Topics in Surface Reconstruction and Spline Theory

Erik Alapää
Master's Thesis:

Topics in
Surface Reconstruction
and
Spline Theory

Erik Alapää,
Luleå University of Technology,
Sweden

Luleå,
August 27, 1999
Abstract

“Surface reconstruction” can be described as the general problem of reconstructing a surface (be it a function surface or the surface of a physical object such as a teapot) from a set of points in $\mathbb{R}^3$ “sampled” on or near the surface.

The report consists of two main parts; the first part (Section 2) is a description of a relatively new approach to surface reconstruction using subdivision surfaces. As a contrast, the second part (Section 3) describes the “classical” spline approach to surface reconstruction, and contains a comparison between two spline-based surface reconstruction algorithms, one that requires the input data to be gridded, and one that allows more general input data. The conclusion of the tests conducted is that for the kind of data used, pre-gridding and applying the “gridded” method is faster than using the method that does not require gridded data.
Acknowledgements

I wish to take the opportunity to thank my supervisor Dr. Inge Söderkvist from the department of Mathematics for his patience, for sharing his knowledge with me, and for his unbending enthusiasm.

This document was typeset using GNU Emacs and LaTeX on a Linux workstation.

Erik Alapää, Luleå, August 27, 1999.
**List of Figures**

1. The three fundamental mesh transformations. .................. 16
2. The neighborhood around a vertex $v$ of valence $n$. .......... 20
3. Basic subdivision masks. ...................................... 20
4. Position and tangent masks. ................................... 20
5. Subdivision masks modified to model sharp features. .......... 21
6. $B$-spline basis function of order 1, i.e degree 0. ............ 28
7. $B$-spline basis function of order 2, i.e degree 1. ............ 28
8. $B$-spline basis function of order 3, i.e degree 2. ............ 28
9. $B$-spline basis function of order 4, i.e degree 3. ............ 29
10. The knots start to move... .................................... 30
11. First and second derivatives still continuous... ............... 30
12. Discontinuous slope! .......................................... 30
13. Graph of a bicubic basis function. ............................ 33
14. The function surfaces used for testing the algorithms. ....... 37
15. Our first test function. ....................................... 38
16. Sampled points from the first test function. .................. 39
17. First reconstruction attempt, too big $s$. ........................ 40
18. An acceptable reconstruction, $s = 1.00$. ...................... 42
19. The function graph of $\sin(r)/r$. .............................. 43
20. The function graph of the pulse function $P$. ........................ 43
21. Effects of gradual decrease of $s$. ............................... 44
22. Level curve plots for fine-tuning of $s$ .......................... 45
23. Surface affected by noise at $s = 0.5$. .......................... 46
24. Final fit, $s = 0.8$. ........................................ 46
25. Reconstruction of step function. ................................ 47
26. Level curve comparison for reconstruction of faster-varying trig. function. .............................................. 47
27. Reconstruction of faster-varying trig. function. ............... 48
28. Contour plot for reconstruction of pulse function $P$ using the second (mesh data) algorithm. ........................... 50
29. Reconstruction of pulse function $P$ using the mesh data algorithm. ........................................ 50
30. Contour plot for reconstruction of pulse function $P$ using the mesh data algorithm and the new, faster pre-gridding. ............... 52
31. Reconstruction of pulse function $P$ using the mesh data algorithm and the new, faster pre-gridding. ................... 53
1 Introduction - What Is Surface Reconstruction?

Surface reconstruction is a large and diverse area of research, involving scientists and engineers from several different disciplines such as numerical analysis, computer graphics, computational geometry etc. Because of this diversity, even the description of the subject “surface reconstruction” as such depends on who you ask. When working with this master's thesis project, I have taken as my working definition a quite broad view of the subject – I have regarded “surface reconstruction” as the general problem of reconstructing a surface (be it a function surface or the surface of a physical object such as a teapot) from a set of points in \( \mathbb{R}^3 \) “sampled” on or near the surface.

There are a lot of applications for surface reconstruction. Below, we give examples of a few:

- Design/Reverse engineering: A designer may want to use an existing object as a basis for shaping the surface of a new product.
- Computer games: A game programmer may want to take samples from the face of a well-known actress to create the polygon face of a character in the game.
- A robotics research scientist may want to reconstruct surfaces from range images in order for the robot to identify some particular objects in its environment.
- A mechanical engineer often wants to take a physical surface and create a computerized representation of it in order to be able to perform calculations on the surface in a CAD/CAM or FEM program.

In this report, we will first (Section 2) have a look at a relatively new approach to surface reconstruction using subdivision surfaces. These methods were developed by Hoppe et al. and more information can be found in [HDD+92], [HDD+93], [HDD+94], [Hop94]. Then (Section 3), as a contrast to the methods developed by Hoppe et al, which are examples of surface reconstruction methods from the computer graphics “camp”, we will have a look at more classical spline methods for function reconstruction – methods that stem mainly from the numerical analysis “camp”. The main part of the spline section consists of a comparison between two different reconstruction algorithms developed by Paul Dierckx [Die93].
2 Triangular Meshes and Subdivision

In this section, the methods developed by Hoppe et al. ([HDD+92], [HDD+93],
[HDD+94], [Hop94]) for reconstructing a surface from a set of unorganized
points will be described. All those papers are easily accessible on the Internet at

What are the characteristics of the methods that will be described below?
Neither the topology, the geometry, nor the presence of boundaries are as-
sumed to be known in advance; All these characteristics of the unknown
surface to be reconstructed are inferred automatically from the data. The
reconstruction is carried out in three phases:

1. Sample a set of unorganized points in IR^3, lying on or near the un-
known surface, and construct from these points a mesh (for a more precise
definition of the concept mesh, see Section 2.1) describing a piecewise linear
surface with triangular faces.

2. Take the mesh from phase 1 and optimize this mesh to get a mesh
that is a closer approximation to the unknown surface, and also has fewer
vertices than the mesh from phase 1.

3. Construct a piecewise smooth surface from the optimized mesh cre-
ated in phase 2. In most cases, this piecewise smooth surface will have
a much greater resemblance to the unknown surface than the piecewise
linear surfaces from phase 1 or 2.

2.1 A few definitions

We begin by making precise what we mean by a surface:

A surface is a "compact, connected, orientable two-dimensional mani-
fold, possibly with boundary, embedded in IR^3". If a surface has no bound-
ary, it will be called a closed surface.

A piecewise linear surface with triangular faces will be called a simpli-
cial surface.

The notation ||x|| denotes the Euclidean length of a vector x, and if X, Y
are two point sets then d(X, Y) denotes the Hausdorff distance between X
and Y (i.e the distance between the two closest points of X and Y).

The point set X = {x_1, x_2, ..., x_n} of sampled points on or near the un-
known surface we want to reconstruct will of course contain sampling er-
rors, and we also need a way to quantify the sparseness of the sampled
point set. For the sampling errors, assume that each of the points x_i ∈ X
can be written as x_i = y_i + e_i, where the y_i are points on the unknown
surface, and also assume that the inequality ||e_i|| ≤ δ, i = 1, ..., n holds. If
δ is the smallest real number such that the inequality holds, then we say
that the sample set X is δ-noisy.

To capture the notion of sampling density, we make the following defi-
nition: Let Y = {y_1, y_2, ..., y_n} be a noiseless sample of a surface M. We
then say that Y is ρ-dense if any sphere with radius ρ and center in M
contains at least one of the sample points in Y. To extend the definition
to noisy samples, we say that the δ-noisy sample X = {x_1, x_2, ..., x_n} is
\( \rho \)-dense if there exists a noiseless \( \rho \)-dense sample \( Y = \{ y_1, y_2, \ldots, y_n \} \) such that \( x_i = y_i + e_i, ||e_i|| \leq \delta, i = 1, \ldots, n \).

2.2 From Unorganized Points in Euclidean 3-space to a Triangular Mesh

2.2.1 Overview

Phase 1 of the surface reconstruction algorithm has two stages - in the first stage, a signed distance function \( f: D \to \mathbb{R} \) is constructed, where \( D \subset \mathbb{R}^3 \) is a region near the data, and \( f \) estimates the signed geometric distance from points in \( D \) to the unknown surface \( M \). The zero set

\[ Z(f) = \{ x \in D | f(x) = 0 \} \]

is the estimate for \( M \). After \( f \) has been created, we enter stage two, where a contouring algorithm uses \( f \) to approximate \( Z(f) \) by a simplicial surface.

Remark: Since \( f \) is a signed distance function, zero is a regular value of \( f \). Thus, the implicit function theorem assures us that the approximation of \( M, Z(f) \), is a manifold. Zero is not a regular value of the unsigned distance function \( |f| \), so if \( |f| \) had been used, the approximation of \( M \) could have had non-manifold structure.

2.2.2 Construction of the Signed Distance Function

The process of constructing the signed distance function begins by computing an oriented tangent plane \( Tp(x_i) \) for each data point \( x_i \) in the sample set. To do this, the \( k \) nearest neighboring points of the point \( x_i \) are collected to form the \( k \)-neighborhood of \( x_i \), denoted \( Nbhd(x_i) \) (the parameter \( k \) is user-defined, assumed to be fixed and not indicated in the notation \( Nbhd(x_i) \)).

The representation of the \( i \)-th oriented tangent plane consists of a center point \( o_i \) together with a unit normal vector \( \hat{n}_i \). The signed distance of any point \( p \in \mathbb{R}^3 \) to the plane \( Tp(x_i) \) is defined to be \( \text{dist}_i(p) = (p - o_i) \cdot \hat{n}_i \). The center \( o_i \) and the normal \( \hat{n}_i \) are computed so that the plane \( \text{dist}_i(p) = 0 \) is the optimal fitting plane of \( Nbhd(x_i) \) in the least-squares sense, i.e. \( o_i \) is the centroid of \( Nbhd(x_i) \) and \( \hat{n}_i \) is computed using principal component analysis: First, the covariance matrix \( CV \) of \( Nbhd(x_i) \) is formed - it is symmetric, has dimension \( 3 \times 3 \), is positive semi-definite and can be written as

\[ CV = \sum_{y \in Nbhd(x_i)} (y - o_i) \otimes (y - o_i), \tag{1} \]

where \( \otimes \) denotes the outer product vector operator; if \( a \) and \( b \) are vectors with components \( a_i \) and \( b_j \), then \( a \otimes b \) is a matrix with \( a_i b_j \) at row \( i \) and column \( j \). If we let \( v \) denote the eigenvector of \( CV \) corresponding to the smallest eigenvalue of \( CV \), then \( \hat{n}_i \) is chosen to be either \(-v\) or \(+v\). The choice of sign determines the orientation of the \( i \)-th tangent plane \( Tp(x_i) \), and must be made so that planes that are close to each other become "consistently oriented". This difficulty will be discussed below.
2.2.3 A Method for Achieving Consistent Tangent Plane Orientations

The problem we are facing can be described as follows: From the sample set \( X \), take any two data points \( x_i, x_j \) that are "sufficiently close" (the notion of "sufficiently close" will be made more precise later). If the unknown sampled surface is smooth, and if the sample set \( X \) is dense, nearby points will have tangent planes that are close to being parallel, i.e. if \( T_p(x_i) = (\mathbf{o}_i, \mathbf{n}_i) \) and \( T_p(x_j) = (\mathbf{o}_j, \mathbf{n}_j) \), then \( \mathbf{n}_i \cdot \mathbf{n}_j \approx \pm 1 \). Since we want the orientation of the tangent planes to be consistent, either \( \mathbf{n}_i \) or \( \mathbf{n}_j \) should be flipped if we have \( \mathbf{n}_i \cdot \mathbf{n}_j \approx -1 \). The problem is that we want \( T_p(x_i) \) to be consistently oriented with all of \( x_i \)'s neighbors. This problem can be modeled as graph optimization - the graph will have one node \( N_i \) for each tangent plane, and \( N_i \) will have edges to all nodes that correspond to data points sufficiently close to \( x_i \). The cost on an edge \((i, j)\) will be \( \mathbf{n}_i \cdot \mathbf{n}_j \), and we want to select tangent plane orientations so as to maximize the total cost of the graph. Unfortunately, this problem can be shown to be NP-hard, which forces us to use an approximation algorithm.

