Topologically Accurate Meshing Using Domain Subdivision Techniques

by

Benjamin T Galehouse

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Mathematics

Courant Institute of Mathematical Sciences

New York University

September 2009

Chee Yap

Acknowledgments

I would particularly like to thank my adviser, Chee Yap, both for introducing me to the world of computational geometry, and for having so much patience as I oriented myself to the task. I would also like to thank Michael Burr for much valuable advice and for both poking holes in my ideas and helping to fill them. I would like to thank all of my committee members, and especially David Bindle for many comments and suggestions.

I'd also like to thank Silvain Pion and INRIA Sophia-Antipolis for a wonderful research opportunity.

Abstract

The following fundamental problem is of theoretical interest and has applications in graphics, computer aided design, and the analysis of polynomial surfaces: Suppose we are given (1) a function $F : \mathbb{R}^n \to \mathbb{R}$, (2) an interval formulation of F and ∇F, (3) an axis aligned closed hypercube $B_0 \subset \mathbb{R}^n$, and (4) a distance $\epsilon > 0$. Assuming 0 is a regular value of F, and some additional conditions on F, find a piecewise linear approximation V of $\{F = 0\}$ in the sense that it lies within ϵ of and is isotopic to $B_0 \cap \{F = 0\}$.

It is often the topological condition which is difficult to ensure. We present a theorem which introduces a new test for topological accuracy. Making use of this, we develop a family of algorithms very similar in form to the Vegter-Plantinga algorithm. They are correct for all n and we implement a variation which is practical when $n \leq 4$. This is the first known numeric (as opposed to algebraic) algorithm which ensures the topological guarantee with $n > 3$. When $n \leq 3$ this algorithm produces a mesh with densities similar to those produced by the Vegter-Plantinga algorithm. For $n = 2$ we describe an advancing boxes algorithm which is based on subdivision followed by an advancing front style progression. It has several unique characteristics, including an ability to ensure good approximations of surface normals and no requirement for precise sign determination of F.

Table of Contents

List of Figures

List of Tables

Introduction

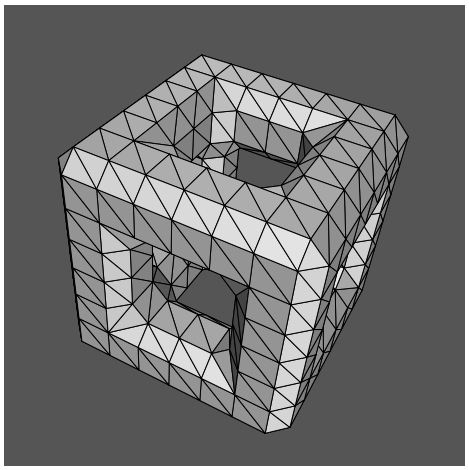


Figure 1: Mesh of a tangle cube $F(x) = 10 + \sum_{i=1}^{3} x_i^4 - 5x_i^2$

Consider the general problem of using an algorithm to find a piecewise linear approximation to an $n-1$ dimensional surface defined by $\{F = 0\} \cap B_0$ for some $F : \mathbb{R}^n \to \mathbb{R}$ and some finite axis aligned "box" $B_0 \subset \mathbb{R}^n$. To formally analyze this problem, we must choose a definition of approximation and assume some capability to partially evaluate F.

The two popular criteria to be a (good) approximation are ϵ-closeness and topological accuracy. ϵ-closeness is the requirement that our approximation lies within a distance ϵ of the actual surface. That is to say, the Hausdorff distance between the sets is less than ϵ. When $\{F = 0\}$ is bounded by B_0, topological accuracy is accepted to mean an isotopy between $\{F = 0\}$ and the approximation. When $\{F = 0\}$ is not bounded, the classical literature is often silent. One obvious strong condition is that there is an isotopy between $\{F = 0\} \cap B_0$ and our approximation within B_0. This thesis will focus on this strong condition, but weaker conditions have also been proposed [2].

One partial evaluation capability on F is the ability to determine the sign of F at some finite or countable collection of points. This capability is self-explanatory. However, for many classes of functions, determining that F is zero at a point is non-trivial. Also, the evaluation of F at a finite number of points fails to provide information about what happens elsewhere. So it is not

possible to formally verify many global invariants using this capability. These limitations lead us to consider another partial evaluation capability.

Interval formulations are a partial evaluation capability. We assume that we have interval formulations of F and ∇F. To see what this would mean in one dimension, let $I^{\mathbb{R}}$ be the set of closed finite intervals on $\mathbb{R}$. Then an interval formulation of G is a function $\square G : I^{\mathbb{R}} \to I^{\mathbb{R}}$ with the property that $G([a,b]) \subseteq \square G([a,b])$ where on the left we use G as an exact set function and on the right we have something easier to understand and compute. This idea has a natural generalization to higher dimensions. Because of the strong constraints provided by $\square F$ and $\square \nabla F$, this partial evaluation capability makes definite statements about the topology of $\{F = 0\}$ possible.

Chapter 1 contains history regarding non-algebraic meshing techniques, especially those techniques which directly influence this work. Each algorithm assumes one or both of the partial evaluation capabilities. Each algorithm produces an approximation of a level hyper-surface, though the definition of approximation varies somewhat.

In Chapter 2 we define formally the partial evaluation capabilities that we will require and develop some analytic and topological tools used in later chapters. Section 2.1 defines terms like mesh, k-box, and k-polytope. In Section 2.2 we state what properties our interval formulations are required to have. Section 2.3 defines some special flows. Using these flows ensures that certain arguments respect the bounded domain of interest B_0. In Section 2.4 we use these flows to prove Theorem 2.1, our main topological result. In Section 2.5 we give an analytic result. Theorem 2.3 is a finite dimensional mountain pass theorem which respects the flows of Section 2.3.

Chapter 3 discusses several algorithms developed using these concepts. In Section 3.1 we document a basic algorithm which separates regions where ∇F is zero from regions where F is zero. This becomes an initial step of our algorithms. In Section 3.2 we provide an algorithm outline based on subdivision of B_0. We show that if code based on this outline terminates, the resulting mesh is topologically accurate. In Section 3.3 we develop an algorithm based on this outline. The resulting algorithm solves the following restriction of the general problem:

Given a function $F : \mathbb{R}^n \to \mathbb{R}$ with non-singular level surface $\{F = 0\}$ and a compact axes aligned box $B_0 \subset \mathbb{R}^n$. Assume that ∇F is Lipschitz within B_0, that is $||\nabla F(p) - \nabla F(q)|| \leq k||p - q||$ for all $p, q \in B_0$ where $|| \cdot ||$ is the euclidian distance on $\mathbb{R}^n$ and k may depend on F, B_0. Assume that we are provided with $\square F$ and $\square \nabla F$ satisfying certain convergence properties.

Assume further that we can evaluate the sign of F at points B_0. Then construct $V \subset B_0$, a piecewise linear approximation of $\{F = 0\} \cap B_0$, such that there is an isotopy between $\{F = 0\} \cap B_0$ and V within B_0.

Vegter and Plantinga [16] solve this problem when $n = 2, 3$, $\{F = 0\} \subseteq B_0^\circ$ where B_0° is the interior of B_0. Our algorithm does not require these additional constraints and seems to be the first non-algebraic algorithm to solve problems of this form when $n > 3$.

Section 3.4 presents a two dimensional advancing boxes algorithm. It has two interesting characteristics. The first is that it does not require exact sign determinations hence it is a purely interval based approach. The second is that it can ensure that the normal vectors of our output always lie within some angular range α of ∇F. The angular range restriction can be locally defined. α need not be a constant across B_0.

In Chapter 4 we discuss our implementation. There is focus on both the issues that were encountered and the techniques and data structures which were selected to deal with them. As part of this, we document the interface to one of the primary data structures. Chapter 5 discusses the final performance of our implementation and includes images of the resulting meshes.

1
History of Non-Algebraic Meshing Techniques

We define a mesh within $\mathbb{R}^n$ to be a complex of $n-1$ dimensional simplexes which together bound some region, some collection of polytopes. These terms are described in more detail in Section 2.1. We are interested in methods which use guaranteed computation techniques [22] to ensure that the resulting mesh approximates a surface of interest.

When F is a polynomial over $\mathbb{Z}$, then there are many algebraic tools which can be applied to meshing the level surfaces of F [6,3]. Purely algebraic techniques can handle higher dimensional problems and even singular surfaces. These algorithms however do not scale well with polynomial degree or with the dimension of the domain. There also exist hybrid algorithms, e.g. interval arithmetic can be used to replace some portions of an algebraic algorithm [7].

1.1 The Marching Cubes Algorithm

An early approach to this type of problem is Lorensen and Cline's marching cubes algorithm [13]. Strictly speaking, the algorithm is not a guaranteed computation algorithm. It is described here because aspects of it are seen in the Vegter-Plantinga algorithm described in Section 1.3.

The algorithm is based on knowing the sign of F on a grid of points which is fixed a priori. Specifically, we evaluate the sign of each point and then fill in each cube, based on the signs of the corners of the cube. For each possible corner sign pattern, the algorithm defines a fixed mesh within the cube. This mesh divides the positive and non-positive corners. These constant meshes are constructed so that the intersection of the mesh with a face g depends only on the sign pattern of the corners of g.

For certain types of real problems this approach is invaluable. For example, if a medical scanning device presents a uniform grid of densities and we wish to visualize density iso-surfaces, then we have precisely the information which marching cubes uses. Given such a mesh of densities, and nothing else, this may well be the best solution. Furthermore, the algorithm is both very

fast and very easy to implement. In fact, the algorithm was considered so simple and obvious that the now expired patent [4] was sometimes touted as an example demonstrating problems with software patents. Avoiding this patent issue is one of the justifications sometimes given for the related marching tetrahedron approach [15].

On the other hand, when F is available in some computable form, this approach tends to be mathematically unsatisfying. The marching cubes algorithm only evaluates F at a finite number of points. Formally, this means that the behavior of F is entirely uncharacterized away from these points. Now, if we do happen to have some sort of bound on the smallest possible feature size for a particular F and the freedom to select a grid, then we can do so with this feature size in mind. However, we might be forced to apply a potentially overly-fine grid to the entire problem. The uniformity of the grid exacerbates the problem of resolution selection.

1.2 Interval Analysis

In order to algorithmically produce information about F at more than a finite number of points, we need to bring in a different evaluation model. For a wide range of functions F, it is possible to find interval formulations $\square F$ which provide the strong constraint $F(B) \subseteq \square F(B)$ over boxes B, with the bare $F(B)$ treated as the (potentially non-computable) exact set function. To start with, if we are given a number system which supports exact computation or controlled rounding, then it is possible to define interval versions of normal arithmetic operations [20]. Implementations of this include the Boost interval library [1] and MPFI [18]. The Boost interval library is particularly flexible because it can be used with a range of number types and uses controlled rounding when necessary.

There are various ways to extend this further: For example, suppose that we have the first k terms of a Taylor expansion for F about a point x and an appropriate error term. Then it is a straightforward exercise to use interval arithmetic primitives to find a formulation of $\square F$ which behaves well over intervals which contain x. However when x is not in the interval of interest, it could be necessary to keep adding terms to ensure that the output interval shrinks. This is related to the interval formulation convergence properties discussed in Section 2.2.

As another example, if a Taylor series is known and the sign changes of F' are well understood,

then it is possible to define optimal interval functions based on monotonicity. If say $-\frac{\pi}{2} \leq a \leq b \leq \frac{\pi}{2}$ then

$$\Box \sin[a, b] = [\sin^-(a), \sin^+(b)]$$

where $\sin^+$ and $\sin^-$ are evaluations of a Taylor series of sin with all rounding during the evaluations taken towards $+\infty$ and $-\infty$ respectively.

Whatever formulation is used, the convergence of $\Box F$ is important. The condition $F(B) \subseteq \Box F(B)$ can be satisfied trivially, but the interval algorithms mentioned here only terminate when $\Box F(B)$ shrinks as B shrinks.

Several models of interval convergence have been developed. One is Lipschitz interval convergence, which is the requirement that $\operatorname{diam} \Box F(B) < k_F \cdot \operatorname{diam} B$, with k_F a constant depending on both F and the initial domain B_0. diam is the standard set diameter function. Another model is quadratic convergence, which, is typically formulated as the requirement that $\operatorname{diam}(\Box F(B)) - \operatorname{diam}(F(B)) \leq k_F \operatorname{diam}(B)^2$. This quadratic convergence is available for polynomials and rational functions, and it is based on evaluation of a Taylor series around the center of B [14,17]. Again, k_F may depend on the initial box B_0. In Section 2.2 we introduce a convergence requirement which can be seen as a relaxation of either of these.

Snyder describes techniques which use interval computations to approach several classes of problems [20]. He applys them to the meshing of implicit curves. The basic idea is that a square is checked against certain interval conditions, and if it does not satisfy them, then it is split. Next, each child square is checked and the process is repeated until all squares satisfy the conditions. Once this is done, some analysis is performed within each square B to approximate the curve $\{F = 0\} \cap B$. This analysis involves finding the intersections of $\{F = 0\} \cap B$ with ∂B and then determining how they are connected within the square.

This approach requires doing a potentially large amount of computation for each square. In particular, if the curve makes tangential contact with a square, it is not clear how the algorithm differentiates this comtact from nearly-tangential contact. Also, the obvious higher-dimensional generalization involves recursively analyzing every k-face of every n-cube containing part of the mesh. This makes the potential issue of tangential contact even more troubling.

1.3 The Plantinga-Vegter Algorithm

Plantinga and Vegter [16] developed an algorithm for the meshing of implicit surfaces. It shares some characteristics with both Snyder's algorithm and the marching cubes algorithm. It supports both two and three dimensions. Initially, it is much like Snyder's algorithm in that space is subdivided into smaller and smaller boxes until every box satisfies some interval analysis based condition. Then, the boxes are tiled with a fixed set of curves or surfaces, much like marching cubes.

There are several limitations to this algorithm, such as: Since a precomputed table is used, it requires a balanced tree. That is, neighboring cubes can only differ in size by a single split. The correctness proof assumes that $\{F = 0\}$ lies entirely within the initial cube or square B_0. This last limitation can be removed. If the correctness conditions are relaxed somewhat from homotopy with $\{F = 0\} \cap B_0$, it is possible to alter the algorithm to give a correctness result when $\{F = 0\} \not\subseteq B_0$. The resulting algorithm requires subdivisions similar in size to normal Vegter-Plantinga [2]. Another limitation of the original Vegter-Plantinga is that it only applies to square and cubical subdivision, but this too can be removed [12].

A more major and fundamental limitation is that the correctness proof for Vegter-Plantinga is based on piecewise deformations and manual case analysis. So their proof approach does not scale well with dimension. Another is that the algorithm requires the ability to evaluate the sign of F at points which are dyadic relative to B_0. By dyadic relative to B_0, we mean those points findable through 2^n way splitting of B_0. So the dyadic points relative to the unit box $[0, 1]^n$ are points with within $[0, 1]^n$ with coordinates expressible as multi-precision floating-point numbers of the form $k \cdot 2^{-j}$ with $k, j \in \mathbb{N}$.

