\mathrm{u}_1, \ldots, \mathrm{u}_n]^\dagger$, one needs to evaluate the resulting function. This is most naturally done by sampling the function value f_i at each vertex i of the input mesh (i.e., $f_i = \sum_j u_j b_j(x_i)$ with x_i the vertex position). The computation can be formulated as a matrix-vector multiplication $\mathbf{f} = E\mathbf{u}$, where $E_{ij} = b_j(x_i)$ is the *evaluation* matrix. As each vertex supports exactly eight basis functions, E is a sparse matrix where each row consists of exactly eight non-zero entries. This regular structure is parallelization friendly and, as we will discuss in the next chapter, can easily be relegated to GPU.

Hat-Basis Incorporation

So far, our discussion has focused on the use of the grid-based system as a stand-alone Poisson solver. However, it is also possible to use it *in conjunction with* the popular, piecewise linear, hat-basis system (resulting in the linear system defined by Equations 3.9 and 3.10). As a majority of graphics applications are developed using the hat-basis for function representation, being able to work with the hat-basis facilitates the integration of our multigrid solver into those applications.

To do this, we first define the finest-level linear system using the hat-basis. Then, on the top of the original grid-based multigrid, an extra restriction operator, R_h^g, is defined in order to transform the constraint from the hat-basis system to the grid-based system; the transpose of R_h^g is used to prolong the solution from the grid-based system to the hat-basis system. Specifically, the modified multigrid solver proceeds as follows:

- Perform a few iterations of Gauss-Seidel relaxation on the hat-basis system.
- Compute the hat-basis residual and transform it into a constraint for the grid-based system using R_h^g.
- Perform a grid-based multigrid solve on the transformed constraint.
- Transform the grid-based solution back to the hat-basis system using $(R_h^g)^\dagger$, in order to correct the hat-basis solution.
- Perform a few iterations of Gauss-Seidel relaxation on the hat-basis system.

Perhaps the most natural choice of R_h^g is to use the transpose of the evaluation matrix (as the coefficients of the hat-basis equal the function values at the mesh vertices). This works when the grid is coarse compared to the input mesh. However, when the grid resolution becomes higher, there will be restricted B-splines whose supports do not contain any vertices, and hence do not receive any residual constraint.

To address this, we propose the use of a *pseudo-Evaluation* matrix that transforms the residual with respect to the hat-basis system defined on the input mesh, to a residual with respect to the hat-basis system defined on the *clipped* mesh, and then to a residual with respect to the Grid-based system:

Here, M_c is the *lumped* mass matrix (obtained by lumping the rows of the original mass matrix to the diagonal) defined using the hat-basis on the input mesh, M_f

$$\underbrace{R_h^g}_{\left\{\substack{\text{Residual}\\ \text{w.r.t.}\\ \text{Grid Basis}}\right\}} := \underbrace{E_f^\dagger}_{\left\{\substack{\text{Residual}\\ \text{w.r.t.}\\ \text{Finer-Hat}}\right\}} \times \underbrace{M_f}_{\left\{\substack{\text{Coefficients}\\ \text{w.r.t.}\\ \text{Finer-Hat}}\right\}} \times \underbrace{P_c^f}_{\left\{\substack{\text{Coefficients}\\ \text{w.r.t.}\\ \text{Coarse-Hat}}\right\}} \times \underbrace{M_c^{-1}}_{\left\{\substack{\text{Residual}\\ \text{w.r.t.}\\ \text{Coarse-Hat}}\right\}}$$

$$\longleftarrow\!\!\!\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\!-\!\!\bullet$$

is the lumped mass matrix defined using the hat-basis on the clipped mesh, P_c^f is the prolongation matrix turning the vertex values of the input mesh into the vertex values of the (refined) clipped mesh, and finally, $E_f^\dagger$ is the transpose of the ordinary evaluation matrix of the input mesh described in the previous section. (Note that, compared to the standard mass matrix, the lumped matrix is efficient to construct and invert.) Since every B-spline has at least one vertex of the clipped mesh on its support, the use of the pseudo-Evaluation matrix helps distribute residuals more evenly.

4.3 Results

In this section, we conduct several experiments evaluating our grid-based finite-element system. We start with a spectral analysis that validates the stability and correctness of our system. We then examine the convergence rate of our multigrid solver and benchmark our performance against other state-of-the-art multigrid solvers.

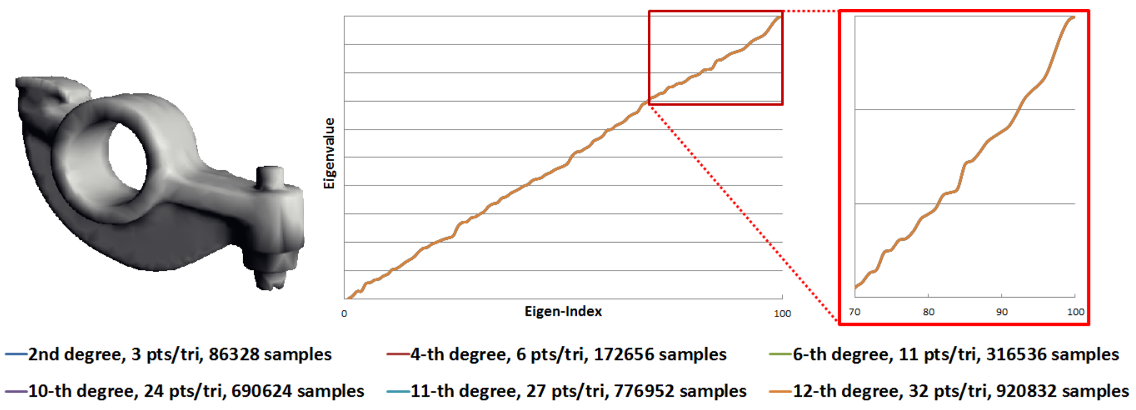


Figure 4.4: Stability of polynomial integrators. We compute the spectrum of the Laplace-Beltrami operator estimated using different integrators. Note that our 3-point integrator produces an identical spectrum as other (more expensive) integrators. The input mesh consists of 6624 triangles (28776 after the mesh is clipped).

4.3.1 Spectral Analysis

The spectral decomposition of the Laplace-Beltrami operator characterizes the frequencies of functions on the surface and has been widely used for analyzing and processing signals on meshes [24, 41], supporting applications such as surface fairing [24], shape matching [42], and mesh editing [41].

In this section, we use spectral analysis as a tool to study the correctness of our finite elements formulation, and to validate the correctness of our implementation. In particular, we solve the generalized eigenvalue problem:

$$L\mathbf{x} = \lambda M\mathbf{x}$$

where λ and $\mathbf{x}$ are the eigenvalues and corresponding eigenvectors. As reviewed in Chapter 3, the positive (semi-)definiteness of L and M guarantees that $\lambda/\mathbf{x}$ are real valued. We use ARPACK [72] to compute the first hundred (smallest) eigenpairs.

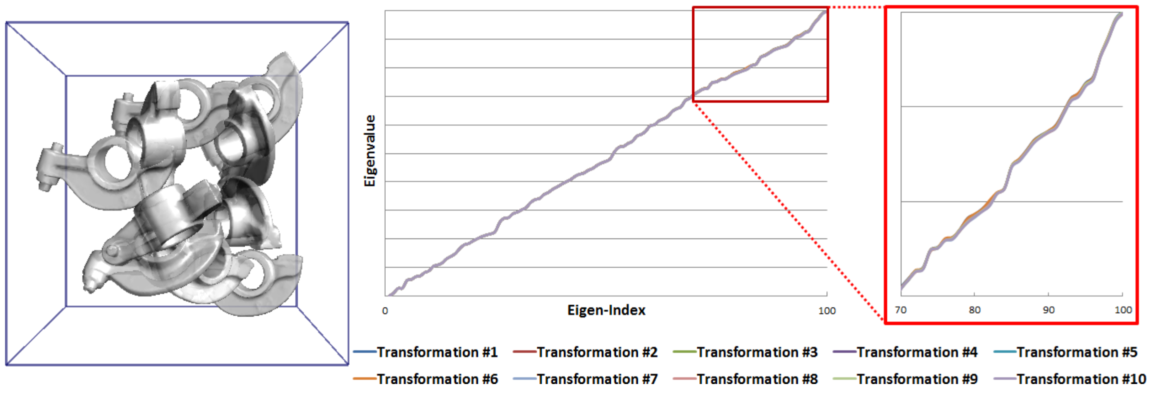


Figure 4.5: Isometry invariance of the estimated Laplace-Beltrami operator. We randomly rotate and translate the model within the voxel grid (*left*). Even though each (rigidly) transformed surface intersects the voxels differently and thus defines a different linear system, the resulting spectra are (almost) identical.

We first analyze the stability of our 3-point polynomial integrator, by comparing the spectrum with the spectra obtained using higher degree integrators. As shown in Figure 4.4, the results are indistinguishable even at the higher frequencies, suggesting a sufficient approximation of the chosen integrator.

Next, we verify that isometry-invariance of the Laplacian is preserved in our system. In particular, we randomly rotate and translate the model within the voxel grid of a fixed resolution (depth 7) and compute the spectrum of each resulting system. As shown in Figure 4.5, the resulting spectra are almost identical. We believe that the slight discrepancy at higher frequencies is due to the facts that: A) As the transformed surfaces intersect the voxels differently, they define systems of different resolutions. B) Using the restriction of 3D B-splines to the surface as test functions improperly couples some function values on disconnected surface regions (we will come back to this issue in Chapter 6).

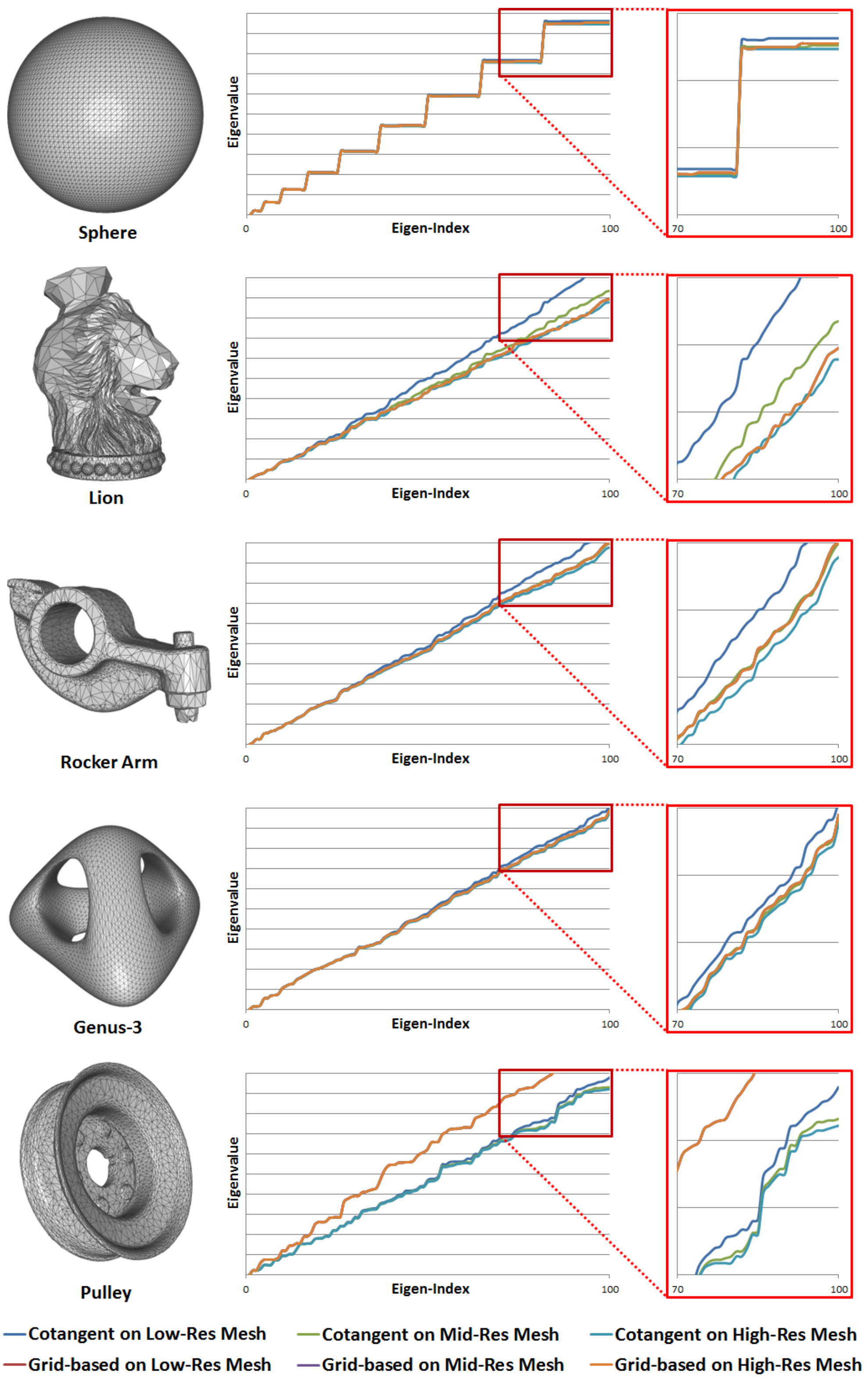


Figure 4.6: Spectrum comparison to the cotangent-weight Laplace-Beltrami operator.

Model	Cotangent-weight Low	Cotangent-weight Mid	Cotangent-weight High	Grid-based (depth 5) Low/Mid/High
Sphere	9,662	38,922	156,242	9,532
Lion	5,720	22,874	91,490	5,528
Rocker Arm	3,312	13,248	52,992	3,226
Genus-3	5,846	23,396	93,596	5,819
Pulley	6,600	26,400	105,600	6,570

Table 4.3: Dimensions of the Laplace-Beltrami operators defined for the different tessellations of the models in Figure 4.6.

Finally, to evaluate the robustness of our finite elements formulation, we compare the spectra obtained from our grid-based Laplacian with those obtained from the cotangent-weight Laplacian. As the dimension of the cotangent-weight system is equal to the number of mesh vertices, we start with meshes whose vertex count matches the dimension of the grid-based system. These meshes are then subdivided to obtain higher resolution cotangent systems defining more faithful spectra.

The results of the experiment are shown in Figure 4.6, with the dimensions of the systems given in Table 4.3. Note that the cotangent-weight systems are sensitive to tessellation: its spectrum converges to the true spectrum when the resolution is increased and the tessellation becomes more regular. In contrast, our grid-based system is agnostic to the tessellation and only considers the surface geometry. More importantly, as indicated in most plots, we are able to estimate the true spectrum more robustly than the cotangent-weight system at the same resolution.

The exception here is the pulley model that consists of narrow cross-sections in 3D, resulting in improper coupling of function values on (geodesically) distant regions

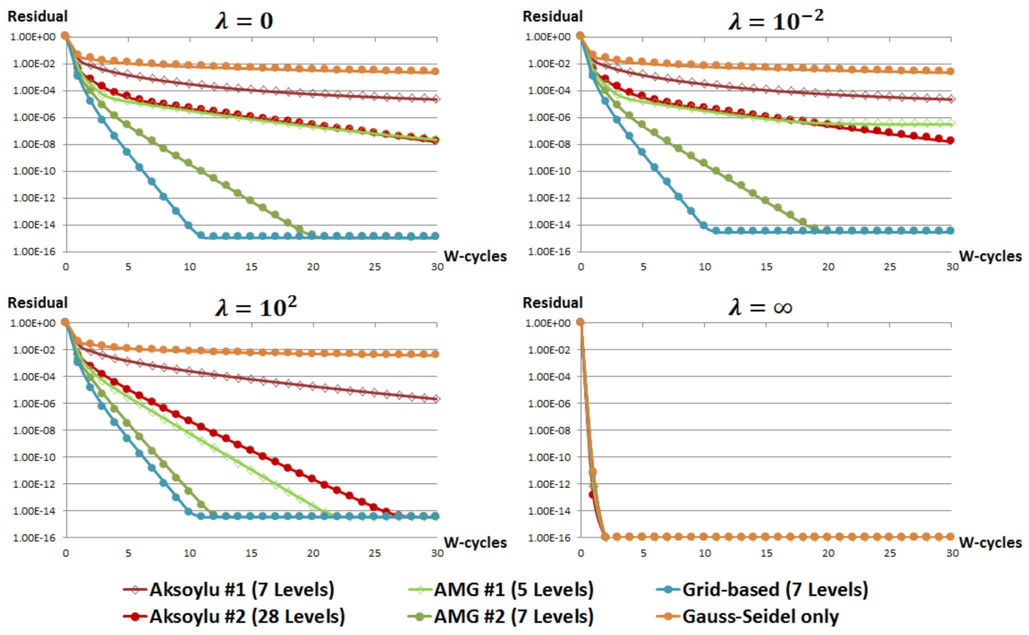


Figure 4.7: Convergence of the multigrid solvers. We solve the screened-Poisson equation for fitting a color function on the mesh. As a standalone solver, we compare our grid-based multigrid against Aksoylu *et al.*'s geometric multigrids [2] and the algebraic multigrids (AMGs) [3,4]. The finest-level linear system is defined using the hat basis.

on the surface. We will investigate this issue more carefully and propose a solution in Chapter 6.

4.3.2 Performance

We now investigate the performance of our multigrid solver. In particular, we compare the convergence rate of our grid-based multigrid to Aksoylu *et al.*'s geometric multigrid [2] and to two implementations of algebraic multigrids (AMGs) [3, 4].