When should two nodes be connected in our graph? We begin by assuming that our surface is a single connected component - for our graph, this implies that it, too, is connected. A reasonable starting point is to form the Euclidean Minimum Spanning Tree for the set of tangent plane centers \( \{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_n\} \) (see e.g. [CLR91]). To this tree, we add more edges to get a denser graph (denser in edges) by adding the edge \((i, j)\) if either \( \mathbf{o}_i \) is in the \( k \)-neighborhood of \( \mathbf{o}_j \) or vice versa. Thus, the graph we construct is a connected graph that encodes geometric proximity (i.e. how close the points are in the Euclidean norm in \( \mathbb{R}^3 \)) of the tangent plane centers \( \mathbf{o}_i \). Such a graph is called a Riemannian Graph.

After constructing our Riemannian Graph, we must decide how to propagate tangent plane orientation. Practical experiments have shown that the order of propagation is important - if propagation is based solely on geometric proximity, the resulting surface can be severely distorted compared to the unknown surface we wanted to approximate. Intuition tells us that it would be best if we propagated orientation along directions of low curvature of the data - to achieve this, we assign the cost \( 1 - |\mathbf{n}_i \cdot \mathbf{n}_j| \) to the edge \((i, j)\) in the Riemannian graph. This cost is non-negative and is low if the tangent planes \( T_p(x_i) \) and \( T_p(x_j) \) are nearly parallel. We can now achieve our goal of propagating orientation along directions of low curvature by simply calculating, and traversing, the Minimal Spanning Tree of the resulting graph. To begin the actual propagation, we need a place to start; A satisfactory approach is to take the tangent plane center with largest \( z \)-coordinate, force the corresponding unit normal vector to point upwards (i.e. in the \( +z \) direction), root the tree at this initial node, and then traversing the MST (Minimal Spanning Tree) in depth-first order. During traversal, each node is assigned an orientation consistent with that of its parent, i.e. if we are at \( T_p(x_i) \) and \( T_p(x_j) \) corresponds to the next tangent plane node to be visited, then the direction of \( \mathbf{n}_j \) is reversed if \( \mathbf{n}_i \cdot \mathbf{n}_j < 0 \).
2.2.4 Computing Distances Using the Oriented Tangent Planes

When the oriented tangent planes have been computed, we are ready to
describe how \( f(p) \), the signed distance from an arbitrary point \( p \in D \subset \mathbb{R}^3 \),
where \( D \) is a region near the data, can be computed. Ideally, if \( M \) is an
orientable surface, the signed distance \( f(p) \) from \( p \) to \( M \) is the distance
from \( p \) to the point \( z \) on \( M \) closest to \( p \), multiplied by \( \pm 1 \) depending on if
\( p \) is above or below the surface \( M \). In our case, we do not know \( M \), but we
can use the oriented tangent planes \( T_p(x_i) \) as local linear approximations
to \( M \). To do this, we first compute which one of the tangent plane centers
\( o_1, o_2, \ldots, o_n \) that is closest to \( p \). Call this center \( o_i \). We then compute \( f(p) \)
using the projection of \( p \) onto the normal to the plane \( T_p(x_i) \), i.e \( f(p) =
\text{dist}_i(p) = (p - o_i) \cdot n_i \). If the the surface \( M \) is known not to have boundaries,
this is all we need to compute \( f(p) \). If \( M \) has borders, things become a little
bit more difficult. Recall that we assume that the set \( X = \{ x_1, x_2, \ldots, x_n \} \)
of data points is \( \rho \)-dense and \( \delta \)-noisy. If there was no noise, we would know
that a point \( z \) with \( d(z, X) > \rho \) could not be a point of \( M \) (by definition of the
concept \( \rho \)-dense). If noise is present, a point \( z \) with \( d(z, X) > (\rho + \delta) \) could
not be a point of \( M \), by similar reasoning. Thus, if the projection \( z \) of a point
\( p \) onto the normal to the closest tangent plane has \( d(z, X) > (\rho + \delta) \), we take
\( f(p) \) to be undefined. These undefined values are used by the contouring
algorithm of stage two to identify boundaries of the unknown surface.

2.2.5 Stage Two – Contour Tracing

After the signed distance function has been computed, a contour tracing
algorithm can be applied. Contour tracing is the extraction of an isosurface
from a scalar function. This is a well-studied problem, and the description
of its different solutions lies beyond the scope of this text. We need only
refer you to [HDD+92] and the references therein to contour tracing papers.
Hoppes et al. ([HDD+92]) also make some short comments on their usage of
the “marching cubes” algorithm for contour tracing with the signed distance
function described above.

2.3 Mesh Optimization

As stated earlier, the goal of this phase is:

“Take the mesh from phase 1 and optimize this mesh to get a mesh
that is a closer approximation to the unknown surface, and also has fewer
vertices than the mesh from phase 1.”

If we look at this goal, we see that there are two competing objectives
- intuitively, if we use more vertices, we can make our piecewise linear
surface a better approximation of the unknown surface, but this conflicts
with our desire to construct a mesh that has a small footprint, i.e can be
described by few vertices. To model these two competing objectives, we
will try to minimize an energy function that contains one term \( E_{\text{rep}} \) that
penalizes meshes with many vertices, and one term \( E_{\text{dist}} \) that penalizes
meshes that are poor approximations to the unknown surface. The energy
function will have one additional regularizing term \( E_{\text{spring}} \) that is used to
prevent spikes in the mesh in regions where there is no data - more on that
issue later. Before we go into more detail about the energy function, we need to make more precise the notion of a mesh (the notation below and the notation \(E_{\text{rep}}, E_{\text{dist}}, E_{\text{spring}}\) is borrowed from [HDD+93]):

A mesh \(M\) is a pair \(M = (K, V)\), where \(K\) is a simplicial complex that describes how the vertices, edges and faces of the mesh are connected, and \(V\) is a set of vertices \(V = \{v_1, v_2, \ldots, v_m\}, v_i \in \mathbb{R}^3\) defining the shape of the mesh in \(\mathbb{R}^3\). Simply put, \(K\) tells us which points are connected, and \(V\) tells us where the points are located in \(\mathbb{R}^3\). To quote Hoppe et al. [HDD+93] "A simplicial complex \(K\) consists of a set of vertices together with a set of non-empty subsets of the vertices, called the simplices of \(K\), such that any set consisting of exactly one vertex is a simplex in \(K\), and every non-empty subset of a simplex in \(K\) is again a simplex in \(K\)." ... "The 0-simplices \(\{i\} \in K\) are called vertices, the 1-simplices \(\{i, j\} \in K\) are called edges, and the 2-simplices \(\{i, j, k\} \in K\) are called faces."

To realize a mesh \(M = (K, V)\) geometrically in \(\mathbb{R}^3\), we can proceed as follows. First, we form the topological realization \(|K|\) of \(K\) in \(\mathbb{R}^3\) by identifying the \(m\) vertices \(v_1, \ldots, v_m \in V\) with the \(m\) standard basis vectors \(e_1, \ldots, e_m\) of \(\mathbb{R}^m\). Then, for each simplex \(s \in K\) we let \(|s|\) denote the convex hull of the vertices of \(s\) in \(\mathbb{R}^m\), and we let \(|K| = \bigcup_{s \in K}|s|\). To return us back to the "real, Euclidean world" \(\mathbb{R}^3\), the linear map \(\phi: \mathbb{R}^m \to \mathbb{R}^3\) that maps the \(i\)-th standard basis vector \(e_i \in \mathbb{R}^m\) to \(v_i \in \mathbb{R}^3\) is formed. The geometric realization of the mesh \(M\) is the image \(\phi_V(|K|)\). Note that we write the map as \(\phi_V\) to emphasize the fact that the map is completely specified by the vertex positions \(v_1, \ldots, v_m\). If the map is 1-1, i.e. if \(\phi_V(|K|)\) is not self-intersecting, then \(\phi_V\) is called an embedding. Of course, only a restricted set of vertex positions \(V\) result in \(\phi_V\) being an embedding.

If \(\phi_V\) is 1-1 (an embedding), then we can find the unique pre-image \(b \in |K|\) of any point \(p \in \phi_V(|K|)\), i.e. \(p = \phi_V(b)\). The vector \(b \in |K|\) is called the barycentric coordinate vector of \(p\) (with respect to \(K\)). Of course, this means that barycentric coordinate vectors are convex combinations of the standard basis vectors \(e_i \in \mathbb{R}^m\) corresponding to the three vertices of a face of \(K\). Thus, any barycentric coordinate vector has at most three non-zero entries. If \(p\) lies on an edge of the mesh, its corresponding barycentric coordinate vector will have only two non-zero entries, and if \(p\) is a vertex, the barycentric coordinate vector will have only one non-zero entry.

\(E_{\text{dist}}\), the measure of how close the mesh approximates the unknown surface, is taken to be the sum of squared Euclidean distances from the set of \(n\) sampled data points \(X = \{x_1, \ldots, x_n\}\) to the mesh, i.e.

\[
E_{\text{dist}}(K, V) = \sum_{i=1}^{n} d^2(x_i, \phi_V(|K|)).
\] (2)

The representation energy term \(E_{\text{rep}}\) that penalizes meshes with many vertices is defined to be proportional to \(m\), the number of vertices of \(K\):

\[
E_{\text{rep}}(K) = c_{\text{rep}}m.
\] (3)

Note that \(E_{\text{rep}}\) is a function of \(K\) only, the set \(V\) of positions of the \(m\) vertices, does not affect \(E_{\text{rep}}\).

During the optimization, vertices can both be added to and removed from the mesh. Obviously, when adding a vertex, the distance energy \(E_{\text{dist}}\)
is likely to be reduced. The representation energy $E_{rep}$ couples a cost to this reduction of $E_{dist}$ - if the reduction of $E_{dist}$ is not “significant enough”, it will be rejected. When removing a vertex from the mesh, the $E_{rep}$ term acts as a compensation for the probable increase in distance energy, thus encouraging the removal. The constant $c_{rep}$ in the representation energy term is a user-specified parameter – a small value of $c_{rep}$ indicates that the goal of obtaining a good fit has priority over the goal of obtaining a sparse representation of the mesh.

Practical tests have showed that if one tries to minimize an energy function with only the two terms $E_{dist}$ and $E_{rep}$, the resulting mesh may have large spikes in it - spikes that do not correspond to any real artifacts on the unknown surface we want so reconstruct. This problem is rooted in the fact that a minimum of $E_{dist} + E_{rep}$ may not exist. As Hoppe et al. have shown, [HDD+93], appendix A.1, one can prove that if a third term, the “spring energy” $E_{spring}$ is added, then the existence of a minimum of the energy function $E = E_{dist} + E_{rep} + E_{spring}$ is guaranteed. How is this spring energy term constructed? Well, one simply places on each of the edges in the mesh a spring with spring constant $K$ and rest length zero:

$$E_{spring}(K, V) = \sum_{\{j,k\} \in K} K ||v_j - v_k||^2. \quad (4)$$

Note that the spring energy is not a smoothness penalty - the goal is not to penalize large dihedral angles in the mesh, since such features may be part of the unknown surface we seek to reconstruct. Instead, $E_{spring}$ is best viewed as a regularizing term acting as a guide that directs the optimization to a local minimum. The magnitude of the spring energy can gradually be decreased as the optimization converges to the solution - more on this issue later.

