UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA APLICADA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LUIZ FERNANDO SCHEIDEGGER

Robust Inter-surface Mapping Using Constrained Laplacian Methods

Trabalho de Graduação

João Luiz Dihl Comba
Orientador

Porto Alegre, December 2010

ACKNOWLEDGEMENTS

This work is the direct culmination of over 4000 hours of dedication and study, during which I may have been absent, and have demanded much from my closest relationships. This is the moment in which I want to thank all these people for their patience and unconditional support.

I wish to thank, first of all, my parents, for their love and personal incentive during my whole life. I also want to thank my advisor, Prof. João Comba, for providing me with motivation and for giving me the honor of working with him for most of my undergraduate years. I have learned much from him, and sincerely hope this work does justice to that. My whole family and friends deserve a special acknowledgement for making my day-to-day life enjoyable during times of increased academic pressure. I am also very grateful for the many discussions and constructive criticisms of two individuals who helped me develop the ideas in this text: Carlos Scheidegger, who happens to be my older brother, and Prof. Luís Gustavo Nonato, whose ideas are key to much of the work presented here.

[Computer Science] is also not really very much about computers. [...] It's not about computers in the same sense that physics is not really about particle accelerators, and biology is not really about microscopes and Petri dishes.

— Dr. Harold Abelson, excerpt from *6.001 – Introductory Undergraduate Subjects in Computer Science*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\mathcal{M}, \mathcal{M}_1, \mathcal{M}_2$	Triangle meshes
$\mathcal{V}_\mathcal{M}$	Set of $\mathcal{M}$'s vertices
$\mathcal{E}_\mathcal{M}$	Set of $\mathcal{M}$'s edges
$\mathcal{F}_\mathcal{M}$	Set of $\mathcal{M}$'s faces
$x_0, x_1, \ldots, x_{n-1}$	Components of an n-dimensional vector x
$x^0, x^1, \ldots, x^{m-1}$	Elements in a sequence of m vectors
$\langle x, y \rangle$	Inner product between x and y
$\|x\|$	The Euclidean norm of x ($\sqrt{\langle x, x \rangle}$)
$\hat{y}$	Normalized version of vector y ($\hat{y} = \frac{y}{\|y\|}$)
a_i^j	Entry in matrix A's i_{th} row and j_{th} column
A^T	Matrix transpose of A
A^{-1}	Matrix inverse of A
$\{\mathbf{i}, \mathbf{j}, \mathbf{k}, \ldots\}$	Canonical basis vectors for $\mathbb{R}^n$
$\mathsf{x}^0, \mathsf{x}^1, \ldots, \mathsf{x}^{n-1}$	Sequence of eigenvectors of a matrix
$\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$	Sequence of eigenvalues of a matrix
$\Delta f, \quad \nabla^2 f$	The Laplacian operator on f
∇f	The gradient of f
α_{ij}, β_{ij}	Opposite angles of the edge between v_i and v_j
$\mathcal{N}_1(v)$	One-neighborhood around v
$\mathcal{M}'$	Least-squares approximation of $\mathcal{M}$
$\mathcal{M}_\mathcal{C} = \{c^0, c^1, \ldots, c^{k-1}\}$	Set of k least-squares constraints for $\mathcal{M}$
$\mathcal{NC}_\mathcal{M}$	Set of new constraints for $\mathcal{M}$
$B_\mathcal{M}$	Least-squares basis for functions over $\mathcal{M}$

ABSTRACT

Triangle meshes are an extremely widespread way to digitally represent geometric data. With the popularization of widely available graphics-related computational power, the size and complexity of datasets has increased dramatically. Today it is not uncommon to find datasets that contain hundreds of thousands or even a few million triangles. Moreover, improvements in shape acquisition technology drastically increase the amount of geometric data available for use. This increase in complexity introduces a critical need to find spatial and topological relationships between different datasets.

In this context, we introduce a novel inter-surface mapping algorithm, which builds a smooth correspondence between two different triangle meshes. Our technique builds a map that allows mesh morphing and detail transfer, among other applications. Unlike previous work, our method does not explicitly require both models to be topologically equivalent; instead, when the models are not homeomorphic, we find a visually pleasing correspondence (since an exact correspondence is, by definition, impossible). We rely on some degree of user interaction to guide the mapping process, where the user defines a sparse set of initial correspondences between the two meshes, using a point-and-click interface.

Our method is entirely based on building a least-squares approximation of the prescribed meshes, using a set of user-defined correspondences, as well as automatically generated constraints. By forcing all vertex coordinates to satisfy Laplace's equation we ensure that points not explicitly constrained are evenly distributed over the meshes.

Keywords: Inter-surface mapping, Mesh parameterization, Laplace's equation, Least-squares meshes.

Mapeamento Inter-Superfícies Robusto Utilizando Métodos Laplacianos com Restrições

RESUMO

Malhas de triângulos são formas extremamente populares de se representar informação geométrica digitalmente. Com o aumento da disponibilidade de poder computacional gráfico, o tamanho e complexidade dos conjuntos de dados vem aumentando dramaticamente. Hoje em dia, não é incomum encontrar conjuntos de dados que possuem centenas de milhares ou até mesmo alguns milhões de triângulos. Além disso, melhorias em tecnologia de aquisição de objetos 3D aumentam drasticamente a quantidade de conjuntos de dados disponíveis para uso. Quando lidamos com tantos modelos com complexidade e tamanho tão variáveis, a necessidade de se determinar relações topológicas e geométricas entre malhas torna-se crítica.

Neste contexto, nós introduzimos um novo algoritmo para mapeamento de superfícies, que constrói uma correspondência entre duas malhas de triângulos distintas. Nossa técnica constrói um mapa que permite fazer o *morphing* de malhas, bem como transferência de detalhes, entre outras aplicações. Ao contrário de trabalhos anteriores, nosso método não requer que ambos os modelos sejam topologicamente equivalentes; ao invés disso, quando os modelos não são homeomórficos, nossa técnica determina um mapeamento que seja visualmente agradável (já que uma correspondência exata é, por definição, impossível). Nós precisamos de alguma interação com o usuário para guiar o processo de mapeamento, onde o usuário define um conjunto esparso de correspondências iniciais entre as duas malhas utilizando uma interface *point-and-click*.

Nosso método é baseado inteiramente em construir uma aproximação de mínimos-quadrados das malhas de entrada, utilizando um conjunto de restrições informado pelo usuário, bem como correspondências geradas automaticamente. Utilizando a equação de Laplace para posicionar vértices que não foram explicitamente restringidos, nosso método garante que estes pontos acabem distribuídos de forma bem comportada ao longo das malhas resultantes.

Palavras-chave: Mapeamento inter-superfícies, Parametrização de malhas, Equação de Laplace, Malhas de mínimos quadrados.

1 INTRODUCTION

Computer-aided geometric modeling is a firmly established technique among artists and members of the computer graphics community alike. With the continuing advance of widely available computational resources such as consumer-level Graphics Processing Units (GPUs), the complexity of digital models increases at a very fast pace. Today it is not uncommon to find models composed of hundreds of thousands or even a few million triangles. Moreover, advances in 3D scanner technology enable artists to obtain high-quality digital representations of real-life objects. Finally, the popularity of implicit surface extraction algorithms such as the ubiquitous marching methods [30, 20] enables the automatic construction of digital models from other representations, such as parametric functions and volume data. This increase in complexity introduces a critical need to find spatial and topological relationships between datasets.

Inter-surface mapping techniques exist to solve this exact problem: to determine a spatial relationship between different geometric models. This relationship defines a continuous map between the two models, enabling detail transfer (such as colors, normals or textures), as well as morphing. Naturally, the map should identify common points between the models: it does not make sense, for instance, to map one model's fingers to another's toes, and so on. However, it is very difficult to find a high quality mapping in a completely automatic fashion. Therefore, most mapping techniques, including ours, rely on some degree of user interaction to identify key points that must be mapped between the models.

In order to discuss inter-surface mapping in more detail, we need to formally define triangle meshes, which are our choice of representation for digital models in this work. Triangle meshes are ubiquitous as representations of geometric data in the computer graphics community, due to their simplicity and flexibility.

1.1 Triangle Meshes

A triangle mesh $\mathcal{M}$ is a set of triangles embedded in $\mathbb{R}^3$ with 2-manifold connectivity (2-manifolds are mathematical formalizations of continuous 2-dimensional surfaces). This means that not all sets of triangles form a triangle mesh: in particular, so-called T-junctions cannot exist in a mesh, as they directly violate the 2-manifold property. Formally, a mesh $\mathcal{M}$ consists of a set $\mathcal{V}_\mathcal{M}$ of vertices, a set $\mathcal{E}_\mathcal{M}$ of edges and a set $\mathcal{F}_\mathcal{M}$ of triangular faces. In order to guarantee that the mesh is a 2-manifold, each edge must be incident to either one or two faces (an edge that is incident to only one face is part of the mesh's boundary).

Triangle meshes are extremely versatile, and can be used to represent geometry

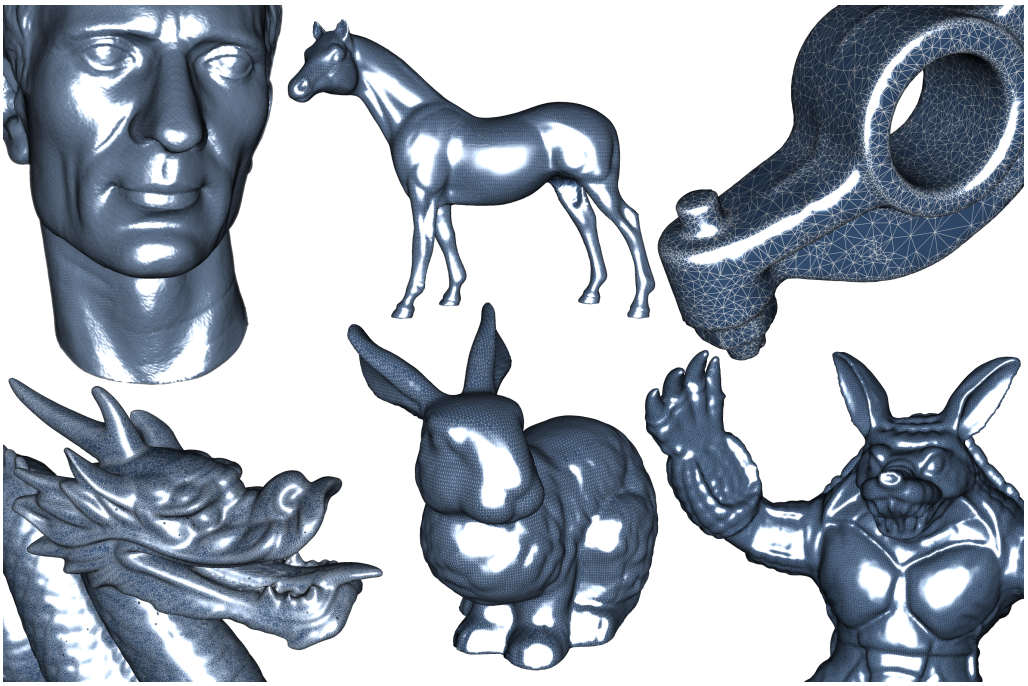


Figure 1.1: **A Gallery of Triangle Meshes.** Triangle meshes provide an extremely versatile way to digitally represent surface data. They can be used to describe models as varied as human and animal figures (top right and center, respectively), fictitious creatures (bottom left and bottom right), and mechanical shapes from CAD software (top right).

ranging from humanoid shapes to digital CAD models (see Figure 1.1). Moreover, they form convenient discretizations of surfaces embedded in $\mathbb{R}^3$, so that continuous properties such as normals, mean curvatures and other differential operators can be properly defined for triangle meshes. In this work, we are particularly interested in the Laplace-Beltrami operator and its discretizations over meshes, as Laplace's equation provides a convenient way to define smooth functions over surfaces (for more details concerning the Laplacian operator, refer to Chapter 4). In particular, we use Laplace's equation to approximate a given mesh using a set of user-defined constraints.

1.2 Inter-surface Mapping

Since different triangle meshes may have widely varying numbers of elements, inter-surface mapping techniques cannot simply define a one-to-one mapping between vertices or triangles. Instead, they must rely on more complex data structures to specify the mapping. Our approach mitigates these issues by framing the mapping problem as a linear-algebraic change of basis, which can be computed simply by solving a sparse linear system.

Unlike previous approaches to inter-surface mapping, our technique enforces user-defined constraints only in a least-squares sense. This relaxation allows us to build our algorithm around a set of sparse, symmetric linear systems, which can be easily solved using Cholesky decomposition. Moreover, these relaxed restrictions allow us to define a "reasonable" mapping even between surfaces of different genus.

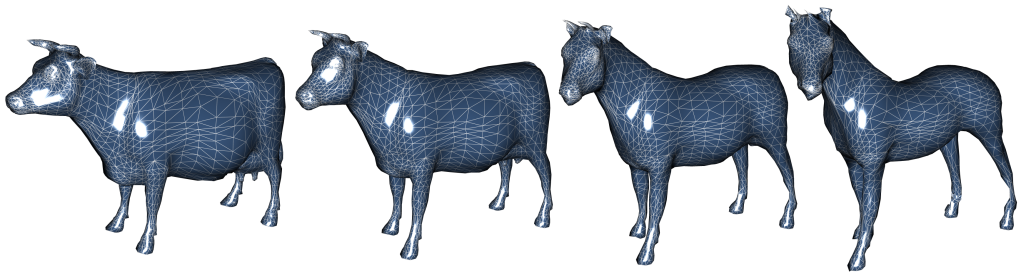


Figure 1.2: **Mesh Morphing.** One of the main applications of inter-surface mapping is to define a smooth morphing between two different but partially isometric models. In this example, a cow model is continuously transformed into a horse model using our technique.

Although a continuous mapping between such surfaces cannot exist (by definition), our method nevertheless generates visually pleasing results. To the best of our knowledge, no other mapping technique can do this. Instead, previous methods make the explicit assumption that both models being mapped are topologically equivalent.

1.3 Method Overview

In order to compute a mapping between two meshes $\mathcal{M}_1$ and $\mathcal{M}_2$, our method requires that the user supply an initial set of correspondences between the models. Our interface makes this specification a simple process of pointing and clicking, where the user selects two vertices, one in each mesh, that must be matched in the mapping. In order to avoid over-taxing the user, our system requires only a sparse set of correspondences. Once these are constructed, our algorithm uses an approximate Voronoi diagram-based method to enrich the correspondences with more vertex pairs. The user can optionally intervene at any time during this process to specify more constraints. After the number of correspondences reaches past a certain threshold (e.g. 10% of the number of vertices), our method builds a least-squares approximation of the geometry of $\mathcal{M}_2$ using the connectivity of $\mathcal{M}_1$ and vice-versa.

These approximations are enough to compute morphing sequences between the models (see example in Figure 1.2), but they do not suffice to specify detail transfer. In order to do this, we build two sparse linear-algebraic bases, one for each mesh. After this, we take any function (such as colors, texture coordinates or normals) defined on one mesh and compute its orthogonal least-squares projection onto the corresponding basis. In order to map the function to the other mesh we must simply solve a linear system corresponding to the appropriate change of basis. This allows our method to compute detail transfer without the need for complex data structures.

1.4 Text Structure

The rest of this document is organized as follows: in Chapter 2 we review relevant previous work. We discuss inter-surface mapping methods, as well as the related problem of surface parameterization and the study of Laplacian mesh editing. Chapter 3 presents a thorough overview of all linear-algebraic concepts we use in our method. We start with the notion of linear transformations, and discuss

basis vectors, eigenanalysis, and linear systems, among others. We also give a brief review of computational methods to store and manipulate large, sparse matrices. In Chapter 4 we discuss in detail the properties and different definitions of the Laplacian operator. Our discussion is focused on defining the Laplacian over progressively more complex domains, and culminates in a description of the possible ways to define this operator directly over triangle meshes. Chapter 5 describes our inter-surface mapping algorithm in detail. We discuss all phases of our technique, from the user-assisted correspondence generation to our approximate Voronoi diagram scheme to enrich the set of correspondences. We also present our method to compute detail transfer, by solving a sparse linear system corresponding to a change of basis. In Chapter 6 we present experimental results and conduct a brief analysis of the theoretical and experimental complexity of our method. Finally, in Chapter 7, we conclude the text with some limitations of our technique, as well as with a discussion of avenues for possible future work.

2 RELATED WORK

Geometry processing is a very active topic in the computer graphics literature, and mesh processing in general has received much attention throughout the years. A full bibliographical revision is beyond the scope of this text, so we restrict ourselves to some illustrative examples of the most important techniques related to our inter-surface mapping framework. We divide our discussion into three main areas: mesh parameterization, inter-surface mapping and finally Laplacian mesh editing. We provide a number of references to related work, and point the reader to comprehensive surveys relevant to each area.

2.1 Mesh Parameterization

Mesh parameterization is concerned with finding a correspondence $\Phi : \mathcal{M} \rightarrow \mathcal{D}$ between a complex triangular mesh $\mathcal{M}$ and a simpler base domain $\mathcal{D}$ (also called the *parameter space*). The base domain, which can be a topological disk, a sphere or even a coarse version of the original mesh, must always have the exact same genus as $\mathcal{M}$.

The early work of Floater [11] presents three methods for mesh parameterization, called *uniform*, *weighted least-squares* and *shape-preserving*, respectively. All these methods use a plane as the parameter space, and differ in how vertices are mapped from $\mathcal{M}$ to $\mathcal{D}$. The uniform parameterization is based on Tutte's constructive proof [52, 53] that every planar graph has a straight line drawing (i.e., a plane embedding where every edge is represented by a straight line), and uses the so-called barycentric parameterization. On the other hand, the weighted least-squares parameterization constrains the vertices on the border of the mesh to lie on the border of a convex polygon in parameter space, and minimizes the Euclidean distance between all vertices simultaneously using a least-squares formulation. Finally, the shape-preserving method also constrains the border vertices, but instead of minimizing the Euclidean distance between vertices, it minimizes the Laplacian of the parameterized mesh in a least-squares sense. In subsequent work [12], Floater also establishes formal conditions under which piecewise linear maps over triangle meshes are injective.

In Matchmaker [22], Kraevoy et al. present a method to compute a planar parameterization of a mesh with user-defined constraints. Their main application is to generate texture maps for complex geometry, where the added constraints allow the user to define point correspondences between the original mesh and the texture map. Thus, the user can specify correspondences for a model's nose, eyes and mouth, for instance. One important advantage of the Matchmaker algorithm is that it enforces the user-defined constraints exactly, while preserving as much as possible

the geometry of the original triangle mesh. In order to ensure that the constraints can always be satisfied exactly, the Matchmaker algorithm may introduce Steiner vertices into the parameterization.

All approaches mentioned so far assume that $\mathcal{M}$ is topologically equivalent to a plane, so that Φ is continuous everywhere. However, this is not the case for most triangle meshes, which have more complex topology. In order to remedy this, many authors decompose the input triangle mesh into quasi-planar *patches*, and then parameterize these patches separately. The work of Lévy et al. [28] presents a robust method to generate quasi-conformal maps to aid the generation of automatic texture atlases (quasi-conformal maps minimize angle distortion in the parameterization). Lévy et al.'s work is based on minimizing, in a least-squares sense, an objective function related to a discrete formulation of the Cauchy-Riemann equations. Their result provides a very well behaved quasi-conformal map, which is ideal for texture atlases and direct texture painting.

Finally, Floater [14] presents a method to parameterize a disconnected point set, and uses this technique to propose a simple triangulation method. His work assumes that some points in the point-set can be placed on the boundary of a convex region in the plane. Then, the algorithm minimizes the distance distortion between all points in a least-squares sense. Finally, it uses a Delaunay triangulation of the points in parameter space to define a triangulation of the original point set.

All methods described above assume that $\mathcal{D}$ is a planar domain, and either map $\mathcal{M}$ directly or break it into patches that are mapped in a piecewise-continuous fashion. However, it is also possible to map $\mathcal{M}$ to base domains other than the plane. Below we will review previous work that maps the input mesh to a spherical domain and to a coarse triangulation, respectively.

2.1.1 Spherical Parameterization

Haker et al. [17] present a method to build a conformal map between any simply-connected triangle mesh and the unit sphere, and use this mapping to automatically define texture coordinates for complex meshes. They explicitly assume that the input mesh is topologically equivalent to the sphere, therefore ensuring a continuous mapping. Praun and Hoppe [39] introduce a similar method that uses a subdivision scheme to avoid under-sampling problems in the parameter domain, and uses a stretch metric to minimize scale distortions. They validate their technique with a remeshing application, where the parameter space is retriangulated, and the new triangulation is projected back into the input domain. Their technique also assumes that the input mesh has genus 0. Finally, the work of Gotsman et al. [15] presents a generalization of Floater's uniform parameterization [11] to spherical domains. They also describe a robust algorithm to compute the parameterization based on spectral graph theory.

2.1.2 Coarse Mesh Parameterization

The main drawback of planar and spherical parameterization methods is that they require a strong topological equivalence between $\mathcal{M}$ and $\mathcal{D}$. In order to remedy this problem, some techniques explicitly *build* a simpler base domain that is still topologically consistent with $\mathcal{M}$.

Eck et al. [10] propose a technique to build a progressive, multi-resolution representation of complex triangle meshes. They construct the base domain by considering

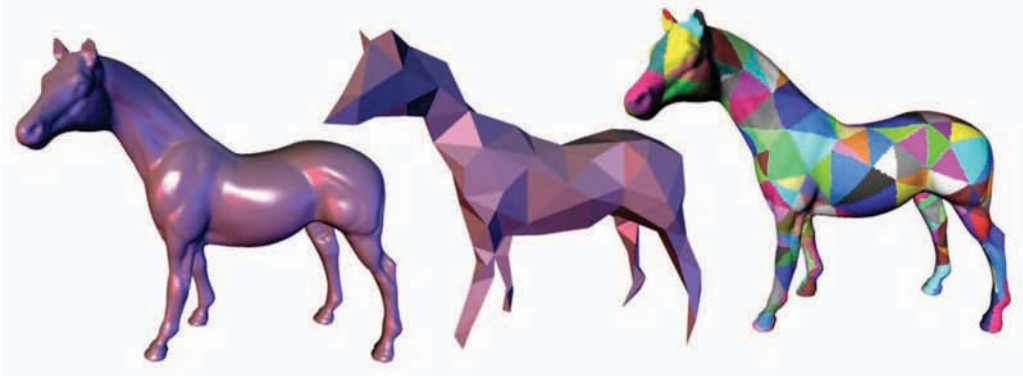


Figure 2.1: **MAPS Parameterization.** An input mesh (left) $\mathcal{M}$ is progressively simplified to a coarse base domain (center) and then parameterized (right). Observe how the MAPS algorithm does not require a planar or spherical base domain. Figure taken and adapted from [26].

a set of seed points on the input mesh, which become cell centers of a Voronoi diagram defined directly over the surface. The dual of this diagram defines a subdivision of the input mesh into a set of triangular patches, which corresponds to the coarse base domain. Each triangular patch is locally parameterized, creating a piecewise-continuous function that maps $\mathcal{M}$ directly into a topologically correct parameter space. Since the construction of the triangle patches depends only on choosing a set of points in the input mesh, this process can be applied to meshes of arbitrary genus. The authors demonstrate their technique with applications that include multi-resolution rendering, progressive mesh transmission and high-quality remeshing.