For Aksoylu *et al.*'s multigrid, we use our own implementation, first defining the system using the per-vertex, hat basis functions [49] and then recursively removing

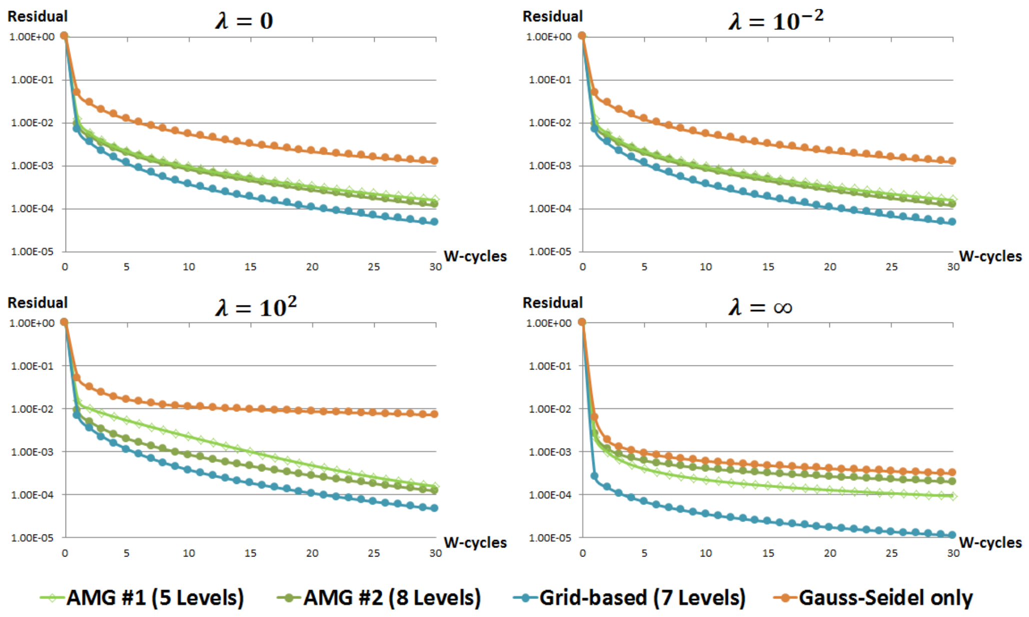


Figure 4.8: Convergence of the multigrid solvers. We solve the screened-Poisson equation for fitting a color function on the mesh. As a standalone solver, we compare our grid-based multigrid against the algebraic multigrids (AMGs) [3, 4]. The finest-level linear system is defined using the grid-based basis.

a maximal independent set of vertices (*Aksoylu* #2). Note that for high resolution meshes, this coarsening process often results in a large number of levels (as each iteration of coarsening only reduces the degree of freedom by a factor of 3/4). Thus, we also consider the configuration (*Aksoylu* #1) that defines restriction/prolongation operators as the composition of successive restriction operators from the original hierarchy, so that the resulting number of levels matches those of the other multigrid implementations used in our experiment.

For the algebraic multigrids, we leverage a modern implementation [73] along with the suggested parameters. In particular, we consider the smoothed-aggregation multigrid [4] (*AMG* #1), which is a popular choice due to its memory efficiency and

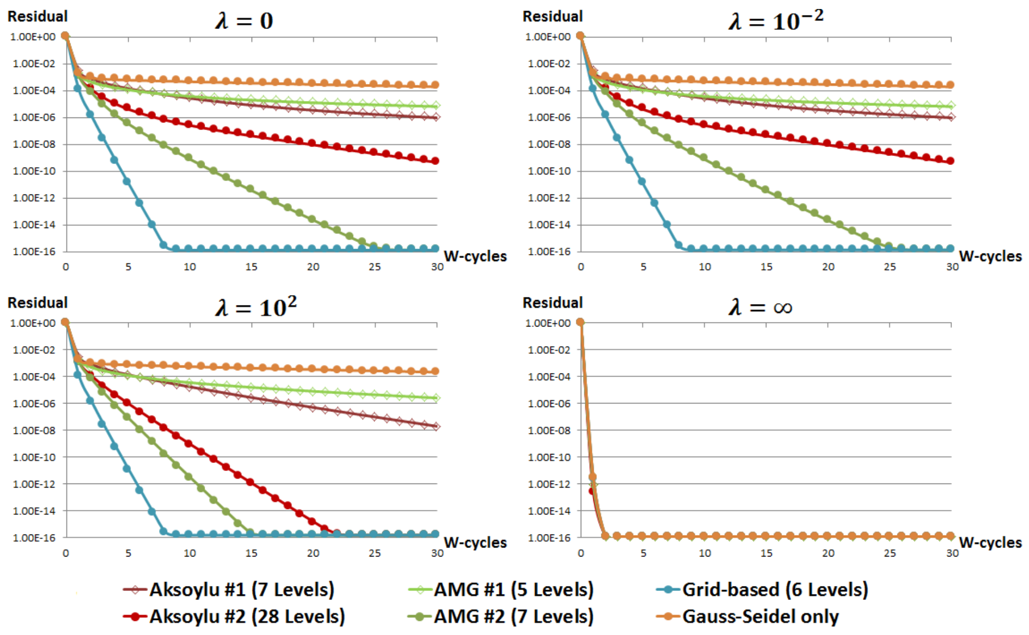


Figure 4.9: Convergence of the multigrid solvers. We solve the screened-Poisson equation for fitting the function consisting of the first hundred eigenfunctions of the Laplacian. As a standalone solver, we compare our grid-based multigrid against Aksoylu *et al.*'s geometric multigrids [2] and the algebraic multigrids (AMGs) [3, 4]. The finest-level linear system is defined using the hat basis.

ease of setup, and the classical Ruge-Stüben AMG [3] (*AMG* #2), which is known to be less aggressive in term of coarsening than the smoothed-aggregation AMG (i.e., resulting in more multigrids levels) and often demonstrates a better convergence rate.

Here we solve a simple function-fitting problem: Given the mesh (see the inset) colored by the function $f : \mathcal{M} \rightarrow [0,1]^3$, we set f as the value constraint and $\vec{s} = \nabla f$ as the gradient constraint. In all the experiments, we use Gauss-Seidel relaxation as the smoother (10 iterations at each level) and a zero vector as the initial guess. To obtain the multigrid hierarchy, we let each coarsening scheme recursively downsample the system matrix until its dimension is smaller than one hundred. A direct solve was performed at the coarsest level.

Note that Aksoylu *et al.*'s multigrid cannot be applied to the system defined using the grid-based basis. We thus apply our grid-based multigrid over the hat-basis system using the pseudo-evaluation matrix described in the previous section. The results of the experiment are presented in Figure 4.7, where we plot the decay of residual norms within the first 30 W-cycles, with multigrid used as standalone solver. As the plots indicate, overall our grid-based multigrid demonstrates a superior convergence rate, consistently reaching the true solution within the first 10 W-cycles. (Note that as λ approaches infinity, the system is dominated by the mass matrix and becomes strongly diagonal dominant, so that the Gauss-Seidel smoother is capable of converging quickly on its own.)

We next evaluate the convergence of our multigrid, as a standalone solver, operating directly on the grid-based function space (Aksoylu *et al.*'s multigrids are thus precluded from the comparison). The results of the experiment are shown in Figure 4.8, where our multigrid outperforms the competing methods, though not as dramatically as in the previous experiments. (The 5×10^{-5} residual in our solution leads to an error no greater than 5×10^{-4} in the reconstructed per-vertex colors, which is well below the displayable resolution and should be acceptable in many graphics applications.)

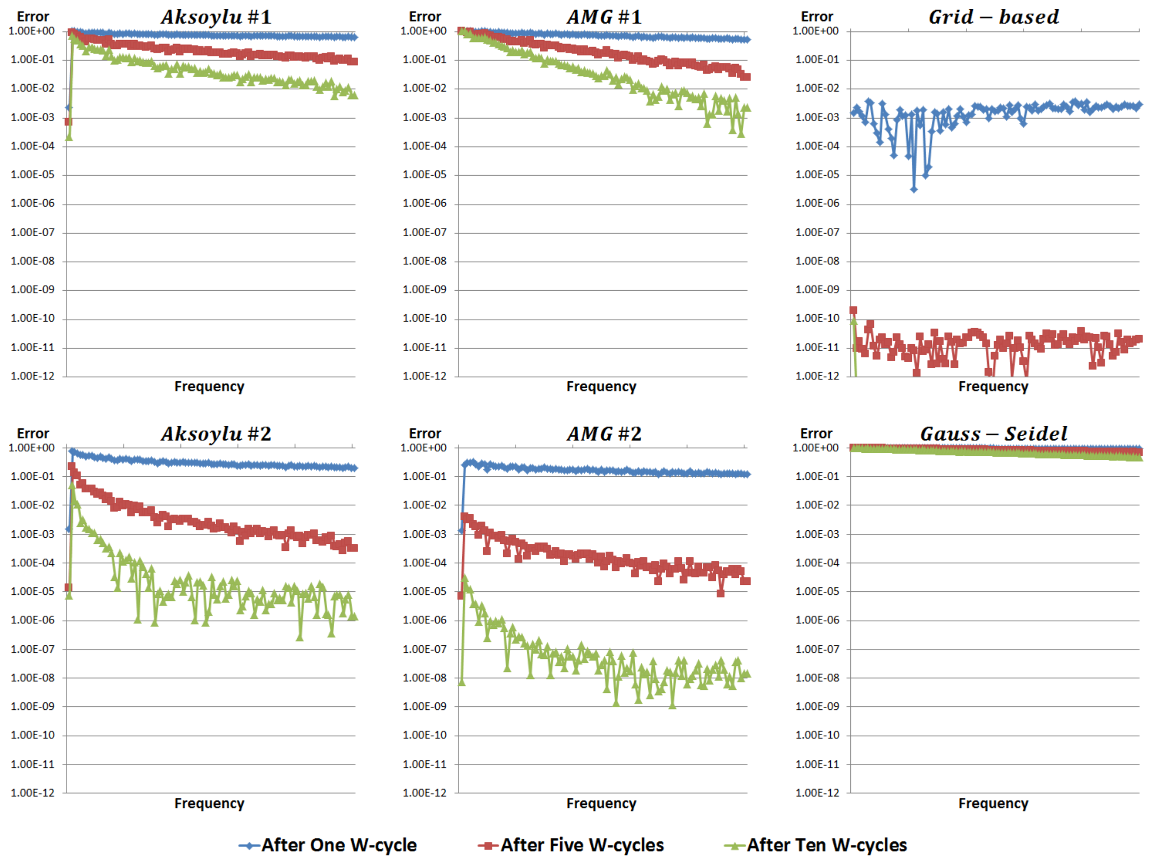
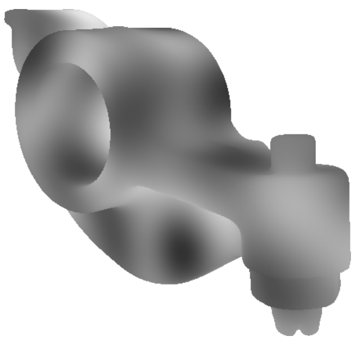


Figure 4.10: The frequency distribution of the errors. Setting the fitted function to be the summation of the first hundred eigenfunctions of the Laplacian, we visualize the frequency coefficients of the errors by projecting the errors onto the eigenfunctions. Note that our multigrid solver more effectively reduces errors from various frequencies.



Finally, we perform one more experiment to help understand the behavior of our multigrid solver in the frequency domain. We use the rocker-arm model and set the fitted values $\mathbf{f}$ to be the summation of the first hundred eigenvectors $\{\mathbf{x}_i\}$ of the Laplacian, as visualized in the inset (here the values are translated and scaled for the purpose of visualization). The advantage of having this setup is that, given reconstructed per-vertex values $\tilde{\mathbf{f}}$, we can observe the frequency distribution of the error, $\mathbf{e} = \mathbf{f} - \tilde{\mathbf{f}}$, by computing the projections of $\mathbf{e}$ onto the $\{\mathbf{x}_i\}$: $\mathbf{e}^\dagger M\mathbf{x}_i$ (that is, the frequency coefficients with respect to the Laplacian eigenbasis).

Figure 4.10 gives the frequency distribution of the errors after one, five, and ten W-cycles. As indicated by the plots, our grid-based multigrid more evenly and effectively suppresses errors across various frequencies, while other configurations suppress low frequency errors less effectively.

Solver Efficiency

Lastly, we investigate the efficiency of our multigrid solver. In Table 4.4, we break down the running time into individual stages. In particular, we observe the time needed for setting up the octree (including mesh clipping), computing the linear system, acquiring the restriction/prolongation operators, setting the evaluation matrix, and performing one W-cycle. As a reference, we also benchmark the running time of the state-of-the-art CHOLMOD solver (compiled by the Intel Math Kernel Library [74] for best performance). To observe the scalability of each method, the experiment is repeated on different resolutions of the dinosaur mesh.[2]

Looking at the results, we see that our grid-based multigrid is comparable to other state-of-the-art methods. Note that, though we have shown in the previous section that we have a better convergence, the time we spend on the initial set-up is significant. As a result, when only solving a static linear system once, the traditional

[2]The higher resolution meshes are obtained by planar 1-to-4 subdivision of the original.

Solver	Computation	Dinosaur Original 213,618 verts.	Dinosaur Subdivided 854,466 verts.	Dinosaur Subdivided 2X 3,417,858 verts.
	Octree Setup	1.25	2.71	7.35
	Grid Linear System	0.31	0.65	1.65
	Hat Linear System	0.21	0.87	3.29
Aksoylu #1	Hat Restriction	3.81	17.67	80.88
	Hat W-cycle	4.80	21.90	102.05
Aksoylu #2	Hat Restriction	4.31	19.89	90.14
	Hat W-cycle	0.60	2.60	10.53
AMG #1	Grid Restriction	0.16	0.16	0.16
	Grid W-cycle	0.65	0.64	0.67
	Hat Restriction	0.12	0.48	1.38
	Hat W-cycle	0.50	2.09	7.00
AMG #2	Grid Restriction	0.21	0.21	0.21
	Grid W-cycle	0.79	0.76	0.72
	Hat Restriction	0.20	0.90	4.25
	Hat W-cycle	0.75	3.79	18.18
Grid	Grid Restriction	0.08	0.08	0.08
	Grid W-cycle	0.66	0.67	0.69
	Evaluation Matrix	0.09	0.31	0.97
	Hat W-cycle	1.01	2.15	6.37
Cholmod	Grid Factorization	2.03	2.05	2.03
	Grid Substitution	0.13	0.12	0.12
	Hat Factorization	0.78	3.82	34.22
	Hat Substitution	0.15	0.46	1.84

Table 4.4: Running time of the multigrid solvers. Here we break down the timing into individual stages. The input mesh is subdivided using a simple mid-point subdivision algorithm. Apart from the multigrids, a direct CHOLMOD solver is also benchmarked for the purpose of reference.

AMG methods are preferable.

As we will demonstrate in Chapter 6, our approach extends to the situations where the linear system is dynamic. While other approaches often need to restart computation in the dynamic setting, our framework allows relegating a large portion of computation to a preprocessing stage, providing a low amortized cost that supports real-time applications.

Discussion and Conclusion

In this chapter, we introduced our grid-based finite elements system. Restricting regular 3D functions to the surface, we have developed a multigrid solver independent of mesh tessellation. We have also described an efficient and parallel implementation of our system. The spectral analysis revealed the correctness of our implementation. The convergence and efficiency analysis demonstrated the competitiveness of our approach as compared with state-of-the-art methods.

In doing so, we have endeavored to reduce linear dependency in the system by adapting an octree so that only the supported B-splines are considered. However, the linear system can still be *semi*-definite when: A) the screening weight is zero, or B) there exists an entirely planar component aligned with one of the axes (i.e., the surface is *not* in *general position*). In the latter case, the linear system is singular because linear dependency exists between basis functions. For example, given two basis functions centered at $(x_1, 0, 0)$ and $(x_2, 0, 0)$ respectively, their values at points

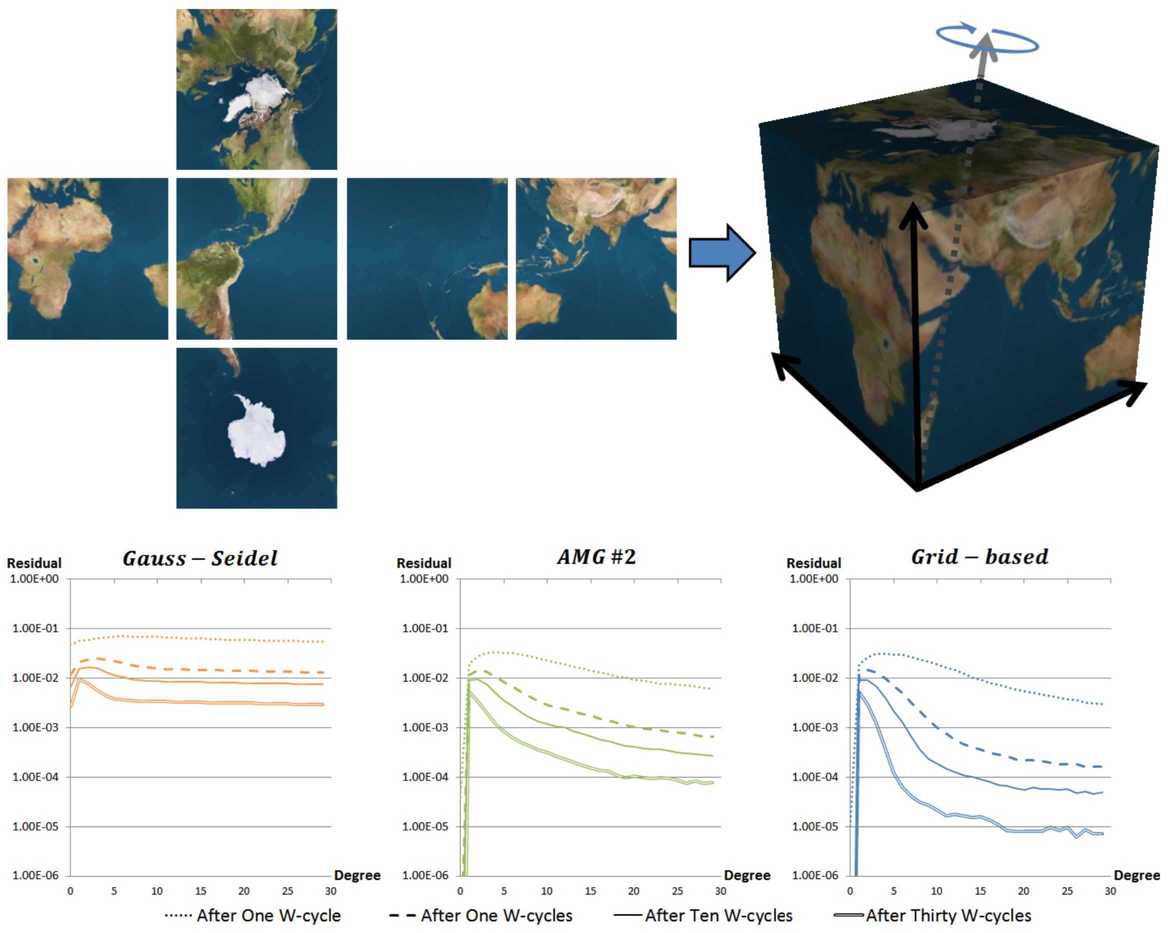


Figure 4.11: Convergence of ill-conditioned systems. The color fitting problem is repeatedly solved over an axis-aligned cube rotated around the $(1, 1, 1)$ vector. The finest-level linear system is defined using the grid-based basis. The residuals are plotted as a function of degrees by which the cube is rotated.

on the YZ plane, $x = x_0$, will only differ by the constant $\frac{B_{x_1}(x_0)}{B_{x_2}(x_0)}$, where B_x is the 1-D B-spline centered at x.

