ACCURATE ADAPTIVE CONTOUR FINDING USING C^1 DATA

by

JOSHUA LOUIS LEVENBERG

B.A. (Reed College) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

MATHEMATICS

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor John A. Strain, Chair
Professor James Demmel
Professor James F. O'Brien

Spring 2003

ACCURATE ADAPTIVE CONTOUR FINDING USING C^1 DATA

by
JOSHUA LOUIS LEVENBERG

Abstract

ACCURATE ADAPTIVE CONTOUR FINDING USING C^1 DATA

by

JOSHUA LOUIS LEVENBERG

Doctor of Philosophy in MATHEMATICS

University of California at Berkeley

Professor John A. Strain, Chair

This thesis develops accurate adaptive computational methods for finding the contours of C^1 functions of one, two, and three variables. The main tools employed are adaptive meshes, piecewise-cubic interpolation with Bézier splines, and robust zero-finding algorithms. G^2 cubic spline contours are produced in the two-variable case. We also introduce a simple new C^1 natural neighbor interpolant.

Professor John A. Strain
Dissertation Committee Chair

To my family and friends,

who supported me through this whole process.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I want to thank John Strain without whom this would never had been possible. I would also like thank Rebecca Middleton who very patiently helped me through the entire process.

Chapter 1

Introduction

Every summer, my family goes backpacking in the Sierra Nevada mountains. Since we do this only once a year, our main concern is how strenuous the hike will be. So we study our topographical (or *topo*) map (Figure 1.1) to see how many altitude lines we are going to cross on our chosen trail. A topo map is a contour map that shows lines of constant altitude. Every line crossed corresponds to an elevation gain or loss of forty feet. Many lines close together indicate that an area is very steep.

More generally, contour maps show lines of constant value, or *level sets*, for some function. In the case of a topo map, the function takes in latitude and longitude and outputs altitude. If we think of latitude as x, and longitude as y, and altitude as z, we have the equation $z = f(x, y)$. Given any particular altitude z, we have one equation with two variables. The solutions will be curves in the xy-plane. Finding these curves is harder than solving problems with the same number of equations as variables, where there are generally a finite number of discrete solutions.

For a function f of three variables, the contours will be surfaces, or *isosurfaces*, in three dimensions. For a vector-valued function of three variables, the contours will be space curves or points, depending on the dimension of the vector. The general problem of finding these level sets of functions is called *contouring*.

Contouring algorithms are tools that can be used to solve problems in many different domains. 3D contours are used to visualize 3D flows [108]. Contouring is used to solve computer aided design and modeling problems that arise in engineering and manufacturing at Boeing [48]. Contouring solves a wide variety of geometric problems [34]:

- Finding ray-traps (or bouncing billiard-ball paths); a sequence of n points on n curves that a light ray or billiard ball will bounce between forever.
- Determining the locus of points equidistant from three curves in a plane.
- Finding surface-surface bisectors representing the safest paths between dangerous regions.

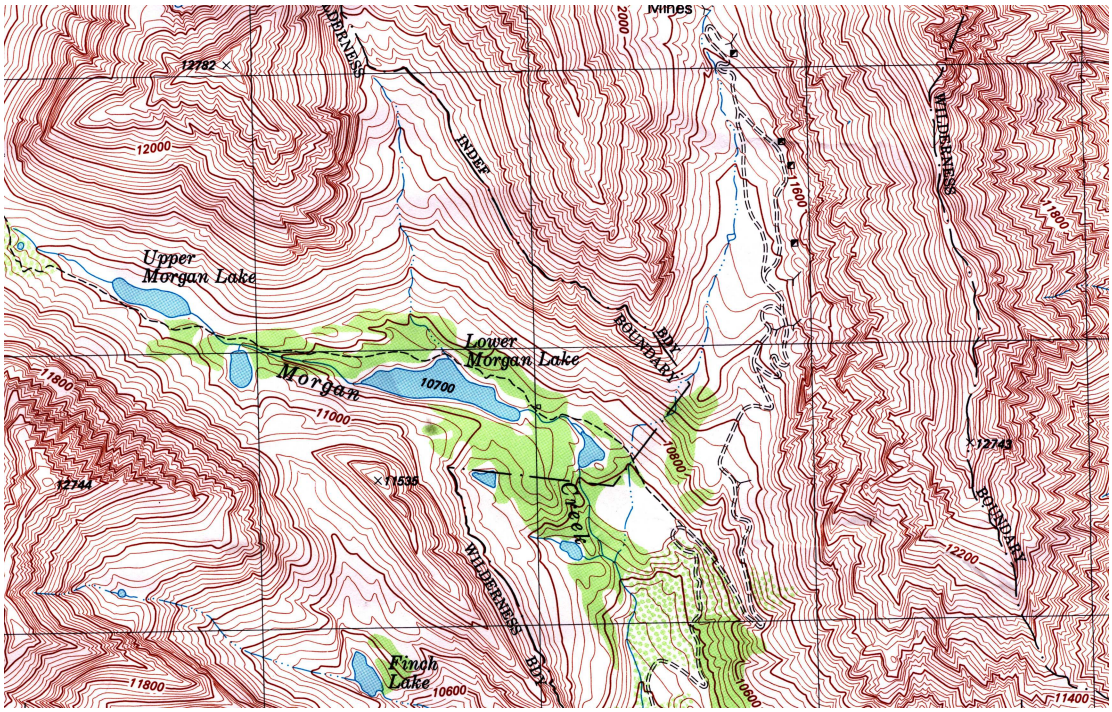


Figure 1.1: Part of a topographical map with contour lines every forty feet of altitude [101]

Often f is the (signed) distance to some surface: contouring then extracts the surface from the function. For example, the problem of simulating moving interfaces between fluid phases is complicated by changing topology. Evolving the distance function to the interface instead of the interface itself handles topological changes without surgery [21, 22, 69, 88, 100].

Distance functions also represent surfaces passing through a large collection of points [20], as in reconstructing a 3D sculpture from laser range-finding data. The Digital Michelangelo Project collects such data for various buildings and sculptures, most notably the *David*.

This thesis addresses the problem of contouring when:

- it is expensive to evaluate the function f,
- we can evaluate the function and its gradient ∇f at arbitrary points or estimate it reliably,
- we want to get to the correct answer within a specified tolerance, and
- the contours should not cross.

Some applications also require accurate tangent or curvature information along contours. Chapter 5 addresses interpolation schemes that build a contourable f from a fixed number of sample values.

1.1 Contributions

The focus of this thesis is on creating a contouring technique that addresses the above concerns and has the following collection of desirable properties:

- Accuracy: very small error bounds are achieved efficiently
- Adaptive: concentrates computational effort near the contour
- Scalable: all interpolation and contouring employs only local data
- Smooth: contours are continuous splines that do not cross and have no corners or cusps; can directly evaluate tangent and curvature of contour
- Robust: degenerate cases are carefully resolved to yield a consistent topology
- Modular: can substitute different interpolation, error estimation, adaptive meshes, etc.
- Stable: uses a spline basis throughout, avoiding (numerically unstable) conversion to the power basis [26, 38, 39]
- Freely usable: this technique is not patented, unlike Marching Cubes [23, 25]

The technique separates naturally into three stages:

1. Sample the function f using an adaptive mesh.
2. Refine the mesh until a cubic spline approximation $\tilde{f}$ is sufficiently close to f.
3. Use spline techniques to contour $\tilde{f}$.

Chapters 2, 3, and 4 describe contouring functions of 1, 2, and 3 variables respectively.

Using uniform sampling and linear interpolation, we would expect $O(1/\sqrt{n})$ error with n samples in 2D. With adaptive sampling, we can get $O(1/n)$ error. With cubic interpolation, we can get $O(1/n^2)$ or $O(1/n^3)$ error. Cubic interpolation also provides valuable curvature information and tangent data.

Greater accuracy can be obtained, though with reduced speed and scalability, using global approaches to interpolation. This is helpful when additional function values are costly or impossible to obtain. Chapter 5 contains a summary of some existing global interpolation techniques and presents a new interpolant which may be used in that situation.

Alternatively, we can get greater speed with much less accuracy using adaptive linear interpolation [17, 56, 100, 107, 115]. Using the modular framework, linear interpolation is compared to cubic interpolation in Chapters 2 and 3.

It has been observed [26, 38, 39] that polynomial operations performed in the Bernstein-Bézier basis are more numerically stable than in the power basis. Furthermore, conversion from the

power basis to the Bernstein-Bézier basis is *ill-conditioned*: this conversion magnifies small errors in the input [30]. The algorithms and implementations described in this dissertation all work in the Bernstein-Bézier basis from the start, and perform no conversions.

1.2 Definitions and Notation

A function f is said to be C^k if the k^{th} derivative of f is defined and continuous on the domain of f. For example, a C^0 function is continuous, and a C^1 function has a continuous derivative.

The following conventions will be used below:

f is the function to be contoured.

$\tilde{f}$ is the current approximation of f.

D is the domain of f.

C is the contour, $f^{-1}(0) = \{x \in D | f(x) = 0\}$.

ϵ is an error tolerance for the contour.

$\binom{n}{r}$ is the number of ways of choosing r items from n, given by $\frac{n!}{(n-r)!r!}$.

$\binom{n}{r_1\ r_2\ r_3}$ is the number of ways of dividing n items into three groups with sizes r_1, r_2, and r_3, given by $\frac{n!}{r_1!r_2!r_3!}$. Here $r_1 + r_2 + r_3 = n$.

$B_i^d(t)$ is the degree-d Bernstein-Bézier basis function of one variable, defined in Section 2.2.

$B_{ijk}^d(s,t,u)$ is a triangular Bézier basis function in 2D, defined in Section 3.2.

$c(t)$ is a curve, parametrized by a function $c : [0,1] \to \mathbf{R}^n$ where n is 2 or 3.

$\vec{\mathbf{d}}$ will represent a unit direction vector in $\mathbf{R}^3$.

Algorithms are presented in a pseudocode employing the following conventions:

- Comments begin with a #.
- $[a, \ldots, z]$ represents a list.
- Sets are enclosed in curly brackets $\{\ldots\}$.
- If a is a list, $a[n]$ is used to get the nth element.
- Functions and procedures are written in SmallCaps.
- Variables are written in *italics*.
- Control keywords are written in **bold face**.

- Blocks of code are indicated by indentation.
- The equal sign (=) is used for assignment, the keyword "**is**" is used to test for equality.
- Named members of a variable are accessed using a period. For example, Algorithm 3.1 has a variable called *State* that contains several variables and functions. The member functions may read or modify the member variables.

An algorithm is either a **function**, if it returns a value, or a **procedure**, if it does not. The other keywords that control the flow of execution are:

while *condition*:
    indented statements

executes the indented statements if the condition is true. After the indented statements have been executed, the condition is tested again and the process repeats. **continue** interrupts the indented statements to start over testing the condition. **break** exits this loop.

for *variable* **in** *list or set*:
    indented statements

sets *variable* to each element of the list or set, executing the indented statements each time. **continue** advances to the next element and continues with the indented statements. **break** exits the loop.

if *condition 1*:
    indented statements 1
else if *condition 2*:
    indented statements 2
else:
    else statements

executes the first set of statements if the first condition is true, the second set of statements if the second condition is true, and the *else statements* if none of the conditions are true. Note that the **else if** and **else** clauses are optional.

Finally, **return** exits from a procedure or function. The return value of a function follows the keyword **return**.

Chapter 2

Finding the roots of a C^1 function of one real variable

2.1 General approach

The overall algorithm for contouring 1D, 2D, and 3D functions is shown in Algorithm 2.1. It is a modular algorithm that uses a specified adaptive mesh, interpolant, and error model (which depends on the interpolant).

The algorithm consists of two main stages: computing an approximation $\tilde{f}$ to f, and contouring $\tilde{f}$. In the first stage, it discards regions where f has no zeros and evaluates f where zeros are likely. The second stage uses the specific form of $\tilde{f}$.

For functions of one variable, interpolation and adaptive sampling is described in Section 2.3, the error model and safe radius computations are described in Section 2.4, and the contouring of $\tilde{f}$ is described in Section 2.6.

2.2 Bézier splines

In the one-dimensional case, we use Bézier splines to represent $\tilde{f}$. Bézier splines represent piecewise cubic functions using a sequence of control points. The geometric relationship between the points and the function is extremely useful.

Bézier splines were independently developed by P. de Casteljau [29] and P. Bézier [16]. De Casteljau employed these splines in the computer-aided design of automobiles. Although de Casteljau discovered them first, Bézier was the first to publish, so they carry his name.

A univariate polynomial of degree d is specified by $d+1$ degrees of freedom. We represent such a polynomial by taking a linear combination of basis functions. For example, in the power basis the basis functions are 1, t, t^2, ..., t^d. A polynomial is determined by the coefficients c_i of the basis

```
procedure CONTOUR(Mesh, Function, Interpolant, ErrorModel, ε1, ε2):
while Mesh.WorkingSet not empty:
    Element = Mesh.WorkingSet.POP()
    SafeRegion = union_i BALL(Element.Vertices[i].Position, Element.Vertices[i].SafeRadius)
    if Element inside SafeRegion:
        Mesh.DISCARDELEMENT(Element)
        continue
    for Vertex in Element.Vertices:
        if Vertex.Value not set:
            Function.EVALUATEVALUE(Vertex)
    if Element.Diameter ≤ ε1+ε2:
        Mesh.ResultSet.PUSH(Element)
    else if ErrorModel.STOPPINGCRITERIA(Element,
                ErrorModel.MAXERROR(Element), ε1):
        Mesh.ResultSet.PUSH(Element)
    else if Element.Vertices.Value signs differ:
        Mesh.REFINE(Element)
    else:
        Mesh.INTIALIZECENTERVERTEX(Element)
        Radius = Mesh.RADIUSTOCOVERELEMENT(Element)
        Element.Center.SafeRadius = Function.SAFERADIUS(Element.Center, Radius)
        if Element.Center.SafeRadius ≥ Radius:
            Mesh.DISCARDELEMENT(Element)
        else:
            Mesh.REFINE(Element)
return Interpolant.CONTOURELEMENTS(Mesh.ResultSet, ε2)
```

Algorithm 2.1: Overall contouring algorithm

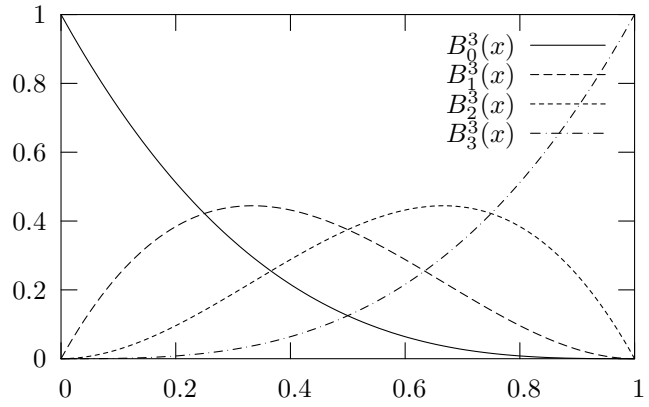


Figure 2.1: The cubic Bézier basis functons

functions:

$$c_0 \cdot 1 + c_1 \cdot t + \ldots + c_d \cdot t^d \,.$$

The basis for Bézier splines is the Bernstein polynomials [40], defined on $[0, 1]$ by (Figure 2.1):

$$B_i^d(t) = \binom{d}{i}(1-t)^{d-i}t^i \qquad \text{for } 0 \le i \le d \,.$$

Given $d+1$ numbers $c_0, \ldots, c_d$, called Bézier *ordinates*, we get the polynomial

$$c(t) = \sum_{i=0}^{d} c_i B_i^d(t) = \sum_{i=0}^{d} c_i \binom{d}{i}(1-t)^{d-i}t^i \,.$$

The basis functions give Bézier splines several desirable properties:

1. Only one of the basis functions is nonzero at the left or right endpoint:

$$B_i^d(0) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$B_i^d(1) = \begin{cases} 1 & \text{if } i = d \\ 0 & \text{otherwise} \end{cases}$$

   This implies that Bézier splines interpolate between the first and last ordinates: $c(0) = c_0$ and $c(1) = c_d$.

2. The Bernstein polynomials sum to one:

$$\sum_{i=0}^{d} B_i^d(t) = \sum_{i=0}^{d} \binom{d}{i}(1-t)^{d-i}t^i = ((1-t)+t)^d = 1 \text{ for all } t$$

This implies that the Bézier splines are *affine invariant*. That means that scaling and translating each coordinate is equivalent to scaling and translating the function:

$$\begin{aligned}\sum_{i=0}^{d}(a\cdot c_i+b)\,B_i^d(t) &= \left(\sum_{i=0}^{d}a\cdot c_i\cdot B_i^d(t)\right)+\sum_{i=0}^{d}b\cdot B_i^d(t)\\ &= a\left(\sum_{i=0}^{d}c_i\cdot B_i^d(t)\right)+b\left(\sum_{i=0}^{d}B_i^d(t)\right)\\ &= a\left(\sum_{i=0}^{d}c_i\cdot B_i^d(t)\right)+b\end{aligned}$$

3. The Bernstein polynomials are nonnegative on the interval $[0,1]$. Since they also sum to one, for every $0\le t\le 1$, $c(t)$ is a convex linear combination of the coefficients c_i. This implies $c([0,1])$ lies in $[\min_i c_i, \max_i c_i]$, so c has no roots on $[0,1]$ when all of the c_i have the same sign.

4. If we define $B_{-1}^d(t)=B_{d+1}^d(t)=0$, then the Bernstein polynomials satisfy the recurrence relation:

$$B_i^d(t)=(1-t)B_i^{d-1}(t)+tB_{i-1}^{d-1}(t)\qquad\text{for } 0\le i\le d.$$

Substituting into the expression for c:

$$\begin{aligned}c(t) &= \sum_{i=0}^{d}c_iB_i^d(t)\\ &= \sum_{i=0}^{d}c_i\left[(1-t)B_i^{d-1}(t)+tB_{i-1}^{d-1}(t)\right]\\ &= \sum_{i=0}^{d-1}\left(c_i(1-t)+c_{i+1}t\right)B_i^{d-1}(t)\end{aligned}$$

gives a recursive procedure, called *de Casteljau evaluation*, that evaluates c at any given t. The only operation it uses is linear interpolation, as shown in Figure 2.2:

$$\begin{aligned}d_0 &= (1-t)c_0+tc_1\\ d_1 &= (1-t)c_1+tc_2\\ d_2 &= (1-t)c_2+tc_3\\ d_3 &= (1-t)d_0+td_1\\ d_4 &= (1-t)d_1+td_2\\ d_5 &= (1-t)d_3+td_4\end{aligned}$$

De Casteljau evaluation is more numerically stable than evaluation in the power basis [26, 38, 39].

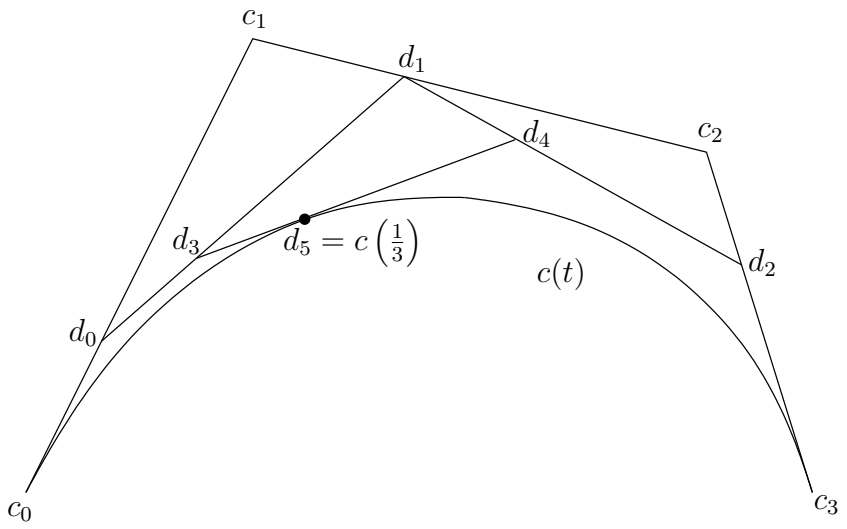


Figure 2.2: de Casteljau evaluation of a cubic ($d = 3$) Bézier spline at $t = \frac{1}{3}$.

Evaluating $c(t)$ using de Casteljau's algorithm requires $O(d^2)$ operations per evaluation point t. Farouki and Rajan [39] give substitutions to transform a Bézier spline into a polynomial that may be evaluated using Horner's method.

Bézier splines can also be used to represent space-curves $c(t) = [0,1] \to \mathbf{R}^n$. In this case, the coefficients c_i are vectors in $\mathbf{R}^n$, called *coordinates*. The term *control points* refers to either coordinates or ordinates. If we connect these adjacent coordinates c_i, c_{i+1} with line segments, we get the *Bézier polygon* for the spline. Since $c(t)$ is a convex linear combination of the control points c_i, $c([0,1])$ lies in the convex hull of its control polygon. An even stronger statement is that Bézier splines have the *variation diminishing property* [40]. That is, no hyper plane has more intersections with the curve than with the Bézier polygon.

Given ordinates for a Bézier spline, $c_i \in \mathbf{R}$, we can construct its graph $g(t) = (t, c(t)) : [0,1] \to \mathbf{R}^2$ as a Bézier spline with coordinates $g_i = \left(\frac{i}{d}, c_i\right)$. Applying the convex hull property to the graph of c gives more detailed information (such as where c could have roots) than to the ordinates c_i alone.

The derivative of a degree d polynomial is a degree $d-1$ polynomial. The derivative of a degree d Bézier spline can be easily represented as a degree $d-1$ Bézier spline: the derivative of $c(t)$ has control points $c'_i = d\,(c_{i+1} - c_i)$:

$$\begin{aligned} c'(t) &= \sum_{i=0}^{d} c_i \frac{d}{dt} B_i^d(t) \\ &= \sum_{i=0}^{d} c_i \binom{d}{i} \frac{d}{dt} \left((1-t)^{d-i} t^i\right) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=0}^{d-1} \left(c_i \binom{d}{i} (d-i)(-1)(1-t)^{d-i-1} t^i \right) \\
&\quad + \sum_{i=1}^{d} \left(c_i \binom{d}{i} i (1-t)^{d-i} t^{i-1} \right) \\
&= \sum_{i=0}^{d-1} \left[-c_i \binom{d}{i} (d-i)(1-t)^{d-i-1} t^i + c_{i+1} \binom{d}{i+1} (i+1)(1-t)^{d-(i+1)} t^i \right] \\
&= \sum_{i=0}^{d-1} \left[-c_i \binom{d}{i} (d-i) + c_{i+1} \binom{d}{i+1} (i+1) \right] (1-t)^{d-i-1} t^i \\
&= \sum_{i=0}^{d-1} \left[-c_i \frac{d!(d-i)}{i!(d-i)!} + c_{i+1} \frac{d!(i+1)}{(i+1)!(d-i-1)!} \right] (1-t)^{d-i-1} t^i \\
&= \sum_{i=0}^{d-1} \left[-c_i d \frac{(d-1)!}{i!(d-i-1)!} + c_{i+1} d \frac{(d-1)!}{i!(d-i-1)!} \right] (1-t)^{d-i-1} t^i \\
&= \sum_{i=0}^{d-1} d\,(c_{i+1} - c_i) \binom{d-1}{i} (1-t)^{d-1-i} t^i
\end{aligned}
$$

In particular, $c'(0) = d(c_1 - c_0)$ and $c'(1) = d(c_d - c_{d-1})$. Therefore the line segments c_0c_1 and $c_{d-1}c_d$ are tangent to the curve $c(t)$ at $t = 0$ and $t = 1$ respectively.

Splines are typically used to represent piecewise-polynomial curves, as in:

$$
p(t) = \begin{cases} a(t) & \text{if } 0 \le t \le 1 \\ b(t-1) & \text{if } 1 \le t \le 2 \end{cases}
$$

For p to be continuous, we need $a(1) = b(0)$ or $a_d = b_0$. For p to be C^1, we need $a_d = b_0$ and:

$$
\begin{aligned}
a'(1) &= b'(0) \\
d\,(a_d - a_{d-1}) &= d\,(b_1 - b_0) \\
a_d - a_{d-1} &= b_1 - b_0
\end{aligned}
$$

This type of continuity is referred to as *parametric continuity* since it depends on the parameterization of the curve.

Visual continuity or *geometric continuity* measures how smooth the curve is ignoring parameterization. A continuous curve is said to be G^1 if its unit tangent vector varies continuously. The function $p(t)$ above is G^1 continuous if $a_d = b_0$ (that is, p is continuous) and a_{d-1}, b_0, and b_1 are collinear. G^2 continuity requires continuous curvature

$$
K(t) = \frac{|c''(t) \times c'(t)|}{|c'(t)|^3}\,.
$$

Subdivision of a Bézier spline divides a spline $c(t)$ into two pieces, so that

$$
c(t) = \begin{cases} a\left(\frac{t}{t_0}\right) & \text{if } t \in [0, t_0] \\ b\left(\frac{t-t_0}{1-t_0}\right) & \text{if } t \in [t_0, 1] \end{cases}
$$

where a and b are Bézier splines with the same degree as c. The control points for a and b are a byproduct of de Casteljau evaluation of $c(t_0)$. For the case of cubic splines, using the notation from Figure 2.2:

$$a_0 = c_0 = c(0) \quad a_1 = d_0 \quad a_2 = d_3 \quad a_3 = d_5 = c(t_0)$$
$$b_0 = d_5 = c(t_0) \quad b_1 = d_4 \quad b_2 = d_2 \quad b_3 = c_3 = c(1)$$

The spline corresponding to an arbitrary subinterval of another spline is constructed by applying this procedure twice, once for each endpoint.

We will often employ cubic Bézier splines, since these can reproduce arbitrary positions and derivatives at their endpoints.

Bézier splines are presented in more detail in many standard texts [18, 55].

2.3 Adaptive cubic interpolation

Returning to the contouring problem, we want to find $f^{-1}(0)$ for some $f : D \to \mathbf{R}$ with $D \subset \mathbf{R}$ a closed interval. We will provide the CONTOUR procedure with inputs f, D, and two tolerances ϵ_1, and ϵ_2. We expect it to output points $x_1, \ldots, x_n$ within $\epsilon_1 + \epsilon_2$ of the roots of f on D.

CONTOUR begins with a mesh consisting of the single interval D. The endpoints of the interval are called vertices, and represent points at which we evaluate the function f and its derivative. The interior of the interval is called an element, and represents a region where we use the interpolant to construct $\tilde{f}$.

If the error model says that the error of approximation in a particular element is more than ϵ_1, that element will be refined. The mesh bisects the interval, inserting a new vertex at the midpoint. The error model is designed to indicate regions near roots which will need more subdivisions than other regions, resulting in an adaptive sampling of f.

Elements are divided into two sets: the *working set* and the *result set*. The working set consists of all elements that still require analysis. Elements are placed in the working set when they are created as part of subdivision. The working set initially contains only the interval D. The result set consists of all elements that do not need further refinement. These are the elements that are interpolated and contoured.

The interpolant takes the values and derivatives of f at the endpoints of each interval, and constructs the unique matching cubic Bézier spline. If we have two adjacent intervals, they share the value and derivative of f at their common point. Thus the piecewise cubic interpolant is C^1.

2.4 Error model

Since a cubic has four degrees of freedom, it is completely determined by two values and two derivatives. Therefore, the interpolant given above reproduces cubics exactly.

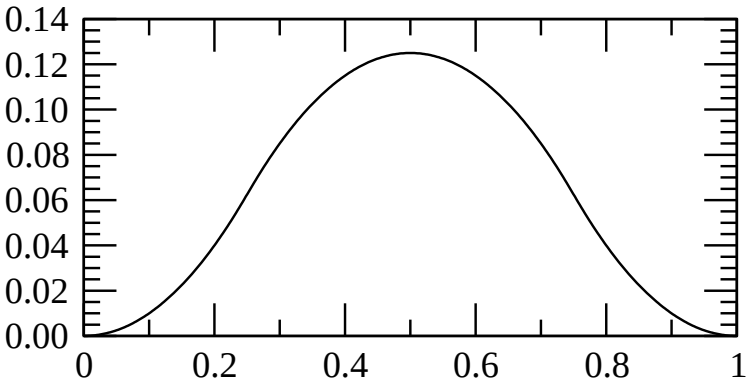


Figure 2.3: Maximum $g(x)$ for C^2 f with $K = 1$ and $h = 1$.

Let us assume that f is C^2. We would like to know the maximum error $\max_x |g(x)|$ where $g(x) = f(x) - \tilde{f}(x)$. Restricting our attention to a single element, we can, without loss of generality assume that the domain is $[0, h]$ and that f and f' agree with $\tilde{f}$ and $\tilde{f}'$ at $x = 0$ and $x = h$. Therefore, $g(0) = g(h) = g'(0) = g'(h) = 0$. Let $K = \max_x |g^{(2)}(x)|$. A straightforward computation shows that the largest possible value for $\max_x |g(x)|$ over all C^2 functions is achieved with

$$g(x) = \begin{cases} Kx^2 & \text{if } x \in \left[0, \frac{h}{4}\right] \\ \frac{Kh^2}{8} - K\left(x - \frac{h}{2}\right)^2 & \text{if } x \in \left[\frac{h}{4}, \frac{3h}{4}\right] \\ K(x-h)^2 & \text{if } x \in \left[\frac{3h}{4}, h\right] \end{cases}$$

as can be seen in Figure 2.3. Note that this g is not C^2, but it is the limit of C^2 functions. This $g(x)$ has a maximum of $\frac{Kh^2}{8}$, achieved at $x = \frac{h}{2}$.

This worst case error also applies in the case that g has Lipschitz derivative. If

$$|g'(x) - g'(y)| < K|x - y| \ ,$$

then again $|g(x)|$ has a maximum of $\frac{Kh^2}{8}$ between two samples.

For a smooth function f, we expect cubic approximation to accurately reproduce f once we have sufficiently resolved the major features of its graph. Adding more samples should cause K to decrease rapidly.

For example, consider a function f that is well approximated by a polynomial $\sum_{i=0}^{k} f_i x^i$ for $x \in [0, h]$. Again the error, $g(x) = (f - \tilde{f})(x)$, and the derivative of the error, $g'(x)$, will be zero at $x = 0$ and $x = h$. This time, however, $g(x)$ will be a polynomial of degree k. We can write a basis for the set of possible polynomials as $g_4(x)$, $g_5(x)$, ..., $g_k(x)$. Here the $g_i(x)$ have degree i, leading coefficient 1, satisfy $g_i(\{0, h\}) = 0$ and $g_i'(\{0, h\}) = 0$. Note that there are no polynomials with degree < 4 satisfying these conditions — 1D cubic approximation reproduces cubics polynomials exactly. These conditions uniquely determine $g_4(x) = x^2(x-h)^2 = x^4 - 2hx^3 + h^2x^2$. For $i > 4$, choose $g_i(x)$ to minimize the ∞-norm $\|g_i\|_\infty = \max_{x \in [0,h]} |g_i(x)|$ subject to the above conditions.

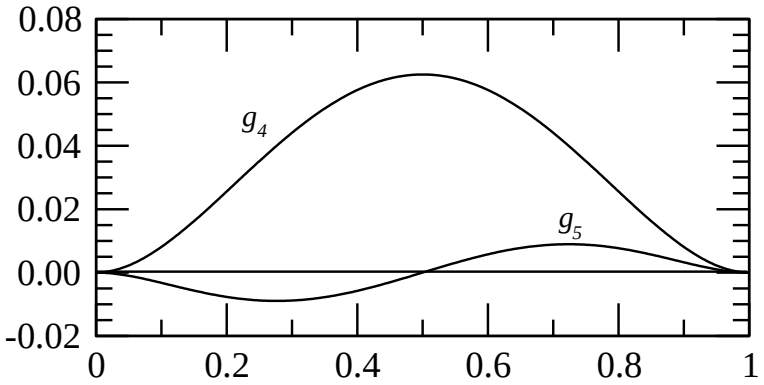


Figure 2.4: Basis for $g(x)$ for polynomial f with $h = 1$

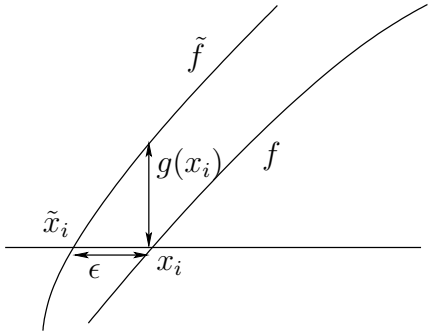


Figure 2.5: Interpolation error and slope determine the distance between the roots of $\tilde{f}$ and f.

For example,

$$g_5(x) = x^2 (x-h)^2 \left(x - \frac{h}{2}\right) = -\frac{h^3}{2}x^2 + 2h^2x^3 - \frac{5}{2}hx^4 + x^5 .$$

Both g_4 and g_5 are depicted in Figure 2.4. Writing g in this basis:

$$g(x) = c_4 g_4(x) + c_5 g_5(x) + \ldots + c_k g_k(x) .$$

By the triangle inequality

$$\|g\|_\infty \leq |c_4| \cdot \|g_4\|_\infty + |c_5| \cdot \|g_5\|_\infty + \cdots + |c_k| \cdot \|g_k\|_\infty .$$

We can directly compute the maximum absolute values of the g_i. The maximum of $|g_4(x)|$, $\frac{h^4}{16}$, occurs at $x = \frac{h}{2}$. The maximum of $|g_5(x)|$, $\frac{h^5\sqrt{5}}{250}$, occurs at $x = h\left(\frac{5\pm\sqrt{5}}{10}\right)$. Therefore the error is

$$\max_{x\in[0,h]} |g(x)| = |c_4| \frac{h^4}{16} + |c_5| \frac{h^5\sqrt{5}}{250} + O(h^6) .$$

For small h, a good estimate for a bound on $|g(x)|$ is $\frac{Kh^4}{16(4!)}$ where K is a bound on $f^{(4)}(x)$.

This $g(x)$ represents the *interpolation error* introduced by using $\tilde{f}$ instead of f. We would like to guarantee that the roots of $\tilde{f}$ are within ϵ_1 of the roots of f. Let x_i be a root of f and $\tilde{x}_i$ be the corresponding root of $\tilde{f}$. Applying the mean value theorem to $\tilde{f}$ on the interval $[\tilde{x}_i, x_i]$ gives

$$\begin{aligned} \tilde{f}'(c) &= \frac{\tilde{f}(x_i) - \tilde{f}(\tilde{x}_i)}{x_i - \tilde{x}_i} \\ &= \frac{g(x_i) - 0}{x_i - \tilde{x}_i} \end{aligned}$$

for some $c \in [\tilde{x}_i, x_i]$. This is illustrated in Figure 2.5. To insure $|x_i - \tilde{x}_i| \leq \epsilon_1$, it is sufficient to have

$$\epsilon_1 \geq \frac{\max_x |g(x)|}{\min_x \left|\tilde{f}'(x)\right|}.$$

This leads to a simple set of criteria for deciding whether to refine an element $[a, b]$ or contour it:

- Compute the interpolation error using, for example, $\frac{K|b-a|^4}{16(4!)}$.
- Compute the minimum of $\left|\tilde{f}'\right|$ on the interval $[a, b]$:
  - Compute the Bézier spline representing the derivative of $\tilde{f}$ using the technique from Section 2.2.
  - Find the convex hull of the ordinates of the derivative spline (an interval).
  - If the convex hull includes 0, refine the interval.
  - Otherwise, take the minimum of the absolute values of the endpoints of the convex hull.
- If the interpolation error divided by the minimum of $|\tilde{f}'|$ is less than ϵ_1, $\tilde{f}$ is a good approximation to f.
- Otherwise, refine the interval.

With the algorithm given above, an interval with a zero derivative will be subdivided until one of two things happens:

- the point with the zero derivative is separated from the root, or
- the interval shrinks to size $\approx \epsilon$.

A more complicated test can reduce the amount of subdivision and function sampling when there is a double root. Consider the case where $a \leq x_i, \tilde{x}_i, m \leq b$, $f(x_i) = 0$, $\tilde{f}(\tilde{x}_i) = 0$, $\tilde{f}'(m) = 0$, $\left|f(x) - \tilde{f}(x)\right| < E$ and $\tilde{f}''(x) \geq L > 0$ for all $a \leq x \leq b$. The case $\tilde{f}''(x) \leq L < 0$ is symmetric. Then, the worst case is when $\tilde{f}''(x) = L$ since, as we will see, a higher second derivative implies lower error. The maximum possible difference between f and $\tilde{f}$ is shown in Figure 2.6. Therefore the maximum possible difference $|x_i - \tilde{x}_i|$ (assuming f and $\tilde{f}$ have the corresponding roots) is when

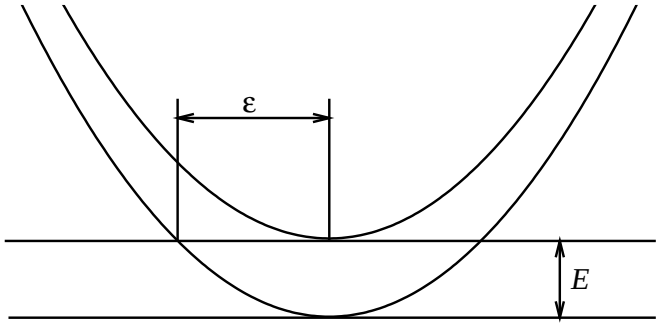


Figure 2.6: Worst case error when $\tilde{f}'(m) = 0$, $\tilde{f}'' > L$, and $\left|f - \tilde{f}\right| \leq E$.

$f(x) = L(x-x_i)^2$ and $\tilde{f}(x) = L(x-x_i)^2 - E$ (or vice versa). In this case, $\tilde{x}_i = x_i \pm \sqrt{\frac{E}{|L|}}$. Therefore, we can stop subdividing when $\sqrt{\frac{E}{|L|}} < \epsilon_1$.

In this section, our goal has been to insure that the interpolation error has not introduced more than ϵ_1 difference in the roots. The root-finding algorithm, in section 2.6, ensures that it finds all roots of $\tilde{f}$ to within ϵ_2. The total error is therefore bounded by $\epsilon_1 + \epsilon_2$. Increasing ϵ_1 decreases the number of samples of f. Increasing ϵ_2 decreases the number of iterations of the root-finding algorithm. Usually $\epsilon_1 > \epsilon_2$, especially if the function is time-consuming to evaluate.

The algorithm in Algorithm 2.1 uses *safe radii* to eliminate elements that have no roots from consideration. Given a vertex v, a radius r is said to be *safe* if $f(x) \neq 0$ for all $|x-v| \leq r$. The SafeRadius function takes the vertex v and a desired radius r_0, determines the maximum value D of the derivative of f within r_0 of v, and returns $\min\{f(v)/D, r_0\}$. Note that the return value will never be larger than r_0. However, if the return value is less than r_0, decreasing r_0 may increase the return value.

2.5 An alternate algorithm

An alternate approach is needed when bounds on the derivative are not available. Both the computation of interpolation error and the culling of elements with no roots are affected. An alternate contouring algorithm is shown in Algorithm 2.2.

Without some bound on the interpolation error, perhaps from a bound on $|f^{(4)}|$ or $\left|\left(f - \tilde{f}\right)''\right|$, we cannot be sure we have subdivided enough to insure the roots have deflected by less than ϵ_1. In this case, we make do with an estimate for the error. We compare the value and derivative of $\tilde{f}$ at the midpoint $x_{\frac{1}{2}}$ of the interval $[x_0, x_1]$ to the actual value and derivative of f at $x_{\frac{1}{2}}$.

On one half of the interval, the difference in the derivatives of f and $\tilde{f}$ serves to cancel out the error, but on the other half the errors reinforce. By restricting to the latter half, we define the error, $g(x)$, between f and $\tilde{f}$ on the interval $[0, r]$ with $r = \frac{x_1 - x_0}{2}$. Note that $g(x)$ has the following

```
procedure CONTOUR(Mesh, Function, Interpolant, ErrorModel, ε1, ε2):
while Mesh.WorkingSet not empty:
    Element = Mesh.WorkingSet.POP()
    for Vertex in Element.Vertices:
        if Vertex.Value not evaluated:
            Function.EVALUATE(Vertex)
    if Element.Diameter ≤ ε1+ε2:
        Mesh.ResultSet.PUSH(Element)
    else:
        Mesh.INTIALIZECENTERVERTEX(Element)
        Function.EVALUATE(Element.Center)
        Error = ErrorModel.ESTIMATEERROR(Element, Interpolant)
        if Element.Vertices.Value signs agree and
                    Interpolant.CULL(Element, Error):
            Mesh.DISCARDELEMENT(Element)
        else if ErrorModel.STOPPINGCRITERIA(Element, Error, ε1):
            Mesh.ResultSet.PUSH(Element)
        else:
            Mesh.REFINE(Element)
return Interpolant.CONTOURELEMENTS(Mesh.ResultSet, ε2)
```

Algorithm 2.2: Alternate contouring algorithm without derivative bounds

properties:

$$\begin{aligned} g(0) &= \left| f(x_{\frac{1}{2}}) - \tilde{f}(x_{\frac{1}{2}}) \right| \\ g'(0) &= \left| f'(x_{\frac{1}{2}}) - \tilde{f}'(x_{\frac{1}{2}}) \right| \\ g\left(r\right) &= 0 \\ g'\left(r\right) &= 0 \end{aligned}$$

These conditions define a cubic polynomial defined on the interval $[0, r]$, $\tilde{g}(x) = (ax + b)\left(\frac{x}{r} - 1\right)^2$, which we will assume is a good approximation of $g(x)$. Here a and b are functions of $g(0)$ and $g'(0)$. For our error estimate, we will take the maximum of $\tilde{g}$ in the interval $[0, r]$ and multiply it by some user-specified safety factor to account for the difference between $\tilde{g}$ and g. To find the maximum of $\tilde{g}(x)$, we set $\tilde{g}'(x) = 0$:

$$\begin{aligned} \tilde{g}'(x) &= a\left(\frac{x}{r} - 1\right)^2 + \frac{2}{r}(ax + b)\left(\frac{x}{r} - 1\right) \\ &= \left(\frac{x}{r} - 1\right)\left(a\frac{x}{r} - a + \frac{2}{r}ax + \frac{2}{r}b\right) \\ &= \left(\frac{x}{r} - 1\right)\left(\frac{3a}{r}x - a + \frac{2b}{r}\right) \\ x &= r \text{ or } \frac{a - \frac{2b}{r}}{\frac{3a}{r}} \end{aligned}$$

Since $x = r$ corresponds to the minimum in the interval, the maximum occurs at $x = \frac{ar - 2b}{3a}$.