Before delving into the description of how the actual minimization of the energy function is done, a short remark on scaling issues seems appropriate. Some applications benefit by having a procedure that is scale-invariant, a condition equivalent to having a unitless energy function $E$. Invariance under Euclidean motion and under uniform scaling is achieved by pre-scaling the data points $X$ and the initial mesh $M_0$ uniformly to fit in a unit cube. Of course, when the optimization is finished, a post-processing step can be used to undo the initial scaling.

### 2.3.1 Minimization of the Energy Function

We want to optimize the energy function

$$E(K, V) = E = E_{dist}(K, V) + E_{rep}(K) + E_{spring}(K, V) \quad (5)$$

over the set $\mathcal{K}$ of simplicial complexes homeomorphic to the initial simplicial complex $K_0$ (for definitions of homeomorphic simplicial complexes e.t.c., see [HDD+93]), and over the set of vertex positions $V$. The optimization will be carried out as two nested loops corresponding to two optimizations, one inner, continuous optimization over the vertex positions $V$ for fixed simplicial complex $K$, and one outer, discrete optimization over the set $\mathcal{K}$. $E(K, V)$
depends on the representation energy parameter $c_{rep}$ and on the spring constant $\kappa$. $c_{rep}$ will be set by the user, while the method of setting $\kappa$ will be described in Section 2.3.4. We begin by describing the inner optimization:

### 2.3.2 Minimizing $E$ for Fixed Simplicial Complex $K$

The problem can be stated as follows: Given an initial guess for the vertex positions $V$, we want to find $E(K, V) = \min_V E(K, V)$, i.e. we want to find the best possible embedding of $K$, which amounts to finding the set of vertex positions $V$ that minimizes $E$ for fixed $K$. Recall that the energy $E$ is

$$E(K, V) = E_{dist}(K, V) + E_{rep}(K) + E_{spring}(K, V)$$

Since $E_{rep} = E_{rep}(K)$, i.e. $E_{rep}$ does not depend on $V$, we need only minimize $E_{dist}(K, V) + E_{spring}(K, V)$.

When evaluating the distance energy $E_{dist}(K, V)$, the distance $d_i$ of each data point $x_i$ to the mesh $M = \phi_V([K])$ has to be computed. For each $i$, $i = 1 \ldots n$, finding this distance $d_i$ is in itself a minimization problem. It can be written

$$d_i^2 = d^2(x_i, \phi_V([K])) = \min_{b_i \in [K]} ||x_i - \phi_V(b_i)||^2.$$

In the expression for $d_i^2$ above, the unknown is the barycentric coordinate vector $b_i \in [K] \subset \mathbb{R}^m$ of the projection of $x_i$ onto the mesh $M$. This means that minimizing $E(K, V)$ for fixed $K$ is equivalent to minimizing a new function

$$E(K, V, B) = \sum_{i=1}^{n} ||x_i - \phi_V(b_i)||^2 + E_{spring}(K, V)$$

over the vertex positions $V = \{v_1, \ldots, v_m\}, v_i \in \mathbb{R}^3$ and over the barycentric coordinates $B = \{b_1, \ldots, b_n\}, b_i \in [K] \subset \mathbb{R}^m$.

The problem of minimizing $E$ for fixed simplicial complex $K$ can now be divided into two subproblems:

1. For fixed set of vertex positions $V$, find the set of optimal barycentric coordinates $B$.

2. For fixed barycentric coordinates $B$, find the set of vertex positions that minimizes $E(K, V, B)$.

These two subproblems are entered into a loop, and in principle, one could keep looping until some convergence criterion is met. In practice, one simply performs a fixed number of iterations.

We first describe point 1 above. For each data point $x_i$ we want to minimize the squared distance $d_i$ defined in Equation 7. For fixed vertex positions $V$, this is equivalent to finding $b_i$ such that

$$b_i = \min_{b_i \in [K]} ||x_i - \phi_V(b_i)||.$$

---

13
The “brute force” method of doing this is to simply project each data point \( x_i \) onto every face of the mesh and choose as \( b_i \) the projection that is closest to \( x_i \). To speed up the process, the faces of the mesh are first inserted into a spatial positioning data structure, and for each data point \( x_i \) only the faces close to \( x_i \) need to be considered. To achieve even more efficiency, the coherence between iterations is exploited - the simplifying assumption is that the projection of a data point is close to its projection from the previous iteration. To be more specific, the point is only projected onto the faces that share at least one vertex with the previous face. This is a method that could, in theory, fail, but it has proven to work well in practice.

Now, we move on to point 2 above, “For fixed barycentric coordinates \( B \), find the set of vertex positions that minimizes \( E(K, V, B) \).” This problem can be split into three separate minimizations, one for each coordinate direction \((x, y, z)\). We describe the minimization of the first coordinates, i.e. the \( x \)-coordinates, the other two coordinate directions are done in the same way.

Let \( e \) denote the number of edges (1-simplices) of the mesh, and also note that \( e \) is \( O(m) \). First, from the set of \( m \) mesh vertices \( \{v_1, \ldots, v_m\} \) we form the \( m \)-vector \( v^1 \) consisting of the first coordinates of the \( v_i \), i.e. if \( v_i = (x_i, y_i, z_i) \) then \( v_i^1 = x_i \). Then, we also form the \((n + e)\)-vector \( d^1 \) consisting of the \( n \) \( x \)-coordinates of the \( n \) data points \( x_i \), followed by \( e \) zeroes. We also form the \((n + e) \times m\) design matrix \( A \) whose first \( n \) rows correspond to the barycentric coordinates for the projected data points, i.e. at row \( i \) we have the barycentric coordinates of the projection of the data point \( x_i \) onto the mesh. Note that this means that the first \( n \) rows of \( A \) each have at most three non-zero entries. Each one of the last \( e \) rows of \( A \) have only two non-zero entries corresponding to the \( x \) part of the spring energies of the \( e \) edges in the mesh. For example, if row \( n + 53 \) corresponds to the 53rd edge of the mesh going from mesh vertex \( v_6 \) to mesh vertex \( v_8 \), then row \( n + 53 \) would have two non-zero entries \( +\sqrt{\kappa} \) and \(-\sqrt{\kappa}\) in its 6th and 8th positions, respectively. (Recall that the spring energy for the edge is \( \kappa ||v_6 - v_8||^2 \)). The matrices are shown below:

\[
A = \begin{bmatrix}
0 & \cdots & b_{11} & 0 & \cdots & b_{13} & 0 & \cdots & b_{12} & 0 & \cdots \\
0 & \cdots & b_{22} & 0 & \cdots & b_{21} & 0 & \cdots & b_{23} & 0 & \cdots \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots \\
0 & \cdots & b_{n3} & 0 & \cdots & b_{n1} & 0 & \cdots & b_{n2} & 0 & \cdots \\
\end{bmatrix}
\]

(10)

\( e \) rows of spring energy terms

\[
v^1 = [x_1, x_2, \ldots, x_m]^T
\]

(11)

\[
d^1 = [d_1^1, d_2^1, \ldots, d_m^1, 0, \ldots, 0]^T
\]

(12)
After forming $A, v^1$, and $d^1$, we can now express the minimization problem for the first coordinate as minimizing

$$\|Av^1 - d^1\|^2$$

over $v^1$. This problem is a linear least squares problem and is solved by using the well-known iterative method of conjugate gradients. There are theoretical bounds on how many iterations that may be required - the method is guaranteed to find the exact solution in as many iterations as there are distinct singular values of $A$. Since $A$ cannot have more than $m$ singular values, this means that we need at most $m$ iterations. However, practical experiments have shown that far fewer iterations will suffice for our purposes - e.g. for meshes with $m$ as large as $10^4$, as few as 200 iterations are enough to generate satisfactory results.

We conclude with a short performance analysis: Hoppes et al. point out that there are two time-consuming operations in each iteration of the conjugate gradient algorithm; the multiplication of $A$ by an $(n + e)$-vector and the multiplication of $A^T$ by an $m$-vector. From the description above, we know that $A$ is sparse, so both operations can be executed in $O(n + m)$ time. $A$ is stored in a sparse form that only requires $O(n + m)$ bytes of RAM for storage. Thus, it is possible to obtain a satisfactory solution in $O(n + m)$ time. It is interesting to note that a typical non-iterative method for solving the same linear least squares problem (e.g. QR decomposition) would require as much as $O(n + m)^2$ to find an exact solution, if we assume that we do not exploit the fact that the matrices involved are sparse.

### 2.3.3 The Outer, Discrete Optimization Over Simplicial Complexes

As we described earlier, the mesh optimization algorithm consists of two nested optimization loops, the inner one (described above) is a continuous optimization over vertex positions $V$ while holding the simplicial complex constant. The outer optimization is a discrete optimization over the set $\kappa$ of simplicial complexes $K'$ homeomorphic to the original simplicial complex $K_0$. So, we want to minimize $E(K)$ over $\kappa$. To do this, we need to define a set of three elementary transformations that take a simplicial complex $K$ to a new simplicial complex $K'$. The three transformations are called edge collapse, edge swap and edge split. This set of elementary transformations is complete in the sense that any simplicial complex homeomorphic to the initial simplicial complex $K_0$ (i.e any $K \in \kappa$) can be obtained by applying a sequence of these transformations to $K_0$. (In fact, only edge split and edge collapse are needed. Edge swap is included because it helps the optimization process to tunnel through small hills in the energy function $E(K)$). The transformations are described in Figure 1.

Having defined these three transformations, we now need to consider this question: "Can the transformations take us out of $\kappa$, the set of simplicial complexes homeomorphic to the initial mesh $K_0$?" The answer is "Yes, if we do not check if the transformation is a legal move". What, then, is a "legal move"? This is the topic of our next discussion.
Obviously, an edge split cannot change the topological type of $K$, so an edge split is always a legal move. For an edge collapse, we need to check three conditions, and for a swap, we need one condition. To describe those conditions, we require some new terminology: An edge $\{i, j\} \in K$ is defined to be a boundary edge if it is a subset of only one face $\{i, j, k\} \in K$, and a vertex $\{i\} \in K$ is defined to be a boundary vertex if there exists a boundary edge $\{i, j\} \in K$ such that $\{i\} \subset \{i, j\}$. Now, for an edge collapse $K \rightarrow K'$ that collapses the edge $\{i, j\} \in K$ to be legal, all of the following three conditions need to be satisfied:

1. $\{k\} \in K$ adjacent to both $\{i\} \in K$ and $\{j\} \in K \Rightarrow \{i, j, k\} \in K$
2. $\{i\} \in K$ and $\{j\} \in K$ are both boundary vertices $\Rightarrow \{i, j\} \in K$ is a boundary edge.
3a) Neither $\{i\} \in K$ nor $\{j\} \in K$ are boundary edges $\Rightarrow K$ has more than 4 vertices or
3b) Either $\{i\} \in K$ or $\{j\} \in K$ are boundary edges $\Rightarrow K$ has more than 3 vertices

The condition that an edge swap transformation $K \rightarrow K'$ that replaces the edge $\{i, j\} \in K$ with the edge $\{k, l\} \in K'$ is legal is equivalent to the condition $\{k, l\} \notin K$. For proofs of the validity of the conditions above, we refer you to [HDD+93].