Fortunately, while this is a serious problem for direct solvers like CHOLMOD, it affects us less significantly because our multigrid solver uses a Gauss-Seidel smoother, which has been shown to be well-behaved even when the linear system is only semi-definite [75, 76]. In this case, the error vectors $\{e^i\}$ (here $e^i := x - x^i$ with x^i the

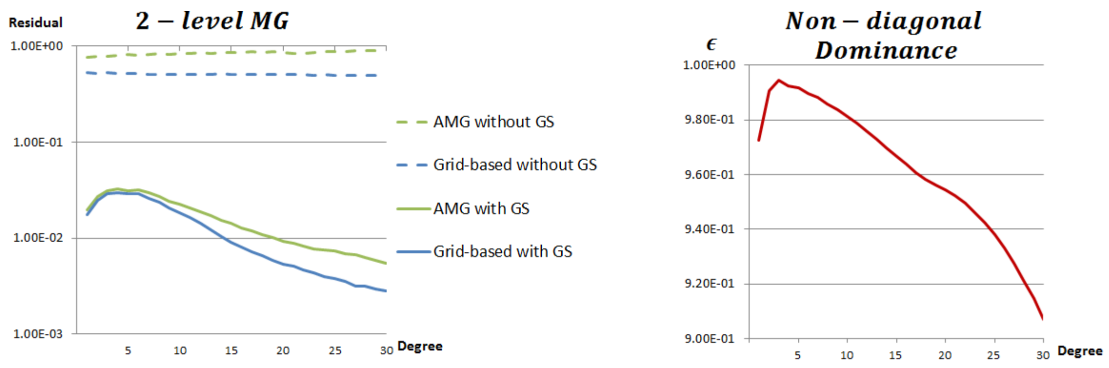


Figure 4.12: Correctness of the correction term from the lower resolution system, solved accurately using a direct solver (*left*). Note that the system is corrected by the same amount regardless the degrees of rotation, but the smoother works more effectively when the system is better conditioned. We have also found that the performance of the smoother correlates with the degree of non-diagonal dominance(*right*).

solution at iteration i and x a fixed solution) converge to a vector living in the null space of the linear system.

In practice, we have found that our solver has more trouble when the surface is *barely* in general position. We evaluate this by designing a simple experiment: We solve the same color-fitting problem as before (here we set $\lambda = 10^{-2}$) on a cube. The cube, initially axis-aligned, is rotated one degree at a time around the vector $(1, 1, 1)$. Hence, the system starts off at special position, and gets pulled away gradually to general position.

The results of this experiment are shown in Figure 4.11, where we plot the residuals after one, five, ten, and thirty W-cycles. As indicated by the plots, our solver behaves surprisingly well when the system is singular (i.e., when there is no rotation applied). Unfortunately, when a slight rotation is applied, making the system near-singular instead, the convergence rate deteriorates noticeably.

To better pinpoint the cause of this behavior, in Figure 4.12 (left), we consider only the first two levels of the multigrid hierarchy, where the coarser level is solved exactly using a direct solver. We notice that, without using any smoother at the finer level, the residuals are corrected by about the same amount (0.5 for our multigrid and > 0.75 for the classical AMG) no matter how the cube is rotated. On the other hand, running the smoother on the systems in general position yields better results. This implies that the problem is *not* because multigrid fails to provide good correction for the finer systems; rather, it is because the smoother performs less effectively when the mesh is close to special position.

We believe that this inadequate convergence relates to the diagonal dominance of the linear system. To see this, we first measure the non-diagonal dominance by setting

$$\epsilon = \frac{\sum_i \varepsilon_i}{\sum_i |M_{ii}|} \tag{4.7}$$

where

$$\varepsilon_i = \begin{cases} -|M_{ii}| + \sum_{i \neq j} |M_{ij}| & \text{if } \sum_{i \neq j} |M_{ij}| > |M_{ii}|, \\ 0 & \text{otherwise.} \end{cases} \tag{4.8}$$

In Figure 4.12 (right), we plot ϵ as a function of rotation degrees. The results closely correlate with the observed convergence, suggesting that the Gauss-Seidel smoother, though proven to converge on symmetric positive definite systems, has trouble converging quickly when the diagonal dominance is weak.

We also evaluate the qualitative implications of convergence by visually inspecting the results. Looking at Figure 4.13 (top), we see that our solution (after one W-cycle) closely approximates the low frequency content when compared to the ground truth solution. Unfortunately, high frequency errors does appear to arise (Figure 4.13, bottom), indicating that this is not a negligible issue.

In future work, we would like to explore this problem more carefully, by either developing effective smoothers tailored to our system or adapting our function space to make the resulting system more diagonally dominant.

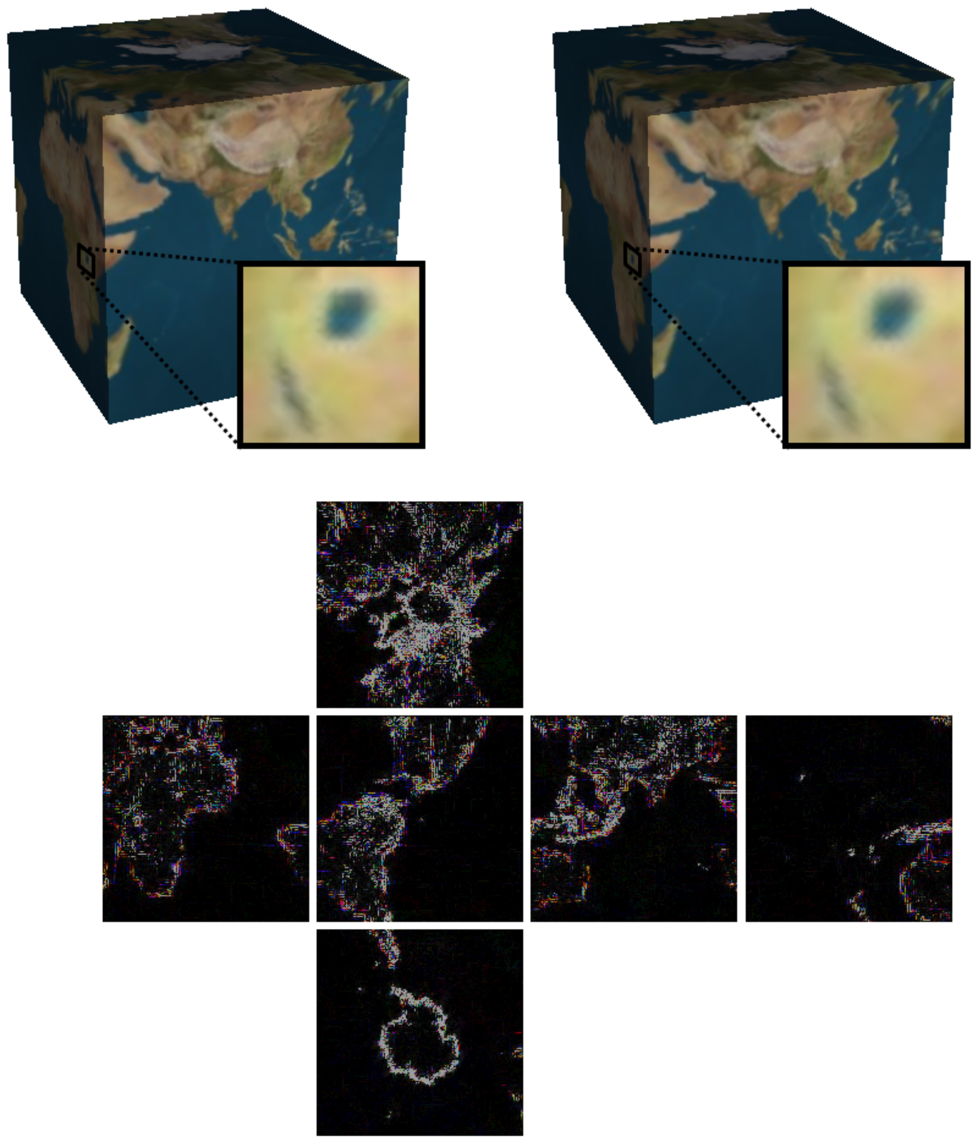


Figure 4.13: Visual comparison of the ground truth solution (*left*) and our solution obtained after one W-cycle (*right*). The cube is rotated by one degree around the axis $(1, 1, 1)$. The L^1 norms of the errors, scaled up by 100, reveal that high frequency errors persist in our solution (*bottom*).

Chapter 5

Changing Metric

In the previous chapter, we introduced our grid-based finite elements system for solving the Poisson equation on meshes. In order to compute the mass and stiffness matrices (by integrating Equation 3.5 and 3.6), we use the metric induced by the embedding of the mesh.

In this chapter, we will show that this is not the only choice and that decoupling the metric from the embedding can expand the applicability of our system. As examples, we will show that this decoupling supports efficient surface evolution (Section 5.1), and enables anisotropic geometry processing (Section 5.2).

5.1 Metric Tracking for Surface Evolution

Our grid-based finite elements system defines a nested hierarchy of functions, supporting an efficient multigrid framework. However, due to its dependence on the embedding of the mesh, the overhead of setting up the framework is not negligible, as it requires constructing an adaptive octree for tracking basis functions, clipping the mesh by faces of octree cells, populating quadrature points, and then integrating.

As a result, for dynamic applications such as surface evolution where the embedding changes with time, the repeated initialization of the multigrid framework becomes a significant cost.

To address this concern, we propose an alternate approach that deforms the test functions *with* the evolving embedding, thus allowing us to re-use the multigrid framework throughout the application. Figure 5.1 gives a visualization of the approach using a 2D model undergoing a deformation (top row). Instead of defining a new hierarchy of test functions at the beginning of each timestep (middle row), we adapt the hierarchy defined using the initial embedding by evolving the test functions with the deforming surface (bottom row).

At first glance, the approach may appear impractical due to the complexity of the deformed test functions. This turns out not to be the case and the approach can be implemented by slightly adapting the original system. The key idea is to consistently formulate the integral on the initial manifold, by pulling back the inner product from the new embedding, allowing much of the computation to be performed

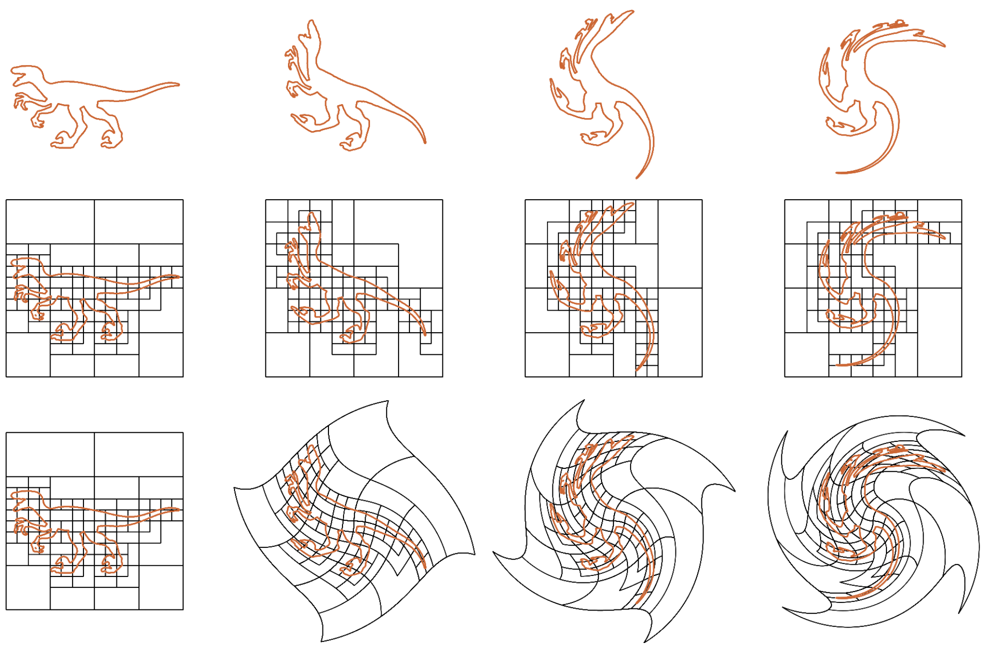


Figure 5.1: A 2D raptor model undergoing a “swirl” deformation (top). Computing a quadtree independently for each deformation, we obtain a temporally-varying spatial indexing structure (middle). Tracking the quadtree with the deformed surface, the indexing structure remains constant, allowing us to reuse information from frame to frame (bottom).

in a pre-processing stage.

Here, we denote by ∇ the gradient operator, independent of metric, taking functions on $\mathcal{M}$ to a section of the cotangent bundle of $\mathcal{M}$. Note that, for a given metric g, one can uniquely relate a tangent vector v to a cotangent vector α_v by setting $\alpha_v(\cdot) = \mathsf{g}(v, \cdot)$. That is, g can be thought of as a map from the tangent space to its dual. This allows us to relate a gradient ∇b, as an element of the cotangent space, to a gradient $\nabla_{\mathsf{g}} b$, as an element of the tangent space, by setting $\nabla_{\mathsf{g}} b = \mathsf{g}^{-1}\nabla b$. As a result, given $\nabla_{\mathsf{g}_1} b$ and $\nabla_{\mathsf{g}_2} b$ defined with respect to two different metrics, g_1 and g_2, one can relate them by $\nabla_{\mathsf{g}_1} b = \mathsf{g}_1^{-1}\mathsf{g}_2\nabla_{\mathsf{g}_2} b$.

Now, with a little abuse of notation, we denote by ∇_t the gradient operator defined with respect to metric g_t (that is, ∇_{g_t}), and denote by $d\mu_t$ the area measure defined with respect to g_t. Using this notation, we formulate the mass and stiffness matrices at an arbitrary time t with respect to the initial Riemannian metric g_0 as

$$\begin{aligned} M_{ij}^t &= \int_M b_i \cdot b_j \; d\mu_t \\ &= \int_M b_i \cdot b_j \cdot \sqrt{|\mathsf{g}_t^{-1}\mathsf{g}_0|} \; d\mu_0 \end{aligned} \tag{5.1}$$

$$\begin{aligned} L_{ij}^t &= \int_{\mathcal{M}} \mathsf{g}_t(\nabla_t b_i, \nabla_t b_j) \; d\mu_t \\ &= \int_{\mathcal{M}} \nabla b_i(\nabla_t b_j) \; d\mu_t \\ &= \int_{\mathcal{M}} \nabla b_i(\mathsf{g}_t^{-1}\mathsf{g}_0\nabla_0 b_j) \cdot \sqrt{|\mathsf{g}_t^{-1}\mathsf{g}_0|} \; d\mu_0 \\ &= \int_{\mathcal{M}} \mathsf{g}_0(\nabla_0 b_i, \mathsf{g}_t^{-1}\mathsf{g}_0\nabla_0 b_j) \cdot \sqrt{|\mathsf{g}_t^{-1}\mathsf{g}_0|} \; d\mu_0 \end{aligned} \tag{5.2}$$

Similarly, the constraint vectors at time t can be formulated as

$$\mathbf{f}_i^t = \int_{\mathcal{M}} f \cdot b_i \cdot \sqrt{|\mathsf{g}_t^{-1}\mathsf{g}_0|} \; d\mu_0 \tag{5.3}$$

$$\mathbf{s}_i^t = \int_{\mathcal{M}} \mathsf{g}_0(\vec{s}, \mathsf{g}_t^{-1}\mathsf{g}_0\nabla_0 b_i) \cdot \sqrt{|\mathsf{g}_t^{-1}\mathsf{g}_0|} \; d\mu_0 \tag{5.4}$$

Note that our formulation so far has not involved any coordinate system. If we choose a frame $\{v_1, v_2\}$ on the tangent space that is *orthonormal with respect to* g_0, the matrix representation of g_0 (with respect to this frame) is simply the identity. Also, given the coefficients $\vec{x}^t = [x_1^t, \ldots, x_n^t]^\dagger$ defining the embedding $X^t(p) = \sum_i x_i^t \cdot b_i(p)$ at time t, we can obtain the differential of the embedding with respect to $\{v_1, v_2\}$ as

$$dX^t|_p = \sum_i x_i^t \Big(\; \frac{\partial(b_i(p))}{\partial v_1} \quad \frac{\partial(b_i(p))}{\partial v_2} \; \Big),$$

which defines the matrix representation of g_t (that is, $\mathrm{g}_t = (dX^t)^\dagger \cdot dX^t$).

Implementation

We implement the approach by slightly adapting the algorithms from the previous chapter (Algorithm 4.1 and 4.2). In particular, we still set up the quadrature points on the (undeformed) mesh as described in Section 4.2. We then choose an orthonormal tangent frame $\{v_1(p), v_2(p)\}$ for a quadrature point. Using Equations 5.1 through 5.4, we modify the formulae for computing the contribution from the quadrature points (with tangent vectors represented with respect to the basis $\{v_1(p), v_2(p)\}$):

$$M^t_{ij} = \sum_{c\in C}\sum_{t\in T(c)}\sum_{p\in P(t)} b_i(p)\cdot b_j(p)\cdot \sqrt{|\mathrm{g}_t(p)|}\cdot\omega(p) \tag{5.5}$$

$$L^t_{ij} = \sum_{c\in C}\sum_{t\in T(c)}\sum_{p\in P(t)} (\nabla_0 b_i(p))^\dagger\cdot \mathrm{g}_t^{-1}(p)\cdot\nabla_0 b_j(p)\cdot \sqrt{|\mathrm{g}_t(p)|}\cdot\omega(p) \tag{5.6}$$

$$\mathbf{f}^t_i = \sum_{c\in C}\sum_{t\in T(c)}\sum_{p\in P(t)} b_i(p)\cdot f(p)\cdot \sqrt{|\mathrm{g}_t(p)|}\cdot\omega(p) \tag{5.7}$$

$$\mathbf{s}^t_i = \sum_{c\in C}\sum_{t\in T(c)}\sum_{p\in P(t)} (\vec{s}(p))^\dagger\cdot \mathrm{g}_t(p)\cdot\nabla_0 b_i(p)\cdot \sqrt{|\mathrm{g}_t(p)|}\cdot\omega(p) \tag{5.8}$$

Note that we as deform the test functions, theoretically the degree of polynomials increases, and we would need to use high degree integrators for stable integration. However, in practice, we have found that the 3-point formula remains sufficiently accurate even for long-term evolution (see the next subsection for empirical validation.)