2
Theory

2.1 Notation

The variable n represents the dimension of our problem statement. It is generally assumed that $n \geq 2$. Unless otherwise stated, this is the only constraint on n. When $n = 1$ these approaches tend to collapse to a root finding scheme, and this aspect is not explored here in detail.

We follow and extend the notation used by Edelsbrunner [8] and others in which structures are characterized by their affine dimension. A k-flat is an affine subspace of dimension k. For any set $\boldsymbol{s} \subseteq \mathbb{R}^n$ the affine dimension of $\boldsymbol{s}$ is the smallest k such that $\boldsymbol{s}$ lies within some k-flat.

Generally, a k-face is the closed convex hull of a collection of points with affine dimension k. That is, a k-face will allways be a convex set within a k-flat. When a k-face is the convex hull of $k+1$ points we will call it a (non-sigular) k-simplex. The other type of k-face which will come up often is a k-box. A k-box is a cartesian product of intervals $\prod_{i=1}^{n}[a_i, b_i]$ with $a_i \leq b_i$ for all intervals and $a_i < b_i$ for exactly k intervals. A k-square is a k-box with the additional constraint that $b_i - a_i = s, 0$ where $s > 0$ is a constant side length. By the definisions already given, $b_i - a_i$ will be s for precisely k values of i. Note that unlike a k-simplex, a k-box is axis aligned by definition.

If an l-box $f_l = \prod_{i=1}^{n}[a_i, b_i]$ and a k-box, $f_k = \prod_{i=1}^{n}[c_i, d_i]$ have the properties that $l \leq k$ and $a_i, b_i \in \{c_i, d_i\}$ for all i then we say that f_l is an l-face **of** f_k. Note that an l-face of a k-square is an l-square.

For our purposes, a k-polytope V consists of a closed bounded subset $V' \subset \mathbb{R}^n$ within a k-flat P, along with a finite set of $(k-1)$-simplexes $\{f_i\}$ such that $\partial_P V' = \bigcup_i f_i$ and $\{f_i\}$ are elements of some simplicial complex. ∂_P denotes boundary relative to the k-flat P.

Occasionally, we might abuse notation somewhat and talk about V as a set. In these cases we of course mean the set V'. Whenever it is not made clear, it should be assumed that the faces $\{f_i\}$ are arbitrarily chosen and fixed for the duration of the discussion. An l-face of a k-polytope when $l \leq k$ is an l-simplex within the simplicial complex induced by $\{f_i\}$.

The unqualified term polytope should be taken to mean an n-polytope. For the most part,

we will only consider n-polytopes, k-boxes and k-simplexes.

2.2 Interval and Function Requirements

In addition to sometimes having geometric meaning, we also treat n-boxes as n dimensional intervals. We require that the interval functions $\square F$ and $\square \nabla F$ are defined for F and its gradient. Both $\square F$ and $\square \nabla F$ take n-boxes as arguments. Note, in particular, that we do not require that they accept k-boxes with $k < n$. The reasons for this are discussed in Section 2.2.1. $\square F$ produces an interval (1-box) as a result. $\square \nabla F$ produces an n-box as a result. We use subscripts to select an interval from an n-box. For example if $\square \nabla F(B) = [1, 2] \times [2, 3]$ then $(\square \nabla F(B))_1 = [1, 2]$ and $(\square \nabla F(B))_2 = [2, 3]$.

We require that $\lim_{\operatorname{diam}(B) \to 0} \operatorname{diam}(\square \nabla F(B)) = 0$, where $\operatorname{diam} B = \sup_{x,y \in B} |x - y|$ is the standard set diameter. However, to show termination on a finite volume, we actually need this property to be uniform. Therefore, we introduce uniform moduli of convergence $\omega_G : \mathbb{R}^+ \to \mathbb{R}^+$ with $\lim_{x \to 0} \omega_G(x) = 0$. We require that ω_F and $\omega_{\nabla F}$ exist such that $\operatorname{diam}(\square F(B)) < \omega_F(\operatorname{diam} B)$ and $\operatorname{diam}(\square \nabla F(B)) < \omega_{\nabla F}(\operatorname{diam}(B))$ when B is any axes aligned cube within B_0.

An even weaker convergence requirement would be that that when a sequence of boxes $B_0 \supset B_1 \supset B_2 \ldots$ converges to a point in the sense that $\bigcap_i B_i = x$ then the sequence $G(B_i)$ converges to $G(x)$. However, without some uniformity of convergence, termination arguments become difficult.

Because B_0 is compact, such ω exist under standard interval arithmetic systems. As mentioned in Section 1.2, the requirement for a modulus of convergence is a milder condition than several other standards for convergence e.g. if $\limsup_{x \to 0} |\omega_G(x)/x| = +\infty$, then our interval formulation does not have the Lipshitz interval convergence property.

2.2.1 On Computability

In addition to the interval requirements, we assume that the sign of F can be determined at the corners of our mesh. Corners of our mesh may appear at any point which is dyadic relative to B_0. When F is a polynomial, this is straightforward. We can perform exact computations with rational numbers or multi-precision floating-point numbers. However, if F is transcendental, and

if $F(x) = 0$ for a corner point x, it isn't clear how to certify this computationally. Certainly there is no general way to show this using a Taylor expansion. Notice that the Vegter-Plantinga and marching cubes algorithms require this sort of computability requirement. One of the reasons that the advancing boxes algorithm described in Section 3.4 is interesting is that it is entirely based on interval evaluations and never requires exact sign computation.

Related to this is the allowance that $\square F$ and $\square \nabla F$ need only accept k-boxes with $k > 0$. Specifically, they are not themselves required to accept points. So long as the input to these has a non-zero set diameter, this scale can be used as an indicator of how much resolution is required. If our computations are being done with Taylor series and variable precision floating point numbers, we can increase the precision, and the number of terms in the Taylor expansion as the input width decreases.

If we required Lipschitz convergence of our interval functions, more implementation care would be required. If we were using variable precision floating point arithmetic, it would be necessary to ensure that the precision selected for all intermediate and final values increased sufficiently quickly. Similarly, the quadratic convergence of the centered normal forms approach is only available for certain families of functions. Because we require the more general ω convergence guarantee, there is more flexibility in our computational model. In addition, this approach makes easier the argument near the end of Section 3.3.1 regarding termination near boundaries.

Also related to this is the choice of number systems with which we perform our computation. Fixed precision floating point numbers, for example, can only represent a finite subset of $\mathbb{R}$. This limits our ability to deal with narrow intervals. For sufficiently simple and well behaved functions, it might be that machine precision floating point numbers provide enough accuracy for our algorithms to work. However, in general, to fulfill our convergence requirements as written we need to switch to a number system which can represent a dense subset of $\mathbb{R}$. The two most obvious choices are $\mathbb{Q}$ and multi-precision floats $\mathbb{Z}[1/2]$. Strictly speaking, multi-precision floats form a ring, not a field. However, approximate division can be used when necessary and the needed convergence property can still be ensured. A downside of using dynamically sized number types is that they tend to require additional heap allocations and this makes them slow, even when the bit length is small. Section 4.2 mentions some experimental results relating to this.

2.3 Flows

If we are given a Lipshitz vector field $J : \mathbb{R}^n \to \mathbb{R}^n$ then by the theory of ordinary differential equations [5] there is a flow $\Gamma : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ associated with J such that $\frac{d}{dt}\Gamma(x,t) = J(\Gamma(x,t))$. Also, given any $x \in \mathbb{R}^n$ there is an individual integral curve $\gamma(t) = \Gamma(x,t)$.

In the following proofs we make use of flows generated by a vector field. The vector field of interest is a perturbation of ∇F. Roughly speaking, we modify ∇F to ensure that flows stay within our initial box B_0. Based on B_0 and for each small $\delta > 0$ we define a modified vector field $\nabla_{\delta,B_0} F$ within B_0 as follows: If x is further than δ from ∂B_0 then $\nabla_{\delta,B_0} F(x) = \nabla F(x)$. If x is in a k-face of B_0 which is perpendicular to a coordinate axis x_i then the i component of $\nabla_{\delta,B_0} F(x)$ is defined to be 0. In between we multiply by a smooth positive cutoff function to reduce $(\nabla_{\delta,B_0} F)_i$. On each k-face of B_0, $\nabla_{\delta,B_0} F$ has at most k non-zero components. For example, $\nabla_{\delta,B_0} F(x) = 0$ when x is a corner of B_0. Notice that if $x \in B_0^\circ$, the interior of B_0 relative to $\mathbb{R}^n$, then $\nabla_{\delta,B_0} F(x)$ is zero iff $\nabla F(x)$ is.

We only consider non-singular hyper-surfaces. In fact, we require that 0 be a regular value of F with respect to our modified ∇_{δ,B_0}. By this we mean that if $F(x) = 0$ then $\nabla_{\delta,B_0} F(x) \neq 0$. This condition is independent of δ and requires that $\{F = 0\}$ does not intersect ∂B_0 in a tangential manner or contain a corner of B_0. Should this modified regularity not hold when 0 is a regular value of F in the global sense, then a perturbation of B_0 will make it regular in our local sense.

After F and δ have been fixed, we can talk about the flow $\Gamma(x,t)$ defined by the field $\nabla_{\delta,B_0} F$. A property of this flow is that it cannot leave B_0, which is a key difference when compared to the flows generated by ∇F. Following our modified definition of regular value, a singular point of F in B_0 is a place where $\nabla_{\delta,B_0} F = 0$. To get a sense of the overall effect of this perturbation, consider Figure 2.1. On the left we have a radial flow away from the center of the square. Globally, the center of the square is the only singular point. On the right, we see that the majority of the flows bend as they approach the boundary. We have eight additional singular points created by our choice of B_0.

Another aspect of this modified definition of singular point is that they are still markers for local minimum and maximum with respect to B_0. On a k-face G of B_0, $\nabla_{\delta,B_0} F$ is related to the k dimensional ∇F found by considering F restricted to the k-dimensional affine subspace

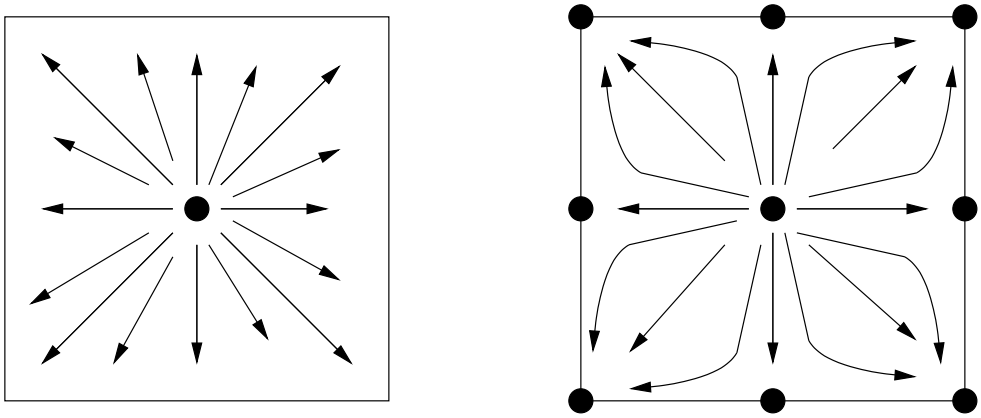


Figure 2.1: Flow modification used to support Theorem 2.1. On the left is an unmodified flow. On the right is the modified flow. Notice that flows which would have escaped B_0 are now stopped at new singular points.

containing G. The only difference is that the missing dimensions are present but zero. So if x is a local extremum of F on B_0 then either $\nabla F(x) = 0 \implies \nabla_{\delta,B_0} F(x) = 0$ or x lies in ∂B_0. If x lies within the interior of a k-face G of B_0, then it must be an extremum of F restricted to G so by the above $\nabla_{\delta,B_0} F(x) = 0$.

Next, we consider the behavior of F along the intgral curves defined by $\nabla_{\delta,B_0} F$ and described by Γ. $F(\Gamma(x,t))$ is a strictly increasing function of t, as long as $\Gamma(x,t)$ is not a singular point. By compactness of B_0 we see that $\Gamma(x,t)$ approaches a singular closed set of F in our modified sense when $t \to \infty$. Using the flows generated by $-\nabla_{\delta,B_0}$, we can extend $\Gamma(x,t)$ in the obvious mmanner include negative values of t.

Because $\nabla_{\delta,B_0} F$ is still Lipschitz, some classical results from the theory of ordinary differential equations apply [5]:

1. The flows do not cross: if $\Gamma(x_1,t_1) = \Gamma(x_2,t_2)$ then $\Gamma(x_1,t_1-t_2) = x_2$ and $\Gamma(x_2,t_2-t_1) = x_1$
2. Γ is a continuous function on $\mathbb{R}^{n+1}$.

2.4 Topology

Our main topological result is Theorem 2.1 which is presented in Section 2.4.1. It assumes that our domain of interest B_0 is an n-box. This situation is the one that we have explored

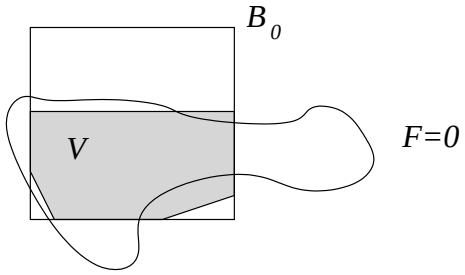


Figure 2.2: Schematic of typical inputs to Theorem 2.1

experimentally. As discussed in Section 2.4.2, we can use the same proof techniques to ensure topological accuracy of an approximation to $\{F = 0\} \cap D$ when D is a much more general domain. Then in Section 2.4.3 we take this further, and describe a way to find a topologically accurate intersection of two implicitly-defined surfaces.

2.4.1 n-box Domain

At this point, we begin using V to refer to a closed polytope which lies within B_0, our initial n-box. As per our definition in Section 2.1 we are concerned with both a volume of n-dimensional space which makes up the polytope, and with a set of $(n-1)$-simplexes which make up the boundary of the polytope.

Furthermore, we are uninterested in those $(n-1)$-simplexes which lie within ∂B_0. Therefore, within the following theorem, we consider V to have the relative topology induced on B_0 by $\mathbb{R}^n$. Put another way, $V^C = B_0 \backslash V$ and $\partial V = V \cap \overline{V^C}$ so that ∂V can not contain an open subset of a face of B_0. An important implication of this is that the none of $(n-1)$-simplexes associated with V lie entirely within ∂B_0.

Given B_0 and this reduced set of simplexes, a polytope is nearly determined. It is merely necessary to indicate the inside. The collection of $(n-1)$-simplexes is the mesh associated with the polytope, and this mesh will be the output of our algorithms.