The MAPS algorithm [26] builds a similar parameter space by progressively simplifying the input mesh until it consists of a small number of triangles, which are also parameterized independently (see Figure 2.1). Their technique constructs a hierarchical representation of the mesh which allows efficient and high-quality multi-resolution editing. Finally, Guskov et al. [16] present Normal Meshes, a surface representation capable of describing the geometry of a mesh using only a single floating-point value per vertex. In order to do this, they also build a hierarchical representation of the mesh. Once they have a coarse domain, they simply store, for each vertex in the input mesh, an offset in the normal direction of its corresponding base triangle. For a more thorough examination of literature concerning mesh parameterization, we recommend Floater's very comprehensive survey [13].

Mesh parameterizations define a continuous, well behaved mapping between an input mesh and a simpler parameter space. However, they cannot directly build a correspondence between *two* complex meshes. To do this, we must turn our attention to inter-surface mapping.

2.2 Inter-surface Mapping

Inter-surface mapping considers the problem of finding a continuous map $\Phi : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ between two meshes $\mathcal{M}_1$ and $\mathcal{M}_2$. Such a mapping allows many interesting applications, such as shape morphing and detail transfer.

Lazarus and Verroust present an early survey concerning shape morphing only [24], where they divide the problem into two main areas: volume-based approaches and

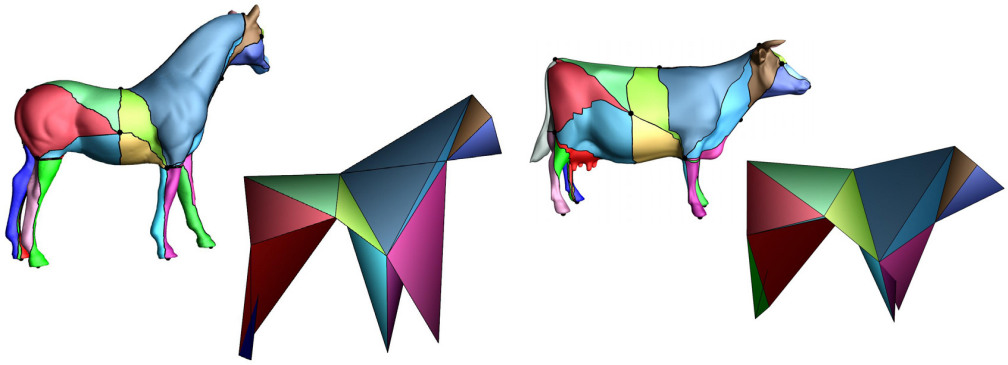


Figure 2.2: **Inter-Surface Mapping.** The work of Schreiner et al. [44] describes the first method capable of computing a high-quality mapping between two triangle meshes without an intermediate domain. Input meshes are partitioned into topologically equivalent sets of triangular patches and simplified to rough base domains, which are optimally mapped in an explicit manner. Figure taken and adapted from [44].

boundary-based approaches. When using triangle meshes we are interested only on boundary-based methods. Most techniques described in [24] either rely on heavy user interaction or map both input meshes into an intermediate common domain.

Lee et al. [25] use the MAPS algorithm to parameterize both input meshes into coarse domains, and employ a further harmonic map to find a correspondence between the parameter spaces. They then apply the correspondence directly to the shape morphing problem. A fundamental limitation of this technique is that it requires both meshes to be topologically equivalent, to allow MAPS to build two compatible parameter spaces.

The work of Praun et al. [40] presents a technique to build a single base domain for both meshes, given a set of user-defined correspondence pairs. Their algorithm decomposes the two input meshes into two sets of equivalent triangular patches. After a hierarchical simplification, both meshes have the exact same connectivity, which makes their mapping trivial. One interesting application of their method is to compute so-called *eigenmeshes*, which are analogous to eigenfaces in image processing [51]. The work of Kraevoy and Alla [21] also produces a cross-parameterization with user-defined feature correspondences.

The work of Schreiner et al. [44] presents the first algorithm capable of generating a high-quality inter-surface correspondence without need of a common intermediate domain. The algorithm works by progressively simplifying both meshes and explicitly computing an optimal mapping between the two coarse representations (Figure 2.2). It then reintroduces the vertices removed during simplification, and optimizes their mapping to minimize shape distortion. This method generates very high quality surface mappings and works for meshes of arbitrary genus (as long as both models have equal topology). However, the implementation of their algorithm is quite complex and its computational cost is very high (the paper reports execution times of "a couple of hours"[sic]).

Finally, the work of Zhang et al. [56] presents a fundamentally different approach to surface matching. Instead of building a data structure which explicitly encodes a mapping between $\mathcal{M}_1$ and $\mathcal{M}_2$, they employ a set of user-defined correspondences to approximate the shape of $\mathcal{M}_2$ using the connectivity of $\mathcal{M}_1$. Their method uses

a constrained least-squares solution to a formulation of Laplace's equation for the vertex coordinates. This approach has the distinctive advantage of simplicity, since it only needs to compute the solution to a sparse symmetric linear system. Moreover, since geometric constraints are satisfied only in a least-squares sense, it is possible to define a correspondence between surfaces of different genus. However, since Zhang et al. do not compute an explicit correspondence between the meshes, they cannot perform detail transfer unless both meshes are mapped to a common domain. Our inter-surface mapping algorithm is similar to Zhang et al.'s work, but we use a different approach to build the set of point correspondences, and we also define an explicit mapping between the meshes (via a linear-algebraic change of basis).

Both Zhang's approach and ours make heavy use of the Laplacian operator over a triangle mesh. This operator has received much attention in recent computer graphics literature, for a variety of applications. Below we give a short overview of relevant work that uses the Laplacian operator to analyze, process and edit triangulations (we refer the reader to a survey by Alexa [4] for more references concerning surface mapping and morphing).

2.3 Laplacian Mesh Editing

Laplacian mesh editing is a very broad area which refers to algorithms and tools to process triangle meshes using the Laplacian operator. For a thorough description of the mathematics involved, see Chapter 4.

The work of Taubin [50] in surface design is one of the earliest examples of methods that consider the Laplacian operator over triangle meshes. This technique introduces a simple, robust surface smoothing algorithm based on an analogy between the frequency domain on the real line and the eigenfunctions of the Laplacian operator over meshes. With this definition, it is possible to consider only the "low-frequency" contributions of the geometry of a mesh, thus removing noise in a non-shrinking fashion. Moreover, Taubin's paper introduces a very simple and elegant two-pass smoothing algorithm, now known as $(\lambda - \mu)$-smoothing.

The Laplacian operator over a triangle mesh is a sparse matrix whose non-zero entries vary depending on the particular formulation chosen by each author. Different weighting schemes define operators with varying characteristics, such as symmetry, positive-definiteness and locality, among others. Meyer et al. [34] introduce a unified framework for the Laplacian operator over triangle meshes, and discuss the characteristics of different particular formulations. Wardetzky et al. [55] provide a proof that it is not possible to define a single Laplacian operator that satisfies all desired properties simultaneously. This explains why there are so many formulations of the operator in the literature, each fitted to a particular application (refer to Chapter 4 for more details).

Sorkine and Cohen-Or use a least-squares solution to the Laplacian equation to define Least-Squares meshes [48]. These meshes are smooth triangulations that approximate a set of given geometric constraints in a least-squares sense (see Figure 2.3 for an example). They use this approach to define a set of intuitive mesh editing tools, as well as to approximate a given surface and to perform smooth mesh completion. In follow-up work [49], Sorkine et al. introduce a geometry-aware method that chooses the set of geometric constraints in an optimal way.

Lévy considers many applications of the Laplacian operator in [32], where he

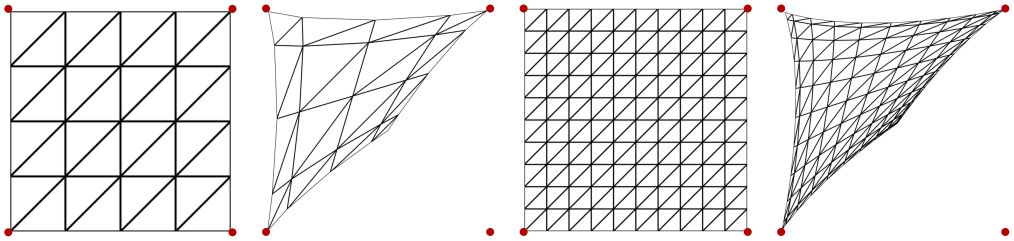


Figure 2.3: **Least-Squares Meshes.** Sorkine and Cohen-Or introduce Least-Squares meshes [48], which are used to approximate given geometric constraints with a fair connectivity. The input connectivity respects the specified constraints (shown here as red dots) in a least-squares sense, while minimizing the Laplacian of the vertex coordinates everywhere else.

demonstrates that geometric symmetries in meshes are naturally captured by the eigenvectors of the Laplacian operator. In follow-up work, Vallet and Lévy introduce the Manifold Harmonics [54], an explicit generalization of the Fourier domain to triangulated manifolds. They describe an efficient way to compute the eigenvectors of the Laplacian (which correspond to sine and cosine waves on the real line), and perform direct signal processing by orthogonally projecting functions into these eigenvectors. They also propose a symmetric formulation of the Laplacian operator based on Discrete Exterior Calculus (DEC). Although their formulation cannot satisfy all properties of the continuous Laplace-Beltrami operator [55], it nevertheless yields very high quality results in practice.

The work of Nealen et al. [37] describes a complete framework to optimize triangle shape and smoothness using the Laplacian operator. They describe some of the existing weighting schemes and provide a comprehensive set of examples of their mesh editing technique. Finally, Sorkine and Alexa describe a complete mesh editing tool that makes extensive use of the Laplacian [47]. This tool allows users to perform intuitive shape deformation using point constraints and line sketches.

We refer the reader to the State-of-the-art report by Sorkine [46], which presents a comprehensive guide to advances in mesh editing tools and techniques that use the Laplacian operator, thus completing our review of previous work. In the next Chapter, we will review the most important mathematical concepts of linear algebra, which will provide us with the tools to define the Laplacian operator over triangle meshes and to solve the subsequent linear systems involved in our algorithm.

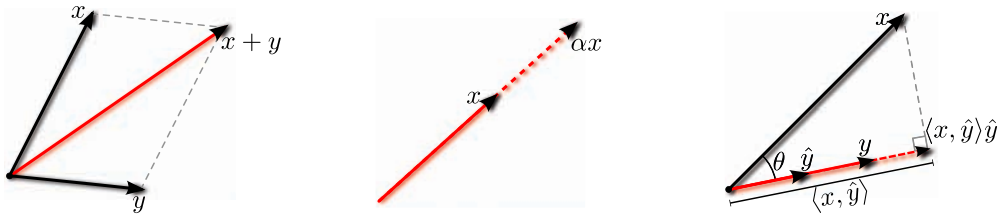
3 LINEAR SYSTEMS AND TRANSFORMATIONS

Linear systems of equations are a very powerful mathematical tool that appear in a diversity of applications. They are the direct generalization of equations of the form $ax = b$, for constant a and b and for $x \in \mathbb{R}$. However, instead of dealing with a single x, the study of linear systems is concerned with equations that may involve many variables simultaneously.

In this Chapter, we will review and study the most widely used algorithms to solve linear systems of equations of the form $Ax = b$, where A is a matrix and x and b are vectors. We will review basic linear-algebraic concepts such as the rank of a matrix, its null space and its eigenvectors and eigenvalues, among others. Much of our discussion is based on Howard Anton's *Elementary Linear Algebra* [5]. We will also discuss different approaches for the solution of linear systems, including direct methods such as Gaussian elimination (via the Doolittle algorithm [19]), as well as iterative methods such as Jacobi [43] and Gauss-Seidel [43] iteration. Finally, we will discuss the Cholesky decomposition [23], a direct method which can be applied to symmetric positive-definite matrices, and which is roughly twice as efficient as the LU decomposition [5].

Linear equations are typically written and solved in terms of vectors embedded into some subspace of $\mathbb{R}^n$ for arbitrary n. These vectors are equipped with three fundamental operations: addition, multiplication by a scalar and an inner product. Addition takes two n-dimensional vectors $x = [x_0, x_1, \ldots, x_{n-1}]^T$ and $y = [y_0, y_1, \ldots, y_{n-1}]^T$ and returns a vector $z = [x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}]^T$ (see Subsection 3.1.1 for an explanation of the superscript T). Vector addition can be understood in terms of the *parallelogram rule*, which is illustrated in Figure 3.1(a). Scalar multiplication, on the other hand, takes an n-dimensional vector x and a scalar α and yields the vector $y = [\alpha x_0, \alpha x_1, \ldots, \alpha x_{n-1}]^T$. This operation scales the input vector by a factor of α, and can be seen in Figure 3.1(b). Finally, the inner product between two vectors x and y returns a *scalar*, usually denoted by $\langle x, y \rangle$, such that $\langle x, y \rangle = x_0 y_0 + x_1 y_1 + \ldots + x_{n-1} y_{n-1}$. The inner product indicates the length of a vector in Euclidean space, since $\|x\| = \sqrt{\langle x, x \rangle}$. Moreover, it can also be used to compute the orthogonal projection of a vector x onto a vector y, by considering $\hat{y} = \frac{y}{\|y\|}$, which points in the direction of y but is of unit length. The value of $\langle x, \hat{y} \rangle$ is the length of x's orthogonal projection onto y, and the projected vector is simply $\langle x, \hat{y} \rangle \hat{y}$. This construction can be examined in Figure 3.1(c). Finally, the inner product also defines a trigonometric relation between two vectors:

$$\langle x, y \rangle = \|x\| \|y\| \cos(\theta),$$



(a) (b) (c)

Figure 3.1: **Fundamental Vector Operations.** The three fundamental operations on vectors: addition, scalar multiplication and inner product (also known as *dot product*). Notice how $x + y$ equals the diagonal of a parallelogram whose sides are x and y (this is known as the *parallelogram rule*) (a). Scalar multiplication only alters a vector's magnitude (b). Finally, the inner product can be used to project a vector x onto another vector y (c).

where θ is the angle spanned by x and y (see Figure 3.1(c)). This property gives us a very convenient way to determine when two vectors are perpendicular (or *orthogonal*) to one another: x and y are perpendicular if and only if $\langle x, y \rangle = 0$ (because $\cos(\pi/2) = 0$).

We can use these three operations to explore other fundamental concepts about vectors and linear spaces. The first construction we need is a *linear combination* of vectors. A linear combination of a sequence of m n-dimensional vectors $x^0, x^1, \ldots, x^{m\ 1}$ is a new vector z formed by multiplying each vector x^i by a scalar y_i and adding them all together. Each y_i defines the weighted contribution of the corresponding vector x_i to the linear combination. Notice how we use subscript notation to refer to the scalars y_i; we do this on purpose, because now we can conveniently gather all y_i into a single m-dimensional vector y. By changing the values of y_i, we obtain different linear combinations of x^i. This suggests that the vectors x^i define an operation which can be applied to arbitrary m-dimensional vectors y. We call this operation a *linear transformation* of y onto z. Notice that we will, throughout the rest of the text, use subscript notation to refer to individual components in a vector, and superscript notation to refer to an individual vector in a sequence. Thus, for a column-major matrix L, element l_i^j belongs to the i^{th} row and j^{th} column.

3.1 Linear Transformations

If we choose a particular sequence of m n-dimensional vectors x^i, we can arrange their elements in a table consisting of n rows and m columns. This arrangement is convenient because by placing any m-dimensional vector y next to this table, we can compute the linear combination z using the standard rules for matrix multiplication (in fact, these rules are *defined* specifically to make this operation possible):

$$\begin{pmatrix} x_0^0 & x_0^1 & & x_0^{m\ 1} \\ x_1^0 & x_1^1 & & x_1^{m\ 1} \\ & \ddots & \ddots & \\ x_{n\ 1}^0 & x_{n\ 1}^1 & & x_{n\ 1}^{m\ 1} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{m\ 1} \end{pmatrix} = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n\ 1} \end{pmatrix} \tag{3.1}$$

A given particular sequence of x^i can be applied to any $y \in \mathbb{R}^m$, thus defining a *Linear Transformation* (or *Linear Map*) from $\mathbb{R}^m$ to $\mathbb{R}^n$. As is obvious from

$$\begin{pmatrix} \vdots & \vdots & \vdots \\ a^0 & a^1 & a^{n-1} \\ \vdots & \vdots & \vdots \end{pmatrix}_{m\times n} \begin{pmatrix} b_0^0 & b_0^1 & b_0^{p-1} \\ b_1^0 & b_1^1 & b_1^{p-1} \\ \vdots & \vdots & \vdots \\ b_{n-1}^0 & b_{n-1}^1 & b_{n-1}^{p-1} \end{pmatrix}_{n\times p} = \begin{pmatrix} \vdots & \vdots & \vdots \\ c^0 & c^1 & c^{p-1} \\ \vdots & \vdots & \vdots \end{pmatrix}_{m\times p}$$

$$c_0 = b_0^0 a^0 + b_1^0 a^1 + \cdots + b_{n-1}^0 a^{n-1}$$

Figure 3.2: **Matrix Multiplication.** The product of two matrices $AB = C$ can be understood as a sequence of linear combinations. Each column c^i of C is formed using the entries of column b^i as weights to a linear combination of all columns of A. We have annotated the matrices in this example with their dimensionalities (in red) to clarify the size requirements for two matrices to be compatible for multiplication.

Equation (3.1), any linear transformation can be represented in matrix form, simply by placing the vectors x^i side-by-side, and its application to a given vector is a matrix-vector multiplication [5].

With this interpretation of linear transformations we can provide a very simple explanation to the standard matrix-matrix multiplication algorithm: when computing the matrix product $AB = C$, each column c^i of the result is the linear combination of all columns of A using b^i as the weighting vector. This also explains the dimensionality requirements for the matrices A and B: if A has n columns, we need exactly n scalars to build their linear combinations, so each column of B must have exactly n elements (meaning B must have exactly n *rows*). C will then have as many columns as B (since each c^i is a linear *transformation* of the corresponding b^i) and as many rows as A, since c^i are linear *combinations* of the columns of A, and must therefore have equal dimension.

3.1.1 Matrix Transposition

Observe that we have chosen to represent vectors as $(n \times 1)$-dimensional matrices to follow convention only. With this representation, linear combinations are realized by right-multiplying a matrix with a column-vector. However, had we chosen to represent vectors as $(1 \times n)$-dimensional matrices, linear combinations would be computed by *left*-multiplying a matrix with a row-vector. Moreover, this operation would represent linear combinations of the *rows* of the matrix, instead of its columns. These two conventions are equivalent, and they lead us to define an operation to convert between them, namely *matrix transposition*.

The *transpose* of an $(m \times n)$-dimensional matrix A, written A^T, is an $(n \times m)$-dimensional matrix whose rows are the columns of A (and, consequently, whose columns are the rows of A). Based on our discussion above, it is easy to see the following important equivalence [5]:

$$(AB)^T = B^T A^T,$$

that is, if we wish to adopt a row-major convention, we must simply transpose all the arguments and reverse their order of multiplication (recall that a row-vector must left-multiply a matrix to generate another row-vector).

One of the interesting properties of matrix transposition is that it allows us to

define inner products directly in terms of a matrix multiplication:

$$\langle x, y \rangle = x^T y$$

Notice that, because both x and y are $(n \times 1)$-dimensional vectors, the matrix multiplication above is possible, and yields a 1×1 matrix as a result (which is simply a scalar).

Matrix transposition allows us to define a specific class of matrices, called *symmetric* matrices, which satisfy the following identity: $A^T = A$. Symmetric matrices have the property that their eigenvectors are all orthogonal (we will study eigenvectors in more detail in Section 3.5).

3.1.2 Fundamental Properties

Linear transformations satisfy two very important properties. Given a transformation represented by the matrix A, two arbitrary vectors x and y, and a scalar α, we have [5]:

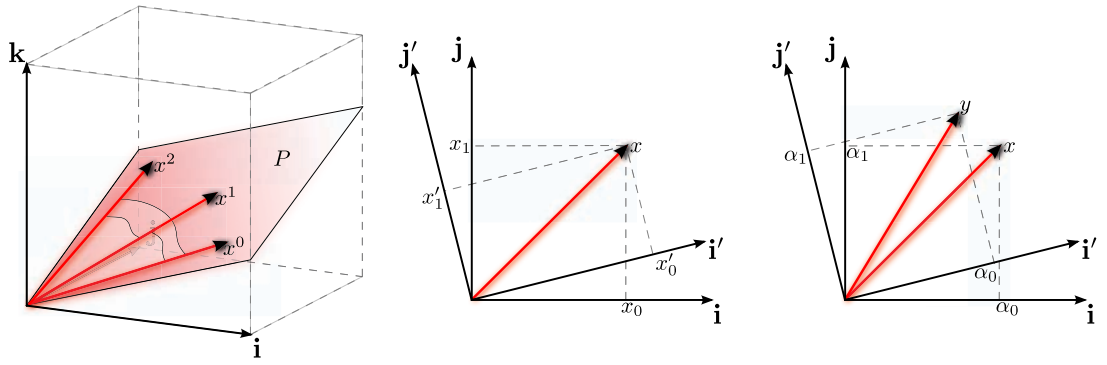
$$\begin{aligned} A(x + y) &= Ax + Ay \quad (\textit{additivity}) \\ A(\alpha x) &= \alpha Ax \quad (\textit{homogeneity}) \end{aligned}$$

In fact, *any* function $f : \mathbb{R}^m \to \mathbb{R}^n$ that satisfies these properties is a linear map and therefore has a corresponding matrix representation.

Given a linear map $A : \mathbb{R}^m \to \mathbb{R}^n$, we can study the properties of vectors under A. In particular, we want to analyze A to determine if it is capable of "reaching" $\mathbb{R}^n$ as a whole, or only some linear subspace of it. In order to examine these properties, we need the concept of *linear independence*, which we introduce below.

3.2 Linear Independence and Basis Vectors

A set of vectors is said to be *linearly independent* if and only if none of the individual vectors in the set can be written as a linear combination of the others [5]. This concept allows us to study properties of the set of all linear combinations of vectors x^i. Linear combinations of a single m-dimensional vector x^0 are always of the form αx^0, and inhabit the same one-dimensional space. With two vectors x^0 and x^1, however, the situation becomes more interesting: if there exists an α such that $\alpha x^0 = x^1$, then x^0 and x^1 are *linearly dependent*, and the vectors obtainable through their linear combinations are the same as in the single-vector case. On the other hand, if x^0 and x^1 are linearly independent, we can build new vectors of the form $y = \alpha_0 x^0 + \alpha_1 x^1$. The set of all possible y spans a plane embedded in $\mathbb{R}^m$. In general, a set of n linearly independent m-dimensional vectors defines an n-dimensional linear subspace of $\mathbb{R}^m$, and whenever we add a new linearly independent vector to this set, we increase the dimension of the subspace by one. Therefore, linear combinations of m linearly independent m-dimensional vectors span $\mathbb{R}^m$ as a whole. This allows us to draw a very important conclusion: any vector $y \in \mathbb{R}^m$ can be written as a linear combination of m linearly independent vectors x^i, also in $\mathbb{R}^m$. Because linear combinations of x^i can be used to obtain any $y \in \mathbb{R}^m$, we say that the sequence x^i *spans* $\mathbb{R}^m$, and we call x^i a *basis* [5] (plural: *bases*) for $\mathbb{R}^m$. In fact, any set of n linearly independent vectors forms a basis for some n-dimensional linear subspace of $\mathbb{R}^m$.