As suggested previously, one major advantage of formulating the integral with respect to the initial Riemannian metric is that much of the information can be re-used. This allows us to speed up the computation by computing (and storing) per-point information in a preprocessing step. The data associated with each quadrature sample are kept throughout evolution and are used for assembling system coefficients at run time.

Note that since we use first-order 3D B-splines centered at grid corners as test functions, each quadrature sample is in the support of exactly eight test functions. We store the following values for each sample p (the numbers in the parentheses indicate the numbers of the required floating point values):

- (1) The weight of the sample, with respect to the initial metric: $\omega(p)$.
- (8) The values of the eight supported B-splines: $b_i(p)$.
- (16) The gradients of the eight supported B-splines expressed as coefficients with respect to the chosen frame: $[\nabla_0 b_i(p)]_{v_1}$ and $[\nabla_0 b_i(p)]_{v_2}$ (i.e., $[u]_{v_i} = \mathsf{g}_0(u, v_i)$).

This sums to a total of $1 + 8 + 16 = 25$ floating point values per quadrature sample[1]. Algorithm 5.1 through 5.3 summarize the pseudocode for constructing the system for an embedding that evolves with time.

5.1.1 Results

To evaluate the approach, we use *mean-curvature flow* (MCF) to evolve the embedding of the input mesh. In particular, we investigate the stability, accuracy, and improved efficiency of the proposed tracking approach.

[1]Depending on applications and forms of constrains, one may also want to store values for f and $\vec{s}$. This will result in three additional floating point values (i.e., $f(p)$, $[\vec{s}(p)]_{v_1}$, and $[\vec{s}(p)]_{v_2}$).

```
**Algorithm III**: SetTimeVaryingSystemMatrices( $\vec{x}$ )
$M$ , $L \leftarrow 0$
for $c \in C$ , $t \in T(c)$ , $p \in P(t)$
  $dX \leftarrow$ ComputeDifferential( $p$ , $\vec{x}$ )
  $\mathsf{g} \leftarrow dX^\dagger \cdot dX$
  for $i \in$ NeighborCorner($p$)
    for $j \in$ NeighborCorner($p$)
      $M_{ij} \leftarrow M_{ij} + b_i(p) \cdot b_j(p) \cdot \sqrt{|\mathsf{g}|} \cdot \omega(p)$
      $L_{ij} \leftarrow L_{ij} + \Big([\nabla_0 b_i(p)]_{v_1}, [\nabla_0 b_i(p)]_{v_2}\Big)\mathsf{g}^{-1}\Big([\nabla_0 b_j(p)]_{v_1}, [\nabla_0 b_j(p)]_{v_2}\Big)^\dagger \cdot \sqrt{|\mathsf{g}|} \cdot \omega(p)$
return ( $M$ , $L$ )
```

Algorithm 5.1: Algorithm for computing the system matrices with a time-varying embedding. The embedding coefficients $\vec{x}$ are given at each timestep t.

```
**Algorithm IV**: SetTimeVaryingConstraintVectors( $\vec{x}$ )
$\mathbf{f}$ , $\mathbf{s} \leftarrow 0$
for $c \in C$ , $t \in T(c)$ , $p \in P(t)$
  $dX \leftarrow$ ComputeDifferential( $p$ , $\vec{x}$ )
  $\mathsf{g} \leftarrow dX^\dagger \cdot dX$
  for $i \in$ NeighborCorner($p$)
    $\mathbf{f}_i \leftarrow \mathbf{f}_i + f(p) \cdot b_i(p) \cdot \sqrt{|\mathsf{g}|} \cdot \omega(p)$
    $\mathbf{s}_i \leftarrow \mathbf{s}_i + \Big([\vec{s}(p)]_{v_1}, [\vec{s}(p)]_{v_2}\Big)^\dagger \mathsf{g}^{-1}\Big([\nabla_0 b_i(p)]_{v_1}, [\nabla_0 b_i(p)]_{v_2}\Big) \cdot \sqrt{|\mathsf{g}|} \cdot \omega(p)$
return ( $\mathbf{f}$ , $\mathbf{s}$ )
```

Algorithm 5.2: Algorithm for computing the constraint vectors with a time-varying embedding. The embedding coefficients $\vec{x}$ are given at each timestep t.

```
**Algorithm V**: ComputeDifferential( $p$ , $\vec{x}$ )
$dX \leftarrow 0$
for $i \in$ NeighborCorner($p$)
  $dX \leftarrow dX + x_i \cdot \Big([\nabla_0 b_i(p)]_{v_1}, [\nabla_0 b_i(p)]_{v_2}\Big)$
return $dX$
```

Algorithm 5.3: Algorithm for computing the differential of the embedding.

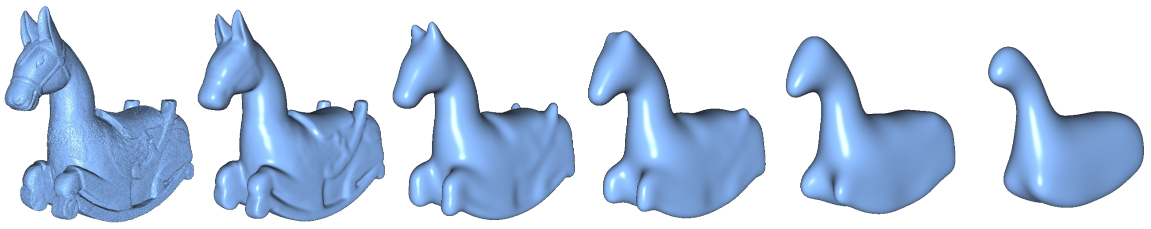


Figure 5.2: Mean-curvature flow of the Isidore Horse after 0, 5, 25, 50, 100, and 200 iterations with step-size $\delta = 1 \times 10^{-5}$.

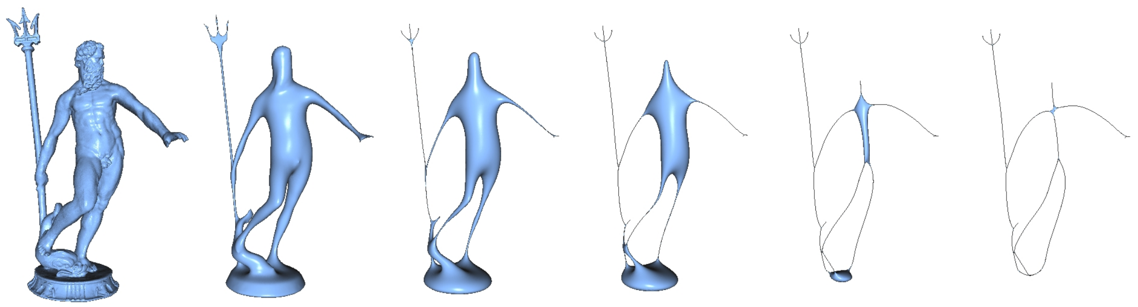


Figure 5.3: Mean-curvature flow of Neptune after 0, 5, 10, 20, 40, and 80 iterations with step-size $\delta = 1 \times 5 \times 10^{-4}$.

Mean-Curvature Flow

MCF is a classical flow that has been widely used to evolve surface geometry, supporting applications such mesh fairing and editing [24, 26]. Formally, MCF is a smoothing process that evolves the surface embedding X subject to the differential equation:

$$\frac{\partial}{\partial t} X = \vec{H}_t = \Delta_t X \tag{5.9}$$

where $\vec{H}_t$ is the mean-curvature vector of the embedded surface at time t, which is also the Laplacian of the embedding.

It is well-known that, unless very small time-steps are taken, explicit integration of Equation 5.9 results in unwanted negation/amplification of high-frequency content, making the integration process unstable. Instead, methods such as Desbrun*et al.*'s use a semi-implicit integration [26]:

$$\frac{X^{t+\delta} - X^t}{\delta} \approx \Delta_t X^{t+\delta} \Longrightarrow (Id - \delta\Delta_t)X^{t+\delta} \approx X^t \tag{5.10}$$

Projecting Equation 5.10 onto $\{b_1, \ldots, b_n\}$, one obtains a screened-Poisson equation with zero-valued gradient constraint:

$$\left(D^t - \frac{\delta}{2}L^t\right)\vec{x}^{t+\delta} = D^t\vec{x}^t \tag{5.11}$$

This equation is stable even in the presence of large time-steps, making it more convenient for simulating large-scale flows in practice.

Tracking Stability

To evaluate the stability of the proposed tracking approach, we repeatedly solve Equation 5.11 to evolve the surface embedding. The results are shown in Figures 5.2 and 5.3, where we evolve the Isidore Horse and the Neptune models using different step-sizes. With smaller time-steps, our approach can stably evolve the mesh over hundreds of iterations and fair out most surface details (i.e., Isidore Horse). On the other hand, with larger time-steps (i.e., Neptune), the approach can obtain the "skeleton" of the mesh, as described in [77]. (In pinching regions, where the metric becomes singular, we keep the quadrature points from contributing to the integrals, thus avoiding the problem of having differentials go to infinity. As a result, the associated coefficients are not changed by the iterative solver and the surface "locks" as it approaches the skeleton.)

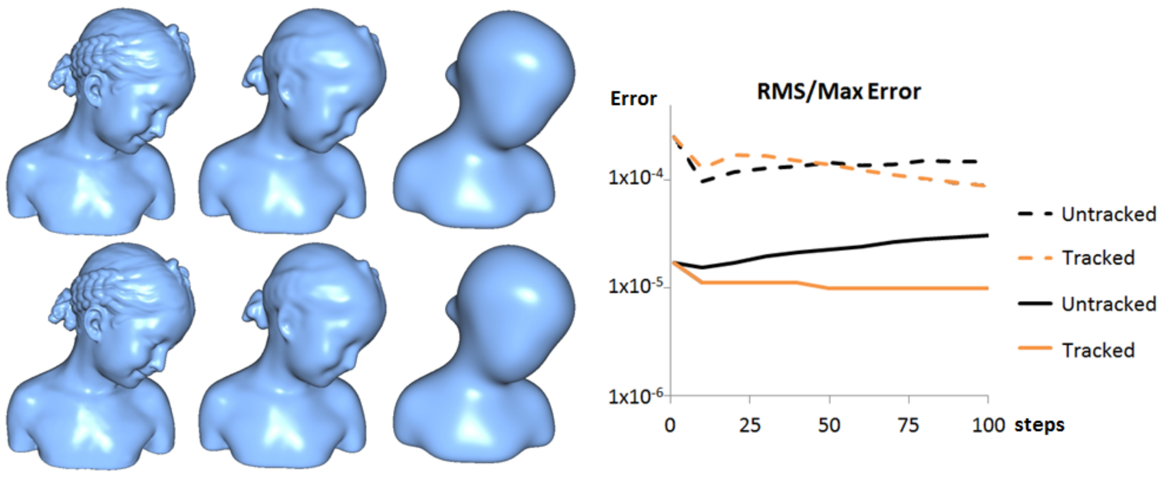


Figure 5.4: Accuracy of the metric tracking method performing MCF. We evolve the Bimba model using both the tracking (top) and non-tracking (bottom) approaches. Here we show the results after one, ten, and one hundreds time-steps of MCF with step-size $\delta = 1 \times 10^{-5}$. The maximum (dashed line) and RMS (sold lines) errors are plotted as a function of time-steps (right).

Accuracy

We evaluate the correctness of our formulation by evolving the model using both the tracking and non-tracking approaches. Figure 5.4 shows the results for the Bimba model after one, ten, and one hundred steps of mean-curvature flow with step-size $\delta = 1 \times 10^{-5}$. Visually, the results of the tracking approach (top row) are indistinguishable from the results of the non-tracking approach (bottom row).

We quantify the errors using the Metro [78] tool, computing the distances from the results obtained using the grid-based systems, to the results obtained from the hat-basis system solved precisely by CHOLMOD. In the right of Figure 5.4, we plot both the maximum (dashed lines) and RMS (solid lines) errors. Note that the errors are small for both methods. Perhaps a little surprisingly, the tracked system performs

Model	Isidore Horse	Neptune	Bimba
Input Vertices	1.1×10^6	5.0×10^5	3.0×10^5
Clipped Triangles	1.0×10^7	4.3×10^6	2.7×10^6
System Dimensions	1.2×10^6	4.7×10^5	3.4×10^5
Setup Time	26 sec.	11 sec.	7 sec.
Update Time	3.7 sec.	1.3 sec.	0.9 sec.
Solve Time	0.47 sec.	0.20 sec.	0.14 sec.

Table 5.4: Running time of our system performing MCF. Note that Setup Time indicates what would have taken to rebuild the system at each time-step.

slightly better than the non-tracked one. We believe this is because the tracked system effectively evolves with the finer, clipped, triangulation, allowing it to capture more fine-grained properties of the flow.

Improved Efficiency

The performance of our system is summarized in Table 5.4, showing the size of the input mesh, the size of the clipped mesh, the dimension of the linear system, the time required for the non-tracking method to rebuild the system at each time-step, the time required for the tracking method to update the system at each time-step, and the time required to solve the linear system.

As the table indicates, without having to adapt an octree and re-clipping the input mesh at the beginning of each time-step, the tracked system obtains a nearly 7-fold speedup in the processing time, confirming the benefit of using the metric tracking approach.

5.2 Anisotropic Geometry Processing

In the previous section, we had shown that the metric used for defining the system does not have to be the metric induced by the embedding. Using the metric pulled back from an evolving surface, one can adapt the system to support surface evolution more efficiently. The approach is easy to implement and requires only minimal changes to the original algorithms.

Indeed, as the system coefficients are obtained by summing up contributions from quadrature samples, modifying the metric only requires changing the formulae for computing the per-point contribution. In theory, the framework is general enough to support arbitrary metrics. Unfortunately, it is difficult to perform such computation in real-time: When the metric changes, the point-wise integration of the system coefficients must be computed anew.

Nevertheless, we observe that one can trade a certain degree of flexibility designing the metric for efficiency. In particular, we restrict ourselves to the subset of metric tensors that are:

- Diagonalizable with respect to the principal curvature directions (with the diagonal entries depending on the curvature value and position), and
- Constant within each grid cell.

We now explain the implications of these restrictions.

Diagonalizability

Our first assumption about the prescribed metric tensor, G, is that if $\{(\kappa_1, v_1), (\kappa_2, v_2)\}$ are the principal curvatures, then G can be diagonalized with respect to the basis $\{v_1, v_2\}$ as

$$G \equiv \begin{pmatrix} \alpha_1^2 & 0 \\ 0 & \alpha_2^2 \end{pmatrix}$$

with α_i defined by $\alpha_i(p) = \alpha(p, \kappa_i(p)) \neq 0$. Since α only depends on the position and the principal curvature, even at umbilic points the metric tensor is well-defined.

The advantage of this assumption is that we can re-use the computation of the system defined by the initial metric to compute the system defined by the prescribed metric (much like what we did previously to accelerate surface evolution). Note that, though the assumption might seem restrictive at first glance, it covers a rich family of metric tensors and has been successfully applied to design anisotropic geometry processing systems [27].

Under this assumption, the system matrices can be expressed as

$$\begin{aligned} M_{ij}^G &= \int_{\mathcal{M}} b_i \cdot b_j \; d\mu_G \\ &= \int_{\mathcal{M}} \alpha_1 \cdot \alpha_2 \cdot b_i \cdot b_j \; d\mu_{\mathrm{g}} \end{aligned} \tag{5.12}$$

$$\begin{aligned} L_{ij}^G &= \int_M G(\nabla_G b_i, \nabla_G b_j) \; d\mu_G \\ &= \int_M \Big([\nabla_{\mathrm{g}} b_i]_{v_1}, [\nabla_{\mathrm{g}} b_i]_{v_2} \Big) \; G^{-1} \; \Big([\nabla_{\mathrm{g}} b_j]_{v_1}, [\nabla_{\mathrm{g}} b_j]_{v_2} \Big)^{\dagger} \cdot \sqrt{|G|} \; d\mu_{\mathrm{g}} \\ &= \int_{\mathcal{M}} \Big(\frac{\alpha_1 \cdot \alpha_2}{\alpha_1^2} \cdot [\nabla_{\mathrm{g}} b_i]_{v_1} \cdot [\nabla_{\mathrm{g}} b_j]_{v_1} + \frac{\alpha_1 \cdot \alpha_2}{\alpha_2^2} \cdot [\nabla_{\mathrm{g}} b_i]_{v_2} \cdot [\nabla_{\mathrm{g}} b_j]_{v_2} \Big) \; d\mu_{\mathrm{g}} \end{aligned} \tag{5.13}$$

Intuitively, $(\alpha_1 \cdot \alpha_2)$ accounts for the area scaling, while $1/\alpha_1^2$ and $1/\alpha_2^2$ account for the fact that scaling a function by α scales its second derivative by $1/\alpha^2$.