Now, we are ready to give an idealized sketch of the outer, discrete opti-
Given an initial simplicial complex $K_0$, we want to minimize $E(K)$ over $K$. Thus, we want to find a sequence of legal moves that takes us from $K_0$ to a minimum of $E(K)$. Recall that the energy function $E(K)$ contains a representation energy term $E_{\text{rep}}$ proportional to the number $m$ of vertices in the mesh. This term is included to make it beneficial to remove vertices from the mesh even when a slight increase in the distance energy $E_{\text{dist}}$ occurs.

Since one of the goals of mesh optimization is to obtain a sparse mesh (i.e. a mesh with few vertices), we first try the edge collapse transformation on some edge of the mesh, because this transformation reduces the number of vertices of the mesh. The simplicial complex before transformation will be denoted $K$, and the new simplicial complex will be denoted $K'$. The transformation is then checked to make sure that it is a legal move (in the rest of the discussion, the legality checks will not be explicitly mentioned, and we will use the terms “transformation” and “legal move” interchangeably). Now, if $E(K') < E(K)$ the legal move is accepted, otherwise it is rejected. If the edge collapse was rejected, we try an edge swap on the same edge and recalculate the energy. If $E(K') < E(K)$, we accept the edge swap, otherwise we try our last resort, an edge split. One reason that this transformation is tried last is because it has the undesirable effect of increasing the number of vertices ($0$-simplices) of the simplicial complex (and in the mesh also, of course). The energy of the resulting mesh $K''$ is again recalculated and if $E(K'') < E(K)$ we accept the legal move, otherwise it is rejected.

This pattern of trials to find beneficial legal moves is then repeated over and over again in the outer optimization loop. If a large number of trials fail to produce a decrease in energy, the search is terminated.

We will now describe some improvements of the idealized algorithm outlined above. All these optimizations are heuristics that capitalize on one simple fact - when changing one part of the mesh (e.g. collapsing an edge), the optimal structure of parts of the mesh that are geometrically far from the change is virtually unaffected.

In the idealized algorithm, we required evaluation of $E(K')$ after each attempt to find a legal move. Since $E(K') = \min_{V} E(K', V)$, this would imply that each trial of an edge collapse, edge swap or edge split would require a full run-through of the inner optimization loop. Instead, fast local heuristics are used. They are all based on the extraction of a small submesh in the vicinity of the transformation, together with the data points projecting onto that submesh. Then, the change in total energy is estimated by only considering the change in energy of the submesh. Note that this estimate is always a bit pessimistic, since a global optimization would only further reduce the energy of the transformed mesh. This implies that this heuristic will never lead us to accept transformations that increase the total energy of the mesh.

One further local heuristic is used to speed up the performance of the optimization - in the description of the idealized algorithm, we paid no attention to the question of which edge in the mesh should be the “victim” of the next set of trials of the three edge transformations. In the actual algorithm, a candidate set is used for this purpose. The candidate set initially contains all edges of the mesh, and one edge is randomly selected from the
set. Then, the edge collapse, edge swap and edge split transformations are tried, in that order. If a transformation reduces the energy, the transformation is accepted and all neighboring edges are added to the candidate set, if they are not already in it. If none of the three elementary transformations yielded a lower energy, the edge is removed from the candidate set.

2.3.4 Setting the Spring Constant

The spring constant $\kappa$ determines the contribution of the spring energy $E_{\text{spring}}$ to the total energy. $E_{\text{spring}}$ is viewed as a regularizing term that guides the optimization to a good local minimum of the total energy. As the mesh optimization produces meshes that are closer and closer to being optimal, the spring constant can be gradually reduced - that is, the mesh optimization algorithm is called several times, each time with a lower value of $\kappa$. An example of a sequence of $\kappa$ values that has been used in actual tests on several different data sets is $\kappa = 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$.

2.4 Piecewise Smooth Surfaces Using Subdivision Methods

We will now describe phase 3. A quick reminder of the purpose of phase 3:

"3. Construct a piecewise smooth surface from the optimized mesh created in phase 2. In most cases, this piecewise smooth surface will have a much greater resemblance to the unknown surface than the piecewise linear surfaces from phase 1 or 2."

In practice, most real-life objects we want to reconstruct are best modeled by piecewise smooth surfaces - the piecewise linear surfaces used in phase 1 and 2 will therefore now be replaced by piecewise smooth surface representations. Recall that the purpose of phase 1 was to reconstruct an unknown surface, and that the purpose of phase 2 was to optimize the phase 1 mesh. If we now in phase 3 are going to optimize with piecewise smooth surface representations, one might ask: "Why not skip phase 2 entirely and go directly from the dense unoptimized mesh of phase 1 to phase 3?"

While this approach would, at least in theory, be feasible, there are two reasons for retaining phase 2: it is computationally more efficient to optimize over piecewise linear surfaces in the early stages of the mesh optimization, and initial estimates of sharp features are much more reliable when they are obtained from the phase 2 mesh.

We will begin by describing subdivision surfaces in general, and then we will summarize how Hopps et al have modified the subdivision rules to be able to model some sharp features of the surfaces we wish to reconstruct.

2.4.1 Smooth Surface Representation With Subdivision Surfaces

A subdivision surface is defined by repeated refinement of an initial control mesh. The subdivision scheme we will describe here is based on triangular meshes and was introduced by Loop [Loo87].
Loop's Subdivision

Loop's subdivision scheme is a generalization of $C^2$ quartic triangular B-splines. Starting with an initial control mesh $M = M^0$, each step of the formal subdivision process transforms the $r$:th mesh $M^r = (K^r, V^r)$ into a new mesh $M^{r+1} = (K^{r+1}, V^{r+1})$, where the vertices $V^{r+1}$ are computed as affine combinations of the vertices $V^r$. Some of the vertices of $V^{r+1}$ have a natural correspondence to the vertices of the previous mesh – those vertices will be called vertex points, while the remaining vertices of $V^{r+1}$ will be called edge points. Let the vertex $v^r$ denote a vertex of $V^r$ having $n$ neighbors $v_1^r, \ldots, v_n^r$; $v^r$ is then said to have valence $n$. Figure 2 shows the neighborhood around a vertex $v^r$ of valence $n$. Also, let $v_i^{r+1}$ denote the edge point of $V^{r+1}$ corresponding to the edge $v_i^r v_i'$. and let $v_i^{r+1}$ denote the vertex point of $V^{r+1}$ corresponding to $v^r$. Now, the positions of $v_i^{r+1}$ and $v_i^{r+1}$ are computed according to the following subdivision rules:

$$v_i^{r+1} = \frac{\alpha(n) v_i^r + v_i^0 + \cdots + v_i^{n}}{\alpha(n) + n}$$

$$v_i^{r+1} = \frac{3v_i^r + 3v_i^r + v_i^{r-1} + v_i^{r+1}}{8}, \quad i = 1, \ldots, n.$$  \hspace{1cm} (14)

The subscripts are to be taken modulo $n$, and $\alpha(n) = \frac{n(1-a(n))}{a(n)}$, where $a(n) = \frac{3}{8} - \frac{2 + \cos(2\pi n)}{8}$. Subdivision rules such as the affine combinations in Equation 14 can be visualized by so-called subdivision masks. Figure 3 shows such masks – the one labeled “vertex mask” corresponds to the equation for the vertex point (the first formula in Equation 14), while the one labeled “edge mask” corresponds to edge points (the second formula in Equation 14).

Properties of Loop's subdivision surfaces

Subdivision surfaces in general are defined as limits of an infinite refinement process. Closed form expressions for these limit surfaces are in most cases not available. In spite of the lack of closed form expressions for the surfaces, various properties of the surfaces can be calculated. For example, exact points on a subdivision surface and exact tangent planes can be calculated.

For a study of subdivision surface properties, Equation 14 is rewritten in matrix form:

$$(v^{r+1}, v_1^{r+1}, \ldots, v_n^{r+1})^T = S_n(v^r, v_1^r, \ldots, v_n^r)^T$$

$$= S_n^{r+1}(v^0, v_1^0, \ldots, v_n^0)^T$$ \hspace{1cm} (15)

The superscript $T$ denotes matrix transpose. $S_n$ is called the local subdivision matrix. As the number of subdivision refinements approach infinity, i.e. $r \rightarrow \infty$, each vertex point $v^r$ approaches a point on the limit surface. From Equation 15 the reader might guess that the limit point can be calculated by analyzing the eigenstructure of $S_n$. In [HCD93] it is shown that the limit point $v^\infty$ can be calculated as

$$v^\infty = \frac{l_1 v^0 + l_2 v_1^0 + \cdots + l_{n+1} v_n^0}{l_1 + l_2 + \cdots + l_{n+1}},$$ \hspace{1cm} (16)
Figure 2: The neighborhood around a vertex $v^r$ of valence $n$.

Figure 3: Basic subdivision masks.

Figure 4: Position and tangent masks.
where \( l_1, \ldots, l_{n+1} \) is the dominant left eigenvector of the local subdivision matrix \( S_n \). In [Loo87] it is shown that in the case of Loop's subdivision surfaces, this affine combination can be visualized with the position mask shown in Figure 4.

The analysis of the smoothness properties can also be made using eigenanalysis of \( S_n \). In [Loo87] and [Rei92] it is shown that Loop's surfaces are tangent plane continuous, and in [HCD93] it is shown that using the two left eigenvectors of \( S_n \) corresponding to the second largest eigenvalue (that eigenvalue has multiplicity 2), tangent vectors of the limit surface can be computed. For Loop's surface the vectors

\[
\begin{align*}
\mathbf{u}_1 &= c_1 \mathbf{v}_1^0 + c_2 \mathbf{v}_2^0 + \ldots + c_n \mathbf{v}_n^0, \\
\mathbf{u}_2 &= c_1 \mathbf{v}_1^0 + c_2 \mathbf{v}_2^0 + \ldots + c_1 \mathbf{v}_n^0,
\end{align*}
\]

where \( c_i = \cos(2\pi i / n) \), span the tangent plane of the surface. Of course, the cross product of the two spanning tangent vectors then gives an exact normal vector to the surface. That normal vector is useful for Phong shading etc. Figure 4 shows tangent masks visualizing the formulas of Equation 17.

**Modifications to Accommodate Tangent Plane Discontinuities**

Many surfaces encountered in practice are only piecewise smooth. It is therefore desirable to have a surface reconstruction algorithm that explicitly models tangent plane discontinuities. In [HDD+94] Hoppe et al. describe an extension of Loop's methods to piecewise smooth surfaces. Figure 5 shows some examples of modified masks from [HDD+94]. When looking at the edge mask in Figure 5, it is obvious that such a mask decouples the behavior of the surface parts that are located on opposite parts of the sharp edge (see below) modeled by the mask.

Hoppe et al. [HDD+94] define extended subdivision rules for several sharp surface features: “A crease is a tangent line smooth curve along which the surface is \( C^0 \) but not \( C^1 \); A corner is a point where three or more creases
meet; finally, a dart is an interior point of a surface where a crease terminates.” These features are sufficient to model many tangent plane discontinuities, but not all—a cone or two coincident darts are examples of surface features that cannot be modeled exactly even by the modified subdivision rules.