(a) (b) (c)

Figure 3.3: **Basis Vectors.** m-dimensional linear spaces can be characterized by a set of m linearly independent vectors, called the *basis* for the space. A basis need not be unique, since any set of linearly independent vectors in a space P can define P (a). When vectors are expressed in terms of their components, the basis must always be specified, as the same vector x can have different components depending on the choice of basis (b), and two different vectors x and y can have the same components when expressed in different bases (c).

Observe that the mapping between a set of vectors and a linear subspace is not unique: any set of m linearly independent vectors $x^0, x^i, \ldots, x^{m\ 1}$ in the same m-dimensional space P can be used to define P (see Figure 3.3(a)). Bases in general are not unique to a space, but they can nevertheless be characterized in the following ways: a basis whose vectors are all orthogonal to one another is called an *orthogonal basis*, and if these vectors are also unit-length, we call this basis an *orthonormal basis*. In particular, the vectors $x^0 = [1, 0, \ldots, 0]^T, x^1 = [0, 1, \ldots, 0]^T, \ldots, x^{m\ 1} = [0, 0, \ldots, 1]^T$ form an orthonormal basis for $\mathbb{R}^m$, also called the *canonical basis* [5].

When a vector is expressed in terms of its components, it typically refers to the canonical basis. However, the numerical components of vectors in $\mathbb{R}^m$ have no intrinsic meaning — they can only be assigned significance when we specify the basis vectors with which the vector is expressed. This happens because the components of a vector are merely weights for a linear combination of the basis vectors which, if changed, produce arbitrarily different results. Figure 3.3 illustrates these concepts with two examples, one where the same vector is expressed as two different sets of components $[x_0, x_1]^T$ and $[x'_0, x'_1]^T$ (3.3(b)), and the other where the same set of components defines two different vectors x and y (3.3(c)), depending solely on the choice of basis. In both these examples, we use the canonical basis for $\mathbb{R}^2$ ($\{\mathbf{i}, \mathbf{j}\}$), as well as a rotated basis $\{\mathbf{i}', \mathbf{j}'\}$.

The concepts of linear independence and basis vectors provide us the tools to analyze the dimensionality of the image of a linear transformation $A : \mathbb{R}^m \rightarrow D$, for $D \subseteq \mathbb{R}^n$. Even though vectors $y \in D$ have n components, D need not be equal to $\mathbb{R}^n$. As we will see in the next Section, this is the case only if A contains at least n linearly independent columns.

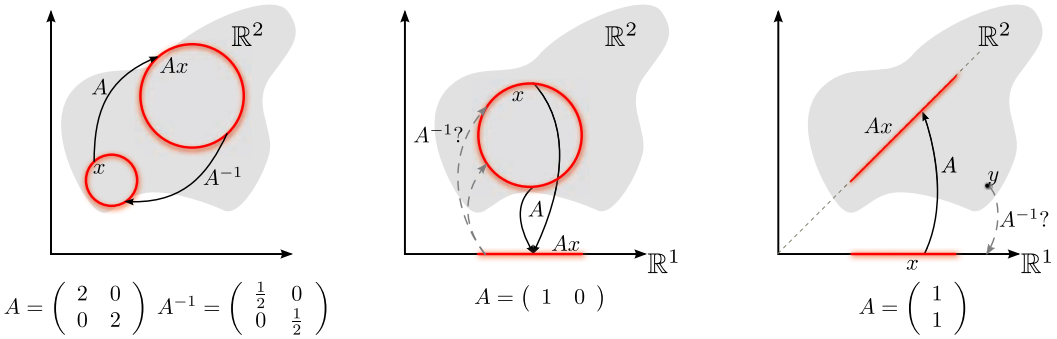


Figure 3.4: **Linear Transformations in $\mathbb{R}^2$ and their Inverses.** A linear transformation A has a well defined inverse if and only if it is a one-to-one mapping where the domain and the image have the same dimensionality. In (a), a full-rank matrix maps vectors from $\mathbb{R}^2$ to $\mathbb{R}^2$, and therefore has a unique inverse. If the mapping takes a higher dimensional space into a smaller dimensional one, its inverse is not functional (b). The same is true for the inverse of a mapping that takes a smaller dimensional space into a larger one (c).

3.3 Rank of a Matrix, Row and Column Spaces

Recall that a sequence of n linearly independent m-dimensional vectors a^i forms a basis for an n-dimensional linear subspace $D \subseteq \mathbb{R}^m$. If these vectors are the columns of a matrix A, then D is exactly the image of the transformation defined by A. We call the dimensionality of this space (or, equivalently, the number of linearly independent columns of A) the *rank* of A. Because A's image is formed by linear combinations of its columns, we call it A's *column space*. Furthermore, we can also consider the space spanned by the *rows* of A, known as its *row space*. It is a well known theorem that the dimensionality of the row and column spaces is always equal to the rank of the matrix, regardless of its dimensions [5]. If a matrix has dimensions $m \times n$, it can have *at most* m linearly independent columns (and at most n linearly independent rows). Because of this, the rank of an $m \times n$ matrix is at most $min(m, n)$. Matrices whose rank is exactly $min(m, n)$ are called *full-rank* matrices; otherwise, they are *rank-deficient*. Since the rank of a matrix is equivalent to the dimensionality of its image, we conclude that in a linear transformation $A : \mathbb{R}^m \rightarrow D$, for $D \subseteq \mathbb{R}^n$, D is equal to $\mathbb{R}^n$ only if $rank(A) = n$.

Given a linear transformation $A : \mathbb{R}^m \rightarrow \mathbb{R}^n$, we can ask whether there exists another transformation $B : \mathbb{R}^n \rightarrow \mathbb{R}^m$, such that $BAx = x$ and $ABx = x$ (we will see very soon that this implies $n = m$). An equivalent requirement is that $BA = AB = I$, where I is the identity transformation for $\mathbb{R}^m$. When such a transformation exists, we call it the *inverse* of A, and denote it by A^{-1}. Notice that $(A^{-1})^{-1} = A$. In the next Section we will investigate conditions under which the inverse can exist, relating these conditions to the dimensionality of a matrix, as well as its rank.

3.4 The Existence of an Inverse Linear Transformation

The first property required for a linear transformation to have an inverse is that *the dimensionality of its domain and range must be the same*, and therefore its

matrix representation must be square [5]. There is a very strong intuition behind this requirement: if the domain has higher dimensionality than the image, there are infinitely many different vectors x_i that are transformed to the same value Ax. This happens because the transformation A must "compress" all vectors of a higher-dimensional space into a smaller-dimensional subspace, and this subspace does not contain enough information to uniquely recover all input vectors. Therefore, the inverse relation, i.e., the relation taking Ax back to each x_i is not even functional, let alone linear (see Figure 3.4(b)). On the other hand, if the domain has *smaller* dimensionality than the image (e.g., a matrix with more rows than columns), we can apply the exact same argument to the inverse transformation: if A were to map a smaller-dimensional space into a larger one, than A^{-1} would have to map a larger-dimensional space back to a smaller one, and would not have an inverse. But since $(A^{-1})^{-1}$ must exist (and must be equal to A), we conclude by contradiction that A^{-1} does not exist (see Figure 3.4(c)). Notice that this does not mean that there is no transformation B such that $BA = I$. B does indeed exist, and it is called the *Moore-Penrose Pseudo-Inverse* of A [6]. B cannot be A's inverse, however, since $AB \neq I$.

A linear transformation whose image has the same dimensionality as its domain is always represented by a square matrix, but the opposite is not always true. This means that an invertible matrix is always square, but not every square matrix is invertible. Consider the following very simple counter-example:

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$

This matrix is square, but the dimensionality of its image is 1, while its domain is $\mathbb{R}^3$. In the above example it is clearly noticeable that non-invertibility happens because the columns of the matrix are not linearly independent. This allows us to formulate a stronger requirement for invertible matrices: an $n \times n$ square matrix is invertible if and only if its columns span $\mathbb{R}^n$. In this case each $x \in \mathbb{R}^3$ defines a unique linear combination $y = Ax$. When this is the case, we can define A^{-1} as the linear transformation that maps each y back to its corresponding x, such that $A^{-1}Ax = AA^{-1}x = x$.

The action Ax of a linear transformation A on a vector x can seem complex and difficult to analyze precisely. However, we can use the additivity property to decompose x into a linear combination of a specific set of vectors whose behavior under A is simple. These vectors are the *eigenvectors* of A, which we will study in more detail below.

3.5 Eigenanalysis

Any linear transformation A has a set of specific vectors x^i that satisfy the following property: $A\mathsf{x}^i = \lambda_i \mathsf{x}^i$ for some scalar λ_i. These vectors are called the *eigenvectors* of A, and each λ_i is called a corresponding *eigenvalue*. The existence and multiplicity of different eigenvectors are closely related to the rank of the linear transformation. Naturally, if x^i is an eigenvector, then so is $\alpha\mathsf{x}^i$ for any scalar α, since $A\alpha\mathsf{x}^i = \lambda_i \alpha \mathsf{x}^i$. Therefore, it makes more sense to think of eigenvectors as particular *directions* rather than individual vectors.

Eigenvectors provide a simple way to understand the action of a linear transformation A, as they formally specify the directions in which A acts simply as a scaling factor. By decomposing an arbitrary vector x into a linear combination of A's eigenvectors, we can derive a strong intuition regarding the behavior of Ax. Any linear transformation $A : \mathbb{R}^m \to \mathbb{R}^n$ always has a set of m linearly independent eigenvectors (although some of its eigenvalues may be 0, as we will see in detail below). Because of this, the set x^i of eigenvectors constitutes a proper basis for $\mathbb{R}^m$, so we can write any m-dimensional vector x as a linear combination of the eigenvectors x^i with weights α_i (we assume for convenience that the eigenvectors are sorted by the magnitude of their eigenvalues, so that x^0 has the smallest eigenvalue, and so on). Thus, if $x = \alpha_0\mathsf{x}^0 + \alpha_1\mathsf{x}^1 + \ldots + \alpha_{m-1}\mathsf{x}^{m-1}$, we can use the additivity property to rewrite Ax as:

$$\begin{aligned} Ax &= A(\alpha_0\mathsf{x}^0 + \alpha_1\mathsf{x}^1 + \ldots + \alpha_{m-1}\mathsf{x}^{m-1}) && \text{(3.3a)} \\ &= A(\alpha_0\mathsf{x}^0) + A(\alpha_1\mathsf{x}^1) + \ldots + A(\alpha_{m-1}\mathsf{x}^{m-1}) && \text{(3.3b)} \\ &= \lambda_0(\alpha_0\mathsf{x}^0) + \lambda_1(\alpha_1\mathsf{x}^1) + \ldots + \lambda_{m-1}(\alpha_{m-1}\mathsf{x}^{m-1}) && \text{(3.3c)} \end{aligned}$$

Equation (3.3c) allows us to make some important observations: first, notice how the behavior of A is completely determined by its eigenvectors and corresponding eigenvalues. In fact, this analysis lets us conclude that the action of a linear transformation is simply a non-uniform scale on the basis defined by its eigenvectors. Therefore, if the eigenvectors are orthogonal and we write A in their basis, A becomes a diagonal matrix whose entries are $\lambda_0, \lambda_1, \ldots, \lambda_{m-1}$. Furthermore, we can see that a square matrix A will be one-to-one (and hence full-rank) if and only if all its eigenvalues are non-zero. We now demonstrate this equivalence (we will split the equivalence into its two implications, and prove each one separately).

If A is one-to-one, then all its eigenvalues are non-zero.

Proof: Assume that the first n eigenvalues of A are 0. Now take two vectors x^0 and x^1 that, when expressed in the basis formed by A's eigenvectors, differ by at most the first n components. Then A cannot be one-to-one, since $Ax^0 = Ax^1$. In essence, A is incapable of "discriminating" between vectors that only differ where A's eigenvalues are zero, since it collapses this entire set of vectors to a single point. Furthermore, the dimension of the subspace where these vectors are located is exactly n, since we have exactly n "degrees of freedom" from which to choose x^i vectors that are different but get mapped to the same point by A.

If all eigenvalues are non-zero, then A is full-rank.

Proof: Any two vectors $x^0 \neq x^1$ must, by definition, differ by at least one component when written in the basis of A's eigenvectors. We can assume without loss of generality that they differ in the component for x^0. Let us call α_0 that component in x^0 and α_1 that component in x^1. Then, we can write x^0 and x^1 thus:

$$\begin{aligned} x^0 &= \alpha_0\mathsf{x}^0 + x^{0^*} \\ x^1 &= \alpha_1\mathsf{x}^0 + x^{1^*}, \end{aligned}$$

where x^{0^*} and x^{1^*} are portions of x^0 and x^1 that we need not characterize precisely.

Now write out Ax^0 and Ax^1:

$$\begin{aligned} Ax^0 &= A(\alpha_0 \mathsf{x}^0 + x^{0^*}) \\ &= \alpha_0 A\mathsf{x}^0 + Ax^{0^*} \\ &= \alpha_0 \lambda_0 \mathsf{x}^0 + Ax^{0^*} \\ Ax^1 &= A(\alpha_1 \mathsf{x}^0 + x^{1^*}) \\ &= \alpha_1 A\mathsf{x}^0 + Ax^{1^*} \\ &= \alpha_1 \lambda_0 \mathsf{x}^0 + Ax^{1^*}. \end{aligned}$$

Since $\alpha_0 \neq \alpha_1$, it follows from the equations above that $Ax^0 \neq Ax^1$, for all $x^0 \neq x^1$. Because we are assuming that A is square (and hence its image has at most the same dimensionality as its domain), and we have just proved that any two different vectors in the domain are mapped to two different vectors in the image, this suffices to prove that A is full-rank, completing our demonstration of the equivalence.

Up to this point we have focused all our attention on the study of situations where we apply a linear transformation A to a vector x to obtain a new vector y. However, the concept of an inverse transformation allows us to ask the following question: given a linear transformation A and a vector b, can we find a new vector x such that $Ax = b$? To find x, we must characterize and solve a set of linear equations, where each equation can be written as $\langle a^i, x \rangle = b_i$ (a^i is the i_{th} column of A and b_i is the i_{th} component of b).

3.6 Characterization of a Linear System of Equations

A linear system can be fully characterized by analyzing the properties of its matrix A. As we observed previously, a matrix is either full-rank or rank-deficient. If A is full-rank, then any vector b in the image of A can be expressed as a unique linear combination of A's columns. This means that $Ax = b$ must have exactly one solution, where x is precisely the vector of weights that forms b. Under such conditions, we say that the system $Ax = b$ is *exactly determined* [5].

On the other hand, if A is rank-deficient, one of two things must happen: $Ax = b$ must either have no solutions or have infinitely many. If the rank of A is less than its number of rows, A is taking a small-dimensional domain into a large-dimensional image. This happens for any $A : \mathbb{R}^m \to \mathbb{R}^n$ where $n > m$. In this case, there is no linear combination of A's columns capable of reaching all vectors in $\mathbb{R}^n$, so $Ax = b$ has, in general, no solutions. In this case we say that A defines an *over-determined* [5] linear system. As we will see in Section 3.9, it is possible to find a vector x whose transformation Ax is the "best approximation" of b, in a sense that we will make more precise later. Of course, if we happen to choose one particular b that *is* in the image of A, then $Ax = b$ *does* have a unique solution, but this is not true for arbitrary $b \in \mathbb{R}^n$.

Finally, if A maps a large dimensional domain into a small dimensional image (any $A : \mathbb{R}^m \to \mathbb{R}^n$ where $n > m$), the system $Ax = y$ has infinitely many solutions, since we are "compressing" a large space into a small one, so an infinite number of vectors x is mapped to a single y. This situation characterizes an *under-determined* [5] system. These three situations are direct consequences of the three possibilities we discussed in Section 3.4 concerning the existence of A^{-1}.

The most direct way to define the solution to $Ax = b$ for an exactly determined system is by writing:

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ \therefore x &= A^{-1}b \end{aligned}$$

Thus $A^{-1}b$ is the solution to $Ax = b$, requiring only a matrix-vector multiplication, *provided we have the inverse* A^{-1}. Unfortunately, matrix inversion is an expensive computational task, and inversion of large matrices is rarely practical. Therefore, we must look for other methods to solve linear systems, especially those involving a large number of equations and variables.

Observe that up to this point we have discussed aspects of linear spaces, matrices and transformations mostly in isolation. However, many of the properties that characterize a matrix are equivalent, so we summarize them below for ease of reference.

The following statements are all equivalent, for a square $n \times n$ matrix A:

- The columns of A are linearly independent
- A is a full-rank matrix
- The column space of A spans $\mathbb{R}^n$
- The columns of A form a basis for $\mathbb{R}^n$
- There exists a matrix A^{-1} such that $A^{-1}A = AA^{-1} = I$
- The eigenvectors of A form a basis for $\mathbb{R}^n$
- All eigenvalues of A are non-zero
- The system $Ax = b$ has exactly one solution for every b

In the next Section we will discuss some of the most widely used iterative and direct methods to solve linear systems. Iterative methods start with an initial guess for the solution and repeatedly modify this guess until it converges to x. Direct methods, on the other hand, typically decompose the matrix A into a product of simpler matrices, and exploit the structure of these matrices to find the solution.

3.7 Iterative Methods for Linear Systems

Iterative methods attempt to solve a linear system $Ax = b$ by choosing an initial guess x^0, and applying some repeated process to this vector, thus forming a sequence x^i that, under some circumstances, converges to the solution vector x. The particulars of the iteration and the circumstances under which it converges vary depending on which method we use, so we will split our discussion below into some of the more popular iterative methods available. We start our discussion with one of the simplest iterative methods, known as *Jacobi iteration*, after German mathematician Carl Gustav Jacob Jacobi. We base our explanation of the Jacobi method on [45].

3.7.1 Jacobi Iteration

The Jacobi iteration method [45] works by splitting A into two parts: D, whose diagonal elements are those of A, and whose off-diagonal elements are zero; and E, whose off-diagonal elements are those of A, and whose diagonal elements are zero. Thus $A = D + E$ (see Figure 3.5). We then write the linear system in terms of these new matrices:

$$\begin{aligned} Ax &= b && \text{(3.4a)} \\ (D+E)x &= b && \text{(3.4b)} \\ Dx &= b - Ex && \text{(3.4c)} \\ x &= D^{-1}b - D^{-1}Ex && \text{(3.4d)} \\ \therefore x &= z + Bx, && \text{(3.4e)} \end{aligned}$$

where $B = -D^{-1}E$ and $z = D^{-1}b$. The advantage of this formulation is that, since D is a diagonal matrix, it is easy to compute D^{-1}: we must simply replace the diagonal entries of D by their reciprocals. To define an algorithm from Equation (3.4), we convert (3.4e) into the following recurrence:

$$x^{i+1} = z + Bx^i \tag{3.5}$$

Given an initial guess x^0, this iterative process completely defines Jacobi's method for linear systems, where $\lim_{i\to\infty} x^i = x$. We will now briefly observe the conditions which must be true of A to ensure convergence, as well as analyze the rate of convergence. Notice that, by definition, if $x^i = x$, then $x^{i+1} = x$ as well. Therefore, x is a *fixed point* of the iteration.

We can use the additivity property to split an arbitrary x^i into two terms: the exact solution x and an *error term* e^i, such that $x^i = x + e^i$. Then we can rewrite our iteration thus:

$$\begin{aligned} x^{i+1} &= z + Bx^i \\ &= z + B(x + e^i) \\ &= z + Bx + Be^i \\ \therefore x^{i+1} &= x + Be^i \end{aligned}$$

Because $x^{i+1} = x + Be^{i+1}$, we have

$$e^{i+1} = Be^i$$

The Equations above demonstrate that the iteration does not affect the correct term x; instead, it repeatedly multiplies the error terms by B. If the sequence $e^0, e^1, \ldots, e^i$ converges to zero, then the Jacobi method converges to the correct solution for the system. This is true for matrices A that are *strictly diagonally dominant*, that is, matrices where the absolute value of the diagonal element of each row is larger than the sum of the absolute values of off-diagonal elements in that row. When this happens, the largest eigenvalue of B will be strictly less than one, guaranteeing that the sequence of error terms will vanish. Furthermore, the speed of convergence is dictated by the magnitude of B's largest eigenvalue, since if e^i has a component in that direction, it will be the slowest component to vanish.

$$A = \begin{pmatrix} a_0^0 & a_0^1 & & a_0^{n-1} \\ a_0^0 & a_1^1 & & a_1^{n-1} \\ & \ddots & \ddots & \\ a_{n-1}^0 & a_{n-1}^1 & & a_{n-1}^{n-1} \end{pmatrix} D = \begin{pmatrix} a_0^0 & 0 & & 0 \\ 0 & a_1^1 & & 0 \\ & \ddots & \ddots & \\ 0 & 0 & & a_{n-1}^{n-1} \end{pmatrix} E = \begin{pmatrix} 0 & a_0^1 & & a_0^{n-1} \\ a_1^0 & 0 & & a_1^{n-1} \\ & \ddots & \ddots & \\ a_{n-1}^0 & a_{n-1}^1 & & 0 \end{pmatrix}$$

Figure 3.5: **Jacobi Iteration Matrices.** A is split into D and E, its diagonal and off-diagonal elements, respectively.

The Jacobi iteration method is the simplest iterative technique for solving linear systems, and has the distinct advantage of being trivially paralellizable, as each row x_i is entirely independent of all the others. Its general definition, as given in Equation (3.5), involves a complete matrix-vector multiplication for each iteration. However, upon closer inspection, we can derive an equivalent formulation where we compute the value of each component in x^{i+1} explicitly. First, observe that the product $D^{-1}b$ yields a vector which is simply b scaled by the reciprocal of A's main diagonal. Moreover, the product $-D^{-1}E$ is merely the off-diagonal elements of A, also scaled by the reciprocal of A's main diagonal. Therefore, each component x_k^{i+1} in Equation (3.5) is b_k minus the inner product of x^i and A's k_{th} row, scaled by D^{-1}. We can write this operation in terms of each component of x^{i+1}:

$$x_k^{i+1} = \frac{1}{a_k^k}\left(b_k - \sum_{j \neq k} a_k^j x_j^i\right), \qquad k = 0, 1, \ldots, n-1 \tag{3.6}$$

Notice how, in order to compute x^{i+1}, we need the elements in x^i. Computationally, this means that we must keep the two most recent elements in the solution sequence in memory at all times; on the other hand, this formulation allows us to solve the equations in parallel for all k. In the next Subsection we will study Gauss-Seidel iteration, which works in a similar fashion, but computes the elements in x^{i+1} sequentially.