The following theorem is our main topological result. It makes few constraints on the construction of V and perhaps has uses beyond the analysis of our subdivision algorithms. Figure 2.2 gives a schematic of how typical inputs to this theorem might look.

Theorem 2.1. *Suppose that ∇F is Lipschitz and that 0 is a regular value of F in our modified sense on the finite n-box B_0. Further suppose that we have a closed polytope $V \subseteq B_0$ and $\exists \delta > 0$ with the properties that:*

1. *For each closed face G of V (again, defined relative to B_0) with exterior pointing normal vector $\hat{n}_G$, we have that $(\hat{n}_G \cdot \nabla_{\delta,B_0} F) > 0$ on G.*
2. *If $\nabla_{\delta,B_0} F(x) = 0$ and $F(x) < 0$ then $x \in V^\circ$, the interior of V.*
3. *If $\nabla_{\delta,B_0} F(x) = 0$ and $F(x) > 0$ then $x \in V^C$, the complement of V relative to B_0.*

Then, ∂V is isotopic within B_0 to the surface $\{F = 0\} \cap B_0$.

Note: Because we consider closed faces, when x is at an edge or vertex of V, there may be multiple faces G containing x. Condition (1) then places multiple constraints on ∇F at x.

The basic idea of this topological theorem is inspired by Morse theory where a lack of critical points is used to create a deformation retract. Condition (1) ensures that our approximated surface is is similar to $\{F = 0\}$ in that its normal is similar to that of a level surface. Conditions (2) and (3) ensure a lack of critical points in the closed symmetric difference between the sets $\{F \leq 0\}$ and V. Intuitively, we expect no barrier to deformation, and the proof formalizes this idea.

Proof. Take any $x \in \partial V$ with $F(x) < 0$. By compactness, as t gets large $\Gamma(x,t)$ approaches a closed set where $\nabla_{\delta,B_0} F = 0$. By conditions (1) $\Gamma(x,t)$ stays within V^C. So by condition (2) we have that $\lim_{t\to\infty} F(\Gamma(x,t)) > 0$. By the intermediate value theorem, there must be a positive time $T(x)$ for which $F(\Gamma(x,T(x))) = 0$. Similarly, when $F(x) > 0$ we can find a negative time $T(x)$ such that $F(\Gamma(x,T(x))) = 0$. Of course we define $T(x) = 0$ when $F(x) = 0$. Note that because $(\nabla_{\delta,B_0} F \cdot \nabla F) > 0$ away from singular points, $T(x)$ is uniquely defined.

Next, we show that $T(x)$ is continuous. First, suppose that $x \in \partial V$ has $F(x) = 0$. By regularity, there is a ball $\boldsymbol{B}(x,\rho)$ and a c such that $(\nabla_{\delta,B_0} F \cdot \nabla F) > c > 0$ in $\boldsymbol{B}(x,\rho)$. Let $M = \sup_{y\in\boldsymbol{B}(x,\rho)} |\nabla_{\delta,B_0} F(y)|$. Then choose $\epsilon > 0$ and take any $y \in \partial V \cap \boldsymbol{B}(x,\rho/2)$ which is sufficiently near to x so that $|F(y)| < c\epsilon$ and $|F(y)| < \frac{c\rho}{2M}$.

So long as the curve $\Gamma(y,\cdot)$ lies within $\boldsymbol{B}(x,\rho)$ the speed of the curve's motion is bounded by M. So in particular, the curve segment $\Gamma(y,[-\frac{\rho}{2M},\frac{\rho}{2M}])$ lies within $\boldsymbol{B}(x,\rho)$. By the definition of

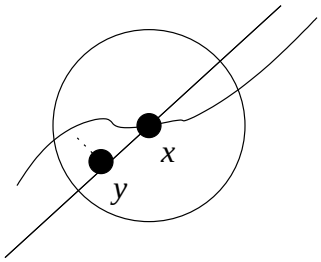


Figure 2.3: The basic continuity argument in Theorem 2.1 is that when y is sufficiently close to x then it reaches $\{F = 0\}$ while staying within a well behaved sphere.

c the derivative of $F(\Gamma(y, \cdot))$ is larger than c in this range, and therefore $F(\Gamma(y, \cdot))$ must change by at least $\frac{c\rho}{2M}$ before leaving $\boldsymbol{B}(x, \rho)$. This gives

$$F\left(\Gamma\left(y, +\frac{\rho}{2M}\right)\right) > F(y) + \frac{c\rho}{2M} > 0$$

and

$$F\left(\Gamma\left(y, -\frac{\rho}{2M}\right)\right) < F(y) - \frac{c\rho}{2M} < 0.$$

Therefore the curve segment between y and $\Gamma(y, T(y))$ lies within $B(x, \rho)$. But then the other constraint $|F(y)| < c\epsilon$ ensures that $|T(y)| < \epsilon$. Therefore, T is continuous at x when $F(x) = 0$. See Figure 2.3 for a schematic of this arrangement.

Now consider when $F(x) \neq 0$. We can define $\partial V_{T(x)}$ to be the image of ∂V under the transformation $\Gamma(\cdot, T(x))$. By continuity of Γ we have that $\Gamma(y, T(x))$ approaches $\Gamma(x, T(x))$ as y approaches x. Then the prior argument with $\partial V_{T(x)}$ in place of ∂V shows that $T(y) - T(x)$ is small when y is near to x.

This gives us that T is continuous for all $x \in \partial V$ and so $\Gamma'(x, s) = \Gamma(x, sT(x))$ defines an isotopy within B_0 between $F(x) = 0$ and ∂V. □

An alternative intuitive proof is that we have a sequence of bijections. There is a bijection between the points on ∂V and the flows defined by Γ which pass throguh them. There is a bijection between these same flows and the points of $\{F = 0\}$.

2.4.2 Other Domains

The basic proof concept used in Theorem 2.1 can be extended to handle other shapes of domain. While our experimental work has focused on the case already proved, we include here a much more general formulation.

Let D be a bounded region which is equal to the closure of its interior, and assume further that ∂D is contained in the union of a finite number of smoothly embedded $n-1$ dimensional manifolds $\{\partial D_1, \ldots, \partial D_k\}$. Let δ be small enough such that for all i, the following property holds: if the distance from a point $x \in D$ to ∂D_i is less than δ, then there is a unique point $p_i(x) \in \partial D_i$ closest to x.

Lemma 2.1. *Assume that D is as above and given F is defined on a neighborhood of D with ∇F Lipschitz, then there exists a modified vector field $\nabla_{\delta,D} F$ defined on D such that:*

1. *$\nabla_{\delta,D} F$ is Lipschitz*
2. *$\nabla_{\delta,D} F(x) = \nabla F(x)$ when the distance from x to ∂D is greater than δ*
3. *$(\nabla_{\delta,D} F(x) \cdot \nabla F(x)) \geq 0$ with equality only when $\nabla_{\delta,D} F(x) = 0$.*
4. *$\nabla_{\delta,D} F(x) = 0$ only when either $\nabla F(x) = 0$ or $x \in D_j$ for j in some finite collection J and $\nabla F(x)$ is in the span generated by the normal vectors to D_j at x.*
5. *The flows generated by $\nabla_{\delta,D} F$ never leave D.*

Proof. Let $P^{\perp}_{i,x}$ and $P^{\|}_{i,x}$ be the projections which decompose vectors at x into components perpendicular and parallel to ∂D_i at $p_i(x)$. Let $\phi_\delta(r)$ be a smooth cutoff function such that $\phi_\delta(r) = 1$ for $r \geq \delta$ and $\phi_\delta(0) = 0$. Then $P_i(x, v) = P^{\|}_{i,x}(v) + \phi_\delta(d(x, p(x)))P^{\perp}_{i,x}(v)$ is a partial function on vector fields within D defined when $d(x, p_i(x))) \leq \delta$. We can define $P_i(x, v) = v$ when $d(x, \partial D_i) > \delta$ to extend this to all of D.

Note that properties 1 through 4 are preserved by the application of P_i to a vector field. Also note that the integral curves of a vector field resulting from P_i cannot leave D by crossing through D_i. This property is preserved by other functions P_j. So we take $\nabla_{\delta,D} F = P_k P_{k-1} \ldots P_1(\nabla F)$. □

Using $\nabla_{\delta,D}$ we can update our definition of regular value to mean that $\{F = 0\} \cap D$ avoids points where $\nabla_{\delta,D} F = 0$. Then we can update Theorem 2.1 to such a region D as follows. Note that the proof does not change significantly:

Theorem 2.2. *Let D be a region satisfying the conditions required for Lemma 2.1. And assume that 0 is a regular value of F in our modified sense on D. Suppose that we have a closed polytope V and a $\delta > 0$ with the properties that:*

1. *For each closed face G of V such that $G \cap D$ is non-empty, with exterior pointing normal vector $\hat{n}_G$, we have that $(\hat{n}_G \cdot \nabla_{\delta,D} F) > 0$ on $G \cap D$.*
2. *If $\nabla_{\delta,D} F(x) = 0$ and $F(x) < 0$ then $x \in V^\circ$.*
3. *If $\nabla_{\delta,D} F(x) = 0$ and $F(x) > 0$ then $x \in V^C$.*

Then, ∂V is isotopic within D to the surface $\{F = 0\} \cap D$.

2.4.3 Intersections

There are several ways to view the effect the modifications made to create ∇_{δ,B_0} or $\nabla_{\delta,D}$. One is that we ensure that the intersections of $\{F = 0\}$ with the faces of B_0 or the boundary components D_i are each correctly approximated by the intersections of ∂V with the corresponding boundaries.

The stated technique can also be used to ensure that $\{F = 0\}$ has a correct intersection with other lower dimensional manifolds. For example, suppose that we are working in $\mathbb{R}^4 = (x, y, z, t)$ with the intention of making a movie. At certain time slices $t = t_0 < t_1 < \cdots < t_k$, we wish to ensure that topologically accurate meshes of $F_{t_i}(x, y, z) = F(x, y, z, t_i)$ are embedded in our global result. In this case, we could further modify $\nabla_{\delta,B_0} F$ to remove the t component of $\nabla F(x)$ when x is near any of the planes $t \in \{t_0, t_1, \ldots, t_k\}$. Then by an argument similar to Theorem 2.1, or by a simple application of it to each of the boxes of the form $B_0 \cap \{t_{i-1} \leq t \leq t_i\}$, we have our result. There is an isotopy of $B_0 \cap \partial V \cap \{t_{i-1} \leq t \leq t_i\}$ to $B_0 \cap \{F = 0\} \cap \{t_{i-1} \leq t \leq t_i\}$ and the restriction of them gives an isotopy of $\partial V \cap \{t = t_i\}$ to $\{F = 0\} \cap \{t = t_i\}$.

Another interesting situation arises if instead of F we are given functions F_1 and F_2 satisfying the computational requirements. Then a natural question to ask is whether we can find correct meshes of $\{F_i = 0\}$ in such a way that the intersection of the meshes is correct. We assume

that the contact is not tangential. Furthermore, assume that we have a meshing algorithm which ensures that the normal vectors of the mesh within a box B lie within the angular range of $\square\nabla F(B)$. Note that this condition is not met by Vegter-Plantinga or the normal checking algorithm of Section 3.3. However, the 2-dimensional advancing boxes algorithm of Section 3.4 does have the necessary property.

Suppose that B_0 is subdivided into a set of boxes $\mathbf{B}$ that are sufficiently fine so that one of the following two conditions applies to each $B \in \mathbf{B}$:

1. There is an i such that $0 \notin \square F_i(B)$
2. The $\square\nabla F_i$ are angularly disjoint.

Note that the second condition implies that $0 \notin \square\nabla F_i(B)$ for both i.

Now suppose we find topologically correct meshes V_i of the sets $\{F_i = 0\}$ subordinate to the boxes $\mathbf{B}$. Further suppose that the the normal vectors of the mesh within each box $B \in \mathbf{B}$ are within the angular range $\square\nabla F(C)$ as described above. In this situation, the following argument holds:

First of all, we deform ∂V_2 slightly to make it into a smooth manifold with borders, with borders only in ∂B_0. We call this manifold $\partial V_2'$. Then deform ∂V_1 to $\{F_1 = 0\}$ in such a way that its intersection with $\partial V_2'$ is deformed bijectively. Because Condition 2 holds in the region of interest, and because of the constraint on the normal vectors of V_2, using $\partial V_2'$ in this way creates no new singular points. Finally, we can deform $\partial V_2'$ to $\{F_2 = 0\}$ in such a way that its intersection with $\{F_1 = 0\}$ is deformed bijectively.

2.5 Modified Mountain Pass

In the previous section, we used the modified flows of Section 2.3 to make a topological claim. The modification was only necessary to account for the effects of being constrained to a box B_0. We now shows that the same flows give us an analytic result related to working within the restricted range B_0. We use this result in Section 3.4.4 to separate the components of $\{F = 0\}$.

Mountain pass theorems are important and well known in analysis [11], and say something about the existence of singular points. In Section 2.3 we introduced a modified definition of

singular points, based on our modified flows. The following is a finite dimensional mountain pass theorem which is correct using this modified definition:

Theorem 2.3. *Assume that F is defined on B_0 with ∇F continuous. Suppose further that we have $F(x) = F(y) = 0$, a regular value of F in our modified sense, and that x and y are in different components of $\{F = 0\}$. Let $\mathbf{\Gamma} = \{\gamma \in C([0,1]; B_0) | \gamma(0) = x, \gamma(1) = y\}$ be the family of continuous curves from x to y. Then*

$$c = \inf_{\gamma \in \mathbf{\Gamma}} \max_{t \in [0,1]} |F(\gamma(t))|$$

is not a regular value of $|F|$, in our modified sense.

This is useful to us because it means that when x and y are in different components of $\{F = 0\}$, then any path between them must pass through a point where $|F|$ is large. By large we mean larger than something which we can compute using interval arithmetic. Section 3.4.4 discusses this further. We prove Theorem 2.3 using the basic form of a proof presented by Jabri [11] and attributed to Courant.

Proof. Find a sequence of paths $\gamma_i \in \mathbf{\Gamma}$ such that

$$\lim_i \max_{t \in [0,1]} |F(\gamma_i(t))| = c$$

and consider the set of accumulation points of γ_i:

$$\boldsymbol{\gamma} = \bigcap_m \overline{\bigcup_{i \geq m} \gamma_i}.$$

Now, $\boldsymbol{\gamma}$ is an intersection of compact connected sets, and therefore is a compact connected set. Furthermore, if m_i is a point on γ_i with $F(m_i) = \max_{t \in [0,1]} |F(\gamma_i(t))|$ then the set $\{m_i\}$ has an accumulation point m in $\boldsymbol{\gamma}$ and $|F(m)| = c$. So c is attained by $|F(\boldsymbol{\gamma})|$ and furthermore we have that $\max_{z \in \boldsymbol{\gamma}} = c$.