We conclude this brief summary of the extended subdivision rules by remarking that a pair \((K, V)\) in the mesh \(M = (K, V)\) to be optimized is now augmented with a third part, \(L\). \(L\) is the subset of edges in the simplicial complex \(K\) that are tagged as sharp. Initially, edges of \(K\) having incident faces that make a dihedral angle above a threshold (e.g. 40 degrees) are tagged as sharp. Then, during the subdivision process, edges that are created through refinement of a sharp edge are tagged as sharp. For more information on the modifications of the subdivision formulas we refer you to [HDD+94].

### 2.4.2 Phase 3: Fitting Piecewise Smooth Subdivision Surfaces

The input to Phase 3 consists of the unstructured set of sampled points \(X = \{x_1, \ldots, x_n\}\) together with the optimized mesh \(M = (K, V)\) from phase 2 (Section 2.3). The simplicial complex \(K\) in \(M\) is then augmented with a subset \(L \subset K\) of sharp edges. As stated above, edges of \(K\) having incident faces that make a dihedral angle above a threshold (e.g. 40 degrees) are tagged as sharp and entered into \(L\). The pair \((K, L)\) is then called a tagged simplicial complex. The triple \(M = (K, L, V)\) is then labeled \(M_0 = (K_0, L_0, V_0)\) and is called a tagged mesh. \(M_0\) becomes the initial mesh for the optimization in phase 3. The objective of phase 3 is to find a tagged mesh that accurately fits the data and has a concise representation. The search space \(\mathcal{M}\) is the space of tagged meshes \(M = (K, L, V)\) where \(K\) is of the same topological type as \(K_0\).

The two competing goals of accuracy and conciseness are represented in an energy function similar to the one in phase 2,

\[
E(K, L, V) = E_{\text{dist}}(K, L, V) + c_{\text{rep}}m + c_{\text{sharp}}e_s, \tag{18}
\]

where \(E_{\text{dist}}(K, L, V)\) is the total squared distance from the sampled points in \(X\) to the subdivision surface, \(c_{\text{rep}}m\) is a penalty on the number of mesh vertices \(m\) and \(c_{\text{sharp}}e_s\) is a penalty on the number of sharp edges \(e_s\). Thus, \(c_{\text{rep}}\) is a parameter that controls the trade-off between the competing goals of fidelity to the data and conciseness of fit, while \(c_{\text{sharp}}\) controls the trade-off between the competing goals of smoothness of the subdivision surface and fidelity to the data. The parameter \(c_{\text{rep}}\) is set by the user. For \(c_{\text{sharp}}\), the setting \(c_{\text{sharp}} = c_{\text{rep}}/5\) has worked well in practice. The reader might notice that the spring energy term in the energy function from phase 2 is absent. The reason for the presence of the spring energy term in phase 2 was that it helped guide the mesh optimization algorithm into a good local energy well. As stated in [HDD+93], the spring energy term has not been necessary in phase 3 for the type of data used during testing.
Minimization of the Energy Function

As in phase 2 (Section 2.3) the problem of minimizing the energy function is divided into two parts: one outer, discrete optimization over tagged simplicial complexes \((K, L)\) and one inner, continuous optimization over the set of control vertex positions \(V\) for fixed \((K, L)\).

Optimizing over \(V\) for fixed \((K, L)\)

If we take a look at Equation 18, we see that when optimizing over \(V\) for fixed \((K, L)\), we only need to consider the \(E_{\text{dist}}\) term (because if \((K, L)\) is fixed, then \(m\) and \(e\) are fixed), i.e. determine

\[
E(K, L) = \min_V E_{\text{dist}}(K, L, V),
\]

the minimum distance energy for fixed \((K, L)\). Ideally, to compute the distance energy \(E_{\text{dist}}\), we would like to be able to project the data points \(x_i\) onto the subdivision surface. Since the surface is defined as the limit of an infinite process, this is not possible. Instead, we project onto a piecewise linear approximation \(\tilde{M}^r\) to the subdivision surface \(S(M)\). The approximation \(\tilde{M}^r\) is obtained by first subdividing the initial mesh \(r\) times (typically \(r = 2\) is used) to obtain a refined mesh \(M^r\) and then calculating the limit positions of the vertices of \(M^r\) by using the position masks. Now, since the subdivision rules enable us to express \(M^r\) as an affine combination of the vertices \(V\), and since the position masks make it possible to express \(\tilde{M}^r\) as an affine combination of \(M^r\)'s vertices, we can, by composition, express \(\tilde{M}^r\) as an affine combination of the vertices \(V\). Thus, if we treat \(V\) as an \(m \times 3\) matrix whose rows are the \((x, y, z)\) coordinate triplets of the vertices, we can express each vertex \(\tilde{v}^r\) of \(\tilde{M}^r\) as \(\tilde{v}^r = y V\), where each entry of the row vector \(y\) is computed as the composition of \(r\) fold subdivision followed by the application of a position mask. Now, since \(\tilde{M}^r\) is a piecewise linear surface, every point of \(\tilde{M}^r\), not just the vertices, can be computed as an affine combination of the vertices \(V\). For each data point \(x_i\), let \(z_i\) denote the point on \(\tilde{M}^r\) closest to \(x_i\). Since \(z_i\) is a point on \(\tilde{M}^r\), it can be written as an affine combination of the vertices \(V\), i.e. \(z_i = y_i V\). Thus, \(E_{\text{dist}}\) can be expressed as

\[
E_{\text{dist}} = \sum_{i=1}^{n} ||x_i - y_i V||^2.
\]

This expression for the distance energy is quadratic in \(V\). This means that for fixed \(y_i\), optimization over \(V\) is a linear least squares problem. Also, since the subdivision rules are local, the vectors \(y_i\) are sparse. The situation described above is amenable to an iterative minimization scheme that alternates between two steps:

1. For fixed \(V\), compute the projections \(y_i V\) of the data points onto \(\tilde{M}^r\).

2. For fixed \(y_1, \ldots, y_n\), optimize the distance energy \(E_{\text{dist}}\) over the vertex positions \(V\).

The second step can be solved using a sparse, iterative conjugate gradient method, as in phase 2. Since the \(y_i\)'s have approximately 12 non-zero
entries, compared to 3 in phase 2, the conjugate gradient method becomes more expensive, but the expense only grows linearly with the number of non-zero entries, which is an acceptable cost.

The Outer, Discrete Optimization Over \((K, L)\)

The outer optimization also closely parallels the one in phase 2. Starting from the initial tagged simplicial complex \((K_0, L_0)\) we want to find a minimum for the energy \(E(K, L)\) by applying a series of elementary transformations to \((K_0, L_0)\). In phase 2, we had three such elementary transformations, and they were shown in Figure 1, Section 2.3. In phase 3, another transformation is added, edge tag, where an edge of \(K\) is tagged as sharp and entered into \(L\). As in phase 2, these four transformations are complete in the sense that given any two tagged simplicial complexes of the same topological type, one can be transformed into the other by applying a series of the four transformations. In Section 2.3 we gave a description of a set of conditions that ensures that the application of an elementary transformation has not changed the topological type of the simplicial complex. That set of conditions is valid in phase 3, too. We can then refine the description of the goal of the algorithm by saying that we want to find a series of legal moves (see Section 2.3) taking the initial tagged simplicial complex \((K_0, L_0)\) into a tagged simplicial complex \((K, L)\) that gives a minimum for the energy.

Again, as in phase 2, the goal is accomplished by a variant of random descent. First, a candidate set of edges is formed. Initially, this candidate set consists of all edges in \(K_0\). Then, one edge from the candidate set is randomly selected, and the four elementary transformations are then tried, in turn, on this edge, until a legal move \((K, V) \Rightarrow (K', V')\) with \(E(K', V') < E(K, V)\) is found. If no such legal move is found, the edge is removed from the candidate set. If a move was accepted, the edges affected by the transformation are added to the candidate set. This process is then repeated until the candidate set is empty.

The cost of recalculating \(E(K', V')\) after each speculative move is too high to make the idealized algorithm outlined above practical. By replacing the exact calculation of \(E(K', V')\) with a much simpler approximation, the algorithm becomes much faster. This simplification is based on the fact that a change in one part of the control mesh \((K, L, V)\) does not affect the optimal positions of vertices far away from the part of the mesh that “moved” because of the change. Thus, when calculating \(E(K', V')\), only the positions of the control vertices close to the affected edge need to be optimized, and only the projections onto \(\tilde{M}r\) of the data points supported by those vertices need to be recalculated. For a more precise description, we refer you to \([HDD+94]\).

Final Remarks

We hope that our presentation has given the reader a reasonably good understanding of the methods developed by Hoppe et al. It is beyond the scope of this text to present all the results of testing the algorithms with different sets of data. For a large set of figures displaying results of such
tests, comparisons with other methods such as NURBS and suggestions for extensions of the algorithms e.t.c, we refer you to [Hop94].
3 Spline Methods

As a contrast to the methods developed by Hoppe et. al. described above, we will now turn our attention to another class of surface reconstruction methods. These methods are based on the use of $B$-splines, and before diving into specifics, we will give a brief background on the general subject of splines and $B$-splines. Note that for simplicity, we have chosen to restrict ourselves to surfaces that can be described as real-valued functions of two real variables. This simple setting makes it easy to focus on the main characteristics of spline methods without getting bogged down by auxiliary details.

3.1 Background on Splines and B-splines

3.1.1 Fundamentals

The definition of a spline is as follows [Die93]:

**Definition 3.1 (Spline Function)** A function $s(x)$ defined on a finite interval $[a, b]$ is called a spline function of degree $k > 0$ (order $k+1$) having as knots the strictly increasing sequence $\lambda_j$, $j = 0, 1, \ldots, g+1$ ($\lambda_0 = a, \lambda_{g+1} = b$) if the following two conditions are satisfied:

1. **Polynomial on Subinterval** On each knot interval $[\lambda_j, \lambda_{j+1}]$, $s(x)$ is given by a polynomial of degree $k$ at most.

   $$s|_{[\lambda_j, \lambda_{j+1}]} \in \mathcal{P}_k, \ j = 0, 1, \ldots, g.$$ 

2. **Continuity** The function $s(x)$ and its derivatives up to order $k - 1$ are all continuous on $[a, b]$.

   $$s(x) \in C^{k-1}[a, b].$$

Definition 3.1 can be extended in several ways. For example, the restriction that the degree $k > 0$ can be relaxed to allow for piecewise constant splines, $k = 0$. To do this, condition 2 of Definition 3.1 above has to be dropped, and in condition 1, $[\lambda_j, \lambda_{j+1}]$ must be replaced by $[\lambda_j, \lambda_{j+1})$. Another very common extension is to drop the requirement that the knot sequence must be strictly increasing to allow for coincident knots, i.e. knot multiplicity $> 1$ As we shall see below, this means that the continuity conditions at the knots will have to be relaxed.