3.7.2 Gauss-Seidel Iteration

Again we split A into two parts: L, which is A's lower triangle, and U, which is A's upper triangle, such that $A = L + U$ (Figure 3.6) [43]. Notice how the diagonal of A is strictly in L. We can then rewrite our original system thus:

$$Ax = b \tag{3.7a}$$
$$(L + U)x = b \tag{3.7b}$$
$$Lx = b - Ux \tag{3.7c}$$
$$\therefore x = L^{-1}(b - Ux) \tag{3.7d}$$

Unlike in the Jacobi iteration, it is not generally possible to find a simple analytical expansion to L^{-1}. However, we can take advantage of L's triangular structure to find a straightforward solution to the linear system in Equation (3.7c). The first equation in that system involves only one variable (since L is nonzero only for x_0^{i+1}).

$$A = \begin{pmatrix} a_0^0 & a_0^1 & & a_0^{n-1} \\ a_1^0 & a_1^1 & & a_1^{n-1} \\ & \ddots & \ddots & \\ a_{n-1}^0 & a_{n-1}^1 & & a_{n-1}^{n-1} \end{pmatrix} L = \begin{pmatrix} a_0^0 & 0 & & 0 \\ a_1^0 & a_1^1 & & 0 \\ & \ddots & \ddots & \\ a_{n-1}^0 & a_{n-1}^1 & & a_{n-1}^{n-1} \end{pmatrix} U = \begin{pmatrix} 0 & a_0^1 & & a_0^{n-1} \\ 0 & 0 & & a_1^{n-1} \\ & \ddots & \ddots & \\ 0 & 0 & & 0 \end{pmatrix}$$

Figure 3.6: **Gauss-Seidel Iteration Matrices.** A is split into L, its lower elements, and U, its strictly upper entries. Notice how A's diagonal is entirely in L.

We can write and solve this equation directly thus:

$$l_0^0 x_0^{i+1} = b_0 - \sum_{j>0} u_j^0 x_j \qquad \textit{(1st. row of } Lx = b - Ux\textit{)}$$
$$\therefore x_0^{i+1} = \frac{1}{l_0^0}\left(b_0 - \sum_{j>0} u_j^0 x_j\right)$$

Once we have the value of x_0^{i+1}, we can use it in the second equation, which depends only on x_0^{i+1} and x_1^{i+1}:

$$l_1^0 x_0^{i+1} + l_1^1 x_1^{i+1} = b_1 - \sum_{j>1} u_j^1 x_j \qquad \textit{(2nd. row of } Lx = b - Ux\textit{)}$$
$$\therefore x_1^{i+1} = \frac{1}{l_0^0}\left(b_1 - \sum_{j>1} u_j^1 x_j - l_1^0 x_0^{i+1}\right)$$

Notice how we need x_0^{i+1} to compute x_1^{i+1}. This is not a problem, however, since we computed x_0^{i+1} in the previous step. Thus we can sequentially compute all elements in x^{i+1}:

$$x_k^{i+1} = \frac{1}{l_k^k}\left(b_k - \sum_{j>k} u_j^k x_j - \sum_{j<k} l_k^j x_j^{i+1}\right), \qquad k = 0, 1, \ldots, n-1$$

Since we know that all values in U and L come from A, we can write the equation above as:

$$x_k^{i+1} = \frac{1}{a_k^k}\left(b_k - \sum_{j>k} a_j^k x_j - \sum_{j<k} a_k^j x_j^{i+1}\right), \qquad k = 0, 1, \ldots, n-1 \tag{3.8}$$

Compare Equations (3.6) and (3.8). They are very similar: both require all off-diagonal elements to compute x_k^{i+1}. The only difference is that the Gauss-Seidel iteration loops through the components of x^{i+1} sequentially, using some of the previous results inside a single iteration. This use of extra information makes the Gauss-Seidel method converge faster, in general, than the Jacobi method [43]. However, it still requires that A be strictly diagonally dominant to ensure convergence.

In the following section, we will discuss *direct* methods for linear systems. Unlike iterative approaches, direct methods decompose A into a product of two or more matrices with some specific structure, and then exploit this structure to solve $Ax = b$ directly.

$$A = LU = \begin{pmatrix} a_0^0 & a_0^1 & & a_0^{n-1} \\ a_1^0 & a_1^1 & & a_1^{n-1} \\ & \ddots & \ddots & \\ a_{n-1}^0 & a_{n-1}^1 & & a_{n-1}^{n-1} \end{pmatrix} = \begin{pmatrix} l_0^0 & 0 & & 0 \\ l_1^0 & l_1^1 & & 0 \\ & \ddots & \ddots & \\ l_{n-1}^0 & l_{n-1}^1 & & l_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} u_0^0 & u_0^1 & & u_0^{n-1} \\ 0 & u_1^1 & & u_1^{n-1} \\ & \ddots & \ddots & \\ 0 & 0 & & u_{n-1}^{n-1} \end{pmatrix}$$

Figure 3.7: **LU Decomposition.** A is factored into the product LU, where L is a lower-triangular matrix and U is an upper-triangular matrix. Notice that, unlike in the Gauss-Seidel iteration, the components of L and U are *not* in general taken from A.

3.8 Direct Methods for Linear Systems

We will review two direct methods: the *LU decomposition* [5] and the *Cholesky decomposition* [23], the latter named after french mathematician André-Louis Cholesky. The Cholesky decomposition is a specific instance of the LU decomposition, which can be used for symmetric positive-definite matrices (a positive-definite matrix A is any matrix that satisfies $x^T Ax > 0$ for all x), and it is roughly twice as fast to compute as the LU decomposition [23].

3.8.1 LU Decomposition

The LU decomposition (named for *Lower-Upper* decomposition) factors A into a product of two matrices L and U, such that $A = LU$, where L and U are lower- and upper-triangular matrices, respectively (see Figure 3.7). These matrices should not, however, be confused with the L and U used to split A in the Gauss-Seidel iteration method.

Now, because $A = LU$, we can solve $Ax = b$ by first solving $Ux = y$ and then solving $Ly = b$. Replacing one linear system by two may seem like a waste of computational effort. However, we can exploit the diagonal structure of L and U to solve these two systems in a very efficient fashion.

In order to factor A, we need the concept of *elementary row operations*: Elementary row operations are modifications to the rows of a matrix that do not alter the solution of the linear system [5]. Consider an arbitrary row k of the system $Ax = b$:

$$a_k^0 x_0 + a_k^1 x_1 + \ldots + a_k^{n-1} x_{n-1} = b_k$$

If we multiply all the elements in this row (including b_k) by a non-zero scalar α, the solution to the system remains the same, since any changes made to the left side of the equation are exactly mirrored on the right side. Moreover, if we add the coefficients of two different rows in the system, the solution also remains constant. In general, any modification of a row in the system using a linear combination of the remaining rows does not affect the result.

Any elementary row operation on A can be computed by left-multiplying A by another matrix. We will use this property to define a sequence of elementary row operations that places A into upper-triangular form, and we will keep the product of these operations as another matrix, whose inverse will be in lower-triangular form. In order to show this, we require the fact that the product of two lower-triangular matrices is itself a lower-triangular matrix [5].

Initially, we must find elementary row operations that eliminate all entries below the main diagonal of A's first column. We do this by adding to each i_{th} row of A the first row multiplied by:

$$l_i^0 = -\frac{a_i^0}{a_0^0}, \qquad i = 1, 2, \ldots, n-1 \tag{3.9}$$

To eliminate elements below the diagonal of a general column j (instead of the first column, as in the example above), we replace the zero indices in Equation (3.9) by j:

$$l_i^j = -\frac{a_i^j}{a_j^j}, \qquad i = 1, 2, \ldots, n-1$$

We then gather all l_i^j into one matrix $L_{(j)}$, which represents the row operation, thus:

$$L_{(j)} = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & l_{j+1}^j & \ddots & \\ & & \vdots & & \ddots \\ 0 & & l_{n-1}^j & & 1 \end{pmatrix}, \qquad j = 0, 1, \ldots, n-2$$

$L_{(j)}$ encodes all the operations needed to eliminate elements below the main diagonal in column j. Therefore, we can define a sequence of matrices $A_{(i)}$, where:

$$A_{(0)} = A$$
$$A_{(i)} = L_{(i-1)} A_{(i-1)}$$

Because each $L_{(j)}$ eliminates elements below the diagonal of a single column, $A_{(n-1)}$ is an upper-triangular matrix, which we take to be the U in $A = LU$. To obtain L, first observe that $(L_{(0)}^{-1}(L_{(1)}^{-1} \ldots (L_{(n-2)}^{-1} L_{(n-2)}) \ldots L_{(1)}) L_{(0)}) A = A$. Thus, if $L_{(n-2)} \ldots L_{(1)} L_{(0)} A = U$, we can define $L = L_{(0)}^{-1} L_{(1)}^{-1} \ldots L_{(n-2)}^{-1}$, guaranteeing, by definition, that $LU = A$. Computing the inverses of $L_{(j)}$ is a very straightforward task. In fact,

$$L_{(j)}^{-1} = \begin{pmatrix} 1 & & & & 0 \\ & \ddots & & & \\ & & 1 & & \\ & & -l_{j+1}^j & \ddots & \\ & & \vdots & & \ddots \\ 0 & & -l_{n-1}^j & & 1 \end{pmatrix}, \qquad j = 0, 1, \ldots, n-2,$$

that is, we obtain the inverse of $L_{(j)}$ simply by replacing the off-diagonal elements of $L_{(j)}$ with their additive inverses.

Now that we have L and U, we solve $Ax = b$ in two steps: we first solve $Ly = b$, and then $Ux = y$. But because L and U are lower- and upper-triangular

matrices respectively, we can solve these systems using forward- and back-substitution. Forward-substitution is the exact process we used in Equation (3.8) to solve $Lx = b - Ux$. Back-substitution, on the other hand, is an analogous method which starts from the last element x_{n-1} (instead of x_0), thus being suitable for upper-triangular matrices.

The process of systematically eliminating off-diagonal elements is known as *Gaussian Elimination* [5], and the technique to build L as a sequence of matrix multiplications is called the *Doolittle algorithm* [19]. One advantage of using LU decomposition is that, once A is factored, we can solve $Ax = b$ for different b without having to recompute the decomposition. The time complexity of LU decomposition is $O(n^3)$ [19].

However, there is a variant of LU decomposition, which can be used whenever A is symmetric and positive-definite, that requires approximately half as many operations. This variant is called the *Cholesky decomposition*, which we discuss below.

3.8.2 Cholesky Decomposition

If A is symmetric and positive-definite, there is a unique decomposition of A such that $A = LL^T$. This decomposition has the advantage that L and L^T have the same non-zero pattern, so we only need to store a single lower-triangular matrix L in memory. Furthermore, the algorithm to compute the Cholesky decomposition is roughly twice as efficient as the standard LU decomposition.

The Cholesky decomposition algorithm is also much simpler than the general LU decomposition method, since it exploits the symmetry of A to directly compute the elements in L. This obviates the need to compute and store elementary row operations. To demonstrate this, we will use a 3×3 matrix as an example:

$$A = LL^T = \begin{pmatrix} l_0^0 & 0 & 0 \\ l_1^0 & l_1^1 & 0 \\ l_2^0 & l_2^1 & l_2^2 \end{pmatrix} \begin{pmatrix} l_0^0 & l_1^0 & l_2^0 \\ 0 & l_1^1 & l_2^1 \\ 0 & 0 & l_2^2 \end{pmatrix} = \begin{pmatrix} {l_0^0}^2 & & \textit{Symm.} \\ l_1^0 l_0^0 & {l_1^0}^2 + {l_1^1}^2 & \\ l_2^0 l_0^0 & l_2^0 l_1^0 + l_2^1 l_1^1 & {l_2^0}^2 + {l_2^1}^2 + {l_2^2}^2 \end{pmatrix}$$

This lets us define direct expressions for the values in L:

$$l_j^j = \sqrt{a_j^j - \sum_{k=0}^{j-2} {l_j^k}^2}, \qquad j = 0, 1, \ldots, n-1 \tag{3.10a}$$

$$l_i^j = \frac{1}{l_j^j} \left(a_i^j - \sum_{k=0}^{j-2} l_i^k l_j^k \right), \quad i = j+1, \ldots, n-1, \quad j = 0, 1, \ldots, n-1 \tag{3.10b}$$

The condition that A be positive-definite ensures that the term under the square root in (3.10a) is always positive. Equations (3.10a) and (3.10b) indicate that we can compute l_i^j using its immediate upper and left neighbors. Therefore, the Cholesky algorithm initially computes l_0^0 and proceeds through L row by row [23].

Once we have $A = LL^T$, we solve $Ax = b$ in the same way as when using the LU decomposition: we compute $Ly = b$ and then $L^T x = y$. The Cholesky decomposition has the same advantage as the LU decomposition: we only need to factor A once to solve the system for arbitrary b.

One of the main applications of the Cholesky decomposition is to obtain *Least-Squares* approximations to over-determined systems of equations. Because the

equations used to solve linear least-squares problems naturally involve a symmetric, positive-definite matrix, the Cholesky decomposition is an excellent approach to solve these systems. We will study Linear Least-Squares methods in detail in the next Section.

3.9 The Least-Squares Method for Over-determined Systems

Recall that, in Section 3.6, we explored linear systems that do not have a solution, that is, systems that are *over-determined*. These situations arise for $m \times n$ matrices whenever $m > n$, where the image of the transformation is embedded in a space larger than the domain of the linear map. In these cases, we can choose a b such that there is no x satisfying $Ax = b$. Nevertheless, it makes sense to ask whether there exists an x^* that "best approximates" x in some sense. Observe that, for an exactly determined system $Ax = b$, $\|Ax - b\| = 0$. Therefore, we can use the standard Euclidean norm to find an x^* such that $\|Ax^* - b\|$ is minimized over all possible x. This is exactly what the method of linear least-squares does. First, we must write $\|Ax^* - b\|$ in terms of a matrix multiplication (recall that inner products can be expressed as matrix multiplications, transposing the first argument):

$$\begin{aligned} \|Ax^* - b\| &= \langle Ax^* - b, Ax^* - b \rangle \\ &= (Ax^* - b)^T (Ax^* - b) \end{aligned}$$

We can distribute the transposition thus:

$$\begin{aligned} (Ax^* - b)^T (Ax^* - b) &= \left((Ax^*)^T - b^T \right) (Ax^* - b) \\ &= \left(x^{*T} A^T - b^T \right) (Ax^* - b) \end{aligned}$$

Now we distribute the matrix multiplications over the subtractions:

$$\left(x^{*T} A^T - b^T \right) (Ax^* - b) = x^{*T} A^T Ax^* - x^{*T} A^T b - b^T Ax^* + b^T b \tag{3.11a}$$

Observe that the terms $-x^{*T} A^T b$ and $-b^T Ax^*$ are actually the same, since

$$\begin{aligned} -x^{*T} A^T b &= -(Ax^*)^T b \\ &= -\langle Ax^*, b \rangle \\ -b^T Ax^* &= -\langle b, Ax^* \rangle \\ &= -\langle Ax^*, b \rangle \end{aligned}$$

This allows us to rewrite Equation (3.11a):

$$\left(x^{*T} A^T - b^T \right) (Ax^* - b) = x^{*T} A^T Ax^* - 2b^T Ax^* + b^T b \tag{3.12a}$$

$$\therefore \|Ax^* - b\| = x^{*T} A^T Ax^* - 2b^T Ax^* + b^T b \tag{3.12b}$$

We wish to find an x^* that minimizes this distance, so we take the derivative of (3.12b) with respect to x^*, and set that to zero:

$$\frac{\partial}{\partial x^*} \left(x^{*T} A^T Ax^* - 2b^T Ax^* + b^T b \right) = 0$$

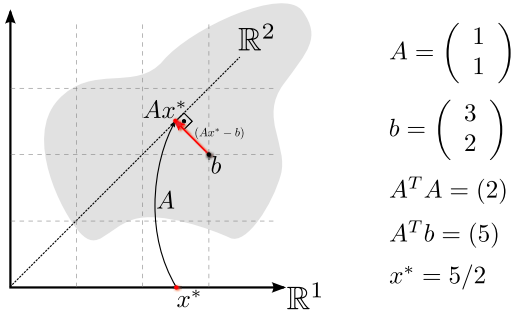


Figure 3.8: **Normal Equations in $\mathbb{R}^2$.** The Normal Equations are used to find a solution x^* that minimizes the norm of the residual vector $(Ax^* \quad b)$. This solution is the best approximation of b in the range of A. Notice how the residual is exactly orthogonal to A's column space.

In this context, the derivative with respect to x^* means a vector that contains the partials $\partial/\partial x_i^*$ for each component i of x^*. With this definition, the derivative behaves very much like it does in single-valued functions: $\frac{\partial}{\partial x^*}(\quad 2b^TAx^*) = \quad 2A^Tb$, and $\frac{\partial}{\partial x^*}(x^{*T}A^TAx^*) = 2A^TAx^*$. Naturally, $\frac{\partial}{\partial x^*}(b^Tb) = 0$, since b^Tb is constant with respect to x^*. Therefore, we can solve the derivatives thus:

$$2A^TAx^* \quad 2A^Tb = 0 \tag{3.13a}$$
$$2A^TAx^* = 2A^Tb \tag{3.13b}$$
$$\therefore A^TAx^* = A^Tb \tag{3.13c}$$

Equation (3.13c) is called the *Normal Equation*, because the residual $Ax^* \quad b$ is orthogonal to the column-space of A (see Figure 3.8), and its solution x^* is the vector that minimizes $\|Ax^* \quad b\|$.

In order to solve the Normal Equations, we must compute A^TA and A^Tb, which are simple matrix multiplications, and solve the linear system in Equation (3.13c). However, since A^TA is symmetric positive-definite (for over-determined matrices), we can use the Cholesky decomposition discussed above to efficiently solve the system [23].

Up to this point, we discussed aspects of matrices and linear transformations from a purely algebraic standpoint. However, when designing algorithms that involve systems containing many hundreds or even thousands of equations, it becomes necessary to implement data structures that are economical in memory. Obviously, the direct storage format for a matrix requires $O(n^2)$ space. This may be acceptable for small matrices, but when n is of the order of magnitude of hundreds or thousands, storing and manipulating data in this format becomes infeasible. Because of this, we need more space-efficient data structures to represent large matrices. If a matrix is dense (that is, all or most of its elements are non-zero), then there is little to be done, since we need information about each individual element in the matrix. However, if the matrix is sparse, thus containing a large amount of zeros, we can save a lot of space by explicitly storing only the non-zero elements in the matrix.

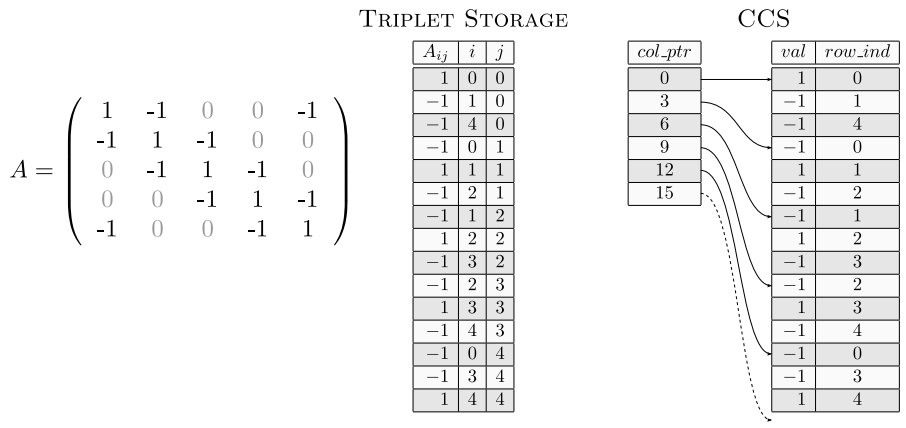


Figure 3.9: **Sparse Matrix Representations.** The Triplet and CCS formats do not explicitly store zeroes in the matrix, which makes them very economical for sparse systems. In the Triplet format, all row- and column-indices are stored along with the numerical entries, while in the CCS format only pointers to the start of each column are stored.

3.10 Storing Large Sparse Matrices Efficiently

Compressed storage formats save space by only representing the non-zero elements in a matrix, along with extra information to locate the row and column indices of each element. In this Section, will review two popular formats for storing sparse matrices: the *Triplet* format and the *Compressed Column Storage*.

3.10.1 Triplet Storage

The Triplet [8] format is very simple: it stores an array containing n_{nz} entries, where n_{nz} is the number of non-zero elements in the matrix. Each entry in this array corresponds to a single non-zero element in the matrix, and is composed of three fields: two integers i and j, representing the row and column of each element, respectively, and a single- or double-precision floating point number, which stores the non-zero value itself. This format requires $3n_{nz}$ positions in memory, which is already vastly superior to the n^2 positions required by the direct storage.

The CCS format shrinks this representation still further, by compressing the column index information for the values in the matrix.

3.10.2 Compressed Column Storage

The Compressed Column Storage (CCS) [8] format exploits the fact that non-zero elements in a column of a sparse matrix are usually placed in consecutive rows. Therefore, it is only necessary to point to the *first* element in each column, and to store explicitly only the row indices.

To implement this, the CCS format uses three arrays: a single- or double-precision floating point array `val`, containing n_{nz} entries (one for each non-zero element in the matrix), and two integer arrays: `row_ind`, also containing n_{nz} entries, and `col_ptr`, containing $n + 1$ entries. `row_ind` indicates the row of each corresponding element in `val`, and `col_ptr` indicates, for each column of A, where its values *start* in `val` and `row_ind`. It is conventional to set `col_ptr`$[n]$, the last element in the array, to n_{nz}, to signal where the last column ends. Observe that `CHOLMOD`, the computational library

we use to solve Least-Squares systems in our work, uses CCS matrices throughout its Application Programming Interface. The CCS format requires $2n_{nz} + n + 1$ memory positions, which is an improvement over Triplet Storage, since most sparse matrices have more than a single non-zero element per column. Figure 3.9 illustrates these two compressed formats with a simple example.

3.11 Final Remarks

In this Chapter, we have reviewed most basic linear-algebraic concepts, including elementary vector operations, the notion of linear combinations, linear subspaces and linear independence, as well as basis vectors and orthogonality. We also studied characterizations of linear systems of equations, including under-determined, exactly determined and over-determined systems. We observed how these characterizations are strongly related to the existence of an inverse matrix, and also reviewed the most popular direct and iterative methods to solve systems of the form $Ax = b$. We reviewed how the Least-Squares method can be used to find an approximate solution to over-determined systems, and explained two compressed storage formats for large matrices. In the next Chapter, we will turn our attention to the Laplace Operator, a very useful differential operator that will be of fundamental importance to our inter-surface mapping algorithm.