We would like this error estimate to be conservative. To check that this procedure gives reasonable values, observe what happens when we apply it to $f(x) = Kg_4(x) = Kx^2(x - h)^2$ on $[0, h]$ from Section 2.4. Since $f(0)$, $f(h)$, $f'(0)$, and $f'(h)$ are all zero, the cubic interpolation gives $\tilde{f}\left(\frac{h}{2}\right) = \tilde{f}'\left(\frac{h}{2}\right) = 0$. The actual value at $x = \frac{h}{2}$ is $E = K\frac{h^4}{2^4}$ — though the derivative there is zero. The predicted error is the maximum of $e(x) = E\left(\frac{4x}{h} + 1\right)\left(\frac{2x}{h} - 1\right)^2$ on $\left[0, \frac{h}{2}\right]$. Using the terminology above, $r = \frac{h}{2}$, $a = \frac{4E}{h}$, and $b = E$. The root is at $x = 0$, so the maximum of $e(x)$ is $e(0) = E$. This is exactly the error.

Now we apply the same procedure to $f(x) = Kg_5(x) = Kx^2(x - h)^2\left(x - \frac{h}{2}\right)$ on $[0, h]$. Again the values and derivatives of f at the endpoints are zero, so the predicted value of f and its derivative at the midpoint is zero as well. In this case the value of f is zero, but the derivative is $D = \frac{Kh^4}{2^4}$. The estimated error is the maximum of a cubic $e(x)$ whose value is 0 at 0 and $\frac{h}{2}$ and whose derivative is D at 0 and 0 at $\frac{h}{2}$. In this case, $e(x) = Dx\left(\frac{2x}{h} - 1\right)^2$. So $r = \frac{h}{2}$, $a = D$, and $b = 0$, and the maximum is at $x = \frac{h}{6}$. Plugging in to $e(x)$ we get an error estimate of $\frac{2Dh}{27} = \frac{2Kh^5}{432} \approx 0.0046296Kh^5$. Compare this with the actual error $\frac{Kh^5\sqrt{5}}{250} \approx 0.0089443Kh^5$, and we see that we need to make some allowance to avoid under-estimating the error. We could either:

1. have a safety factor of at least 2,

2. conservatively estimate the error as $|\Delta f| + |\Delta f'| \frac{2h}{27}$, or

3. multiply the difference between the derivatives of f and $\tilde{f}$ by 2 before making the estimate.

We use the third option since it most accurately estimates the error. Note that the error term from the difference in derivative has a small coefficient that shrinks with h, so it will tend to be negligible unless the difference in value happens to be very small.

Once the error has been determined, we can sometimes determine that f has no zeroes on a particular element. We simply take the spline representation for $\tilde{f}$ and see if the convex hull of the ordinates is within the error margin of zero. If not, the element need not be considered further. This costs more computations than the safe radius method described above, but it can be more accurate and may save evaluations of f.

2.6 Finding the zeros of piecewise-cubic functions of one real variable

Finding the roots of polynomials is a very old problem [99, 92]. Many iterative methods [99], including for example the Newton method, can find one root of a general function to high accuracy, starting from a good estimate of its location. These techniques can efficiently increase the accuracy of a root once it has been located, but do not guarantee location of *all* the roots. For polynomial functions, special techniques such as Sturm sequences [99] and resultants [4, 61] can be used to determine all the roots, but operate in the power basis rather than the Bézier basis.

A simple robust divide-and-conquer technique for finding the roots of a spline is *bisection*. We start with the Bézier ordinates for the spline over the whole interval. If the ordinates are either all positive or all negative, then there are no roots and we are done. Otherwise, we divide the interval into two equal pieces, and compute the ordinates for the spline restricted to those two pieces. The ordinates for each piece are tested, and either discarded or subdivided. This process continues until the interval being tested is smaller than the error tolerance, ϵ_2. In most cases, only the pieces containing roots are subdivided. Each subdivision reduces our error in half, adding one binary digit to our answer. Thus, bisection gives linear convergence.

An alternative technique is called the *interval Newton method* [47]. In this technique, the Newton method is applied to intervals representing the domain and derivative of the function. The resulting intervals are intersected with the domain and the procedure is repeated. This technique converges quadratically to roots — the number of correct digits doubles in every iteration.

We have implemented a third technique developed by Sherbrooke and Patrikalakis [90]. This method converges quadratically, and has a generalization to higher dimensions. A description of this technique is given in Section 2.6.1.

```
function RootsOfSpline(Ordinates, Threshold, ϵ):
WorkList = [ ConvexIntersection(Ordinates) ]
ResultList = [ ]
while WorkList not empty:
    CurrentInterval = WorkList.Pop()
    CurrentOrdinates = SubSpline(Ordinates, CurrentInterval)
    IntersectionInterval = ConvexIntersection(CurrentOrdinates)
    if IntersectionInterval not empty:
        NewInterval = CurrentInterval.SubInterval(IntersectionInterval)
        if NewInterval.Length() < 2*ϵ:
            ResultList.Push(NewInterval.MidPoint())
            # factor out root here
        else if IntersectionInterval.Length() > Threshold:
            # probably have two distinct roots
            WorkList.Push(NewInterval.SubInterval([0,.5]))
            WorkList.Push(NewInterval.SubInterval([.5,0]))
        else:
            WorkList.Push(NewInterval)
return ResultList
```

Algorithm 2.3: Roots of a spline in 1D.

All of these techniques work directly with polynomial splines of any degree and are guaranteed to locate all roots in the interval in exact arithmetic. We chose the technique by Sherbrooke and Patrikalakis since it is easy to understand how it works, it extends to multiple dimensions, and it was the choice of Grandine and Klein [49].

2.6.1 A spline root finder using convex hulls

The basis of the Sherbrooke-Patrikalakis root finding algorithm is the graph convex hull property of Bézier splines described in Section 2.2. Given any Bézier spline $\tilde{f} : [0,1] \to \mathbf{R}$ with ordinates $\{b_0, b_1, \ldots, b_n\}$, its graph $G = \left\{ \left(x, \tilde{f}(x) \right) | x \in [0,1] \right\}$ lies entirely in the convex hull of the control points $\left\{ (0, b_0),\ (\frac{1}{n}, b_1),\ \ldots,\ (1, b_n) \right\}$. Therefore, the roots of $\tilde{f}$ must lie inside the intersection of this convex hull with the x-axis. The algorithm alternates computing the intersection of the x-axis with the convex hull of the control points, giving a new interval, with computing the ordinates corresponding to $\tilde{f}$ restricted to that new interval.

If the new interval is more than *Threshold* times the length of the old interval, then the

algorithm falls back to using bisection. This usually happens when the interval contains two distinct roots. Without bisection the interval must always contain both and cannot shrink. *Threshold* is generally between 0.6 and 0.8.

Pseudocode for the algorithm is in Algorithm 2.3. The functions MidPoint, Length, and SubInterval are defined, given intervals x, y, by:

x.MidPoint()=(x.begin+x.end)/2
x.Length()=(x.end-x.begin)
x.SubInterval(y)=[x.begin+y.begin*x.Length(), x.begin+y.end*x.Length()]

The function SubSpline(*Ordinates*, *SubInterval*) returns the ordinates of the spline with ordinates *Ordinates* on $[0, 1]$ restricted to the subinterval *SubInterval*, as described in Section 2.2. The algorithm always subdivides the original interval to avoid accumulation of errors.

This algorithm converges quadratically to roots as long as f intersects the x-axis transversely. At double roots, we get linear convergence.

2.6.2 Robustness

We employ two strategies to ensure all roots were found. First, before applying the convex-hull-intersection algorithm (described in Section 2.6.4), we replace each ordinate by a vertical segment ϵ_R tall. This prevents small errors from making tiny ordinates miss the x-axis. Second, if an interval ever shrinks to smaller than ϵ_2, we enlarge it to be exactly ϵ_2 wide. When an interval gets very small (much smaller than ϵ_2, typically), small errors can cause the interval to miss an actual root.

Here ϵ_R does not affect the accuracy of the roots x_i. It is present to avoid making incorrect decisions in the presence of floating point errors. ϵ_R can be much smaller than ϵ_1 or ϵ_2. Note that ϵ_R is applied to values in the range of f and $\epsilon_{1,2}$ are applied to values in the domain.

In order to disambiguate the case where two roots are very close together, we can apply *deflation*: we factor out any roots that we have found before continuing. In higher dimensions, deflation is complicated.

2.6.3 Deflation in the Bézier basis

Assume we have a degree n Bézier spline $\tilde{f} : [0, 1] \to \mathbf{R}$ with ordinates $\{b_0, \ldots, b_n\}$, and $x_0 \in [0, 1]$ where $f(x_0) = 0$. We want to compute the spline ordinates for $\tilde{f}(x)/(x - x_0)$. Since $\tilde{f}(x)$ is a polynomial with a zero at x_0, we know $\tilde{f}(x)$ has $x - x_0$ as a factor. Dividing two polynomials in the Bernstein basis in general requires solving a linear system, which is much more expensive than the synthetic division algorithm applied in the power basis. However, we have derived a simple formula for the case where the root is at one of the endpoints of the interval.

We can force the root to be at an endpoint by first subdividing the spline at x_0 with de Casteljau's algorithm from Section 2.2. This results in two splines: one covering $[0, x_0]$ and one

covering $[x_0, 1]$:

$$\begin{aligned} \text{Left half} \quad & f_l(t) : [0,1] \to \mathbf{R} \quad f_l(t) = \tilde{f}(t \cdot x_0)\,, \\ \text{Right half} \quad & f_r(t) : [0,1] \to \mathbf{R} \quad f_r(t) = \tilde{f}(x_0 + t(1-x_0))\,. \end{aligned}$$

Then $x - x_0$ can be factored out of both.

The functions f_l and f_r are defined by the Bézier ordinates $\{l_0, \ldots, l_n\}$ and $\{r_0, \ldots, r_n\}$ with $l_0 = b_0$, $r_n = b_n$, and $l_n = r_0$. Since we subdivided at a root, l_n and r_0 are in fact both 0, and these values need not be computed. Dividing by $x - x_0$ gives us splines of degree $n - 1$.

$$\begin{aligned} l'_0 &= -\frac{l_0}{x_0} \\ l'_1 &= -\frac{n}{n-1}\frac{l_1}{x_0} \\ l'_2 &= -\frac{n}{n-2}\frac{l_2}{x_0} \\ &\vdots \\ l'_{n-1} &= -\frac{n}{1}\frac{l_{n-1}}{x_0} \\ r'_0 &= \frac{n}{1}\frac{r_1}{1-x_0} \\ r'_1 &= \frac{n}{2}\frac{r_2}{1-x_0} \\ &\vdots \\ r'_{n-1} &= \frac{n}{n}\frac{r_n}{1-x_0} \end{aligned}$$

Note that $l'_{n-1} = r'_0$ should hold. These will be zero if $\tilde{f}$ has a double root at x_0. If we only care about finding roots we can leave off the $\frac{n}{x_0}$ and $\frac{n}{1-x_0}$ terms: these only scale the function and do not change the roots. This conveniently avoids problems in the cases $x_0 = 0$ and $x_0 = 1$.

We can then proceed to find roots on either side of x_0 with these new splines. If *WorkList* is a queue, roots will be visited in left-to-right order. As a result, we need only the right hand sub-spline after factoring out a root. This algorithm does not seem to have been described in the literature. Here is a derivation of the r'_i formula (the only one we use).

Theorem 1 *If $\tilde{f} : [0,1] \to \mathbf{R}$ is a degree $n > 0$ Bézier spline with ordinates $\{f_0, \ldots, f_n\}$ where $f_0 = 0$, then $\tilde{f}(x) = xg(x)$ where $g(x)$ is a Bézier spline with degree $n-1$ with ordinates $g_i = f_{i+1}\frac{n}{i+1}$ for $i = 0 \ldots n-1$.*

Proof: By definition of Bézier splines:

$$\tilde{f}(x) = \sum_{i=0}^{n} f_i \binom{n}{i} x^i (1-x)^{n-i}\,.$$

Since $f_0 = 0$, the first term in this sum is zero. Every other term has $i > 0$ and therefore contains x to a positive power. Factoring x out from the entire sum gives:

$$\tilde{f}(x) = x \sum_{i=1}^{n} f_i \binom{n}{i} x^{i-1}(1-x)^{n-i}\,.$$

Then we may re-write the sum as a degree $n-1$ Bézier spline thus:

$$\begin{aligned}
\sum_{i=1}^{n} f_i \binom{n}{i} x^{i-1}(1-x)^{n-i} &= \sum_{j=0}^{n-1} f_{j+1} \binom{n}{j+1} x^{j}(1-x)^{n-(j+1)} \\
&= \sum_{j=0}^{n-1} f_{j+1} \frac{\binom{n}{j+1}}{\binom{n-1}{j}} \binom{n-1}{j} x^{j}(1-x)^{(n-1)-j} \\
&= \sum_{j=0}^{n-1} f_{j+1} \frac{n}{j+1} \binom{n-1}{j} x^{j}(1-x)^{(n-1)-j} \\
&= \sum_{i=0}^{n-1} g_i \binom{n-1}{i} x^{i}(1-x)^{(n-1)-i}\,.
\end{aligned}$$

■

When $x_0 \neq 0$, we need to divide out $x - x_0$ from f_r. Over the interval $[x_0, 1]$, this factor goes linearly from 0 to $1 - x_0$. Dividing by $1 - x_0$ returns us to the case where we are factoring out a term going linearly from 0 to 1.

2.6.4 Convex hull intersection

The convex-hull-intersection algorithm should, given four vertical intervals $I_0 = (0, [y_{00}, y_{01}])$, ..., $I_3 = (1, [y_{30}, y_{31}])$, compute the intersection of the convex hull of $I_0, \ldots, I_3$ with the x-axis.

We divide this into 9 possibilities. The first segment is either above, below, or crossing the x-axis. The last segment can be in either of those three positions as well. These 9 possibilities fall into the 4 cases, shown in Figure 2.7. In all cases, we either know an endpoint directly (when the first or last segment crosses the x-axis) or can determine it using only half of the input data — either the bottoms or tops of the vertical segments:

1. If both I_0 and I_3 cross the x-axis, the intersection is the whole interval.

2. If one crosses the x-axis and the other is above or below the x-axis (4 possibilities), we know one endpoint and have to find the other. In the example shown in Figure 2.7, we know the left endpoint and we need only look at the bottoms of the vertical line segments to determine the right endpoint.

3. If one is above the x-axis and the other below the x-axis (2 possibilities), one endpoint (the left endpoint in the example given in Figure 2.7) comes from the tops of the vertical line segments, the other from the bottoms.

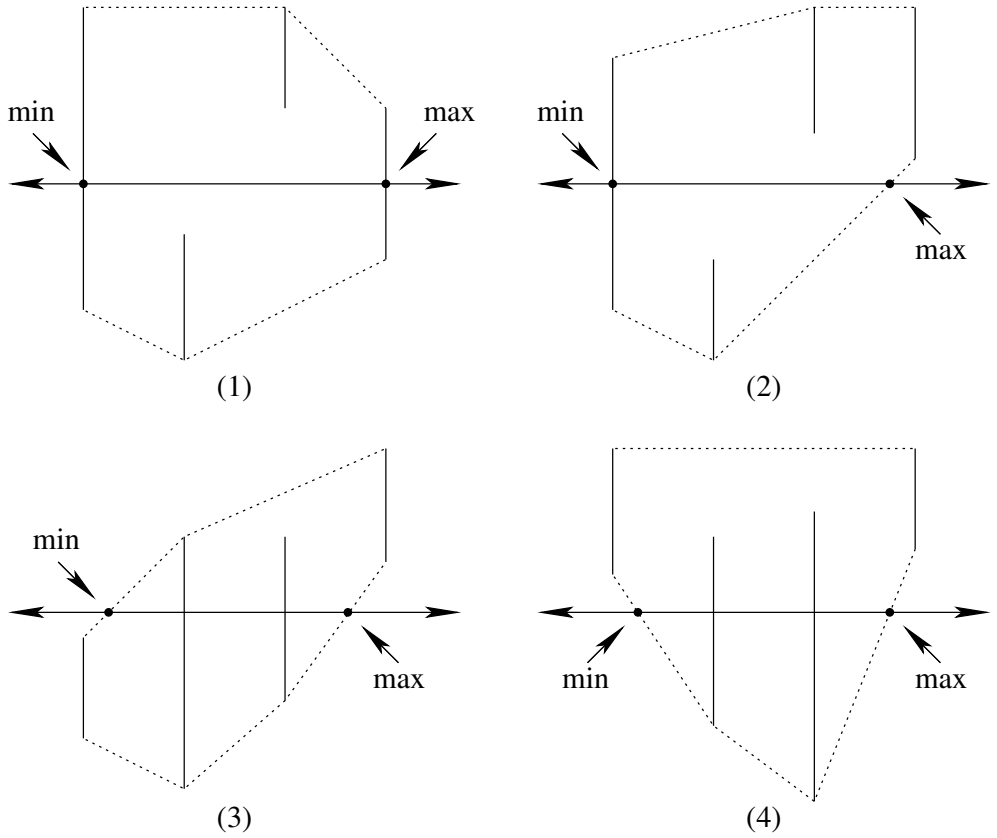


Figure 2.7: Examples of cases 1–4 for finding the intersection of the convex hull (dotted lines) of four vertical line segments with the x-axis.

4. If both segments are above the x-axis or both are below the x-axis (2 possibilities), then we only need to look at the bottoms (as in the figure) or tops of the vertical line segments. In this case, there may be no intersection at all.

Since the problem is small, we can resolve these cases with brute force. For each possible pair of the four segment endpoints, we intersect the corresponding line segments with the x-axis. As an optimization, if we are trying to find the left endpoint, we can ignore any segment strictly to the right of any intersection we have already located.

For case 4, there is an additional optimization that is especially helpful if there are more than four vertical intervals (if, for example, we are dealing with a polynomial of higher degree). Assume that there are sign changes, so the intersection is non-empty. We segment the points we are considering into four sets:

1. the points before the first sign change,
2. the points between the first and second sign changes,
3. the points between the last two sign changes, and
4. the points after the last sign change.

If there are only two sign changes (such as in Figure 2.7(4)), sets 2 and 3 will be the same. We then consider only lines going through one point from each of the first two sets or each of the last two sets.

Contouring in 2D and 3D (Chapters 3 and 4) requires finding many roots of splines and therefore many convex hull intersections. The optimizations given above lead to a performance improvement in all of these cases.

2.7 Results

We tested our contouring routines with $f(x) = \sin(100x^2)/(10x)$ (Figure 2.8), which has 32 roots on the interval $[0, 1]$. The error tolerance ϵ was set to values between 0.001 and 2×10^{-9}. These tests executed too quickly to time reliably.

The cubic methods required many fewer evaluations to get high accuracy (Figure 2.9). With the maximum derivative set to 20, 177 function and 125 derivative evaluations gave a maximum error of 5.5×10^{-7}. If instead the error was estimated using the sampling method of Section 2.5, that same error bound required 221 function and 221 derivative evaluations. Linear interpolation required 617 function evaluations to achieve a maximum error of 6.7×10^{-7}. The cubic method was also very scalable, reducing the error by a factor of 15 with 10% more evaluations. The linear method, in contrast, reduced the error by a factor of 5 with 10% more evaluations. In all cases, the average error was between one-half and one-quarter of the maximum error.

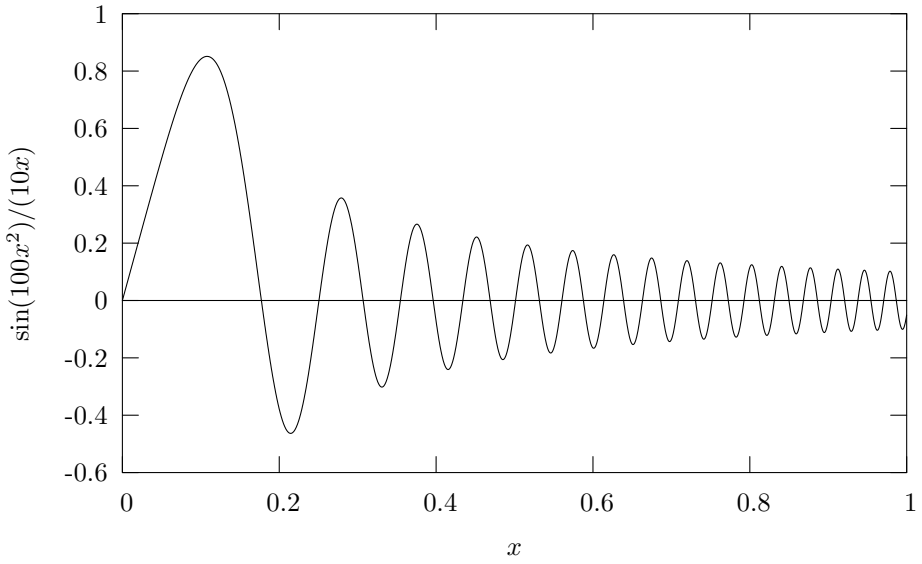


Figure 2.8: The test function $f(x) = \sin(100x^2)/(10x)$

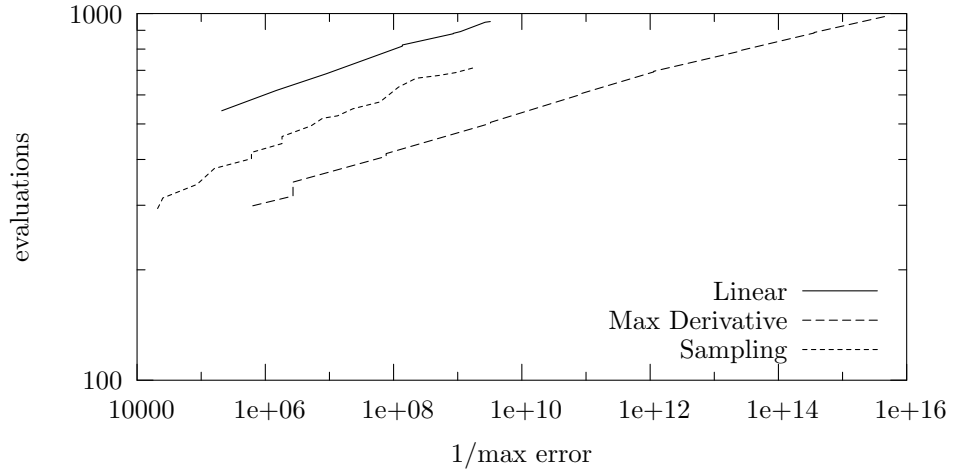


Figure 2.9: Total number of function and gradient evaluations needed to achieve the given maximum error

Chapter 3

Finding the contours of a C^1 function of two real variables

In the two dimensional case, that is $f : D \to \mathbf{R}$ where D is a compact subset of $\mathbf{R}^2$, the contours are curves in the plane. This is a much harder problem than the 1D case, and there are more choices for the interpolation and adaptive mesh schemes.

A flowchart of our approach is given in Figure 3.1. First, an error estimate guides adaptive sampling of the function. An approximation to the function is constructed by cubic interpolation of the samples. Each cubic patch is then contoured: the patch is divided into panels and a piecewise-cubic spline is fit to the contours within each panel. This step employs 1D and 2D root finding at several points. Finally, the contours from each cubic patch are connected to the contours of neighboring patches to yield a globally defined curve set.

3.1 Previous work

Previous 2D contouring schemes are either grid-based or gridless. Gridless methods query the function to follow individual contours. Grid methods assume the function value is known only at some prespecified grid points. If the function is available at arbitrary points, an adaptive grid can be used to reduce the number of function evaluations without loss of accuracy. There are also useful subsidiary techniques designed for contouring particular classes of functions such as quadratic polynomials.

Gridless methods [35] trace each contour continuously around the domain. Given the last two points computed on the contour, extrapolation steps forward to find points on the left and right sides of the contour. If the sign of the function differs at the two points, the next point on the contour is determined by linear interpolation, as in Figure 3.2. Otherwise the step size is cut in half and the process is repeated until a sign change is detected.

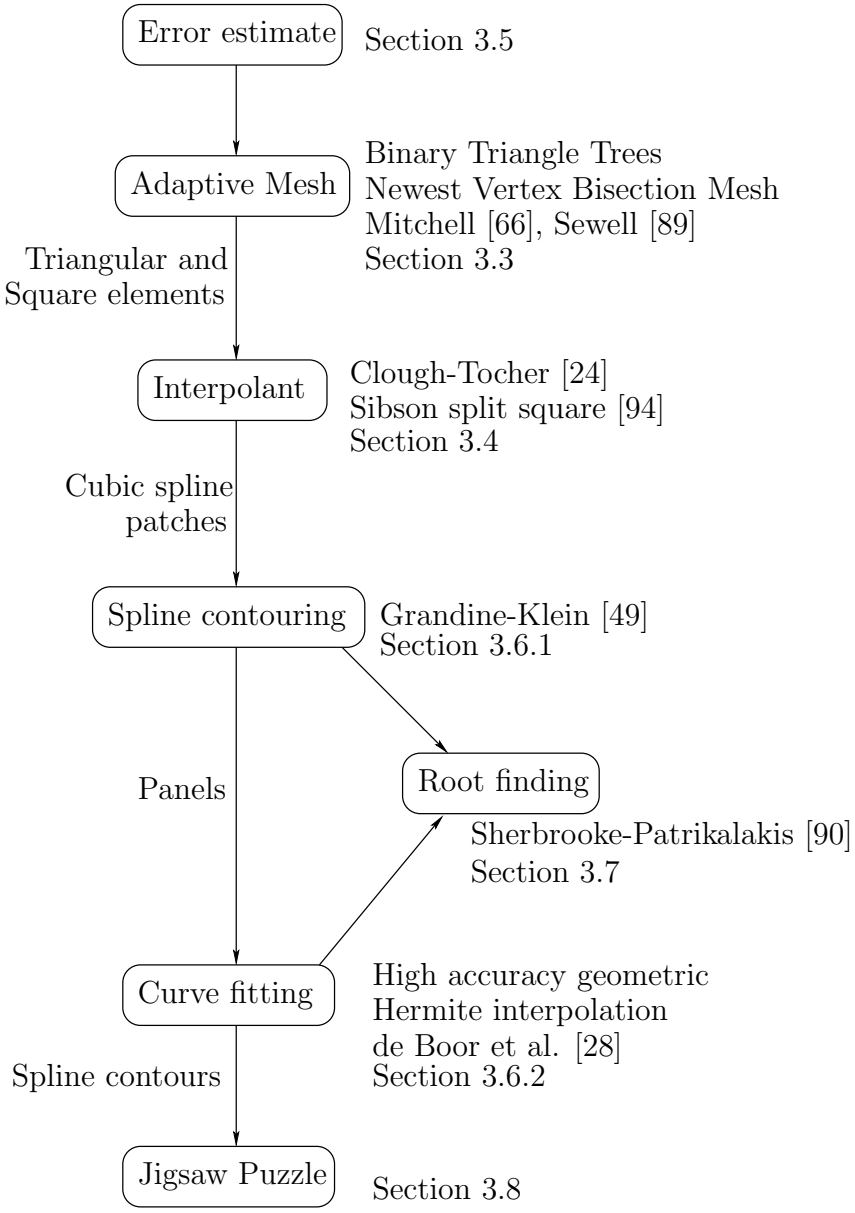


Figure 3.1: Data flow between modules of the 2D contouring algorithm

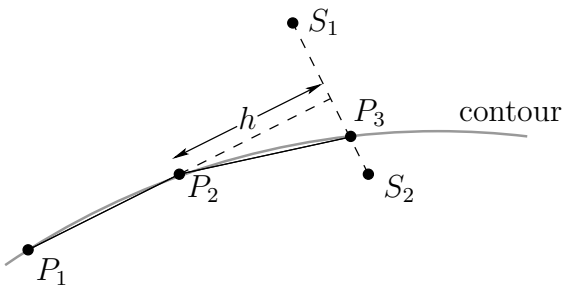


Figure 3.2: Given the last two points on the contour, P_1 and P_2, two sample points S_1 and S_2 are extrapolated. If $f(S_1)$ and $f(S_2)$ have different signs, then the next point on the contour, P_3, is found by linear interpolation. Otherwise the step size h is halved and the step repeated.

Typically the function is evaluated on some coarse grid to find points on each contour to initiate this process. Gridless methods are adaptive: they sample the function more densely in areas where the contours have high curvature. They may require many function evaluations, however.

Grid methods take a function that has already been evaluated on a set of points, such as a uniform rectangular grid or a triangular mesh. They typically use piecewise-linear interpolation to construct an approximation to the function that is easy to contour. The resulting contours may be smoothed as a post-process using a curve-fitting routine. Petersen [73] has summarized several grid methods.

Petersen [73] gives a particularly sophisticated grid method for scattered data. He first uses cubic interpolation to approximate f, and then used subdivision and degree reduction to build piecewise-linear contours. Peterson's algorithm:

- triangulates the initial positional data, minimizing the maximum interior angle;
- estimates the gradient at each vertex by Little's method [58];
- builds a piecewise-cubic Clough-Tocher interpolant [24] (see Section 3.4.3) to the known data;
- discards cubic patches that do not contain a piece of the contour;
- uses a lower degree approximation to the patch if it does not introduce too much error, and otherwise subdivides the patch;
- contours each patch when it is reduced to a linear spline, resulting in a single line segment.

Petersen's criteria for lowering the degree of a patch involves the minimum of the magnitude of the gradient at the corners of the patch.

Strain [100] uses restricted quadtrees to contour a moving interface. The adaptive nature of quadtrees allows most of the sampling to be concentrated near the contour.

All of the above techniques yield piecewise-linear (polygonal) contours. Techniques that produce higher-order contour representations typically only apply to functions represented in a particular spline basis. Grandine and Klein [49] have a contouring method applicable to functions given as tensor-product splines. Elber and Kim [34] have an approach for rational splines. Worsey and Farin [111] have a technique for contouring a bivariate quadratic polynomial over a triangle. Fu [43] and Preusser [77] considered the case of a bicubic Bézier patch.

To apply these techniques to an arbitrary function, that function must first be approximated with an appropriate spline. We use cubic interpolation to approximate the function. Each cubic piece is then contoured using a technique similar to that of Grandine and Klein [49]. This approach is modular; a different contouring technique could be employed as long as it was paired with a compatible interpolant.

3.2 Bézier patches

There are two natural generalizations of Bézier splines to 2D: *rectangular Bézier patches* and *triangular Bézier patches*. We will primarily use the latter, but a description of the former follows for completeness.

Rectangular Bézier patches, also known as *tensor product Bézier patches*, are defined on the unit square $[0,1]^2$. The set of degree-d tensor product patches is the span of all monomials where the exponents of x and y (in the power basis) are each at most d, $\sum_{i=0}^{d}\sum_{j=0}^{d} c_{ij}x^i y^j$. For $d=3$, these patches are called *bicubic* polynomials. A Bernstein basis for rectangular Bézier patches consists of all products of 1D Bernstein basis elements with degree at most d in x and y:

$$B^d_{i\otimes j}(x,y) = B^d_i(x)B^d_j(y)\,.$$

Ordinates $c_{ij} \in \mathbf{R}^n$ define $C(x,y) : [0,1]^2 \to \mathbf{R}^n$ by

$$\begin{aligned} C(x,y) &= \sum_{i=0}^{d}\sum_{j=0}^{d} c_{ij}B^d_{i\otimes j}(x,y) \\ &= \sum_{i=0}^{d}\sum_{j=0}^{d} c_{ij}B^d_i(x)B^d_j(y) \\ &= \sum_{i=0}^{d}\sum_{j=0}^{d} c_{ij}\binom{d}{i}(1-x)^{d-i}x^i\binom{d}{j}(1-y)^{d-j}y^j\,. \end{aligned}$$

These coordinates c_{ij} are naturally arranged in a square grid called the *control mesh* for the Bézier patch, see Figure 3.3(a).

Rectangular Bézier splines have the convex hull property and the variation diminishing property described in Section 2.2. They are affine invariant and pass through the four corner control points. Restricting C to the boundary of the domain yields a degree-d Bézier spline on each edge.

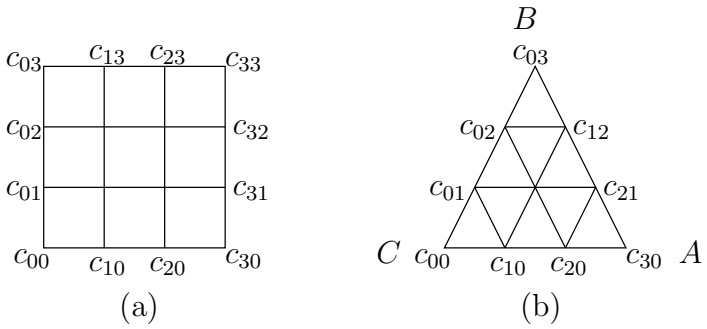


Figure 3.3: The control mesh for (a) rectangular and (b) triangular Bézier spline patches with degree $d = 3$

The control points for these boundary curves are a subset of the control points for the whole patch. For example, the restriction to $[0,1] \times \{0\}$ has control points $c_{00}, \ldots, c_{d0}$.

Triangular Bézier patches have a *total degree* of d. That means that the *sum* of the exponents of x and y in each monomial is at most d. In the power basis, triangular Bézier patches have the form

$$\sum_{i,j \geq 0,\ i+j \leq 3} c_{ij} x^i y^j \, .$$

By stars and bars [81], there are $\binom{d+2}{2}$ coefficients c_{ij}. The argument is as follows: consider d stars which we want to divide into three bins labeled i, j, and k. There are $\binom{d+2}{2}$ ways of lining up d stars and 2 bars. If $d = 3$, one way would be:

$$\star\star \,|\, \star \,|\, \, .$$

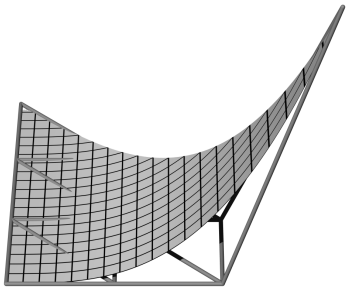
Let the number of stars before the first bar be i (2 in the example), the number between the two be j (1 in the example), and the number after the last bar k (0 in the example). In this way, we identify each configuration of stars and bars with a distinct element of the power basis $x^i y^j 1^k = x^i y^j$, $i, j \geq 0$, $i + j \leq d$.

Given a triangle defined by vertices $A, B, C \in \mathbf{R}^2$, any point inside the triangle can be uniquely represented as $sA + tB + rC$ where $s, t, u \in [0,1]$ and $s + t + u = 1$. These *barycentric coordinates* are the natural parameters for a triangular Bézier patch. The basis functions are indexed by $i, j, k \geq 0$ with $i + j + k = d$:

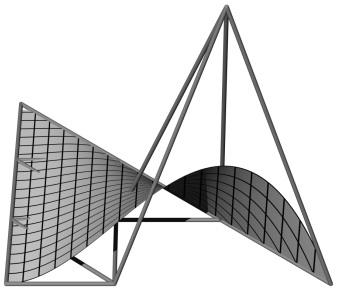
$$B^d_{ijk}(s,t,u) = \binom{d}{i\ j\ k} s^i t^j u^k \, .$$

Here $\binom{d}{i\ j\ k} = \frac{d!}{i!j!k!}$. These control points are form a triangular grid, as in Figure 3.3(b). Three representative basis functions are shown in Figure 3.4.

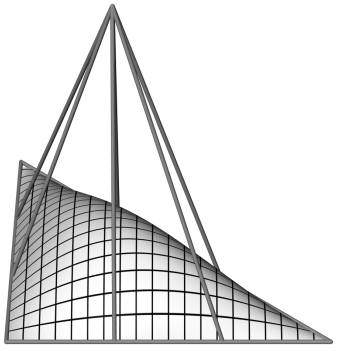
Triangular Bézier patches also have the convex hull and variation diminishing properties of Section 2.2. They are affine invariant and restriction to the boundary yields three 1-dimensional



(a) $B^3_{300}(x, y, 1 - x - y)$



(b) $B^3_{201}(x, y, 1 - x - y)$



(c) $B^3_{111}(x, y, 1 - x - y)$

Figure 3.4: Basis functions for triangular Bézier patches, with the corresponding control polygon.

degree-d Bézier splines. The control points for these boundary curves are the control points along the boundary of the patch. Since the total degree is d, the restriction to any line segment is a degree d polynomial in one variable.

We will typically work on the reference triangle with vertices $A = (1, 0)$, $B = (0, 1)$, and $C = (0, 0)$. In that case, $s = x$, $t = y$, and $u = 1 - x - y$, and the Bézier patch is

$$\sum_{i,j \geq 0, i+j \leq d} c_{ij} \binom{d}{i\ j\ d-i-j} x^i y^j (1-x-y)^{d-i-j}$$

using the substitution $k = d - i - j$. The partial derivative of this expression with respect to x (or y) is another Bézier patch with degree $d - 1$, and with control points given by $c'_{ij} = d\,(c_{i+1,j} - c_{ij})$ (respectively $c'_{ij} = d\,(c_{i,j+1} - c_{ij})$).

De Casteljau's algorithm generalizes to evaluate triangular patches. As in 1D, it takes d steps, and each step consists of several linear interpolations. Each step is an application of the *de Casteljau operator* which takes barycentric coordinates (s, t, u) with $s + t + u = 1$, plus the control points $[c_{ij}]_{i+j \leq d}$ for a degree $d \geq 1$ patch and produces control points $[c'_{ij}]_{i+j \leq d-1}$ for a degree $d-1$ patch, according to the formula:

$$c'_{ij} = s c_{i+1,j} + t c_{i,j+1} + u c_{ij}\,.$$

See Figure 3.5(a). Applying this operator d times with the same (s, t, u) results in a single control point with the value of the original patch at (s, t, u), as in Figure 3.5(b). Schumaker and Volk [87] give an alternative representation of Bézier patches that permits more efficient evaluation.

The intermediate values generated in this process can be used to subdivide a triangular patch into three sub-patches, as in Figure 3.5(c). These sub-patches reproduce the values of the original patch but on smaller domains, see Figure 3.5(d). This can be useful, but repeated application of this three-way split results in long, skinny triangles. It is possible, however, to use the de Casteljau operator to find the control points for an arbitrary triangle. Assume the new triangle T' has vertices v_0, v_1, v_2 located in the original triangle T at barycentric coordinates (s_k, t_k, u_k) for $k = 0, 1, 2$. Define C_k to be the de Casteljau operator using barycentric coordinates (s_k, t_k, u_k). Then the c'_{ij} coordinate of T' is $C_0^i C_1^j C_2^{d-i-j}$ applied to the control points of T. Further details about patch subdivision may be found in Böhm et al. [18]. More information on Bézier patches can be found in the literature [18, 36, 55].

3.3 Mesh refinement

In the 1D case, the domain of the function to be contoured was an interval. In the 2D case, the domain D consists of an arbitrary finite triangulation in $\mathbf{R}^2$. In fact, the techniques used in this chapter work equally well with any triangulation of a closed 2D manifold with boundary. Most commonly the domain will be a simple rectangle such as $D = [0, 1]^2$.

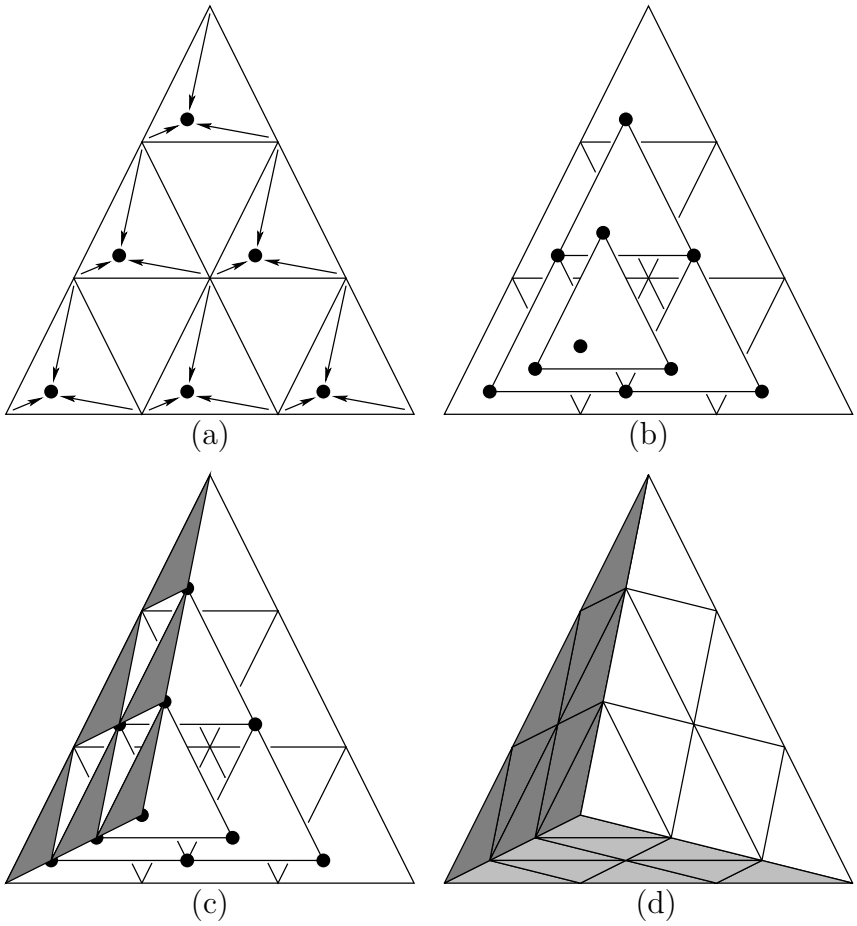


Figure 3.5: Evaluation and subdivision of a triangular Bézier patch by the de Casteljau operator: (a) one application of the de Casteljau operator with $d = 3$, (b) the d steps for evaluating the patch at a point. The points computed by the de Casteljau operator may be used to subdivide a patch into three sub-patches. (c) Points used by one of the sub-patches. (d) Three sub-patches cover the same domain as the original patch.

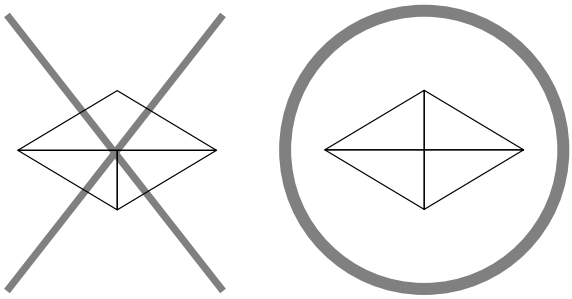


Figure 3.6: A refinement scheme should avoid creating hanging nodes/T-junctions.