As stated in Definition 3.1, in each separate knot interval the spline function is simply a polynomial of some degree $k$, (order $(k+1)$). Thus, any spline $s(x)$ can be written as ([Die93])

$$s(x) = p_{k,j}(x) = \sum_{i=0}^{k} a_{i,j} (x - \lambda_j)^i \quad \text{if} \quad \lambda_j \leq x \leq \lambda_{j+1}, \ j = 0, \ldots, g. \quad (21)$$

It can be shown that the set of spline functions satisfying conditions 1 and 2 of Definition 3.1 above are a vector space (the operations of vector addition
and scalar multiplication are the usual ones, e.g. if $s_1$ and $s_2$ are spline functions of the same degree and defined on the same set of knots, their sum $s_1 + s_2$ as vectors is the spline $(s_1 + s_2)(x) = s_1(x) + s_2(x))$. This space will be denoted as $\eta_k(\lambda_0, \lambda_1, \ldots, \lambda_{g+1})$ or shorter, $\eta_k$. Note that the coefficients $a_{i,j}$ are not all independent – e.g. the continuity condition (condition 2 above) imposes constraints on the $a_{i,j}$. It can be shown ([Die93]) that the vector space $\eta_k(\lambda_0, \lambda_1, \ldots, \lambda_{g+1})$ has dimension $g + k + 1$. Thus, instead of the representation in Equation 21, it would be useful to have a more compact representation of a spline function $s \in \eta_k$. The standard representation for vectors (i.e. functions) $s \in \eta_k$ is the set of $B$-spline basis functions; by using the so-called truncated power function

$$
(x-c)^+ = \begin{cases} 
(x-c)^k, & \text{if } x \geq c, \\
0, & \text{if } x < c,
\end{cases}
$$

(22)

the (normalized) $B$-spline (basis function) $N_{i,k+1}$ of degree $k$ with knots $\lambda_i, \ldots, \lambda_{i+k+1}$ can be expressed explicitly as

$$
N_{i,k+1}(x) = (\lambda_{i+k+1} - \lambda_i) \sum_{j=0}^{k+1} \frac{(\lambda_{i+j} - x)^+}{\prod_{l=0, l \neq j}^{k+1} (\lambda_{i+j} - \lambda_{i+l})}. 
$$

(23)

One aspect of the $B$-spline basis functions that is very important in actual computations and applications is the local support property:

$$
N_{i,k+1}(x) = 0 \quad \text{if } x \notin [\lambda_i, \lambda_{i+k+1}].
$$

Now, by using the knots $\lambda_j, j = 0, 1, \ldots, g+1$ ($\lambda_0 = a, \lambda_{g+1} = b$) we can construct $g - k + 1$ linearly independent $B$-spline basis functions (or shorter, $B$-splines) of degree $k$. Since our vector space $\eta_k(\lambda_0, \lambda_1, \ldots, \lambda_{g+1})$ has dimension $g + k + 1$, we need $2k$ more basis vectors. To get these, additional knots are introduced at both “ends” of the knot vector sequence. Specifically, one introduces the additional knots

$$
\lambda_{-k} \leq \lambda_{-k+1} \leq \cdots \leq \lambda_{-1} \leq \lambda_0 = a, \\
b = \lambda_{g+1} \leq \lambda_{g+2} \leq \cdots \leq \lambda_{g+k} \leq \lambda_{g+k+1}.
$$

(24)

With these knots, the “missing” $B$-splines can be constructed, and we have a basis for our vector space $\eta_k$. Then, we are able to express any spline $s(x) \in \eta_k$ as a linear combination of the $B$-splines:

$$
s(x) = \sum_{i=-k}^{g} c_i N_{i,k+1}(x),
$$

(25)

where the $c_i$ are called the $B$-spline coefficients of $s(x)$. We would also like to point out that the $B$-spline representation has the desirable property of numerical stability. For more information on $B$-spline properties, selection of the additional knots e.t.c., we refer you to [Die93]. Figures 6-9 show some examples of $B$-spline basis functions and the polynomial pieces they are built of. The thin lines in the figures show the knot positions and the polynomial parts the $B$-splines are built of.
Figure 6: B-spline basis function of order 1, i.e. degree 0.

Figure 7: B-spline basis function of order 2, i.e. degree 1.

Figure 8: B-spline basis function of order 3, i.e. degree 2.
We conclude this brief introduction to splines with some figures that demonstrate the effect of coincident knots. A simple rule for understanding the effect of coincident knots is

"order - knot multiplicity = number of continuity conditions at knot".

For example, if we have a cubic 

$B$-spline (i.e. degree 3) and a knot with multiplicity 3, the rule gives $4 - 3 = 1$ continuity condition at the knot. This means that we can only require that the $B$-spline itself is continuous – no conditions can be imposed on the first or second derivatives of the $B$-spline. Figure 9 shows a cubic $B$-spline with a uniform knot sequence $\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4 = (0, 1, 2, 3, 4)$. Figures 10 - 12 demonstrate what happens to the cubic $B$-spline when we let the knots that initially are at 1 and 2 draw closer and closer to 3. In Figure 12, we have 3 coincident knots and we can see that the first derivative of the $B$-spline is no longer continuous at the triple knot, i.e. the $B$-spline is $C^0$ but not $C^1$ at the triple knot. If we first had let two knots coincide (before we let 3 knots coincide), the $B$-spline graph would have had continuous slope at the double knot, but not continuous curvature, i.e. it would have been $C^1$ but not $C^2$ at the double knot.
Figure 10: The knots start to move...

Figure 11: First and second derivatives still continuous...

Figure 12: Discontinuous slope!
3.1.2 Smoothing Splines

In Section 3.3 we will make a comparison of two surface reconstruction methods that are both based on bivariate smoothing tensor product splines. The purpose of this section (Section 3.1.2) and Section 3.2.2 is to motivate our interest in smoothing splines and to give a short description of the smoothing spline criteria.

A group of classic methods for curve fitting with splines are the least-squares free-knot algorithms. These algorithms create splines that minimize the (weighted) sum $\delta$ of squared distances from points on or near the original curve to the spline approximant, i.e.

$$\delta = \sum_{r=1}^{m} (w_r(y_r - s(x_r)))^2,$$

(26)

where the $w_r$ are the weights, the $(x_r, y_r)$ are the points on or near the curve one wants to approximate. As stated in [Die93], Chapter 5, free knot least-squares methods are very useful and can give excellent results in many applications. However, the computational cost increases very rapidly with the number of knots, and a badly chosen set of initial knots may cause failure to find optimal knot positions. Therefore, the group of algorithms based on the least-squares approximation criterion is ill-suited for data sets that would require more than approximately 10 interior knots. Instead, a smoothing spline algorithm might be appropriate. The class of smoothing splines that will be described below are faster and more flexible than least-squares splines for many applications. The smoothing approximation criterion for the single-variable case can be described as follows:

Find a spline of specified degree $k$ that solves the constrained minimization problem of minimizing

$$\tilde{\eta} = \sum_{i=1}^{g} (s^{(k)}(\lambda_i^+) - s^{(k)}(\lambda_i^-))^2,$$

(27)

subject to the constraint

$$\delta = \sum_{r=1}^{m} (w_r(y_r - s(x_r)))^2 \leq S.$$

(28)

Recall that a spline of degree $k$ with no coincident knots has continuous derivatives at up to order $k - 1$ at the knots. We now want to minimize the squared sum $\eta$ of the discontinuity jumps in the $k$th order derivative of our smoothing spline $s$, subject to the constraint that the spline does not deviate too much from the set of data points we want to approximate. Having laid the foundation for understanding the smoothing spline criterion in the most basic setting, the single-variable case, we will now give a short description of bivariate tensor product splines in Section 3.2.1 before moving on to discussing bivariate tensor product smoothing splines in Section 3.2.2.
3.2 Splines in More than One Dimension

3.2.1 Tensor Product B-splines

We will now take a brief look at one of the most common generalizations to several variables of the B-splines we encountered in Section 3.1.1. This generalization is called (bivariate) tensor product splines and its definition closely parallels the univariate case, so the reader is recommended to compare it with Definition 3.1.

Definition 3.2 (Tensor Product Spline) Consider the strictly increasing (knot) sequences

\[ a = \lambda_0 < \lambda_1 < \cdots < \lambda_y < \lambda_{y+1} = b, \]
\[ c = \mu_0 < \mu_1 < \cdots < \mu_h < \mu_{h+1} = d. \]

The bivariate function \( s(x, y) \) is called a tensor product spline of degrees \( k \) (x-dir.) and \( l \) (y-dir.) on the rectangle \( R = [a, b] \times [c, d] \) with knot sequences \( \lambda_i, i = 0, \ldots, y+1 \) in the x direction and \( \mu_j, j = 0, \ldots, h+1 \) in the y direction if the following two conditions are satisfied:

1. **Polynomial on Subrectangle** On each separate subrectangle \( R_{i,j} = [\lambda_i, \lambda_{i+1}] \times [\mu_j, \mu_{j+1}], i = 0, \ldots, y, j = 0, \ldots, h \), the function \( s(x, y) \) is a bivariate polynomial of degrees \( k \) and \( l \) in the x and y directions, respectively, i.e.

\[ s(x, y)|_{R_{i,j}} \in P_k \otimes P_l. \]

2. **Continuity** The function \( s(x, y) \) and all its partial derivatives of order up to \( k - 1 \) in the x direction and of order up to \( l - 1 \) in the y direction are continuous on \( R \), i.e.

\[ \frac{\partial^{i+j}s(x, y)}{\partial x^i \partial y^j} \in C(R), i = 0, \ldots, k - 1, j = 0, \ldots, l - 1. \]

We will denote the vector space of functions satisfying Definition 3.2 by \( \eta_{k,l}([\lambda_0, \lambda_{y+1}] \times [\mu_0, \mu_{h+1}]) \) or simply \( \eta_{k,l} \). This vector space has dimension \((y+1)(h+1)\) and by introducing extra knots in a way similar to the univariate case, we get a basis of functions \( N_{i,k+1}(x)M_{j,l+1}(y) \) enabling us to express any spline \( s(x, y) \in \eta_{k,l} \) as a linear combination

\[ s(x, y) = \sum_{i=-k}^{g} \sum_{j=-l}^{h} c_{i,j} N_{i,k+1}(x)M_{j,l+1}(y). \]

As in the single-variable case, the \( c_{i,j} \) are called the B-spline coefficients.

We conclude this brief discussion of tensor product splines by showing an example of a bivariate B-spline basis function in Figure 13.

3.2.2 Bivariate Smoothing Splines

The smoothing criterion for a bivariate smoothing spline is quite similar to the smoothing criterion we met in Section 3.1.2. Given an appropriate
Figure 13: Graph of a bicubic basis function.
smoothing norm $\eta(c)$ (more about the smoothing norm later), where $c$ represents the matrix of $B$-spline coefficients, we look for a spline function $s(x, y)$ that is a solution to the following constrained minimization problem:

Minimize

$$\eta(c)$$

subject to the constraint

$$\delta(c) \leq S.$$  \hspace{1cm} (33)

As in the single-variable case, $\delta(c)$ is a sum of squared distances between the data and the spline approximant. However, the exact form of $\delta(c)$ will depend on which one of the two methods in Section 3.3 we are talking about. The first method takes as input data an array of $m$ data points $(x_r, y_r, z_r)$, so in this case $\delta(c)$ becomes

$$\delta(c) = \sum_{r=1}^{m} (w_r(z_r - s(x_r, y_r)))^2.$$ \hspace{1cm} (35)

In the second method, the input data must be gridded, i.e. it takes as input a set of function values $z_{q,r}$ corresponding to the grid points $(x_q, y_r)$, $q = 1, \ldots, m_1$, $r = 1, \ldots, m_2$. This implies that $\delta(c)$ becomes a double sum

$$\delta(c) = \sum_{q=1}^{m_1} \sum_{r=1}^{m_2} (w_{q,r}(z_{q,r} - s(x_q, y_r)))^2.$$ \hspace{1cm} (36)

Note that the dependence of $\delta$ on $c$ in the right-hand sides of Equations 35 and 36 is implicit. It is of course caused by the dependence of $s(x, y)$ on $c$, as can be seen in Equation 32.