Similarly, the constraint vectors become

$$\mathbf{f}_i^G = \int_{\mathcal{M}} \alpha_1 \cdot \alpha_2 \cdot f \cdot b_i \; d\mu_{\mathrm{g}} \tag{5.14}$$

$$\mathbf{s}_i^G = \int_{\mathcal{M}} \Big(\frac{\alpha_1 \cdot \alpha_2}{\alpha_1^2} \cdot [\vec{s}]_{v_1} \cdot [\nabla_{\mathrm{g}} b_i]_{v_1} + \frac{\alpha_1 \cdot \alpha_2}{\alpha_2^2} \cdot [\vec{s}]_{v_2} \cdot [\nabla_{\mathrm{g}} b_i]_{v_2} \Big) \; d\mu_{\mathrm{g}} \tag{5.15}$$

Constant metric structure

Our second assumption is that α_1 and α_2 are piecewise constant. This is motivated by the observation that, in the extreme case, if α_1 and α_2 are constant everywhere, one can move their contribution outside the integrals. As a result, given prescribed metrics, we can efficiently construct new linear systems by rescaling the components in the original linear system. For example, under this assumption the new stiffness matrix can be assembled as

$$L^{\mathrm{g}} = \frac{\alpha_1 \cdot \alpha_2}{\alpha_1^2} L^{(v_1)} + \frac{\alpha_1 \cdot \alpha_2}{\alpha_2^2} L^{(v_2)}$$

where

$$L_{ij}^{(v)} = \int_{\mathcal{M}} [\nabla_{\mathrm{g}} b_i]_v \cdot [\nabla_{\mathrm{g}} b_j]_v \; d\mu_{\mathrm{g}}$$

This global assumption, however, is too restrictive as it does not allow for a spatially-varying metric: We would like to speed up the computation without sacrificing the flexibility in designing the metric. In practice, we have found that restricting prescribed metric tensors to be *piecewise constant on grid cells* provides a good trade-off between flexibility and speed. The design decision is well aligned with the implementation described in Chapter 4, where we compute each system coefficient by summing integral contributions from individual grid cells.

Implementation

Leveraging the two assumptions regarding the prescribed metric tensor, our implementation of the approach computes the linear system as

$$M_{ij}^G = \sum_{c\in C} \alpha_1(c)\cdot\alpha_2(c)\cdot\Big(\sum_{t\in T(c)}\sum_{p\in P(t)} b_i(p)\cdot b_j(p)\cdot\omega(p)\Big) \tag{5.16}$$

$$\begin{aligned} L_{ij}^G = \sum_{c\in C}\Bigg(& \frac{\alpha_1(c)\cdot\alpha_2(c)}{\alpha_1^2(c)}\cdot\Big(\sum_{t\in T(c)}\sum_{p\in P(t)} [\nabla_{\mathbf{g}} b_i(p)]_{v_1}\cdot[\nabla_{\mathbf{g}} b_j(p)]_{v_1}\cdot\omega(p)\Big) + \\ & \frac{\alpha_1(c)\cdot\alpha_2(c)}{\alpha_2^2(c)}\cdot\Big(\sum_{t\in T(c)}\sum_{p\in P(t)} [\nabla_{\mathbf{g}} b_i(p)]_{v_2}\cdot[\nabla_{\mathbf{g}} b_j(p)]_{v_2}\cdot\omega(p)\Big)\Bigg) \end{aligned} \tag{5.17}$$

$$\mathbf{f}_i^G = \sum_{c\in C} \alpha_1(c)\cdot\alpha_2(c)\cdot\Big(\sum_{t\in T(c)}\sum_{p\in P(t)} f(p)\cdot b_i(p)\cdot\omega(p)\Big) \tag{5.18}$$

$$\begin{aligned} \mathbf{s}_i^G = \sum_{c\in C}\Bigg(& \frac{\alpha_1(c)\cdot\alpha_2(c)}{\alpha_1^2(c)}\cdot\Big(\sum_{t\in T(c)}\sum_{p\in P(t)} [\vec{s}(p)]_{v_1}\cdot[\nabla_{\mathbf{g}} b_i(p)]_{v_1}\cdot\omega(p)\Big) + \\ & \frac{\alpha_1(c)\cdot\alpha_2(c)}{\alpha_2^2(c)}\cdot\Big(\sum_{t\in T(c)}\sum_{p\in P(t)} [\vec{s}(p)]_{v_2}\cdot[\nabla_{\mathbf{g}} b_j(p)]_{v_2}\cdot\omega(p)\Big)\Bigg) \end{aligned} \tag{5.19}$$

where $\alpha_i(c)$ is the (constant) value of the function α_i on cell c.

In a preprocessing stage, we compute (and store) the components of the per-cell contribution that are independent of the prescribed metric tensor. Then, at runtime, the system coefficients are assembled efficiently using the precomputed information. For example, by precomputing

$$Int^{L_\ell}(c,i,j) = \sum_{t\in T(c)}\sum_{p\in P(t)} [\nabla_{\mathbf{g}} b_i(p)]_{v_\ell}\cdot[\nabla_{\mathbf{g}} b_j(p))]_{v_\ell}$$

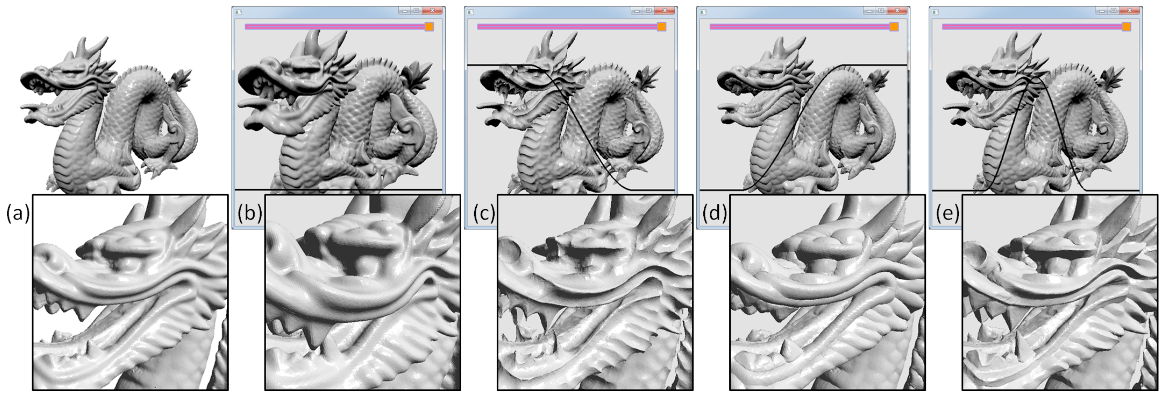


Figure 5.5: Anisotropic detail sharpening: Starting with an initial model (a), global sharpening is applied to the geometry to enhance the detail (b). By adapting the direction of sharpening to the curvature in different ways, a rich space of geometry-aware sharpening filters are realized (c-e). Though the model consists of almost one million vertices and a new system is constructed and solved each time the filter is changed, our method still supports geometry processing at interactive rates.

we can assemble the stiffness matrix when the metric is prescribed by setting

$$L_{ij} = \sum_{c\in C} \Big(\frac{\alpha_1(c)\cdot\alpha_2(c)}{\alpha_1^2(c)} Int^{L_1}(c,i,j) + \frac{\alpha_1(c)\cdot\alpha_2(c)}{\alpha_2^2(c)} Int^{L_2}(c,i,j) \Big)$$

5.2.1 Results

To evaluate the approach, we implement a real-time system that performs anisotropic geometry filtering through the solution of a screened-Poisson equation. The idea is to extend the screened-Poisson formulation of gradient domain image processing [20, 79] to meshes. In particular, we set the value constraint f to be the embedding of the mesh and set the gradient constraint to be $\vec{s} = \sigma\nabla f$. Here, $\sigma \geq 0$ is a user-controlled variable dictating modulation of surface details. (When σ is bigger/less than one, the gradients are amplified/dampened, and we achieve a sharp-

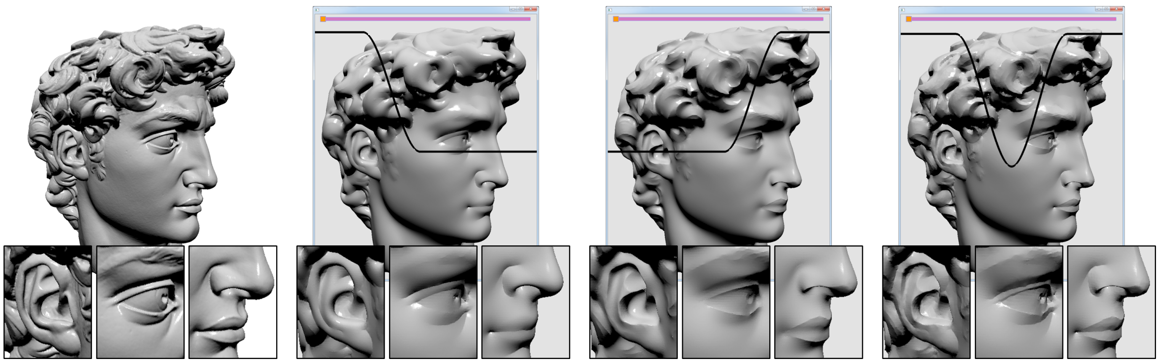


Figure 5.6: Anisotropic detail smoothing: Examples of geometric effects obtained by adapting the Riemannian metric to the curvature. Starting with the original model (left), global smoothing constraints were applied. The surfaces, from left to right, are obtained by amplifying the fidelity term (λ) in directions of: large negative curvature, large positive curvature, large absolute curvature.

ening/smoothing effect.) Figure 1.3 gives a snapshot of our system performing the (isotropic) geometry filtering on the Armadillo Man. We provide a simple slider interface (at the top of the window) for users to adjust σ, and we also support a spraycan interface to allow users to define a spatially variable σ.

We allow users to interactively adjust the metric by specifying the *transfer function* $\alpha(\kappa)$, which determines the scaling of the inner product in principal curvature directions with principal curvature value κ:

$$\alpha_i(p) = \alpha(\kappa_i(p))$$

As described in the previous section, in order to speed up the computation, we need α_1 and α_2 to be piecewise-constant on grid cells. To this end, we associate a minimum and maximum curvature value with each grid cell, obtained by taking the area-weighted average of the minimum and maximum curvatures of the triangles contained within

Model	Vertices	DoFs	Frames/Seconds		
			Solve	**RHS**	**Matrix**
Lucy	2.6×10^5	$3 \times 1.5 \times 10^5$	40	20	8
Buddha	5.4×10^5	$3 \times 2.1 \times 10^5$	31	15	6
Armadillo	1.7×10^5	$3 \times 2.6 \times 10^5$	30	13	5
Dragon	4.4×10^5	$3 \times 2.7 \times 10^5$	27	12	5
Isidore	1.1×10^6	$3 \times 2.8 \times 10^5$	27	12	4
Formaa	1.0×10^6	$3 \times 3.2 \times 10^5$	24	11	4
David	2.0×10^6	$3 \times 4.2 \times 10^5$	20	9	3

Table 5.5: Performance Summary: Statistics of the geometric complexity, numbers of degrees of freedom, and frame-rate.

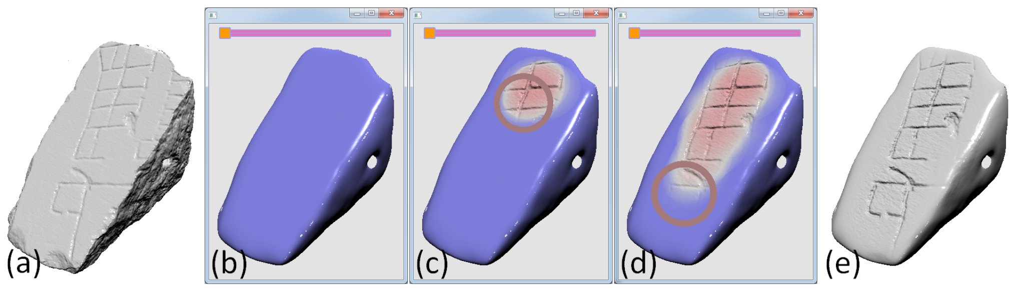


Figure 5.7: Selective detail enhancement: Starting with the original model (a), a user applies global smoothing by specifying that all gradients should be dampened (b). The user then specifies that the top face of the tablet should be sharpened by selectively amplifying gradients in that area (c-d). The final results accentuates the floor plan in the Forma Urbis fragment and hides detail in the fracture region (e).

the cell (curvatures are computed using Trimesh2 [80]). Figures 5.5 and 5.6 give snapshots of our system performing anisotropic sharpening/smoothing on meshes. Here we provide a simple profile-curve interface for users to specify the transfer function.

We test our anisotropic geometry processing system on several input meshes. A summary of the performance can be seen in Table 5.5. The three running times correspond to the three states of the system:

1. **Solve** When the user has not specified any edits, the system performs a multi-grid solve at each frame and updates the vertex positions.

2. **RHS** When the user modifies the gradient scales (σ), the right-hand-side of the system is computed before solving and updating the vertex positions.

3. **Matrix** When the user modifies the filter by adjusting the metric, both the system matrices and right-hand-side are computed before solving and updating the vertex positions.

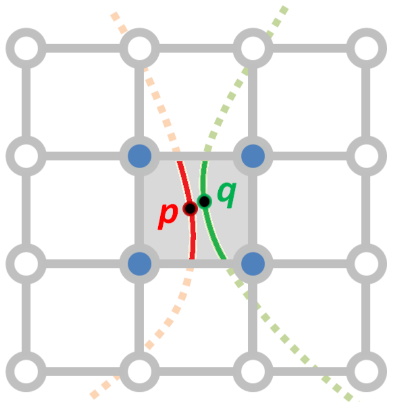
As the table indicates, even for high resolution meshes, our multigrid solver supports interactive modification of both the gradient scale and the underlying Riemannian metric.

Chapter 6

Connectivity Awareness

So far we have described our grid-based finite elements system for solving Poisson-like problems on meshes (Chapter 4). We have also described the extension of the system that modifies metrics for different applications (Chapter 5). In this chapter, we describe another important extension of the system that addresses a fundamental issue with our approach.

Perhaps the most distinct characteristic of our approach is its use of an *extrinsic* function space (i.e., the construction of our function space depends on the embedding of the input mesh). While the use of 3D test functions imbues our system with the desired regularity, we effectively supplant geodesic distances with the Euclidean ones. Specifically, function values on locally disconnected components tend to be *coupled* when they are close in 3D, with points adjacent in 3D having similar function values *even if they are geodesically distant.*



The inset shows a 2D example, where each of the four B-splines supported on the cell takes similar values on p and q. Note that this behavior not only diminishes the richness of our function space, but also affects the performance of our multigrid solver. This is because disconnected regions are more likely to support the same basis function at coarser resolutions. As pointed out in previous work [70], improperly linking basis functions at coarse resolutions can deteriorate convergence of geometric multigrid solvers.

6.1 Enriched Function Space

Our goal is to address the value coupling issue without sacrificing the regularity and nesting structure of the original function space. The key idea is to make the construction of the function space *connectivity-aware*. In particular, we propose to enrich the function space by *splitting* existing functions such that the support of each new function is connected.

Figure 6.1 demonstrates the idea using a 2D example. In the original framework, we would have defined a single function supported on both disjoint components. In the new framework, we use two separate functions instead, which are both derived from the same 3D function but supported on different components.

Formally, letting $\mathcal{M} \bigcap \text{supp}(b_i) = \bigcup_m R_{i,m}$ be a decomposition of the intersection

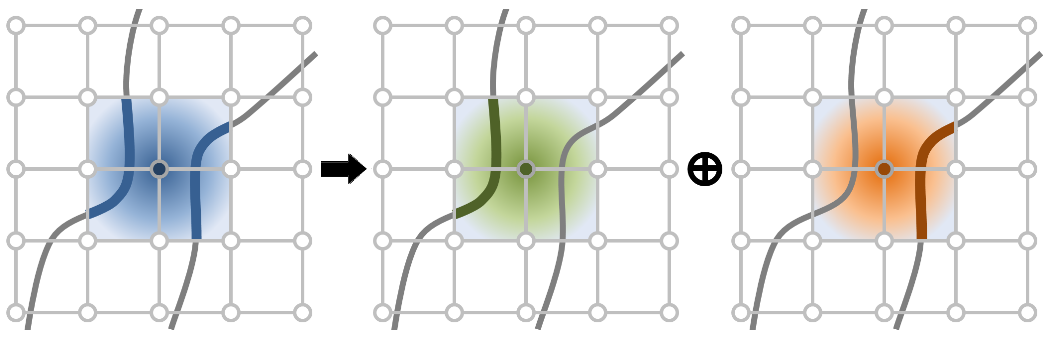


Figure 6.1: Adaptive splitting of test functions. In the original approach, the test functions are chosen independent of the mesh, possibly resulting in disconnected components in their supports (left). In contrast, the alternate approach refers to mesh connectivity and assigns a separate test function to each component (middle and right).

of b_i's support with the mesh into connected components, we define our new test functions by

$$b_{(i,m)}(p) = b_i(p) \cdot \chi_{i,m}(p)$$

with $\chi_{i,m}$ the indicator function on $R_{i,m}$. Thus, $b_{(i,m)}$ is a B-spline centered at corner i and supported on the m-th component of the support of b_i.

Here we make two important observations about the formulation. First, the sum of the new functions centered at corner i is equivalent to the original function centered at corner i (that is, $b_i(p) = \sum_m b_{(i,m)}(p)$). Second, if the value of the original prolongation stencil $P_i(i')$ (where i' is a corner index on the finer grid) is not equal to zero, then one of the following statements is true:

- $\mathrm{supp}(b_{(i',m')}) \bigcap \mathrm{supp}(b_{(i,m)}) = \mathrm{supp}(b_{(i',m')})$

- $\mathrm{supp}(b_{(i',m')}) \bigcap \mathrm{supp}(b_{(i,m)}) = \emptyset$

Using these observations, we formulate the nesting of the new function space as

$$\begin{aligned} b_{(i,m)}(p) &= b_i(p) \cdot \chi_{i,m}(p) \\ &= \left(\sum_{i'} P_i(i') \cdot b_{i'}(p) \right) \cdot \chi_{i,m}(p) \\ &= \sum_{i'} P_i(i') \cdot \sum_{m'} b_{(i',m')}(p) \cdot \chi_{i,m}(p) \\ &= \sum_{i',m'} P_i(i') \cdot \mathcal{I}(i,j,i',m') \cdot b_{(i',m')}(p) \qquad (6.1) \end{aligned}$$

where $\mathcal{I}(i,m,i',m')$ is the function that is one only when the support of $b_{(i,m)}$ contains the support of $b_{(i',m')}$ and is zero otherwise. Equation 6.1 indicates that the new function space not only preserves the nesting structure, but allows us to use a similar stencil for system upsampling/downsampling.