We will sample the function f at the vertices of the initial triangulation, but this will generally not be sufficient to resolve f to the desired accuracy. We need a scheme for refining this mesh. In order to make a continuous interpolant, the mesh should be *conforming*: the intersection of any two triangles should be either a common edge, a vertex, or empty. Our refinement scheme should not introduce *hanging nodes* (also known as *T-vertices* or *T-junctions*, see Figure 3.6) into the mesh, since they cause discontinuities in the interpolant.

There are many 2D adaptive mesh refinement algorithms that create conforming meshes. Almost any of these could be used in our contouring framework. We use binary triangle trees, also known as newest-vertex-bisection meshes [66, 89], since they use a single triangle shape and a single type of split.

One mesh refinement algorithm, red-green triangulations [9], uses two types of splits. First any element that needs to be refined is divided into four congruent triangles using the *regular* or *red split*. Any triangle with two or three (red) split neighbors is also split using the red rule. Triangles with one red neighbor are split in half to remove the hanging node. This is called a *closure* or *green split*. Binary triangle trees are simpler since they have a single type of split. Binary triangle trees have the property that refined meshes are properly nested in their (coarser) ancestors. Also, binary triangle trees create meshes with fewer triangles.

Restricted quadtrees [106, 95, 84] produce almost the same meshes as binary triangle trees. The refinement rule for a quadtree divides a square into four equal squares. As long as adjacent squares differ by at most one level of refinement, the quadtree can be triangulated without hanging nodes. A binary triangle tree is somewhat simpler to program and can represent a larger class of adaptive meshes.

The refinement rule for longest-edge-bisection [80] always splits the longest edge of a triangle. The neighboring triangle is first recursively split until its longest edge is the same as the first triangle's. Then both triangles may be split without introducing a hanging node. The recursion will always terminate because every recursive call is splits a longer edge. Longest-edge-bisection of an isosceles right triangle gives the same mesh as newest-vertex bisection, but the decision procedure

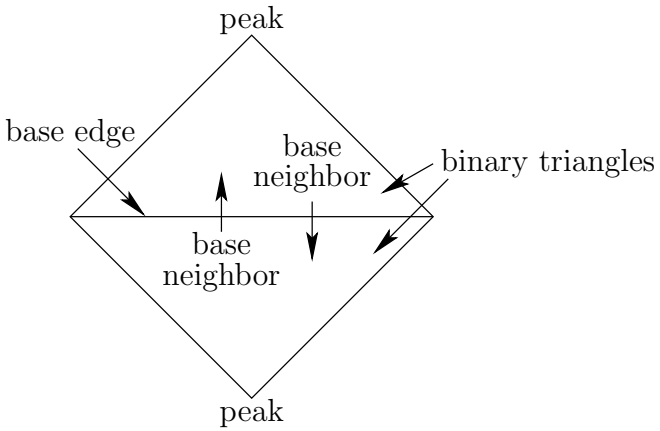


Figure 3.7: A diamond is two binary triangles oriented base-to-base.

for the latter is based solely on the combinatorial topology of the mesh, and does not require any edge lengths. Longest-edge-bisection can create many different conjugacy classes of triangles, but the smallest angle is bounded away from zero.

Binary triangle trees start with a *base* or *coarse mesh* of triangles, called *binary triangles*, covering the domain D. This triangulation is the coarsest that can be represented by the binary triangle tree. Every triangle in the base mesh has a distinguished edge, called the *base edge*, identified by the opposite vertex, called the *peak* of the triangle. Typically, the vertices are identified by the order they are listed.

The *base neighbor* of a binary triangle T is the neighboring triangle across the base edge. Ideally, a triangle and its base neighbor will both have the common edge as the base edge. This configuration is called a *diamond*, depicted in Figure 3.7. It is also desirable for the base edge to be the longest of a triangle. If the domain is a unit square, the base mesh will be chosen to consist of two isosceles right triangles forming a diamond. A technique for choosing base edges for arbitrary triangulations is given by Mitchell [66].

The refinement operation for binary triangle trees is called a *split*. The split operation takes a diamond and bisects each of the triangles, see Figure 3.8. This is accomplished by bisecting the base edge, adding one vertex to the mesh. This new vertex is connected to the peaks of the triangles in the diamond, adding two edges to the mesh. The net effect is that two triangles are replaced with four. The new vertex is the peak of the new, or *child*, triangles. This explains the newest-vertex-bisection terminology: the peak marks the most recent vertex in any given triangle and defines how it will next be split.

There is another split operation for binary triangles whose base edge is on the boundary of the triangulation. This type of split also adds a vertex at the midpoint of the base edge. One new edge is introduced, and the binary triangle is replaced with two smaller binary triangles. Both

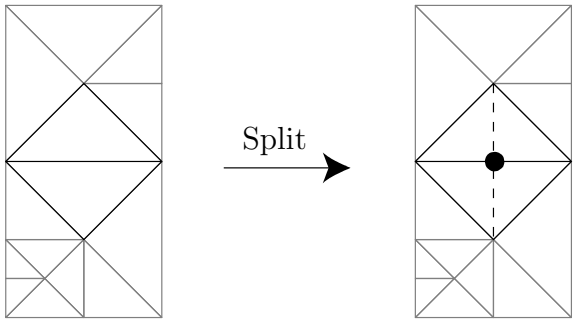


Figure 3.8: The split operation refines the mesh without introducing hanging nodes. The split introduces a new vertex and replaces two triangles with four.

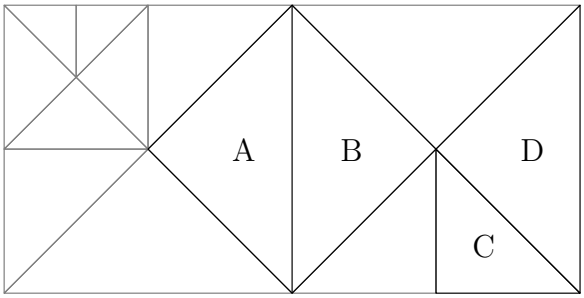


Figure 3.9: Adjacent triangles are always within one level of refinement.

versions of the split operation maintain the invariant that adjacent triangles are within one level of refinement (see Figure 3.9). More specifically, the base edge of a triangle is always either:

1. a boundary edge (triangle D),
2. the base edge of the neighboring triangle at the same level of refinement (triangles A and B, forming a diamond), or
3. the leg of the neighboring triangle which is one level coarser (triangle C).

The split operation has been given for the first two cases. In the third case splitting the triangle would introduce a hanging node (see Figure 3.6), causing the mesh to be non-conforming. This problem is resolved by splitting the neighboring triangle first, in what is called a *force split*. A force split may cause other force splits (as in Figure 3.10), but this process will always terminate — every step increases the size of the triangle and decreases the level of refinement.

Repeated application of the split operation to every triangle gives Figure 3.11. This is the $(4, 8^2)$ Laves lattice [51]. Since the split operation takes right isosceles triangles to right isosceles triangles, the mesh has triangles with a single similarity class.

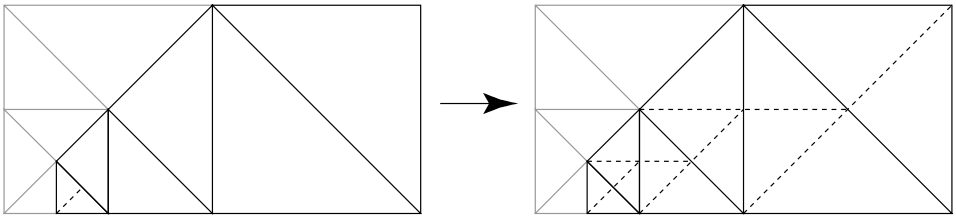


Figure 3.10: Splitting a triangle may force other triangles to split.

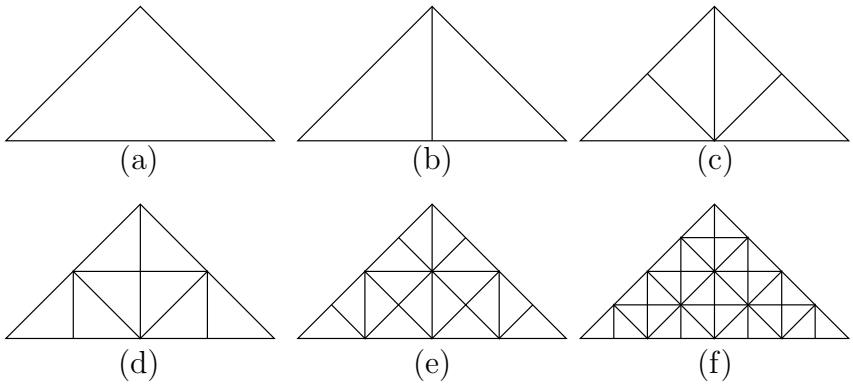


Figure 3.11: Uniform subdivision of a binary triangle results in 2^n triangles in level n.

To store a binary triangle tree, we use two (C++) vectors: one for the triangles of the mesh, one for the vertices. Each vertex stores a position, a function value, and any other data needed by the contouring algorithm (such as function gradients or a safe radius). Each triangle stores:

- the indices (in the triangle vector) of its three neighbors,
- the indices (in the vertex vector) of its three vertices and the center vertex (if any), and
- the next triangle in the result set (if this triangle is in the result set).

The center vertex is defined to be the bisector of the base edge. When a given triangle is split, it is replaced with one of its child triangles; the other child triangle is added to the end of the vector. Some care is needed to keep everything straight: splitting a triangle in the working set may cause triangles in the result set to split as well. This is why each element needs to know whether it is in the result set.

Newest-vertex-bisection meshes have been generalized to n dimensions (see Section 4.2).

3.4 Interpolation

In 1D, we constructed an approximation to f by cubic Hermite interpolation (Section 2.3). This exactly reproduces cubic polynomial functions, and gives C^1 continuity between elements. If the derivative of the function f is not known, there are a number of simple schemes for estimating a derivative based on neighboring values.

In 2D, given values and gradients at the vertices of a triangle, we would like to construct cubic spline patches that are both faithful (ideally reproducing cubic polynomials exactly) and match smoothly at patch boundaries. C^1 continuous interpolation is especially important for contouring — the continuity of the contours is at most the continuity of the function being contoured. Without a C^1 continuous interpolant, the computed contours can have corners or cusps.

Unfortunately, getting C^1 continuity and reproducing cubics is much more difficult in the 2D case. In fact, whether the vertices of a triangulation can be interpolated by a C^1 function with a single cubic polynomial per triangle is an open problem of approximation theory.

As a a result, there are several different interpolation techniques, each with different trade-offs. For a comprehensive survey see [55].

3.4.1 Nine Parameter Interpolant

For a triangular cubic spline patch, nine of the ten control ordinates are immediately determined from f and ∇f at the triangle vertices. The last control point is in the center and can be chosen to give *quadratic precision*: to reproduce quadratic polynomial surfaces exactly. However, no choice of the center control point based solely on the data on the corners (the *local data*) will give C^1 continuity between adjacent triangles.

3.4.2 C^1 Hermite Interpolant

To guarantee C^1 continuity with a single triangular patch covering each triangle, a quintic polynomial and C^2 data at each vertex is required. In general, to get C^r continuity, all ordinates within a distance $2r$ from the edge must be constrained. Thus C^{2r} data at each vertex is required. If the degree of the patch is less than $4r + 1$, some ordinates must be determined from the (conflicting) information at two different corners.

Finally, with a quintic polynomial interpolant, there are $\binom{5+2}{2} = 21$ ordinates to determine. Since only 9 degrees of freedom are specified (the values and gradients at the three corners of the triangle), additional information must be estimated or synthesized from neighboring elements.

3.4.3 Clough-Tocher Interpolant

The Clough-Tocher interpolant [24] divides each triangle into three subtriangles, called *micro-elements*. A triangular cubic Bézier patch is used on each micro-element. This scheme gives addi-

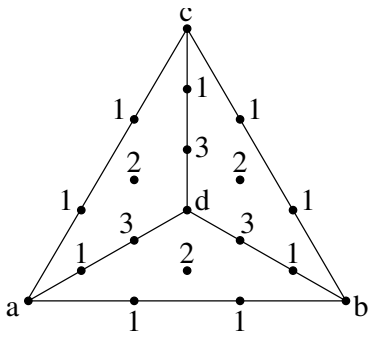


Figure 3.12: The Clough-Tocher interpolant divides triangular elements into three micro-elements.

tional control points without raising the degree of the element and decreases coupling between the constraints on each edge of the original triangle.

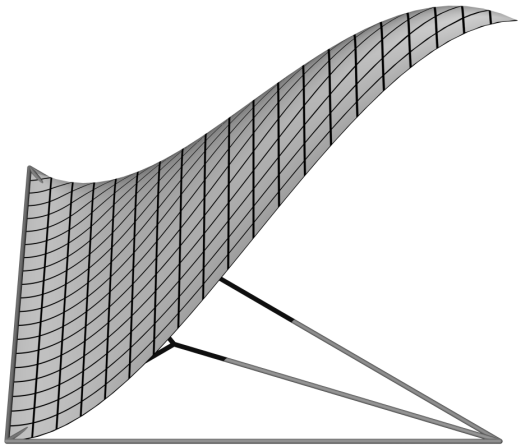
There are 19 ordinates to determine (see Figure 3.12). Each of the three cubic patches has 10 ordinates, but some are shared between the micro-elements to enforce C^0 continuity between the micro-elements. The values of the function at a, b, c determine the ordinates at the vertices. The gradients at the three corners determine the ordinates labeled 1 in the figure.

The ordinates labeled 2 in the figure are determined by the *cross-boundary derivatives* at the midpoint of each edge. C^1 continuity is achieved by having adjacent triangles agree on the cross-boundary derivative on their shared side, in addition to the values and gradients at the endpoints of that side. One choice that gives C^1 continuity without data from the neighboring triangle is to use *condensation of parameters*. The cross-boundary derivative of a cubic is generically a quadratic, but we can use our degree of freedom to make it linear by picking the cross-boundary derivative that is the average of the values at the endpoints. However, this choice prevents us from reproducing cubic polynomials exactly.

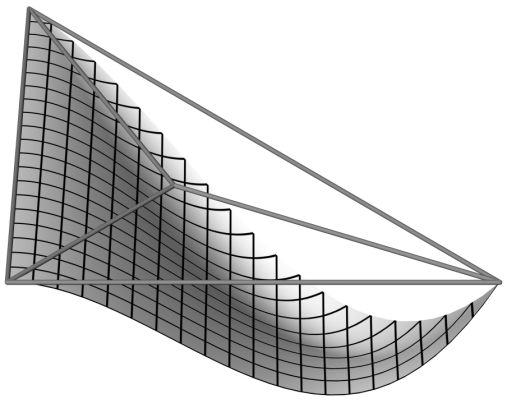
The remaining control points are chosen so that the micro-elements meet each other with C^1 continuity. Ordinates of type 3 are chosen to lie in the plane containing the nearest type 2 control points and the type 1 control point on the same (internal) edge. By affine invariance, this will always be achieved by taking the average of those three control points.

The last ordinate, d, is chosen to lie in the plane of the three type 3 ordinates. Hence d is the average of the type 3 ordinates. This enforces C^1 continuity at d, because the tangent plane at d is spanned by d and the type 3 ordinates. It turns that out that this also enforces C^2 continuity at d [1, 36]. Basis elements for Clough-Tocher interpolation are shown in Figure 3.13.

This interpolation scheme reproduces quadratic functions exactly. In order to reproduce cubic functions, a more sophisticated estimate of the cross-boundary derivative is needed. For example, the C^2 discontinuity between adjacent macro-elements can be minimized [60]. If the function is cubic, the C^2 discontinuity can be eliminated. For contouring, reducing the C^2 discontinuity between elements produces contours with smoother curvature.



(a)



(b)

Figure 3.13: Basis functions for Clough-Tocher interpoloation.

3.4.4 Powell-Sabin Interpolants

There are two Powell-Sabin interpolants [75]. One uses 6 micro-elements per macro-element, the other uses 12. Both use a quadratic spline patch on each micro-element. Typically, the first version is used when the largest angle of the macro-element is smaller than 75 degrees. By subdividing at the center of the inscribed circle as the dividing point (instead of the center of the circumscribed circle), the six-element version can be used exclusively. Powell-Sabin interpolation is popular for contouring because it is easier to contour a quadratic patch (yielding rational quadratic spline contours) than a cubic. A robust algorithm for contouring quadratics is given in [111].

3.4.5 Triangle-Square interpolant

Binary triangle trees have two types of elements: right isosceles triangles and squares (referred to as diamonds in Section 3.3). We combine the cubic split-square interpolant of Sibson [94] with Clough-Tocher interpolation on the triangles. The split-square interpolant matches with Clough-Tocher interpolation on boundaries with C^1 continuity. The procedure for constructing Sibson's split-square interpolant is:

1. Divide the square into four cubic triangular micro-elements (see Figure 3.14).
2. Set vertices a, b, c, and d to the values from the function we are interpolating.
3. Set ordinates of type 1 from the value and gradient of the nearest corner.
4. Set ordinates of type 2 so that the cross-boundary derivative at the midpoint of the edge is halfway between the value at the corners.
5. Ordinates of type 3 must be set to the average of the two adjacent type 2 ordinates for continuity. This also ensures that the type 3 ordinates lie in a plane: Indeed let the type 2 ordinates be $y_1, \ldots, y_4$ and the type 3 ordinates be $z_1 = (y_1 + y_2)/2, \ldots, z_4 = (y_4 + y_1)/2$. The z_i are planar if $z_1 + z_3 - z_2 = z_4$, or
$$\frac{y_1 + y_2}{2} + \frac{y_3 + y_4}{2} - \frac{y_2 + y_3}{2} = \frac{y_4 + y_1}{2}$$
which is always true.
6. The central vertex e must lie in the plane spanned by the type 3 ordinates for continuity. This can be accomplished by setting e to the average of all four type 3 ordinates, or the average of two opposite type 3 ordinates. In terms of the y_i above, the value at e is $\frac{1}{4}(y_1 + y_2 + y_3 + y_4)$.

This scheme reproduces quadratics exactly and the micro-elements join with C^1 continuity. Basis elements for Sibson split-square interpolation are shown in Figure 3.15.

Sibson also gives a piecewise-quadratic interpolant [94] that matches the Powell-Sabin interpolant on triangles (Section 3.4.4).

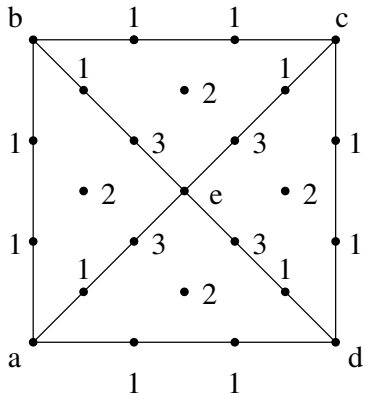


Figure 3.14: The Triangle-Square interpolant divides square elements into four triangular micro-elements.

3.5 Error model

As in the 1D case, we want to compute the interpolation error divided by the slope of the function (or our approximation) at the contour. The interpolation error can be estimated either by using a bound on the third derivative of f or by sampling f at the midpoint of the element. The slope of the function is the magnitude of the gradient.

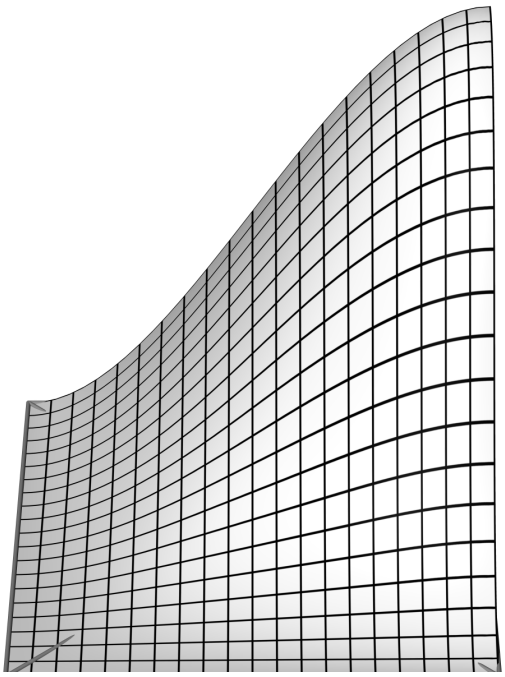
Since the triangle-square interpolant of Section 3.4.5 reproduces quadratics exactly, the error terms are approximately piecewise-polynomials of degree at least 3. As in the 1D case, we will first construct a basis for these polynomials.

Consider the function $f = x^3$ on the right isosceles triangle with vertices $(0, 0)$, $(0, h)$, $(h, 0)$. Figure 3.16(a) exhibits f as a cubic Bézier patch. To compute the error for Clough-Tocher interpolation on this triangle, sample the value and gradient of f at the three corners of the triangle. The Clough-Tocher construction builds the three cubic Bézier triangles shown in Figure 3.16(b). To compare the two, we can use the de Casteljau algorithm (Section 3.2) to subdivide f into three cubics in the same configuration as Clough-Tocher (Figure 3.16(c)). The difference is shown in Figure 3.16(d). Note that the interpolant agrees with f along the boundary, where it reproduces cubics. Also observe that the second column and row of the difference are zero — since the cross-boundary derivative is constant along the left and bottom edge. The maximum of the difference is at one of the points where the gradient equals zero. Using the equation solver from Section 3.7 below yields:

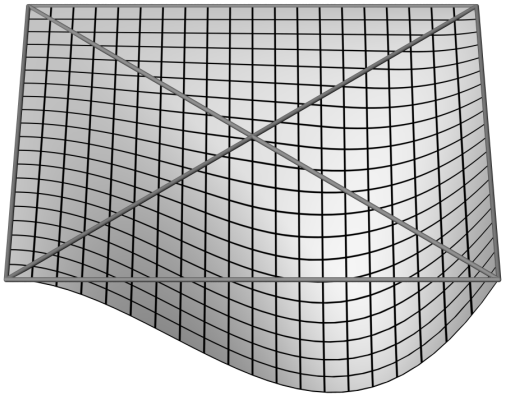
$$\text{Maximum of } 0.0283967h^3 \text{ achieved at } (.4127712h, .4127712h)\,.$$

This result has been verified using Maple.

Applying the same process to $3x^2y$, shown in Figure 3.17, gives a maximum error of $\frac{24}{361}h^3 \approx 0.0664820h^3$ at $(\frac{8h}{19}, \frac{4h}{19})$. By swapping x and y — equivalent to flipping these diagrams over the line



(a)



(b)

Figure 3.15: Basis functions for Sibson split-square interpoloation.

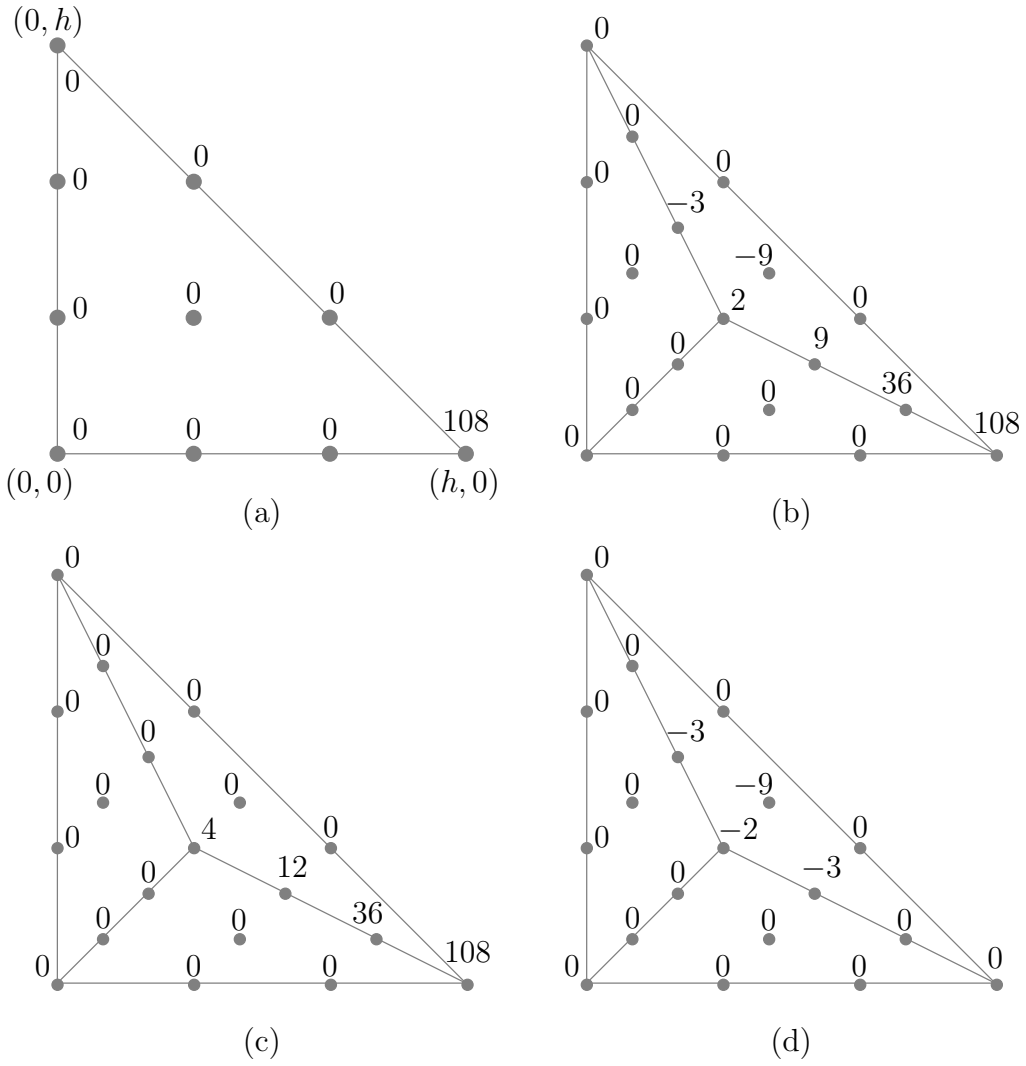


Figure 3.16: All of the ordinates in these diagrams should be multiplied by $\frac{h^3}{108}$. (a) The cubic x^3 restricted to an isosceles right triangle of size h, in the Bézier basis. (b) The Clough-Tocher approximation to the same function. Subdividing (a) using the de Casteljau operator results in (c). (d) The error: (b)-(c).

$x = y$ — we get the error bounds for $3xy^2$ and y^3. We can construct a bound on the interpolation error of a cubic polynomial using the triangle inequality:

$$\left(\frac{0.0283967}{6}\,|f_{xxx}| + \frac{0.0664820}{6}\,|f_{xxy}| + \frac{0.0664820}{6}\,|f_{xyy}| + \frac{0.0283967}{6}\,|f_{yyy}|\right) h^3$$

So if we have a bound on some measure of the third derivative, say

$$|f^{(3)}| = \sqrt{f_{xxx}^2 + 3f_{xxy}^2 + 3f_{xyy}^2 + f_{yyy}^2} \leq K\,,$$

we can maximize the expression above: The error is at most $0.0112538Kh^3$. Note that the given expression for $|f^{(3)}|$ is insensitive to rotation and so can be used without knowing the orientation of the interpolating elements.

We now apply this procedure to the Sibson split-square interpolant on the square $(0,0)$, $(h,0)$, (h,h), $(0,h)$. If $f = x^3$ or $f = y^3$, $f_{xy} = \frac{\partial^2 f}{\partial x \partial y} = 0$, and therefore the cross-boundary derivative is constant on all four sides. In this case, the interpolant exactly reproduces f. This leaves the function $f = x^2y$, shown as four cubic Bézier patches in Figure 3.18(a). The xy^2 case is symmetric. Note $f_{xy} = 2x$, which is constant for the vertical edges of the square. This implies the cross-boundary derivative, f_x, varies (at most) linearly along those edges. The ordinates of the second and second-to-last columns in the Sibson split-square interpolant, Figure 3.18(b), are therefore correct.

The maximum of the magnitude of the Sibson split-square error, Figure 3.18(c), $0.039446h^3$ occurs at $(0.5h, 0.774292h)$. The interpolation error on cubic polynomials combines this term with the symmetric term arising from $f = x^2y$:

$$\left(\frac{0.039446}{2}\,|f_{xxy}| + \frac{0.039446}{2}\,|f_{xyy}|\right) h^3\,.$$

If $|f^{(3)}| \leq K$ the corresponding error bound is $\frac{0.039446Kh^3}{\sqrt{6}} \approx 0.016104Kh^3$.

Without a bound on the third derivative of f, we can instead estimate the interpolation error. We begin by sampling the value of f at the center of the element and compare it to the interpolated values of $\tilde{f}$. In Figure 3.16(d), the difference in the cross-boundary derivative at the subdivision point $(.5h, .5h)$ is $\frac{3h^2}{4}\sqrt{2} \approx 1.0606602h^2$. The error of $0.0230377h^3$ is the difference in the cross-boundary derivative, $\Delta f_\perp$, multiplied by $\frac{2(0.0230377)h\sqrt{2}}{3} \approx 0.0217202h$. For the components in the direction of the boundary edge, Δf and $\Delta f_\parallel$, use the calculation is the same as the 1D case. This gives an error bound of

$$|\Delta f| + |\Delta f_\perp| 0.0217202h + |\Delta f_\parallel| \frac{2h\sqrt{2}}{27}\,.$$

This bound is useful only if it also applies to the error from Figure 3.17(d). In that case, the cross-boundary derivative is again $\frac{3h^2}{4}\sqrt{2} \approx 1.0606602h^2$ in magnitude. This follows since the two diagonal rows of ordinates closest to the diagonal edge are the same between Figure 3.16(d) and 3.17(d), up

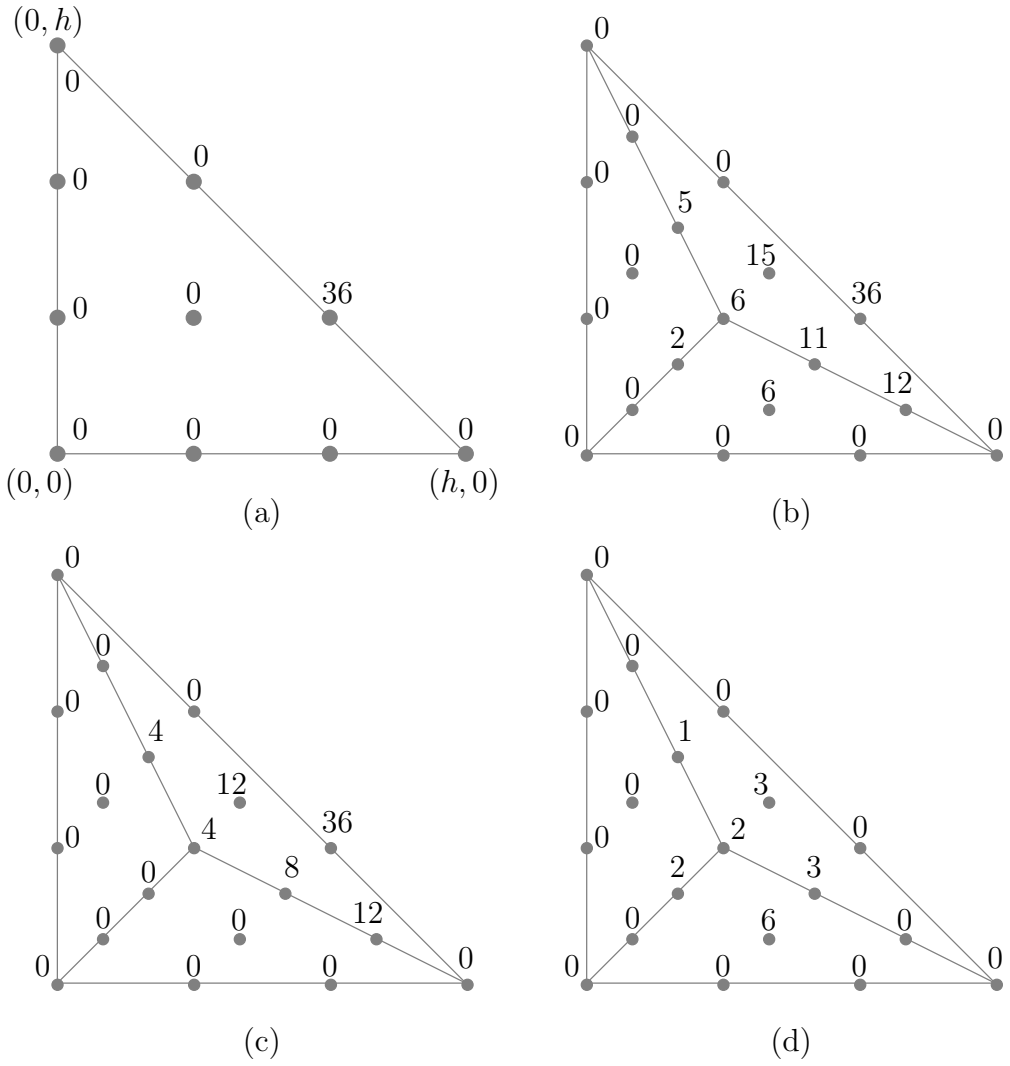


Figure 3.17: All of the ordinates in these diagrams should be multiplied by $\frac{h^3}{36}$. (a) The cubic $3x^2y$ restricted to an isosceles right triangle of size h, in the Bézier basis. (b) The Clough-Tocher approximation to the same function. Subdividing (a) using the de Casteljau operator results in (c). (d) The error: (b)-(c).

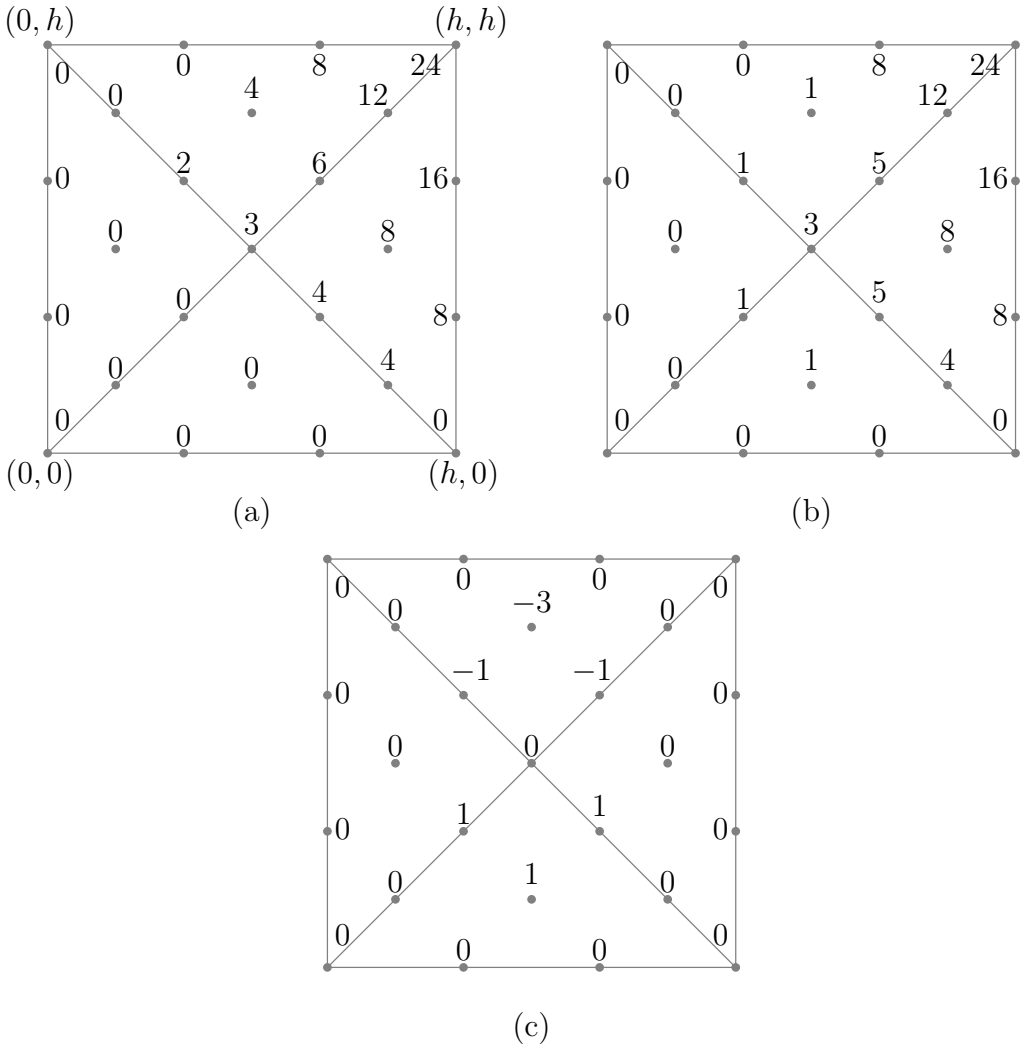


Figure 3.18: All of the ordinates in these diagrams should be multiplied by $\frac{h^3}{24}$. (a) The cubic x^2y restricted to a square of size h, in the Bézier basis. (b) The Sibson split-square approximation to the same function. (c) The error: (b)-(a).

to sign. To get the error of $\frac{24}{361}h^3 \approx 0.0664820h^3$, we multiply $|\Delta f_\perp|$ by $\frac{4(24)h}{3(361)\sqrt{2}} \approx 0.062680h$. The final error bound is

$$|\Delta f| + |\Delta f_\perp| 0.062680h + |\Delta f_\parallel| 0.104757h\,. \tag{3.1}$$

Of course, this should be multiplied by a user-specified safety factor.

For the Sibson split-square case (Figure 3.18), we compute the difference in the value Δf and gradient $(\Delta f_x, \Delta f_y)$ at the center of the element $(.5h, .5h)$. The interpolant reproduces x^3 and y^3 exactly, while $f = x^2y$ yields $|\Delta f_y| = \frac{h^2}{4}$ and $\Delta f = \Delta f_x = 0$. By symmetry, $f = xy^2$ yields $|\Delta f_x| = \frac{h^2}{4}$ and $\Delta f = \Delta f_y = 0$. In either case, the maximum error is $0.039446h^3$, which is $0.157784h$ times the difference in gradient. This gives the conservative error bound:

$$|\Delta f| + (|\Delta f_x| + |\Delta f_y|)\, 0.157784h\,. \tag{3.2}$$

Combining this with the formula for error on triangles (Equation 3.1) provides a reasonable error estimator for the triangle-square interpolant without an a priori bound on $|f^{(3)}|$.

In the 1D case we constructed $\min\left|\tilde{f}'\right|$ a lower bound on the magnitude of the derivative of $\tilde{f}$. In the 2D case we would like a lower bound $\min\left|\nabla\tilde{f}\right|$ on the magnitude of the gradient of $\tilde{f}$. With the triangle-square interpolant $\tilde{f}$ is locally a cubic polynomial represented as a Bézier patch. Differences of adjacent ordinates produce quadratic Bézier patches representing $\tilde{f}_x$ and $\tilde{f}_y$. The convex hull of these ordinates provides minimum and maximum values for $\tilde{f}_x$ and $\tilde{f}_y$. We can give a lower bound for $\left|\nabla\tilde{f}\right|$ using

$$\operatorname{minmag}(a,b) = \begin{cases} 0 & \text{if } 0 \in [a,b] \\ \min(|a|,|b|) & \text{otherwise}\,. \end{cases}$$

Then

$$\min\left|\nabla\tilde{f}\right| \le \sqrt{\operatorname{minmag}\left(\min(\tilde{f}_x), \max(\tilde{f}_x)\right)^2 + \operatorname{minmag}\left(\min(\tilde{f}_y), \max(\tilde{f}_y)\right)^2}\,. \tag{3.3}$$

Unfortunately, it is very common for these intervals to contain zero, even when the gradient is large on the contours. Further, this is a conservative lower bound for $|\nabla f|$ that will typically not be achieved. A more accurate estimate can be computed from the quartic Bézier patch corresponding to $f_x^2 + f_y^2$ by taking the square root of the minimum ordinate. This may still be quite pessimistic since it is unlikely that the minimum gradient magnitude occurs on the contours.

Optimistically, the values of $\left|\tilde{f}_x\right|$ and $\left|\tilde{f}_y\right|$ will be near their average values. We can compute the average values of $\tilde{f}_x$ and $\tilde{f}_y$ exactly: we simply take the average of the ordinates of their Bézier patch. This works since each of the quadratic basis elements has the same volume. Indeed up to symmetry, there are only two types of control points: edge and corner. A representative corner basis function gives volume:

$$\int_\triangle B_{200}^2\, dA \;=\; \int_{s+t+u=1,\ s,t,u\ge 0} s^2\, dA$$

$$\begin{aligned} &= \int_0^1 \int_0^{1-s} s^2 \, dt \, ds \\ &= \int_0^1 (1-s)s^2 \, ds \\ &= \int_0^1 s^2 \, ds - \int_0^1 s^3 \, ds \\ &= \frac{1}{3} - \frac{1}{4} = \frac{1}{12} \end{aligned}$$

A representative edge basis function gives the same volume:

$$\begin{aligned} \int_{\triangle} B_{110}^2 \, dA &= \int_{s+t+u=1,\ s,t,u\geq 0} st \, dA \\ &= \int_0^1 \int_0^{1-s} st \, dt \, ds \\ &= \int_0^1 \frac{1-s}{2} s \, ds \\ &= \frac{1}{2}\int_0^1 s \, ds - \frac{1}{2}\int_0^1 s^2 \, ds \\ &= \frac{1}{4} - \frac{1}{6} = \frac{1}{12} \end{aligned}$$

If $(f_x)_{\min} = \min(\tilde{f}_x)$ and $(f_x)_{\max} = \max(\tilde{f}_x)$ have the same sign, then the average of $\left|\tilde{f}_x\right|$ is the same as the average of $\tilde{f}_x$. Otherwise, we need to make some assumption about the distribution in order to estimate the average value of $\left|\tilde{f}_x\right|$.

The simplest assumption is that all values in $\left[(f_x)_{\min}, \overline{f_x}\right]$ and $\left[\overline{f_x}, (f_x)_{\max}\right]$ are equally likely. This is a piecewise-constant approximation to the distribution. Let $p(s)$ be the distribution defined on $[(f_x)_{\min}, (f_x)_{\max}]$. It must satisfy:

$$\begin{aligned} \int p(s) \, ds &= 1 \\ \int sp(s) \, ds &= \overline{f_x} \end{aligned}$$

assuming

$$p(s) = \begin{cases} L & \text{if } s \leq \overline{f_x} \\ R & \text{if } s > \overline{f_x}\,. \end{cases}$$

The linear system

$$\begin{aligned} L\left(\overline{f_x} - (f_x)_{\min}\right) + R\left((f_x)_{\max} - \overline{f_x}\right) &= 1 \\ \frac{L}{2}\left(\overline{f_x}^2 - (f_x)_{\min}^2\right) + \frac{R}{2}\left((f_x)_{\max}^2 - \overline{f_x}^2\right) &= \overline{f_x} \end{aligned}$$

determines L and R. Once L and R are determined, the absolute value of the distribution can be created, and the mean found, as in Figure 3.19. This gives a crude estimate for the average of $|f_x|$, but it will always be greater than zero.