This tells us that $c > 0$ because otherwise x and y would be connected by $\{F = 0\}$, contradicting the assumption that they lie in different components. Also $\boldsymbol{z} = \{z \in \gamma : |F(z)| = c\}$, the

set where this minimum is attained, is compact and not empty. To prove the theorem, we will show that $\boldsymbol{z}$ contains a point where F is singular in our modified sense.

We will use contradiction, so suppose that $(\nabla_{\delta,B_0} F \cdot \nabla F) > \alpha > 0$ on $\boldsymbol{z}$. Define

$$\boldsymbol{z}_\epsilon = \{z \in B_0 : \exists w \in \boldsymbol{z} \text{ with } |w - z| < \epsilon\}$$

as an ϵ-neighborhood of $\boldsymbol{z}$ and choose ϵ small enough so that $(\nabla_{\delta,B_0} F \cdot \nabla F) > \alpha/2$ on $\boldsymbol{z}_\epsilon$ and also so that $x, y \notin \boldsymbol{z}_\epsilon$ and $0 \notin F(\boldsymbol{z}_\epsilon)$.

Let ρ be a smooth cutoff functions with $\rho(\boldsymbol{z}) = \{1\}$ and $\operatorname{supp}\rho \subseteq \boldsymbol{z}_\epsilon$. Define $\eta : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ to be the flow generated by the vector field $-\operatorname{sgn}(F) \cdot \rho \cdot \nabla_{\delta,B_0} F$. Then $|F(\eta(z,\cdot))|$ is non-increasing, and furthermore has a derivative less than $-\alpha$ when both $z \in \boldsymbol{z}$ and $t \in (0, \epsilon/M)$ where we let $M = \sup_{z \in \boldsymbol{z}_\epsilon} |\nabla_{\delta,B_0} F|$.

Now $\eta(\boldsymbol{\gamma}, \epsilon/M)$ is a deformation of $\boldsymbol{\gamma}$ within B_0 and is therefore connected and compact. η does not affect x or y, so there is a $\gamma \in \Gamma$ which lies within $\eta(\boldsymbol{\gamma}, \epsilon/M)$. But $|F(\eta(\boldsymbol{z}, \epsilon/M))| < c - \frac{\alpha\epsilon}{M}$ and $|F(\eta(z,\cdot))|$ is non-increasing everywhere. So

$$\sup \left| F\left(\eta\left(\boldsymbol{\gamma}, \frac{\epsilon}{M}\right)\right)\right| < c$$

contradicting the minimization property which defined $\boldsymbol{\gamma}$ and $\boldsymbol{z}$.

□

3
Algorithms

Given Theorem 2.1, two natural approaches to our problem come to mind. The first is to perform spatial subdivision followed by some simple meshing scheme until interval arithmetic directly confirms that the conditions of Theorem 2.1 are met. The second is to construct a mesh in such a way that the conditions of Theorem 2.1 are met by construction. This chapter explores both of these ideas.

Section 3.1 discusses a singularity covering algorithm which is used as an initial step for our later algorithms. Section 3.2 lays out a normal checking algorithm, or more correctly, a template for an algorithm. To complete the algorithm a particular "corner cutting" function H must be chosen, and it must be proved that the algorithm terminates with the chosen H. Correctness however, comes from following the template. Section 3.3 defines a pair of functions H_V and $H_{V'}$ and proves that the template does terminate and therefore yields an algorithm when H is either of these functions. Finally, Section 3.4 presents a two dimensional algorithm which constructs a mesh through subdivision followed by direct reference to $\square\nabla F$.

In the sequel, we consider a candidate closed polytope $V \subseteq B_0$, which may or may not satisfy the three conditions of Theorem 2.1. As described previously, such a polytope can be defined by a mesh of $(n-1)$-simplexes which make up ∂V.

At every stage of our algorithm we will have a subdivision of B_0 into a collection $\mathbf{B}$ of boxes with disjoint interiors. Suppose that our mesh describing V is subordinate to such a collection $\mathbf{B}$, in the sense that each face of V lies within some box $B \in \mathbf{B}$. Then, with a little special handling at the borders, there are interval arithmetic conditions based on $\square F(B)$ and $\square\nabla F(B)$ which when true for all $B \in \mathbf{B}$ indicate that the three conditions of Theorem 2.1 are met by V for some $\delta > 0$.

3.1 Singularity Covering

We start by arranging for conditions (2) and (3) of Theorem 2.1 to be met. Because $\nabla_{\delta,B_0} F$ is different than ∇F we do not necessarily have the interval formulation inclusion $\nabla_{\delta,B_0} F \subseteq$

Table 3.1: Singularity Covering Algorithm

```
1:  initialize U to be a set of boxes, initially {B_0}.
2:  initialize S_{V°}, S_{∂V}, S_{V^C} to be sets of boxes, initially empty.
3:  while there is a box B ∈ U do
4:    remove B from U
5:    if □F(B) > 0 then
6:      place B in S_{V^C}
7:    else
8:      if □F(B) < 0 then
9:        place B in S_{V°}
10:     else
11:       if 0 ∉ □_B ∇F(B) then
12:         place B in S_{∂V}
13:       else
14:         split B and place the resulting child boxes into U
```

$\Box\nabla F(B)$.

We define a modified interval operation $\Box_B \nabla F(B)$ in such a way that $\nabla_{\delta,B_0} F \subseteq \Box_B \nabla F(B)$ when δ is sufficiently small. The definition of $(\Box_B \nabla F)_i$ depends on whether B intersects a k-face of B_0 which is perpendicular to the x_i axis. If not, then $(\Box_B \nabla F)_i = (\Box \nabla F)_i$. If so, then $(\Box_B \nabla F)_i$ is equal to the convex hull of $(\Box \nabla F)_i \cup \{0\}$. The result of this is that $\nabla_{\delta,B_0} F \subseteq \Box_B \nabla F$ when δ is sufficiently small. Specifically, either δ must be smaller than the distance from B to ∂B_0 or δ must be smaller than the side length of B.

With this definition in place, we can subdivide using the Singularity Covering Algorithm of Table 3.1. This algorithm simply subdivides each box until one of the following three conditions is met, and places the resulting boxes accordingly:

1. $\Box F(B) > 0$ If the test on line 5 is true we place B into S_{V^C} indicating that later algorithms should place it in the compliment of V.

2. $\Box F(B) < 0$ If the test on line 8 is true we place B into S_{V° indicating that later algorithms should place it in the interior of V.

3. $0 \notin \Box_B \nabla F(B)$ If the test on line 11 is true we place B into $S_{\partial V}$ indicating to later algorithms that it is safe for ∂V to intersect B.

The existence of ω_F, $\omega_{\nabla F}$ and the regularity of $\{F = 0\}$ in our modified sense together imply

that there is diameter s such that if $B \subset B_0$ is a box with $\operatorname{diam} B < s$ then $0 \in \Box F(B)$ implies that $0 \notin \Box_B \nabla F(B)$. This ensures termination.

Suppose that after running this algorithm we find a V with the properties that $\bigcup S_{V^\circ} \subseteq V^\circ$ and $\bigcup S_{V^C} \subseteq V^C$. Then $\partial V \subseteq \bigcup S_{\partial V}$ and we notice the following:

If a box B satisfied condition 1 above then any singularities in B are positioned to satisfy property 3 of Theorem 2.1. Similarly, If B satisfied condition 2 then any singularities in B are positioned to satisfy property 2 of Theorem 2.1. No box with a singularity will satisfy condition 3. So this singularity covering algorithm ensures that V satisfies properties 2 and 3 of Theorem 2.1.

3.2 Normal Checking

At each stage of our subdivision algorithm we will have a division of B_0 into boxes $\mathbf{B}$. In this section, we will assume that we have a function H which given $B \in \mathbf{B}$ will find a section of mesh within B.

To understand the input which will be passed to H, we need some terminology regarding the collection of 1-faces induced by $\mathbf{B}$. When a 1-face is minimal, in the sense that it is not the union of smaller 1-faces induced by $\mathbf{B}$, then we call it a link. When one end of a link is positive and the other is non-positive, we call it a crossing.

Once we divide B_0 into boxes $\mathbf{B}$, we will use H to construct a candidate V. Specifically, $H(B)$ will give the closed polytope that will then become $B \cap V$. We will consider different versions of H. All of them will be based on the sign of F at the corners of the subdivision which lie on ∂B. All of them will give B itself if the corners on ∂B are all non-positive. All of them will give $\varnothing$, the empty set, if all of the corners are positive.

To start with, we will consider H equal to the trivial H_I:

$$H_I(B) = \begin{cases} B & \text{any corner in } \partial B \text{ is non-positive} \\ \varnothing & \text{all corners in } \partial B \text{ are positive} \end{cases}$$

We will make reference to the exterior faces of $H(B)$. More than a direct understanding of the volume chosen by H, we always need a computational method to identify the faces of $H(B)$

Table 3.2: Normal Checking Algorithm

```
1: begin with S_{V°}, S_{∂V}, and S_{V^C} as output by the singularity covering algorithm of Table 3.1
2: while there is a box B in S_{∂V} do
3:    remove B from S_{∂V}.
4:    if H(B) is undefined  then
5:       mark B to be split
6:    else
7:       for all exterior face G of H(B) do
8:          if (□_G ∇F(B) · n̂_G) contains a non-positive value. (normal check) then
9:             mark B to be split
10:   if B is marked to be split then
11:      split B and add its children to S_{V°}, S_{∂V}, or S_{V^C} as the singularity covering algorithm
         of Table 3.1 in would.
12:      for all B′ a neighbor box of B which is bigger than the children of B do
13:         if B′ ∉ S_{V°} and B′ ∉ S_{V^C} and B′ ∉ S_{∂V}  then
14:            add B′ to S_{∂V}
15: for all box B with some corners positive and some non-positive do
16:   emit the exterior faces of H(B)
```

which will become the faces of V. We refer to these as the exterior faces of $H(B)$, and being able to find them is a key requirement on the definition of H. In the case of H_I we need to look at the neighboring boxes across each of the $2n$ faces of B and call each face an exterior face if one of the boxes has all positive corners. For all H that we consider, there are no exterior faces when all the corners in ∂B have the same sign.

When $B \in \mathbf{B}$ is no larger than any of its neighbors, we call B simple. In this case, ∂B contains the 2^n corners of B, the $n2^{n-1}$ links connecting them, and no other corners or links. The definition of H_I given above works for all B, but our latter definitions of H might not work for certain corner sign patterns. That is, H may be a partial function. For each H there must be a computational method to determine whether $H(B)$ is defined. Also, H must be defined on all simple boxes, or at the very least all sufficiently small simple boxes.

In order to actually check that condition (1) of Theorem 2.1 is met, we again define a modified interval function $\square_G \nabla F(B)$. If G intersects a k-face of B_0 which is perpendicular to x_i then $(\square_G \nabla F(B))_i$ is the convex hull of $(\square \nabla F(B))_i \cup \{0\}$. Otherwise it is simply $(\square \nabla F(B))_i$. If $(\square_G \nabla F(B) \cdot \hat{n}_G) > 0$ then condition (1) is satisfied for G when δ is smaller than the distance between G and the faces of B_0 which it does not intersect. This brings us to the code in Table 3.2.

This template of an algorithm is used in Section 3.3, in which other H are developed. Note

that some of the tests are trivial when $H = H_I$. For this code, $S_{\partial V}$ is a working set of uncertain boxes. The basic idea is that we take a box out of $S_{\partial V}$ and either categorize it, or determine that it must be split.

The collections S_{V° and S_{V^C} remember boxes which are to be placed in the interior of V or the compliment of V. At line 11 we reference the logic for this and place newly created child boxes accordingly.

For some $H \neq H_I$ we will have cases where $H(B)$ is not defined. When this happens we need to split B, and the test at line 4 makes this happen. Similarly, the normal check at line 8 ensures that condition (1) of Theorem 2.1 is satisfied by the output faces which lie within B. In order for this template of an algorithm to actually provide an algorithm, we must find an H such that these tests do not cause splits when B is sufficiently small.

A final complication is that splitting a box B can introduce new corners on the boundaries of neighbors of B. So even when $H = H_I$, if B' is a neighbor of B then splitting B might change $H(B')$. Starting at line 10, there is code to make sure that if B has a neighbor B' which which is larger than the new children of B, then $H(B')$ is reevaluated.

Drawing it all together, we see that the code terminates and is therefore an algorithm if H is such that the normal check at line 8 is satisfied when B is a sufficiently small simple box. When H has this property, then Theorem 2.1 applies and we have a solution to our problem. So we now focus on the analysis of possible functions H. The primary concern is that we need for normal vector $\hat{n}_G$ of each exterior faces G to eventually satisfy the normal check: $(\square_G \nabla F(G) \cdot \hat{n}_G) > 0$. This condition is necessary to ensure that the result satisfies property 1 of Theorem 2.1. The majority of Section 3.3 is dedicated to proving this property for certain H.

3.3 Convexity Based Cutting

In the case of our initial H_I it does not seem clear that the computational method terminates. Consider a sequence of squares with one negative corner oriented as in Figure 3.1. As the squares shrink the line segment $\overline{ab}$ remains parallel to ∇F along the bottom face. This means that the perpendicular to $\overline{ab}$ remains perpendicular to ∇F. This means that we will continue to have $0 \in (\square \nabla F(B) \cdot \hat{n}_{\overline{ab}})$. So long as this arrangement is preserved, we will not pass the normal check

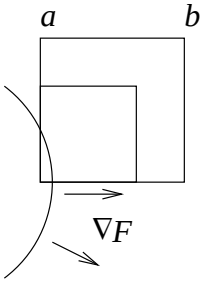


Figure 3.1: Schematic of situation in which we might not terminate if we use H_I. The basic arrangement could be preserved as the squares shrink in size. The normal to line segment $\overline{ab}$ perpendicular to ∇F. Taking a smaller box with the same lower left corner position doesn't necessarily fix the problem.

at line 8.

This is actually not a proof that we have a case for which $H = H_I$ fails to terminate. It is unclear whether the arrangement can be preserved for an infinite sequence of boxes. However, the example illustrates the basic obstruction that we wish to avoid. Certainly, the exact positioning of the curve relative to the lower left hand corner of the box in the diagram could dramatically affect the amount of subdivision required.

Intuitively, to rectify the situation, we might "cut off" the corner at b to give an improved approximation of the surface normal. One way to do this is to define H using convex hulls. Recall the network N of links in our cubical complex, and that a crossing is a link with a particular sign change. Let $H_V(B)$ to be the convex hull of all non-positive corners in ∂B and the midpoints of all crossings which lie in ∂B. Figure 3.2 shows a few simple 2-dimensional examples of $H_V(B)$. This H is simple to describe, but as discussed below we do have to be careful that the faces "match up".