The smoothing norm naturally becomes more complex in the bivariate case. However, the idea is similar – we want to minimize the discontinuity jumps of the spline at the knot positions. To construct the norm, we begin by considering what would happen if the spline $S(x, y)$ was one single polynomial throughout the entire rectangle $R = [a, b] \times [c, d]$. Then we would have

$$\frac{\partial^{k+j}s(\lambda_0 +, y)}{\partial x^k \partial y^j} \equiv \frac{\partial^{k+j}s(\lambda_0 -, y)}{\partial x^k \partial y^j},$$

$q = 1, \ldots, g; j = 0, \ldots, l; \mu_0 \leq y \leq \mu_{k+1},$ \hspace{1cm} (37)

$$\frac{\partial^{l+i}s(x, \mu_0 +)}{\partial x^l \partial y^i} = \frac{\partial^{l+i}s(x, \mu_0 -)}{\partial x^l \partial y^i},$$

$r = 1, \ldots, h; i = 0, \ldots, l; \lambda_0 \leq x \leq \lambda_{g+1}.$ \hspace{1cm} (38)

If we let

$$a_{i,q} = N_{i,k+1}^{(k)}(\lambda_0 +) - N_{i,k+1}^{(k)}(\lambda_0 -)$$ \hspace{1cm} (39)

and

$$b_{j,r} = M_{j,l+1}^{(l)}(\mu_0 +) - M_{j,l+1}^{(l)}(\mu_0 -),$$ \hspace{1cm} (40)
then Equations 37 and 38 can be rewritten by differentiating the expression for a bivariate tensor spline (Equation 32). We get

\[
\sum_{i=-k}^{g} c_{i,j} a_{i,q} = 0, \quad q = 1, \ldots, g, \quad j = -l, \ldots, h
\]  

(41)

and

\[
\sum_{j=-l}^{h} c_{i,j} b_{j,r} = 0, \quad r = 1, \ldots, h, \quad i = -k, \ldots, g.
\]  

(42)

Of course, when the spline \( s(x,y) \) is not one single polynomial throughout the entire rectangle \( R = [a,b] \times [c,d] \), we cannot fulfill all the conditions in Equations 41 and 42. We can, however, try to minimize the discontinuity jumps by defining a smoothing norm \( \eta(c) \) as

\[
\eta(c) = \sum_{q=1}^{g} \sum_{j=-l}^{h} (\sum_{i=-k}^{g} c_{i,j} a_{i,q})^2 + \sum_{r=1}^{h} \sum_{j=-l}^{h} (\sum_{i=-k}^{h} c_{i,j} b_{j,r})^2.
\]  

(43)

For details on algorithms for solving the constrained minimization problem defined in Equations 33 and 34 in the two cases of gridded and non-gridded data, we refer you to [Die93]. We now move on to the description of the tests we have done to compare the two algorithms (note that in this context, we use the words “algorithm” and “method” interchangeably, i.e. “the two methods” and “the two algorithms” mean the same thing).

### 3.2.3 The Smoothing Factor \( S \) and How to Set It

From now on, we will often refer to the upper bound \( S \) for the sum of squared residuals in Equation 34 as the smoothing factor. The smoothing factor plays a very important role for the behavior of the two smoothing algorithms we will test in Section 3.3. Take a look at Equations 33 and 34! They model the competing requirements of smoothness of fit (Equation 33) and fidelity of the fit to the data (Equation 34). Broadly speaking, it is evident that if \( S \) is set too big, we prioritize smoothness over fidelity of fit. In the extreme case, we will get a perfectly smooth spline, i.e. \( s(x,y) \) will become one single polynomial throughout the entire rectangle \( R = [a,b] \times [c,d] \) (because with one single polynomial throughout \( R \), the discontinuity jumps will all be 0). At the other extreme, with \( S \) very small, we prioritize the minimization of the least-squares distance sum over smoothness, and \( s(x,y) \) will be the least-squares spline corresponding to the data. Further information about setting \( S \) can be found in this quote from the file surfit.f from the FITPACK library:

"by means of the parameter s, the user can control the tradeoff between closeness of fit and smoothness of fit of the approximation. if s is too large, the spline will be too smooth and signal will be lost ; if s is too small the spline will pick up too much noise. in the extreme cases the program will return an interpolating spline if s=0 and the weighted least-squares polynomial (degrees kx,ky)if s is very large. between these extremes, a properly chosen s will result in a good compromise between closeness of fit and smoothness of fit. to decide whether an approximation, corresponding to a certain s is
satisfactory the user is highly recommended to inspect the fits graphically.

Recommended values for \( s \) depend on the weights \( w(i) \). If these are taken as \( 1/d(i) \) with \( d(i) \) an estimate of the standard deviation of \( z(i) \), a good \( s \)-value should be found in the range \((m-sqrt(2*m),m+sqrt(2*m))\). If nothing is known about the statistical error in \( z(i) \) each \( w(i) \) can be set equal to one and \( s \) determined by trial and error, taking account of the comments above. The best is then to start with a very large value of \( s \) (to determine the least-squares polynomial and the corresponding upper bound \( fp0 \) for \( s \)) and then progressively decrease the value of \( s \) (say by a factor 10 in the beginning, i.e. \( s=fp0/10, fp0/100, \ldots \) and more carefully as the approximation shows more detail) to obtain closer fits.

To choose \( s \) very small is strongly discouraged. This considerably increases computation time and memory requirements. It may also cause rank-deficiency (\( ier<-2 \)) and endanger numerical stability.

As recommended, we chose to set \( S \) manually by inspecting surface plots and level curve plots. Since the sampling errors were modeled as uniform (no single sample is expected to be more accurate than another), we set all the weights to 1, as recommended in the quote above. The tests will be described extensively in Section 3.3.

### 3.3 A Comparison of Scattered Data Versus Mesh Data Methods

#### 3.3.1 Introduction

In this section, we will consider the following problem: Given \( m \) points \((x_i, y_i), i = 1, \ldots, m\) in the plane and \( m \) corresponding \( z \)-values sampled from an unknown surface, decide which one of two methods is more appropriate for reconstructing a reasonable approximation of the original surface. The methods we will compare are described in the book [Die93] and they are both based on tensor product \( B \)-splines. One method requires the input data to be specified on a rectangular grid, while the other method does not require gridded data (we will refer to the methods as the gridded (or mesh) and the scattered data methods, respectively). Thus, to make a fair comparison between the two methods, we consider also the time/CPU cycles needed to pre-grid the data when the mesh data method is used.

To conduct the tests needed to determine which of the above methods is best suited for the kind of surface reconstruction problem mentioned above, we have used implementations available on the Internet, coming from the FITPACK library described in [Die93]. FITPACK routines can be downloaded from the netlib database free of charge. The FITPACK library is written in standard FORTRAN as so many other mathematical software libraries. Since FORTRAN is an old language and seldom the language of choice when teaching today's students programming, and the author himself has practically no knowledge of FORTRAN, the question of how to use the FITPACK routines from another language, preferably C or C++, presented itself. Since this question might obviously be of some general interest, this report includes a description of how to call FORTRAN routines from C++ (see Section 4).

To begin the tests, we also needed samples from some surfaces. A natural and convenient choice is to simulate this by choosing a wide range of
functions of two variables and then take as samples the $z$-values (i.e. function values) at random points of the domains of the functions. To get an indication of the kinds of functions we used, take a look at the graphs in Figure 14. Note that we will refer to the function derived from $\frac{\sin(r)}{r}$ as the pulse function.

**Figure 14:** The function surfaces used for testing the algorithms.
3.3.2 Tests Using the Scattered Data Algorithm

The first algorithm we tested is a smoothing tensor-product spline algorithm designed to take scattered points from a function surface and reconstruct a reasonable approximation of the surface. Thus, to test the performance of this algorithm, we simply choose a bivariate function, generate some function values at random points of the domain, add some random errors to the “sampled” data and feed the data into the algorithm. The aim of our tests is to find smoothing parameters that give acceptable reconstructions of the surfaces, and to time the algorithm for comparisons with the other algorithm. We begin with a nice, slowly varying and smooth function, \( \sin(x)\sin(y) \), and the domain \( D \) is chosen so that the \( \sin \) function will go through a full period in the \( x \)-direction and a half period in the \( y \)-direction, i.e. \( D = \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid 0 \leq x \leq 2\pi, 0 \leq y \leq \pi \} \). The function graph is depicted in Figure 15.

The sampled points, including the sampling error (1000 points were “sampled”, and the maximum error was set to 0.05, 5 percent of the sine function’s maximum value) are shown in Figure 16.

We first calculated a \( B \)-spline surface with smoothing factor \( s = 100 \) (see Section 3.2.3 for more information about the smoothing factor). The surface is shown in Figure 17. Note that in this section, we refer to the smoothing factor as \( s \) instead of \( S \), because the FITPACK code uses the name \( s \) (while the book [Die93] uses \( S \)).
Figure 16: Sampled points from the first test function.
Figure 17: First reconstruction attempt, too big $s$. 

Spline surface corresponding to $\sin(x) \cdot \sin(y)$, $s=100$

Error, i.e. difference (actual surface − spline surface)
The lower graph of Figure 17 is a plot of the error, i.e. the difference between the original surface and the $B$-spline approximation. The number of knots needed was 8 in both the $x$- and the $y$-directions. The maximum error is approximately 0.217 and the sum of squared residuals is approx. 3.08. This indicates that we can use a smaller smoothing factor $s$. We try $s = 1.00$ and get the surface shown in Figure 18.

We now get 11 knots in the $x$-direction and 10 knots in the $y$-direction. The maximum error is approx. 0.037 and the sum of squared residuals is approx 0.999. The error plot (in Figure 18) also shows that we get a better fit with $s = 1.00$.

We will now use a more difficult function surface, which will be referred to as the pulse function $P$. First, recall the shape of the surface corresponding to the radially symmetrical bivariate function $\sin(r)/r$, where $r$ is the distance along the $x$-$y$-plane to $(0,0)$. The surface is shown in Figure 19.

For convenience, we want to use the same domain in this and the following examples as we had in the first example, i.e. $D = \{(x,y) \in \mathbb{R} \times \mathbb{R} \mid 0 \leq x \leq 2\pi, 0 \leq y \leq \pi\}$. Therefore, we will move the center for the $\sin(r)/r$ from $(0,0)$ to the center of the rectangular domain $D$. We also want a “sharper peak”, so we will scale the $x$ and $y$ variables with a factor $2\pi$. Finally, we want the function surface to remain at $z = 0$ after it has first come down to this value, to make the surface a bit more difficult to construct. Mathematically, we can describe the pulse function by setting $r = \sqrt{(x-\pi)^2 + (y-\pi/2)^2}$. $P$ can then be written as

$$P(r) = \begin{cases} \sin(2\pi r)/(2\pi r) & : r < 1/2 \\ 0 & : r \geq 1/2 \end{cases}, (x,y) \in D.$$ 

The function surface of the pulse function is as shown in Figure 20.

Beginning with a large value of the smoothing factor $s$, we gradually decrease $s$ until we obtain an acceptable fit. Figure 21 illustrates the gradual improvements.