4 THE CONTINUOUS AND DISCRETE LAPLACE OPERATORS

The Laplace operator (also called the Laplacian) of a function f (denoted by Δf or $\nabla^2 f$) is a widely used differential operator named after the french mathematician Pierre-Simon de Laplace. Intuitively, it represents the "roughness" of a function, as it captures how much the function at a given point differs from its immediate neighbors. In fact, as we will see below, the application of the Laplacian acts exactly as a high pass filter over a signal, and this property is commonly used in image processing to design edge-detection filters [57]. The Laplacian is a linear operator that maps the set of scalar-valued functions to itself, and is formally defined as the *divergence* of the *gradient* of a function [31].

The simplest way to characterize the Laplacian operator works on twice differentiable functions over Euclidean spaces. However, the Laplacian can be easily extended to regular grids arising from finite-differences methods, to arbitrary weighted graphs and even to more general Riemannian manifolds (where it is known as the Laplace-Beltrami operator [42]). In this Chapter, we will review in detail all these approaches to defining the Laplacian operator, concluding with discrete formulations that are directly suitable for use with triangle meshes.

4.1 The Continuous Laplace Operator in $\mathbb{R}^n$

As we mentioned previously, the Laplace operator for a twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as the divergence of the gradient of f. Below we explore this definition in order to provide some intuition towards the behavior of the Laplacian operator.

First, recall that the gradient of a scalar field is a *vector field* whose vectors point in the direction of steepest ascent for the scalars at each point. The divergence, on the other hand, is an operator that maps a vector field back to a scalar field whose elements represent the *source* or *sink* magnitude of the input vector field. Intuitively, sources and sinks are regions in the domain where at least part of the surrounding vectors point away from or into the same location, respectively. Because of this, the divergence can be used to indicate how much a vector field changes at an infinitesimal neighborhood around each point in the domain. For a constant vector field (i.e. the gradient of a linear function), the divergence will always be zero, as there are no source or sink components anywhere in the field (vector fields that have this property are called *incompressible* [31]).

We can combine these two definitions to conclude that the Laplacian operator

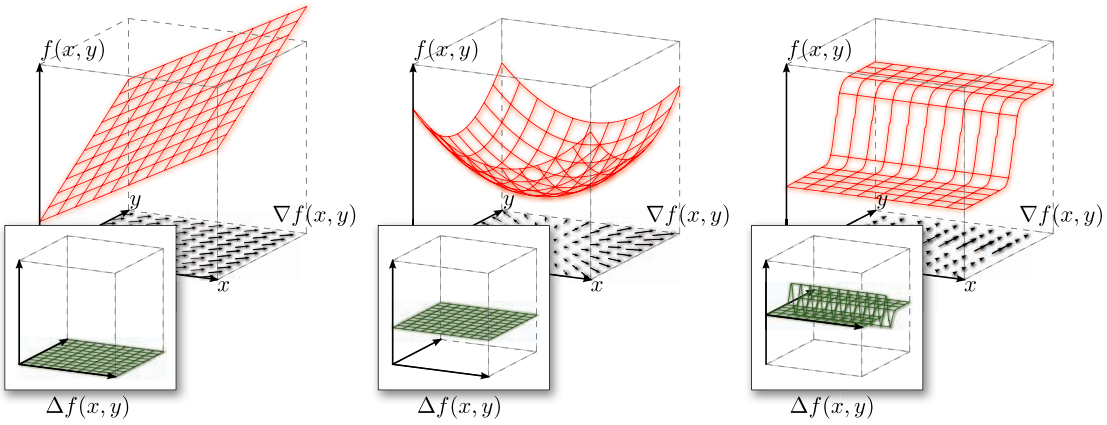


Figure 4.1: **Continuous Laplacian.** Three different functions $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, along with their gradients and associated Laplacians. A linear function (a) has constant gradient over the entire domain, and therefore zero Laplacian. A positive-definite quadratic form (b) has a linearly varying gradient field, and a non-zero Laplacian. Finally, a step function (c) has a characteristic gradient field and a clearly visible wedge on the Laplacian.

gives a measure of how much a function's gradient changes around infinitesimal neighborhoods. This is analogous to a second derivative, in the sense that it gives a measure of the function's "roughness" over the domain. In fact, for a continuous function $f : \mathbb{R} \rightarrow \mathbb{R}$, the Laplacian operator *is* the second derivative, and in general manifolds the Laplacian is the simplest second-order differential operator that can be defined [42]. Figure 4.1 illustrates these concepts with some simple functions defined over $\mathbb{R}^2$.

4.1.1 Deriving the Laplacian in Cartesian Coordinates

To express the continuous Laplacian operator in Cartesian coordinates, we must simply expand the definitions of the gradient and divergence of a twice differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, where $x_1, x_2, ..., x_n$ denote the coordinate axes for the domain:

$$grad\ f = \nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ... \frac{\partial f}{\partial x_n}\right]^T \tag{4.1}$$

The definition of the divergence is also straightforward. For a vector field $\mathbf{F}$, the divergence operator is defined thus[1]:

$$div\ \mathbf{F} = \nabla \cdot \mathbf{F} = \sum_{i=1}^{n} \frac{\partial \mathbf{F}_i}{\partial x_i} \tag{4.2}$$

where $\mathbf{F}_i$ are the coefficients that multiply the canonical basis vectors for $\mathbb{R}^n$ to form $\mathbf{F}$. The divergence operator can be thought of, with some abuse of notation, as the "vector" $[\partial/\partial x_0, \partial/\partial x_1, \ldots, \partial/\partial x_{n\ 1}]^T$, which justifies our use of dot product notation in Equation (4.2).

[1]We follow the usual notation of boldface uppercase letters to denote vector fields and lowercase letters to denote scalar fields

Combining Equations (4.1) and (4.2) trivially gives us the definition of the Laplacian operator in Cartesian coordinates:

$$\Delta f = \nabla^2 f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}$$

Because the Laplacian is a linear operator, it is interesting to study its set of *eigenfunctions* and their corresponding *eigenvalues*. Eigenfunctions are exact analogues of eigenvectors in linear algebra — i.e. functions f where $\Delta f = \lambda f$ for some scalar λ, called the eigenvalue of f. Eigenvalues are in general complex numbers, but since the Laplacian is a self-adjoint operator [36], all its eigenvalues are real. This also implies that a discretization of the Laplacian must result in a symmetric matrix, as we will demonstrate in Section 4.2.

The Spectral Theorem states that any self-adjoint operator can be diagonalized in some basis. This means that self-adjoint operators can be thought of as non-uniform scales in some coordinate system. In particular, this coordinate system is perfectly aligned with the operator's eigenfunctions, and the scaling factors are the eigenfunctions' corresponding eigenvalues. This also motivates what is known as the *eigendecomposition* of a symmetric matrix A [6]:

$$A = Q\Lambda Q^{-1},$$

where Q is a matrix whose columns are the eigenvectors of A, Λ is a diagonal matrix consisting of A's eigenvalues, and Q^{-1} is the inverse of Q. Notice that, since the eigenvectors of A are all orthogonal (again via the Spectral Theorem), Q and Q^{-1} are unitary transformations. Q^{-1} can be understood as a rotation that takes the orthogonal basis formed by the eigenvectors of A to the canonical basis, and Q is its inverse (taking the canonical basis back to the basis formed by the eigenvectors of A). Therefore, an eigendecomposition of a matrix A can be understood as a sequence of three distinct operations: first, vectors are rotated to align themselves with the canonical basis; then, they are scaled by the eigenvalues of A; and finally, they are rotated back to their original directions. This observation explains our previous statement that self-adjoint operators can be understood as a scale in some orthogonal basis. Below we will explore the eigenfunctions of the Laplacian operator, and show that its application over a function acts as a high-pass filter.

For 1-dimensional Euclidean spaces, the set of eigenfunctions of the Laplacian corresponds exactly to sine and cosine waves of increasing frequency. To show this, we will express a generic trigonometric wave as $f(x) = asin(\phi x + x_0)$, where a represents the amplitude, ϕ the frequency and x_0 the phase information for the wave. We use this representation because it contains both sine and cosine waves in one simple formula (since $cos(x) = sin(x + \pi/2)$). Now, we simply carry out the differentiations:

$$f(x) = asin(\phi x + x_0) \tag{4.3a}$$

$$\frac{\partial f}{\partial x} = \phi acos(\phi x + x_0) \tag{4.3b}$$

$$\Delta f = \frac{\partial^2 f}{\partial x^2} = -\phi^2 asin(\phi x + x_0) \tag{4.3c}$$

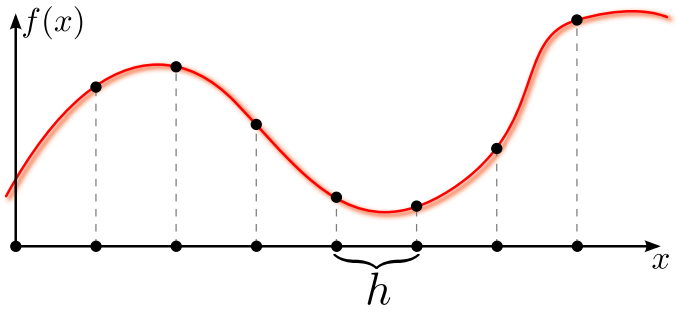


Figure 4.2: **Regular Discretization of a 1-dimensional Grid.** A continuous function $f(x)$ can be approximated by sampling its values at regular intervals of h.

Dividing Equation (4.3c) by (4.3a) gives us ϕ^2, which corresponds to f's eigenvalue λ. Observe that as the frequency of the waves increases, so does the magnitude of their eigenvalues $|\phi^2|$. This means that the Laplacian operator emphasizes waves of higher frequencies and attenuates waves of lower frequencies, making it a typical high-pass filter. Also notice that the Laplacian eigenfunctions correspond exactly to the basis in which the standard Fourier transform is applied. This property remains true for Laplacian operators over different domains, including multi-dimensional Euclidean spaces, as well as regular grids and undirected graphs. We can therefore build processes analogous to Fourier analysis on these non-standard domains [54].

In many computational applications the domain of interest in discretized with some small step-size to approximate a continuous region. In the next section we will derive approximations for the Laplacian operator using finite-differences methods over n-dimensional Euclidean spaces.

4.2 Discrete Laplacian on a Regular Grid

Suppose we have a twice differentiable function $f : \mathbb{R} \to \mathbb{R}$ which will be approximated by sampling it over a discrete set of points set a distance of h apart (see Figure 4.2 for an example). We can find an approximation for Δf using a finite difference scheme such as this:

$$\Delta f = f''(x) \simeq \frac{f'(x + h/2) \quad f'(x \quad h/2)}{h} \tag{4.4}$$

We chose a step-size of $h/2$ because if we expand the derivative terms in Equation (4.4), again using finite differences, we get

$$f'(x + h/2) = \frac{f(x + h) \quad f(x)}{h} \tag{4.5a}$$

$$f'(x \quad h/2) = \frac{f(x) \quad f(x \quad h)}{h} \tag{4.5b}$$

which allows us to substitute (4.5a) and (4.5b) into (4.4), resulting in

$$\Delta f(x) = f''(x) \simeq \frac{f(x + h) \quad 2f(x) + f(x \quad h)}{h^2} \tag{4.6}$$

Equation (4.6) defines the standard Laplacian *stencil* for a 1-dimensional function, and it can be directly extended to higher dimensional spaces. In the case of a

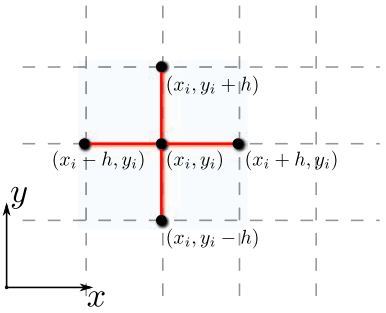


Figure 4.3: **Five-point Stencil for a 2-dimensional Grid.** The Laplacian on a regular grid can be approximated by sampling a central element i, located at (x_i, y_i), and its four immediate neighbors. This process is analogous to what is done in one dimension, and can be extended to arbitrarily high-dimensional Euclidean spaces.

regular, isotropic two-dimensional grid, we get the widely used *five-point stencil* (see Figure 4.3):

$$\Delta f(x,y) \simeq \frac{f(x+h,y) + f(x \quad h,y) + f(x,y+h) + f(x,y \quad h) \quad 4f(x,y)}{h^2} \tag{4.7}$$

The five-point stencil approximation to the Laplacian is widely used as an edge-detection filter in image processing, where it is implemented with a convolution kernel equivalent to the stencil [57]. The finite differences scheme used to derive Equations (4.6) and (4.7) is a second-order approximation, and therefore incurs an error of $O(h^2)$ [19].

4.2.1 Matrix Representation of the Laplacian

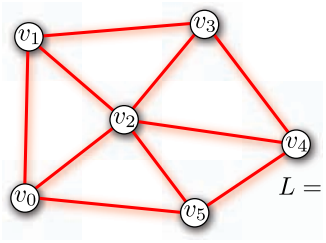
Let us now assume that f is defined over a finite $m \times n$ grid, for positive m and n, where $h = 1$. Furthermore, assume that this grid has a toroidal topology, that is, the rightmost points are actually connected to the leftmost points, "wrapping around" the x-axis, and that the same is true for the top and bottom points. This is done merely for convenience, as it obviates the need to construct special cases for the boundaries. Given this regular grid, we can represent f as a mn dimensional vector, that is, $f = [f_0, \ f_1, \ \ldots, \ f_{mn \ 1}]^T$. Because the Laplacian is a linear operator, we can write Δf in matrix form, thus:

$$\Delta x = \quad Lx,$$

where L is the following *symmetric* matrix

$$L = \begin{pmatrix} 4 & -1 & -1 & & & -1 & -1 \\ -1 & 4 & -1 & -1 & & & -1 \\ -1 & -1 & 4 & -1 & -1 & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ -1 & & & -1 & -1 & 4 & -1 \\ -1 & -1 & & & -1 & -1 & 4 \end{pmatrix}$$

Notice the difference in sign between the elements of L and our previous formulation of the stencil. This allows us to bridge the gap between the five-point stencil Laplacian



$$L = \begin{pmatrix} 3 & -1 & -1 & 0 & 0 & -1 \\ -1 & 3 & -1 & -1 & 0 & 0 \\ -1 & -1 & 5 & -1 & -1 & -1 \\ 0 & -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 0 & -1 & 3 \end{pmatrix}$$

Figure 4.4: **A Simple Graph and its associated Laplacian Matrix.** Notice how L is symmetric, meaning its eigenvectors and eigenvalues are all real and orthogonal. Moreover, the number of non-zero eigenvalues of L equal the number of connected components in the graph.

and the generalized *Graph Laplacian*, where the standard sign convention is like L above [41]. With this convention, the eigenvalues of L are all *non-negative*, instead of being all non-positive, and the eigenvectors remain unchanged. Furthermore, the regular structure of the entries of L demonstrates our choice of a toroidal topology: had we kept the boundary points, rows corresponding to points on the edge would contain only three 1 elements (and their on-diagonal value would be 3), and the four rows corresponding to the corner points would contain only two 1 entries, with a 2 on the main diagonal.

Because L is a symmetric matrix, all its eigenvalues are real, and all its eigenvectors are orthogonal. As we will see below, the rank of L is mn 1, which means that L has only one zero eigenvalue (in fact, the number of zero eigenvalues of the Laplacian in general is equivalent to the number of connected components in the domain [7, 41]). Therefore, the eigenvectors of L can be used to define an orthogonal basis for some (mn 1)-dimensional subspace of $\mathbb{R}^{mn}$. This orthogonal basis, along with the zero-eigenvector of L, defines a full-rank orthogonal basis for $\mathbb{R}^{mn}$, which means any function f can be expressed as a linear combination of these basis vectors. Recall that the eigenvectors of the continuous Laplacian form a basis for the Fourier domain; this is also the case for 2-dimensional grids (and, in fact, general graphs and Riemmanian manifolds), meaning that the Laplacian eigenvectors provide us with a tool to do Fourier analysis on domains other than the real line. As we will see below, the eigenvectors of a modified graph Laplacian allow us to do signal processing on functions defined directly over a triangle mesh, including implementing low-pass and high-pass filters. To do this, however, we must further generalize our definition of the Laplacian to work on arbitrary undirected graphs.

4.3 Discrete Laplacian on Undirected Graphs

Suppose we have a graph $G = (V, E)$, where V is the set of vertices, E is the set of edges, and v_i is the i_{th} vertex of G. We may use this graph as the domain for our functions f, which means that f can now be expressed as $|V|$-dimensional vectors $f = \begin{bmatrix} f_0, & f_1, & \ldots, & f_{|V|} & _1 \end{bmatrix}^T$, where f_i is the value of f on v_i. We can define a Laplacian operator that acts on these functions in much the same way as we did

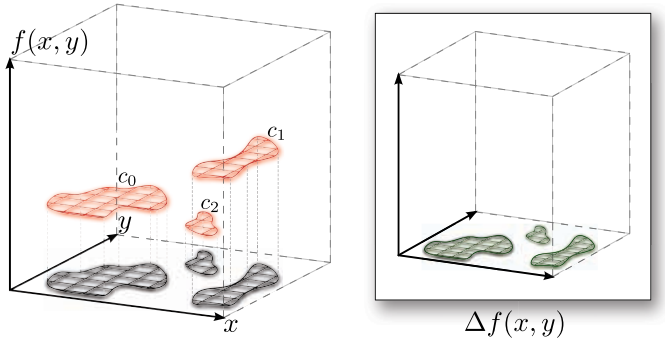


Figure 4.5: **A Piecewise-constant Function and its Laplacian.** In a domain that has more than one connected component any piecewise-constant function will be a 0-eigenfunction of the Laplacian, as long as it is constant on each component. In this example, any particular choice of c_0, c_1 and c_2 will yield a function in the Laplacian's null space. This is true for the continuous as well as the discrete formulations of the Laplacian operator.

previously, by defining the *Graph Laplacian* L as a $|V| \times |V|$ symmetric matrix thus:

$$l_i^j = \begin{cases} degree(v_i) & i = j \\ 1 & (i,j) \in E \\ 0 & otherwise \end{cases} \tag{4.8}$$

This definition is equivalent to our previous definition of L for 4-regular graphs with a toroidal topology, and it respects the usual sign convention for the Graph Laplacian matrix [41].

As we mentioned in Section 4.2, the multiplicity of the 0 eigenvalue of L equals the number of connected components in the graph. We can demonstrate this using two well known facts from Spectral Graph Theory: the eigenvalues of the disjoint sum of two graphs G and H is the union of the eigenvalues of G and H with the multiplicities added [41] and the multiplicity of the 0 eigenvalue in a connected graph is exactly one [7]. Therefore a graph G with k connected components has the 0 eigenvalue with multiplicity k, as G is the disjoint sum of k connected graphs H_i for $0 \leq i < k$, each with the 0 eigenvalue appearing exactly once.

Using the rank-nullity theorem [33], we can show that $rank(L) = |V| \quad k$, where k is the number of connected components in the graph and $|V|$ is the number of vertices. This means that linear systems involving L have singular solutions with k degrees of freedom, and therefore we will need to add k linearly independent rows to L to turn it into a non-singular matrix. This fact is very relevant to our mapping technique, where we need to add a number of constraints to equations of the form $\Delta f = 0$ to determine f uniquely.

Knowing that a connected graph has the eigenvalue 0 with multiplicity one, it makes sense to ask what is its corresponding eigenvector. If we take $f = [1, 1, \ldots, 1]^T$, then $\Delta f = 0$ for any connected graph, since the 1 entries in the matrix negate the contribution of each vertex's individual degree. In fact, any constant f will have this very same effect. This is true for any definition of the Laplacian, including the continuous definitions we explored in Section 4.1, where this fact can be explained by the observation that a constant function has zero second derivatives. Notice that

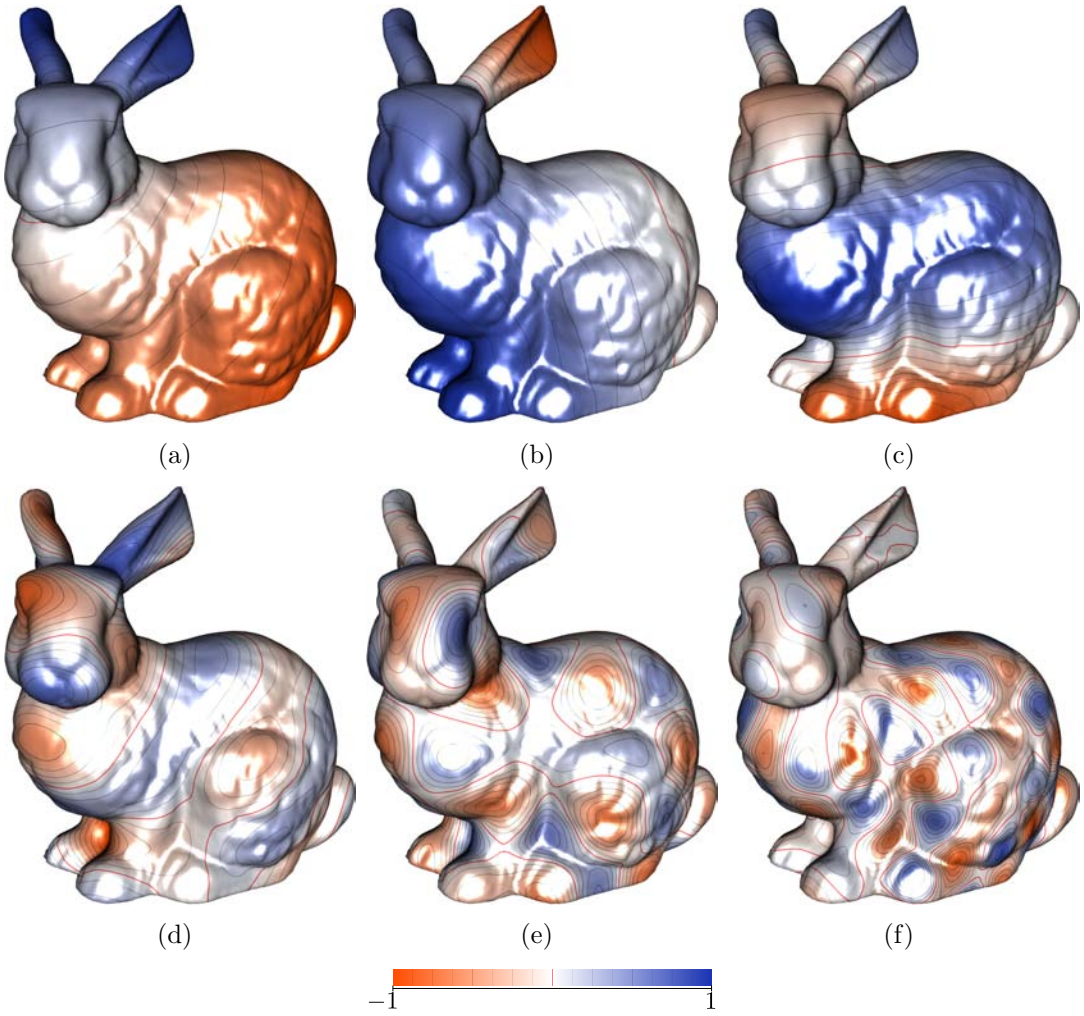
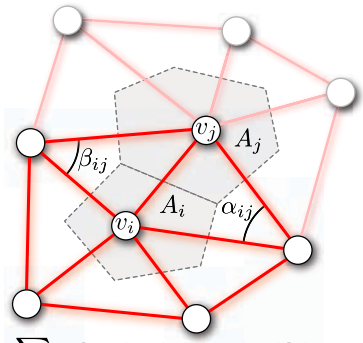


Figure 4.6: **Laplacian Eigenvectors.** The eigenvectors of the Laplacian matrix on a mesh define functions that share many similarities with sine waves of increasing frequency. This Figure shows the first non-constant eigenvector (a), also called the Fiedler vector [32], along with the second (b), 10th (c), 50th (d), 100th (e) and 150th (f) eigenvectors.

if $\Delta f = 0$ for $f = [1, 1, \ldots, 1]^T$, then $\Delta f = \lambda_0 f$, where $\lambda_0 = 0$. Thus we find that the eigenvector corresponding to the 0 eigenvalue is the constant function. Moreover, when a graph G is composed of k connected components H_k, any function that is constant in each component will be a 0-eigenvector of L, despite the fact that the constants can be different for different connected components. This provides some intuition to the fact that the dimension of L's null space equals the number of connected components in the domain (see Figure 4.5).