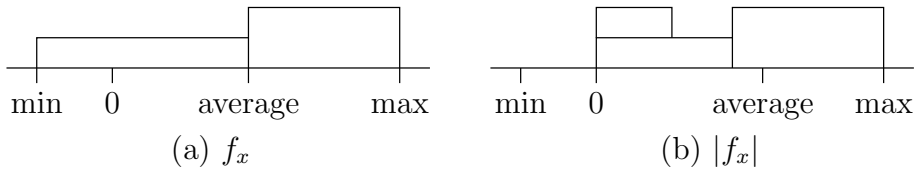


Figure 3.19: (a) Assuming the distribution of values of f_x is piecewise-constant. (b) The corresponding distribution for $|f_x|$.

Finally, we set the *optimism* level, a number between 0 and 1, to linearly interpolate between the lower bound given in equation 3.3 and the estimate of the average value of the gradient magnitude. Even a small value for optimism will ensure this number is greater than zero. This is important to avoid division-by-zero errors in the computation of the deflection of contours due to a given interpolation error.

It is possible to check that the a given optimism level is not too optimistic at the end of the contouring process. As each cubic element is contoured, the gradient magnitude is computed at a large number of points along the contour, especially at the curve fitting stage of Figure 3.1. If the minimum magnitude is returned, it can be checked against the minimum magnitude that would be safe with the given triangulation. If it is too small, the optimism level can be reduced and the contouring restarted.

Much simpler and less reliable error estimates are often used in the literature. For example, Petersen et al. [74] takes the minimum magnitude of the gradients at the vertices.

3.6 Finding the zero set of cubic functions of two real variables

Traditionally, either linear or quadratic interpolants (such as Powell-Sabin, see Section 3.4.4) have been used to interpolate 2D data. Quadratic triangular Bézier patches are relatively easy to contour, and linear patches are trivial to contour. The zero set of a bivariate quadratic polynomial consists of conic sections. These may be represented exactly using a rational quadratic spline. There are several schemes for finding these contours given a quadratic in the Bernstein-Bézier basis, for example Marlow and Powell's Fortran implementation [62]. A stable, robust, fast algorithm is in [111].

Contouring bicubic Bézier patches [43, 49, 77] involves a much larger space of functions, with 16 degrees of freedom compared to 6 for quadratic elements. The contours of a bicubic polynomial can have as many as 8 components (Figure 3.20). We have modified Grandine-Kleine's technique [49] to contour cubic polynomials.

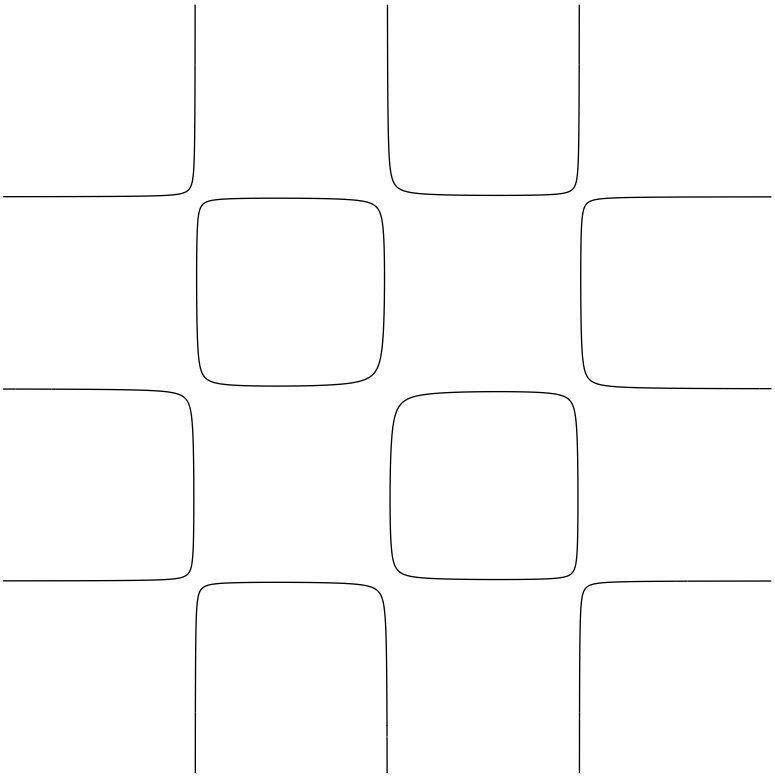


Figure 3.20: The zero set of the bicubic function $c(x)c(y)+0.0125$ with $c(t) = 3(1-2t)(1-4t)(3-4t)$ consists of 8 components in the square $0 \leq x, y \leq 1$.

3.6.1 Modified Grandine-Klein contouring

We interpolate f by triangular cubic Bézier patches (Section 3.4.5). To find the contours of these patches, we extend Grandine and Klein's algorithm for finding the contours of rectangular tensor-product spline patches [49] to handle triangular cubic spline patches. This section describes the parts of Figure 3.1 from *Spline Contouring* to *Curve Fitting*. These techniques may be extended to higher degree polynomials with a few straightforward modifications.

Figure 3.21 depicts the normal operation of the algorithm. The first step (Figure 3.21(a)) is to find *boundary* and *critical points*.

The boundary points are zeros of the function on the boundary of the triangle, or equivalently the points where the contours intersect the boundary. They are found by applying the 1D root finder from Section 2.6.1 to the control ordinates along each edge of the triangle. Boundary points are classified as either *entering* or *leaving* depending on the gradient of f. Note that ∇f is perpendicular to the contours of f, and therefore defines the tangent and normal to the contour. If the contour inside the triangle is below the intersection, we say the contour is leaving and the intersection point is a leaving point. If the contour extends above the intersection point, we classify the point as entering. Special cases such as horizontal tangents or zero gradients are discussed below.

Critical points are points where the function and the horizontal component of the gradient are both zero. Section 3.7 below details how to find these points. By examining the second partial derivatives of the function, we can determine the curvature at critical points and classify the points as either minima or maxima (min and max respectively in Figure 3.21(a)) of a contour. If the curvature is zero, the point is classified as horizontal. The zero-gradient special case is discussed below.

Next, a horizontal line is drawn through each of the boundary and critical points, dividing the triangle into trapezoidal *panels*; see Figure 3.21(b). The 1D cubic root finder from Section 2.6.1 is applied to each of these horizontal lines to find other intersections along the panel boundaries. New roots found this way are labeled *no transition*. Next, each of the roots on the panel boundaries are analyzed to determine their *below* and *above* numbers. The below number for a point is the number of contour segments ending at this point from the panel below. Similarly, the above number gives the number of contour segments starting at that point and extending into the panel above. The sum of the below and above numbers is the *valence* of that point. The below and above numbers usually determined by the classification of a point: a maximum has $(\text{below}, \text{above})$ of $(2, 0)$, an entering point has $(0, 1)$, and a no-transition point has $(1, 1)$. In Figure 3.21(b), the below and above numbers are adjacent to the corresponding contour point.

The sum of the above numbers along a panel boundary equals the sum of the below numbers one line higher, and these should agree with the number of intersections of any horizontal line within the panel with the contours. Our code checks this by counting the roots of the cubic restricted to the horizontal line midway within each panel.

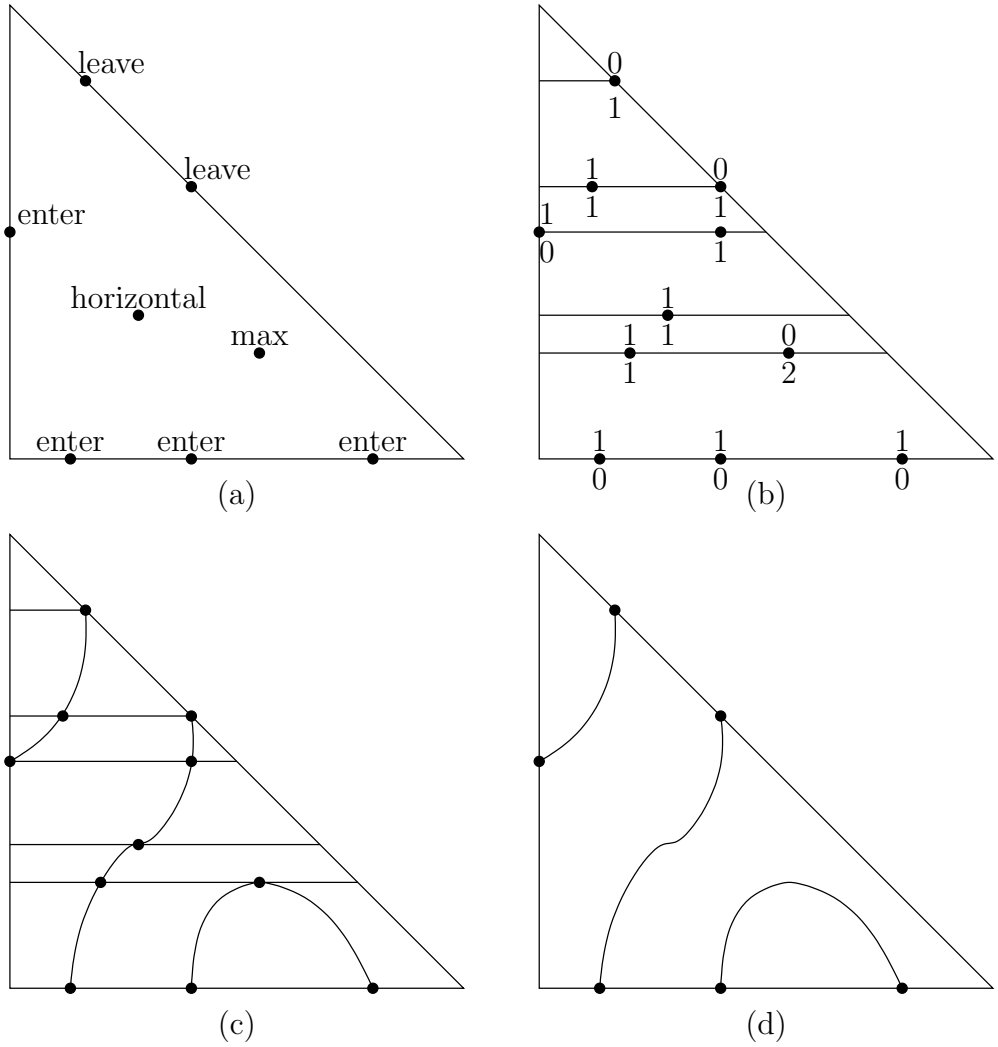


Figure 3.21: Our modified Grandine-Klein algorithm for contouring a spline patch has four main steps. (a) First the boundary points and interior critical points are determined. (b) The triangle is divided into panels. The intersections with the panel boundaries are labeled with their *below* and *above* numbers. (c) Contours within the panel are determined using the technique of de Boor et al. [28]. (d) The spline contours are connected according to the combinatorial topology at the panel boundaries.

The third step is to find the contour segments within each panel. The main tool used in this step is the technique of de Boor et al. [28]: a cubic spline with specified position, tangent, and curvature at the two endpoints is used to construct a single spline segment representing the contour. Next, the closest point on the contour to the midpoint of the spline is computed. If the distance between these two points is above the error threshold, the tangent and curvature are measured at the midpoint on the contour. The process of constructing a spline approximating the contour is recursively repeated on each half, producing cubic splines fitted to the contour and joining with G^2 continuity.

Finally, the contour segments are glued together at the panel boundaries. This process is divided into two stages. First, vertically-stacked contour segments are concatenated. Then, each sequence of the form (segment, maximum, segment, minimum, ...) is identified. These sequences can either start and end at edges or form a loop. In either case, every other segment is reversed and the result concatenated.

3.6.2 The complete algorithm

The above summary does not explain how many of the special cases are handled. Grandine and Klein's original algorithm does not address many of the following situations, and can allow contours to cross if the function has a saddle point:

- Zero gradient
- Zero curvature at a critical point
- Horizontal tangent at an edge
- Zero boundary edge
- *Thin* panel (any panel that is $O(\epsilon)$ thick)
- Spurious roots

Our modifications to the algorithm provide for all of these situations. An outline of the full algorithm is given in Algorithm 3.1, using the following modules:

FactorOutZeroSides If any edge of the triangle patch is identically zero then a linear factor must be divided out of the cubic. Otherwise we will not be able to detect where contours enter or leave the triangle. Dividing the linear factor out of a 2D cubic polynomial is much like deflation applied to a 1D polynomial (Section 2.6.3): Assume that the cubic is y times a quadratic factor. The bottom row of ordinates are all zero, and the remaining basis elements have y as a factor. Factoring out y from a cubic basis element gives a constant multiple of

```
function ContourCubicTriangle(Cubic, ϵ, ϵ_R):
if all Cubic.Ordinates positive, negative or zero:
    return [ ]
State.Cubic=Cubic
State.ϵ=ϵ
State.ϵ_R=ϵ_R
State.FactorOutZeroSides()
State.FindRootsOnEachSide()
if State.DetermineGoodSide() returns error:
    return State.ThreeStraightLinesCase()
return State.AddZeroSides(TryOrientation(State))

function TryOrientation(State):
State.ComputeGradients()
State.InsertEdgePoints()
State.InsertInteriorPoints()
if State.TData is empty:
    return [ ]
State.FindRootsAtSameT()
State.RemoveRedundantInterior()
State.InitializePanels()
if State.Panels is empty:
    return State.NoPanelsCase()
State.ComputeGuideLines()
State.SetBeforeAfter()
State.BuildStrands()
State.SplineForStrands()
return State.StitchConnectedStrands()
```

Algorithm 3.1: Contouring a cubic Bézier triangle in 2D.

one of the quadratic basis elements. The degree raising procedure for Bézier patches is then applied to change the quadratic into a cubic:

$$\begin{array}{cccc} a_{03} & & & \\ a_{02} & a_{12} & & \\ a_{01} & a_{11} & a_{21} & \\ 0 & 0 & 0 & 0 \end{array} = y \begin{pmatrix} a_{03} & & \\ \frac{3}{2}a_{02} & \frac{3}{2}a_{12} & \\ 3a_{01} & 3a_{11} & 3a_{21} \end{pmatrix}$$

$$= y \begin{pmatrix} a_{03} & & & \\ \frac{1}{3}a_{03} + a_{02} & \frac{1}{3}a_{03} + a_{12} & & \\ a_{02} + a_{01} & \frac{1}{2}(a_{02} + a_{12} + 2a_{11}) & a_{12} + a_{21} & \\ 3a_{01} & a_{01} + 2a_{11} & 2a_{11} + a_{02} & 3a_{21} \end{pmatrix}.$$

FindRootsOnEachSide passes a subset of the 10 ordinates for the triangular cubic spline to the 1D spline root-finder (Section 2.6.1), which determines all intersections of the contours with the edges of the triangle.

DetermineGoodSide DetermineGoodSide considers the three orientations of the cubic triangle. An orientation is considered *good* if there is no horizontal line (other than possibly the top vertex) where the cubic is identically zero. If there is a root from FindRootsOnEachSide on two sides within ϵ of the same height, this function checks two points equally spaced between the two edges at that height. If there are zeros of the function within ϵ of both points, then this orientation is *bad*.

Once a good side is known, the *cubic finder* may be initialized. Given a height, the cubic finder returns a 1D cubic spline that is the restriction of the triangular cubic to the horizontal line at that height. It requires the two 1D cubic polynomials defining the function restricted to the non-horizontal edges, and the quadratics defining the partial derivative in the horizontal direction along those two edges.

NearAZero This function is used by DetermineGoodSide (above), InsertEdgePoints, and InsertInteriorPoints (below) to see if a Bézier patch has a zero within ϵ of some point. This is determined by subdividing to a 2ϵ square neighborhood of the point and then determining if two ordinates have different signs.

ThreeStraightLinesCase A cubic that has no good side can only be the product of three linear terms, where each term's contour is a line parallel to a distinct side of the cubic patch. This case is contoured by dividing the straight lines at any intersection points that fall within the triangle. At those intersection points, the contours are glued together using the contour merging routine from Section 3.8 below.

Grandine and Klein deal with this case by allowing arbitrary orientations of the spline patch to be considered. This is needed when the degree of the patch is greater than 3.

ComputeGradients This procedure computes the partial derivatives of the cubic in the s and t directions. Each is a triangular quadratic patch. The ordinates of each quadratic are the differences between adjacent ordinates (in either the s or t direction) in the cubic, times 3 (the degree of a cubic).

InsertEdgePoints For each root found in FindRootsOnEachSide, this computes a gradient, transition type, and curvature. This corresponds to the first half of Figure 3.21(a). Edge transition types can be:

Entering if the gradient, and therefore normal, indicates the contour is entering the triangle. These also occur when the gradient is vertical and the curvature indicates this point is a minimum.

Leaving is the opposite of entering.

No Transition if the contour is tangent to this edge.

Horizontal at Edge if the gradient is vertical and the curvature is zero.

Unknown at Edge if the gradient is zero.

Isolated can happen at the corners or when the contour's only point in triangle is a tangent point.

The tests to see if a value are zero actually use NearAZero to be sure that the zero is not missed due to inaccuracies. We detect roots within ϵ of the corners and use additional tests to make sure they are classified correctly and do not appear on two separate sides. Every edge point is labeled with a side from 0 to 2, except corner points which are numbered from 3 to 5.

InsertInteriorPoints This function simultaneously solves $\tilde{f} = 0$ and $\tilde{f}_s = \frac{\partial \tilde{f}}{\partial s} = 0$ to find interior critical points. See Section 3.7 for one algorithm that finds all solutions to these two equations. It then computes a transition type and curvature for these points. Transition types may be:

Minimum if the sign of $\tilde{f}_{ss}/\tilde{f}_t$ is negative.

Maximum if the sign of $\tilde{f}_{ss}/\tilde{f}_t$ is positive.

Horizontal if the curvature of the contour is zero.

Unknown if the gradient is zero.

If one of the roots is within ϵ of an edge, it checks that the point was not already added in InsertEdgePoints. This completes Figure 3.21(a).

If no points are found in the above two procedures, this cubic has no contours in the domain.

FindRootsAtSameT For every point x found in InsertEdgePoints and InsertInteriorPoints, the 1D root finder (Section 2.6.1) is applied to the restriction of the cubic to the horizontal

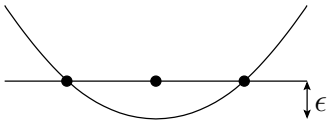


Figure 3.22: Errors in finding a minimum or maximum can lead to additional roots.

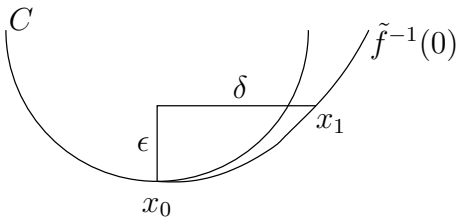


Figure 3.23: To determine if two points x_0 and x_1 are redundant as in Figure 3.22, we compute the osculating circle C at the critical point x_0. Here δ is the difference in s between the two points and $\tilde{f}^{-1}(0)$ is the contour we would like to find.

line containing x. Any new roots found are added with the transition type of No Transition. This begins step 2 above (Figure 3.21(b)).

RemoveRedundantInterior Consider the situation where a minimum x_0 is found in InsertInteriorPoints is off by ϵ. Depending on the direction of the error, FindRootsAtSameT will find either zero or two neighboring roots. If it finds two (Figure 3.22) we would like to eliminate them as redundant.

Let x_1 be a point neighboring x_0, and let their difference in s, $|s_1 - s_0|$, be denoted δ. Let K_0 and K_1 be the computed curvature of the contour at x_0 and x_1 respectively. A simple quadratic approximation to the contour is given by $t - t_0 = \frac{K_0}{2}(s - s_0)^2$. Plugging in $t - t_0 = \epsilon$ shows that δ be as large as $\sqrt{\frac{2\epsilon}{K_0}}$, much larger than ϵ. We would like to avoid accidentally eliminating a root that comes from a nearby contour. We do this by comparing the osculating circles (determined by the curvature and tangent) of adjacent roots at the same t. Close contours have very different osculating circles, but small changes along a single contour leave the osculating circle nearly fixed.

The difference between the centers of the osculating circles at x_0 and x_1 comes from two sources: displacement due to x_1 not lying on the osculating circle from x_0, and deflection due to the change in slope between the osculating circle and x_1.

A reasonable bound on the deflection term is

$$2\epsilon + \left|\frac{dK}{d\delta}\right| \frac{\delta^3}{3!}.$$

The 2ϵ is from the ϵ error determining the position of the two points. The rest of the error is

from the inaccuracy of the quadratic approximation, and is therefore the cubic term from the Taylor series. To be completely conservative, $\left|\frac{dK}{d\delta}\right|$ should be the maximum on $[0, \delta]$. From the data we have, however, we can use the estimate $\left|\frac{K_1 - K_0}{\delta}\right|$.

The deflection term is approximately the change in angle times the radius. The radius is the inverse of curvature, so to be conservative we use $r_{\max} = \frac{1}{\min(|K_0|, |K_1|)}$. The change in angle is approximately the change in slope $\left|\frac{dK}{d\delta}\right| \frac{\delta^2}{2}$ since $\tan\theta \approx \theta$ for small θ. Putting all these terms together gives

$$2\epsilon + \left|\frac{dK}{d\delta}\right| \frac{\delta^3}{6} + \left|\frac{dK}{d\delta}\right| \frac{\delta^2 r_{\max}}{2} .$$

Plugging in $\delta \approx \sqrt{2\epsilon r_{\max}}$, to keep the expression small for two distinct contours, gives

$$2\epsilon + \left|\frac{dK}{d\delta}\right| \frac{(2\epsilon r_{\max})^{3/2}}{6} + \left|\frac{dK}{d\delta}\right| \frac{2\epsilon r_{\max}^2}{2} .$$

This can safely be multiplied by a constant such as 10 since the difference for adjacent contours will be much larger.

InitializePanels We have now classified the contours at various t values. InitializePanels divides the remainder of the domain into trapezoids between two such t values. It also evaluates the cubic at the t value halfway between the top and bottom of the panel. This determines how many strands of the contour connect the bottom of the panel with the top. For very thin panels, less than 6ϵ high, the count of the number of roots at the midpoint of the panel is unreliable. Thus instead of generating a thin panel, the roots are merged into a thick panel boundary. Instead of a single t value for the boundary, a range of t values is stored. Further, different points within the boundary may have different t values. Every thick boundary line uses guide lines (see below) to resolve contour segments and the points within the boundary line.

NoPanelsCase The above procedure finds no panels only when all roots of the cubic are isolated. Each of these is output as a degenerate spline with all four control points at the same location.

ComputeGuideLines Points with a transition type of Unknown or Horizontal can occur at the top or bottom of a panel. ComputeGuideLines evaluates a *guide line* 2ϵ away from the edge of the panel. The tolerances for computing the roots along the guide line are shrunk until it has the same number of roots as the mid-line of the panel. The goal of computing guide lines is to resolve the ambiguity at the unknown points by extrapolating the contour from nearby points.

SetBeforeAfter For each contour point in panel boundaries, we need to say how many strands of the contour merge at there and how many spring from that point. For a maximum or a minimum, this is easily established. For unknown points we need to extrapolate from the guide line. We could use either linear extrapolation or the osculating circle. Since the distances are

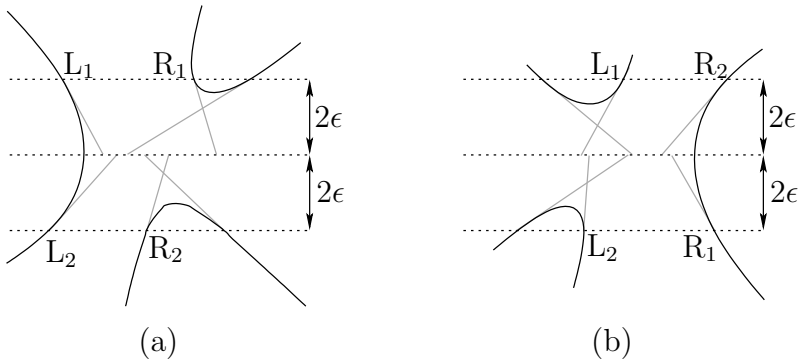


Figure 3.24: Resolution of an Unknown transition point with valence 6. Contours are extrapolated from 2ϵ above and below the transition point.

so small, linear extrapolation is sufficiently accurate. Each strand is extrapolated to the panel boundary, and associated with the closest root on the panel boundary. This completes step 2 (Figure 3.21(b)).

At this point, the panel is checked for consistency. The sum of the before or after numbers is compared to the number of roots along the mid-line of that panel. If there is a disagreement, the cubic patch is rotated, and the contouring starts over, as in DetermineGoodSide. This prevents a nearly horizontal contour contained entirely within a thin panel from creating havoc later.

BuildStrands This procedure determines the topology of of the contour segments. It iterates through the the points along the panel boundaries resolving the topology there. For most transition types, this is straightforward. The tricky cases are Unknown and Unknown at Edge. The resolution of the Unknown at Edge cases is deferred until these contours are merged with the adjacent triangle's contours (see Section 3.8 below). Unknown points with a valence of 2 are straightforward, since they are equivalent to a Maximum, a Minimum, or a No Transition.

For Unknown points with a valence of 4 or 6, we need to extrapolate from where the contours intersect the guidelines 2ϵ above and below the Unknown point (Figure 3.24). Each guideline intersection is extrapolated to the horizontal line with t equal to the Unknown point. The left-most (L_1 in the figure) and right-most (R_1) intersections with the mid-line are then used to determine which pairs of contours are to be connected according to these rules:

1. Along the top and bottom, contours should alternate L and R.
2. The left-most (and right-most) contours on the top and bottom should have the same letter.
3. If the left-most (right-most) contours are labeled L (R) then they should be connected.

4. Along the top and bottom, contours should be connected to adjacent contours so that the left contour is labeled R and the right contour is labeled L.

Note that R_1 may disagree with the labeling given by L_1. In this case the second-left-most L_2 and second-right-most R_2 are used to resolve the tie. The figure shows why these rules are sensible; they derive from the fact that the tangent lines will intersect closer to the contour than the asymptotes of the approximating hyperbola. Furthermore, since a cubic polynomial is being contoured, no single line can intersect contours more than three times. This greatly constrains the possible configurations.

SplineForStrands To interpolate between the sequence of points located on the contour, we need a method of performing at least G^1 interpolation with cubic splines. A cubic spline in the plane has 8 degrees of freedom, 2 for each of the 4 control points. We begin with 4 constraints: the position of both end points, p_0 and p_1. We can compute the unit tangent, t_0, t_1, and curvature, k_0, k_1, information at the end points to add 4 more constraints. As long as the signs of the curvature are consistent with the positions and tangents, this determines a cubic curve [28]. Since adjacent curves will match in position, tangent, and curvature, we will end up with G^2 interpolation.

Let $c_0, \ldots, c_3$ be the control points of the cubic Bézier spline we are trying to find. The position constraints determine $c_0 = p_0$ and $c_3 = p_1$ directly. The tangent constraints restrict c_1 and c_2 to lie on lines tangent to the curve: thus $c_1 = c_0 + \alpha t_0$, $c_2 = c_3 - \beta t_1$ with $\alpha, \beta \geq 0$. Finally, we impose the curvature constraints. Let $c^{**} = \frac{c' \times c''}{|c'|^3}$, so $c^{**}(0) = K_0$ and $c^{**}(1) = K_1$, yielding a quadratic 2×2 system of equations for α and β:

$$a_0\alpha = a_1 - a_2\beta^2$$
$$b_0\beta = b_1 - b_2\alpha^2 .$$

It has been shown [28] that this system will have solutions if the curve is subdivided sufficiently. Furthermore, the curve will preserve convexity and will be accurate to 6th order: The error will be $O(h^7)$ if the end points are a distance h apart.

The system may be solved by the 2D simultaneous spline solver from Section 3.7. Alternatively, α or β can be eliminated by substituting one equation into the other. The result is a quartic polynomial and we may find its roots using the 1D spline root finder from Section 2.6.1. The latter approach proved faster and more reliable. In either case we must derive bounds on the maximum possible values that α and β can take.

If any of the coefficients are zero, we can eliminate one variable and solve the system using the quadratic formula. If a_0 and a_2 have the same sign, then we may immediately conclude that $\alpha \leq \frac{a_1}{a_0}$ and $\beta \leq \sqrt{\frac{a_1}{a_2}}$. If b_0 and b_2 have the same sign, this also leads to bounds on α and β. That leaves the case where a_0a_2 and b_0b_2 are both negative, as depicted in Figure 3.25. In

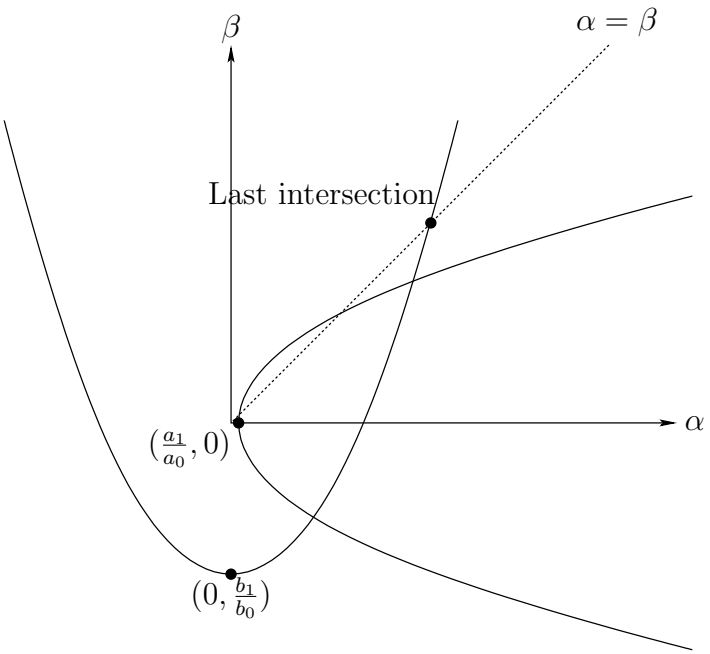


Figure 3.25: Plots of the solutions to the two quadratic equations in terms of α and β.

this case, we can find the largest intersection of the two equations with the line $\alpha = \beta$, and that will be a bound on α and β. After that point, the solutions to the first equation are to the right of the line, and the solutions to the second equation are above the line.

If no solution can be found, we set α and β to one-third of the distance between p_0 and p_1.

In either case, we determine the halfway point, $\tilde{h}$, on the computed spline curve. Using the cubic finder and the 1D root finder, we find the point on the contour h_0 with the same t as $\tilde{h}$. If this distance between $\tilde{h}$ and h_0 is less than the error threshold, the spline approximation is good enough. Otherwise, we consider the line tangent to the contour at h_0. The t value of the point on that line closest to $\tilde{h}$ is used to compute h_1. If h_1 and $\tilde{h}$ are still too far apart, the problem is divided at h_1 and the above procedure is applied to each half.

To connect a point to an Unknown at Edge point, we use a quadratic spline that matches the tangent and curvature at one of the end points. This, in effect, extrapolates the tangent and curvature from the point where it is known to the point where the gradient is zero. In the construction above, the only points connected to an Unknown at Edge point are on a guideline 2ϵ either above or below. The quadratic approximation is quite accurate on such short intervals.

StitchConnectedStrands We now have a collection of strands increasing monotonically in t. This function stitches together those connected at a common maximum or minimum. It reverses

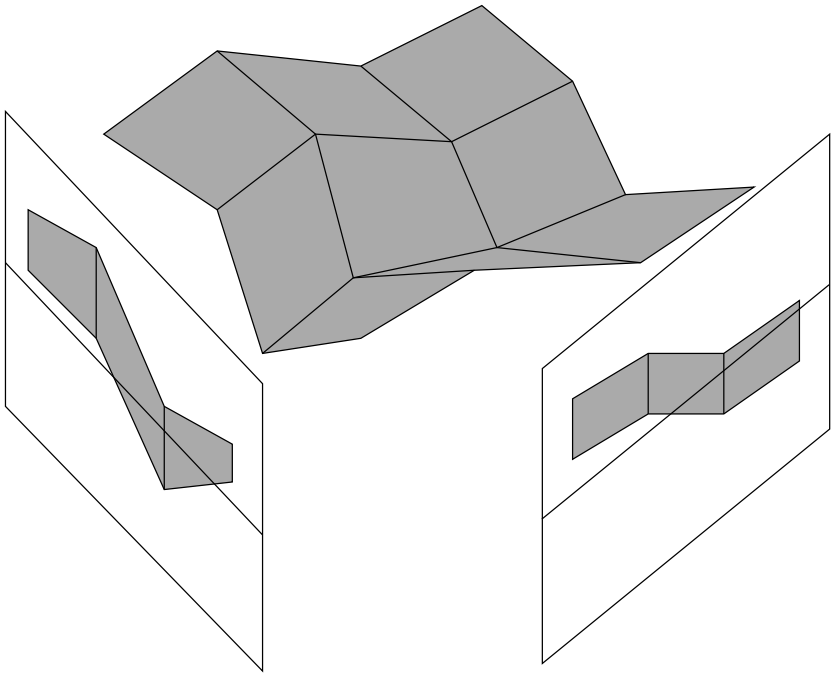


Figure 3.26: The PPI algorithm projects the control polygon onto two perpendicular planes.

every other segment and then returns the final collection of splines representing the contours.

AddZeroSides AddZeroSides adds the straight line contours noted in FactorOutZeroSides back into the solution set. It divides each straight line segment at the contour intersection points found by FindRootsOnEachSide. These will be connected appropriately when this triangle is glued to its neighbor in the jigsaw puzzle stage (Section 3.8).

Situations arise where the transition types at the critical points do not agree with any consistent contour topology. These cases usually only occur when ϵ was large relative to the size of the cubic patch. Specific checks for topological inconsistencies are included in NoPanelsCase, SetBeforeAfter, and BuildStrands. When any problem is detected, the algorithm restarts after rotating the triangular patch. If all of the good orientations have been tried, ϵ is reduced.

3.7 A modified Sherbrooke-Patrikalakis equation solver

The contouring algorithm requires a method for finding all of the roots of a system of polynomial equations within a particular region. In particular, InsertInteriorPoints requires the points (s,t) where $\tilde{f}$ and $\tilde{f}_s$ both vanish. Both $\tilde{f}$ and $\tilde{f}_s$ are represented by Bézier patches, cubic and quadratic repsectively.

There are several possible approaches [49] to solving this problem — we follow Grandine and

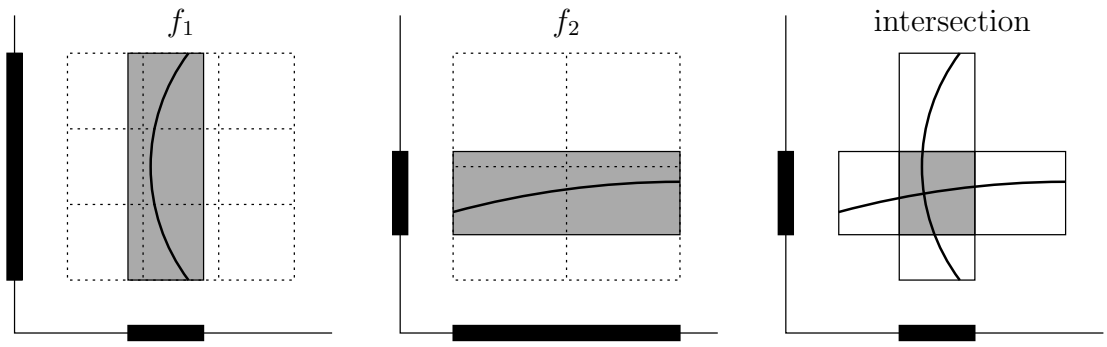


Figure 3.27: Given the convex-hull intersections of the two functions, restrict to where they are both potentially zero.

Klein's use of an algorithm by Sherbrooke and Patrikalakis [90]. The 1D version of this algorithm was described in Section 2.6.1. The generalization to higher dimensions is called the *projected-polyhedron intersection*, or *PPI*, algorithm. Let $f_1, f_2, \ldots, f_n$ be Bézier patches in $\mathbf{R}^n$. We will consider the case $n = 2$, but the approach works for any dimension n.

1. Let the control polyhedron for f_i be defined by the control points $\{(x_{j1}, \ldots, x_{jn}, y_j)\}$. For each $k \in \{1, \ldots, n\}$, we project these control points to $\mathbf{R}^2$ by mapping $(x_{j1}, \ldots, x_{jn}, y_j)$ to (x_{jk}, y_j). The case $n = 2$ is depicted in Figure 3.26.

2. Compute the ConvexIntersection (Section 2.6.4) for each i and k.

3. For each k, intersect all the intervals. We are left with a n-box which contains all of the common zeros of the f_i. The case $n = 2$ is depicted in Figure 3.27.

4. If the box is not much smaller in some dimension than the original domain, split the box in half along that axis.

5. Repeat this procedure for the set $\{f_i\}$ restricted to each box.

The original algorithm uses rectangular regions and rectangular (tensor product) splines. Some changes are needed to apply this approach to triangular splines:

1. The last step takes the restriction of f_i to a box. This can be accomplished with subdivision for rectangular splines. For triangular splines, two (or more if $n > 2$) patches are required to cover the rectangle. See Section 3.7.3 below for an efficient method of creating the second patch.

2. The rectangular region for roots determined by the algorithm can lie outside of the original triangle (see Figure 3.28). Thus, the results of the algorithm need to be filtered.

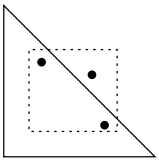


Figure 3.28: The modified Sherbrooke-Patrikalakis solver may find solutions outside of the original triangular domain if they are contained in the bounding box of the solutions inside the triangle.

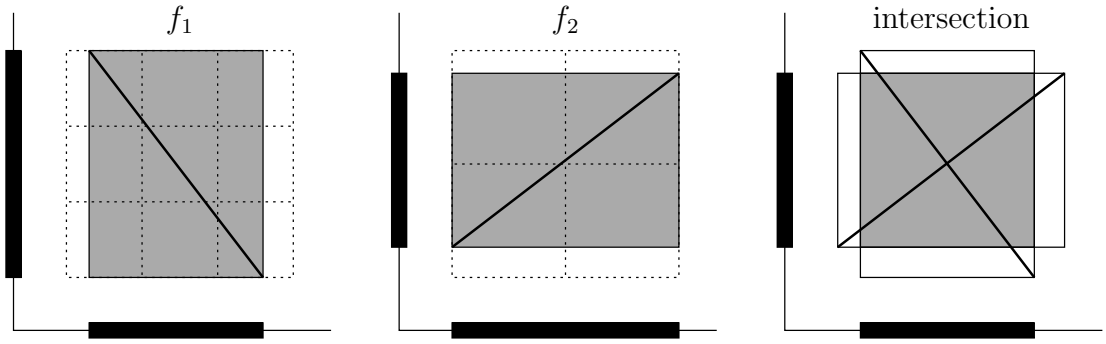


Figure 3.29: The PPI algorithm can converge slowly even for linear functions f_1, f_2.

3. A given root can be returned twice: The problem arises when the boxes for the two triangular patches making up a rectangle overlap. Our approach merges the boxes into a single box containing both patches if they overlap. An alternative solution considers the two patches together: take the union of the control points from both patches to create a single box for both.

3.7.1 Convex hull

Observe that we never need to generate the actual convex hull of the control points, only the intersection of the convex hull with the x-axis. Further note that the input is three or four vertical line segments, equally spaced along the x-axis. Thus the convex hull intersection algorithm of Section 2.6.4 works without modification.

3.7.2 Quadratic convergence

While the PPI algorithm has quadratic convergence in 1D, it converges only linearly in higher dimensions. In 1D, the algorithm gives the exact answer for any linear function, and, by the variation diminishing property of Bézier splines, the deviation from linear shrinks as the interval is subdivided. In the 2D case, it can fail to converge quickly even for linear functions (Figure 3.29).

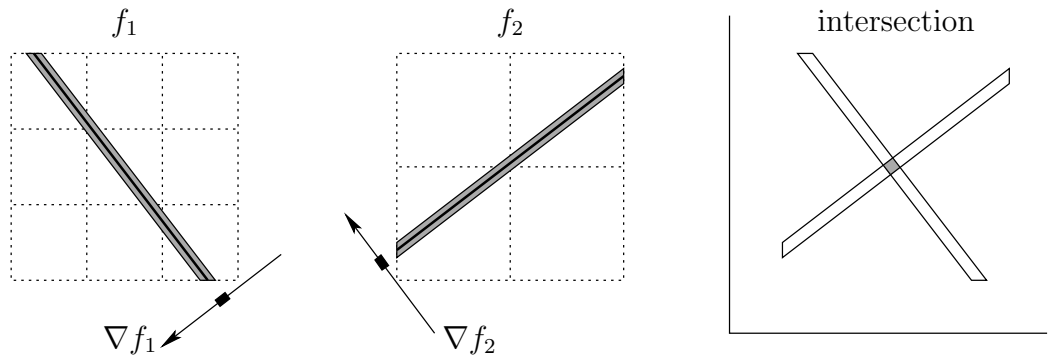


Figure 3.30: Projecting onto the gradient direction can provide much quicker convergence.

Sherbrooke and Patrikalakis [90] have constructed an algorithm with quadratic convergence, the *linear-programming intersection* algorithm. Due to the additional expense of solving a linear programming problem at each step, it is not always faster than the original slowly-converging formulation.

Elber and Kim [34] have a different modification to the PPI algorithm that uses normal cones to detect when a region contains a single root. When this happens, they switch to using Newton's method, which converges quadratically to the roots.

There is a simple modification that gives the exact answer for linear functions. The idea is to project onto a direction that is parallel to the gradient of the function instead of each of the coordinate axes (Figure 3.30). We call this the *gradient-projection intersection*, or *GPI*, algorithm. The ConvexIntersection function takes longer to run since there will be more input points, but it will be run only once per function instead of n times.