In order to fine-tune the smoothing factor $s$, contour plots of the original surface and the $B$-spline surfaces are very useful. In all of the following contour plots, the level curves of the original surface are plotted as continuous curves, while the spline surface level curves are dashed. The upper plot of Figure 22 shows a contour plot for $s = 1.0$. We try to lower the value of $s$ to 0.8 to get a better fit. The result is also shown in Figure 22. If we go much lower, to $s = 0.5$, the fit gets worse, because it becomes too affected by the sampling errors (i.e. the data is almost interpolated, we do not get enough smoothing of the errors). This is very clear in the bottom plot of Figure 22. The inadequate smoothing is also evident if we look at the spline surface obtained with $s = 0.5$, as shown in Figure 23. Thus, we choose $s = 0.8$ as our final $s$ value for this surface. The final $B$-spline surface is displayed in Figure 24.

As a last note, we remark that when choosing $s$ too low, i.e. $s = 0.5$ the calculations were much slower and we got 23 knots in the $x$ direction, 25 in the $y$ direction and the maximum error was approx. 0.299. As a comparison, $s = 0.8$ gave 17 knots in the $x$ direction, 16 in the $y$ direction and the maximum error was much smaller, approx. 0.088.

Before moving on to testing the second algorithm (the one that operates on gridded data) we will take a quick look at the first algorithm's behavior for
Figure 18: An acceptable reconstruction, $s = 1.00$. 

Spline surface corresponding to $\sin(x) \sin(y)$, $s=1.00$.
Figure 19: The function graph of $\frac{\sin(r)}{r}$.

Figure 20: The function graph of the pulse function $P$. 
two more surfaces. Since the last example (the pulse function surface) was worked out in such detail, the reader should by now have sufficient understanding of the general behavior of the algorithm (influence of smoothing factor e.t.c). Thus, we will now only present a few graphs for the remaining two surfaces. First, a discontinuous function that looks like a cube with the side-walls and bottom removed (since the discontinuities are exactly where the side-walls would have been in a cube); the original surface is shown in Figure 14 on page 37. For lack of better name, we will refer to this surface as the step function even though it is not really a pure bivariate Heaviside step. The reconstructed surface is shown in Figure 25. The smoothing factor 14.4 was chosen by visual inspection. As in previous cases, the surface became wildly oscillating if the smoothing factor was chosen too small.

As our final example, we take the bivariate function \( \sin(\pi x) \sin(\pi y) \) whose graph is also shown in Figure 14 on page 37. We found that choosing the smoothing factor \( s = 0.9 \) gave a good fit. Figure 26 is a contour plot of the original function and the spline surface. As before, the level contours of the spline surface are shown dashed, while the level contours of the original surface are continuous.

Finally, we present a plot of the reconstructed surface in Figure 27.
Level curves get closer with \( s = 0.8 \).

Level curves affected by noise at \( s = 0.5 \).

**Figure 22:** Level curve plots for fine-tuning of \( s \).
Figure 23: Surface affected by noise at $s = 0.5$.

Figure 24: Final fit, $s = 0.8$. 
Figure 25: Reconstruction of step function.

Figure 26: Level curve comparison for reconstruction of faster-varying trig. function.
Figure 27: Reconstruction of faster-varying trig. function.
3.3.3 Tests Using Pre-Gridding of Data

The main difference between the two algorithms is that the second one requires that the points in its input have to be organized in a grid, i.e. a rectangular mesh. We will refer to the second algorithm as the “mesh data algorithm”. Since we want to be able to use “un-gridded” sampled point sets from surfaces, we will need to create a grid of points from the unorganized samples. The time for this pre-gridding must of course be taken into consideration when comparing the two surface reconstruction algorithms.

Another difference is the smoothing factor $s$. As described in section 3.2.3 at page 35, the smoothing factor for the gridded data algorithm is the sum of squared residuals at the grid points, while $s$ in the scattered data algorithm is the sum of squared residuals at the actual sampled points.

The creation of the grid requires the user to input the size of the grid. For simplicity, we use a uniform grid, i.e. the step length in the $x$-direction is the same all throughout the rectangle containing the samples and so is the step length in the $y$-direction. As in the tests of the first algorithm, the $(x, y)$-values of all the sampled points from the function surfaces are contained within the rectangle $D = \{(x, y) \in \mathbb{R} \times \mathbb{R} | 0 \leq x \leq 2\pi, 0 \leq y \leq \pi \}$. To maintain the same step length in both directions, the number of grid points in the $x$ direction are therefore always taken as twice the number of grid points in the $y$ direction. We of course use the same maximum sampling error as before, i.e. 0.05. We begin with the pulse function surface shown in Figure 14. Our aim is to obtain a fit of comparable quality to the one we got with the first algorithm, and to compare the execution speed of the algorithms.

Recall that the final result with the first algorithm on the pulse function surface was a fit with 17 knots in the $x$ direction, 16 in the $y$ direction and maximum error approx. 0.088. The time required to complete the calculation was 4.1 seconds. This fit was obtained using a smoothing factor of 0.8, but as stated above, the smoothing factors of the two algorithms are not directly comparable. Instead, we now try to find a grid that is reasonably sparse and gives similar max. error with its optimal smoothing factor. After similar experiments as with the first algorithm, but now with the added degree of freedom coming from the choice of grid size, we found that a 70x35 grid and smoothing factor $s = 3.6$ gave a fit with maximum error approx 0.088, 18 knots in the $x$ direction and 14 in the $y$ direction. The time required was 1.53 seconds for the whole computation. The grid creation part of the execution time was 1.39 seconds. The contour plot of the surface approximant are shown in Figure 28; as before the level curves of the spline approximant are dashed. We also present a plot of the spline surface in Figure 29.

Additional experiments with other test functions show similar results – the mesh data algorithm was significantly faster on all test functions. This is explained from a linear algebra point of view in section 8.2.2 of [Die93]. In simple terms, the problem that is inherently two-dimensional in the case of the first algorithm can be split into several one-dimensional problems of much less complexity when the input data is specified on a rectangular grid. One drawback of using pre-gridding is that individual weighting of the data values is no longer an option.
Figure 28: Contour plot for reconstruction of pulse function $P$ using the second (mesh data) algorithm.

Figure 29: Reconstruction of pulse function $P$ using the mesh data algorithm.
Thus, it seemed reasonable to state that pre-gridding the data and then using the mesh data algorithm was better than using the scattered data algorithm. However, we can do even better - just before finishing this thesis, I invented a much faster algorithm for the pre-gridding; as you may have noticed, the pre-gridding part dominated the execution time for the mesh data algorithm. The pre-gridding algorithm I had used throughout the tests was the simplest possible: for each grid point, scan through the array of sampled points and find the one that is closest to the grid point. This works well, but is not very fast, e.g. with a 60x30 grid and 1000 sampled points, $60 \times 30 \times 1000 = 180000$ distance computations would have to be made!

The new algorithm I invented does things the other way around - its outer loop goes through all the sample points and tries to assign each sample point to a nearby grid position. Then, the algorithm does a clean-up by interpolating a function value to those grid points that have not yet been assigned one. To do this clean-up, a label matrix is used to keep track of which grid positions have not yet been assigned a function value. The algorithm is described in pseudo-code below:

```pseudo-code
initialize the label matrix (make all grid positions unmarked)
for (each point in the array of sampled points)
{
    do all the following for the current sample point:
        compute a corresponding grid position
        store the sample point’s z value at that grid position
        make a mark in the label matrix
}
for (each grid position)
{
    if (not marked in label matrix)
    {
        interpolate to get a z value
    }
}
```

Because this new algorithm for pre-gridding was invented at such a late stage, and because the conclusions I needed for my master’s thesis were already in place, I have not re-run the huge batch of tests (of which the reader has seen only the most important ones) with the new pre-gridding. However, the speed-up was so great that it would seem a waste not to include at least a hint of the new results. With 1000 sample points (as before) and a 50x25 grid, the pre-gridding now took less than 0.01 seconds and the whole computation took 0.07 seconds! The reason I chose a sparser grid was that the fit was better than with the denser 70x35 grid used with the previous pre-gridding algorithm. I want to emphasize that the speed-up was tremendous with all grid sizes. Figures 30 and 31 show contour and surface plots for the spline obtained with the new pre-gridding algorithm. With a smoothing factor $\alpha = 1.9$, the maximum error was 0.148623, and the algorithm produced 18 knots in the $x$ direction and 14 in the $y$ direction. I would like to point out that the interpolation in the new pre-gridding is rather crude and that it can probably be improved to get even better results.
without sacrificing too much execution speed. There is also an abundance of gridding methods available in commercial software such as Matlab or downloadable from the Internet. One suggestion for further work could be to investigate if such gridding methods might be of use for the kind of problems described in this master's thesis.

Thus, the conclusion of the tests is that for situations and surface data similar to our test conditions, I recommend pre-gridding and then using the gridded data algorithm.
Figure 31: Reconstruction of pulse function \( P \) using the mesh data algorithm and the new, faster pre-gridding.
4 Using Legacy Fortran Code From C++

The description given here is very brief and tailored to the specific needs for my master's thesis project. A more extensive description of C++/FORTRAN issues can be found in [Arn].

So, what has to be done to use legacy FORTRAN code from C++? First, from the package described in [Arn], one has to include the header file fortran.h which contains the necessary definitions for macros emulating FORTRAN datatypes etc. Then, to call a FORTRAN subroutine, all that needs to be done is to convert the function name into lower case and add an underscore to it (this is caused by linker behavior under most UNIX systems), e.g. if the original name of the subroutine is BISPEV, the name you refer to it in a C++ program will be "bispev". When calling subroutines, parameters are always passed by reference in FORTRAN. This is one of the reasons why it is easier to call FORTRAN programs from C++ than from C; C++ has reference data types, while C has not. To call the subroutine bispev, we insert the function prototype into our C++ program:

```cpp
SUBROUTINE bispev_(
    REAL*, INTEGER&, REAL*, INTEGER*, REAL*, INTEGER*, REAL*,
    INTEGER*, REAL*, INTEGER*, REAL*, INTEGER*, INTEGER*,
    INTEGER*, INTEGER&);
```

and then all we have to do is to make the actual call:

```cpp
bispev_(
    tX, nX, tY, nY, c, kX, kY, xGrid, mX2, yGrid, mY2, z2, wrk2, lwrk2, iwrk2,
    kwrk2, ier2);
```

The package [Arn] is partially prepared to achieve portability between standard UNIX systems and Microsoft Windows. To do this, however, requires some external processing (see [Arn]). Since portability was unimportant in this case, some modifications to the file fortran.h were made to be able to call the FORTRAN code as easily as possible, as described above. The changes are shown below (the consist of “commenting out” the F77 macros.) The compiler was then called with F77_STUB_REQUIRED enabled on the command line.

```cpp
... part of fortran.h:
#define F77_STUB_REQUIRED
#ifdef F77_STUB_REQUIRED
    /* Typically, this branch is for Unix computers */
    
    // C++ stub functions:
    #define SUBROUTINE inline void
    #define INTEGER_FUNCTION inline INTEGER
    #define REAL_FUNCTION inline REAL
    #define LOGICAL_FUNCTION inline LOGICAL
    #define DOUBLE_PRECISION_FUNCTION inline DOUBLE_PRECISION

    // FORTRAN functions
    #define SUBROUTINE_F77 extern "C" void
    #define INTEGER_FUNCTION_F77 extern "C" int
    #define REAL_FUNCTION_F77 extern "C" float
    #define LOGICAL_FUNCTION_F77 extern "C" int
    #define DOUBLE_PRECISION_FUNCTION_F77 extern "C" double

```
/*

#define SUBROUTINE extern "C" void
#define INTEGER_FUNCTION extern "C" int
#define REAL_FUNCTION extern "C" float
#define LOGICAL_FUNCTION extern "C" int
#define DOUBLE_PRECISION_FUNCTION extern "C" double

*/
References