If B has multiple smaller neighbors $\{B_i\}$ across a face B then the union of sets $H_V(B_i) \cap B$ might not be convex, in which case it will be smaller than $H_V(B) \cap G$. A solution to this problem is to declare this to be a situation in which $H_V(B)$ does not exist. We also declare that $H_V(B)$ is undefined when the midpoints of crossings in B are all on some face of B. Also, using this definition, we find that a face of $H_V(B)$ is exterior iff all of its corners are midpoints. The splitting required when $H_V(B)$ is not defined enforces a certain degree of uniformity of resolution in regions where the surface is concave. This uniformity does limit the performance benefits of

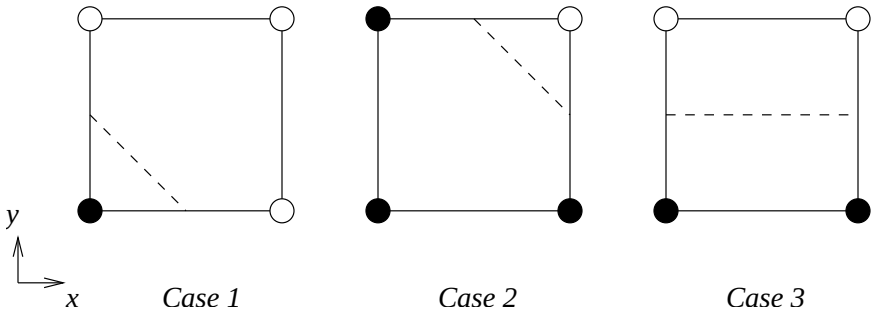


Figure 3.2: A few of the possible corner signs in the 2 dimensional case. Filled circles are non-positive corners, open circles are positive corners.

adaptive subdivision.

In Section 3.3.1 we give a proof that H_V works in the 2 dimensional case, along with a sketch of how to extend the proof to higher dimensions. Then there is a full proof in Section 3.3.3. This means that the algorithm template in Table 3.2 with $H = H_V$ does terminate and is an algorithm in all dimensions. Note that convex hull finding scales poorly with dimension, and convexity is not an essential property of H_V. Also, the matching condition described in the previous paragraph forces a uniformity of box size whenever V is concave. For these reasons the problem of finding a better H is interesting.

3.3.1 2-d Termination

We need to show that the normal check on line 8 of Table 3.2 is eventually successful. We only need to consider simple boxes. Even if the normal check failed for all non-simple boxes, we would effectively revert to a uniform mesh. The regularity of the surface and uniform convergence mean that when B is small, the angular range of $\square\nabla F$ is small. This makes the following theorem useful.

Theorem 3.1. *In the* 2 *dimensional case, if* $\square\nabla F(C)$ *covers a range with angular diameter less than* $\pi/4$*, and if* G *is an exterior face of* $H_V(C)$*, then* $\square\nabla F(C) \cdot \hat{n}_G > 0$*.*

Proof. The restriction on the angular range $\square\nabla F$ means that at least one of $(\nabla F)_x$ and $(\nabla F)_y$ has well-defined sign.

If they both have well-defined sign, then the signs of the corners of B are constrained by

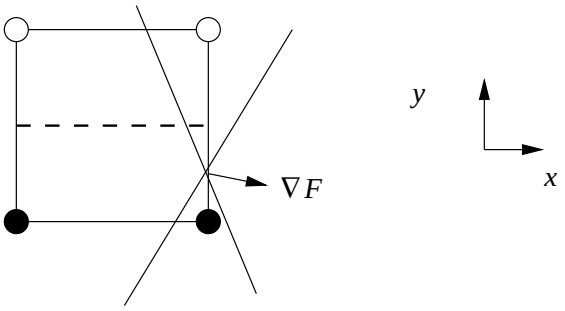


Figure 3.3: The impossibility of Case 3 when ∇F is sufficiently close to the x axis.

this. If they are both positive and the box has corners $(0,0)$ and $(1,1)$ then $(0,0)$ is necessarily non-positive and $(1,1)$ is necessarily positive. Up to symmetry between x and y, this leaves three possible corner sign patterns. They are shown in Figure 3.2, and it is easy to see that in all cases we have $\square\nabla F \cdot \hat{n}_G > 0$.

If only one component has well defined sign, then we have more possibilities. Suppose that $(\nabla F)_x$ is a strictly positive interval while $(\nabla F)_y$ has indeterminate sign. In this case we have more possible corner sign patterns. We have the same cases in Figure 3.2, as well as the top to bottom reflections of all three cases and reflection along $x = y$ of Case 3 in Figure 3.2.

To handle Cases 1 and 2, and their reflections, notice that the x coordinate axis lies with our angular constraint. This implies that every value in the interval $(\nabla F)_x$ is strictly larger than every value in the interval $(\nabla F)_y$. Therefore, when $\hat{n}_G$ is at an angle of $\pi/4$ or $-\pi/4$ to the x axis, the $(\nabla F)_x$ interval dominates the dot product.

The $x = y$ reflection of Case 3 is trivial, so what remains is Case 3 and its top-bottom reflection. For Case 3 to occur the actual curve $F = 0$ must cross both vertical links. However, the angular values of ∇F are constrained to a set of diameter less than $\pi/4$ and this set includes the x axis. This constraint on the tangents to the curve constrains the curve itself to a cone of aperture less than $\pi/4$ with orientation that contains the y axis. If we choose any point on one of the vertical links, and draw a cone with such an aperture and orientation, such as in Figure 3.3, we find that it does not contain any point of the other vertical link. So the angular constraint prevents Case 3 from occurring. □

The higher-dimensional version follows the same basic form. When the angular range is

sufficiently small, we can partition the set of dimensions into a set A_1 with well-defined sign and a set A_0 which is dominated by any desired fixed factor by every dimension in A_1. By making the factor high enough, we handle all cases where $(\hat{n}_G)_i \neq 0$ with some $i \in A_1$. Then we once again use localization of $F = 0$ to eliminate the possibility of $\hat{n}_G$ lying entirely within A_0. The full proof is in Section 3.3.3, but a statement of the theorem when applied to H_V is:

Theorem 3.2. *There is a $\rho > 0$ depending only on dimension such that if $F(x) = 0$, x in a simple box B, and $\square\nabla F(B)$ has angular range with diameter less than ρ then for each $(n-1)$ dimensional face G of $H_V(B)$ with corners that are midpoints of crossings, we have that $(\square\nabla F \cdot \hat{n}_G) > 0$.*

Theorem 3.2 ensures that internal boxes will eventually pass the norm test. To see this notice that the existence of $\omega_{\nabla F}$ and the fact that $0 \notin \nabla F$ on $\{F = 0\}$ implies that there is a size ϵ such that if $\text{diam}(D) < \epsilon$ and B intersects $\{F = 0\}$ then $\square\nabla F(C)$ has diameter less that ρ.

To handle boundary points, let G_0 be some face of B_0. We are going to define a special $\square_{G_0}$ on the surface of G_0 as follows. If C_G is a square within G_0 then it is a face of some box B which lies in B_0 and contacts G. Define $\square_{G_0}\nabla F(C_G)$ as the projection of $\square\nabla F(C)$ onto the hyperspace parallel to G.

There is no guarantee that these modified lower-dimensional interval functions have the same modulus of convergence as our base interval function. However, basic continuity and compactness of B_0 shows that such a modulus exists. We then apply Theorem 3.2 to each lower-dimensional l-face using these modified interval functions. Then eventually $\square\nabla F(C)\cdot\hat{n}_G > 0$ and furthermore if G crosses a face G_0 of B_0 then $\square_{G_0}\nabla F(C \cap G_0) \cdot \hat{n}_{G\cap G_0} > 0$ within the hyperplane parallel to G_0.

So we have that both $\square\nabla F(C)$ and the projection of $\square\nabla F(C)$ onto G_0 lie on the correct side of the hyperplane parallel to G. The set $\square_G\nabla F$ is the convex hull of these sets. So $\square_G\nabla F$ lies entirely within the correct open half space.

3.3.2 Termination Over Rectangles

We first discussed termination when space is subdivided into squares. Current implementation work is focused on this square case. However, it is worth noting that only minor changes to the 2-dimensional proof of Theorem 3.1 are required to handle rectangles with a fixed aspect ratio.

Theorem 3.3. *If B is a rectangle with sides s_x, s_y with $s_x \leq s_y$, and if $\square\nabla F(C)$ covers a range with angular diameter less than $\alpha = \arctan(s_x/s_y)$ and if G is an exterior face of $H_V(C)$ then $(\square\nabla F(C) \cdot \hat{n}_G) > 0$.*

Proof. Very few changes are required from the proof of Theorem 3.1. Because $\alpha \leq \pi/4$ it is still the case that at least one component of F has well-defined sign, and the constraints on the corner signs caused by this does not change. The situation is still trivial when $(\square\nabla F(C))_x$ and $(\square\nabla F(C))_y$ both have well-defined sign.

It also is still the case that $(\square\nabla F(C))_x$ does not have a well-defined sign iff the angular range of $\square\nabla F(C)$ contains the y axis, and similarly for the x axis. What changes is that we must consider individually the possibility of the angular range of $\square\nabla F(C)$ containing each axis.

Consider Case 1 and Case 2 of Figure 3.2. In these cases, we have

$$(\hat{n}_G)_x = \frac{s_y}{\sqrt{s_x^2 + s_y^2}}$$

and

$$(\hat{n}_G)_y = \frac{s_x}{\sqrt{s_x^2 + s_y^2}}.$$

Now, from our angular constraint, when the y axis lies within $\square\nabla F(C)$, we see that $\square\nabla F(C)$ lies angularly strictly within $\arctan(s_x/s_y)$ of the y axis, so

$$\frac{(\square\nabla F(C))_x}{(\square\nabla F(C))_y} < \frac{s_x}{s_y},$$

and so $(\square\nabla F(C) \cdot \hat{n}_G) > 0$ in Cases 1 and Case 2. When $\square\nabla F(C)$ contains the x axis, the inequality to handle the equivalents of Case 1 and Case 2 is even stronger and easier.

What remains is the elimination of Case 3. Just as in the square case, we argue on geometric grounds that it is impossible for $\{F = 0\}$ to intersect both of the lines parallel to the y axis when ∇F lies strictly within an angle of $\arctan(s_x/s_y)$ to the x axis. □

Because of this theorem, we can consider the possibility of dividing space into rectangles instead of squares. We could choose a heuristic for splitting along the x or y direction somewhat

arbitrarily. All that is required is some limitation to allow a constant bound on the aspect ratio of the rectangles.

3.3.3 n-d Termination

In this section, we analyze H_V and a related $H_{V'}$. We define $H_{V'}(B)$ as the closure of the complement of the convex hull of all positive corners in the rectangle B. Alternatively meshing F with $H_{V'}$ gives the same output as meshing $-F$ with H_V. We work in terms of the following properties. We show that if H satisfies all of them when B is simple, then termination occurs.

Property 3.4. Up to rescaling, there are only a finite number of possible $H(B)$. In particular, there is a finite collection of normal vectors which the faces of $H(B)$ may have.

Property 3.5. $H(B)$ correctly separates the corners of B. If a corner is non-positive, then it is in $H(B)$. If a corner is positive, then it is not in $H(B)$.

Property 3.6. The corners of any exterior face G of $H(B)$ are all midpoints of crossings of B.

Property 3.7. None of the crossing whose midpoints are corners of a face G lie within the affine plane containing G.

Property 3.8. Suppose that the corner signs of B are consistent with $\nabla F_i \geq 0$, that is suppose that every crossing in B parallel to the i axis goes from a non-positive value to a positive value as x_i increases. Then we require that $(\hat{n}_G)_i \geq 0$. Similarly when the corner signs allow the possibility that $\nabla F_i \leq 0$ on B, we require that $(\hat{n}_G)_i \leq 0$.

These properties hold for H_V and $H_{V'}$. Property 3.8 is perhaps the least obvious and can be proved for H_V as follows:

Proof. For notational convenience suppose that we have $(\nabla F)_i \geq 0$. Further, suppose that B is positioned at the origin as $B = \{0 \leq x_i \leq 1\}$. Take any any face G of $H(B)$. Let P be the affine hyperplane containing G. It separates all of the non-positive corners and crossing midpoints from some of the positive corners of B. That is, it slices off certain positive corners. Furthermore, because it is the face of a convex hull it does so in a maximal manner, which is to say that

there is not a plane which slices off the same set of corners, and eliminates more of the points in $B/H(B)$.

Now, if $(\hat{n}_P)_i$ is negative, construct an alternative hyperplane P' as follows. Take the intersection of P with $x_i = 0$ and then extend it to a hyperplane by parallel translation in the x_i direction. By the sign consistency $(\nabla F)_i \geq 0$ the corners in $x_i = 1$ which are above points excluded by P are positive and safe to exclude. Similarly, the excluded midpoints must also be safe to exclude. But since $(\hat{n}_P)_i$ was negative, P' actually separates strictly more of the points in B and we have a contradiction. This proves that Property 3.8 holds for H_V. The proof for $H_{V'}$ is similar. □

If H is such that Properties 3.4-3.8 hold, then we claim that the following theorem holds and then Theorem 3.2 becomes a basic application of this. We use $\hat{\nabla}F(x)$ to represent the unit vector associate with ∇F at x. So for example, $\boldsymbol{B}(\rho, \hat{\nabla}F(x))$ is the ball of radius ρ centered at $\nabla F(x)/|\nabla F(x)|$

Theorem 3.9. *There is a $\rho > 0$ depending only on dimension such that if $F(x) = 0$, x in a box B, and $\square\hat{\nabla}F(B) \subseteq \boldsymbol{B}(\rho, \hat{\nabla}F(x))$ then for each $(n-1)$ dimensional exterior face G of $H(B)$, we have that $(\square\nabla F \cdot \hat{n}_G) > 0$.*

Proof. First, we need a technical lemma based on partitioning the coordinate indices $\{1, \ldots, n\}$ into sets A_0 and A_1. Given a point $\vec{y} \in \mathbb{R}^n$, we are going to analyze these divisions in term of the absolute values $|\vec{y}_i|$ of the individual coordinates of $\vec{y}$. Given such a partition, a point $\vec{y}$ with $|\vec{y}| = 1$ and a radius ρ, we define:

$$\alpha(\rho, \vec{y}, A_0, A_1) = \begin{cases} \infty & \min_{i \in A_1} |\vec{y}_i| \leq \rho \text{ or } A_1 \text{ empty} \\ 0 & \min_{i \in A_1} |\vec{y}_i| > \rho \text{ and } A_0 \text{ empty} \\ \frac{\max_{i \in A_0} |\vec{y}_i| + \rho}{\min_{i \in A_1} |\vec{y}_i| - \rho} & \text{otherwise} \end{cases}$$

By choosing cases where α is finite, we force A_1 to become those dimensions for which $\boldsymbol{B}(\rho, \vec{y})$ has a well-defined sign and large magnitude.