The definition of the Graph Laplacian can be directly applied to triangle meshes, simply by considering a graph G where its vertices and edges are the vertices and edges of the mesh, respectively. This construction allows us to extend all the properties of the Laplacian directly to triangle meshes, including a Fourier-like basis to represent functions over the mesh (see Figure 4.6). Below we will explore some problems with using the standard Graph Laplacian on triangle meshes, and review existing literature that proposes modifications to the original Laplacian matrix to overcome some of these problems.



$$\Delta f(v_i) = \sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j) / \sqrt{A_i A_j}$$

Figure 4.7: **DEC Cotangent Weights Laplacian.** This formulation of the discrete Laplacian takes mesh geometry into account, using the angles α_{ij} and β_{ij} as well as the dual cell areas A_i and A_j. The dual cell areas represent the areas of the Voronoi cells of vertices v_i and v_j.

4.4 The Laplacian Operator over Triangle Meshes

The Graph Laplacian operator defined in Section 4.3 is not entirely suitable for use with a triangle mesh, since it does not take any geometric information about the vertices' locations into account. In particular, the Graph Laplacian (which is also known as the *Combinatorial* Laplacian) has the same set of eigenfunctions for graphs with different embeddings [54] and, as an approximation to the continuous Laplacian, it supposes a uniform sampling of the mesh [34]. Ideally, one would like to define a discrete Laplacian that is independent of the particular meshing of the surface, and that maintains desirable properties such as symmetry and positive-definiteness, among others. However, Wardetzky et. al [55] prove that such an operator does not exist in general.

Despite these limitations, many different discretized Laplacian operators exist in the literature [38, 34, 32]. The simplest Laplacian formulation that takes geometry into account is the ubiquitous *cotangent-weights* scheme [34, 38]. This approach involves modifying the Laplacian equations thus:

$$\Delta f(v_i) = \sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i \quad v_j) \tag{4.9}$$

where $\mathcal{N}_1(v_i)$ denotes the 1-ring of the i_{th} vertex, and α_{ij} and β_{ij} are the angles opposite the edge (v_i, v_j) (see Figure 4.7). Equation (4.9) is simply a different arrangement of terms for a single row of the Laplacian matrix L, since if we distribute the multiplication over $(v_i \quad v_j)$ we get the following entries for the matrix:

$$\begin{aligned} l_i^j &= [cot(\alpha_{ij}) + cot(\beta_{ij})] \\ l_i^i &= \sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] \end{aligned}$$

We will adopt this summation convention to describe the remaining Laplacian formulations. The cotangent-weights scheme incorporates some geometrical information into the Laplacian, and Hildebrandt et. al [18] show that it converges to the continuous Laplace-Beltrami operator for sufficiently fine meshes, under some

Discrete laplacian operators	
Name	Equation
Combinatorial	$\sum_{j \in \mathcal{N}_1(v_i)} (v_i - v_j)$
Cotangent Weights	$\sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)$
Normalized Cotangent Weights	$\sum_{j \in \mathcal{N}_1(v_i)} \frac{[cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)}{A_i}$
Symmetrized Cotangent Weights	$\sum_{j \in \mathcal{N}_1(v_i)} \frac{[cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)}{(A_i + A_j)}$
DEC Cotangent Weights	$\sum_{j \in \mathcal{N}_1(v_i)} \frac{[cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)}{\sqrt{A_i A_j}}$

Table 4.1: **Summary of Discrete Laplacians on Triangle Meshes.** Different formulations for the discrete Laplacian operator exist on triangle meshes, each with its own set of desirable properties. Unfortunately, there is no single discrete Laplacian that has *all* the properties present in the continuous operator.

conditions. However, this formulation is still dependent on mesh sampling [54], and therefore it is usually normalized by the dual cell area A_i of the vertex v_i:

$$\Delta f(v_i) = \sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)/A_i$$

The dual cell area of a vertex is the area of the Voronoi cell corresponding to that vertex on the mesh. Unfortunately, this normalization scheme violates the condition that L must be symmetric, since elements l_i^j and l_j^i will be divided by A_i and A_j respectively, which are in general different values. This normalization is therefore completely unsuitable for any kind of spectral analysis. Levy [32] proposes an empirical symmetrization solution that restores eigenvector orthogonality, where the cotangent weights are normalized by the sum of the dual areas of vertices v_i and v_j:

$$\Delta f(v_i) = \sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)/(A_i + A_j)$$

However, as Vallet and Levy [54] point out, this normalization is still dependent on the meshing. To remedy this, they use a formulation based on Discrete Exterior Calculus (DEC) to carefully define and symmetrize a Laplace-Beltrami operator that is truly mesh-independent (see Figure 4.7):

$$\Delta f(v_i) = \sum_{j \in \mathcal{N}_1(v_i)} [cot(\alpha_{ij}) + cot(\beta_{ij})] (v_i - v_j)/\sqrt{A_i A_j}$$

However, not even this careful definition of the Laplacian satisfies all desired properties, as it is not positive semi-definite for general meshes (it usually requires meshes with good-quality triangles to maintain positive semi-definiteness). Table 4.1 summarizes this taxonomy of Laplacian operators, including their formulas for ease of reference. Notice how the only thing that differs between formulations is the weighing scheme for the term $(v_i - v_j)$, and also how $l_i^i = -\sum_{j \in \mathcal{N}_1(v_i)} l_i^j$ always.

The Laplacian operator, regardless of the domain in which it is defined, appears in many physical and computational applications, where it is typically used to solve *Laplace's Equation*, which we will discuss in some detail below.

4.5 Laplace's Equation

There is a particular set of functions f that have zero Laplacian everywhere, i.e., functions for which:

$$\Delta f = 0 \tag{4.10}$$

Equation (4.10) is known as Laplace's Equation [31], and functions that satisfy this property are known as *harmonic functions*. Naturally, harmonic functions can be defined and studied in any domain in which a Laplacian can be defined, including all the examples we discussed previously. The non-homogeneous version of Laplace's equation, which can be written as $\Delta f = g$, is known as *Poisson's Equation*, after french mathematician Siméon-Denis Poisson [31]. Recall that the Laplacian informally represents a function's "roughness", as it measures how much $f(x)$ differs from the average of its immediate neighbors. Because of this, harmonic functions are typically very smooth across their domain.

In order to solve Laplace's Equation on triangle meshes, we must set up the following linear system:

$$-Lf = 0,$$

where L is any formulation of the discrete Laplacian and f is an $n \times 1$ vector, where n is the number of vertices in the mesh. However, as we discussed in sections 4.2 and 4.3, L is a rank-deficient matrix; in particular, the rank of L in this case is $n - 1$ for a connected mesh. A rank deficient linear system has no unique solution, since the matrix is not invertible. To remedy this, we must add linearly independent equations to L until it is full-rank. As we will see in Chapter 5, these extra equations inform *constraints* to the Laplacian system, and allow us to guide the harmonic functions with Dirichlet boundary conditions.

4.6 Final Remarks

In this Chapter, we have explored in detail the Laplacian operator defined over many different domains, including continuous Euclidean spaces, regular lattice grids, undirected graphs and finally triangle meshes. We derived some intuition regarding its geometrical interpretation, in terms of the "roughness" of a function, and also studied its spectral composition, concluding, among other things, that the eigenvectors of the Laplacian operator always define an orthogonal basis analogous to the Fourier domain on the real line, and that the multiplicity of the 0 eigenvalue equals the number of connected components in the domain. When studying discretizations of the Laplacian over triangle meshes, we observed that there are different possible formulations, sharing some properties with the continuous operator. We have also discussed the fact, due to Wardetzky et al. [55], that there exists no discrete version of the Laplace operator capable of satisfying all properties of the continuous case.

5 LAPLACIAN SURFACE MAPPING

In the previous two Chapters, we reviewed all the theoretical machinery necessary to describe our inter-surface mapping algorithm in detail. Because our method is loosely based on Least-Squares meshes [48], we will start this Chapter with a review of this technique.

5.1 Least-Squares Meshes

Least-Squares meshes are triangle meshes with a prescribed connectivity that approximate a set of user-defined geometric constraints. The connectivity information is extracted directly from an input mesh, and the geometric constraints are a set of control points also taken from the input mesh. Figure 5.1 shows examples of Least-Squares meshes constructed using different sets of constraints.

We are given an input mesh $\mathcal{M}$ with n vertices, and wish to compute a new mesh, $\mathcal{M}'$, such that both $\mathcal{M}$ and $\mathcal{M}'$ have identical connectivity. Moreover, we are also given a set of control vertices $\mathcal{M}_\mathcal{C} = \{c^0, c^1, \ldots, c^{k-1}\} \subset \mathcal{V}_\mathcal{M}$ whose spatial positions must be preserved in $\mathcal{M}'$. The Least-squares meshes method distributes all other vertices smoothly over the surface of $\mathcal{M}'$, by minimizing the Laplacian of the vertices' coordinates. Recall, from Chapter 4, that the Laplacian of a function behaves as a local measure of its "roughness". This immediately suggests that the geometric coordinates of the vertices in $\mathcal{M}'$ can be determined by solving the following linear systems:

$$Lx = 0, \qquad Ly = 0, \qquad Lz = 0, \tag{5.1}$$

where L is the Laplacian matrix for $\mathcal{M}$ (which can be defined in any of the forms present in Table 4.1), and x, y and z are all $n-$dimensional vectors that correspond to the x, y and z spatial coordinates of each vertex in $\mathcal{M}'$.

However, as we discussed in Chapter 4, the Laplacian matrix is rank-deficient, and therefore the linear systems in (5.1) are under-determined. In order to remedy this, we must add extra linearly independent rows to L, to raise its rank up to n. This is where the user-defined constraints are employed: we add k indicator rows to L, and to the right-hand-side vectors, to ensure that the vertices in $\mathcal{M}_\mathcal{C}$ respect their constraints. We assume, without loss of generality, that $\mathcal{M}_\mathcal{C}$ consists exactly of the first k vertices in $\mathcal{M}$. Therefore, the augmented linear system for the x coordinates

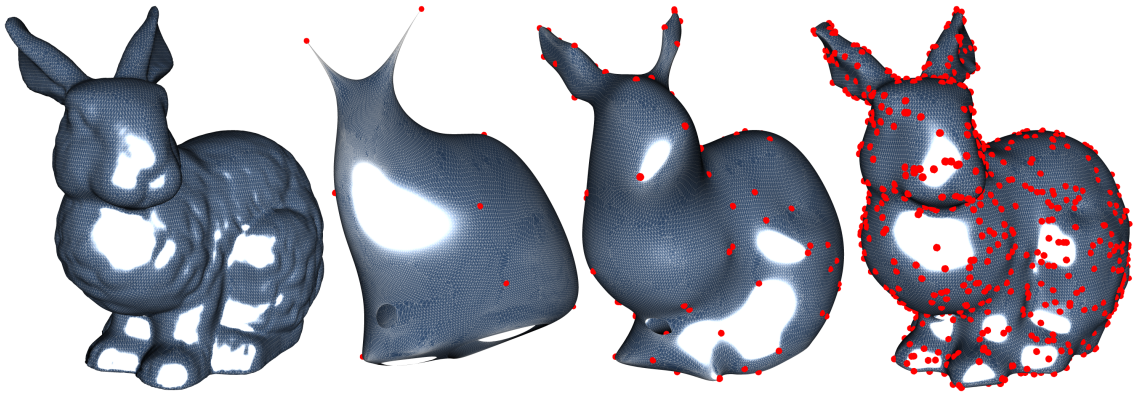


Figure 5.1: **Least-Squares Meshes.** Least-Squares Meshes approximate a set of geometric constraints with a prescribed connectivity. In this example, the geometry of the Stanford Bunny (extreme left) is reconstructed using 10 constraints (mid-left), 100 constraints (mid-right) and finally 1000 constraints (extreme right). Constrained vertices are drawn as red circles.

is described as (we denote this augmented matrix A):

$$Ax = \left(\begin{array}{ccccc} & & L & & \\ \hline \alpha & 0 & 0 & \dots & 0 \\ 0 & \alpha & 0 & \dots & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & \alpha & 0 \end{array}\right) x = \left(\begin{array}{c} 0 \\ \hline \alpha c_x^0 \\ \alpha c_x^1 \\ \vdots \\ \alpha c_x^{k-1} \end{array}\right) = b, \tag{5.2}$$

where α is a constraint satisfaction weight, and c_x^i are the x-coordinates of the vertices in $\mathcal{M}_\mathcal{C}$. We will focus our discussion on solving the equations only for the x-coordinates of the vertices, as the y- and z-coordinates can be solved in the exact same manner.

Intuitively, the augmented rows force the linear system to satisfy the identities $\alpha x^i = \alpha c_x^i$, in addition to satisfying $Lx = 0$. This is precisely our intention: to ensure that the vertices in $\mathcal{M}_\mathcal{C}$ have their geometric positions preserved. With this formulation, the solution vector x contains the x-coordinates for the vertices in the Least-Squares mesh $\mathcal{M}'$. The values in x not only ensure a fair distribution of vertices, as the solution is a minimum for the Laplacian operator, but also preserve the position of constrained points.

When $|\mathcal{M}_\mathcal{C}| > 1$, however, the augmented linear system becomes over-determined, and has in general no exact solution. We discussed this problem in detail in Chapter 3, and reviewed the method of linear least-squares, which provides the best possible approximation for an over-determined system, by solving the normal equations:

$$A^T Ax = A^T b.$$

For a full-rank matrix A, these equations define a symmetric, positive-definite linear system, which can be directly solved using Cholesky Decomposition. Once x, y and z are computed as described above, we build $\mathcal{M}'$ simply by using $\mathcal{M}$'s connectivity information and the newly computed geometry.

Because we solve Equation (5.2) in a least-squares sense, the solution does not satisfy the user-informed constraints *exactly*. Rather, it builds an intermediate

solution that approximates the constraints while still minimizing the Laplacian operator. The constraint satisfaction weights, introduced in (5.2), exist to control the balance between minimizing the Laplacian and satisfying the constraints. The higher their value, the more importance the constraints are given over minimizing the Laplacian, and vice-versa. In our implementation we adopted a value for α of 10^6, which causes the constraints to be satisfied exactly up to the numerical precision of the machine.

Least-Squares Meshes provide an effective way to approximate a set of geometric constraints with a prescribed connectivity. However, there is no inherent meaning in the geometric constraints provided. If, for instance, we build a least-squares mesh $\mathcal{M}'_1$ using geometric information from a *second* mesh $\mathcal{M}_2$, we can immediately construct an approximate correspondence between $\mathcal{M}_1$ and $\mathcal{M}_2$. This is precisely the method of Manifold Parameterization [56], which we describe below.

5.2 Manifold Parameterization

Similarly to Least-Squares Meshes, the Manifold Parameterization method builds a Least-Squares approximation to a mesh $\mathcal{M}_1$ using a set of geometric constraints. However, these constraints come from a *second* mesh $\mathcal{M}_2$. Therefore, the resulting mesh $\mathcal{M}'_1$ approximates the geometry of $\mathcal{M}_2$ using the connectivity of $\mathcal{M}_1$. Formally, this requires the specification of two sets of constraints $\mathcal{M}_{1C} \subset \mathcal{V}_{\mathcal{M}_1}$ and $\mathcal{M}_{2C} \subset \mathcal{V}_{\mathcal{M}_2}$. Moreover, there must be an explicit correspondence between the constraints defined over both meshes. These correspondences typically match vertices that share some geometric meaning: if the mapping being computed is between two faces, for instance, the correspondences must identify geometric features such as the tips of the noses, or the corners of the mouths and eyes, among others.

Given sufficiently dense sets of correspondences, computing the geometry of $\mathcal{M}'_1$ becomes a matter of solving (5.2), again in a least-squares sense, with the difference that $c_0, c_1, \ldots, c_{k-1}$ now come from the corresponding constraints in $\mathcal{M}_2$, instead of $\mathcal{M}_1$. As Figure 5.1 shows, Least-Squares Meshes require a fairly large number of vertices to accurately reconstruct the desired geometry. It is infeasible to require a user to specify such a large set of constraints. Rather, the original Manifold Parameterization method [56] computes a dense set of correspondences from sparse user input. In order to do this, the method computes an initial guess of the Least-Squares mapping using only the user-generated constraints. It then builds new correspondences by choosing local maxima of Mesh Saliency [27] in $\mathcal{M}_2$ and iteratively matching them with the closest points in $\mathcal{M}'_1$. Each match defines a new, slightly improved, version of $\mathcal{M}'_1$, whose final iteration is the result of the algorithm.

The original Manifold Parameterization method is inherently asymmetric, since it considers only the mapping of $\mathcal{M}_1$ into $\mathcal{M}_2$. Moreover, Mesh Saliency is a poor method to define new correspondences, especially for simpler geometric forms such as torii and ellipsoids (where the Saliency is mostly uniform). Our method addresses these issues: we introduce a novel correspondence sorting method that is fully symmetric, and use an intrinsic technique to find new points which does not depend on local properties such as Mesh Saliency.

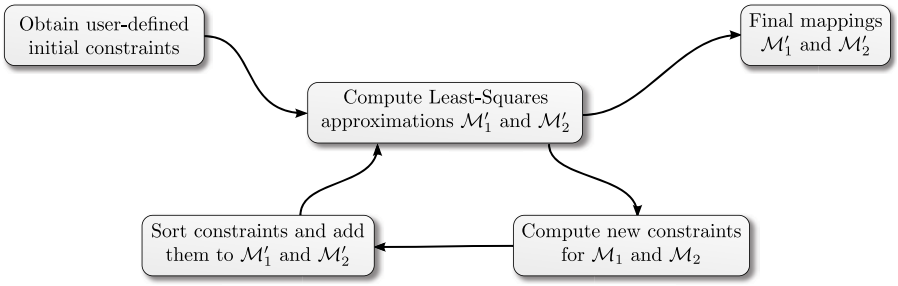


Figure 5.2: **Inter-surface Mapping Workflow.** Our inter-surface mapping algorithm is based on computing Least-Squares approximations $\mathcal{M}'_1$ and $\mathcal{M}'_2$ to two input meshes $\mathcal{M}_1$ and $\mathcal{M}_2$. To do this, we iteratively compute new constraint candidates based on an approximate Voronoi diagram scheme (bottom right), and add them to the mapping according to a symmetric sorting scheme (bottom left).

5.3 Inter-surface Mapping Algorithm

Our inter-surface mapping algorithm is similar to the Manifold Parameterization method: given two partially isometric meshes $\mathcal{M}_1$ and $\mathcal{M}_2$, we compute an approximate Least-squares mapping between the models. However, unlike Manifold Parameterization, our method computes both $\mathcal{M}'_1$ and $\mathcal{M}'_2$ (the approximate geometry of $\mathcal{M}_1$ using the connectivity of $\mathcal{M}_2$) simultaneously.

Our algorithm starts by requiring the user to inform a sparse set of correspondences between the two meshes. This is done using a simple point-and-click interface. Once the user has defined a few dozen correspondences, we compute initial Least-Squares approximations of $\mathcal{M}'_1$ and $\mathcal{M}'_2$. Our method then generates a new set of candidate constraints on both surfaces, using an intrinsic method that we describe in more detail in Section 5.3.1. We do not, however, create a correspondence for these points directly. Instead, we sort all candidate points according to a symmetric sorting scheme, explained in Section 5.3.2, which also determines a pairing vertex for each candidate. After the new constraints are paired and sorted, our algorithm iteratively adds these correspondences to the current mappings, slightly improving $\mathcal{M}'_1$ and $\mathcal{M}'_2$. This entire process is repeated until $\mathcal{M}'_1$ and $\mathcal{M}'_2$ are sufficiently close to $\mathcal{M}_2$ and $\mathcal{M}_1$, respectively (in our experiments, we set a fixed number of iterations to generate new constraints). Figure 5.2 shows a simplified workflow of our algorithm.

5.3.1 Intrinsic Point Enrichment

Given a set of constrained vertices in either $\mathcal{M}_1$ or $\mathcal{M}_2$, we wish to find new vertices to constrain in the next iteration of our algorithm. For simplicity, we will focus our discussion on $\mathcal{M}_1$ only; the process is entirely analogous for $\mathcal{M}_2$. Notice that in this stage of the algorithm we are not yet interested in finding a *correspondence* between vertices on both meshes — this will be done in the next step, during sorting. Instead, in this step we only wish to find new vertices over a single mesh that uniformly cover its geometry.

To choose these new vertices, we build an approximate Voronoi diagram over the surface of the mesh. A Voronoi diagram is a decomposition of a metric space into a disjoint set of *cells*, each of which corresponds to a given *site* [9]. These cells satisfy the property that all points inside a particular cell are closer to its corresponding

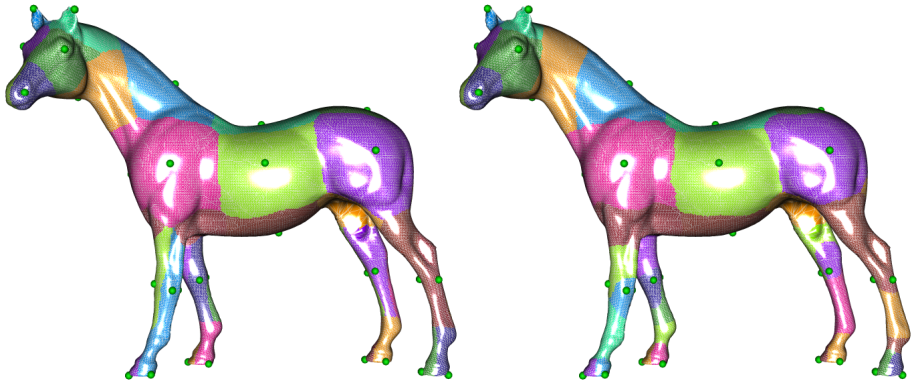


Figure 5.3: **Approximate Voronoi Diagrams.** We implemented two approximate Voronoi diagram schemes, one based on Dijkstra's algorithm (left) and one based on solutions to Laplace's equation (right). We use the nodes in the Voronoi diagram as new constraints in our inter-surface mapping algorithm. The green dots shown in this picture are the sites for the diagrams.

site than to any other site. Edges between cells are sets of vertices equally distant from two sites. Finally, points where more than two cells meet are called *Voronoi nodes*, which are equally distant from all the sites corresponding to the cells that meet at that node. We compute an approximate Voronoi Diagram over $\mathcal{M}_1$, whose sites are the constrained vertices from the previous iteration of our algorithm. Once this is done, we simply use the nodes in the diagram as new constrained vertices for the next iteration.