Implementation

As the proposed framework still uses regular B-splines centered at grid corners, most of the implementation remains unchanged, except the setting up of test functions. In order to obtain our test functions, we start by decomposing $\mathcal{M}$ into surface patches $\mathcal{S}$, such that each surface patch $s \in \mathcal{S}$ is contained entirely within a grid cell. The test functions are then defined by looking for connected components within the eight adjacent voxels around each grid corner i: test function $b_{(i,m)}$ is instantiated using a B-spline centered at i and supported exclusively on the m-th component of the support of b_i. Pseudocode of the process is given in Algorithm 6.1.

```
**Algorithm VI**: SetConnectivityAwareTestFunctions( )
for each grid corner $i$
    $m = 0$
    $Q = \{s \in \mathcal{S} \mid s$ resides in $i$'s adjacent voxels$\}$
    while $Q \neq \emptyset$
        $R_{i,m} \leftarrow Q.RemoveOne()$
        for each $s' \in Q$ adjacent to $R_{i,m}$
            $R_{i,m} \leftarrow R_{i,m} \cup s'$
            $Q.Remove(s')$
        $m \leftarrow m + 1$
        $b_{(i,m)}$ := B-spline centered at $i$ and supported on $R_{i,m}$
return $\{b_{(i,m)}\}$
```

Algorithm 6.1: Algorithm for adaptively setting up test functions.

6.2 Results

The increased richness of the new function space is highlighted in Figure 6.2, where the texture defined on the self-intersecting knot model is projected onto both the adaptive and non-adaptive bases. For this case, there does not exist any continuous 3D function whose restriction to the mesh can closely represent the texture.

In the rest of the chapter, we further evaluate how this adaptive, connectivity-aware function space improves the performance of our grid-based finite elements system. In particular, we revisit the spectral analysis from Chapter 4, examine the resulting multigrid convergence, and compare the efficiency of the new solver with the state-of-the-art CHOLMOD solver [9].

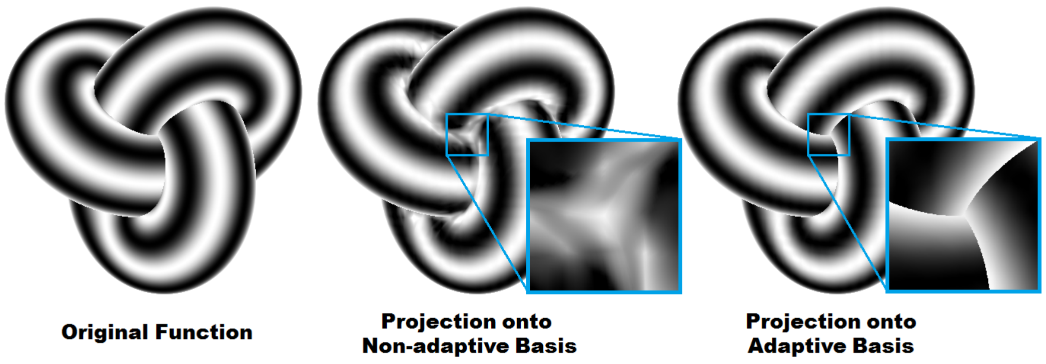

Original Function
Projection onto Non-adaptive Basis
Projection onto Adaptive Basis

Figure 6.2: Increased richness of the proposed function space. A “knot” model with a stripe texture (left) is projected onto the original function space (middle) and the adaptive function space (right). Due to the coupling of function values, the non-adaptive approach fails to reproduce the correct texture when points are close in Euclidean space but are geodesically distant. By designing the new function space to be aware of local connectivity, we can fit the geodesically distant patches independently, resulting in an accurate reproduction of the original texture.

6.2.1 Spectral Analysis

In Chapter 4, we leveraged spectral analysis for evaluating the correctness and quality of the Laplace-Beltrami operator estimated by our grid-based system. The results generally showed that our system was robust, with the exception of the pulley model that consists of narrow cross-sections - precisely where the coupling of values would occur. In this section, we investigate whether the new function space improves the spectral behavior.

For both the non-adaptive and adaptive spaces of functions, we compute the spectrum at increasing grid resolutions. The results are shown in Figure 6.3. As demonstrated in the plot, the spectra of the new operator quickly converge to the ground truth, which is estimated using the cotangent-weight operator defined over a dense tessellation of the mesh.

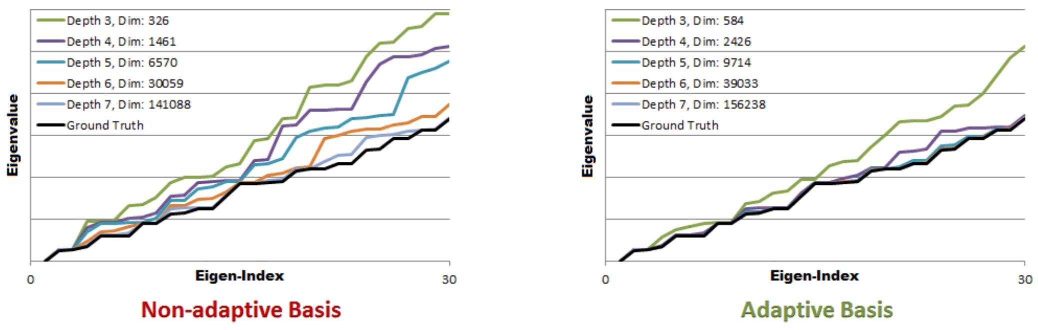


Figure 6.3: Stability of the spectrum of the Laplace-Beltrami operators estimated using the non-adaptive and adaptive function spaces. We compute the spectra of the non-adaptive operator (*left*) and the adaptive operator (*right*) at various grid resolutions for the pulley model. As the resolution increases, the adaptive spectra more quickly converge to the ground-truth.

As in Chapter 4, we also verify the isometry-invariance of the Laplacian. We apply random translations and rotations to the model before computing the spectrum. The results are shown in Figure 6.4, where the adaptive operator reproduces the spectrum more consistently than the non-adaptive operator.

6.2.2 Convergence Analysis

Next, we investigate how convergence behavior is improved by our adaptive function splitting. To make the residuals obtained using the non-adaptive and adaptive bases comparable, we use an input model (Figure 6.5, left) that does not require any splitting operation at the finest level of the function space (though it is still necessary at coarser ones). As in Chapter 4, we solve the screened-Poisson equation on the mesh to fit the color function. This time, we initialize the solution with random values between zero and one.

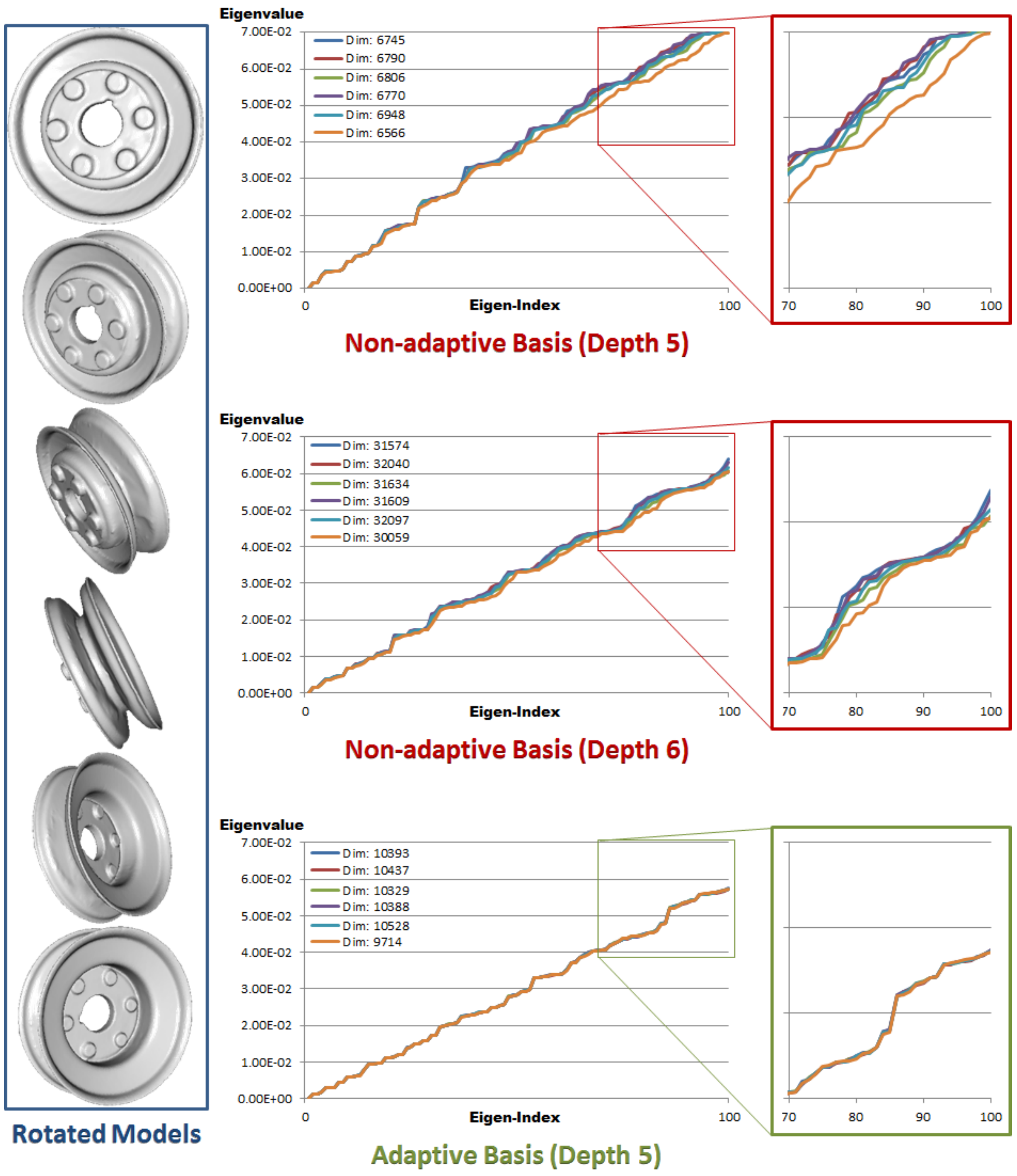


Figure 6.4: Isometry-invariance of the estimated Laplace-Beltrami operator. We compute the spectra of the non-adaptive operator (*top* and *middle*) and the adaptive operator (*bottom*) for the different rotations of the pulley model. The zoom-ins accentuate the superior stability of the adaptive operator (*right*).

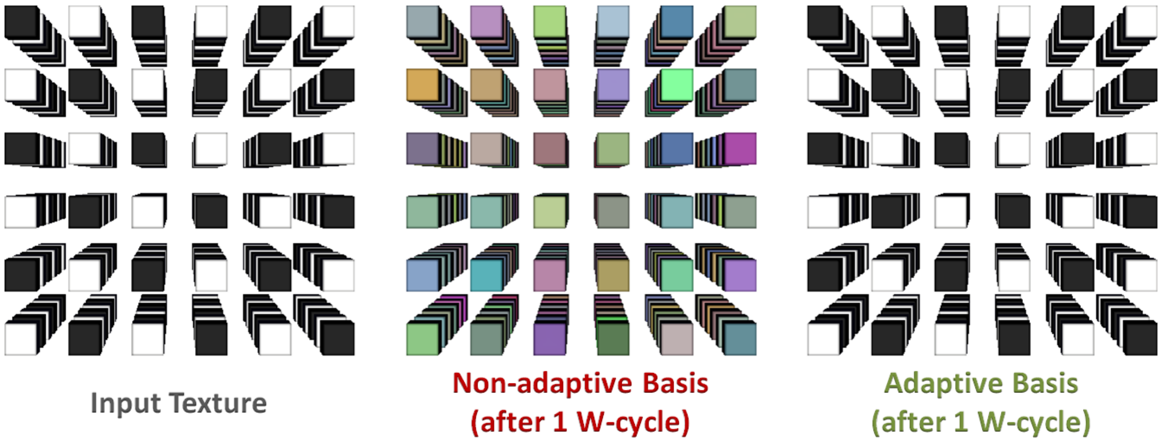

Input Texture
Non-adaptive Basis (after 1 W-cycle)
Adaptive Basis (after 1 W-cycle)

Figure 6.5: Color fitting of a 3D checker-board texture on the model consisting of equidistantly-spaced $6 \times 6 \times 6$ unit-cubes (*left*). The screened-Poisson equation with a screening weight $\lambda = 0.01$ is solved using a grid of depth 5. The coefficients of the initial guess are generated randomly with values between 0 and 1.

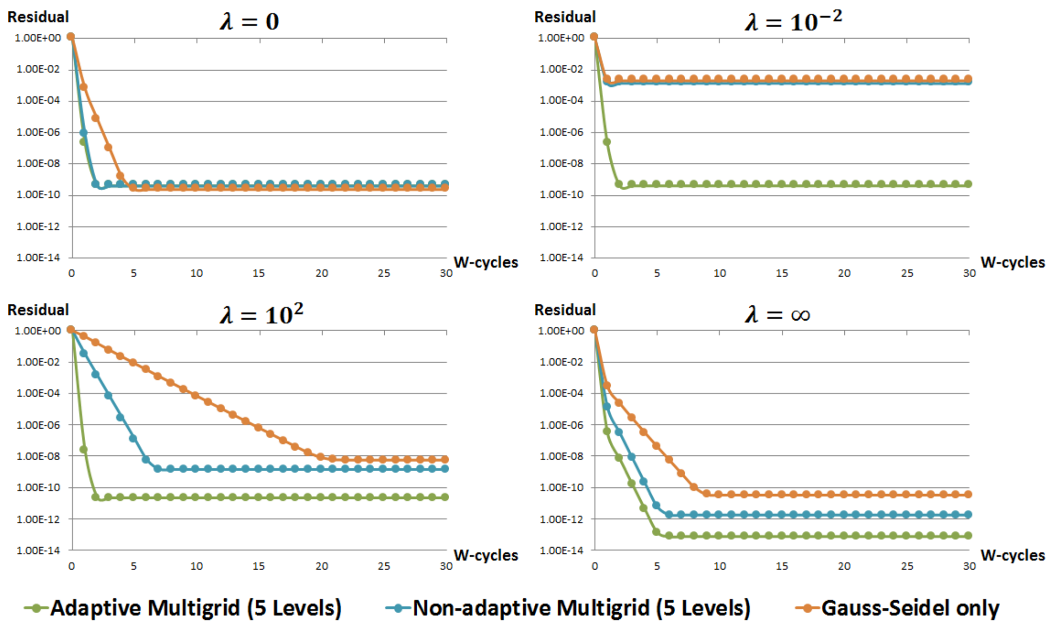


Figure 6.6: Convergence of the multigrid solvers. We solve the screened-Poisson equation for fitting the color function on the unit-cubes (Figure 6.5, left). We compare the convergence rates of the multigrid methods defined using the adaptive and non-adaptive bases. As the splitting operation is not needed at the finest grid resolution, the two bases define the same fine-level linear system (and thus the residuals are comparable).

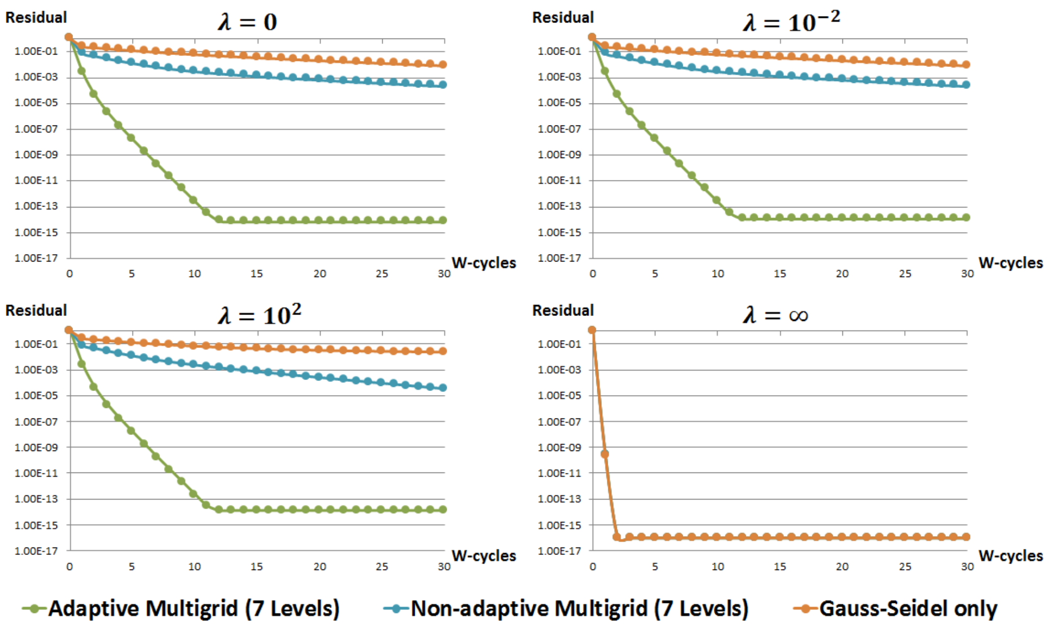


Figure 6.7: Convergence of the multigrid solvers. Here we use the knot model (Figure 6.2, left) as the input. The finest-level linear system is defined using the Hat-basis.

The visual results obtained using the two approaches are presented in Figure 6.5. Here, we observe that the non-adaptive solver does not produce a satisfactory solution, despite the lack of value coupling at the highest resolution. We believe this is because the coupling at coarser spaces prevents multigrid from generating a meaningful correction term.

More quantitative results of the experiment are presented in Figure 6.6, where we observe the convergence rates of the two solvers over 30 W-cycles. As expected, the adaptive solver has a superior convergence rate.