α has a number of important properties. It is of course non-negative. If all other parameters are fixed but ρ decreases, then α decreases. If all other parameters are fixed, and A_0 is not

empty, then α is a continuous function of $\vec{y}$ onto $\mathbb{R}^{0+} \cup \{\infty\}$. Even when A_0 is empty, α is an upper semi-continuous function of $\vec{y}$.

The following lemma shows that the choice of $\vec{y}$ is in some sense irrelevant.

Lemma 3.1. *For any $\beta > 0$ there is a $\rho > 0$ such that for every $|\vec{y}| = 1$ there exists a partition A_0, A_1 so that $\alpha(\rho, \vec{y}, A_0, A_1) < \beta$.*

Proof. Consider $\alpha(\rho, \vec{y}) = \min_{A_0, A_1} \alpha(\rho, \vec{y}, A_0, A_1)$. For fixed ρ this is an upper semi-continuous function of $\vec{y}$. For fixed $\vec{y}$, we see that $\lim_{\rho \to 0} \alpha(\rho, \vec{y}) = 0$. So by compactness of the unit ball, we find that Lemma 3.1 holds. □

To see why this lemma is useful, realize that the condition $\square\hat{\nabla}F(B) \subseteq \boldsymbol{B}(\rho, \hat{\nabla}F(x))$ combined with Lemma 3.1 implies a partition so that $(\square\nabla F)_i$ has well defined sign for all $i \in A_1$ and also that the magnitude of these intervals dominates the intervals when $i \in A_0$. Specifically, for any β we can ensure that:

$$\max_{i \in A_0} \max_{x \in B} |\nabla F(x)_i| < \beta \min_{i \in A_1} \min_{x \in B} |\nabla F(x)_i|$$

Continuing the proof of Theorem 3.9, we will take $\beta > 0$ to be fixed later, and will take ρ smaller than that given by Lemma 3.1. Given B and x as in the statement of the theorem, we find that there is a partition A_0, A_1 so that $\alpha(\rho, \hat{\nabla}F(x), A_0, A_1) \leq \beta$. With some abuse of notation we will also use A_0 and A_1 to represent the associated sets of vectors. Specifically, we use them as orthogonal subspaces with direct sum $\mathbb{R}^n$.

Notice that $\alpha(\rho, \vec{y})$ is finite if $\rho < 1/\sqrt{n}$, so A_1 is non-empty. Now, for each face G of $H(B)$, we break things down based on how $\hat{n}_G$ projects into A_1 and A_0.

Using Property 3.8, we see that for each face G of interest, if β is sufficiently small and $(\hat{n}_G)_i \neq 0$ with $i \in A_1$, then $(\hat{n}_G \cdot \square\nabla F) > 0$. Because of the finiteness required by Property 3.4, there is a β small enough to ensure this for all faces of all possible $H(B)$. So, we only need to consider the possibility that $\hat{n}_G$ lies entirely within the span of A_0. This is prevented from happening analogously to how Case 3 was prevented from happening in Section 3.3.1. Consider the way in which localizing ∇F localizes $\{F = 0\}$. In the 2 dimensional case of Section 3.3.1 we were constrained to a cone; here we view the constraint as being near a particular hyperplane, as per the following lemma:

Lemma 3.2. *Fix any $x \in C$ with $F(x) = 0$. Then if there is a vector $\vec{f}$ such that the angle between $\vec{f}$ and $\nabla F(y)$ is less than some $\theta < \pi/2$ for all $y \in C$, then $\{F = 0\} \cap C$ lies within $\operatorname{diam}(C) \tan\theta$ of the affine hyperplane R perpendicular to $\vec{f}$ which passes through x.*

Proof. The condition $\theta < \pi/2$ causes the implicit function theorem to apply. We can find a function on R which measure how far from R in the $\vec{f}$ direction we must move to reach $\{F = 0\}$. Furthermore, $\tan\theta$ is a Lipschitz constant for this function. □

In order to apply this to the proof of Theorem 3.9, we first argue that by making β and ρ sufficiently small we can ensure that a suitable vector $\vec{f}$ exists with $\vec{f} \in A_1$. Take $\theta > 0$ to be a small constant to be determined later.

The definition of β implies that the projection of $\hat{\nabla}F$ onto A_0 has length less than $n\beta$. So we find that there is an β sufficiently small and depending only on θ so that the angle between $\hat{\nabla}F$ and the projection of $\hat{\nabla}F$ onto A_1 will be less than $\theta/2$. Similarly, by making ρ small, we can ensure that the difference in angle between $\hat{\nabla}F(x)$ and $\hat{\nabla}F(y)$ for other $y \in C$ is less than $\theta/2$.

So by Lemma 3.2 we have that $\{F = 0\} \cap B$ lies within a neighborhood of an affine hyperplane perpendicular to $\vec{f}$. Furthermore, the width of the neighborhood, measured as a fraction of $\operatorname{diam}(B)$ is basically $\tan(\theta)$. This can be made as small as necessary by taking β and ρ small.

But if $\hat{n}_G$ stays entirely within A_0 then it is perpendicular to $\vec{f} \in A_1$ and $\vec{f}$ is tangent to the hyperplane P containing G. Since G is a face, it contains a ball $\boldsymbol{B}_r$ with radius r within P. Notice further that r is depends only on sign pattern of the corners and the choice of face. So we can take $r = c\operatorname{diam}(B)$ where $c > 0$ is determined only by dimension.

Now, consider the orthogonal projection of B onto the line spanned by $\vec{f}$. By Property 3.6, all of the corners of G must be from crossings, and in fact they must be parallel to A_0. Otherwise G would cut a line parallel to A_1 which implies $(\hat{n}_G)_i \neq 0$, and we would be in the previous case.

So these crossings collapse to points under this projection. Furthermore $\boldsymbol{B}_r$ must project to a line segment of length r. Also, every crossing contains a point in $\{F = 0\} \cap B$. So we find that there are at two points of $\{F = 0\} \cap B$ which differ by r under this projection.

But this contradicts the localization of $\{F = 0\} \cap B$ caused by Lemma 3.2 when θ is small enough so that $\tan\theta < c$. By making β sufficiently small we can be sure that $\hat{n}_G$ does not lie entirely within A_0. This proves Theorem 3.9.

□

3.4 Advancing Boxes

We now consider another way to mesh, one that does not require the ability to evaluate signs at the corners of boxes. We only consider the 2-dimensional problem. Our detailed explanations will work from the innermost loop to the outermost loop, but a top down overview of the algorithm is as follows:

1. Cover the singularities of $F \cap B_0$ with positive and negative regions. Find a band (ϵ_-, ϵ_+) of regular values of F. This is detailed in Section 3.4.4.

2. Further refine the covering to have regions containing at most one component of $\{F = 0\} \cap B_0$. This is also detailed in Section 3.4.4 and makes use of the mountain pass theorem from Section 2.5.

3. Within each region containing a component of $\{F = 0\} \cap B_0$ we find a starting point $x \in (\epsilon_-, \epsilon_+)$. We then extend this into a chain of line segments. The logic for ensuring that this becomes either a loop, or a connection between two points on ∂B_0 is described in Section 3.4.3.

4. In order to create the chains of line segments, we use a concept called illumination to analyze what points are reachable from a particular x. Section 3.4.2 describes the mechanics of finding another line segment. Section 3.4.1 formally introduces the concept of illumination.

3.4.1 Illumination

The inner core of the algorithm is based on extending our chain of line segments. Given an endpoint x_i we will need to find x_{i+1} satisfying the following two conditions:

1. A normal of the line segment $\overline{x_i x_{i+1}}$ lies within $\square \nabla F$

2. x_{i+1} remains in the chosen band of regular values i.e. $\{\epsilon_- < F(x_{i+1}) < \epsilon_+\}$.

Condition (1) above helps to ensure that condition (1) of Theorem 2.1 will be satisfied by our result. Condition (2) above is used to ensure that conditions (2) and (3) of Theorem 2.1 will be satisfied by our result. Conditions (2) above is also used to ensure termination. Note that (ϵ_-, ϵ_+) can be chosen to lie within any neighborhood containing 0, but as we will see, the actual (ϵ_-, ϵ_+) used might or might not contain 0.

We develop the concept of illumination as a way to ensure condition (1) above. In this section, we assume that we have a subdivision $\mathbf{B}$ of B_0 into boxes. Furthermore, we assume that this subdivision is the result of something like the Singularity Covering Algorithm of Table 3.1. Specifically, we require that for any box B of interest in the following, we have $0 \notin \square\nabla F(B)$. The exact details are discussed in Section 3.4.4.

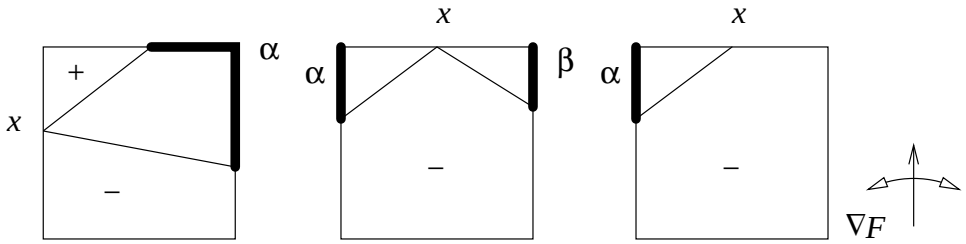
Under the conditions above, let x be a point on a link l_x, but not on a corner. That is to say l_x is the side of the smallest box containing x. Then y is in the illumination of x ($y \in \operatorname{illum} x$) iff the following conditions hold:

1. $y \notin l_x$
2. There is a $B \in \mathbf{B}$ such that $x \in \partial B$ and $y \in \partial B$
3. y can be connected to x by a line segment within B which has a normal vector lying within the angular range of $\square\nabla F(B)$

We use $\operatorname{illum}_B x$ to represent $B \cap \operatorname{illum} x$. Figure 3.4 shows examples of illum_B. When the cone generated by the right angles to $\square\nabla F(B)$ cuts across l_x we say that $\operatorname{illum}_B x$ is transverse, and Figure 3.4(a) is a typical example of this. When the interior of the cone contains l_x, we say that $\operatorname{illum}_B x$ is tangent, and a typical example of this is shown in Figure 3.4(b). When a side of the cone is coincident with l_x we say that $\operatorname{illum}_B x$ is a singular illumination and this is illustrated in Figure 3.4(c).

A segment chain γ is a chain of line segments. A segment chain γ has two endpoints. We call them γ_+ and γ_- with names chosen them so that $F(\gamma_+) \geq F(\gamma_-)$. Each segment in a chain will be a sub-segment of 1-face of a box in $\mathbf{B}$.

In all cases, $\operatorname{illum}_B x$ has one or two components, each of which is a segment chain within ∂B. This set is the intersection of certain cones with $\partial B - l_x$. We take a moment to consider the



(a) Transverse Illumination (b) Tangent Illumination (c) Singular Illumination

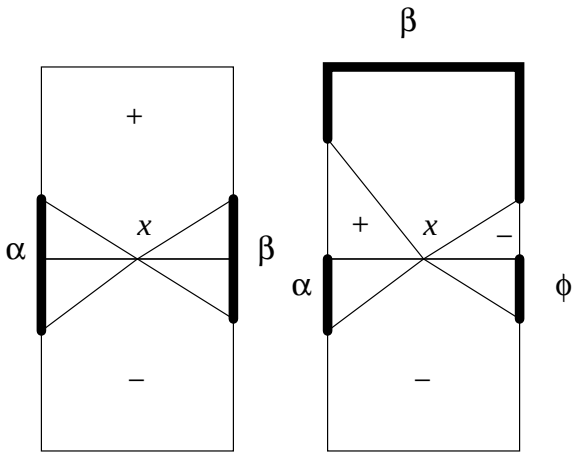
Figure 3.4: Three cases of illumination within a single square, categorized by the angular range of $\square\nabla F$ compared to the orientation of the side containing x.

regions which are not in these cones of possible surface. If y is in such a region then $\square\nabla F(C)$ tells us that either $F(y) < F(x)$ or $F(y) > F(x)$. In Figure 3.4 regions are marked $+$ or $-$ depending on which of these conditions must hold. We next consider what this means for the endpoints of our illumination components.

In Figure 3.4(a), because of the $+$ and $-$ regions we see that α_+ is along the top of the box and $F(x) \leq F(\alpha_+)$ while α_- is along the right side of the box and $F(x) \geq F(\alpha_-)$ So, by the intermediate value theorem, we can find a $y \in \operatorname{illum} x$ such that $F(y) \in (\epsilon_-, \epsilon_+)$. Notice that this never requires an exact calculation of $F(y)$. It only requires a calculation sufficient to certify that $F(y)$ lies within a particular finite interval. Finding such a y is exactly how we advance our algorithm when we cross into a box B such that $\operatorname{illum}_B x$ is transverse. When a component γ of $\operatorname{illum} x$ is such that $F(\gamma_-) \leq F(x) \leq F(\gamma_+)$, we say that γ is balanced.

In fact, we can handle singular illuminations such as Figure 3.4(c) in the same way. Because the top line is at worst perpendicular to ∇F we know that F is non-decreasing as we move left along l_x to α_+. So yet again, the component of the illumination in this box is balanced and there must be a $y \in \operatorname{illum} x$ such that $F(y) \in (\epsilon_-, \epsilon_+)$. For the remainder, we treat the upper left corner of Figure 3.4(c) as containing a zero width region marked $+$, and do not differentiate singular illumination from transverse illumination.

To understand the case of tangential illuminations, we must consider what is on the other side of l_x. So we now focus on $\operatorname{illum} x$ rather than $\operatorname{illum}_B x$. One possibility is that we have another case of tangential illumination. Tangential illumination occurs when the angular range of $\square\nabla F$ contains a normal vector of l_x. Because $\square\nabla F$ is a box, if its angular range contains a



(a) tangent — tangent (b) tangent — transverse

Figure 3.5: When one of the boxes adjacent to l_x is tangential, we must merge the two boxes to find good illumination components.

normal of l_x, it cannot contain a parallel of l_x. So if B_1 and B_2 are the boxes on opposite sides of l_x and $\text{illum}_{B_1} x$, $\text{illum}_{B_2} x$ are both tangential, then $\square\nabla F(B_1)$ and $\square\nabla F(B_2)$ must contain the same normal to l_x. This gives a result like Figure 3.5(a). We see that $\text{illum}\, x$ is made up of two components, both of which are balanced.

The other case is that we have a tangential illumination on one side of l_x and a transverse illumination on the other side. This gives a merged figure something like Figure 3.5(b). We see that in this case $\text{illum}\, x$ has two balanced components, α, β and one component ϕ which is not balanced.