The construction of the Voronoi diagram requires a notion of distance between points in the mesh and the sites. However, we cannot simply compute the Euclidean distance in $\mathbb{R}^3$, since it does not consider folds and bends in the model. To compute an exact Voronoi diagram it would be necessary to determine the Geodesic distance between pairs of points on the surface of the mesh (the Geodesic distance between two vertices u and v is the length of the shortest continuous path that goes from u to v). In our algorithm, however, we consider only *approximate* geodesic distances, computed using one of the two methods explained below.

The first method is simply Dijkstra's algorithm over the underlying graph of the mesh. We use the Euclidean length of the edges as their weights, and execute the algorithm once for every site. This generates a set of approximate distance fields, where the distance between an arbitrary vertex v and a site c_i is approximated by the length of the shortest edge-path between v and c_i. The second method, on the other hand, involves solving Laplace's equation over the mesh, with a set of least-squares boundary conditions, exactly as explained in Section 5.1. To compute the distance field corresponding to a site c_i, we solve $Lx = 0$ with augmented rows that enforce the following boundary conditions:

$$x_j = \begin{cases} 0 & c_i = v_j \\ 1 & otherwise, \end{cases} \qquad j = 0, \ldots, k-1$$

where v_j is the vertex in the mesh corresponding to x_j. The vector x therefore describes the smoothest possible function (since it minimizes the Laplacian) that is zero in c_i and one in every other site. Although this is not an exact distance field, it is an acceptable approximation to compute the Voronoi diagram. Figure 5.3 illustrates

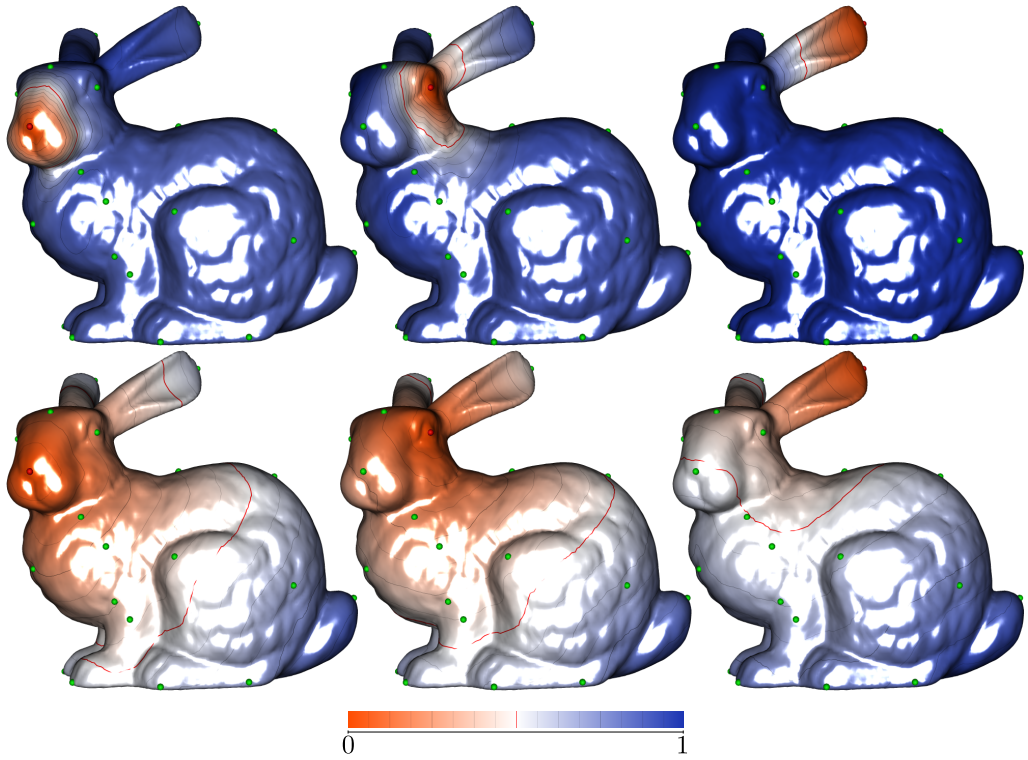
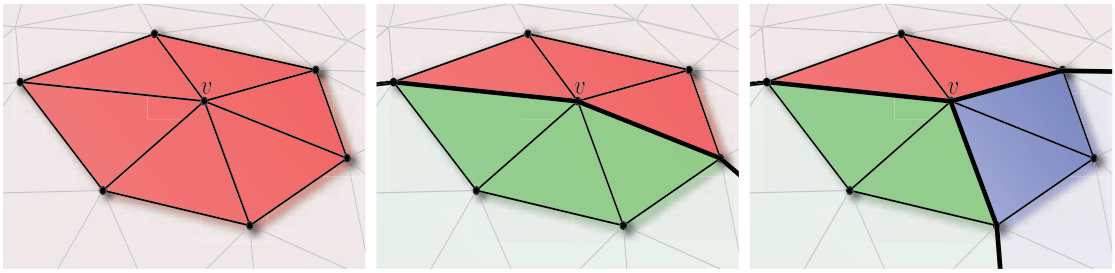


Figure 5.4: **Approximate Distance Fields.** We compute approximate distance fields based on a set of constrained vertices. The top row shows distance fields computed using the solution to Laplace's equation, and the bottom row shows fields built with Dijkstra's algorithm. We show results for three different sites c_i, which are represented as red points. All other constraints are drawn as green points.

the diagrams output by our method. Our process is called *intrinsic* because it considers only intrinsic surface properties to generate the Voronoi diagrams, i.e., properties that are inherent to the surface itself, irrespective of its embedding in $\mathbb{R}^3$. This is particularly true for the distance maps based on Laplace's equation, as the (Combinatorial) Laplacian matrix depends only on the connectivity of the mesh.

Both methods described above generate a set of approximate distance fields, each corresponding to one site. These distance fields are illustrated in Figure 5.4. The distance fields allow us to compute the Voronoi diagram simply by classifying each triangle in the mesh as belonging to one of the Voronoi cells. To do this, we find the distance field that has smallest value in the barycenter of the triangle, and add this triangle to the field's corresponding cell. We use linear interpolation to compute the value of the fields in the triangles' barycenters. Once all triangles have been classified, the construction of the Voronoi diagram is complete. We must then find the nodes in the diagram, to set them as new constrained vertices.

All vertices in the mesh can be classified into one of three categories, depending on their neighboring faces: if all neighboring faces of a vertex belong to the same Voronoi cell, we call this vertex an *internal* vertex. If the neighboring faces belong to exactly two cells, we call the vertex an *edge* vertex, as it lies exactly on an edge between two Voronoi cells. Finally, if the neighboring faces belong to more than two cells, this vertex corresponds exactly to a node in the Voronoi diagram. This classification, illustrated in Figure 5.5, suggests an immediate algorithm to find the



(a) Internal vertex. (b) Edge vertex. (c) Voronoi node vertex.

Figure 5.5: **Vertex Classification.** We classify a vertex v as being *internal* (a), an *edge vertex* (b) or a *node* (c). This classification is defined entirely using v's neighboring triangles. We then use the node vertices as new constraints for the surface mapping.

nodes: we traverse all vertices, and choose only those whose neighboring faces belong to more than two different Voronoi cells.

Once this vertex classification is complete, we have a new set of vertices for each mesh $\mathcal{M}_1$ and $\mathcal{M}_2$, which we use as new constraints in our inter-surface mapping. However, the new vertices in $\mathcal{M}_1$ do not yet correspond to any vertex in $\mathcal{M}_2$, and vice-versa. The next step in our algorithm determines this correspondence.

5.3.2 Symmetric Constraint Sorting

After computing and classifying the Voronoi diagrams, our algorithm computes two sets of new constraint vertices, which we call $\mathcal{NC}_1$ and $\mathcal{NC}_2$, corresponding to new constraints in $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. For each vertex in $\mathcal{NC}_1$ we must find a corresponding vertex in $\mathcal{M}_2$, and vice-versa. We do this via a symmetrized version of the Manifold Parameterization scheme.

We initially find, for each vertex v_i in $\mathcal{NC}_1$, its corresponding vertex v_i' in $\mathcal{M}'_1$. The vertex v_i' is therefore the position of the new constraint in the current Least-Squares approximation of $\mathcal{M}_2$. We then iterate over all vertices in $\mathcal{M}_2$ and find the one that is closest to v_i'. We store this vertex pair, along with their distance, for each v_i in $\mathcal{NC}_1$. We apply this process to $\mathcal{NC}_2$ analogously. Because of this, we call our approach *symmetric*: we do not give any precedence to $\mathcal{M}_1$ or $\mathcal{M}_2$ in the process, since we compute both pairings for $\mathcal{M}'_1$ and $\mathcal{M}'_2$ simultaneously.

Once we have candidate pairs for all vertices in $\mathcal{NC}_1$ and $\mathcal{NC}_2$, we choose the pair with overall smallest distance and add it to $\mathcal{M}'_1$ and $\mathcal{M}'_2$ (by adding a new augmented row to the Laplacian matrices). We remove this vertex pair from its originating set, recompute all pairs and distances, and repeat this process until both $\mathcal{NC}_1$ and $\mathcal{NC}_2$ are empty.

The process of building approximate Voronoi diagrams and iteratively adding new constraints completes one iteration of our algorithm, which increases the number of Least-Squares constraints in both $\mathcal{M}'_1$ and $\mathcal{M}'_2$ by $|\mathcal{NC}_1| + |\mathcal{NC}_2|$. We repeat this process until $\mathcal{M}'_1$ is sufficiently close to $\mathcal{M}_2$ and vice-versa. In our experiments, we required no more than three iterations to ensure a good approximate inter-surface mapping.

The algorithm we just described builds a geometric approximation of $\mathcal{M}_2$ using the connectivity of $\mathcal{M}_1$ and vice-versa. This is enough information for applications that require only a geometric mapping, such as computing morphing sequences.

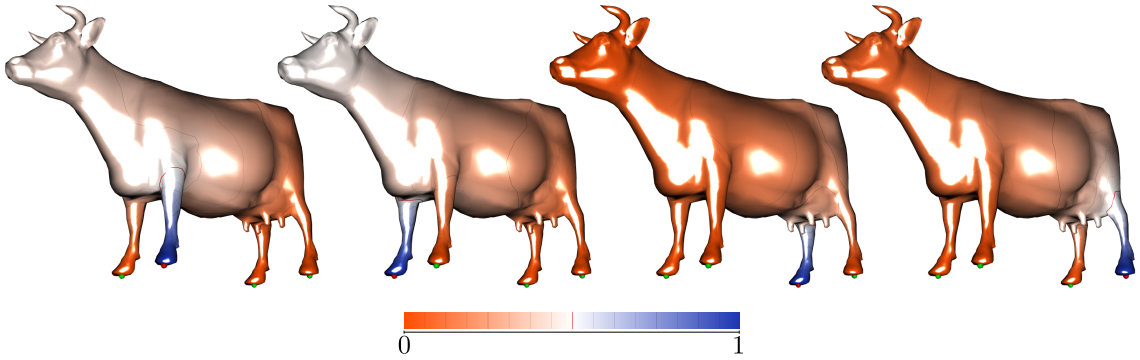


Figure 5.6: **Least-Squares Basis.** We build a vector basis consisting of k linearly independent vectors to represent functions over a mesh. In this Figure, vertex constraints are highlighted in green, and the selected constraint c_i is shown in red (in this example, $k = 4$). Notice how the values of these basis vectors are approximately reversed with respect to Figure 5.4 (a consequence of the inverted boundary values).

However, if we wish to transfer information between the surfaces of the two models, such as colors or texture coordinates, we require extra machinery. We describe our proposal to perform detail transfer in the next Section.

5.3.3 Information Transfer between $\mathcal{M}_1$ and $\mathcal{M}_2$

We consider the problem of transferring real-valued functions between $\mathcal{M}_1$ and $\mathcal{M}_2$. We deal only with mapping functions in one direction; the reverse is solved analogously. Given a function $f : \mathcal{M}_1 \rightarrow \mathbb{R}$, we wish to find a new function $g : \mathcal{M}_2 \rightarrow \mathbb{R}$ such that the values of f and g are the same for matched points. We restrict ourselves to functions defined over the vertices of each mesh, which can therefore be represented as column-vectors.

Recall that once we have finished computing $\mathcal{M}_1'$ and $\mathcal{M}_2'$, we also have dense sets of correspondences $\mathcal{M}_{1C}$ and $\mathcal{M}_{2C}$, with k vertices each. We use these correspondence points to build two bases for k-dimensional linear spaces embedded in $\mathbb{R}^{n_1}$ and $\mathbb{R}^{n_2}$ (one for each mesh), where n_1 and n_2 are the number of vertices in $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. To build the basis for $\mathcal{M}_1$ (which we denote $B_{\mathcal{M}_1}$), for example, we compute a set of k n_1-dimensional vectors using a method similar to what we described in Section 5.3.1. We solve Laplace's equation $L{b_{\mathcal{M}_1}}^i = 0$ for each ${b_{\mathcal{M}_1}}^i$ with boundary conditions set thus:

$$ {b_{\mathcal{M}_1}}^i_j = \begin{cases} 1 & c_i = v_j \\ 0 & otherwise, \end{cases} \qquad j = 0, \ldots, k-1 $$

where v_j is the vertex in the mesh corresponding to ${b_{\mathcal{M}_1}}^i_j$. Instead of a distance field, these vectors are similar to hat functions with a peak on v_i (for the i_{th} constraint). Figure 5.6 shows an example of a basis built this way. We build $B_{\mathcal{M}_2}$ for $\mathcal{M}_2$ using the exact same method.

Now, given a function $f : \mathcal{M}_1 \rightarrow \mathbb{R}$ we wish to find coefficients $x = [x_0, x_1, \ldots, x_{k-1}]^T$ that represent f as a linear combination of the vectors in $B_{\mathcal{M}_1}$. Recall, from Chapter 3, that this is equivalent to solving the following linear system:

$$ B_{\mathcal{M}_1} x = f \tag{5.3} $$

In general, however, $k < n_1$, so the system above is over-determined (meaning $B_{\mathcal{M}_1}$ does not span a space capable of exactly representing f). Again we address this issue by solving (5.3) in a Least-squares sense:

$$B_{\mathcal{M}_1}{}^T B_{\mathcal{M}_1} x = B_{\mathcal{M}_1}{}^T f$$

Because $B_{\mathcal{M}_1}$ is a full-rank matrix, we can also solve this system using Cholesky Decomposition. The solution vector x represents the coefficients that best approximate f using b, in a least-squares sense.

We can now build $g : \mathcal{M}_2 \to \mathbb{R}$ by considering $B_{\mathcal{M}_2}$, the basis for functions in $\mathcal{M}_2$. Because there is an inherent one-to-one correspondence between vectors in $B_{\mathcal{M}_1}$ and $B_{\mathcal{M}_2}$ (since they are built with matched correspondences), we use the coefficients x computed for $B_{\mathcal{M}_1}$ to build g in $B_{\mathcal{M}_2}$ with a simple matrix-vector multiplication:

$$g = B_{\mathcal{M}_2} x$$

With this method we reduce the problem of transferring real-valued functions between two meshes to two linear-algebraic computations: a linear system and a matrix-vector multiplication. In order to transfer multi-dimensional data such as colors, normals or texture coordinates, we simply transfer each dimension of the data separately.

5.4 Final Remarks

In this Chapter we introduced and explained in detail our inter-surface mapping algorithm. Our technique is based on iteratively building two Least-Squares meshes $\mathcal{M}_1'$ and $\mathcal{M}_2'$, using a set of vertex-to-vertex correspondences between two input meshes $\mathcal{M}_1$ and $\mathcal{M}_2$. In order to generate a dense set of constraints, we employ an approximate Voronoi diagram-based method to generate new candidate constraints, and a symmetrized point sorting approach that alternates between adding constraints from $\mathcal{M}_1$ and $\mathcal{M}_2$.

In the next Chapter, we will present experimental results and discuss them in detail. We will describe all the datasets used, and also discuss issues such as running time and memory consumption of our algorithm.

6 RESULTS AND DISCUSSION

In this Chapter, we discuss the experimental results obtained using our inter-surface mapping algorithm. We describe all the models we used, as well as the characteristics of our software and test system. We present results that demonstrate mesh morphing, both between simple and complex shapes, and also illustrate detail transfer via a color transfer example. We begin this discussion with a description of our software and experimental configuration.

6.1 Software and Experimental Testbed

We have implemented our software, `LSMapper`, entirely in C++, using Nokia's Qt library for the interface and OpenGL 3.0 for drawing. We have also implemented GLSL fragment shaders to generate high quality renderings of our results. The full code is free software, distributed under the GNU Foundation's General Public License v3.0 [3], and is available for download from `http://www.inf.ufrgs.br/~lfscheidegger/lsmapper`. In addition to computing the surface maps, our code also contains functionality to compute Least-Squares Meshes [48], as well as compute and display the approximate Voronoi diagrams described in Chapter 5. We conducted all experiments described here on an Intel® Core2™ Duo E8400@3.00Ghz, with 4Gb of RAM memory, and an NVidia® GeForce® 9800GT graphics card, running the Ubuntu 10.04 Lucid Lynx Linux distribution, and GNU's gcc compiler, version 4.4.3, with the `-O3` full optimization flag set.

Interacting with our system is very simple: the user initially loads a pair of triangle meshes $\mathcal{M}_1$ and $\mathcal{M}_2$, which are displayed on the upper left and upper right viewports, respectively. Once this is done, the user can start choosing the initial correspondence points. To do this, it is necessary to double-click somewhere over one of the meshes, thus selecting one vertex for the correspondence, and then to double-click on the corresponding vertex on the other mesh. The process of picking the correspondences automatically generates the augmented rows for the Laplacian matrices, which are initialized when the user loads the meshes. Once the number of matched pairs is satisfactory, the user clicks on "Bake Mapping". This computes one iteration of our algorithm, adding a large number of automatically generated correspondences. The lower left and lower right viewports display the most up-to-date versions of the Least-Squares approximations $\mathcal{M}_1'$ and $\mathcal{M}_2'$. The user can repeat this process arbitrarily, to generate denser correspondences. It is also possible to manually add more correspondences to the mapping at any time. Figure 6.1 illustrates our system's Graphical User Interface.

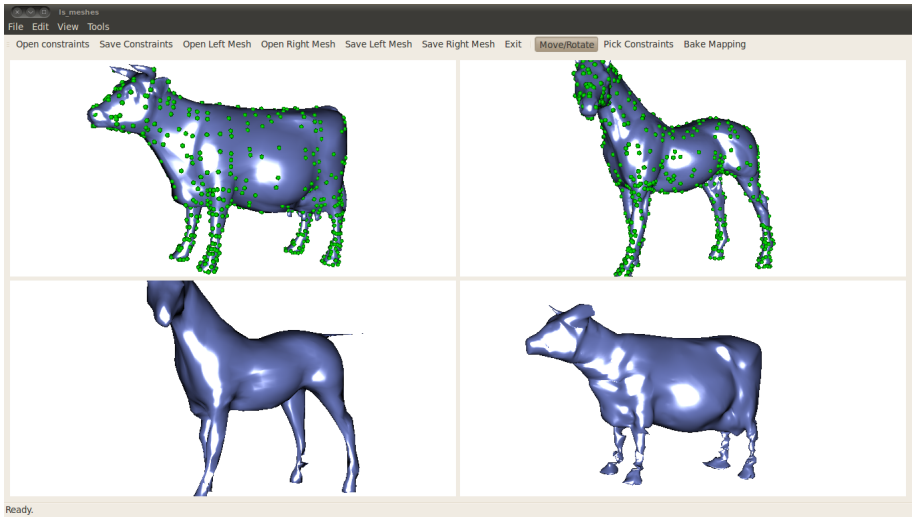


Figure 6.1: **LS Mapper.** Our software interactively computes Least-Squares approximations to two input meshes, and displays the results graphically. The top left viewport displays $\mathcal{M}_1$ and the top right displays $\mathcal{M}_2$. The bottom viewports show $\mathcal{M}_1'$ and $\mathcal{M}_2'$, respectively.

6.2 Experimental Results

We have conducted experiments on many different triangle meshes. Table 6.1 shows a summary of the relevant information for each model, including their number of vertices and faces. Although we did not use the Stanford Bunny dataset in any experiment, we include it in this Table because it appears as an illustrative example for many concepts in previous Chapters of this text. The Stanford Bunny and Armadillo models were obtained from the Stanford 3D Scanning Repository [2], whereas all other models (including the Deformed Armadillo) were obtained via the Aim@Shape project [1].

Recall from Chapter 5 that our algorithm depends on a choice of approximate distance map. In our implementation, we have used the formulation based on solving a set of sparse Laplacian equations, instead of using Dijkstra's algorithm. Although Dijkstra's algorithm provides a better approximation, we have observed that for situations where the number of constraints is more than a few dozen both methods generate very similar results, and the Laplacian formulation is much faster to compute. Moreover, our technique also requires an appropriate choice of Laplacian matrix (refer to Table 4.1 for a full list of possibilities). For simplicity and stability, our current implementation uses the Combinatorial Laplacian.

Table 6.2 contains symbolic names for the mapping experiments we conducted. These names should be used when referring to Table 6.3, which contains relevant information about all mappings. The largest triangle meshes with which we tested our method contain more than 300.000 triangles and over 150.000 vertices. Despite this large size, `LSMapper` never required more than 30 minutes to obtain a result. Figure 6.2 illustrates the output of all the experiments. The extreme left and extreme right columns depict $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively (constrained vertices are shown in green). The center left and center right columns illustrate $\mathcal{M}_2'$ and $\mathcal{M}_1'$, in this order. Observe how $\mathcal{M}_2'$ resembles $\mathcal{M}_1$ and how $\mathcal{M}_1'$ resembles $\mathcal{M}_2$. We color-map these models according to the geometric error of the approximation, normalized

Figure 6.2: **Surface Mapping Results.** We have conducted four different mapping experiments, with varying models. The extreme left and extreme right columns illustrate the input meshes $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. The center columns depict $\mathcal{M}_2'$ and $\mathcal{M}_1'$, in this order. Observe how $\mathcal{M}_2'$ approximates the geometry of $\mathcal{M}_1$ and how $\mathcal{M}_1'$ resembles $\mathcal{M}_2$. The colormap illustrates the geometric error of the approximations, normalized by the diagonal of each model's bounding box. Also, the final constraints are visible as green dots on $\mathcal{M}_1$ and $\mathcal{M}_2$.