We can estimate the gradient of a Bézier patch from the control points at the four corners of the patch. There are a few cases to consider:

- If the estimated gradient for one function is near zero, we substitute the direction perpendicular to the other function's gradient.
- If both functions have estimated gradients near zero, we fall back on the coordinate axes.
- If the estimated gradients for both functions are nearly parallel, the problem is poorly conditioned. Small changes to the control points will cause the solutions to move a great distance. This shows up in the PPI algorithm as spurious solutions near the actual solution.
- In the worst case, f_1 and f_2 share a contour. The PPI algorithm outputs a large number of points spaced ϵ apart along that contour. We would rather detect this case and return an error. Using GPI, when the gradients are nearly parallel, we can analyze the patch ordinates to see if both functions are nearly zero throughout a rectangle perpendicular to the gradient.

3.7.3 Subdivision

Subdivision computes the Bézier ordinates for the two triangular patches corresponding to a rectangular sub-domain of a triangular Bézier patch. The ordinates from the first patch are computed using the standard technique [18].

The ordinates for the second patch may then be computed by a simple linear function of the first patch. In the cubic case, the function is defined by

$$\begin{array}{cccc} a_{03} & & & \\ a_{02} & a_{12} & & \\ a_{01} & a_{11} & a_{21} & \\ a_{00} & a_{10} & a_{20} & a_{30} \end{array} \quad \mapsto \quad \begin{array}{cccc} a_{03} & b_{20} & b_{10} & b_{00} \\ & a_{12} & b_{11} & b_{01} \\ & & a_{21} & b_{20} \\ & & & a_{30} \end{array}$$

where the b_{ij} are:

$$\begin{aligned} b_{20} &= a_{03} + a_{12} - a_{02} \\ b_{10} &= a_{01} - 2a_{11} + a_{21} - 2a_{02} + 2a_{12} + a_{03} \\ b_{00} &= a_{30} + a_{03} - a_{00} - 6a_{11} + 3a_{10} - 3a_{20} + 3a_{01} + 3a_{21} - 3a_{02} + 3a_{12} \\ b_{11} &= a_{12} + a_{21} - a_{11} \\ b_{01} &= a_{10} - 2a_{20} + a_{30} - 2a_{11} + 2a_{21} + a_{12} \\ b_{02} &= a_{21} + a_{30} - a_{20}\,. \end{aligned}$$

For the quadratic case

$$\begin{array}{ccc} a_{02} & & \\ a_{01} & a_{11} & \\ a_{00} & a_{10} & a_{20} \end{array} \quad \mapsto \quad \begin{array}{ccc} a_{02} & b_{10} & b_{00} \\ & a_{11} & b_{01} \\ & & a_{20} \end{array}$$

where the b_{ij} are defined by:

$$\begin{aligned} b_{10} &= a_{02} + a_{11} - a_{01} \\ b_{00} &= a_{00} - 2a_{10} + a_{20} - 2a_{01} + 2a_{11} + a_{02} \\ b_{01} &= a_{11} + a_{20} - a_{10}\,. \end{aligned}$$

This map guarantees that the two patches match up with at least C^0 continuity, because the ordinates match along the common boundary, even in the presence of roundoff error in the computation.

These equations are derived by expressing the basis elements of one patch in the coordinate system of the other. If the lower-left triangle has coordinates (s,t) with $s,t \in [0,1]$ and $s+t \leq 1$ and the upper-right triangle has coordinates (s',t'), then $s' = 1-s$ and $t' = 1-t$. Given f written as $\sum_{i,j,k=d-i-j} a_{ij} B^d_{ijk}(s,t)$ coordinates b_{ij} such that $f = \sum_{i,j,k=d-i-j} b_{ij} B^d_{ijk}(s',t')$ are given in

the quadratic case by

$$\begin{aligned}
a_{00}B^2_{002}(s,t) &= a_{00}(1-s-t)^2 = a_{00}\left(1-(1-s')-(1-t')\right)^2 = a_{00}(-1+s'+t')^2 \\
&= a_{00}(-1)^2 B^2_{002}(s',t') = a_{00}B^2_{002}(s',t') \\
a_{10}B^2_{101}(s,t) &= a_{10}(2)s(1-s-t) = a_{10}(2)(1-s')(-1)(1-s'-t') \\
&= -2a_{10}\left((1-s'-t')+t'\right)(1-s'-t') \\
&= -2a_{10}\left((1-s'-t')^2 + t'(1-s'-t')\right) \\
&= a_{10}\left(-2B^2_{002}(s',t') - B^2_{011}(s',t')\right) \\
a_{20}B^2_{200}(s,t) &= a_{20}s^2 = a_{20}(1-s')^2 = a_{20}\left((1-s'-t')+t'\right)^2 \\
&= a_{20}\left((1-s'-t')^2 + 2t'(1-s'-t') + t'^2\right) \\
&= a_{20}\left(B^2_{002}(s',t') + B^2_{011}(s',t') + B^2_{020}(s',t')\right) \\
a_{01}B^2_{011}(s,t) &= a_{01}(2)t(1-s-t) = -2a_{01}(1-t')(1-s'-t') \\
&= -2a_{01}\left((1-s'-t')+s'\right)(1-s'-t') \\
&= -2a_{01}\left((1-s'-t')^2 + s'(1-s'-t')\right) \\
&= a_{01}\left(-2B^2_{002}(s',t') - B^2_{101}(s',t')\right) \\
a_{11}B^2_{110}(s,t) &= a_{11}(2st) = 2a_{11}(1-s')(1-t') \\
&= 2a_{11}\left((1-s'-t')+t'\right)\left((1-s'-t')+s'\right) \\
&= 2a_{11}\left((1-s'-t')^2 + s'(1-s'-t') + t'(1-s'-t') + s't'\right) \\
&= a_{11}\left(2B^2_{002}(s',t') + B^2_{101}(s',t') + B^2_{011}(s',t') + B^2_{110}(s',t')\right) \\
a_{02}B^2_{020}(s,t) &= a_{02}t^2 = a_{02}(1-t')^2 = a_{02}\left((1-s'-t')+s'\right)^2 \\
&= a_{02}\left((1-s'-t')^2 + 2s'(1-s'-t') + s'^2\right) \\
&= a_{02}\left(B^2_{002}(s',t') + B^2_{101}(s',t') + B^2_{200}(s',t')\right) .
\end{aligned}$$

Summing up the terms on the left-hand side gives $\sum_{i,j,k=d-i-j} b_{ij}B^d_{ijk}(s',t') = f$. Writing f in terms of the expressions on the right-hand side gives

$$\begin{aligned}
f &= a_{00}B^2_{002}(s',t') + a_{10}\left(-2B^2_{002}(s',t') - B^2_{011}(s',t')\right) + \\
&\quad \ldots + a_{02}\left(B^2_{002}(s',t') + B^2_{101}(s',t') + B^2_{200}(s',t')\right) .
\end{aligned}$$

Grouping the coefficients of $B^2_{ijk}(s',t')$ together gives

$$\begin{aligned}
f &= (a_{00} - 2a_{10} + a_{20} - 2a_{01} + 2a_{11} + a_{02})B^2_{002}(s',t') + \\
&\quad \ldots + a_{02}B^2_{200}(s',t') .
\end{aligned}$$

So $b_{00} = a_{00} - 2a_{10} + a_{20} - 2a_{01} + 2a_{11} + a_{02}$ and so on through $b_{20} = a_{02}$. A similar but longer computation gives the linear equations for the cubic case.

3.8 Jigsaw Puzzle

The jigsaw puzzle module joins the contours found on individual triangular Bézier patches. For each patch, we have a list of spline segments. A contour consists of a starting side of the triangle, an ending side, and a list of control points for the spline curve. Starting and ending sides are numbered 0 to 2, unless the contour goes through a corner of the triangle, in which case a value from 3 to 5 is stored. Contours forming a loop inside a patch have starting and ending sides set to -1. In addition, for each triangular patch, we have an integer identifier, and a neighbor on each side.

The jigsaw puzzle routines maintain a hash table mapping triangle identifiers to a list of puzzle sides that have not been matched up. Each puzzle side stores an edge identifier, the neighboring triangle's identifier, and pointers to the two puzzle corners. Puzzle corners are stored in a separate list with a single integer per corner. The last number used as either a corner or edge identifier is also stored.

As each triangle is contoured it is added to the puzzle using a routine called AddPiece. Once all of the triangles are added, the FinishPuzzle routine cleans up any unresolved puzzle sides. AddPiece is where most of the work is accomplished (Algorithm 3.2). Given a triangle T to add to the puzzle, its sides are renumbered to either match adjacent triangles that are already contoured, or a previously unused identifier. Corners get similar treatment.

Once the renumbering is complete, contours are merged. Edges that were already in the puzzle are added to the list of identifiers to be merged and then removed from the puzzle. When T has two neighbors already in the puzzle, there are two possibilities: either the neighbors agree or disagree about the corner they have in common. In the first case, this is the last triangle incident to that corner and the corner is added to the merge list. In the second case, this triangle is connecting two triangles that were not locally connected, and the corners are renumbered to agree.

The MergeContours routine is responsible for connecting contours together. It first scans through all contour ends looking for sides in the *ToMerge* list. It gathers together all ends with the same side number that are within ϵ of each other into a *star*. Then ends in each star are sorted according to their angle. The angle is determined from the vector connecting the end point to the adjacent point in the contour control point list. The two ends that have adjacent angles and an angle difference closest to π are connected. Combining ends with adjacent angles ensures that we do not introduce contour crossings. Isolated points are detected and eliminated when they are attached to other contours. Loops can form if the two ends of the contour are in the same star. Otherwise, the two contours are reversed if necessary and then concatenated.

At the end, FinishPuzzle is run. This scans through the puzzle data structure and collects a list of edge and corner identifiers that have yet to be merged. These are then processed using MergeContours.

```
procedure AddPiece(TriContours, TriNeighbors, TriId):
Renumber = [-1, -1, -1, -1, -1, -1]
ToMerge = []
for Side in 0..2:
    if TriNeighbors[TriId] in IdToPuzzleSide:
        # Neighbor has already been contoured
        PuzzleTri = IdToPuzzleSide[TriNeighbors[TriId]]
        for PuzzleSide in PuzzleTri:
            if PuzzleSide.Neighbor is TriId:
                break
        Renumber[Side] = PuzzleSide.EdgeId
        ToMerge.Append(PuzzleSide.EdgeId)
        for Corner in Side:
            if Renumber[Corner] is -1:
                Renumber[Corner] = PuzzleSide.CornerId[Corner]
            else if Renumber[Corner] is PuzzleSide.CornerId[Corner]:
                # This triangle completes this corner
                ToMerge.Append(Renumber[Corner])
            else:
                # This triangle connects two components
                PuzzleSide.CornerId[Corner] = Renumber[Corner]
        PuzzleTri.Remove(PuzzleSide)
    else:
        # Neighbor has not been contoured yet
        MaxId = MaxId+ 1
        Renumber[Side] = MaxId
        IdToPuzzleSide[TriId].PuzzleSide.EdgeId = MaxId
        IdToPuzzleSide[TriId].PuzzleSide.Neighbor = TriNeighbors[TriId]
for Corner in 3..5:
    if Renumber[Corner] is -1:
        # New corner
        MaxId = MaxId+ 1
        Renumber[Corner] = MaxId
Renumber(TriContours, Renumber)
MergeContours(PuzzleContours, TriContours, ToMerge)
```

Algorithm 3.2: Routine for adding a triangle to the jigsaw puzzle.

	Linear	Cubic
Execution time	$\frac{1.8 \times 10^{-3}}{\epsilon^{3/5}}$ seconds	$\frac{7.6 \times 10^{-3}}{\sqrt[6]{\epsilon}}$ seconds
Function evaluations	$\frac{27}{\sqrt{\epsilon}}\ f$	5 f and 5 ∇f
Output vertices	$\frac{10}{\sqrt{\epsilon}}$	$\frac{20}{\sqrt[7]{\epsilon}}$
Maximum Error	$\frac{\epsilon}{6.4}$	$\frac{\epsilon}{10}$
Average Error	$\frac{\epsilon}{11}$	$\frac{\epsilon}{100}$

Table 3.1: Approximate asymptotic performance of contouring $f_\circ$ with linear and cubic interpolation.

3.9 Results

For our first test case, we contoured the circle that is the zero set of

$$f_\circ(x, y) = \left(10x - \frac{5}{2}\right)^2 + \left(10y - \frac{5}{2}\right)^2 - 4$$

on the unit square $[0, 1]^2$. For comparison, we used both the triangle-square interpolant from Section 3.4.5 and a simple linear interpolant. Both cases used binary triangle trees to adaptively sample the function. These tests were run on a 450MHz Pentium II computer with 128 MB RAM running Microsoft Windows NT. The error tolerance ϵ was set to values between 0.125 and 6×10^{-8}.

The performance of contouring with linear and cubic interpolation is summarized in Table 3.1. For linear interpolation, most of the time was spent in AddPiece. As ϵ shrank, processing time shifted from 75% connecting line segments and 25% adaptive meshing and sampling to 82% and 18% respectively.

Since the triangle-square interpolant reproduces quadratic functions exactly, the adaptive sampling stops immediately. The asymptotic behavior of the cubic case was dominated by the curve-fitting procedure, which has excellent convergence. 99.3% of the processing time was spent contouring individual cubics. The algorithm allocates most of the allowable error to interpolation error since that results in the fewest function evaluations and triangles to contour. In this case there was no interpolation error, so the final error was quite small. Cubic contouring in this case was faster, more accurate, and output fewer vertices than linear contouring (Figure 3.32).

The remaining functions were tested with three variations of cubic interpolation:

Max Derivative A bound on the third derivative was used to control sampling.

Sampling The function was sampled at a point inside each element. If the value and gradient of the function was too far from the interpolant, the element was subdivided.

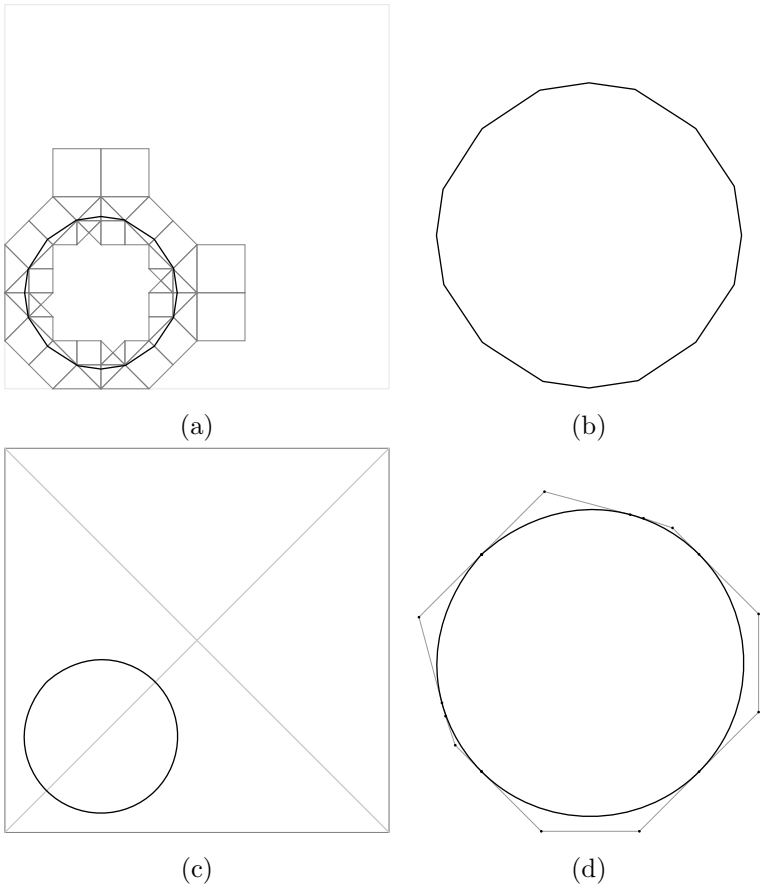


Figure 3.31: $f_\circ$ has a single circular contour. (a) and (b): the linear approximation to that contour found using a linear interpolant. It has 25 vertices, a maximum error of 0.00781, and required 168 function evaluations. The cubic approximation, in (c) and (d), has 19 control points, a maximum error of 0.00159, and required 5 function and gradient evaluations. In both cases, ϵ was set to 0.0625. The elements contoured from the adaptive mesh are shown in (a) and (c). No subdivisions were necessary for the cubic interpolant; in (c), the light gray lines show the boundaries of the micro-triangles from Sibson's split-square interpolant. (d) the control polygon used for the cubic contour approximation.

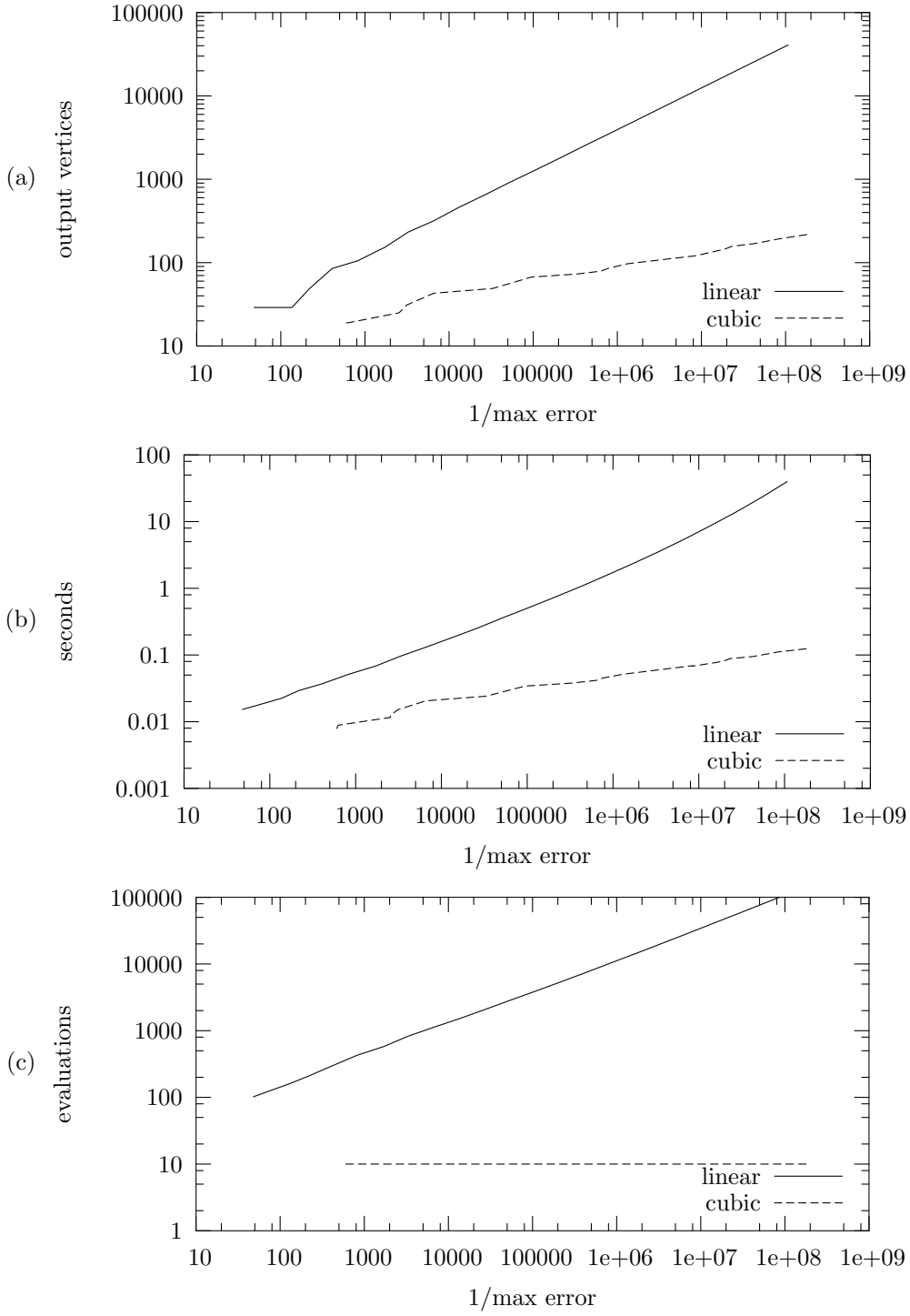


Figure 3.32: These plots graph (a) the size of the control polygon, (b) the run-time in seconds, and (c) the number of function and gradient evaluations, against one over the maximum error for linear and cubic contouring of $f_\circ$.

	Linear	Max Derivative	Sampling	Cubic Precision
ϵ	.05	.05	.05	.2
Max error	.023	.00051	.0035	.0059
Evaluations	617	694	530	13
Seconds	.13	.5	.3	.05
Output size	156	762	559	62

Table 3.2: Parameters for the contours of f_θ in Figure 3.35

Cubic Precision The interpolant uses the cross-boundary derivative from the actual value of the function's gradient. This allows the interpolant to reproduce cubic functions exactly at the expense of additional gradient evaluations. The error metric Sampling was used, without the gradient term (Equations 3.1 and 3.2).

For linear interpolation, we used a variation of the sampling method. The difference between the function and the interpolant at the midpoint correctly estimates the quadratic error term. This does not accurately predict the error because the cubic error term may be large even when the quadratic term vanishes, so the quadratic error term was added to the element's radius cubed times a constant. The constant was set for each function to ensure the maximum error was less than the specified ϵ.

Function $f_b(x, y)$ is bicubic, defined by $c(x)c(y)+0.0125$ with $c(t) = 3(1-2t)(1-4t)(3-4t)$. There are 8 contours of f_b (Figure 3.20). The gradient is extremely small where two contours approach each other. Examples of the resulting contours are shown in Figure 3.33. The performance of the various algorithms is graphed in Figure 3.34.

Function $f_\theta(x, y)$ is cubic, defined by $9(x-y)(25(x+y-1)^2+100(x-y)^2-8)+0.01$. The Cubic Precision technique did not subdivide the domain at all, since in that case the interpolant reproduced the function exactly. Figure 3.35 shows the contours with the parameters in Table 3.2. As expected, the Cubic Precision method also scales the best, as can be seen in Figure 3.36.

We have found the error estimate is more easily determined using cubic contouring. For linear contouring, the value of the ad hoc coefficient of the cubic term was crucial to obtaining the specified accuracy. This coefficient had to be set for each function, often by trial and error. The cubic contouring method reproduces quadratic functions exactly and correctly estimates the cubic and quartic terms of the error. The accuracy of the algorithm was largely insensitive to the magnitude of the coefficient of the fifth order term — it served primarily to force the first subdivision.

These performance numbers could be improved by additional optimization. In particular, the above times are based on the PPI algorithm with linear convergence. Also, the jigsaw puzzle routine could cache the contours associated with each side. This would improve speed (at the expense of memory usage) unless main memory is exhausted.

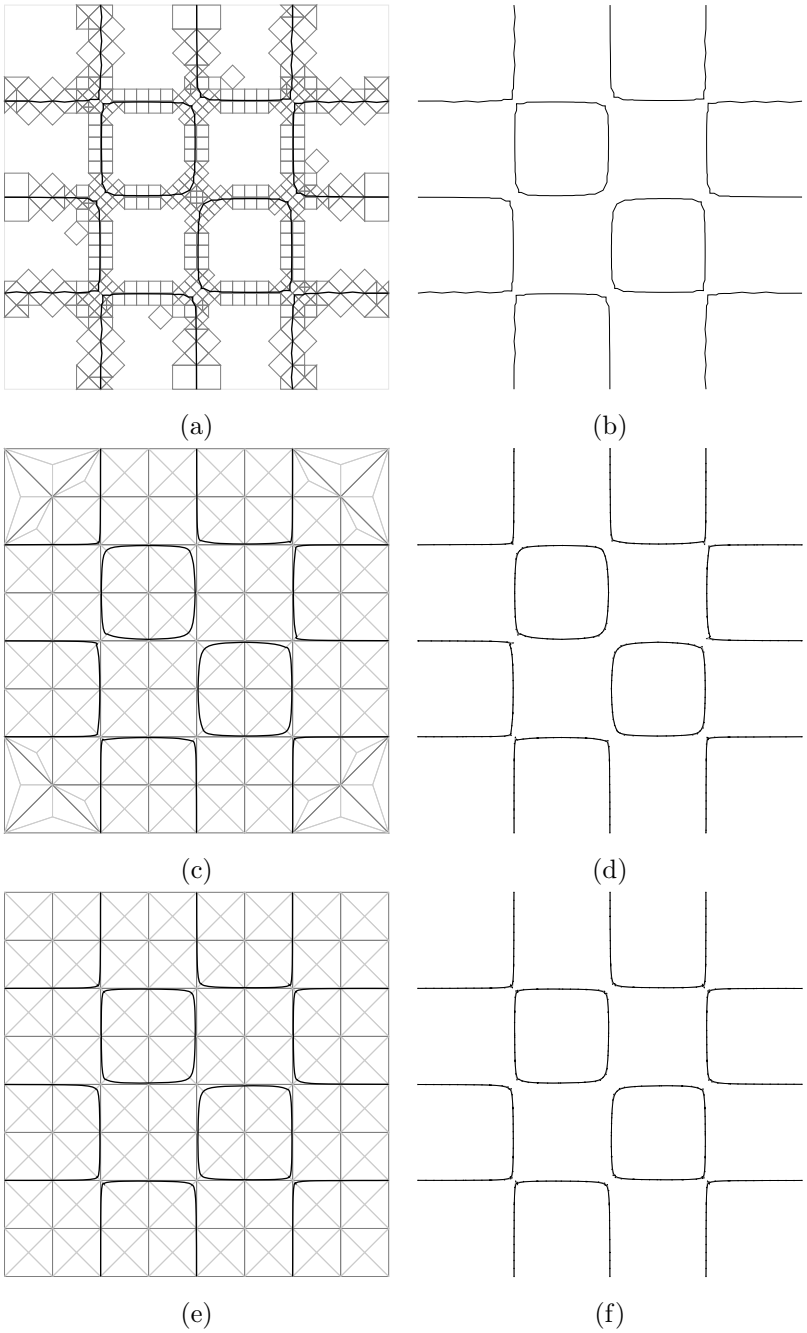


Figure 3.33: Here f_b was contoured using the Linear method (a)–(b), the Sampling method (c)–(d), and the Cubic Precision method (e)–(f). The adaptive mesh is shown on the left; the control polygon on the right.

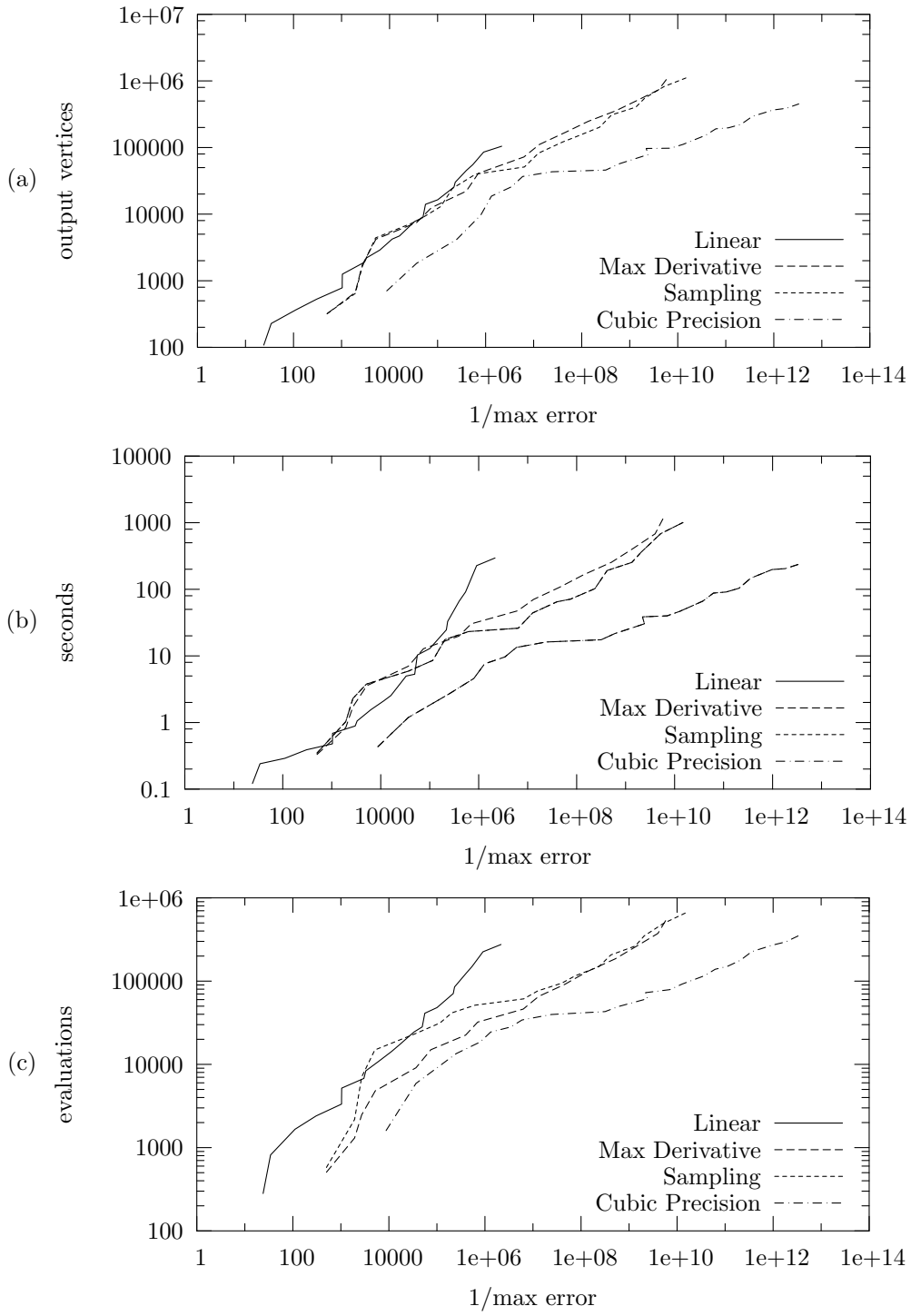


Figure 3.34: These plots graph (a) the size of the control polygon, (b) the run-time in seconds, and (c) the number of function and gradient evaluations, against one over the maximum error for linear and cubic contouring of f_b.

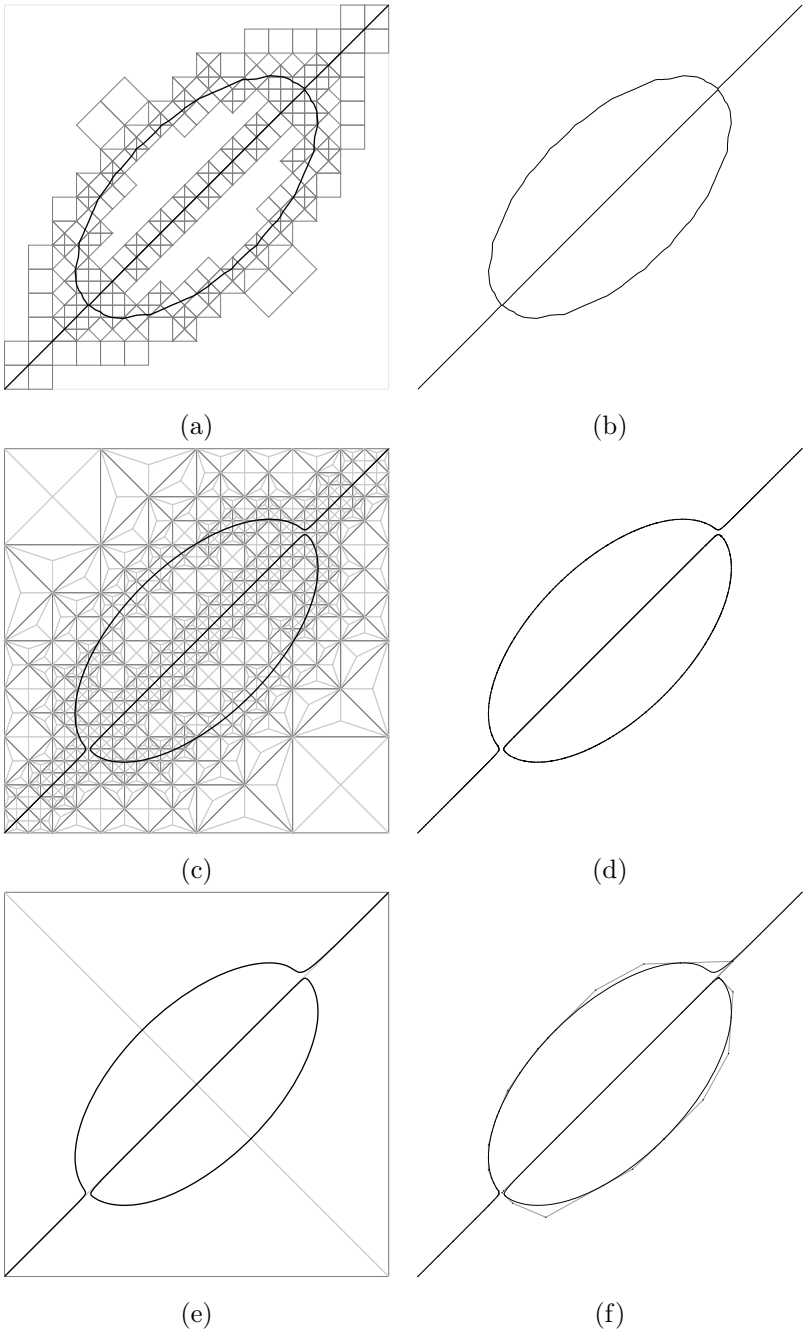


Figure 3.35: Here f_θ was contoured using the Linear method (a)–(b), the Max Derivative method (c)–(d), and the Cubic Precision method (e)–(f). The adaptive mesh is shown on the left; the control polygon on the right.

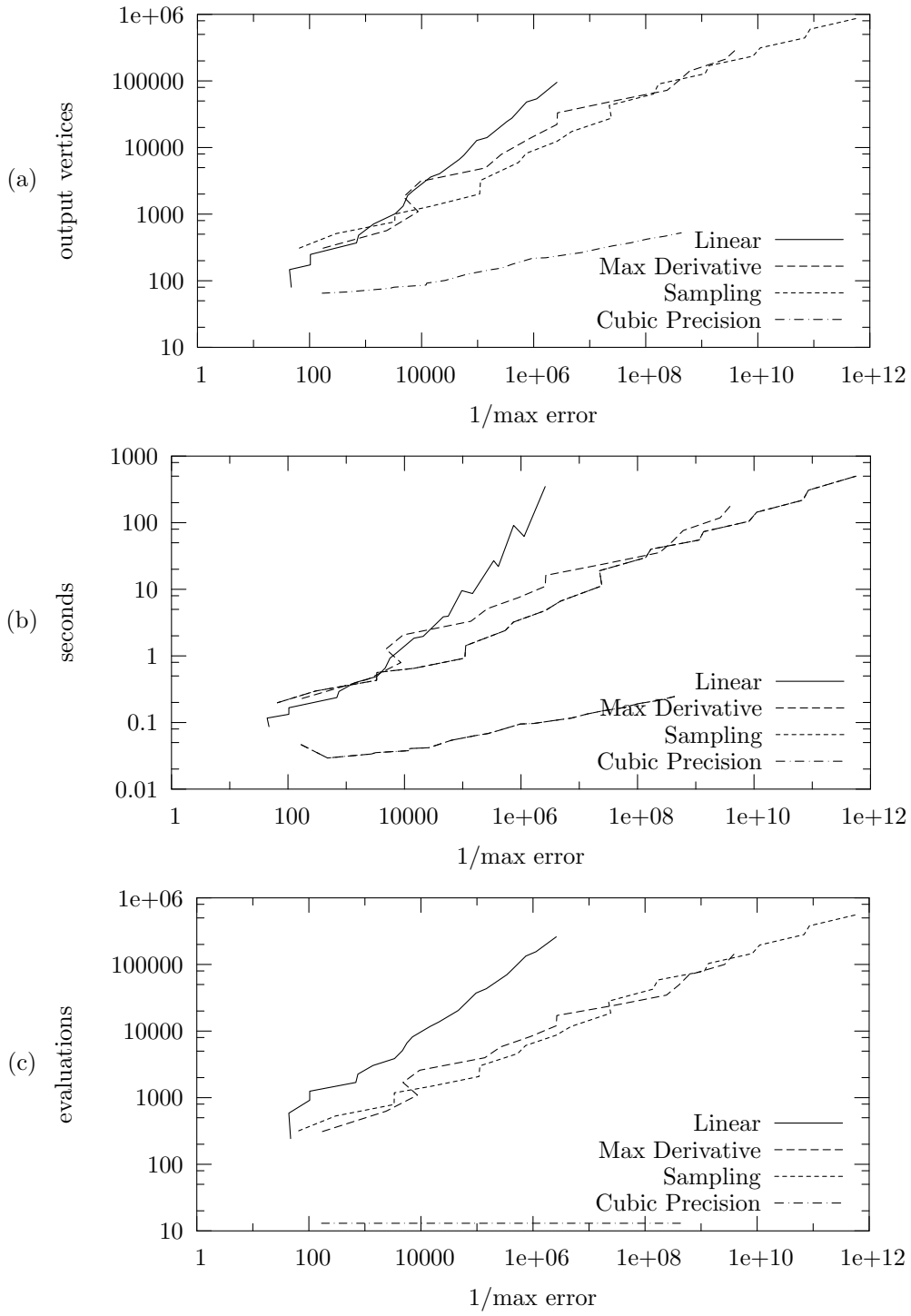


Figure 3.36: These plots graph (a) the size of the control polygon, (b) the run-time in seconds, and (c) the number of function and gradient evaluations, against one over the maximum error for linear and cubic contouring of f_θ.

Chapter 4

Contouring C^1 functions of three variables

Let $f : [0,1]^3 \to \mathbf{R}$ be C^1, and assume for simplicity that $\nabla f \neq 0$ on the zero set $C = \{x \in [0,1]^3 | f(x) = 0\}$. We wish to find the contours C, which will consist of surfaces contained in the domain $D = [0,1]^3$ of f.

4.1 Previous work

A number of methods for 3D contouring have been developed. Many are based on "Marching cubes" [59], which:

- assumes that the function is known on a regular grid,
- performs piecewise-linear interpolation along the edges of the grid to find the vertices of the resulting mesh,
- uses a look-up table based on the signs of function values at cube corners to find out how to triangulate the vertices on the edges of that cube, and
- outputs piecewise-linear contours with roughly uniform resolution (a few polygons per cube intersecting the surface).

The marching cubes algorithm has several drawbacks:

- It is patented in the United States [23, 25].
- The look-up table has 2^{2^n} entries for contouring in dimension n. For three dimensions, that means 256 cases to compute, which can be error prone. Luckily, this reduces to 14 cases after taking symmetry into account. For higher dimensions, the number of cases is prohibitive.

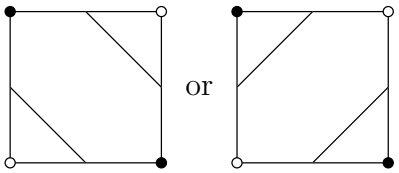


Figure 4.1: If a square has alternating signs at the corners, there are two possible ways for marching cubes to contour the interior of the square.

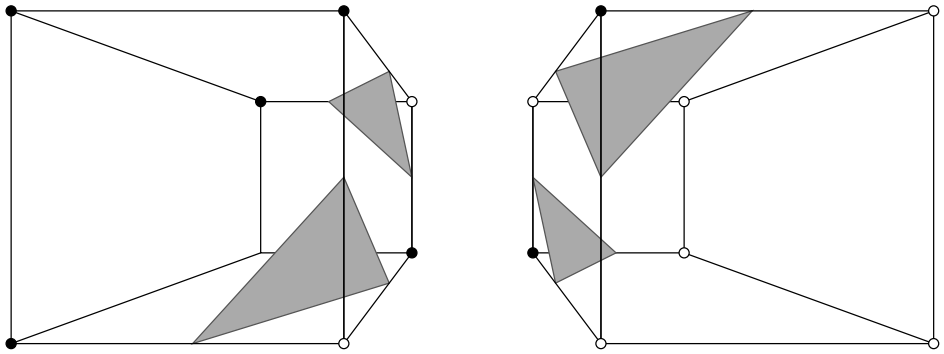


Figure 4.2: The ambiguity of contouring the faces of the cube can lead to meshes with gaps.

- It does not naturally support adaptive grids.
- Some cases in the table are ambiguous. This is a direct result of determining the contours from the corner data rather than an approximation to the function.

The last point is particular cause for concern as it can result in meshes with gaps or holes in 3D [68]. In 2D, the only ambiguous cases are where the corners alternate in sign (see Figure 4.1). In the 3D case, consider a face shared between two cubes. If the signs of the corners alternate, the two cubes should be consistent about how the face is contoured, otherwise gaps result, as in Figure 4.2. The original formulation of marching cubes [59] gave a method for generating the table that was inconsistent on faces, and so would output meshes that were not closed.

There are many ways of disambiguating these cases. In general there is a trade-off between fidelity to the function values and simplicity. Gelder and Wilhelms [44] have an excellent review of the different methods, including some statistics indicating how often various ambiguities arise. In summary:

- The simplest fix is to make some convention, such as insisting that the negative (or positive) corners are always in the same component. This allows a (carefully chosen) 256 entry table to be used, and so admits a fast implementation [7].
- Wyvill et al. [113] used the average value of the corners of the face to decide which corners to

connect. This introduces many sub-cases which increases the implementation complexity but only modestly affects speed.

- Neilson and Hamann [68] consider the contour topology induced from bilinear interpolation on ambiguous faces. In this case the contours form a hyperbola, but once the topology is resolved polygons are output as usual.

- A more sophisticated approach is to consider information from multiple cells. Gelder and Wilhelms [44] compute a gradient estimate at each of the cell corners. This is then fed into gradient consistency heuristics to determine the topology on ambiguous faces. It correctly resolves the topology of quadratic functions.

Most of these methods require additional points in the interior of some cells to allow the contours to be tessellated or to prevent intersecting contours.

There has been some research on marching cubes with adaptive meshes. Weber et al. [107] have a method using adaptive meshes with cell-centered data. They use additional look-up tables for the shapes (such as pyramids and triangular prisms) used to stitch between two resolution levels.

Bloomenthal [17] uses an octree to adaptively sample the function. He resolves ambiguous cases by adding a vertex to the center of the cube, and then dividing the cube into 12 tetrahedra. His polygonalization routine can deal with adjacent octree cells at different levels of refinement.

There has been some work on improving marching cubes in other directions. For example, Ju et al. [56] not only use an octree to adaptively sample the function, but also make use of normal data at intersection points. They generate a single vertex per cell instead of a vertex per edge, greatly reducing the number of polygons output.