Finally, let us consider what changes when l_x lies between two boxes which are not the same size. Recall that l_x is a side of the smaller box, and part of a side of the larger box. Figure 3.6 contains two typical illumination diagrams. We see that the above arguments are not changed and $\text{illum}\, x$ continues to contain 2 or 3 components, each of which is a segment chain.

Finally, we note that illum is transitive. Assembling these properties into a lemma, we claim the following:

Lemma 3.3. *Suppose that x lies on the interior of a line segment l_x between two boxes C_1 and C_2. Suppose further that $0 \notin \square\nabla F(C_1)$ and $0 \notin \square\nabla F(C_2)$. Then* $\text{illum}\, x$ *has the following properties:*

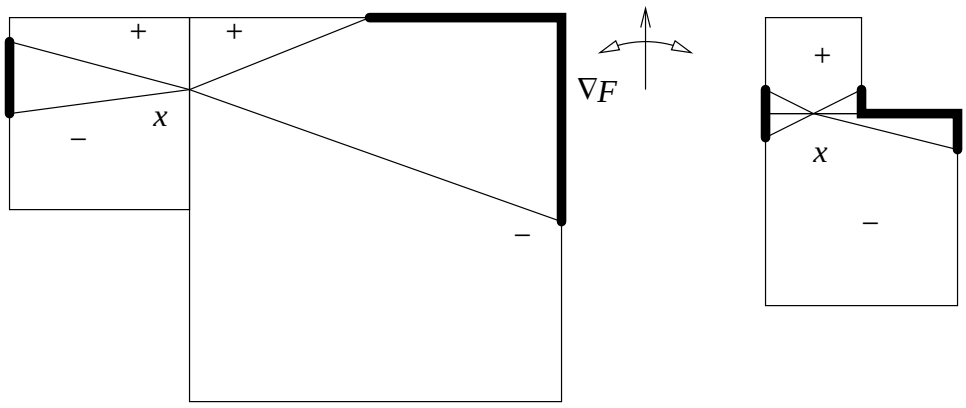


Figure 3.6: Two examples of what illumination of x might look like when $\square\nabla F$ is near to vertical and x lies between boxes of differing sizes.

1. $\operatorname{illum} x \subseteq \partial(C_1 \cup C_2)$

2. $\operatorname{illum} x$ *has two or three components* γ_i.

3. *Exactly two of the* γ_i *are balanced.*

4. $x \in \operatorname{illum} y$ *if and only if* $y \in \operatorname{illum} x$

3.4.2 Box Advancement and One-Sidedness

Using the illumination concept in the previous section, we will create chains of points. Given x_{i-1}, x_i we find x_{i+1} using the algorithm in Table 3.3. In addition, it might be necessary to shrink (ϵ_-, ϵ_+). The reasons for this are discussed in Section 3.4.4.

The last part of the algorithm in Table 3.3 chooses $x_{i+1} \in \operatorname{illum} x_i$, so the line segment $\overline{x_i x_{i+1}}$ has a normal which lies within $\square\nabla F$. The logic of the first part is necessary to ensure that the correct normal of $\overline{x_i x_{i+1}}$ lies within $\square\nabla F$. It must be the normal which lies on the same side of the chain of segments as the corresponding normal of $\overline{x_{i-1} x_i}$. In addition to being a requirement for Theorem 2.1 to apply, this one-sidedness property prevents us from zig zagging back and forth.

When l_x lies between B_1 and B_2, and both $\operatorname{illum}_{B_1} x$ and both $\operatorname{illum}_{B_2} x$ are transverse illuminations, then we have a situation something like the left side of Figure 3.5. We see that the checks at lines 3 and 6 in Table 3.3 ensure that we move across the boxes. We then have

Table 3.3: Box Advancement Algorithm

1: Input: 1) the current end of a point sequence x_{i-1}, x_i 2) a range constraint (ϵ_-, ϵ_+) Preconditions: $x_{i-1} \in \text{illum}\, x_i$, $F(x), F(x_{i=1} \in (\epsilon_-, \epsilon_+)$
2: Let γ_1, γ_2 be the balanced components of $\text{illum}\, x_i$
3: **if** $x_{i-1} \in \gamma_1$ **then**
4: Let$\gamma' = \gamma_2$
5: **else**
6: **if** $x_{i-1} \in \gamma_2$ **then**
7: let $\gamma' = \gamma_1$
8: **else**
9: **if** x_{i-1} is in the same box as γ_1 **then**
10: let $\gamma' = \gamma_1$
11: **else**
12: let $\gamma' = \gamma_2$
13: let x' equal the midpoint of γ'.
14: **while** $F(x') \notin (\epsilon_-, \epsilon_+)$ **do**
15: **if** $F(x') < \epsilon_-$ **then**
16: reduce γ' by removing the portion of the segment chain between x' and γ'_-
17: **else**
18: reduce γ' by removing the portion of the segment chain between x' and γ'_+
19: output x' as x_{i+1}

a principally horizontal chain of line segments and it is the upward pointing normals which lie within $\square\nabla F(B_1)$ and $\square\nabla F(B_2)$.

Next consider when l_x lies between B_1 and B_2 when both $\text{illum}_{B_1} x$ and both $\text{illum}_{B_2} x$ are transverse illuminations, then we have a situation something like the left side of Figure 3.6. We see that the check at lines 3 and 6 ensure that we move across the boxes. We then have a principally horizontal chain of line segments and it is the upward pointing normals which lie within $\square\nabla F(B_1)$ and $\square\nabla F(B_2)$.

Next consider when l_x lies between B_1 and B_2 when both $\text{illum}_{B_1} x$ and both $\text{illum}_{B_2} x$ are tangential. Then we have something like Figure 3.5(a) and we again see that it is the upward pointing normals which lie within $\square\nabla F(B_1)$ and $\square\nabla F(B_2)$.

Finally, we have the case when $\text{illum}_{B_1} x$ is transverse and $\text{illum}_{B_2} x$ is tangential. In this case, $\text{illum}\, x$ has 3 components. When x_{i-1} is in one of the balanced components, then the tests and lines 3 and 6 place x_{i+1} in the other balanced component, and the one sidedness argument is no different. If x_{i-1} is in the unbalanced component, then choosing the balanced component in the same box preserves one sidedness. The test at line 9 makes this happen. In Figure 3.5(b)

Table 3.4: Component Finding

1: Given: 1) **B** a subdivision of B_0 with associated sets S_{V°, $S_{\partial V}$, S_{V^C} as produced by the code in Table 3.5 or Table 3.1 2) (ϵ_-, ϵ_+) such that $\sup_{B \in S_{V^\circ}} \square F(B) < \epsilon_-$ and $\epsilon_+ < \sup_{B \in S_{V^C}} \square F(B)$ 3) $x = x_1$ a point on a link induced by **B** such that $F(x) \in (\epsilon_-, \epsilon_+)$
2: **if** $x_1 \in \partial B_0$ **then**
3: find γ, a balanced component of illum x_1.
4: find $x_2 \in \gamma$ such that $F(x_2) \in (\epsilon_-, \epsilon_+)$ using the loop at line 14 of the box advancement routine in Table 3.3
5: output the line segment $\{x_1, x_2\}$
6: initialize $i = 2$
7: initialize γ_1' to the empty segment chain
8: **else**
9: find illum x, call the balanced components γ_1 and γ_2
10: find $\gamma_1' \subseteq \gamma_1$ such that $(F(\gamma_-), F(\gamma_+)) = (\epsilon_-', \epsilon_+') \subseteq (\epsilon_-, \epsilon_+)$
11: replace (ϵ_-, ϵ_+) with $(\epsilon_-', \epsilon_+')$ for the remainder of this algorithm.
12: choose arbitrary (and temporary) $x_0 \in \gamma_1'$
13: initialize $i = 1$
14: **while** $x_i \notin \gamma_1'$ and $x_i \notin \partial B_0$ **do**
15: find x_{i+1} using the Box Advancement Algorithm in Table 3.3 with
16: output the line segment $\{x_i, x_{i+1}\}$
17: increase i by 1
18: **if** $x_i \notin \gamma_1'$ and $x_1 \notin \partial B_0$ **then**
19: output the line segment $\{x_i, x_1\}$
20: **else**
21: set x_0 to x_2, set i to 1
22: goto line 14

say, we would have that the upward pointing normal vectors are in $\square \nabla F$ of the lower box.

3.4.3 Component Meshing

Suppose that we have an x on a link l_x induced by **B**, such that $F(x) \in (\epsilon_-, \epsilon_+)$. The basic operation to find a mesh component containing x is the already described box advancement algorithm in Table 3.3. However, some logic is required to detect termination and so on. Pseudo-code for this is component finding algorithm presented in Table 3.4.

The algorithm takes as input a band (ϵ_-, ϵ_+) and an initial point x. How we start depends on whether $x \in \partial B_0$. If so, then the code following line 2 takes x_1 as x and finds x_2 in illum x_1. Notice that when $x \in \partial B_0$ the fact that $x \in B \in S_{\partial V}$ ensures that $\text{illum}_B\, x$ is not tangential. This means that $\text{illum}_B\, x$ has only one component, and it is balanced. Then the loop starting at line 14 finds additional segments until we reach ∂B_0 again.

Table 3.5: Modified Singularity Covering Algorithm

```
1: initialize U to be a set of boxes, initially containing B_0.
2: initialize S_{V^∘}, S_{∂V}, S_{V^C} to be sets of boxes, initially empty.
3: while there is a box B ∈ U do
4:   remove B from U
5:   if 0 ∉ □_C ∇F then
6:     place B in S_{∂V}
7:   else
8:     if □F(B) > 0 then
9:       place B in S_{V^C}
10:    else
11:      if □F(B) < 0 then
12:        place B in S_{V^∘}
13:      else
14:        split B and place the resulting child boxes into U
```

When $x \notin \partial B_0$ we set up a components γ_1 of $\operatorname{illum} x$ as a guard. That is, we find a reduced (ϵ_-, ϵ_+) and a reduced γ_1' so that $(\epsilon_-, \epsilon_+) \subseteq (F(\gamma_{1-}), F(\gamma_{2-}))$. This ensures that as the algorithm progresses it will either hit γ_1' or ∂B_0. The the same loop at line 14 finds additional segments until one of these events occurs. When it finds ∂B_0 the test at line 18 notices and run the loop again working in the other direction.

3.4.4 Algorithm

To start with, we localize the singularities and find an appropriate region for the mesh to be in. The first part of this is essentially the same as the singularity covering algorithm in Table 3.1. However, as an optimization, we prefer to have more elements in $S_{\partial V}$ at this stage, so we use the reordered algorithm in Table 3.5. Note also that if we wish to ensure that the normal vectors of faces of V lie within an angular range α of ∇F we can replace the check at line 5 with the condition that $\square\nabla F(C)$ have angular ranges less than α.

Once this is done we further refine our subdivision according to the component separation algorithm in Table 3.6. Using the resulting subdivision, we find (ϵ_-, ϵ_+), a range of regular values of F. Then we find $S_{\partial V}$ such that $B \in S'_{\partial V}$ implies that $\square F(B) \subseteq (\epsilon_-, \epsilon_+)$. To do this we must add additional boxes to S_{V° and S_{V^C}. However, we require that the new boxes not get too close to zero — we insist that for $B \in S_{V^C} \cup S_{V^\circ}$ that $\square F(B)$ does not intersect $(\epsilon_-/2, \epsilon_+/2)$.

Table 3.6: Component Separation Algorithm

find a subdivision $\mathbf{B} = S_{V^\circ} \cup S_{\partial V} \cup S_{V^C}$ of B_0 using the singularity covering algorithm of Table 3.5.
Find $0 > \epsilon_- \geq \sup_{B \in S_{V^\circ}} \Box F(B)$
Find $0 < \epsilon_+ \leq \inf_{B \in S_{V^C}} \Box F(B)$
initialize $S'_{\partial V}$ to an empty set of boxes
while $S_{\partial V}$ is not empty **do**
  choose $B \in S_{\partial V}$, then remove B from $S_{\partial V}$
  if $\Box F(B) > \epsilon_+/2$ **then**
    place B into S_{V^C}
  else
    if $\Box F(B) < \epsilon_-/2$ **then**
      place B into S_{V°
    else
      if $\Box F(B) \subseteq (\epsilon_-, \epsilon_+)$ **then**
        place B into $S'_{\partial V}$
      else
        split B and place its children into $S_{\partial V}$

As a result of this, the new subdivision of B_0 into $\mathbf{B} = S_{V^\circ} \cup S'_{\partial V} \cup S_{V^C}$ has an additional property. It is still the case that any component of $\{F = 0\}$ lies within some connected component of $\bigcup S'_{\partial V}$. However we can also show that each connected component of $\bigcup S'_\partial$ contains at most one connected component of $B_0 \cap \{F = 0\}$.

Proof. By contradiction, suppose that $F(x) = F(y) = 0$ with x and y in different components of $B_0 \cap \{F = 0\}$. Further suppose that x and y lie within a connected component of $\bigcup S_\partial$. Then by definition there is a path γ from x to y which remains in $\bigcup S'_{\partial V}$. By the construction of $S'_{\partial V}$, we know that $F(\gamma) \subseteq (\epsilon_-, \epsilon_+)$. However, because γ connects two components of $\{F = 0\}$ in B_0, we have by Theorem 2.3 that there is a singular point $z \in B_0$ such that $F(z) \in (\epsilon_-, \epsilon_+)$. This contradicts the inequalities ensured by the choice of ϵ_- and ϵ_+. □

Furthermore, if a connected component of $\bigcup S'_{\partial V}$ contains a component of $B_0 \cap \{F = 0\}$ then it must have boxes from both S_{V° and S_{V^C} on its boundary. For if not, we have that the component contains an extreme value of F, which contradicts the construction of $S_{\partial V}$.

Using these subroutines, the final algorithm is fairly simple and presented in Table 3.7. A component of $\bigcup S'_{\partial V}$ may be spurious, that is it might not contain a component of $\{F = 0\}$. However, as noted above, when it does contain such a component, it is bordered by at least one

Table 3.7: Advancing Boxes Algorithm

1: find a subdivision $S_{V^\circ} \cup S'_{\partial V} \cup S_{V^C}$ of B_0 using the component separation algorithm of Table 3.6.
2: **for** each connected component $\boldsymbol{s}$ of $\bigcup S'_{\partial V}$ **do**
3: **if** $\partial \boldsymbol{s}$ intersects elements of both S_{V° and S_{V^C} **then**
4: find segment chain γ between S_{V° and S_{V^C} within $\boldsymbol{s}$
5: find an x along γ such that $F(x) \in (\epsilon_-/2, \epsilon_+/2)$
6: using x, the current subdivision, and $(\epsilon_-/2, \epsilon_+/2)$ as inputs use the component finding algorithm in Table 3.4 to find a mesh component within $\boldsymbol{s}$

box from S_{V° and one from S_{V^C}

4
Implementation Notes

A goal of this research is the ability to produce actual meshes. Several implementations were developed over the course of this research. All implementation work was in C++ and heavy use was made of the Boost Libraries [1], CGAL [**?**], and the C++ Standard Type Library. Some early prototypes also used the MPFI library [18].