We also repeat the experiment where the hat basis is used to define the finest-level linear system. This time we use the knot model (Figure 6.2, left) as the input. (Although for this mesh the two grid-based systems are different at the highest res-

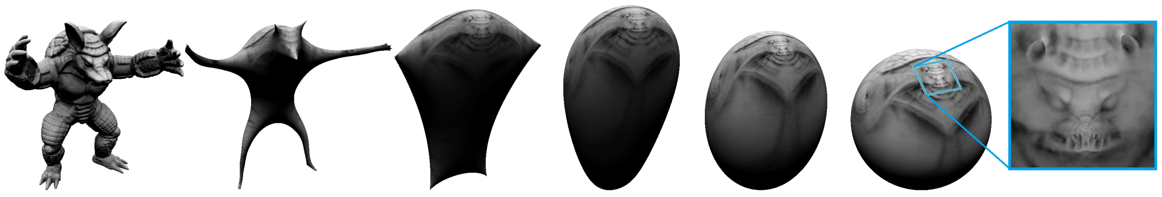


Figure 6.8: Conformalized Mean-Curvature Flow applied to the Armadillo Man. From left to right, we show the 0th, 1st, 3rd, 5th, 10th and 30th steps of the flow. Note that the flow conformally evolves the mesh to a sphere (right).

olution, the use of the same hat-basis to define the finest-level linear system allows us to compare the residuals.) The results are shown in Figure 6.7, where we again observe superior performance of the adaptive approach.

6.2.3 Surface Flow Application

As discussed in the previous chapters, the preprocessing time of our grid-based system is significant compared to the solver time. As a result, the approach is best suited for dynamic applications where the linear system changes over time. We thus evaluate the approach in this context.

Conformalized Mean Curvature Flow

In this section, we evaluate the proposed adaptive system using conformalized mean-curvature flow (cMCF) [81]. This recently proposed flow has been shown to converge to a conformal parameterization when acting on genus-zero surfaces (see Figure 6.8 for an example). At each time t, we solve a semi-implicit system as described in [26]:

$$\Big(D^t + \frac{\delta}{2}L^0\Big)\vec{x}^{t+\delta} = D^t\vec{x}^t \tag{6.2}$$

Experiment Setup

When performing surface flow, it is often necessary to consider the tradeoffs between the computational cost and the solution accuracy. Factors affecting the computational cost include the temporal stepsize (as taking smaller timesteps increases the temporal resolution but leads to longer running time) and the solver time per step (as using more accurate solvers increases the accuracy within each timestep but leads to longer running time). In practice, a natural question to ask is: *Given a budgeted computational cost, what is the best accuracy that can be achieved?* To answer the question, we designed the following experiment.

We first simulate the ground truth cMCF on a high resolution brain model consisting of 1.4 millions vertices (Figure 6.11, left). The evolution time is targeted at $t = 50$ and we take a tiny stepsize $\delta = 10^{-2}$ to flow the surface toward the target. At each time step, we use the hat-basis to define the system, which is then solved precisely using CHOLMOD. The simulation takes more than ten hours to generate all the evolved surfaces. Having computed these, we can later generate the ground truth at arbitrary time τ by linearly interpolating between the vertex positions at time $\lfloor\frac{\tau}{\delta}\rfloor\delta$ and $\lceil\frac{\tau}{\delta}\rceil\delta$.

Results

We consider the following configurations of systems and solvers:

- Hat-basis System solved by CHOLMOD
- Hat-basis System solved by AMG
- Non-adaptive grid-based system solved by CHOLMOD
- Non-adaptive grid-based system solved by AMG
- Non-adaptive grid-based system solved by Grid-based Multigrid
- Adaptive grid-based system solved by CHOLMOD
- Adaptive grid-based system solved by AMG
- Adaptive grid-based system solved by Grid-based Multigrid

To ensure the best performance of CHOLMOD, the symbolic factorization is only performed once in the preprocessing. For AMG, we choose Ruge-Stueben's classical AMG [3], as it was shown to have a better convergence rate in Chapter 4. [1] The tolerance of (relative) residual errors is set to 5×10^{-4} for both AMG and grid-based multigrids.

[1]We use the implementation of AMG (as stand-alone solvers) from *amgcl* [73] with the default parameters.

Configuration (System + Solver)		Dim.	NNZ	Time/Step	Steps
Hat	+ Cholmod	1.38×10^6	9.67×10^6	3.78	26
	+ AMG			3.09	32
Non-adaptive	+ Cholmod	1.21×10^6	2.22×10^7	11.08	9
	+ AMG			5.28	19
	+ Grid-MG			1.47	68
Adaptive	+ Cholmod	1.27×10^6	2.28×10^7	9.26	10
	+ AMG			5.02	20
	+ Grid-MG			1.52	66

Table 6.2: Statistics for the different configurations, giving the system dimension, the number of non-zero entries, the average time spent for each time step (including the time for updating the system/solver and the time for solving the system), the total number of steps, and the temporal step size δ.

The computational budget of each configuration is one hundred seconds. As the evolution time is fixed at $t = 50$, the temporal stepsize δ taken by each configuration depends on how quickly each spatial system can be solved, as summarized in Table 6.2.

We compare the evolved surfaces obtained from each configuration to the ground truth. Results are shown in Figure 6.9, where we plot the RMS error ($\sqrt{\sum_i \|v_i^e - v_i^g\|^2}$ with v^e and v^g the evolved and ground truth vertex positions) as a function of evolution time. The errors at the end of the flow are visualized in Figure 6.10.

Here we make two observations. First, although the direct solver is capable of computing exact solutions, the expensive cost prevents us from taking small time-steps and eventually leads to large errors. Second, the non-adaptive and adaptive systems perform equally well in the beginning (when the flow is essentially smoothing),

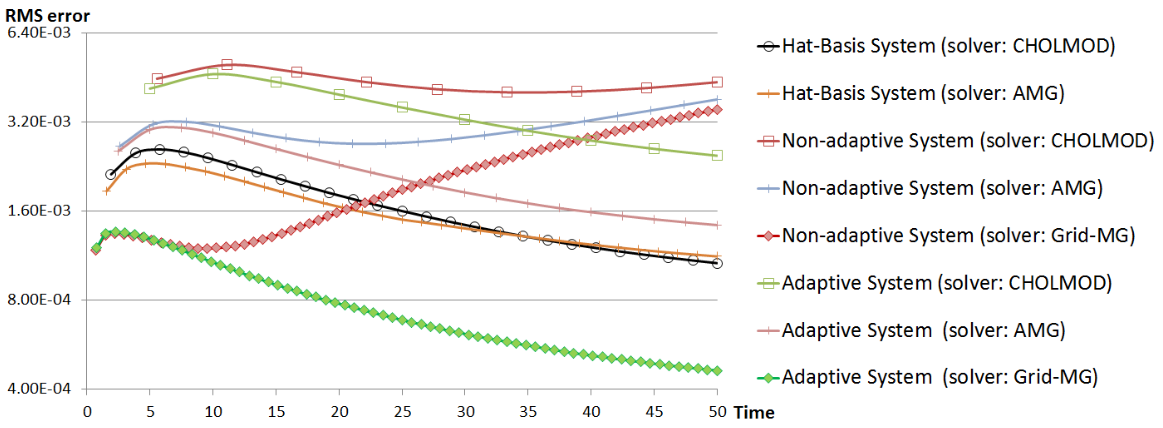


Figure 6.9: Error comparison of the different approaches performing *cMCF* on a brain model consisting of 1.4 million vertices. The RMS error is plotted as a function of evolution time. The ground truth is simulated using CHOLMOD to solve the hat-basis system taking a tiny stepsize $\delta = 0.01$. The computational budget is fixed at one hundred seconds, so that the number of steps (visualized by the tick marks) is determined by the efficiency of the solver.

but then the non-adaptive one deteriorates quickly with time (when the flow tries to pull apart the two hemispheres of the brain).

In Figure 6.11, we examine the two surfaces resulting from the non-adaptive and adaptive systems at the end of the flow (both are solved by the grid-based multigrid). The zoom-ins highlight the problem of the non-adaptive approach, where the values of the embedding function are coupled across the two hemispheres of the brain and cannot flow independently.

Discussion

Using the adaptive formulation, we resolve the problem of value coupling across disconnected components. However, the approach *cannot* decouple values for points on the *same* component. The problem is manifest when one uses a low-resolution grid

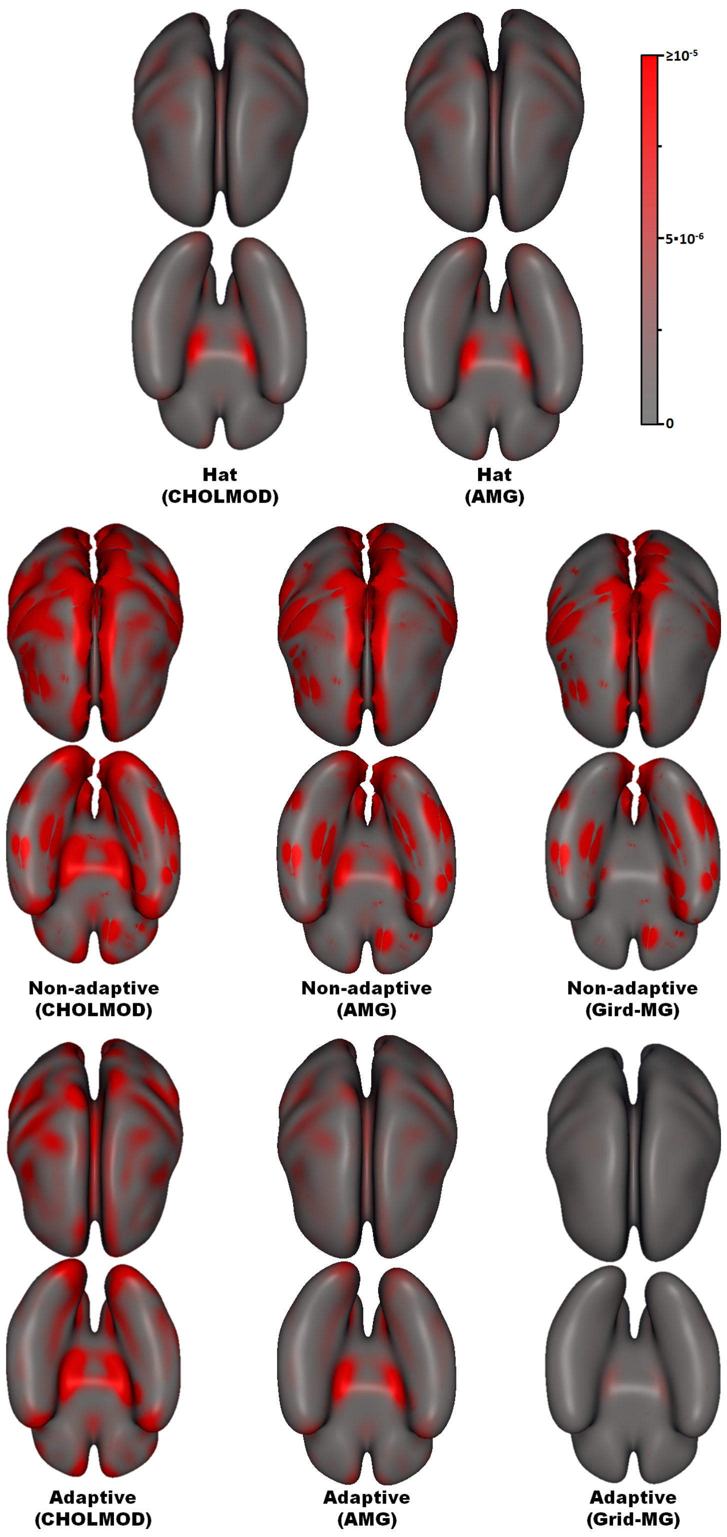


Figure 6.10: Error visualization of *cMCF* at $t = 50$. Renderings show L^2 distance to the ground truth. Note that for the non-adaptive systems, errors accumulate around "pinched" regions.

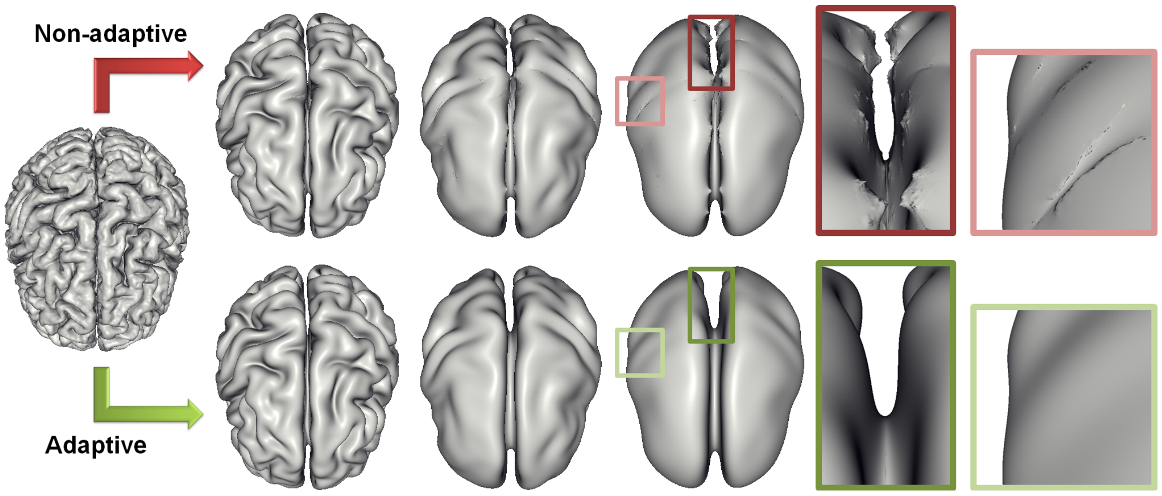


Figure 6.11: The brain undergoing *cMCF* with the non-adaptive (*top*) and adaptive (*bottom*) function spaces used to define the system for the input mesh (*left*). The two systems have about the same dimension and are both solved using multigrid. Here we show the evolved surfaces at $t = 10$, $t = 25$, and $t = 50$ (*middle*). The zoom-ins highlight the benefit of using the adaptive, connectivity-aware system, which is able to decouple the function values at points that are close in Euclidean space, allowing them to flow independently (*right*).

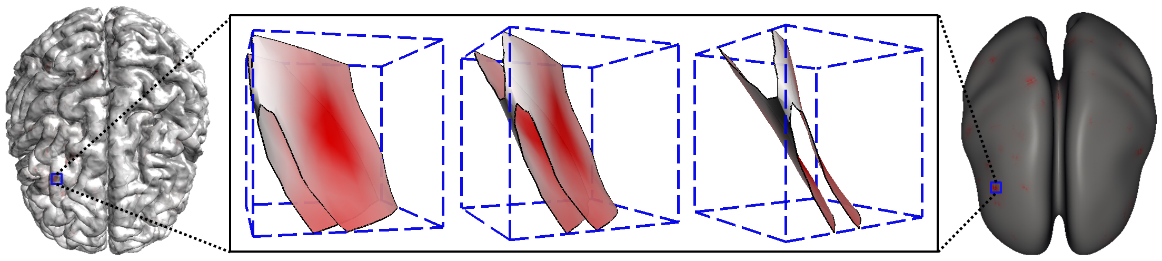


Figure 6.12: Value coupling within a connected component near a high curvature region of the brain (*left*). When we define our system using a lower resolution grid, there are basis functions supported on two parallel patches. Rendering the support from different perspectives, we observe that the function cannot be split because the two patches are connected at the corner (*middle*). As a result, performing *cMCF* using this low resolution system yields high errors (*right*, drawn as in Figure 6.10).

to define a linear system over surfaces with high-curvature regions.

As an example, Figure 6.12 visualizes the support of a basis function defined using a lower resolution grid. The support contains two flat regions that meet near a corner. Consequently, running cMCF at this low resolution results in pronounced errors in these regions.

In practice, however, we have only found this to be an issue when very coarse resolution grids are used. As demonstrated in this chapter, when the resolution of the grid matches that of the tessellation, our approach is both accurate and efficient.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have presented a novel finite elements system for solving the Poisson equation on meshes. Defining a function space by restricting regular 3D functions to the surface, as described in Chapter 4, we address several issues from previous work. Unlike traditional approaches, our method is tessellation-independent and gives direct control over system complexity. More importantly, the resulting function space comes with a multi-resolution structure supporting an efficient multigrid solver. From an implementation point of view, the regularity of the function space can be leveraged in parallelizing the computation.

The benefits of our system come with a price though. As the approach requires adapting an octree and clipping the input mesh, the overhead for setting up the system is not negligible. As a result, for applications where the mesh evolves with time, a straightforward implementation of the approach would be impractical. As we have shown in Chapter 5, by tracking the metric structure, one can efficiently evolve test functions with the mesh. This allows us to amortize and reuse much of the precomputation, avoiding a costly initialization of the multigrid hierarchy at the beginning of each time-step.

There has been significant research in the use of anisotropic diffusion for performing feature-aware surface editing. Such approaches often require modifying metrics and are difficult to apply in real-time. In this work, we have shown that, by leveraging the regularity of 3D functions and restricting ourselves to a rich subset of metric tensors, one can significantly speed up the computation and support anisotropic geometry processing on high resolution meshes in real-time.

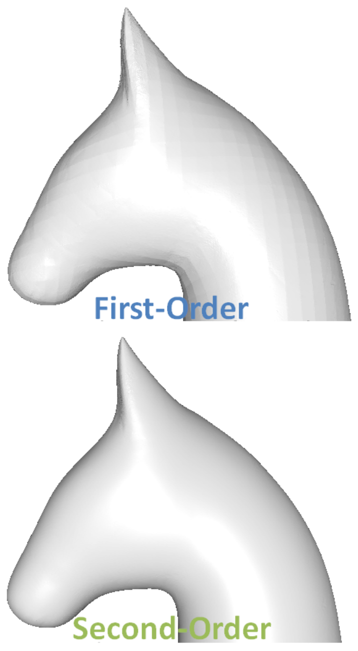
Finally, although the use of 3D functions provides our system with the desired regularity, it limits the richness of our function space because Euclidean distances are used as a proxy for geodesic proximity. To address this concern, in Chapter 6 we have proposed enriching the function space by splitting the existing 3D functions so that the support of each new function is connected. The extension makes our function space behave like an intrinsic one.

Throughout the thesis, we have conducted a set of experiments demonstrating the competitiveness of our approach. These include spectral analysis, convergence analysis, and efficiency analysis. We have shown that, when applied to various geometry-processing tasks, our approach is effective in solving dynamic linear systems and is well suited for real-time applications.