A simpler solution to the ambiguity problem is to use simplices instead of cubes. An n-simplex is the convex hull of $n+1$ points in $\mathbf{R}^n$, that is, a triangle in 2D or a tetrahedron in 3D. Using simplices has a number of consequences:

- There are only 2^{n+1} possible choices of signs at the vertices. Therefore a table-based approach can use a table of size 8 in 2D, 16 in 3D, or 32 in 4D.

- Of those cases, only $\lceil \frac{n+2}{2} \rceil$ are different after symmetry. In 3D, for example, either the vertices are all the same sign, one is different, or two vertices have each sign.

- The contours produced are the exact contours of piecewise-linear interpolation on each tetrahedron. There are no ambiguous cases, and the function that is actually contoured is represented explicitly.

- If the data lies on a regular grid, there are several ways to generate a tetrahedral mesh with vertices on the grid (e.g. Section 4.2).

- There are simpler means of adaptively sampling the function to contour.

The main disadvantage of using simplices is an increase in the number of polygons output. In 3D, this technique is called *marching tetrahedra*. Payne and Tog [70] first applied this technique in order to visualize the subcortex of a brain from volumetric neurobiological data. They used a decomposition of the unit cube into 5 tetrahedra — four associated with corners of the cube and one contained in the interior of the cube.

Zhou et al. [114] have a hybrid technique that combines a tetrahedral decomposition of a uniform cubical grid with bilinear interpolation on the faces. This technique is very similar to Neilson and Hamann's technique [68], but considers many fewer cases.k

Zhou et al. [115] gave a multiresolution framework for approximating volume data. They produce an adaptive tetrahedral mesh that approximates volumetric data and use marching tetrahedra to contour the results. Gerstner [45, 46] used an adaptive tetrahedral mesh to visualize several (transparent) isosurfaces simultaneously.

Petersen et al. [74] uses a cubic interpolant to approximate a function, and then adaptively computes a linear approximation within a small tolerance of the cubic. The resulting approximations live on tetrahedra, which are easily contoured once the linear approximation is constructed. Longest-edge bisection maintains a consistent mesh.

Witkin and Heckbert [109] gave a method for interactive manipulation of implicit surfaces. They put a large number of mutually-repelling particles, called *floaters*, on the surface. These were used as a dynamic visual representation of the surface and as handles on the surface that could be directly manipulated. During editting pauses, the floaters were connected into a polygonal representation for the surface. Hart [53] has empolyed Morse theory to enable and detect changes in the surface topology in this interactive process.

Bottino et al. [19] described the *shrinkwrapping* technique for contouring an implicit surface. They assumed that the function to contour was of the form $F_T(x) = f(x) - T$ where $f(x) \to 0$ as $|x| \to \infty$. This type of function often arises when modelling implicit surfaces. They start with a large sphere that approximates the contours of F_t with t close to zero. They then move the vertices of the sphere as they increase t up to T. The mesh undergoes topological changes when t passes through any critical value of f.

Stander and Hart [97] explain how this technique is an application of Morse theory. They give an alternate method called *inflation* that starts with a large value of t which they decrease until they get the contour. They then address the problem of changing this mesh as the function is interactively altered.

The technique described below also uses Morse theory to resolve the topology of a surface, but by considering the critical points of a height function applied to the contour. This is similar to a recent technique of Hart [54], where he considers the problem of unwanted blending when designing implicit surfaces.

The algorithm given in this chapter samples the function adaptively and outputs a mesh with higher sample density in regions of high curvature. It performs cubic interpolation of func-

tion values and gradients. Previous techniques almost all output linear (polygonal) contours. Our algorithm outputs a mesh of cubic and quartic Bézier patches.

For the related problem of intersecting two surfaces, see Section 4.5 below.

4.2 Adaptive sampling and meshing

Given: $f : D \to \mathbf{R}$, we want $C = f^{-1}(0)$. We will, as before, first fit an adaptive piecewise-cubic approximation to f. The three-dimensional analog of a triangular (2-simplex) Bézier patch is defined on a tetrahedron (a 3-simplex). Thus our first goal is to create an adaptive conforming tetrahedral mesh of the domain D, the unit cube $[0, 1]^3$.

In 2D, a mesh is conforming as long as it has no hanging nodes, that is, having no vertices in the interior of an element's edge. This criteria is insufficient in higher dimensions. In 3D, a mesh is conforming if every intersection of two tetrahedra is a face, edge, or vertex of the mesh.

Maubach [64] reviews several adaptive 3D meshes, including generalizations of red-green triangulations, longest-edge-bisection meshes, and newest-vertex-bisection meshes.

Bey [15] generalized red-green triangulations to 3D, and later to arbitrary dimensions $n \geq 2$. The regular, or *red*, refinement rule divides a tetrahedron into 2^d pieces, each congruent to the original tetrahedron. To maintain a conforming mesh, a closure rule, called *green refinement*, is used to connect different levels of refinement. The green closure rule greatly increases the number of cases the algorithm needs to consider [83].

Tetrahedra bisection techniques have simpler algorithms for maintaining consistency, and use fewer tetrahedrons due to their finer granularity. Rivara [80] has a generalization of longest-edge-bisection. However, techniques using only the combinatorial topology instead of the edge lengths result in fewer conjugacy classes and faster code. These are generalizations of binary triangle trees (Section 3.3). The pioneering work by Bänsch [11] applies the technique to solving the three-dimensional Navier-Stokes equations for fluid flow [10], and has been extensively refined [71, 57, 63, 115, 6]. We follow the approach of Maubach [63] since it has been generalized to higher dimensions, is simple to implement, and has received extensive analysis [64].

Maubach [63] generalizes these techniques to any dimension $n \geq 2$. The case $n = 2$ is equivalent (except for vertex labels) to binary triangle trees. This technique implicitly labels the the vertices of each simplex by keeping them in a specific order. He gives rules for subdividing groups of tetrahedrons that share an edge that maintain consistency of the mesh. Unfortunately, there is no known procedure for consistently ordering the vertices of all the simplices of a general simplicial complex.

The initialization step creates a tetrahedralization of the unit n-cube with correct vertex ordering. This procedure creates a cube out of $n!$ n-simplices, one for every n-permutation. If π is

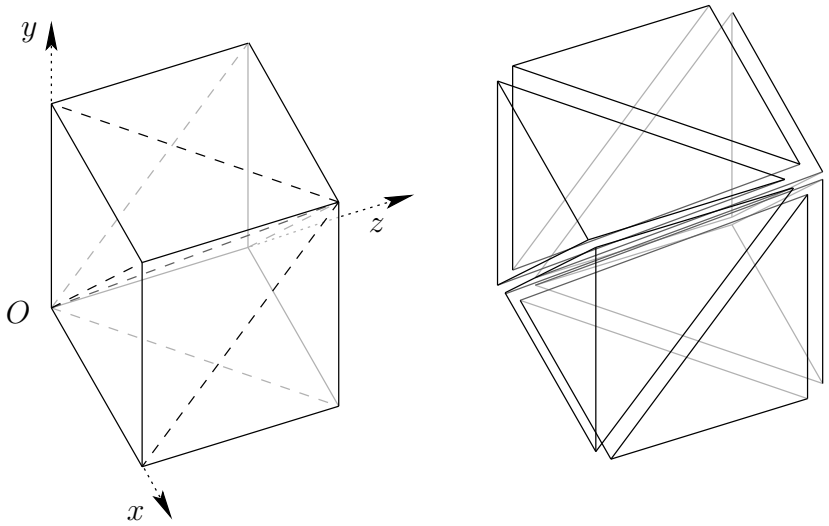


Figure 4.3: A simplicial complex for the cube consisting of 6 tetrahedra

a permutation of $1, \ldots, n$, let $\pi_i = (\delta_1, \ldots, \delta_n) \in \mathbf{R}^n$ where

$$\delta_j = \begin{cases} 1 & \text{if } \pi^{-1}(j) \leq i, \\ 0 & \text{otherwise.} \end{cases}$$

From this we define a simplex $T_\pi = T_{(\pi(1)\ldots\pi(n))}$ by the ordered list of vertices $\pi_0, \pi_1, \ldots, \pi_n$. The collection $\{T_\pi\}$ fpr all n-permutations π is a conforming simplicial mesh of the unit cube. For $n = 3$, this gives six tetrahedra (Figure 4.3):

$$\begin{aligned}
T_{(1\ 2\ 3)} &: (0,0,0),\ (1,0,0),\ (1,1,0),\ (1,1,1) \\
T_{(1\ 3\ 2)} &: (0,0,0),\ (1,0,0),\ (1,0,1),\ (1,1,1) \\
T_{(2\ 1\ 3)} &: (0,0,0),\ (0,1,0),\ (1,1,0),\ (1,1,1) \\
T_{(3\ 1\ 2)} &: (0,0,0),\ (0,1,0),\ (0,1,1),\ (1,1,1) \\
T_{(3\ 2\ 1)} &: (0,0,0),\ (0,0,1),\ (0,1,1),\ (1,1,1) \\
T_{(2\ 3\ 1)} &: (0,0,0),\ (0,0,1),\ (1,0,1),\ (1,1,1)\,.
\end{aligned}$$

We assign each of these simplices a *level* of 0.

Maubach [64] derives a formula for $\textsc{Neighbor}(T_\pi, l)$, the neighbor to T_π across the face opposite T_π's l^{th} vertex. We maintain an array of the $n+1$ neighbors along with each simplex, and this formula initializes these arrays for the base mesh above:

$$\textsc{Neighbor}(T_\pi, l) = \begin{cases} T_{\pi\circ\sigma_l} & \text{if } 1 \leq l \leq n-1, \\ 0 & \text{if } l = 0 \text{ or } l = n. \end{cases}$$

Here, 0 represents no neighbor (the l face of T_π is on the boundary), and σ_l is the transposition of l and $l+1$.

function BISECT(*Simplex* = (X_0, ..., X_n), *Level*):
$K = n - \text{MOD}(Level, n)$
$Z = \frac{1}{2}(X_0 + X_k)$
$Descendant_0 = X_0, X_1, \ldots, X_{k-1}, Z, X_{k+1}, \ldots, X_n$
$Descendant_1 = X_1, \ldots, X_k, Z, X_{k+1}, \ldots, X_n$
return [[$Descendant_0$, *Level*+1], [$Descendant_1$, *Level*+1]]

Algorithm 4.1: An n-dimensional simplex bisection algorithm

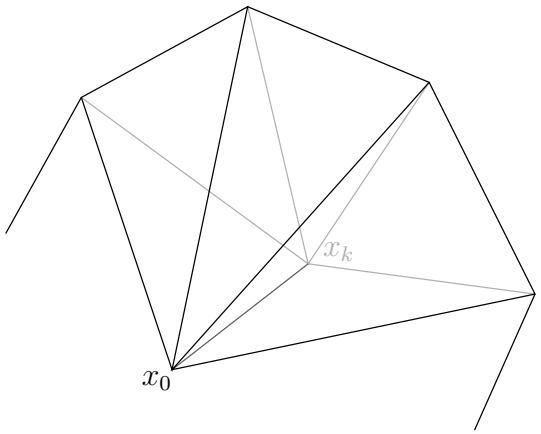


Figure 4.4: In 3D, the tetrahedra sharing an edge form a wheel.

Given a simplex $x_0, \ldots, x_n$, the *bisect* operation divides it in two along the x_0x_k edge, where k depends on the level of the simplex. This results in two simplices, one level more refined. Pseudocode for the bisect operation is given in Algorithm 4.1.

The *refine* operation is like the split operation of binary triangle trees. It performs all bisections of neighboring simplices in order to bisect a given simplex and maintain a conforming mesh. Pseudocode is given in Algorithm 4.3. The first step is to find all simplices neighboring the x_0x_k edge, refining them if they are at a coarser level (Algorithm 4.2). For $n = 3$, this neighborhood will be a wheel of tetrahedrons (Figure 4.4). Each simplex is bisected, adding a single vertex to edge x_0x_k. Finally, neighbor relationships are updated, using the relations from Maubach [64].

Maubach shows that repeated refinement does not result in simplices with small angles. In fact, the resulting simplices fall into at most n equivalence classes, where equivalence is defined up to rigid motion, reflection, and scaling.

Arnold et al. [6] give another tetrahedral bisection technique, based on tetrahedra with marked edges. Certain markings give the same refinements as Maubach's algorithm restricted to three dimensions. Consistent markings can be build for any tetrahedral base mesh.

```
function RefineEdgeNeighborhood(Simplex):
Level = Simplex.Level
K = n − Mod(Level, n)
Front = { Simplex }
Simplices = { Simplex }
while Front ≠ {}:
    NewFront = {}
    for Simplex in Front:
        for I in {1, . . . , n} \ {K}:
            Neighbor=Neighbor(Simplex, I)
            if Neighbor not in Simplices:
                while Neighbor.Level ≠ Level:
                    Refine(Neighbor)
                    Neighbor=Neighbor(Simplex, I)
                NewFront = NewFront ∪ {Neighbor}
                Simplices = Simplices ∪ {Neighbor}
    Front = NewFront
return Simplices
```

Algorithm 4.2: Algorithm to find the neighborhood of an edge to be bisected, and make sure all simplices in that neighborhood are refined to the same level.

procedure REFINE(*Simplex*):
$K = n -$ MOD(*Simplex.Level*, n)
Neighborhood = REFINEEDGENEIGHBORHOOD(*Simplex*)
for *Simplex* **in** *Neighborhood*:
    Current = BISECT(*Simplex*)
    Descendants[*Simplex*] = *Current*
    NEIGHBOR(*Current*[0], 0) = *Current*[1]
    NEIGHBOR(*Current*[1], *K*-1) = *Current*[0]
    NEIGHBOR(*Current*[0], *K*) = NEIGHBOR(*Simplex*, *K*)
    NEIGHBOR(*Current*[1], *K*) = NEIGHBOR(*Simplex*, 0)
for *Simplex* **in** *Neighborhood*:
    for L **in** $\{1, \ldots, n\} \setminus \{K\}$:
        NEIGHBOR(*Descendants*[*Simplex*][0], *L*) = *Descendants*NEIGHBOR(*Simplex*, *L*)[0]
    for L **in** $\{0, \ldots, K-2\}$:
        NEIGHBOR(*Descendants*[*Simplex*][1], *L*) = *Descendants*NEIGHBOR(*Simplex*, *L*+1)[1]
    for L **in** $\{K+1, \ldots, n\}$:
        NEIGHBOR(*Descendants*[*Simplex*][1], *L*) = *Descendants*NEIGHBOR(*Simplex*, *L*)[1]
REPLACE(*Neighborhood*, *Descendants*)

Algorithm 4.3: An n-dimensional simplex refinement algorithm

4.3 Interpolation

To get a C^1 interpolant with a single polynomial per element, a degree 9 polynomial and C^4 data at each vertex is required [3]. Luckily, there are generalizations of both Clough-Tocher and Powell-Sabin to three-dimensions. These increase the number of micro-elements per macro-element but can use much less data and a smaller degree spline to get the same continuity.

The first such generalization was given by Alfeld in 1984 [1]. In that technique, each tetrahedron is divided into 4 quintic micro-elements. It uses C^2 data to produce a C^1 interpolant with cubic precision.

Worsey and Farin described a technique in 1987 [110] that uses 12 cubic micro-elements and C^1 data to provide C^1 continuity. Their technique uses $\frac{(n+1)!}{2}$ micro-elements in dimension n and specializes to the Clough-Tocher scheme for $n = 2$.

Generalizations of Powell-Sabin interpolants have been given more recently [112]. For C^1 continuity, these require 24 quadratic micro-tetrahedra per macro-element. There are some restrictions on the triangulation. If the circumcenters of each face of the triangulation lies in the interior of that face, the circumcenters may be used as the split points for that triangulation.

There also exist multivariate rational finite element interpolants (see Hoschek and Lasser [55], page 186). An extensive survey of interpolation techniques has been given by Alfeld [3].

As in the 2D case, we use Worsey and Farin's cubic interpolant. The error analysis proceeds in much the same way as before (Section 3.5). We need to consider the dfference of $f_{ijk} = x^i y^j z^k$, with $i+j+k = 3$, and the interpolant $\tilde{f}_{ijk}$. For each monomial, we have a formula relating the difference in value and gradient at the subdivision point to the magnitude of the error $\max \left| f_{ijk} - \tilde{f}_{ijk} \right|$. Finally, we take the maximum error for a given difference over the $\binom{5}{2} = 10$ different f_{ijk}.

4.4 Finding the contour surface

We have constructed a piecewise-cubic approximation $\tilde{f}$ to f on D. This approximation is given by a collection of tetrahedrons tessellating D (specified by the location of each vertex of each tetrahedron), the topology of those tetrahedrons (specified by an ordered four-tuple of adjacent tetrahedrons for each tetrahedron), and a cubic polynomial on each tetrahedron (given by $10+6+3+1 = 20$ Bézier ordinates) that match up with C^1 continuity. If f is given as a C^1 piecewise-polynomial (of any degree), the following techniques may be applied directly, without first approximating.

We would like to construct a C^1 piecewise-cubic surface approximation to $\tilde{f}^{-1}(0)$, consisting of Bézier patches. Below, we will not distinguish between f and $\tilde{f}$. For ease of exposition, we first consider the case where C does not intersect the boundary of the domain D.

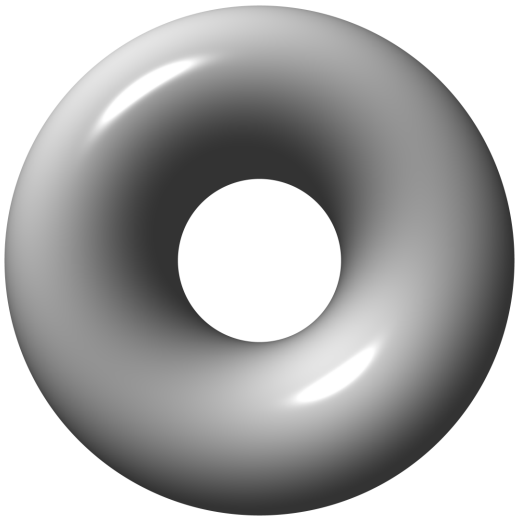


Figure 4.5: Torus $(x^2 + y^2 + z^2 + r_+^2 - r_-^2)^2 - 4r_+^2(x^2 + z^2) = 0$ with $r_- = 1$, $r_+ = 2$

4.4.1 Morse Theory

Handle bodies are a type of CW-complex used to construct manifolds explicitly [53, 65]. In the context of 2 dimensional manifolds there are three types of handles that make up a handle body:

- 0-handles: these are represented by the neighborhood of a point. Adding a 0-handle introduces a new component with a boundary topologically equivalent to a circle.
- 1-handles: these are represented by the neighborhood of a (curved) line segment. To add a 1-handle, one needs to specify where in the boundary-so-far to attach the two endpoints of the handle, and how the handle twists. For orientable manifolds (such as any zero set), there will never be an 'odd twist' as in a Möebius strip. If the two ends of a 1-handle attach to the same component of the boundary, the resulting boundary has an additional component (one circle becomes two). If the ends of a 1-handle attach to different components of the boundary, the resulting boundary has one fewer component (two circles become one). In the latter case, if the boundaries are to different components of the manifold, those components are joined as well.
- 2-handles: these are represented by a disk. 2-handles are attached to a single component of the boundary, closing it off.

Following Bott we will use the running example of a torus T oriented vertically, depicted in Figure 4.5. T will be the defined by the zero set of the function $f(x, y, z) = (x^2 + y^2 + z^2 + r_+^2 - r_-^2)^2 - 4r_+^2(x^2 + z^2)$ with $0 < r_- < r_+$ (the Figure shows $r_- = 1$, $r_+ = 2$). One way of breaking it

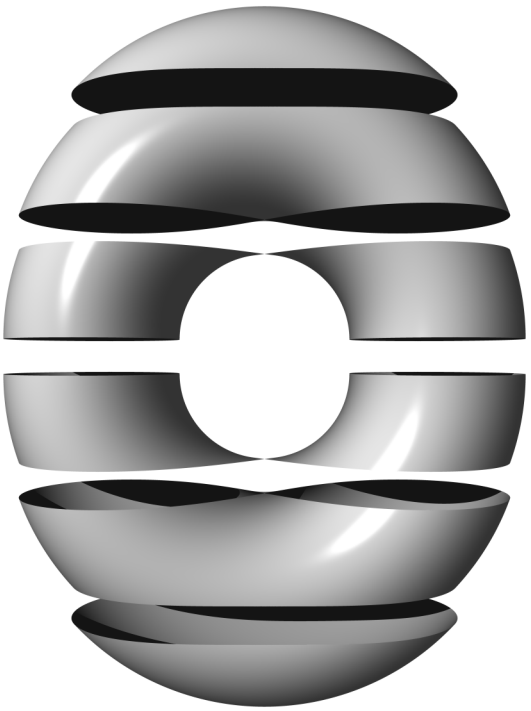


Figure 4.6: Torus from Figure 4.5 divided into $h^{-1}([-3,-2])$, $h^{-1}([-2,-1])$, $h^{-1}([-1,0])$, $h^{-1}([0,1])$, $h^{-1}([1,2])$, and $h^{-1}([2,3])$.

into four handles is shown in Figure 4.7. This shows a 0-handle at the bottom, two 1-handles in the middle, and a 2-handle at the top.

Morse theory [65] provides an algorithm for constructing a handle body for a compact manifold M given a *Morse function*: Let h be a C^2 function from M to $\mathbf{R}$. Since M is compact, the image of h is compact and therefore contained in a finite closed interval. The *critical points* of h are the points where the gradient ∇h is zero. At any critical point, h is approximated by a quadratic form with symmetric matrix or *Hessian* d^2h. A critical point is said to be *degenerate* if the Hessian is singular. If h has no degenerate critical points, then h is said to be a *Morse function* on M. The critical points of any Morse function are all isolated.

The *index* of a critical point is the number of negative eigenvalues. A *critical value* is the image of any critical point.

For our torus example, consider $h(t) = z$, where $t = (x, y, z) \in T$. The critical points are $(0, 0, -r_+ \pm r_-)$ and $(0, 0, r_+ \pm r_-)$. None of them are degenerate with this h. For $h(t) = y$ there are two circles of critical points $x^2 + z^2 = r_+^2$ $y = \pm r_-$, all degenerate.

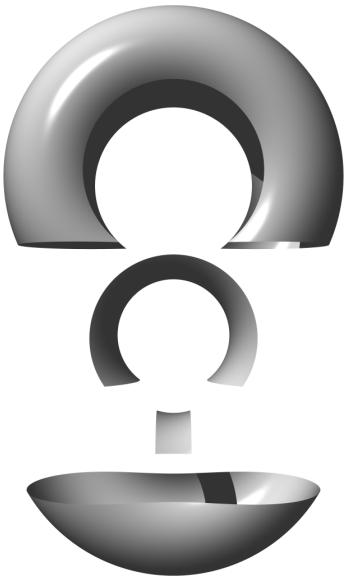


Figure 4.7: Torus from Figure 4.5 divided into a 0-handle at the bottom, two 1-handles in the middle, and a 2-handle at the top.

The key observation of Morse theory [65] is that $h^{-1}([-\infty, t])$ only changes topologically when t passes through a critical value (Figure 4.6). The change in topology is equivalent to attaching a k-handle where k is the index of the critical point (Figure 4.7). If $[t_1, t_2]$ does not contain any critical values, $h^{-1}([t_1, t_2])$ is topologically $h^{-1}(t_1) \times [t_1, t_2]$.

4.4.2 Applied Morse Theory

To construct C from f, the function to contour, we will first define h on D. We want h restricted to C to be a Morse function. Choose any unit vector $\vec{\mathbf{d_3}}$ and let $h(d) = \left\langle d, \vec{\mathbf{d_3}} \right\rangle$ for all $d \in D$. Pick $\vec{\mathbf{d_1}}, \vec{\mathbf{d_2}} \in S^2$ to form an orthonormal basis with $\vec{\mathbf{d_3}}$. Let $f_i = \frac{\partial f}{\partial \vec{\mathbf{d_i}}}$ and $f_{ij} = \frac{\partial^2 f}{\partial \vec{\mathbf{d_i}} \partial \vec{\mathbf{d_j}}}$ for $i, j \in \{1, 2, 3\}$.

h has critical points where ∇f is parallel to $\vec{\mathbf{d_3}}$. They are found by solving:

$$\begin{aligned} f(x, y, z) &= 0 \\ f_1(x, y, z) &= 0 \\ f_2(x, y, z) &= 0\,. \end{aligned}$$

We can solve this system for any Bézier spline f, using a modified Sherbrooke-Patrikalakis equation solver, as in Section 3.7.

If the solutions are a discrete set of points, we would like them to be non-degenerate.

That is equivalent to having a nonzero value for $f_{11}f_{22} - f_{12}^2$. If not, we choose a new $\vec{\mathbf{d_3}}$ (and corresponding $\vec{\mathbf{d_1}}, \vec{\mathbf{d_2}}$). By Sard's theorem [96], for almost all choices of $\vec{\mathbf{d_3}}$, zero will be a regular value of the function $g = (f, f_1, f_2) : C \to \mathbf{R}^3$. We need to assume f_3 is nonzero at the critical points, that is $\nabla f(x, y, z) \neq 0$ for all (x, y, z) where $f(x, y, z) = 0$. Otherwise C may not have the topology of a manifold.

For our torus example, the y vector is a bad choice for $\vec{\mathbf{d_3}}$, but anything else will work. With $\vec{\mathbf{d_1}}, \vec{\mathbf{d_2}}, \vec{\mathbf{d_3}} = \vec{\mathbf{x}}, \vec{\mathbf{y}}, \vec{\mathbf{z}}$, we solve

$$\begin{aligned} f(x,y,z) &= (x^2+y^2+z^2+r_+^2-r_-^2)^2 - 4r_+^2(x^2+z^2) = 0 \\ f_1 = f_x &= 4x\left(x^2+y^2+z^2-r_+^2-r_-^2\right) = 0 \\ f_2 = f_y &= 4y\left(x^2+y^2+z^2+r_+^2-r_-^2\right) = 0\,. \end{aligned}$$

So either $x = 0$ or $x^2+y^2+z^2 = r_+^2+r_-^2$. Either $y = 0$ or $x^2+y^2+z^2+r_+^2 = r_-^2$, but $x^2+y^2+z^2 \geq 0$ and $r_+ > r_-$, so $y = 0$. Substituting $y = 0$ and $x^2 + z^2 = r_+^2 + r_-^2$ into $f = 0$ gives

$$\begin{aligned} \left(r_+^2+r_-^2+r_+^2-r_-^2\right)^2 &= 4r_+^2\left(r_+^2+r_-^2\right) \\ \left(2r_+^2\right)^2 &= 4r_+^4+4r_+^2r_-^2 \\ 4r_+^4 &= 4r_+^4+4r_+^2r_-^2 \\ 0 &= 4r_+^2r_-^2\,. \end{aligned}$$

Since r_+ and r_- are both nonzero, this is impossible. We conclude $x = 0$ (in addition to $y = 0$) at all critical points. Plugging $x = 0$ and $y = 0$ into $f = 0$ gives a fourth degree polynomial equation:

$$\begin{aligned} \left(z^2+r_+^2-r_-^2\right)^2 &= 4r_+^2z^2 \\ z^4+r_+^4+r_-^4+2z^2r_+^2-2z^2r_-^2-2r_+^2r_-^2 &= 4r_+^2z^2 \\ \left(z^2\right)^2-\left(2r_+^2+2r_-^2\right)\left(z^2\right)+\left(r_+^4+r_-^4-2r_+^2r_-^2\right) &= 0\,. \end{aligned}$$

Applying the quadratic formula gives

$$\begin{aligned} z^2 &= \frac{\left(2r_+^2+2r_-^2\right) \pm \sqrt{\left(2r_+^2+2r_-^2\right)^2 - 4\left(r_+^4+r_-^4-2r_+^2r_-^2\right)}}{2} \\ &= r_+^2+r_-^2 \pm \sqrt{\left(r_+^2+r_-^2\right)^2-\left(r_+^4+r_-^4-2r_+^2r_-^2\right)} \\ &= r_+^2+r_-^2 \pm \sqrt{r_+^4+2r_+^2r_-^2+r_-^4-\left(r_+^4+r_-^4-2r_+^2r_-^2\right)} \\ &= r_+^2+r_-^2 \pm \sqrt{4r_+^2r_-^2} \\ &= r_+^2+r_-^2 \pm 2r_+r_- \\ &= \left(r_+ \pm r_-\right)^2\,. \end{aligned}$$

So $z = r_+ \pm r_-$ or $z = -(r_+ \pm r_-)$, giving critical points $(0, 0, -r_+ \pm r_-)$ and $(0, 0, r_+ \pm r_-)$ as before. To classify the critical points, we need to compute the second derivatives:

$$\begin{aligned}
f_{xx}(x,y,z) &= 4\left(x^2+y^2+z^2-r_+^2-r_-^2\right)\\
f_{yy}(x,y,z) &= 4\left(x^2+3y^3+z^2+r_+^2-r_-^2\right)\\
f_{xy}(x,y,z) &= 8xy\\
f_{xx}(0,0,z) &= 4\left(z^2-r_+^2-r_-^2\right)\\
f_{yy}(0,0,z) &= 4\left(z^2+r_+^2-r_-^2\right)\\
f_{xy}(0,0,z) &= 0\,.
\end{aligned}$$

We can find local quadratic approximations at each critical point given the length of the gradient. Since f_x and f_y are both zero, we only need

$$\begin{aligned}
f_z(x,y,z) &= 4z\left(x^2+y^2+z^2-r_+^2-r_-^2\right)\\
f_z(0,0,z) &= 4z\left(z^2-r_+^2-r_-^2\right)\,.
\end{aligned}$$

We can now give a parametric form of the quadratic approximation to the surface C near a critical point (x_0,y_0,z_0). A quadratic approximation

$$f\left((x_0,y_0,z_0)+u\vec{\mathbf{d_1}}+v\vec{\mathbf{d_2}}+w\vec{\mathbf{d_3}}\right)=wf_3+f_{11}\frac{u^2}{2}+f_{12}uv+f_{22}\frac{v^2}{2}$$

to f follows from Taylor's theorem since $f=f_1=f_2=0$ at (x_0,y_0,z_0). Setting $f=0$ and solving for w gives:

$$w=\frac{f_{11}u^2+2f_{12}uv+f_{22}}{2f_3}\,.$$

Rewriting this in parametric form gives the result:

$$C(u,v)=(x_0,y_0,z_0)+u\vec{\mathbf{d_1}}+v\vec{\mathbf{d_2}}-\frac{f_{11}u^2+2f_{12}uv+f_{22}v^2}{2f_3}\vec{\mathbf{d_3}}\,.$$

If we diagonalize the quadratic form

$$H_f=\frac{-1}{2f_3}\begin{bmatrix} f_{11} & f_{12}\\ f_{12} & f_{22}\end{bmatrix}$$

we get a change of basis that eliminates the uv term in the local parameterization. This also allows us to determine the index of the critical point (the number of negative eigenvalues). We use CLAPACK [5] routines for solving the eigensystem.

For the torus, the quadratic form is already diagonal:

$$\begin{aligned}
H_f &= \frac{-1}{2f_z}\begin{bmatrix} f_{xx} & f_{xy}\\ f_{xy} & f_{yy}\end{bmatrix}\\
&= \frac{-1}{8z\left(z^2-r_+^2-r_-^2\right)}\begin{bmatrix} 4\left(z^2-r_+^2-r_-^2\right) & 0\\ 0 & 4\left(z^2+r_+^2-r_-^2\right)\end{bmatrix}\\
&= \frac{-1}{2z\left(z^2-r_+^2-r_-^2\right)}\begin{bmatrix} z^2-r_+^2-r_-^2 & 0\\ 0 & z^2+r_+^2-r_-^2\end{bmatrix}\,.
\end{aligned}$$

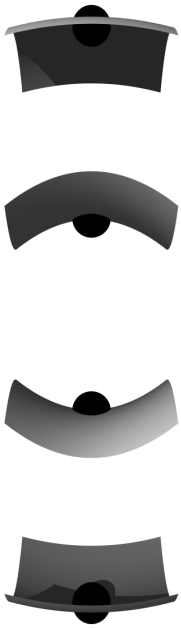


Figure 4.8: Quadratic approximations to the torus from Figure 4.5 at the four critical points

With $r_- = 1$ and $r_+ = 2$, the resulting approximations (depicted in Figure 4.8) are:

$$\begin{aligned} z_1 &= -\frac{x^2}{6} - \frac{y^2}{2} + 3 \\ z_2 &= -\frac{x^2}{2} + \frac{y^2}{2} + 1 \\ z_3 &= +\frac{x^2}{2} - \frac{y^2}{2} - 1 \\ z_4 &= +\frac{x^2}{6} + \frac{y^2}{2} - 3\,. \end{aligned}$$

For an index 0 critical point, we get an elliptic paraboloid cup (local minimum). For an index 1 critical point, we get a hyperbolic paraboloid saddle. For an index 2 critical point, we get an elliptic paraboloid cap (local maximum).

The next step is to connect these critical points with curves lying inside the manifold. To do this efficiently, we will need some spatial data structure. Luckily, the adaptive mesh structure can fill this role as well. We proceed as follows:

1. For every cup-type critical point, form the ϵ neighborhood approximation and insert it into the spatial data structure.

2. Next, process every cap and saddle in increasing order of $\vec{\mathbf{d_3}}$ coordinate. Each of these has at least one eigenvalue corresponding to a direction where the contour is concave down. For

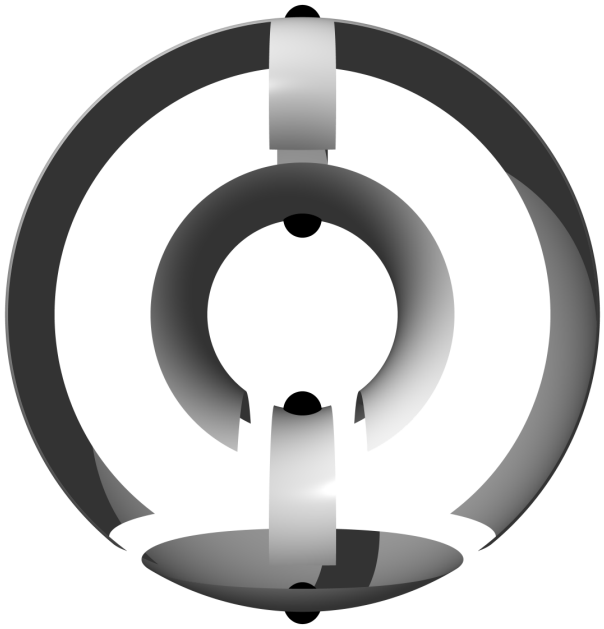


Figure 4.9: Steepest descent applied to the four critical points of the torus from Figure 4.5

each such, compute two curves $\{b_i^+, b_i^-\}$, one going in the eigenvector direction and one going in the opposite direction.

3. Each curve will follow the direction of *steepest descent* with respect to $\vec{\mathbf{d_3}}$. That is, they will follow the projection of the vector $(0, 0, -1)$ onto the contour C. See Section 4.4.3 for details. The curve is extended until it crosses the border of some patch already in our spatial data structure.

4. Add (a possibly thickened version of) this curve to the spatial data structure. Continue with the next curve or critical point.

Applying this to our torus example, we get a cup at the bottom, a 1-handle attached to the front and back of the cup, another 1-handle attached to the left and right of the first 1-handle, and a 2-handle attached to the top 1-handle and the left and right of the cup. See Figure 4.9.

Below, we will see that the steepest descent construction can always proceed until the curve reaches a critical point. We add every local minimum to the spatial data structure before tracing out the curves, then trace curves starting at critical points in increasing order of h. This ensures the tracing procedure will always terminate by hitting something in the spatial data structure.

4.4.3 Steepest descent

The steepest descent direction $v(x)$ at a point x is the projection of $\vec{\mathbf{d_3}}$ onto the tangent space to C at x. Thus $v(x) = \frac{v_0(x)}{|v_0(x)|}$ where

$$v_0(x) = \vec{\mathbf{d_3}} - \nabla f(x) \frac{\left\langle \nabla f(x), \vec{\mathbf{d_3}} \right\rangle}{|\nabla f(x)|^2} .$$

If $n(x) = \frac{\nabla f(x)}{|\nabla f(x)|}$, then $v_0(x) = \vec{\mathbf{d_3}} - n(x) \left\langle n(x), \vec{\mathbf{d_3}} \right\rangle$. Note that this formula is defined everywhere on the domain D of f except where $v_0(x) = 0$, where ∇f is parallel to $\vec{\mathbf{d_3}}$.

An exact solution to the differential equation $\frac{dc}{dt}(x) = v(x)$ will stay in the manifold since $v(x)$ is in the tangent space of C for all $x \in C$.

Given a point x_0 in C, so $f(x_0) = 0$, we want to construct a cubic Bézier curve $c(t) = \sum_{i=0}^{3} c_i \binom{3}{i} (1-t)^{3-i} t^i$ with the following properties:

$$\begin{aligned}
c(0) &= x_0 && (4.1)\\
f(c(1)) &= 0 && (4.2)\\
c''(0) & \text{ good} && (4.3)\\
c''(1) & \text{ good} && (4.4)\\
\frac{c'(0)}{|c'(0)|} &= v\left(c(0)\right) && (4.5)\\
\frac{c'(1)}{|c'(1)|} &= v\left(c(1)\right) && (4.6)\\
\frac{c'(\frac{1}{2})}{|c'\left(\frac{1}{2}\right)|} &= v\left(c\left(\frac{1}{2}\right)\right) . && (4.7)
\end{aligned}$$

Equation 4.1 simply gives the starting point for the curve. Equation 4.2 requires c to stay within the contour C. Since our eventual goal is to compute C, it is worth enforcing this constraint explicitly. Equations 4.5 through 4.7 require c to satisfy the differential equation and ensure that c stays within the contour to first order.

In equations 4.3 and 4.4, $c''(t)$ is considered good at $x = c(t)$ if $c(t)$ stays within the manifold to second order near x. This is only possible if equations 4.1, 4.2, 4.5, and 4.6 are already satisfied.

Lemma 2 *Given $c : \mathbf{R} \to \mathbf{R}^3$, and $t \in \mathbf{R}$, let $x = c(t)$. Then $f(c(t+\Delta t))$ will be $O(\Delta t^3)$ if $f(x) = 0$, $\frac{c'(t)}{|c'(t)|} = v(x)$, and $|\nabla f(x)| \langle c''(t), n(x) \rangle + f_{vv}(x) |c'(t)|^2 = 0$ where $f_{vv} = \frac{\partial^2 f}{\partial^2 v(x)}$.*

Proof: We first approximate f near $x = c(t)$ with a second-order Taylor series in the $n(x)$, $v(x)$, $b(x) = n(x) \times v(x)$ Frenet-Serret coordinate system [31]

$$f(x + \Delta x) \quad = \quad f\left(x + \Delta x_n n(x) + \Delta x_v v(x) + \Delta x_b b(x)\right)$$

$$\begin{aligned}
&\approx f(x) + \langle \Delta x, \nabla f(x) \rangle + \frac{1}{2} \Delta x^T \begin{bmatrix} f_{nn} & f_{nv} & f_{nb} \\ f_{nv} & f_{vv} & f_{vb} \\ f_{nb} & f_{vb} & f_{bb} \end{bmatrix} (x)\, \Delta x \\
&= 0 + |\nabla f(x)| \cdot \Delta x_n + \frac{1}{2} \left(f_{nn} \Delta x_n^2 + f_{vv} \Delta x_v^2 + f_{bb} \Delta x_b^2 \right) \\
&\quad + f_{nv} \Delta x_n \Delta x_v + f_{nb} \Delta x_n \Delta x_b + f_{vb} \Delta x_v \Delta x_b
\end{aligned}$$

We then approximate $c(t)$ with a second order Taylor series in the same coordinate system:

$$\begin{aligned}
c(t + \Delta t) &\approx c(t) + \Delta t c'(t) + \frac{1}{2} \Delta t^2 c''(t) \\
&= x + \Delta t |c'(t)| v(x) + \frac{1}{2} \Delta t^2 [\ \langle c''(t), n(x) \rangle\, n(x) + \\
&\qquad\qquad \langle c''(t), v(x) \rangle\, v(x) + \\
&\qquad\qquad \langle c''(t), b(x) \rangle\, b(x)\ ] \\
&= x + \Delta t |c'(t)| v(x) + \frac{1}{2} \Delta t^2 \left[c_n'' n(x) + c_v'' v(x) + c_b'' b(x) \right]
\end{aligned}$$

So $c(t + \Delta t) \approx x + \Delta x$ with

$$\Delta x = \begin{bmatrix} \frac{\Delta t^2}{2} c_n'' \\ \Delta t |c'(t)| + \frac{\Delta t^2}{2} c_v'' \\ \frac{\Delta t^2}{2} c_b'' \end{bmatrix}$$

and

$$\begin{aligned}
f(c(t + \Delta t)) &\approx f(x + \Delta x) \\
&\approx |\nabla f(x)| \frac{1}{2} \Delta t^2 c_n'' + \frac{1}{2} f_{nn} \left(\frac{\Delta t^2}{2} c_n'' \right)^2 \\
&\quad + \frac{1}{2} f_{vv} \left(\Delta t |c'(t)| + \frac{\Delta t^2}{2} c_v'' \right)^2 + \frac{1}{2} f_{bb} \left(\frac{\Delta t^2}{2} c_b'' \right)^2 \\
&\quad + f_{nv} \frac{\Delta t^2}{2} c_n'' \left(\Delta t |c'(t)| + \frac{\Delta t^2}{2} c_v'' \right) + f_{nb} \frac{\Delta t^2}{2} c_n'' \frac{\Delta t^2}{2} c_b'' \\
&\quad + f_{vb} \left(\Delta t |c'(t)| + \frac{\Delta t^2}{2} c_v'' \right) \frac{\Delta t^2}{2} c_b'' \\
&= |\nabla f(x)| \frac{1}{2} \Delta t^2 c_n'' + \frac{1}{2} f_{vv} \left(\Delta t |c'(t)| \right)^2 + O(\Delta t^3) \\
&= \frac{1}{2} \Delta t^2 \left(|\nabla f(x)| c_n'' + f_{vv} |c'(t)|^2 \right) + O(\Delta t^3)
\end{aligned}$$

so as long as $|\nabla f(x)|\, c_n'' + f_{vv}(x)\, |c'(t)|^2 = 0$, $f(c(t + \Delta t))$ is zero to second order. ∎

Note that we have 12 degrees of freedom to choose the c_i and equations 4.1–4.7 express 12 $(= 3 + 1 + 1 + 1 + 2 + 2 + 2)$ constraints. Equation 4.1 immediately determines c_0, allowing us to reduce this to 9 degrees of freedom with 9 constraints.