Example Datasets		
Name	Vertices	Faces
Stanford Bunny	34834	69451
Cow	2903	5804
Horse	48485	96966
Large Torus	1000	2000
Small Torus	1000	2000
Armadillo	172974	345944
Deformed Armadillo	165954	331904
Left Hand	53054	105860
Right Hand	112729	225152

Table 6.1: **Summary of Example Datasets.**

Experiment Keys		
Experiment	$\mathcal{M}_1$	$\mathcal{M}_2$
E1	Cow	Horse
E2	Large Torus	Small Torus
E3	Armadillo	Deformed Armadillo
E4	Left Hand	Right Hand

Table 6.2: **Experiment Keys.** This Table contains symbolic names for the experiments reported in Table 6.3.

by the diagonal of the model's bounding box. Notice how none of the models incur an approximation error of more than 10%. In fact, as is evident by the models' predominantly orange colors, geometric error is very close to zero almost everywhere. Furthermore, the Laplacian solver distributes the vertices evenly across the surface, ensuring high triangle quality in $\mathcal{M}_1'$ and $\mathcal{M}_2'$, with respect to the popular incenter/circumcenter radii ratio.

Another very important aspect of our method is its ability to automatically handle triangle meshes with incompatible topology. To the best of our knowledge, only `LSMapper` and the Manifold Parameterization technique are able to do this. Other algorithms explicitly assume that the input meshes are topologically equivalent. Although it does not usually make sense to map meshes with different topologies,

Experimental Data				
Experiment	Initial Const.	Total Const.	Cholesky Dec.	$\mathcal{M}_1'$ and $\mathcal{M}_2'$
E1	52	807	0.03s/1.01s	3m 1.81s
E2	16	245	0.01s/0.01s	1.92s
E3	41	909	1.20s/2.93s	8m 40.01s
E4	53	1297	5.67s/4.37s	29m 29.34s

Table 6.3: **Experimental Results.** This Table presents the experimental results of our method. We report the number of user-specified constraints, the number of final constraints in the mapping, the time taken to compute the Cholesky factorization of the Laplacian matrices for $\mathcal{M}_1$ and $\mathcal{M}_2$ and the total time to compute $\mathcal{M}_1'$ and $\mathcal{M}_2'$. Refer to Table 6.2 for the datasets used in each experiment.

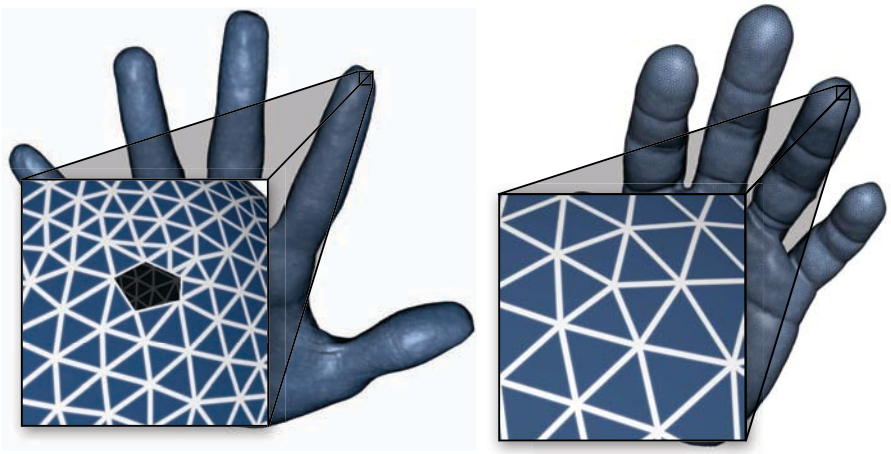


Figure 6.3: **Mapping Meshes with Different Topologies.** Our method builds a surface map even between meshes with different topologies. The Right Hand dataset has a hole at the tip of the index finger. Even though the Left Hand does not have the same hole (and hence the same topology), our algorithm can compute a compatible surface map between the models.

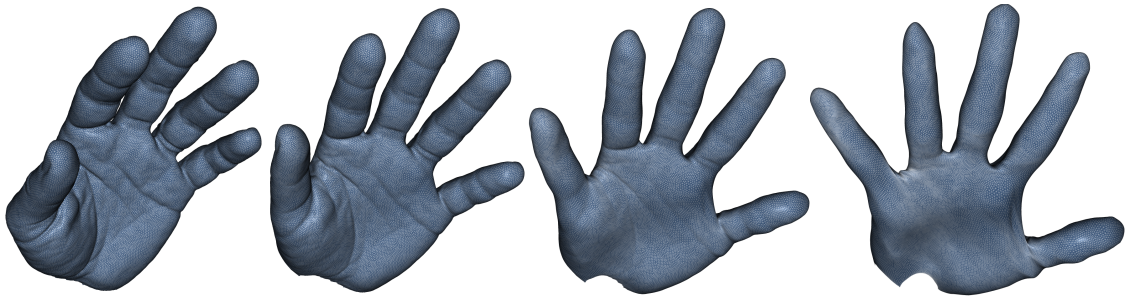


Figure 6.4: **Morphing Sequence.** Once $\mathcal{M}_1'$ is computed, we trivially compute a morph sequence between $\mathcal{M}_1$ and $\mathcal{M}_1'$. This Figure illustrates $\mathcal{M}_1$, $1/3\mathcal{M}_1 + 2/3\mathcal{M}_1'$, $2/3\mathcal{M}_1 + 1/3\mathcal{M}_1'$ and $\mathcal{M}_1'$ respectively. The boundary next to the wrist gets distorted because there are no constraints there, and the Laplacian minimizer does not respect boundaries exactly.

sometimes defects in the models, such as holes and cracks, may invalidate the topology of otherwise perfectly compatible surfaces. This is exactly the case in Experiment **E4**: a small hole on tip of the Right Hand's index finger went almost entirely unnoticed by the authors (see Figure 6.3). Although this small defect should not preclude a mapping between the models, it is sufficient to make them topologically distinct, and thus render most previous techniques unusable. Our method entirely disregards this problem, and produces the results shown in Figure 6.2.

6.3 Discussion

6.3.1 Applications

One of the main applications of our technique is to compute mesh morphing sequences. Once we have, for instance, the Least-Squares approximation $\mathcal{M}_1'$, it is possible to compute a continuous deformation sequence that smoothly takes

Figure 6.5: **Least-Squares Detail Transfer.** We build a pair of Least-Squares bases on $\mathcal{M}_1$ and $\mathcal{M}_2$ which allows detail transfer between both meshes. In this example, we map a circular color pattern directly from $\mathcal{M}_1$ to $\mathcal{M}_2$, without the need for an intermediate, parameterized, domain. However, our method does not exactly capture high frequency components, such as the sharp edges in the pattern in this Figure.

the geometry of $\mathcal{M}_1$ into $\mathcal{M}_1'$. Since there is an immediate one-to-one mapping between vertices in $\mathcal{M}_1$ and $\mathcal{M}_1'$ (recall that they have, by construction, identical connectivity), we compute this morphing sequence using a linear interpolation of the vertices' coordinates. Figure 6.4 illustrates this for Experiment **E4**.

Recall, from Chapter 5, that our algorithm not only computes the Least-Squares approximations $\mathcal{M}_1'$ and $\mathcal{M}_2'$, but also maps real-valued functions between $\mathcal{M}_1$ and $\mathcal{M}_2$. It is possible to transfer, e.g., the colors of one model directly onto another, with different geometry *and* different connectivity. We illustrate this construction in Figure 6.5. Our detail transfer method differs from most previous work in the key sense that we do not require an intermediate domain to compute the transfer — it is done directly from $\mathcal{M}_1$ to $\mathcal{M}_2$ via a simple change of basis.

However, one main drawback of our detail transfer algorithm is its inability to accurately reconstruct high frequencies present in the original function. This happens because we use a sparse Least-Squares approximation of f, and high frequencies are lost during the projection. This problem can be addressed by computing more iterations of the main workflow, thus adding more constraints to both meshes. We have specifically chosen a signal with strong high frequency components in Figure 6.5 to illustrate this limitation.

6.3.2 Algorithm Complexity

Our method is designed to be separable into a number of iterations of the main method described in Chapter 5, which can be repeated to improve the quality of the results. Because of this, we will mainly discuss the complexity of the steps involved in each iteration.

We use the computational library `CHOLMOD` for all linear-algebraic operations in the implementation. This library allows us to compute the Cholesky decompositions of the Laplacian matrices for each mesh only once, and keep them in memory. This

computation takes $O({n_{nz}}^3)$ for each matrix. However, since this is done only once, it is not the most expensive part of our algorithm. Table 6.3 reports the time taken, in seconds, to compute the Cholesky decompositions. Once the factorizations are computed, it is possible to solve linear systems in $O(n_{nz})$.

Each new correspondence must modify the Laplacian matrices, adding indicator rows. However, we do not recompute the factorization every time, since `CHOLMOD` contains functionality to update the LL^T factorization in $O(1)$, when this update is restricted to adding or removing rows and columns from the original matrix. This is exactly our case, as each new constraint adds a single indicator row to the matrix.

Computing the approximate distance maps is a more costly operation. Depending on the choice of method, this can be done in either $O(nc)$ (using the Laplacian equations) or $O(m \log n)$ (using Dijkstra's algorithm), where c is the current number of constraints and m is the number of edges in the mesh. Finally, the symmetrized constraint sorting is the most expensive part of our algorithm. With a KD-Tree to optimize Nearest-Neighbor searches, each query takes $O(\log n)$ time for points uniformly distributed in space [35]. Therefore, constraint sorting runs in $O(n[c \log c + c \log n])$. Each step must sort the set of new correspondences, done in $c \log c$, and find nearest neighbors for all constraints, done in $c \log n$. Observe that, although we sort the constraints only to pick the one with minimum distance, a Heap would not improve the complexity of our algorithm. This happens because when adding a new constraint we must update the distances of *all* correspondences still in the queue. This is equivalent to reconstructing the entire Heap at every extraction operation, thus nullifying any theoretical performance improvement.

In the next Section, we discuss some limitations of our current method. In particular, we deal with the fact that we do not optimize constraint placement in any way, and also mention possibilities to turn our correspondence selection technique into a fully intrinsic method.

6.3.3 Limitations and Future Work

As we have mentioned in Chapter 5, our constraint selection method, especially the formulation based on solutions to Laplace's equation, is intrinsic to the model's topology: we do not use geometry information such as Mesh Saliency or curvature to choose new point constraints. However, when determining the *pairs* between correspondences, we use the Euclidean distance between vertices on both meshes. In effect, this means that our method is only partially intrinsic. The main advantage of a *fully* intrinsic method would be its ability to compute a mapping between two models that are homeomorphic, but not isotopic. Homeomorphic models are just topologically equivalent (i.e., there is some mapping between them which preserves neighborhoods). Isotopy, on the other hand, is a stronger requirement: two models are isotopic if and only if there exists a continuous deformation of one model into the other without any self-intersections. The Trefoil Knot (Figure 6.6) is a classic example of a model that is homeomorphic, but not isotopic, to the Torus.

We are currently investigating ways with which to turn our technique into a fully intrinsic method. One possibility we are considering is to maintain the Voronoi Cells adjacent to the new constraints on each mesh. If, for instance, vertex v on $\mathcal{M}_1$ touches cells c_0, c_1 and c_2, and vertex u on $\mathcal{M}_2$ touches these same cells (albeit on the other mesh), we can immediately place u and v into correspondence. Our preliminary experiments with this method have shown that merely defining this

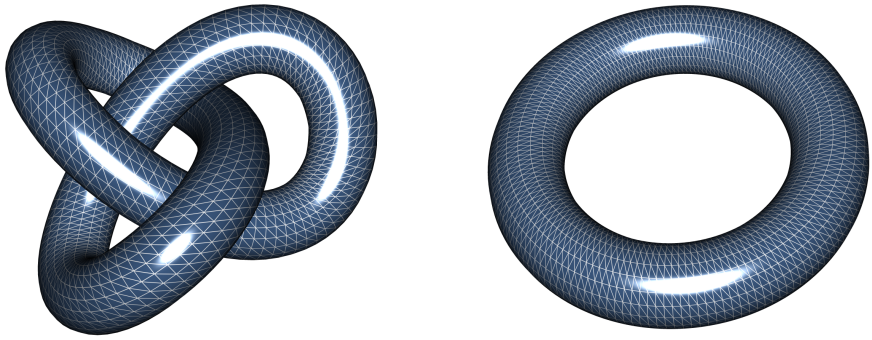


Figure 6.6: **Homeomorphism vs. Isotopy.** A fully intrinsic surface mapping method should be capable of mapping homeomorphic models that are not isotopic, such as the Trefoil Knot (left) and the Torus (right). Even though there is no morphing sequence without self-intersections, it should still be possible to map functions between the models, as they are topologically equivalent. Our method cannot currently do this.

correspondence generates poor mappings. In order to remedy this, it might be possible to define a metric that measures the quality of a mapping. With this metric, we may be able to optimize the correspondence, by possibly replacing u or v with one of their immediate neighbors, and iterating this process while the metric is improving. In fact, it might even be possible to define correspondences that are not strictly point-to-point: instead of merely adding an indicator row to the matrix, we add a row with three non-zero entries α_0, α_1 and α_2 (for the three vertices of an arbitrary triangle), with the property that $\alpha_0 + \alpha_1 + \alpha_2 = \alpha$ (recall that α is the constraint satisfaction weight). In effect, this construction allows correspondences to be enforced between points directly over the triangles on a mesh, instead of just on its vertices. However, this discussion is highly speculative, and we hope to arrive at more definite results as we conduct further experiments.

7 CONCLUSION

We have presented a novel inter-surface mapping algorithm that computes a Least-Squares approximation of a mesh $\mathcal{M}_2$ using another mesh $\mathcal{M}_1$'s connectivity, and vice-versa. Our method performs an order of magnitude faster than existing techniques, and is very robust with respect to imperfections in the input data. In particular, our method is unaffected when mapping models with different numbers of elements (vertices and faces), as well as models with poor triangle quality. We derive the robustness of our method from a stable formulation of the discrete Laplacian operator over triangle meshes, which is guaranteed to be always symmetric positive-definite. This way, the solutions to linear systems of the form $Lx = 0$, which pervade our method, always exist and can be efficiently computed using Cholesky decomposition.

In this text, we have reviewed the mathematical background necessary to develop our method, including detailed discussions of many linear-algebraic concepts, as well as a thorough presentation on the Laplacian operator. We introduce `LSMapper` as an extension and improvement over the Manifold Parameterization method, which is itself loosely based on the technique of Least-Squares Meshes. The main contributions of our novel technique are a semi-intrinsic constraint selection algorithm and a symmetric constraint sorting step. These contributions allow us to build the mapping between $\mathcal{M}_1$ and $\mathcal{M}_2$ in both directions simultaneously.

There are many interesting avenues for future work to further improve the technique. As we discussed in Chapter 6, our constraint selection method is still not fully intrinsic. This means that we cannot yet build a consistent mapping between two homeomorphic models that are not isotopic (such as the example of the Trefoil Knot and the Torus). In order to do this, we require a constraint selection and pairing method that does not consider the Euclidean distance between points on both meshes to determine the vertex pairs. Moreover, `LSMapper` determines constraint pairs only once, and makes no subsequent optimization of these pairs. This can lead to a poor selection of constraints, especially when the two meshes being mapped are very different from one another. An interesting approach to remedy this problem is to define some metric that can measure the quality of a current mapping, and try to modify the constraint pairs to optimize this metric. Finally, one issue that is left open in our method is how to determine the initial sparse set of mesh correspondences: we currently rely on user input for this information. However, the recent work of Lipman and Funkhouser [29] shows that it is possible to automatically find a set of point correspondences between partially isometric models. In the future, we plan to seamlessly integrate their technique into `LSMapper`, rendering our surface mapping workflow fully automatic.

REFERENCES

[1] The Aim@Shape Project. See `http://shapes.aim-at-shape.net/`.

[2] The Stanford 3D Scanning Repository. See `http://www.graphics.stanford.edu/data/3Dscanrep/`.

[3] The GNU General Public License, v3.0, 2007. See `http://www.gnu.org/licenses/gpl.html`.

[4] Marc Alexa. Recent Advances in Mesh Morphing. *Computer Graphics Forum*, 21:173 – 198, 2002.

[5] Howard Anton. *Elementary Linear Algebra*. John Wiley & Sons, Inc., 2000.

[6] Stephen Barnett. *Matrices: Methods and Applications*. Clarendon Press, 1990.

[7] Fan R. K. Chung and S. t. Yau. Eigenvalues of graphs and Sobolev inequalities. *Combinatorics, Probability & Computing*, 4:11–25, 1995.

[8] Timothy A. Davis. User Guide for CHOLMOD: a sparse Cholesky factorization and modification package, 2008.

[9] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, April 2008.

[10] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary meshes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 173–182, 1995.

[11] Michael S. Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14:231–250, April 1997.

[12] Michael S. Floater. One-to-one piecewise linear mappings over triangulations. *Mathematics of Computation*, 72:685–696, April 2003.

[13] Michael S. Floater and Kai Hormann. Surface parameterization: a tutorial and survey. In *Advances in Multiresolution for Geometric Modelling*, pages 157–186. Springer, 2005.

[14] Michael S. Floater and Martin Reimers. Meshless Parameterization and Surface Reconstruction. *Computer Aided Geometric Design*, 18:77–92, March 2001.

[15] Craig Gotsman, Xianfeng Gu, and Alla Sheffer. Fundamentals of spherical parameterization for 3D meshes. In *Proceedings of the 30th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '03, pages 358–363, 2003.

[16] Igor Guskov, Kiril Vidimče, Wim Sweldens, and Peter Schröder. Normal Meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '2000, pages 95–102, 2000.

[17] Steven Haker, Sigurd Angenent, Allen Tannenbaum, Ron Kikinis, Guillermo Sapiro, and Michael Halle. Conformal Surface Parameterization for Texture Mapping. *IEEE Transactions on Visualization and Computer Graphics*, 6:181–189, April 2000.

[18] Klaus Hildebrandt, Konrad Polthier, and Max Wardetzky. On the convergence of metric and geometric properties of polyhedral surfaces. *Geometriae Dedicata*, 123:89–112, 2005.

[19] Joe Hoffman. *Numerical Methods for Engineers and Scientists*. CRC Press, 2001.

[20] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 339–346, 2002.

[21] Vladislav Kraevoy and Alla Sheffer. Cross-parameterization and compatible remeshing of 3D models. In *Proceedings of the 31st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '04, pages 861–869, 2004.

[22] Vladislav Kraevoy, Alla Sheffer, and Craig Gotsman. Matchmaker: constructing constrained texture maps. In *Proceedings of the 30th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '03, pages 326–333, 2003.

[23] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. Prentice Hall, Inc., 1974.

[24] Francis Lazarus and Anne Verroust. Three-dimensional metamorphosis: a survey. *The Visual Computer*, 14:373–389, 1998.

[25] Aaron W. F. Lee, David Dobkin, Wim Sweldens, and Peter Schröder. Multiresolution mesh morphing. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 343–350, 1999.

[26] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: multiresolution adaptive parameterization of surfaces. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 95–104, 1998.

[27] Chang Ha Lee, Amitabh Varshney, and David W. Jacobs. Mesh saliency. In *Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '05, pages 659–666, 2005.

[28] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 362–371, 2002.

[29] Yaron Lipman and Thomas Funkhouser. Möbius voting for surface correspondence. In *Proceedings of the 36th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '09, pages 72:1–72:12, 2009.

[30] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 163–169, 1987.

[31] Miroslav Lovrić. *Vector Calculus*. John Wiley & Sons, Inc., 2007.

[32] Bruno Lévy. Laplace-Beltrami Eigenfunctions: Towards an Algorithm That "Understands" Geometry. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006*, pages 13–. IEEE Computer Society, 2006.

[33] Carl Dean Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, 2000.

[34] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds, 2002.

[35] Andrew W. Moore. An Intoductory Tutorial on Kd-Trees, 1991.

[36] Shigeyuki Morita. *Geometry of Differential Forms*. American Mathematical Society, 2000.

[37] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Laplacian mesh optimization. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, GRAPHITE '06, pages 381–389, 2006.

[38] Ulrich Pinkall and Konrad Polthier. Computing Discrete Minimal Surfaces and Their Conjugates. *Experimental Mathematics*, 2:15–36, 1993.

[39] Emil Praun and Hugues Hoppe. Spherical parametrization and remeshing. In *Proceedings of the 30th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '03, pages 340–349, 2003.

[40] Emil Praun, Wim Sweldens, and Peter Schröder. Consistent mesh parameterizations. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 179–184, 2001.

[41] Kenneth H. Rosen, editor. *Handbook of Discrete and Combinatorial Mathematics*. CRC Press, 2000.

[42] Steven Rosenberg. *The Laplacian on a Riemannian Manifold*. London Mathematical Society Student Texts 31. Cambridge University Press, 1997.

[43] Yousef Saad. *Iterative methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[44] John Schreiner, Arul Asirvatham, Emil Praun, and Hugues Hoppe. Inter-surface mapping. In *Proocedings of the 31st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '04, pages 870–877, 2004.

[45] Jonathan Richard Shewchuck. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, 1994.

[46] Olga Sorkine. Differential representations for mesh processing. *Computer Graphics Forum*, 25(4):789–807, 2006.

[47] Olga Sorkine and Marc Alexa. As-Rigid-As-Possible Surface Modeling. In *Proceedings of Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 109–116, 2007.

[48] Olga Sorkine and Daniel Cohen-Or. Least-squares Meshes. In *Proceedings of the Shape Modeling International 2004*, pages 191–199, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Olga Sorkine, Daniel Cohen-Or, Dror Irony, and Sivan Toledo. Geometry-Aware Bases for Shape Approximation. *IEEE Transactions on Visualization and Computer Graphics*, 11(2):171–180, 2005.

[50] Gabriel Taubin. A Signal Processing Approach To Fair Surface Design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 351–358, 1995.

[51] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3:71–86, January 1991.

[52] W. T. Tutte. Convex representation of graphs. *Proceedings of the London Mathematical Society*, 10:304–320, 1960.

[53] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13:743–768, 1960.

[54] Bruno Vallet and Bruno Lévy. Spectral Geometry Processing with Manifold Harmonics. *Computer Graphics Forum (Proceedings of Eurographics)*, 2008.

[55] Max Wardetzky, Saurabh Mathur, Felix Kälberer, and Eitan Grinspun. Discrete laplace operators: no free lunch. In *Proceedings of the fifth Eurographics symposium on Geometry processing*, pages 33–37. Eurographics Association, 2007.

[56] Lei Zhang, Ligang Liu, Zhongping Ji, and Guojin Wang. Manifold parameterization. *Lecture Notes in Computer Science (Proceedings of CGI)*, 2006.

[57] Djemel Ziou and Salvatore Tabbone. Edge detection techniques - an overview. *International Journal of Pattern Recognition and Image Analysis*, pages 537–559, 1998.