7.2 Future Work

Our grid-based approach gives rise to a new way of thinking about finite elements on surfaces. Though we have largely explored the idea in the thesis, there are still several venues for future research:

Higher-order Functions



In the thesis, we use first-order 3D B-splines as test functions. The functions have limited supports, resulting in a very sparse system. However, the C^0 nature of test functions could lead to artifacts when the resolution of the grid is low. The inset shows an example where we solve a screened-Poisson equation to dampen the gradients of the embedding (as in Chapter 5). Using first-order B-splines (top) results in artifacts at cell boundaries. On the other hand, using second-order B-splines (bottom) helps alleviate the limitation by providing C^1 continuity. Unfortunately, the use of higher-order B-splines reduces the sparsity of the system and makes it more expensive to construct and solve. We are still exploring ways to mitigate the increased cost so that the efficiency of the resulting solver can be practical.

Convergence Improvement

Throughout the thesis, we use Gauss-Seidel as the smoother. Although this works well in most cases, in Chapter 4, we have shown that the solver can converge slowly when the linear system is barely in general position and non-diagonally dominant. In future work, we would like to explore more effective smoothers tailored to our system. We would also like to consider adjusting our function space so that the resulting system becomes more diagonally dominant.

Many-core Implementation

In this work, we have chosen to pursue an implementation of the multigrid solver that is primarily CPU-based. In the future, we would like to consider a GPU-based implementation leveraging the regularity of the function space. Though conceptually similar to the volumetric Poisson solver proposed in [82], extending our approach to the GPU is more challenging, because in the volumetric system the matrix coefficients depend only on relative cell positions (and thus can be compactly represented by a stationary stencil). On the other hand, our surface-based Poisson system requires the explicit computation and storage of matrix coefficients, since their values depend on how the surface passes through the grid cells and on the spatially-varying anisotropy.

Out-of-core Implementation

We would also like to extend our approach to an out-of-core implementation. In particular, taking as input an out-of-core mesh that has been clipped and stored in the *streaming mesh* format [83], the system coefficients can be computed in a streaming fashion. Because the grid-based system is independent of mesh tessellation, the computation of the multigrid hierarchy is easier than in previous work [66], as the complicated analysis of irregular connectivity can be avoided. Having system coefficients sorted by heights of corresponding basis functions, one can leverage the our-of-core temporal blocking technique developed for regular domains and perform k V-cycles in $k + 1$ streaming passes [1].

Bibliography

[1] M. Kazhdan and H. Hoppe, "Streaming multigrid for gradient-domain operations on large images," *ACM Transactions on Graphics (SIGGRAPH '08)*, vol. 27, no. 3, pp. 21:1–21:10, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360620

[2] B. Aksoylu, A. Khodakovsky, and P. Schröder, "Multilevel solvers for unstructured surface meshes," *SIAM Journal of Scientific Computing*, vol. 26, pp. 1146–1165, 2005.

[3] J. Ruge and K. Stueben, "Algebraic multigrid," *Frontiers in Applied Mathematics*, vol. 3, pp. 73–130, 1987.

[4] P. Vanek, J. Mandel, and M. Brezina, "Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems," *COMPUTING*, vol. 56, pp. 179–196, 1996.

[5] P. Pérez, M. Gangnet, and A. Blake, "Poisson image editing," *ACM Transactions on Graphics (SIGGRAPH '03)*, pp. 313–318, 2003.

[6] O. Sorkine, D. Cohen-Or, Y. Lipman, C. Rossl, and H. Seidel, “Laplacian surface editing,” in *Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, 2004, pp. 179–188.

[7] M. Kazhdan, M. Bolitho, and H. Hoppe, “Poisson surface reconstruction,” in *Proceedings of the fourth Eurographics Symposium on Geometry processing*, ser. SGP ’06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 61–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=1281957.1281965

[8] LIS V.1.2.52, “Lis: a library of iterative solvers for linear systems,” http://www.ssisc.org/lis/, 2010.

[9] T. A. Davis, “User guide for CHOLMOD: a sparse cholesky factorization and modification package,” http://www.cise.ufl.edu/research/sparse/cholmod/CHOLMOD/Doc/UserGuide.pdf, 2011.

[10] M. Botsch, D. Bommes, and L. Kobbelt, “Efficient linear system solvers for mesh processing,” in *Proceedings of the 11th IMA international conference on Mathematics of Surfaces*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 62–83.

[11] B. Horn, “Determining lightness from an image,” *Computer Graphics and Image Processing*, vol. 3, pp. 277–299, 1974.

[12] Y. Weiss, "Deriving intrinsic images from image sequences," in *International Conference on Computer Vision*, 2001, pp. 68–75.

[13] G. Finlayson, S. Hordley, and M. Drew, "Removing shadows from images," in *European Conference on Computer Vision*, 2002, pp. 129–132.

[14] R. Fattal, D. Lischinksi, and M. Werman, "Gradient domain high dynamic range compression," in *ACM SIGGRAPH*, vol. 21, 2002, pp. 249–256.

[15] A. Levin, A. Zomet, S. Peleg, and Y. Weiss, "Seamless image stitching in the gradient domain," in *European Conference on Computer Vision*, 2004, pp. 377–389.

[16] A. Agarwala, M. Dontcheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin, and M. Cohen, "Interactive digital photomontage," *ACM Transactions on Graphics (SIGGRAPH '04)*, pp. 294–302, 2004.

[17] J. Sun, J. Jia, C.-K. Tang, and H.-Y. Shum, "Poisson matting," *ACM Transactions on Graphics (SIGGRAPH '04)*, vol. 23, no. 3, pp. 315–321, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015721

[18] A. A. Gooch, S. C. Olsen, J. Tumblin, and B. Gooch, "Color2gray: salience-preserving color removal," *ACM Transactions on Graphics (SIGGRAPH '05)*, vol. 24, no. 3, pp. 634–639, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1073204.1073241

[19] D. Lischinski, Z. Farbman, M. Uyttendaele, and R. Szeliski, “Interactive local adjustment of tonal values,” *ACM Transactions on Graphics (SIGGRAPH ’06)*, vol. 25, no. 3, pp. 646–653, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1141911.1141936

[20] P. Bhat, C. L. Zitnick, M. Cohen, and B. Curless, “Gradientshop: A gradient-domain optimization framework for image and video filtering,” *ACM Trans. Graph.*, vol. 29, no. 2, pp. 10:1–10:14, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1731047.1731048

[21] A. Orzan, A. Bousseau, P. Barla, and J. Thollot, “Structure-preserving manipulation of photographs,” in *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, 2007, pp. 103–110.

[22] J. McCann and N. Pollard, “Real-time gradient-domain painting,” *ACM Transactions on Graphics (SIGGRAPH ’08)*, 2008.

[23] A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin, “Diffusion curves: a vector representation for smooth-shaded images,” *ACM Transactions on Graphics (SIGGRAPH ’08)*, vol. 27, no. 3, pp. 92:1–92:8, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360691

[24] G. Taubin, “A signal processing approach to fair surface design,” in *ACM SIGGRAPH*, 1995, pp. 351–358.

[25] L. Kobbelt, S. Campagna, J. Vorsatz, and H.-P. Seidel, “Interactive multi-resolution modeling on arbitrary meshes,” in *ACM SIGGRAPH*, 1998.

[26] M. Desbrun, M. Meyer, P. Schröder, and A. Barr, “Implicit fairing of irregular meshes using diffusion and curvature flow,” in *ACM SIGGRAPH Conference Proceedings*, 1999, pp. 317–324.

[27] U. Clarenz, U. Diewald, and M. Rumpf, “Anisotropic geometric diffusion in surface processing,” in *Visualization ’00: Proceedings of the 11th IEEE Visualization 2000 Conference (VIS 2000)*, 2000, pp. 497–405.

[28] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr, “Discrete differential-geometry operators for triangulated 2-manifolds,” in *Visualization and Mathematics*, Berlin, Germany, 2002.

[29] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher, “Geometric surface smoothing via anisotropic diffusion of normals,” in *Proceedings of the conference on Visualization ’02*, ser. VIS ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 125–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=602099.602117

[30] C. L. Bajaj and G. Xu, “Anisotropic diffusion of surfaces and functions on surfaces,” *ACM Trans. Graph.*, vol. 22, no. 1, pp. 4–32, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/588272.588276

[31] O. Sorkine and D. Cohen-Or, “Least-squares meshes,” in *Proceedings of Shape Modeling International.* IEEE Computer Society Press, 2004, pp. 191–199.

[32] A. Nealen, T. Igarashi, O. Sorkine, and M. Alexa, “Laplacian mesh optimization,” in *Proceedings of ACM GRAPHITE*, 2006, pp. 381–389.

[33] K. Hildebrandt and K. Polthier, “Constraint-based fairing of surface meshes,” in *Proceedings of the fifth Eurographics Symposium on Geometry Processing*, ser. SGP ’07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 203–212. [Online]. Available: http://dl.acm.org/citation.cfm?id=1281991.1282019

[34] M. Alexa, “Differential coordinates for local mesh morphing and deformation,” *The Visual Computer*, vol. 19, pp. 105–114, 2003.

[35] Y. Lipman, O. Sorkine, D. Cohen-Or, D. Levin, C. Rössl, and H.-P. Seidel, “Differential coordinates for interactive mesh editing,” in *Shape Modeling International*, 2004, pp. 181–190.

[36] Y. Yu, K. Zhou, D. Xu, X. Shi, H. Bao, B. Guo, and H.-Y. Shum, “Mesh editing with Poisson-based gradient field manipulation,” *ACM Transactions on Graphics (SIGGRAPH ’04)*, vol. 23, pp. 644–651, 2004.

[37] J. Huang, X. Shi, X. Liu, K. Zhou, L.-Y. Wei, S.-H. Teng, H. Bao, B. Guo, and H.-Y. Shum, “Subspace gradient domain mesh deformation,” *ACM*

Trans. Graph., vol. 25, no. 3, pp. 1126–1134, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1141911.1142003

[38] L. Shi, Y. Yu, N. Bell, and W. wen Feng, "A fast multigrid algorithm for mesh deformation," *ACM Transactions on Graph*, vol. 25, no. 3, pp. 1108–1117, 2006.

[39] O. Sorkine and M. Alexa, "As-rigid-as-possible surface modeling," in *Proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing*, 2007, pp. 109–116.

[40] M. Botsch and O. Sorkine, "On linear variational surface deformation methods," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 1, pp. 213–230, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2007.1054

[41] B. Vallet and B. Lévy, "Spectral geometry processing with manifold harmonics," *Computer Graphics Forum (Eurographics '08)*, vol. 2, 2008.

[42] M. Reuter, F.-E. Wolter, and N. Peinecke, "Laplace-spectra as fingerprints for shape matching," in *Symposium on Solid and Physical Modeling*, 2005, pp. 101–106.

[43] R. M. Rustamov, "Laplace-beltrami eigenfunctions for deformation invariant shape representation," in *Symposium on Geometry Processing*, 2007, pp. 225–233.

[44] M. Ovsjanikov, J. Sun, and L. Guibas, “Global intrinsic symmetries of shapes,” *Computer Graphics Forum (SGP ’08)*, vol. 27, pp. 1341–1348, 2008.

[45] J. Sun, M. Ovsjanikov, and L. Guibas, “A concise and provably informative multi-scale signature based on heat diffusion,” in *Eurographics Symposium on Geometry Processing (SGP)*, 2009.

[46] F. de Goes, S. Goldenstein, and L. Velho, “A hierarchical segmentation of articulated bodies,” in *Proceedings of the Symposium on Geometry Processing*, ser. SGP ’08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 1349–1356. [Online]. Available: http://dl.acm.org/citation.cfm?id=1731309.1731315

[47] Y. Lipman, R. Rustamov, and T. Funkhouser, “Biharmonic distance,” *ACM Transactions on Graphics*, vol. 29, no. 3, Jun. 2010.

[48] H. Zhang, “Discrete combinatorial laplacian operators for digital geometry processing,” in *SIAM Conference on Geometric Design, 2004*. Press, 2004, pp. 575–592.

[49] U. Pinkall and K. Polthier, “Computing discrete minimal surfaces and their conjugates,” *Experimental Mathematics*, vol. 2, pp. 15–36, 1993.

[50] M. S. Floater, “Mean value coordinates,” *Computer Aided Geometric Design*, vol. 20, p. 2003, 2003.

[51] M. Alexa and M. Wardetzky, “Discrete laplacians on general polygonal meshes,” *ACM Trans. Graph.*, vol. 30, no. 4, pp. 102:1–102:10, Jul. 2011. [Online]. Available: http://doi.acm.org/10.1145/2010324.1964997

[52] M. Fisher, B. Springborn, A. I. Bobenko, and P. Schroder, “An algorithm for the construction of intrinsic delaunay triangulations with applications to digital geometry processing,” in *ACM SIGGRAPH 2006 Courses*, ser. SIGGRAPH ’06, 2006, pp. 69–74.

[53] A. I. Bobenko and B. A. Springborn, “A discrete Laplace-Beltrami operator for simplicial surfaces,” *Discrete Comput. Geom.*, vol. 38, no. 4, pp. 740–756, Dec. 2007.

[54] M. Wardetzky, S. Mathur, F. Kälberer, and E. Grinspun, “Discrete laplace operators: no free lunch,” in *Proceedings of the fifth Eurographics symposium on Geometry processing*, ser. SGP ’07. Eurographics Association, 2007, pp. 33–37.

[55] T. A. Davis and W. W. Hager, “Modifying a sparse cholesky factorization,” *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 606–627, 1999.

[56] T. A. Davis, “Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method,” *ACM Transactions On Mathematical Software*, vol. 30, no. 2, pp. 196–199, 2004.

[57] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 302–325, September 2005.

[58] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.).* Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[59] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial.* Society for Industrial and Applied Mathematics, 2000.

[60] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J. W. Ruge, "Robustness and scalability of algebraic multigrid," *SIAM Journal of Scientific Computing*, vol. 21, no. 5, pp. 1886–1908, 2000.

[61] M. Brezina, J. Cleary, R. Falgout, V. Henson, J. Jones, T. Manteuffel, S. McCormick, and J. Ruge, "Algebraic multigrid based on element interpolation (AMGe)," *SIAM Journal of Scientific Computing*, vol. 22, no. 5, pp. 1570–1592, 2001.

[62] T. Chartier, R. D. Falgout, V. E. Henson, J. Jones, T. Manteuffel, S. McCormick, J. Ruge, and P. S. Vassilevski, "Spectral AMGe (ρAMGe)," *SIAM Journal of Scientific Computing*, vol. 25, no. 1, pp. 1–26, 2003.

[63] L. Kobbelt, S. Campagna, J. Vorsatz, and H. Seidel, "Interactive multi-resolution

modeling on arbitrary meshes," in *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998, pp. 105–114.

[64] R. Schneider and L. Kobbelt, "Geometric fairing of irregular meshes for free-form surface design," *Computer Aided Geometric Design*, vol. 18, pp. 359–379, 2001.

[65] N. Ray and B. Levy, "Hierarchical least squares conformal map," in *Pacific Graphics*, 2003, p. 263.

[66] X. Shi, H. Bao, and K. Zhou, "Out-of-core multigrid solver for streaming meshes," *ACM Transactions on Graphics (SIGGRAPH Asia '09)*, vol. 28, no. 5, 2009.

[67] K. Höllig, *Finite Elements Methods with B-splines.* Philadelphia, PA: Society for Industrial and Applied Mathematics, 2003.

[68] D. Day and M. Taylor, "A new 11 point degree 6 cubature formula for the triangle," *Sixth International Congress on Industrial Applied Mathematics (ICIAM07) and GAMM Annual Meeting*, vol. 7, pp. 1 022 501–1 022 502, 2007.

[69] C. Cowper, "Gaussian quadrature formulas for triangles," *International Journal of Numerical Methods in Engineering*, vol. 7, pp. 405–408, 1973.

[70] C. Dick, J. Georgii, and R. Westermann, "A hexahedral multigrid approach for

simulating cuts in deformable objects," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 11, pp. 1663–1675, 2011.

[71] B. Smith, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[72] T. S. S. I. Project, http://www.caam.rice.edu/software/ARPACK/, 1997.

[73] amgcl, "`https://github.com/ddemidov/amgcl`," 2013.

[74] I. Math Kernel Library 10.3, "http://software.intel.com/en-us/articles/intel-mkl/," 2012.

[75] H. B. Keller, *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, vol. 2, no. 2, pp. 281–290, 1965.

[76] J. Wu, Y. Lee, J. Xu, and L. Zikatanov, "Convergence analysis of iterative methods for semidefinite systems." *Journal of Computational Mathematics*, vol. 26, no. 6, pp. 797–815, 2008.

[77] O. Au, C. Tai, H. Chu, D. Cohen-Or, and T. Lee, "Skeleton extraction by mesh contraction," *ACM Transactions on Graphics (SIGGRAPH '08)*, vol. 27, no. 3, 2008.

[78] Metro V.4.07, "http://vcg.soureforge.net/index.php/Metro," 2007.

[79] P. Bhat, B. Curless, M. Cohen, and L. Zitnick, "Fourier analysis of the 2D

screened Poisson equation for gradient domain problems," in *European Conference on Computer Vision*, 2008, pp. 114–128.

[80] T. . S. Rusinkiewicz), "http://gfx.cs.princeton.edu/proj/trimesh2/," 2009.

[81] M. Kazhdan, J. Solomon, and M. Ben-Chen, "Can Mean-Curvature Flow be Modified to be Non-singular?" *Computer Graphics Forum (Symposium on Geometry Processing)*, 2012.

[82] K. Zhou, M. Gong, X. Huang, and B. Guo, "Data-parallel octrees for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 5, pp. 669–681, May 2011. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2010.75

[83] M. Isenburg and P. Lindstrom, "Streaming meshes," in *Proceedings of Visualization.* IEEE Computer Society, 2005, pp. 231–238.

Vita

Ming Chuang was born in Taiwan, August 5, 1982. He received his B.S. degree in Computer Science and Information Engineering from the National Central University, Taiwan, in 2004. He was commissioned as a second lieutenant in the ROC Army from 2004 to 2006. He received his M.S.E. degree in Computer Science from the Johns Hopkins University in 2008. He joined the Computer Graphics Group at the Johns Hopkins University in 2008 as a PhD student. His research mainly focused on digital geometry processing and parallel computing. At Hopkins, he had been a TA for several classes, including Algorithms and Computer Graphics. From June 2011 to December 2011, he worked for Pixar Animation Studio as a research intern, where he developed experimental tools for animation artists.