Our end goal is to contour the manifold, so the conditions 4.1–4.6 are the most important to satisfy. At any point x_0, the differential equation determines $v_0 = v(x_0)$. The local quadratic approximation to C near x_0 and a step size δ, we can extrapolate a position for $x_1 = c(1) = c_3$. However, x_1 will typically not lie exactly in C, so we project using

$$P(x_1) = x_1 - \nabla f(x_1) \frac{f(x_1)}{|\nabla f(x_1)|^2} \,.$$

This may need to be iterated if x_1 starts far from C. Alternatively, we can add terms that account for the second and third derivative of f. Since f is locally a cubic polynomial, this will give $P(x_1)$ in C in one step.

Once we have x_1 satisfying $f(x_1) = 0$, we can then solve 4.1–4.6 directly. We will use the last condition, 4.7, to determine when the step size δ is too large.

Let $k = |c'(0)|$ and $l = |c'(1)|$. By equation 4.5, $c'(0) = kv(c(0))$ and $c_1 = c_0 + c'(0)/3 = c_0 + kv(c_0)/3$. By equation 4.6, $c'(1) = kv(c(1))$ and $c_2 = c_3 - c'(1)/3 = c_3 - lv(c_3)/3$. Differentiating the Bézier formula for $c(t)$ twice gives us

$$\begin{aligned} c''(0) &= 6c_0 - 12c_1 + 6c_2 \\ c''(1) &= 6c_1 - 12c_2 + 6c_3 \,. \end{aligned}$$

Given c_0, c_3, $v(c_0)$, and $v(c_3)$, these second derivatives are linear functions of k and l. From Lemma 2 c must satisfy:

$$\begin{aligned} |\nabla f(c(0))| \, \langle c''(0), n(c(0)) \rangle + f_{vv}(c(0)) \, |c'(0)|^2 &= 0 \\ |\nabla f(c(1))| \, \langle c''(1), n(c(1)) \rangle + f_{vv}(c(1)) \, |c'(1)|^2 &= 0 \,. \end{aligned}$$

We know $c(\{0,1\})$, so $|\nabla f(c(\{0,1\}))|$, $n(c(\{0,1\}))$, and $f_{vv}(c(\{0,1\}))$ are determined. Since $c''(0)$ and $c''(1)$ are linear functions of k and l, the above equations reduce to:

$$\begin{aligned} a_{0k}k + a_{0l}l + b_0k^2 &= 0 \\ a_{1k}k + a_{1l}l + b_1l^2 &= 0 \end{aligned}$$

for some constants a_* and $b_{0,1}$. This reduces to a single quartic equation that we can efficiently solve with the spline root solver from Section 2.6.1 (cf. SplineForStrands from Section 3.6.1).

All that remains is the choice of δ. We would like to choose δ as large as possible subject to three conditions:

1. $c(t)$ should stay in the same component of the contour C. We inspect the value of f at $x_{\frac{1}{2}}$, the midpoint of the current segment, to check for this eventuality.

2. $c(t)$ should be within ϵ of C for all t. This condition can be verified by comparing $\left| f(x_{\frac{1}{2}}) / \nabla f(x_{\frac{1}{2}}) \right|$ to ϵ.

3. $c(t)$ should head monotonically downwards. This condition can be verified by projecting the Bézier control polygon of $c(t)$ onto the $\vec{\mathbf{d_3}}$ axis. If the control points are monotonically decreasing, so is $c(t)$. Otherwise, the first and second derivative of $\left\langle c(t), \vec{\mathbf{d_3}} \right\rangle$ are computed as 1D Bézier splines (of degrees 2 and 1 respectively). If the second derivative has no zeros, or the first derivative evaluated at the zero of the second derivative is negative, then $\left\langle c(t), \vec{\mathbf{d_3}} \right\rangle$ is monotonically decreasing. This follows from the requirement that $c(t)$ head downwards at both $t = 0$ and $t = 1$.

Due to the accuracy of the interpolating spline used to construct $c(t)$, $\delta = O(\sqrt[7]{\epsilon})$ can satisfy the second condition. We initialize δ conservatively with $\sqrt[5]{\epsilon}$. Then we double δ after any step when conditions 1–3 are satisfied, and halve δ otherwise. The steepest descent procedure is therefore:

Procedure: Given error tolerance ϵ, initial point x_0, and initial vector v_0 where $f(x_0) = 0$ and $v_0 = v(x_0)$ if $v(x_0)$ exists. Set $\delta = \sqrt[5]{\epsilon}$.

1. Extrapolate from x_0 a distance δ using v_0 and the curvature of C to find x_1.
2. Project x_1 onto the manifold by applying P until $\frac{f(x_1)}{|\nabla f(x_1)|}$ is small.
3. Compute $c(t)$ with $c(0) = x_0$, $c'(0)$ parallel to v_0, $c(1) = x_1$ and satisfying conditions 4.1–4.6 above.
4. Find $x_{\frac{1}{2}} = c(\frac{1}{2})$.
5. If $\frac{c'(\frac{1}{2})}{|c'(\frac{1}{2})|}$ is far from $v\left(x_{\frac{1}{2}}\right)$ or $f\left(x_{\frac{1}{2}}\right) / \left|\nabla f\left(x_{\frac{1}{2}}\right)\right|$ is large, divide δ in half and repeat.
6. Otherwise, output the control points for $c(t)$, double δ, set $x_0 = x_1$, $v_0 = v(x_1)$, and repeat.

4.4.4 Skeleton refinement

Our algorithm relies on the following.

Claim: The curves $\{b_i^{\pm}\}$ will divide the zero set into surfaces, each topologically equivalent to a disk.

Proof: Consider a component B of the complement $C \setminus \cup b_i^{\pm}$. Its highest point cannot be a saddle point, since we could take a small step away from a saddle perpendicular to the boundary and get higher. Therefore, the highest point must be a cap. From Morse theory, we know the cap was a disk attached along a circle in the skeleton. We have subdivided the cap into four by shooting four curves along steepest descent directions, but each piece is still topologically a disk. ∎

In the exact solution to the steepest descent problem, every curve segment $b_i^{\pm}$ would start (highest point) at a cap or a saddle and end (lowest point) at either a cup or a saddle. Since the integrator is imperfect, and the handles have been thickened, the segments may end on the boundary of any lower handle.

The last step will be to fill in the skeleton with triangular Bézier patches. In order to meet the error bounds, we need to refine the skeleton until the patches will be small enough that the error tolerance are satisfied.

We measure the length of the edges from the steepest descent process using $\int K(s) \cdot ds$, where edges are parametrized by arc-length s and $K(s)$ is the curvature. The result is nondimensional, and insensitive to scale. To reduce the error, pick the longest edge under this metric, and bisect it. Since the edge resulted from the steepest-descent process, we need to split it with a curve running horizontally. The procedure of Section 4.4.3 computes horizontal curves if the definition of $v(x)$ is changed in Equations 4.5 through 4.7. The end result is a horizontal edge added to the collection of vertical edges.

We continue to split the longest edge (horizontally if vertical, vertically otherwise) until the longest edge is shorter than the relative error threshold. If we ensure every edge satisfies $\int K(s) \cdot ds < L$, then we know a unit sphere would be subdivided until at least $2\pi/L$ edges are required to go around any great circle. In this way, we ensure that the sampling density of the surface C is proportional to its curvature.

In the case where $f^{-1}(0)$ does intersect the boundary of the domain, we need to first compute that intersection. This is done using the 2D cubic contour finder, applied to the faces of tetrahedrons on the boundary. Every minimum or maximum (with respect to the direction $\vec{\mathbf{d_3}}$) in these boundary curves has to be classified, as above. The only differences are: f_1 and f_2 will usually be nonzero and will determine the steepest descent directions, not H_f, and we never follow a direction that points out of the tetrahedron.

4.4.5 Filling in the Skeleton

Finally we take a net of cubic Bézier spline curves and find Bézier patches that fill in the skeleton. Peters [72] has given a method of doing this with cubic and quartic patches that have no cusps and fit together smoothly.

Consider a loop in the skeleton consisting of n cubic splines. Quartic patches are needed when $n > 4$ or when a certain symmetry constraint is not met. The n sided hole is filled with n triangular patches (Figure 4.10). The outer coordinates are determined from the net of curves. The next layer inside is determined from the C^1 and consistency criteria. The remaining coordinates are determined, working from the outside in, using weighted averages of the coordinates already known.

Peters proves that the resulting Bézier patches match with C^1 continuity; for surfaces, C^1 means continuity of the oriented tangent plane.

4.4.6 The 3D contouring algorithm

Here is a summary of the 3D contouring algorithm:

1. Pick a random unit vector $\vec{\mathbf{d_3}}$.

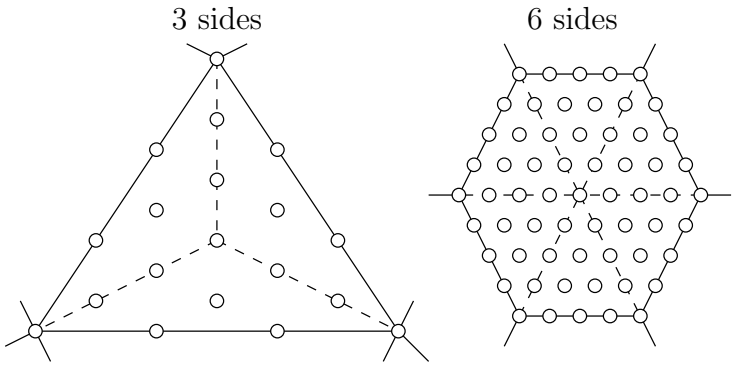


Figure 4.10: Peters' patch generation algorithm fills n-sided holes with n triangular patches.

2. Choose $\vec{\mathbf{d_1}}$ and $\vec{\mathbf{d_2}}$ to form an orthonormal basis with $\vec{\mathbf{d_3}}$.

3. Solve the system:

$$\begin{aligned} f(x,y,z) &= 0 \\ f_1(x,y,z) &= 0 \\ f_2(x,y,z) &= 0 \end{aligned}$$

   to get the critical points P. If the solutions are not a discrete collection of points, go back to step 1.

4. Compute and store f_{11}, f_{22}, and f_{12} for each element of P.

5. Compute $f_{11}f_{22} - f_{12}^2$ for each element of P. If any of them are zero, go back to step 1.

6. Compute and store f_3 for each element of P. If any of them are zero, return an error.

7. Find the eigenvalues and eigenvectors of

$$H_f = \frac{-1}{2f_3}\begin{bmatrix} f_{11} & f_{12} \\ f_{12} & f_{22} \end{bmatrix}$$

   for each element of P. Classify them by index.

8. Contour f restricted to the boundary of D. Add every local minimum and maximum (according to $\vec{\mathbf{d_3}}$) to P (assuming they are not already in P). Initialize the skeleton and spatial data structure with the curves connecting the minimums and maximums.

9. Add every index 0 element of P to the skeleton. Add the quadratic neighborhood of those elements to the spatial data structure.

10. Apply the steepest descent procedure to every index 1 and 2 element of P. The skeleton now accurately represents C.

11. Refine the skeleton until the error metric is satisfied.

12. Compute a collection of patches for every loop in the skeleton.

13. Return the patches along with the adjacency information.

4.5 Extension to surface intersection

The intersection of surfaces $C_1, C_2 \subset \mathbf{R}^3$ is generically a collection of space curves. Surface intersection algorithms find and parametrize these space curves by functions $c_i : [0,1] \to \mathbf{R}^3$. Pratt and Geisow [76] have a survey of solutions to this problem. Different techniques are needed if the surfaces are parametrized or if the surfaces are given as the contours of a function.

First, consider parametrized surfaces given by $F, G : [0,1]^2 \to \mathbf{R}^3$. In this case, the goal is to find $u, v, s, t \in [0,1]$ such that $F(u,v) - G(s,t) = 0$. The answer will be in the form of a parametrized curve $(u(\tau), v(\tau), s(\tau), t(\tau)) : [0,1] \to [0,1]^4$. The actual intersection is then given by $F(u(\tau), v(\tau))$ or equivalently $G(s(\tau), t(\tau))$.

Grandine and Klein [49] have generalized their technique for finding contours in the plane (Section 3.6.1) to find the intersection of parametrized surfaces. They work in the 2D u, v space in order to simplify the problem. There are a few complications not present in the standard 2D contouring problem:

- Contours can begin and end due to s and t crossing 0 or 1. These contours have an endpoint in the interior of the u, v space.
- The points where the contours begin and end satisfy a 3×3 system of polynomial equations. There are eight of these corresponding to the eight faces of the unit hyper-cube $[0,1]^4$.
- The critical points satisfy a 4×4 system of polynomial equations.
- Many of the tests, such as deciding if a point is a minimum or maximum, have a different form.

Müllenheim [67] derives and analyzes an iterative procedure for finding intersection points. He shows that the method has quadratic convergence, and then applies the algorithm to get a sequence of intersection points and unit tangent vectors. These are then interpolated using Hermite interpolation.

Barnhill and Kersey [12] present a marching method which traces intersection curves in the direction of tangent vectors at intersection points. Intersection curves are approximated by piecewise-linear functions. Bajaj et al. [8] present a similar method that applies to surfaces given either parametrically or as the zero set of a function, using a third-order Taylor approximation with variable-length steps. The extrapolated points are improved with Newton iteration, and special care is taken for singular points.

Second, suppose the two surfaces are each the zero set of a given function. Let $D \subset \mathbf{R}^3$ be the domain, for example the unit cube $[0,1]^3$. Given $f = (f_1, f_2) : D \to \mathbf{R}^2$, the desired zero set is $f^{-1}((0,0)) = f_1^{-1}(0) \cap f_2^{-1}(0)$. The techniques from this chapter may be used to compute $f_1^{-1}(0)$. The end result is a triangle mesh along with a Bézier patch for each element. The Bézier patches define maps from a canonical triangle $T = \{(s,t) | s,t \geq 0, s+t \leq 1\}$ into $D \subset \mathbf{R}^3$. We can compose this map with f_2 to get a map from the triangle mesh into $\mathbf{R}$. The 2D algorithm from Chapter 3 now applies directly, using the computed mesh as a base mesh that may be further refined.

Chapter 5

Scattered data interpolation

The contouring algorithms presented in Chapters 2–4 require a function f to contour. Often the value (or value and gradient) of f is known at some finite number of prescribed points, $\{P_i\}$. In some cases, the P_i may form a regular grid or have some other structure, but in general they will be placed arbitrarily. For example, measurements of oil deposits are only given where holes have been bored. Similarly, atmospheric measurements may only be known at specific weather stations.

Alfeld [3] has a survey of interpolation techniques that apply in multiple dimensions. Franke [42] has reviewed and compared the performance of many 2D interpolation methods. A more recent survey is [86], that considers methods that do not require triangulating the data sites. Below is a summary of a few of the more relevant methods. All of these methods apply in all dimensions $d \geq 1$.

5.1 Finite element interpolation

One approach is to triangulate the domain and then use an interpolant such as Clough-Tocher or Powell-Sabin (Section 3.4) on each triangle. Typically the data sites can be triangulated using either a Delaunay triangulation [27, 91] or a data-dependent triangulation [78].

The Delaunay triangulation of the set $\{P_i\}$ is a tessellation of the convex hull of the $\{P_i\}$ consisting of triangles. The only vertices of this triangulation are the P_i. The Delaunay triangulation can be defined in a number of equivalent ways:

- Define the *Voronoi tile*, or *Thiessen polyhedron*, of P_i to be the set of points closer to P_i than P_j for any $j \neq i$. Define the *Voronoi tessellation*, also known as the *Dirichlet tessellation* or the *Thiessen tessellation*, to be the polygonal mesh consisting of the Voronoi tiles of the P_i. The edges of the Voronoi tessellation are subsets of the perpendicular bisectors separating pairs of data sites. Two data sites whose Voronoi tiles share an edge are said to be *natural neighbors*. The Delaunay triangulation is the dual of the Voronoi tessellation (Figure 5.1). The edges of a Delaunay triangulation connect natural neighbors. There is a triangle in the

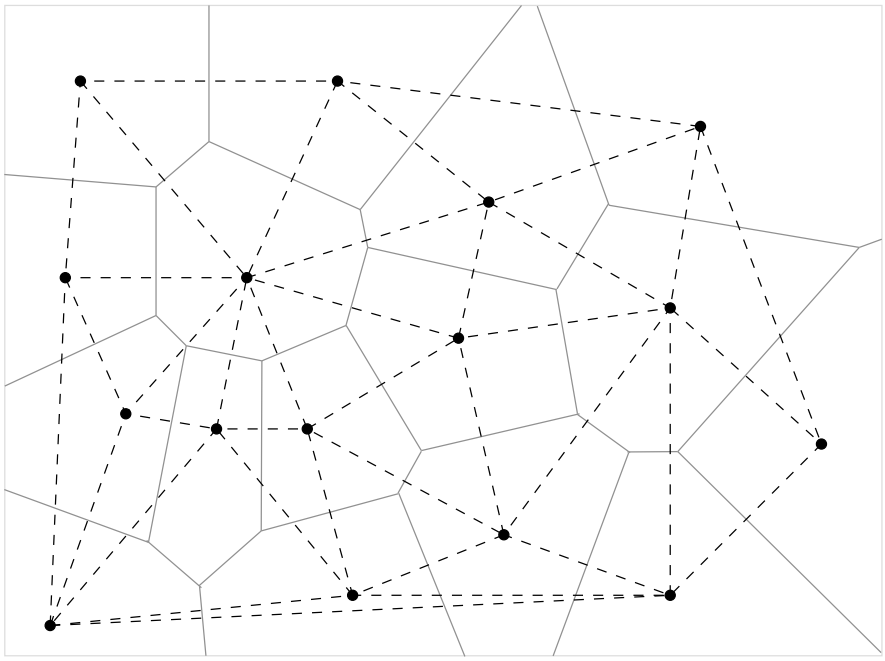


Figure 5.1: Given a fixed set of points, the Voronoi tessellation (gray lines) is dual to the Delaunay triangulation (dashed lines).

Delaunay triangulation corresponding to every vertex of the Voronoi tessellation.

- A Delaunay triangulation is also the triangulation of the P_i satisfying the *circumcircle criteria*. That is, the circumscribed circle of every triangle in the triangulation contains no data site in its interior.

- A Delaunay triangulation is the triangulation of the P_i that is globally optimal under the *max-min angle criteria*. That is, of all possible triangulations of the P_i, the Delaunay triangulation is the one with the largest minimum angle — where ties are resolved by looking at the second smallest angle, and so forth.

Since this construction gives us a piecewise-polynomial approximation to the function, the adaptive mesh stage of the contouring techniques of Chapters 2–4 may be skipped. Finding a Delaunay triangulation costs $O(N \log N)$ for N data sites. Once the triangulation has been found, interpolation is very efficient [33].

This technique can be generalized to higher dimensions. Given a tetrahedralization of the data sites, any of the interpolants of Section 4.3 can be built. Delaunay triangulations also extend to higher dimensions.

One disadvantage of this technique is that moving the data sites can cause an abrupt change in the interpolant if it causes the triangulation to change.

5.2 Radial basis functions

Finite element interpolation methods are efficient to compute since each interpolant uses a small local subset of the data. However, a higher quality interpolant can be constructed by global analysis of the function values at all data sites. One family of such methods uses Radial Basis Functions, whose value depends only on the distance to a prescribed point, so $R_i(x) = R(|x - P_i|)$. The interpolant $\tilde{f}$ is built with a radial basis function at each data site P_i plus a polynomial

$$\tilde{f}(x) = \sum_{i=1}^{N} \alpha_i R\left(|x - P_i|\right) + \sum_{j=0}^{m} \beta_j p_j(x)\,.$$

The α_i and β_j are determined by solving the linear system:

$$\begin{aligned} \tilde{f}(P_i) &= f(P_i) \qquad \forall 1 \le i \le N \\ \sum_i \alpha_{i=1}^{N} p_j(P_i) &= 0, \qquad \forall 1 \le j \le m\,. \end{aligned}$$

The last conditions ensure that the interpolant reproduces every polynomial in the span of the p_j exactly.

Different choices of radial basis functions R and the polynomials p_j lead to different interpolants. Hardy multiquadrics [52] and Duchon thin plate splines [32] are the best-known radial basis interpolants. Hardy uses $R(r) = (c^2 + r^2)^{\pm 1/2}$ for some constant c that depends on the separation between data sites. The thin plate splines are defined by $R(r) = r^{2k} \log r$ for some integer k. There are many other choices for R [86]. Some lead to poorly conditioned linear systems, when N is large. Even when the system is well-conditioned, it can be fairly time-consuming and require a lot of memory to solve.

Recent research [102, 14, 13, 20] describes how generalize the fast multipole method [50] to efficiently compute radial basis functions. These involve some method of computing the aggregate effect of clusters of data sites, and then arranging those clusters in a spatial hierarchy.

Turk and O'Brien [105] used radial basis splines to morph between models in 2D and 3D. They explain how the same techniques can be used to reconstruct a 3D surface from several (not necessarily parallel) slices. Savchenko et al. [85] discuss interpolating between slices as well as the reconstruction of a surface from scattered points.

Turk et al. [104] address construction of a function whose contours pass through a specified collection of points in 3D. Designed as a modelling tool, they allow several types of constraints on the resulting surface, such as specifying normals at the input points.

5.3 Natural neighbor interpolation

Sibson [93] developed an interpolation technique called *natural neighbor interpolation*. This technique uses Voronoi tessellations (Section 5.1), to decide how to interpolate data points. Given a

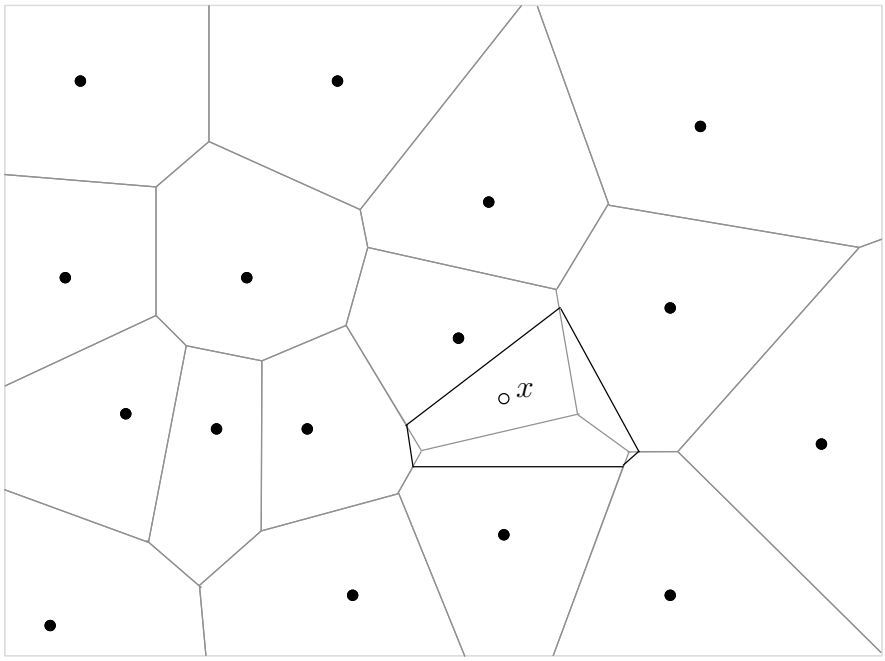


Figure 5.2: To evaluate the original natural neighbor interpolant at a point x, add x to the Voronoi tessellation for the data sites. The tile formed by adding x consists of area that used to belong to tiles from the original tessellation. The value at x is found from a weighted average of the original data sites; the weight is proportional to the area of the overlapping tiles.

Voronoi tessellation of the data sites, two sites are called *natural neighbors* if their Voronoi tiles share a face. This is equivalent to the two sites being connected by an edge in the Delaunay triangulation.

Assume we have a Voronoi tessellation $\{t_i\}$ of $\{P_i\}$. Consider adding a point x to this Voronoi tessellation (Figure 5.2). Let t_x be the Voronoi tile of x in the new tessellation. The area $|t_x|$ of t_x was part of tiles from $\{P_i\}$. Define the *natural neighbor weight* $W_i(x)$ *of* P_i to be the area of overlap between t_x and t_i divided by the area of t_x:

$$W_i(x) = \frac{|t_x \cap t_i|}{|t_x|} .$$

These weights sum to one, and express $x = \sum_i P_i W_i(x)$ as a convex combination of the data sites P_i. Thus the $W_i(x)$ define *Sibson coordinates*, similar to barycentric coordinates.

Given function values F_i at the P_i, the C^0 *natural neighbor interpolant* is

$$N_0(x) = \sum_i F_i W_i(x) .$$

This interpolant reproduces linear functions exactly, and reduces to piecewise-linear interpolation in the 1D case. It is C^∞ except at points x where the number of natural neighbors of x changes,

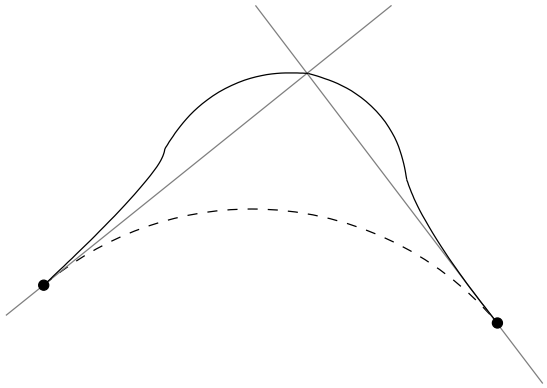


Figure 5.3: The Prussian helmet effect (black curve) of mixing first-order approximations (gray lines). We would prefer the dashed curve.

on the circumcircles of the Delaunay triangulation of the data sites $\{P_i\}$ [37]. It is C^0 at the data sites. It is local since $W_i(x)$ is zero except when x and P_i are natural neighbors.

A straightforward implementation of natural neighbor interpolation constructs the Delaunay triangulation of the data sites in $O(n \log n)$ operations. Each evaluation of the interpolant takes $O(\log n)$ time, though this can be improved if consecutive evaluations are nearby. This can be reduced to $O(n)$ and $O(1)$ time, respectively, if the data sites are distributed quasi-uniformly [86].

A C^1 interpolant requires the gradient V_i at each data site. Let $T_i : \mathbf{R}^d \to \mathbf{R}$ be the tangent plane function $x \mapsto \langle x - P_i, V_i \rangle$.

The simplest C^1 method uses the natural neighbor weights to mix the tangent planes $x \to F_i + T_i(x)$ around each P_i. The mixing function $m_i(x)$ should have value 1 at P_i, value 0 at P_j for $j \neq i$, a 0 gradient at every P_j, and $\sum_i m_i(x) = 1$. Our interpolant $x \mapsto \sum_i (F_i + T_i(x))\, m_i(x)$ then has at least linear precision. Standard choices such as $W_i(x)^2 / \sum_j W_j(x)^2$ or [93]:

$$m_i^S(x) = \frac{W_i(x)}{|x - P_i| \sum_j \frac{W_j(x)}{|x - P_j|}}$$

suffer from the *Prussian helmet* effect [93] (Figure 5.3): The interpolant must pass through the intersection of the tangent planes and thus cannot reproduce the dashed line in the figure.

Sibson [93] constructed a C^1 modification to the C^0 natural neighbor interpolant that does not suffer from the Prussian helmet effect. His approach combines the C^1 interpolant

$$H_1^S(x) = \sum_i (F_i + T_i(x))\, m_i^S(x) = \frac{\sum_i \frac{W_i(x)}{|x - P_i|} (F_i + T_i(x))}{\sum_j \frac{W_j(x)}{|x - P_j|}},$$

and the C^0 interpolant N_0 to reproduce a spherical quadratic (the span of linear functions and $x^T x$), yielding

$$N_1(x) = \frac{S_a(x) N_0(x) + S_b(x) H_1^S(s)}{S_a(x) + S_b(x)}$$

where

$$\begin{aligned} S_a(x) &= \frac{\sum_i W_i(x)|x-P_i|}{\sum_i W_i(x)/|x-P_i|}\,, \\ S_b(x) &= \sum_i W_i(x)|x-P_i|^2\,. \end{aligned}$$

Sibson's N_1 interpolant is undefined at data sites $x = P_i$, but its limit as $x \to P_i$ is F_i. Its gradient at P_i is the specified gradient V_i, it reproduces spherical quadratics, it reduces to Hermite cubic interpolation in the 1D case, and the interpolant varies C^1 smoothly as the data sites move.

Another C^1 method of natural neighbor interpolation is due to Farin [37]. Sibson coordinates $W_i(x)$ are interpreted as barycentric coordinates in a Bézier simplex. The dimension of the simplex is the number of nonzero $W_i(x)$ minus one. Each vertex of the simplex is placed at a natural neighbor of x. For linear Bézier simplices, this reduces to the standard C^0 natural neighbor interpolant. Farin uses cubic Bézier simplices to achieve C^1 continuity and quadratic precision. The ordinates of the simplex are determined in a way similar to the nine-parameter interpolant (Section 3.4.1). This method reduces to piecewise-cubic interpolation in 1D.

5.4 New Natural Neighbor Interpolant

We take a different approach to solving the Prussian helmet problem. Consider 1D Hermite interpolation, with the P_i in increasing order of x-coordinate. Restricting to the interval $[P_i, P_{i+1}]$, and letting $L = P_{i+1} - P_i$, $t = (x - P_i)/L$, the interpolant is given by a sum of cubic polynomials:

$$F_i(2t^3 - 3t^3 + 1) + V_i(Lt)(1-t)^2 + F_{i+1}(3t^3 - 2t^3) + V_{i+1}(Lt^2)(t-1)\,.$$

Rewriting this in terms of $T_i(x) = V_i(x - P_i) = V_i Lt$ and $T_{i+1}(x) = V_{i+1}(x - P_{i+1}) = V_{i+1}L(t-1)$ gives:

$$F_i(2t^3 - 3t^3 + 1) + T_i(x)(1-t)^2 + F_{i+1}(3t^3 - 2t^3) + T_{i+1}(x)(t^2)\,.$$

By symmetry the basis function for F_i is the reflection about $t = 1/2$ of the basis function for F_{i+1}: for example replacing t by $1 - t$ in $3t^3 - 2t^3$ gives

$$3(1-t)^2 - 2(1-t)^3 = 3 - 6t + 3t^2 - 2 + 6t - 6t^2 + 2t^3 = 1 + 2t^3 - 3t^2\,.$$

The basis functions for T_i and T_{i+1} are similarly symmetric. However, Hermite interpolation uses different basis functions for the function values and the tangent-line functions. Using the same weight for F_i as for $T_i(x)$ produces the Prussian helmet effect.

Since $W_i(x)$ and $W_{i+1}(x)$ restricted to $x \in [P_i, P_{i+1}]$ are exactly $1 - t$ and t, $W_i(x)^2$ is an obvious mixing function for $T_i(x)$.

Our new C^1 natural neighbor interpolant is defined by:

$$N_J(x) = \sum_i \left(F_i m_i(x) + T_i(x) W_i(x)^2\right)$$

where:

$$m_i(x) = \frac{J(W_i(x))}{\sum_j J(W_j(x))}$$

for any C^1 function $J(t)$ satisfying:

$$\begin{aligned} J(0) &= J'(0) = 0 \\ J(t) &> 0 \text{ for } t \in (0,1]\,. \end{aligned}$$

(For example, $J(t) = t^2$.) The $\sum_j J\left(W_j(x)\right)$ in the denominator of $m_i(x)$ ensures that $\sum_i m_i(x) = 1$. Furthermore, since $W_i(P_j) = \delta_{ij}$,

$$m_i(P_k) = \frac{J(W_i(P_k))}{\sum_j J(W_j(P_k))} = \frac{J(\delta_{ik})}{\sum_j J(\delta_{jk})} = \frac{J(\delta_{ik})}{J(1)} = \delta_{ik}\,.$$

So setting $x = P_k$ in $N_J(x)$ gives

$$\begin{aligned} N_J(P_k) &= \sum_i \left(F_i m_i(P_k) + T_i(P_k) W_i(P_k)^2\right) \\ &= \sum_i \left(F_i \delta_{ik} + T_i(P_k)\delta_{ik}^2\right) \\ &= F_k \cdot 1 + T_k(P_k)\cdot 1^2 + \sum_{i\neq k}\left(F_i \cdot 0 + T_i(P_k)\cdot 0^2\right) \\ &= F_k + 0 + 0 = F_k\,. \end{aligned}$$

Therefore, $N_J(x)$ interpolates the given function values at the P_k.

Observe that $\sum_j J(W_j(x)) > 0$ for all x, and so the denominator in $m_i(x)$ is never 0. So $N_J(x)$ is a weighted sum of continuous functions, and is therefore continuous. In fact, $N_J(x)$ is C^1 for all dimensions. From the properties of W_i, it is sufficient to check that the directional derivative $\frac{\partial N_J}{\partial v}(x) = (N_J)_v(x)$ is continuous at $x = P_j$ for all unit vectors $v \in R^d$. This is somewhat tricky to verify since $(W_i)_v(P_i)$ does not exist. Let $D(x) = \sum_j J(W_j(x))$ be the denominator of $m_i(x)$. Using limits, the continuity of W_i, J, and T_i, and $W_i(P_j) = \delta_{ij}$, we compute:

$$\begin{aligned} D(P_i) &= \sum_j J(W_j(P_i)) = \sum_j J(\delta_{ij}) = J(1) \\ \lim_{x\to P_i} D_v(x) &= \lim_{x\to P_i} \sum_j J'(W_j(x))(W_j)_v(x) \\ &= \lim_{x\to P_i} \left(J'(W_i(x))(W_i)_v(x) + \sum_{j\neq i} J'(W_j(x))(W_j)_v(x)\right) \\ &\quad \text{since for } j\neq i,\ J'(W_j(P_i)) = J'(0) = 0 \text{ and } (W_j)_v \text{ continuous at } P_i \\ &= \lim_{x\to P_i} J'(W_i(x))(W_i)_v(x) \\ (m_i^J)_v(x) &= \frac{J'(W_i(x))(W_i)_v(x)D(x) - J(W_i(x))D_v(x)}{D(x)^2} \ \forall x \notin \{P_j\} \\ \lim_{x\to P_j} (m_i^J)_v(x) &= \lim_{x\to P_j} \frac{J'(W_i(x))(W_i)_v(x)D(x) - J(W_i(x))D_v(x)}{D^2(x)} \end{aligned}$$

$$
\begin{aligned}
&= \lim_{x\to P_j} \frac{J'(W_i(x))(W_i)_v(x)J(1) - J(W_i(x))D_v(x)}{J(1)^2} \\
&= \lim_{x\to P_j} \frac{J'(W_i(x))(W_i)_v(x)J(1) - J(W_i(x))J'(W_j(x))(W_j)_v(x)}{J(1)^2} \\
&= \lim_{x\to P_j} \frac{J'(\delta_{ij})(W_i)_v(x)J(1) - J(\delta_{ij})J'(1)(W_j)_v(x)}{J(1)^2}\,.
\end{aligned}
$$

If $i \neq j$, this becomes,

$$
\lim_{x\to P_j} \frac{J'(0)(W_i)_v(x)J(1) - J(0)J'(1)(W_j)_v(x)}{J(1)^2} = 0
$$

since $J'(0) = J(0) = 0$ and the remaining terms are bounded. If $i = j$ we get:

$$
\lim_{x\to P_i} \frac{J'(1)(W_i)_v(x)J(1) - J(1)J'(1)(W_i)_v(x)}{J(1)^2} = 0\,.
$$

Either way, $\lim_{x\to P_j}(m_i^J)_v(x) = 0$. By Taylor's theorem [82],

$$
\limsup_{h\to 0} \frac{|m_i^J(P_j + vh) - m_i^J(P_j)|}{|h|} = \limsup_{h\to 0} |(m_i^J)_v(P_j + y_h v)|
$$

where $y_h \in (0, h)$ depends on h. Since the right-hand side is zero, we find that $(m_i^J)_v$ exists and is zero. Thus $\sum_i F_i m_i(x)$ is C^1 and has gradient zero at every P_i.

The $T_i(x)W_i(x)^2$ term satisfies:

$$
\begin{aligned}
\lim_{x\to P_j} \left(T_i(x)W_i(x)^2\right)_v &= \lim_{x\to P_j} \left((T_i)_v(x)W_i(x)^2 + T_i(x)(W_i)_v(x)W_i(x)\right) \\
&= \lim_{x\to P_j} W_i(x)\left(\langle V_i, v\rangle W_i(x) + T_i(x)(W_i)_v(x)\right)
\end{aligned}
$$

If $i \neq j$ then $W_i(x) \to 0$ and the right-hand expression is bounded so the limit is 0. If $i = j$, then

$$
\lim_{x\to P_i} W_i(x)\left(\langle V_i, v\rangle W_i(x) + T_i(x)(W_i)_v(x)\right) = \langle V_i, v\rangle + \lim_{x\to P_i} T_i(x)(W_i)_v(x)
$$

and the last term is zero since $T_i(x) \to 0$ and $(W_i)_v(x)$ is bounded. Summing the terms of $(N_J)_v$ we see that $(N_J)_v(P_i)$ exists and is equal to $\langle V_i, v\rangle$, and conclude $(\nabla N_J)(P_i) = V_i$.

For $d = 1$, there are explicit formulas for the interpolant. For x in the interval $[P_i, P_{i+1}]$, $W_j(x) = 0$ unless $j = i$ or $j = i + 1$. In that interval

$$
\begin{aligned}
N_J(x) &= F_i m_i(x) + T_i(x)W_i(x)^2 + F_{i+1}m_{i+1}(x) + T_{i+1}(x)W_{i+1}(x)^2 \\
&= \frac{F_i J(W_i(x)) + F_{i+1}J(W_{i+1}(x)}{J(W_i(x)) + J(W_{i+1}(x))} + T_i(x)W_i(x)^2 + T_{i+1}(x)W_{i+1}(x)^2\,.
\end{aligned}
$$

Let $t = (x - P_i)/(P_{i+1} - P_i)$. Then $W_i(x) = 1 - t$ and $W_{i+1}(x) = t$ so

$$
N_J(x) = \frac{F_i J(1-t) + F_{i+1}J(t}{J(1-t) + J(t)} + T_i(x)(1-t)^2 + T_{i+1}(x)t^2\,.
$$

The simplest choice for $J(t)$ is t^2, which makes $N_J(x)$ into a piecewise-rational-quadratic interpolant of the F_i. To get cubic Hermite interpolation, set $J(t) = 3t^2 - 2t^3$. Then $J(1-t) + J(t) =$

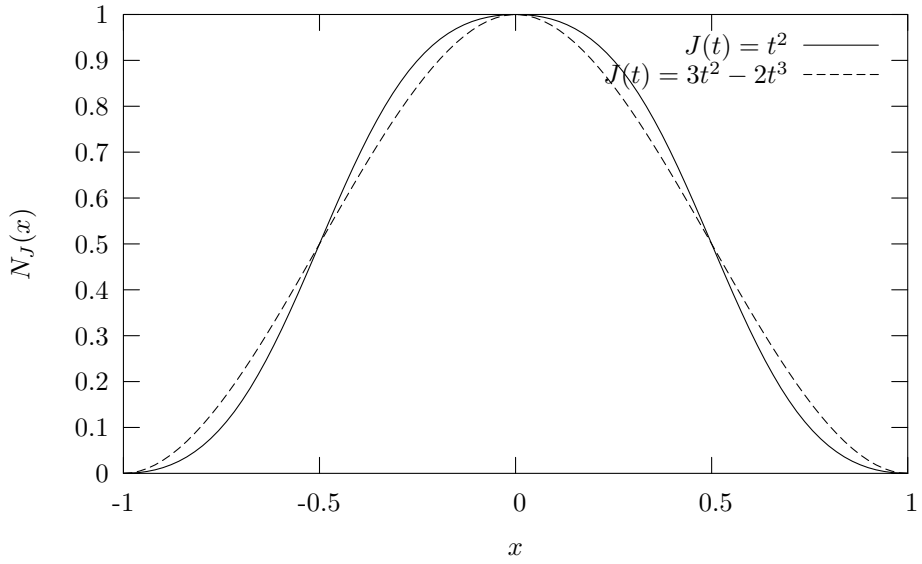


Figure 5.4: The basis functions for N_J depend on the choice of $J(t)$. Graphs of N_J with $J(t) = t^2$ and $J(t) = 3t^2 - 2t^3$ are shown.

Method	Error with 33 samples	Error with 100 samples
N_0	.289	.173
N_1	.112	.021
N_J with $J(t) = t^2$	.199	.144
N_J with $J(t) = 3t^2 - 2t^3$	.186	.107
N_{J2}	.143	.084

Table 5.1: Error between various natural neighbor interpolants and $\mathcal{F}$.

$(1 + 2t^3 - 3t^2) + (3t^2 - 2t^3) = 1$, so the denominator drops out. The numerator becomes the standard Hermite basis functions. Basis functions for N_J with these two choices of $J(t)$ are shown in Figure 5.4.

To test these methods, we apply them to Franke's function [42] (Figure 5.5):

$$\mathcal{F}(x,y) = \frac{3}{4}e^{-\frac{(9x-2)^2+(9*y-2)^2}{4}} + \frac{3}{4}e^{-\frac{(9*x+1)^2}{49}-\frac{(9*y+1)^2}{10}} + \frac{1}{2}e^{-\frac{(9*x-7)^2+(9*y-3)^2}{4}} - \frac{1}{5}e^{-(9*x-4)^2-(9*y-7)^2}.$$

We sample this function at the 33 sites $P_i \in [0,1]^2$ shown in Figure 5.6. From these samples build each of the natural neighbor interpolants, as shown in Figures 5.7(a) through 5.10(a). Our implementation uses Shewchuk's code TRIANGLE [91] to find Voronoi tessellations. The error shown in Figures 5.7(b) through 5.10(b) is the difference between the interpolant and the original function. N_1 performed the best. Replacing $J(W_i)$ with $\frac{W_i}{|x-P_i|}$ (following Sibson [93]) gave an interpolant N_{J2} that performed better than N_J without the complexity of N_1 (Table 5.1).

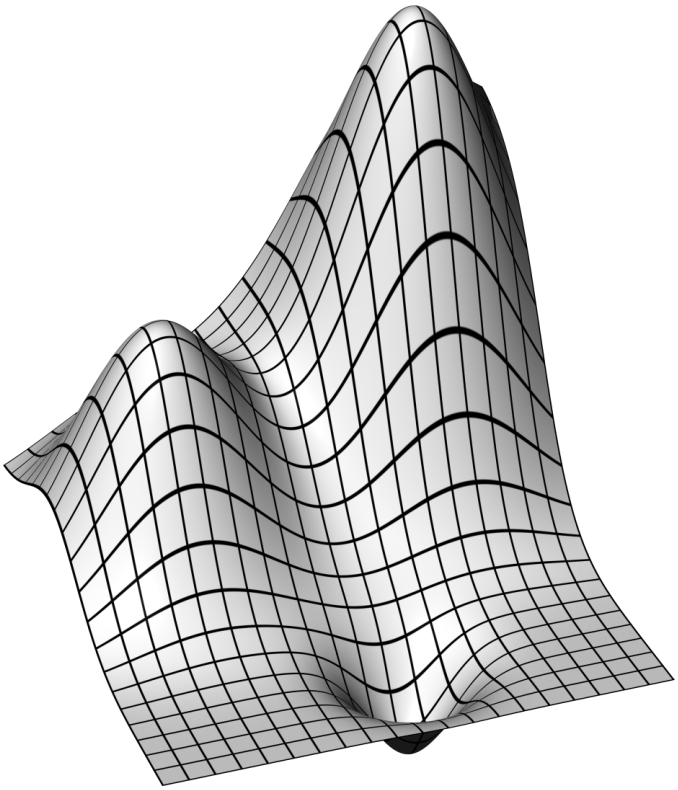


Figure 5.5: Franke's test function, $\mathcal{F}(x, y)$

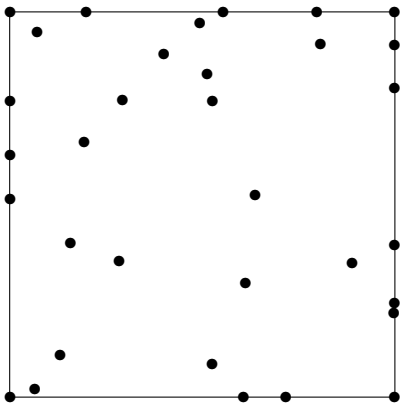
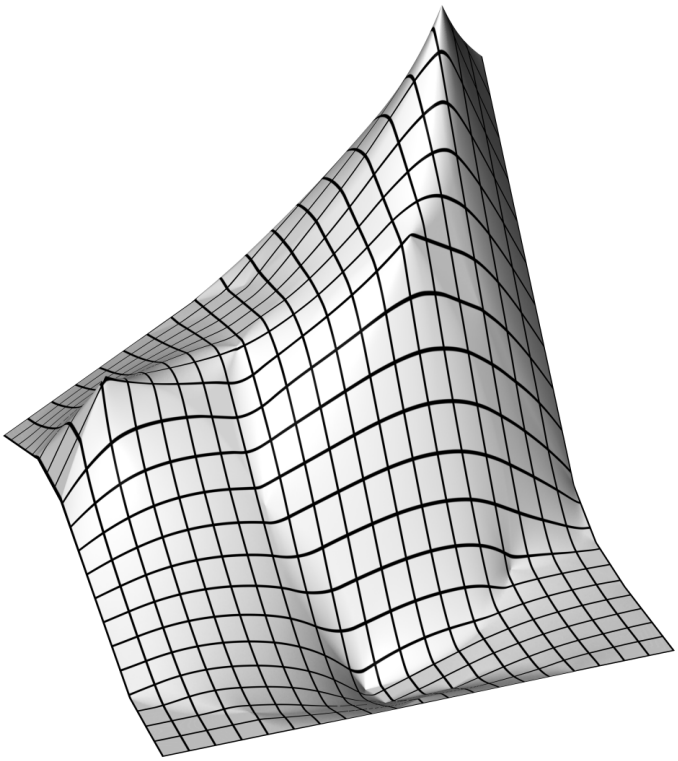
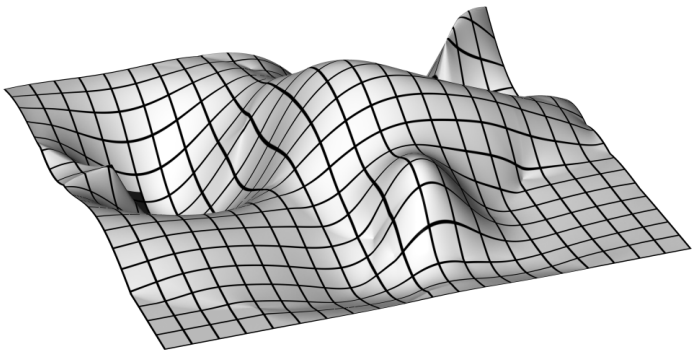


Figure 5.6: The 33 sample sites include the corner points and points on the boundary.

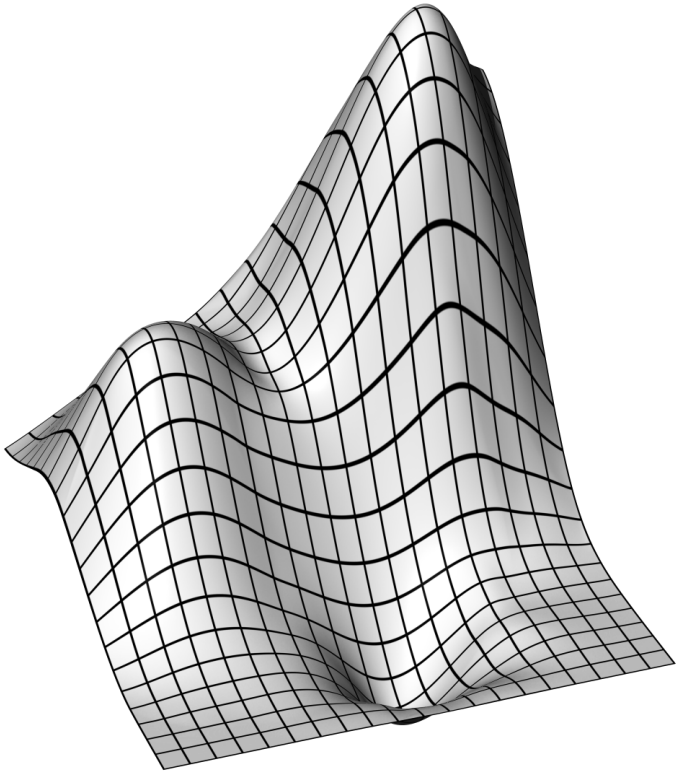


(a)

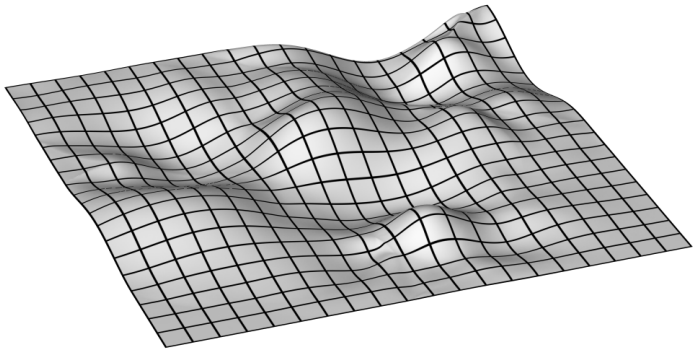


(b)

Figure 5.7: The C^0 natural neighbor interpolant (a) has corners at the data sites. The error $N_0 - \mathcal{F}$ is shown in (b).

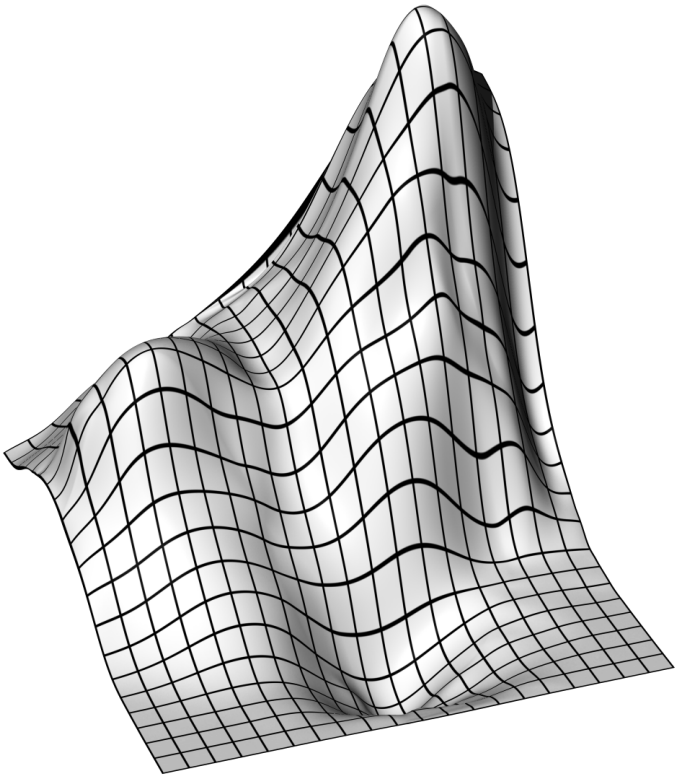


(a)

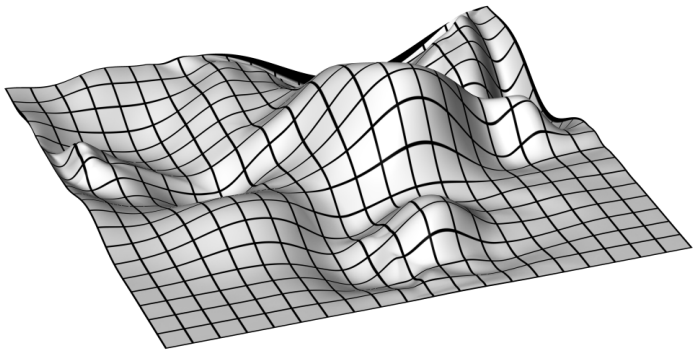


(b)

Figure 5.8: The C^1 natural neighbor interpolant (a) has spherical quadratic precision. The error $N_1 - \mathcal{F}$ is shown in (b).

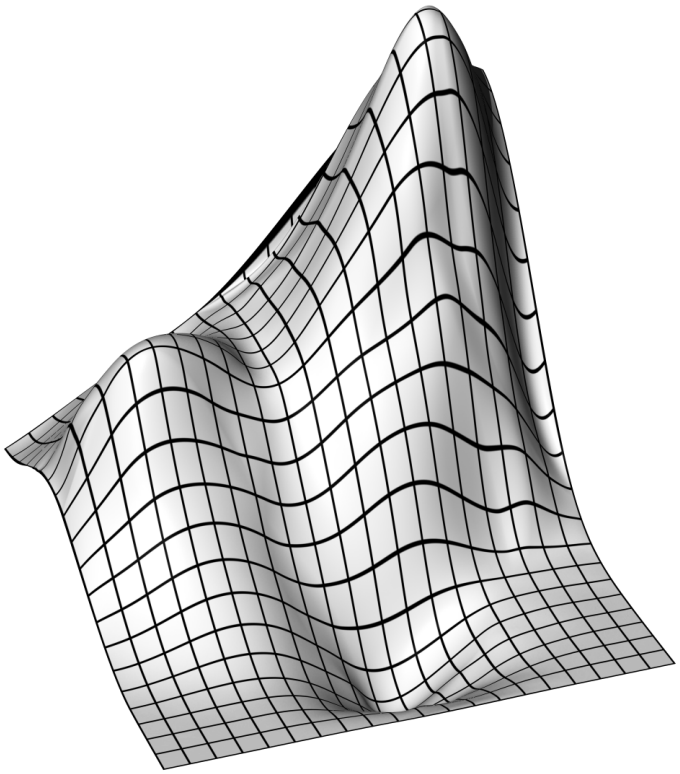


(a)

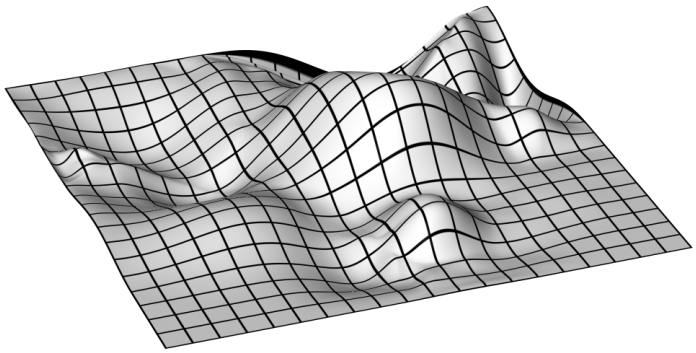


(b)

Figure 5.9: N_J (a) with $J(t) = t^2$. The error $N_J - \mathcal{F}$ is shown in (b).

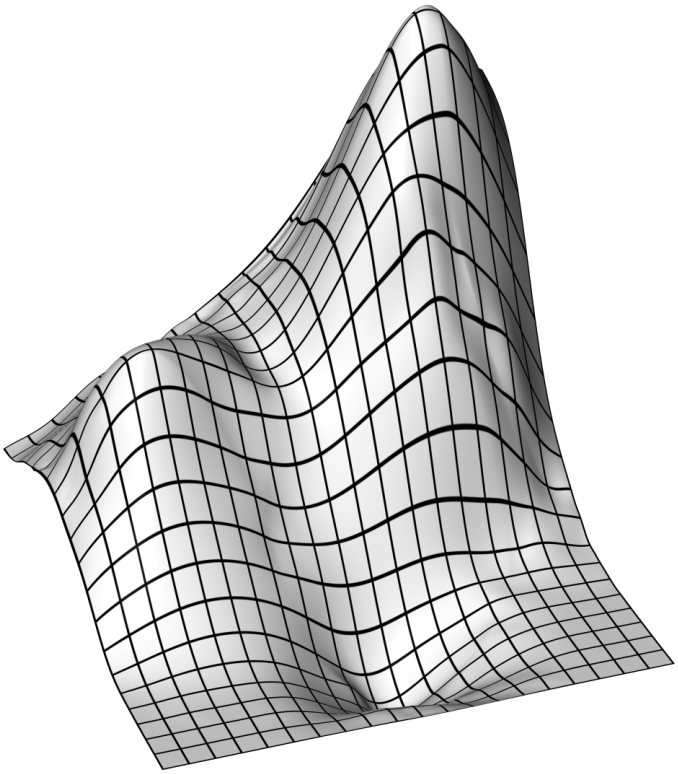


(a)

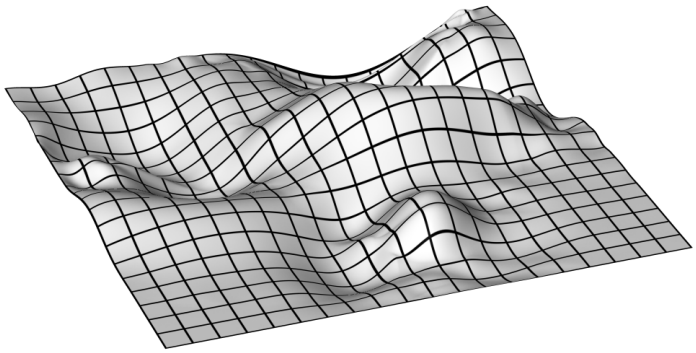


(b)

Figure 5.10: N_J (a) with $J(t) = 3t^2 - 2t^3$. The error $N_J - \mathcal{F}$ is shown in (b).



(a)



(b)

Figure 5.11: N_{J2} (a) employs the $m_i^S(x)$ mixing function. The error $N_{J2} - \mathcal{F}$ is shown in (b).

5.5 Gradient estimation

There are two situations where gradient estimation is useful. Given function values at a fixed set of data sites, gradient values at each site are necessary for natural neighbor interpolation. In this situation, global methods can be appropriate since they yield the highest quality derivatives [42]. Unfortunately, global methods are also the most computationally intensive and can be impractical for large data sets. Global methods can often be converted into local methods by applying them to a subset of points, often the nearest k points $\{x_1, \ldots x_k\}$ to the data site x_0 in question. Sometimes a weighting factor is introduced when making a method local: $(R - |x_i - x_0|)^2$ and $\left(\frac{R-|x_i-x_0|}{R|x_i-x_0|}\right)^2$ are common choices (for $R \geq |x_k - x_0|$).

Gradient estimation is also useful when f can be sampled at arbitrary points, but the gradient of the function is unavailable. Under these circumstances, we would like to estimate the gradient using the values at nearby data sites while the function is being adaptively sampled. The gradient determines whether further subdivision is necessary, so a local method can be used to compute the gradient at the new point alone. The gradients can be recomputed once the adaptive sampling is complete.

Alfeld [3] has a short summary of gradient estimation methods. Franke [42] contrasted several gradient estimation schemes to evaluate finite-element interpolation schemes. Stead [98] compared several ad hoc local schemes. Here we review a few representative techniques.

One global method is to perform a global optimization to reduce some energy functional. Renka and Cline [79] minimize the clamped elastic plate functional over all Clough-Tocher interpolants (Section 3.4.3) of a given triangulation. The resulting linear system was ill-conditioned, so iterative methods for solving the system converge slowly. Alfeld [2] minimized the integral of some derivative (typically $f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2$ or $f_{xxx}^2 + 3f_{xxy}^2 + 3f_{xyy}^2 + f_{yyy}^2$) over all possible piecewise-quintic interpolants (Section 3.4.2) over the triangulation. This results in a sparse, positive-definite linear system assembled out of the contribution from each element. Alfeld employs a standard direct method for solving this system.

Another global method is to build a global interpolant of the data, and then differentiate to find the gradient. Stead [98] found that using Hardy multiquadric interpolantation (Section 5.2) of the 20 closest points to each data site performed very well on a variety of data. This method, however, does not have any polynomial precision.

Such techniques are commonly used to perform local gradient estimation. One local approximation is constructed at each data site, usually chosen to minimize a weighted least-squares error measurement. Different techniques use different spaces of approximations and different weight functions. Stead [98] tested two of these schemes. In one, a planar least squares approximation [41] to the closest 9 points $\{x_0, \ldots, x_8\}$ to a data site x_0 is found. In the other, a quadratic least squares approximation [42] to the closest 11 points $\{x_0, \ldots, x_{10}\}$ to a data site x_0 was used. In both cases, the data site itself was included in the collection of points. The furthest point was used to

set $R = |x_k - x_0|$ in the weight function $w_{R,x_0}(x_i) = (R - |x_i - x_0|)^2$. Stead found the quadratic approximation gave very good results for simpler surfaces.

Sibson [93] used a weighting system that used the natural neighbor machinery. Let the *natural neighbor weight* from P_i to P_j be defined by $W_{ji} = W_j(P_i)$ after P_i is removed from the set of data sites. There are always at least $d+1$ natural neighbors to P_i in dimension d, which is sufficient to define a spherical quadratic passing through P_i. Of course, there may be more natural neighbors, in which case we can use the weighted least squares fit given by $V_i = H_i^{-1} y_i$, where

$$\begin{aligned} H_i &= \sum_j W_{ji} \frac{(P_j - P_i)(P_j - P_i)^T}{|P_j - P_i|^2} \text{ and} \\ y_i &= \sum_j W_{ji} \frac{(P_j - P_i)(F_j - F_i)}{|P_j - P_i|^2} . \end{aligned}$$

This is a particularly efficient choice when combined with a natural neighbor interpolant.

Chapter 6

Conclusions and future work

6.1 Future work

There are a few areas where the algorithms presented here could be further developed or extended.

6.1.1 Robust topology

We can contour a function f with any of the previous techniques. However, small changes in the input can lead to large changes in the resulting curve. When the zero plane passes through a saddle point, the intersection is two crossing lines (Figure 6.1(a)). The contouring algorithm from Chapter 3 avoids creating contours that cross, and will output either Figure 6.1(b) or (c). That decision is very sensitive to round-off error and small changes in the function.

One approach to solving this problem is to contour two functions

$$f_{\pm} = f \pm \epsilon \left| \nabla f \right|$$

instead of one. Given f and ∇f we can approximate $\nabla f_{\pm}$ by

$$\begin{aligned} \nabla f_{\pm}(x,y) &= \nabla f(x,y) \pm \epsilon \nabla \left| \nabla f(x,y) \right| \\ &= \nabla f \pm \left(\left| \nabla f \left(x + \frac{\epsilon}{2}, y \right) \right| - \left| \nabla f \left(x - \frac{\epsilon}{2}, y \right) \right| \right), \end{aligned}$$

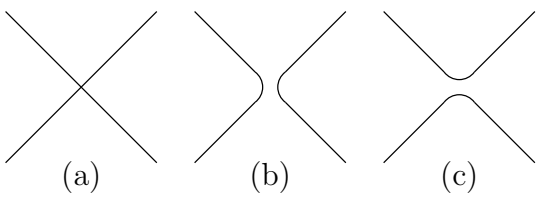


Figure 6.1: Small changes to a function at a saddle point can change the contour topology.

$$\left|\nabla f\left(x, y+\frac{\epsilon}{2}\right)\right| - \left|\nabla f\left(x, y-\frac{\epsilon}{2}\right)\right|\Big) \ .$$

The actual contours of f will lie between the contours of f_+ and f_-. The topology of f_+ will differ from f_- if there is an ambiguous situation with the contours of f. Note that there is a limit to how well we can resolve contours near roots where the gradient is 0, such as the saddle point example given above. In this situation, $f_\pm$ may be more than ϵ apart.

6.1.2 Minimizing C^2 discontinuity

In the implementation given above, the cross-boundary derivative is found by either linear interpolation from the vertices or sampling the function. It is possible, however, to compute a cross-boundary derivative that reproduces cubic polynomials exactly without additional samples of the function.

Mann [60] has a technique for minimizing the cross boundary C^2 discontinuity for Clough-Tocher interpolation. If the data is consistent with a cubic, the resulting interpolant is C^2 and reproduces that cubic. Minimizing the C^2 discontinuity of the interpolant also reduces the curvature discontinuity of the resulting contours at element boundaries (the current method is curvature continuous everywhere else). For applications that measure the curvature of the contour, this is an important feature. The error metrics would have to be updated for this new interpolant.

6.1.3 Contouring piecewise-C^1 functions

Distance functions, and signed distance functions, are an important example of functions we would like to contour. The derivative of a distance function has unit magnitude almost everywhere. The discontinuities of the derivative of a distance function in 1D are where the derivative jumps between $+1$ and -1. In 2D the distance function to a polygon will consist of pieces of cones and planes. The Gaussian curvature of a 2D distance function is very near zero except at derivative discontinuities.

One approach that would work with a large class of piecewise-C^1 functions is to assume we have some sort of bound on the second derivative. If we ever notice the second derivative is larger, we can then break the patch into two pieces. We can extrapolate from the two pieces to find the intersection: a C^1 contouring problem!

For the particular case of distance functions, we could instead use an interpolant that reproduces distance functions exactly. Tsai [103] has a method for approximating a distance function on a regular grid using C^0 data. With C^1 data at each vertex, we can expect to get more accurate contours.

For a particular distance function f and point x, $f(x)$ and $\nabla f(x)$ (where $|\nabla f(x)|$ is assumed to be 1) determine the point on the nearest contour to x, $z(x) = x - f(x)\nabla f(x)$. Consider an element in our mesh, in 2D this would be a triangle or square. We would like to reconstruct f from $f(x_i)$ and $\nabla f(x_i)$ for the vertices x_i of the element. In practice, we may not have enough information to reconstruct f, so we will construct an approximation $\tilde{f}$.

Unfortunately, it is difficult to make this construction local. Given a triangle ABC, the point closest to C may be closer to the midpoint of AB than points closest to A and B. This means that there is a trade off between locality and accuracy in any scheme hoping to reproduce distance functions.

One simple approach is to combine the distance functions to $z(x_i)$ for each of the three corners. To combine two unsigned distance functions f_1 and f_2, set $f(x) = \min\{f_1(x), f_2(x)\}$. To combine two signed distance functions f_1 and f_2, take:

$$f(x) = \begin{cases} f_1(x) & \text{if } |f_1(x)| < |f_2(x)|, \\ f_2(x) & \text{otherwise.} \end{cases}$$

To make this local, the contribution from corners could be blended away near the opposite side.

6.2 Contributions

The main contributions of this thesis are:

- a modular framework of contouring algorithms (Section 2.1, Section 2.5, and in particular, Algorithms 2.1 and 2.2),
- implementations of several contouring algorithms,
- zero gradient and ϵ-panel handling for Grandine-Klein contouring (Section 3.6.1),
- a modification to Sherbrooke-Patrikalakis for quadratic convergence that is relatively fast (Section 3.7.2),
- a three-dimensional spline contouring algorithm based on Morse theory (Section 4.4), and
- a new C^1 scattered-data interpolant using natural neighbor interpolation (Section 5.4).

In addition to the above, there have been several minor contributions:

- modifications to contouring algorithms for robustness (Section 2.6.2),
- a method of factoring out a root of a 1D spline (Section 2.6.3),
- a simple, fast convex-intersection algorithm (Section 2.6.4),
- how to combine Clough-Tocher's and Sibson's techniques for interpolation over binary triangle trees (Section 3.4.5),
- an extensive error analysis of my 1D and 2D interpolation (Sections 2.4 and 3.5),
- the *optimism* error estimate (Section 3.5),

- an adaption of Grandine-Klein contouring to cubic splines (Section 3.6.1),
- an adaption Sherbrooke-Patrikalakis simultaneous solver to cubic splines (Section 3.7),
- a method, given a cubic Bézier patch, of determining the ordinates for an adjacent patch representing the same cubic (Section 3.7.3), and
- an algorithm for integrating vector fields in sub-manifolds of $\mathbf{R}^3$ using spline collocation (Section 4.4.3).

6.3 Conclusion

There are several themes that recur throughout this dissertation. The most obvious is contouring with the combination of an adaptive mesh, a cubic interpolant, and a technique for finding the contours of a spline. Another is the pervasive use of splines which led to the re-use of techniques to solve many different problems. We have found these algorithms to be quite robust and general purpose. While the details of contouring a spline vary in different dimensions, the same key idea is used to resolve the topology. In both 2D and 3D, topological transitions occur at the boundary and at the critical points of the height function. It is these themes that tie this work together.

Bibliography

[1] P. Alfeld. A trivariate Clough-Tocher scheme for tetrahedral data. *CAGD*, 1:169–181, 1984.

[2] Peter Alfeld. Derivative generation from multivariate scattered data by functional minimization. *Computer Aided Geometric Design*, 2:281–296, 1985.

[3] Peter Alfeld. Scattered data interpolation in three or more variables. In Tom Lyche and Larry L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design*, pages 1–33. Academic Press, 1989.

[4] Eugene L. Allgower, Kurt Georg, and Rick Miranda. The method of resultants for computing real solutions of polynomial systems. *SIAM Journal on Numerical Analysis*, 29(3):831–844, 1992.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[6] Douglas N. Arnold, Arup Mukherjee, and Luc Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM Journal on Scientific Computing*, 22(2):431–448, 2001.

[7] Ehud Artzy, Gideon Frieder, and Gabor T. Herman. The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm. *Computer Graphics and Image Processing*, 15:1–24, 1981.

[8] C.L. Bajaj, C.M. Hoffmann, R.E. Lynch, and J.E.H. Hopcroft. Tracing surface intersections. *CAGD*, 5:285–307, 1988.

[9] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, pages 3–17, 1983.

[10] Eberhard Bänsch. An adaptive finite-element strategy for the three-dimensional time-dependent Navier-Stokes equations. *Journal of Computational and Applied Mathematics*, 36:3–28, 1991.

[11] Eberhard Bänsch. Local mesh refinement in 2 and 3 dimensions. *Impact of Computing in Science and Engineering*, 3:181–191, 1991.

[12] R. E. Barnhill and S. N. Kersey. A marching method for parametric surface/surface intersection. *CAGD*, 7:257–280, 1990.

[13] R. K. Beatson, W. A. Light, and S. Billings. Fast solution of the radial basis function interpolation equations: Domain decomposition methods. *SIAM Journal on Scientific Computing*, 22(5):1717–1740, 2000.

[14] R. K. Beatson and G. N. Newsam. Fast evaluation of radial basis functions: Moment-based methods. *SIAM Journal on Scientific Computing*, 19(5):1428–1449, 1998.

[15] Jürgen Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[16] P. Bézier. *Numerical Control, Mathematics and Applications*. Wiley, 1972.

[17] Jules Bloomenthal. Polygonization of implicit surfaces. *CAGD*, 5:341–355, 1988.

[18] Wolfgang Böhm, Gerald Farin, and Jürgen Kahmann. A survey of curve and surface methods in CAGD. *CAGD*, 1:1–60, 1984.

[19] Andrea Bottino, Wim Nuij, and Kees van Overveld. How to shrinkwrap through a critical point: an algorithm for the adaptive triangulation of iso-surfaces with arbitrary topology. In *Implicit Surfaces '96*, pages 53–72, Oct 1996.

[20] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In *SIGGRAPH 2001*, pages 67–76. ACM, Aug 2001.

[21] A. J. Chorin. Flame advection and propagation algorithms. *Journal of Computational Physics*, 35:1–11, 1980.

[22] A. J. Chorin. Curvature and solidification. *Journal of Computational Physics*, 58:472–490, 1985.

[23] Harvey E. Cline and William E. Lorensen. System and method for the display of surface structures contained within the interior region of a solid body. US Patent #4,710,876, Jun 1985.

[24] R. W. Clough and J. L. Tocher. Finite element stiffness matrices for the analysis of plate bending. In *1st Conference on Matrix Methods in Structural Mechanics*, pages 525–545, 1965.

[25] Carl R. Crawford. Minimization of directed points generated in three-dimensional dividing cubes method. US Patent #4,885,688, Nov 1987.

[26] M. Daniel and J. C. Daubisse. The numerical problem of using Bézier curves and surfaces in the power basis. *CAGD*, 6:121–128, 1989.

[27] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry. In *Algorithms and Applications*. Springer-Verlag, Heidelberg, second edition, 2000.

[28] Carl de Boor, Klaus Höllig, and Malcom Sabin. High accuracy geometric Hermite interpolation. *CAGD*, 4:269–278, 1987.

[29] P. de Casteljau. Outillage méthodes calcul. Technical report, André Citroën Automobiles S. A., 1959.

[30] James Demmel. On condition numbers and the distance to the nearest ill-posed problem. *Numerische Mathematik*, 51(3):251–289, July 1987.

[31] Manfredo Perdigao do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall College Div., 1976.

[32] J. Duchon. Splines minimizing rotation invariant semi-norms in Soblev spaces. In W. Schempp and K. Zeller, editors, *Constructive Theory of Functions of Several Variables. Lecture Notes in Mathematics*, volume 571, pages 85–100. Springer, 1977.

[33] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *Comm. ACM*, 29:669–679, 1986.

[34] Gershon Elber and Myung-Soo Kim. Geometric constraint solver using multivariate rational spline functions. In *Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 1–10. ACM Press, 2001.

[35] K. J. Falconer. A general purpose algorithm for contouring over scattered data points. Technical report, National Physical Laboratory, Teddington, Middlesex, England, 1971.

[36] Gerald Farin. Triangular Bernstein-Bézier patches. *CAGD*, 3:83–127, 1986.

[37] Gerald Farin. Surfaces over Dirichlet tessellations. *CAGD*, 5:281–292, 1990.

[38] R. T. Farouki and V. T. Rajan. On the numerical condition of polynomials in Bernstein form. *CAGD*, 4:191–216, 1987.

[39] R. T. Farouki and V. T. Rajan. Algorithms for polynomails in Bernstein form. *CAGD*, 5:1–26, 1988.

[40] A. R. Forrest. Interactive interpolation and approximation by Bézier polynomials. *Computer Journal*, 15:71–79, 1972.

[41] R. Franke. Smooth surface approximation by a local method of interpolation at scattered points. Technical Report NPS 53-78-002, Naval Postgraduate School, Monterey, CA, 1978.

[42] Richard H. Franke. Scattered data interpolation: test of some methods. *Mathematics of Computation*, 38(157):181–200, Jan 1982.

[43] Qingxiang Fu. The intersection of a bicubic Bézier patch and a plane. *CAGD*, 7:475–488, 1990.

[44] Allen Van Gelder and Jane Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 13(4):337–375, Oct 1994.

[45] Thomas Gerstner. Fast multiresolution extraction of multiple transparent isosurfaces. In *Data Visualization '00*, pages 35–44. Springer, 2001.

[46] Thomas Gerstner. Fast multiresolution extraction of multiple transparent isosurfaces. *Computers & Graphics*, 26(2):219–228, 2002.

[47] Thomas A. Grandine. Computing zeroes of spline functions. *CAGD*, 6:129–136, 1989.

[48] Thomas A. Grandine. Applications of contouring. *SIAM Review*, 4(2):297–316, 2000.

[49] Thomas A. Grandine and Frederick W. Klein IV. A new approach to the surface intersection problem. *CAGD*, 14:111–134, 1997.

[50] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[51] B. Grünbaum and G. Shephard. *Tilings and Patterns*. W. H. Freeman, 1987.

[52] R. L. Hardy. Multiquadric equations of topography and other irregular surfaces. *Journal of Geophysical Research*, 76:1905–1915, 1971.

[53] John C. Hart. Morse theory for implicit surface modeling. In Hans-Christian Hege and Konrad Polthier, editors, *Mathematical Visualization*, pages 257–268. Springer-Verlag, Oct 1998.

[54] John C. Hart. Using the CW-complex to represent the topological structure of implicit surfaces and solids. In *Implicit Surfaces '99*, pages 107–112. Eurographics/SIGGRAPH, Sep 1999.

[55] Josef Hoschek and Dieter Lasser. *Fundamentals of Computer Aided Geometric Design*. A K Peters, 1993. Originally published in 1989 by B. G Teubner, Stuttgart, under the title Grundlagen der geometrischen Datenverarbeitung. Translated by Larry L. Schumaker.

[56] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of Hermite data. *ACM SIGGRAPH*, pages 339–346, 2002.

[57] Igor Kossaczký. A recursive approach to local mesh refinement in two and three dimensions. *Journal of Computational and Applied Mathematics*, 55:275–288, 1994.

[58] F. F. Little. Convex combinations surfaces. In R. E. Barnhill and W. Böhm, editors, *Surfaces in Computer Aided Geometric Design*. North-Holland, 1983.

[59] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[60] Stephen Mann. Cubic precision Clough-Tocher interpolation. *CAGD*, 16:85–88, 1999.

[61] Dinesh Manocha and James Demmel. Algorithms for intersecting parametric and algebraic curves I: simple intersections. *ACM Transactions on Graphics*, 13(1):73–100, January 1994.

[62] S. Marlow and M.J.D. Powell. A Fortran subroutine for plotting the part of a conic that is inside a given triangle. Technical Report R-8336, Atomic Energy Research Establishment, Harwell, United Kingdom, 1976.

[63] Joseph M. Maubach. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM J. Sci. Comput.*, 16(1):210–227, 1995.

[64] Joseph M. Maubach. The efficient location of neighbors for locally refined n-simplicial grids. In *Proc. 5th Int. Meshing Roundtable*, pages 137–153. Sandia Nat. Lab., Oct 1996.

[65] J. Milnor. *Morse Theory*, volume 51 of *Annals of Mathematics Studies*. Princeton University Press, 1963.

[66] W. F. Mitchell. *Unified Multilevel Adaptive Finite Element Methods for Elliptic Problems*. PhD thesis, U.I. at Urbana CS Dept., 1988. Report No. UIUCDCS-R-88-1436.

[67] Gregor Müllenheim. Convergence of a surface/surface intersection algorithm. *CAGD*, 7:415–423, 1990.

[68] Gregory M. Neilson and Bernd Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. *IEEE Visualization*, pages 83–91, 1991.

[69] Stanley Osher and Ronald P. Fedkiw. *The Level Set Method and Dynamic Implicit Surfaces*. Springer Verlag, 2002.

[70] Bradley A. Payne and Arthur W. Tog. Surface mapping brain function on 3D models. *IEEE Computer Graphics and Applications*, 10(5):33–41, 1990.

[71] Ronaldo Marinho Persiano, João Luiz Dihl Comba, and Valéria Barbalho. An adaptive triangulation refinement scheme and construction. In *Proceedings of the VI Sibgrapi (Brazilian Symposium on Computer Graphics and Image Processing)*, Oct 1993.

[72] Jörg Peters. Smooth mesh interpolation with cubic patches. *Computer-Aided Design*, 22:109–120, 1990.

[73] Carl S. Petersen. Adaptive contouring of three-dimensional surfaces. *CAGD*, 1:61–74, 1984.

[74] Carl S. Petersen, Bruce R. Piper, and Andrew J. Worsey. Adaptive contouring of a trivariate interpolant. In Gerald E. Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 385–395. SIAM, 1987.

[75] M. J. D. Powell and M. A. Sabin. Piecewise quadratic approximations on triangles. *ACM Transactions on Mathematical Software*, 3:316–325, 1977.

[76] M. J. Pratt and A. D. Geisow. Surface/surface intersection problems. In J. A. Gregory, editor, *The Mathematics of Surfaces*, volume 1, pages 117–142. Claredon Press, 1986.

[77] Albrecht Preusser. Algorithm 671 — FARB-E-2D: Fill area with bicubics on rectangles — a contour plot program. *ACM Transactions on Mathematical Software*, 15(1):79–89, March 1989.

[78] Ewald Quak and Larry L. Schumaker. Cubic spline fitting using data dependent triangulations. *CAGD*, 7:293–301, 1990.

[79] R. L. Renka and A. K. Cline. A triangle based C^1 interpolation method. *Rocky Mountain Journal of Mathematics*, 14:334–351, 1984.

[80] María-Cecilia Rivara. Local modification of meshes for adaptive and/or multigrid finite-element methods. *Journal of Computational and Applied Mathematics*, 36:79–89, 1991.

[81] K.H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, fourth edition, 1999.

[82] Walter Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, third edition, 1976.

[83] Detlef Ruprecht and Heinrich Müller. A scheme for edge-based adaptive tetrahedron subdivision. In Hans-Christian Hege and Konrad Polthier, editors, *Mathematical Visualization*, pages 61–70. Springer-Verlag, Oct 1998.

[84] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[85] Vladimir V. Savchenko, Alexander A. Pasko, Oleg G. Okunev, and Tosiyasu L. Kunni. Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum*, 14(4):181–188, October 1995.

[86] Robert Schaback. Remarks on meshless local construction of surfaces. In Roberto Cipolla and Ralph Martin, editors, *IMA Conference on the Mathematics of Surfaces IX*, pages 34–58. Springer, 2000.

[87] Larry L. Schumaker and Wolfgang Volk. Efficient evaluation of multivariate polynomials. *CAGD*, 3(2):149–154, 1986.

[88] J. A. Sethian. *Level Set Methods*. Cambridge University Press, 1996.

[89] E. G. Sewell. *Automatic generation of triangulations for piecewise polynomial approximation*. PhD thesis, Purdue University, 1972.

[90] Evan C. Sherbrooke and Nicholas M. Patrikalakis. Computation of the solutions of nonlinear polynomial systems. *CAGD*, 10:379–405, 1993.

[91] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[92] Jerry Shurman. *Geometry of the Quintic*. Wiley-Interscience, January 1997.

[93] Robin Sibson. A brief description of natural neighbour interpolation. In Vic Barnett, editor, *Interpreting Multivariate Data*, chapter 2, pages 21–36. John Wiley & Sons, 1981.

[94] Robin Sibson and Fraeme D. Thomson. A seamed quadratic element for contouring. *The Computer Journal*, 24(4):378–382, 1981.

[95] Ron Sivan and Hanan Samet. Algorithms for constructing quadtree surface maps. In *Proceedings of the 5th international symposium on spatial data handling*, volume 1, pages 361–370, Aug 1992.

[96] Michael Spivak. *A Comprehensive Introduction to Differential Geometery*, volume 1. Publish or Perish, Inc., second edition, 1979.

[97] Barton Stander and John C. Hart. Guaranteeing the topology of an implicit surface polygonization. In *SIGGRAPH 97*, pages 279–286, Aug 1997.

[98] Sarah E. Stead. Estimation of gradients from scattered data. *Rocky Mountain Journal of Mathematics*, 14(1):265–280, 1984.

[99] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, second edition, 1993.

[100] John Strain. A fast Semi-Lagrangian contouring method for moving interfaces. *Journal of Computational Physics*, 170:373–394, Jun 2001.

[101] United States Geological Survey. Mt. Morgan Quadrangle, California, 7.5 minute series (topographic), 1982.

[102] D. Suter. Fast evaluation of radial basis/spline functions: Multipoles without multipoles. Technical Report MECSE1992-1, Department of Electrical and Computer Systems Engineering, Monash University, October 1992.

[103] Yen-hsi Richard Tsai. Rapid and accurate computation of the distance function using grids, 2000.

[104] Greg Turk, Huong Quynh Dinh, James O'Brien, and Gary Yngve. Implicit surfaces that interpolate. In *International Conference on Shape Modeling and Applications 2001*, pages 62–71, May 2001.

[105] Greg Turk and James O'Brien. Shape transformation using variational implicit functions. In *SIGGRAPH 99*, pages 335–342, August 1999.

[106] Brian von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *Proceedings SIGGRAPH 87*, pages 103–110. ACM SIGGRAPH, 1987.

[107] Gunther H. Weber, Oliver Kreylos, T.J. Ligocki, J.M. Shalf, Hans Hagen, Bernd Hamann, and Kenneth I. Joy. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In D.S. Ebert, J.M. Favre, and R. Peikert, editors, *Data Visualization 2001 (Proceedings of "VisSym '01")*, pages 25–34, Vienna, Austria, 2001. Springer-Verlag.

[108] Rudiger Westermann, Christopher Johnson, and Thomas Ertl. A level-set method for flow visualization. In *IEEE Visualization*, pages 147–154, 2000.

[109] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics (Annual Conference Series)*, 28:269–277, July 1994.

[110] A. J. Worsey and G. Farin. An n-dimensional Clough-Tocher interpolant. *Constructive Approximation*, 3:99–110, 1987.

[111] A. J. Worsey and G. Farin. Contouring a bivariate quadratic polynomial over a triangle. *CAGD*, 7:337–351, 1990.

[112] A. J. Worsey and B. Piper. A trivariate Powell-Sabin interpolant. *CAGD*, 5:177–186, 1988.

[113] G. Wyvill, C. McPheeters, and B. Wyvill. Data structures for soft objects. *The Visual Computer*, 2(4):227–234, Aug 1986.

[114] Y. Zhou, W. Chen, and Z. Tang. An elaborate ambiguity detection method for constructing isosurfaces within tetrahedral meshes. *Computers & Graphics*, 19(3):355–364, 1995.

[115] Yong Zhou, Baoquan Chen, and Arie Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In *Proceedings IEEE Visualization 1997*, pages 135–142. IEEE Computer Society, ACM Press, 1997.