4.1 Data Structures

Given the pseudo-code in Table 3.2, a surprisingly difficult aspect was the development of a data structure to manage the n-boxes in the subdivision. It is easy to implement an oct tree (or a 2^n-tree) in a naive manner. However, we need certain access methods which seem to demand additional structure. To begin with, the basic computation of $H(B)$ requires finding all of the crossings which lie on the boundary of B. Furthermore, if B then does need to be split we will need to find all of the n-boxes neighboring B. Furthermore, for our purposes, neighboring does not simply mean the n-boxes which lie across each of the $2n$ different $(n-1)$-faces of B but also the n-boxes which lie diagonally across each l-face for all $0 < l < n$.

It was decided that the data structure requirements for spatial subdivision algorithms are universal enough to warrant a reusable solution. In light of this, a library was developed following the basic format of a CGAL module to maintain spatial subdivision tree. It will be submitted as possible future module to CGAL.

Instead of relying on pointers for navigation, a pointer-free approach was developed. The basic idea is that each $n-box$ is represented by a sequence of nibbles where each nibble contains n bits. The length of the sequence is determines the depth of the square. So for example, in Figure 4.1 on the left we see the four squares at the first level. On the right we see one of the squares subdivided further. In this way, every possible square within the tree is uniquely associated with a string of bits. This sort of pointer-free approach has been used for a long time. According to Samet, "It is difficult to determine the origin of this technique" [19].

To attach data to an n-box, we can place the data in a hash-table keyed by a hash of the string

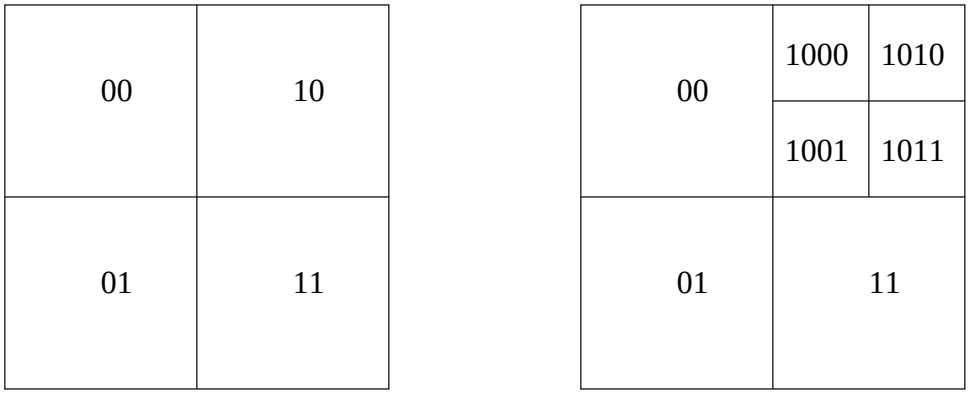


Figure 4.1: Pointer free quad-tree approach.

of bits. To find data associated with, say, a neighboring n-box we perform bit manipulations on the string in order to find the string associated with the neighbor. Then we look up the neighbor's data in the hash-table. Furthermore, we can associate each lower dimensional k-box with a specific n-box. That is, given a k-box B_k, there is exactly one n-box B such that B_k is a face of B containing the corner $(\min_B x_1, \ldots, \min_B x_n)$. So we can assign B_k the string assigned to B concatenated with a bit pattern indicating along which k dimensions B_k has non-zero size. Using this approach, the library supports attaching data to each k-face.

In addition, we arrange things so that the functionality attached to a k-box can access the functionality attached to a j box. To do this, a curiously recurring template pattern is used within the implementation.

4.1.1 Instantiation and Navigation

At the top level, the library provides an implementation of the `SpatialSubdivisionTree` concept:

```
template <int a_dim, template SpatialSubdivisionDataAdder>
class SpatialSubdivisionTree {
  using std::bitset;
 public:
  template <int i_dim> rect;
  template <int i_dim> rect_reference;
  rect_reference<a_dim> root_reference();
};
```

The resulting object contains and maintains the storage of a tree structure. Most interactions with this structure are through nested classes. Within `SpatialSubdivisionTree` we have `rect <i_dim>` classes. Each represents a single rectangular region of the tree with internal dimension `i_dim`. The instantiation of these objects is totally managed by `SpatialSubdivisionTree` objects. As described later, through the use of a template parameter, the user of the library can add data and functionality to the `rect` objects.

The `SpatialSubdivisionTree` concept is modeled by the `Spatial_subdivision_tree` template. The first template parameter, `a_dim`, is an unsigned integer that indicates the ambient, or maximum dimension. Values of `a_dim` as high as 8 are supported, though it is not clear how practical current algorithms would be with such a high splitting factor. The second parameter, `SpatialSubdivisionDataAdder`, is a template parameter which adds data to the `rect` objects.

The primary capability provided is to navigate and access data in the tree structure. The navigation methods are provided through the `RectReference` concept:

```
template <dim_idx i_dim>
class RectReference {
public:
  bool is_null();
  std::bitset<a_dim> collapsed_bits();
  template <dim_idx b_dim>
  RectReference<b_dim> find_boundary(bitset<a_dim> positive_sides,
                                     bitset<a_dim> negative_side);
  template <dim_idx b_dim>
  RectReference<b_dim> find_boundary(bitset<b_dim> dummy,
                                     bitset<a_dim> positive_sides,
                                     bitset<a_dim> negative_side);
  template <dim_idx c_dim>
  RectReference<c_dim> find_coboundary(bitset<a_dim> positive_sides,
                                       bitset<a_dim> negative_sides);

  template <dim_idx c_dim>
  RectReference<c_dim> find_coboundary(bitset<c_dim> dummy,
                                     bitset<a_dim> positive_sides,
                                     bitset<a_dim> negative_sides);
  bool is_split();
  void split();
  RectReference<i_dim> get_parent();
  RectReference<i_dim> get_child(bitset<i_dim> loc);
  RectReference<i_dim> get_child_raw(bitset<a_dim> loc);
  rect<i_dim> &get_rect();
```

```
  unsigned int get_depth();
  template <class Visitor> bool visit_leaves(Visitor &v);
  template <class Visitor>
  bool visit_border_leaves(Visitor &v,
                           bitset<i_dim> plus_sides,
                           bitset<i_dim> minus_sides );
  template <class Visitor, dim_idx b_dim>
  bool visit_neighbors(Visitor &v,bitset<i_dim> plus_sides,
                       bitset<i_dim> minus_sides);
  template <class Visitor, dim_idx b_dim>
  bool visit_neighbors(bitset<b_dim> dummy,
                       Visitor &v,
                       bitset<i_dim> plus_sides,
                       bitset<i_dim> minus_sides );
};
```

In addition to providing navigation and data access methods, the `RectReference` classes are small, copy-able and comparable so that they works with the C++ standard library container classes. Also, a `RectReference` may be null. Null references can be returned from certain of the navigation methods. The `isnull`() method tells the user if a reference is null. If `isnull`() is true, most of the other methods will not work.

When `i_dim` < `a_dim` the rectangle associated with a `RectReference` has zero width in certain dimensions. We say that it has been collapsed in these dimensions, and the `collapsed_bits` method returns a mask which indicates these dimensions.

Starting with any given rectangular cell, we can attempt to navigate in any of the $2n$ possible directions, or in some mixture of them. We have two basic types of motion.

If we are moving in a direction in which our current rectangle has width — if we move in a direction which has not been collapsed — then we collapse those directions and find a lower dimensional boundary rectangle. The `find_boundary` method does this. It collapses the dimensions indicated by `positive_sides` via positive displacement. It collapses the dimensions indicated by `negative_sides` via negative displacements. The dimension of the output rectangle must be specified as a template parameter. In some circumstances this can be done directly, in some cases limitations of C++ require the use of the `dummy` parameter to pass this information to the compiler. `find_boundary` never returns a null reference.

Analogous to the `find_boundary` method, there is a `find_coboundary` method which finds a

neighboring rectangle by expanding some set of displacements. The interface is similar, however it is possible for `find_coboundary` to return a null reference. Specifically, `R.find_coboundary(A,B)` returns `S` such that `S.find_boundary(B,A)` returns `R`, assuming that such an `S` exists in the current tree structure. If no such `S` exits, then `R.find_coboundary(A,B)` returns a null `RectReference`.

```
template <dim_idx a_dim, dim_idx i_dim, class RectBase>
class SpatialSubdivisionDataAdder : public  RectBase {
 public:
  void init_as_child(rect_top & parent, bitset<a_dim> rawloc) {
    RectBase::init_as_child(parent, rawloc);
  }
  template <dim_idx c_dim>
  void init_as_boundary(rect<c_dim> & coboundary,
                        bitset<a_dim> positive_dims,
                        bitset<a_dim> negative_dims) {
    RectBase::init_as_boundary(coboundary, positive_dims,
                                           negative_dims);
  }
  void init_as_root() {
    RectBase::init_as_root();
  }
  void do_split() {
    RectBase::do_split();
  }
  void post_init() {
    RectBase::post_init();
  }
};
```

```
class RectBase {
 public:
  typedef SpatialSubdivisionTree outer_top;
  typedef outer_top::rect<i_dim> rect_top;
  typedef outer_top::rect_handle<i_dim> rect_reference_top;
  rect_reference_top &get_reference();
  void init_as_root();
  void init_as_child(rect_top & parent, bitset<a_dim> rawloc);
  template <dim_idx c_dim>
  void init_as_boundary(rect<c_dim> & cbdry,
                        bitset<a_dim> positive_dims,
                        bitset<a_dim> negative_dims);
  void post_init();
  void do_split();
};
```

When a SpatialSubdivisionTree is instantiated, the second template parameter must itself be a template which models the SpatialSubdivisionDataAdder concept. When a rect<i_dim > class is instantiated, this template is passed parameters which to indicate both the ambient and internal dimensions of the rect which is being created. It is also passed a template parameter RectBase which will be a model of the RectBase concept. SpatialSubdivisionDataAdder must produce a subclass of the input RectBase. The RectBase concept provides several public typedefs to allow SpatialSubdivisionDataAdder to work with the final SpatialSubdivisionTree class. Specifically, outer_top is the type of the containing SpatialSubdivisionTree. Also rect_top and rect_reference_type are convenient declarations of outer_top::rect<i_dim> and outer_top::rect_reference<i_dim>.

Finally, there are a number of methods which SpatialSubdivisionDataAdder may implement if the user desires notification of certain events. There are three initialization methods. When a rect is instantiated exactly one of these will be called. If the rect can be seen as the child of a larger rect of the same dimension, then init_as_child is called. If init_as_child does not apply, but the rect can be seen as the boundary of some higher dimensional box, then init_as_boundary is called. Finally, if neither of the previous two apply, init_as_root is called. Actually, this only happens for the root rect with a_dim = i_dim.

It is intended that the first two init methods propagate data as necessary from the parent or co-boundary rect object. Until this has been completed for a rect r, data might be missed if we created a new boundary or child of r. Therefore, the init methods of r should refrain from accessing boundaries or children of r.

And yet, it might be useful for a rect to access boundaries or even children as part of its initialization. For example, it might be the case that every time a rect of a certain dimension is created, then it should either meet some immediately testable condition, or be split. To support such functionality, after init_as_child or init_as_boundary is called, the method post_init is called. If initialization code which accesses a child or boundary rect is needed, then the library user may override this method to do so.

The final method that an implementation may override is do_split. This method is called to notify the user of the library that the rect in question is being split. In the example section we increment a counter whenever an init method is called and decrement the counter every time

this method is called. The result is a count of the number of leaf nodes within the tree structure.

When a `SpatialSubdivisionDataAdder` implements any of these methods, it should make an identical call to the same method on the `RectBase` input parameter and parent class. This requirement makes it possible to combine multiple `SpatialSubdivisionDataAdder` templates in a straightforward manner. Examples of this are shown in Section 4.1.3.

4.1.2 Other Operations

As described in the previous section, the bulk of the navigation capability provided by the library is through objects which model the `RectReference` concept. The only method which induces a change in the underlying geometric structure is `RectReference::split()`. This method splits a leaf node. Only leaf nodes may be split, and `RectReference::is_split()` indicates wither the node has been split and is now an internal node.

```
class Visitor{
public:
  bool visit (toVisit v);
}
```

It is of course possible to create an iterator for an oct-tree or a quad-tree. However, it is easier to implement and perhaps a bit more natural to use a visitor pattern to loop through the nodes of a tree. To support this, `RectReference` provides the template methods `visit_leaves` , `visit_border_leaves`, and `visit_neighbors` which all take a reference to an object which models the `Visitor` concept.

4.1.3 Examples

As an initial example, the following template can be used to count the number of rectangles of various dimensions:

```
template <CGAL::dim_idx a_dim, CGAL::dim_idx i_dim, class B>
class rect_counter : public B {
public:
  static int count;
  static int leaf_count;
```

```
  void do_split() {
    B::do_split();
    leaf_count--;
  }

  void init_as_child(typename B::rect_top & parent,
                     std::bitset<a_dim> pos){
    B::init_as_child(parent, pos);
    count++;
    leaf_count++;
  }

  template <CGAL::dim_idx c_dim>
  void init_as_boundary(
       typename B::outer_top::template rect<c_dim> & init_by,
       std::bitset<a_dim> p_dims,
       std::bitset<a_dim> m_dims) {
     B::init_as_boundary(init_by, p_dims, m_dims);
     count++;
     leaf_count++;
  }
  void init_as_root() {
    B::init_as_root();
    count++;
    leaf_count++;
  }
};
template <CGAL::dim_idx a_dim, CGAL::dim_idx i_dim, class B>
int rect_counter<a_dim, i_dim, B>::count = 0;
template <CGAL::dim_idx a_dim, CGAL::dim_idx i_dim, class B>
int rect_counter<a_dim, i_dim, B>::leaf_count = 0;
```

We see that this template uses the initialization methods to count the rectangles as they are instantiated. Because the static variable declarations are within the template, they are templated and we find a count for each sort of object. One fine point is that we only count the rectangle objects when they are instantiated, and this is only done when the associated `RectReference` is somehow requested. We can do this automatically by implementing the `post_init` and `do_split` methods as follows:

```
template <CGAL::dim_idx a_dim, CGAL::dim_idx i_dim, class B>
class rect_counter_all : public rect_counter<a_dim, i_dim, B> {
public:
```

```
  void post_init() {
    rect_counter<a_dim, i_dim, B>::post_init();
    std::bitset<a_dim> collapsed =
                       this->get_reference().collapsed_bits();
    for (unsigned int i=0; i < a_dim; i++) {
      if (! collapsed[i]) {
        this->get_reference()
                .find_boundary(std::bitset<i_dim-1>(),
                               std::bitset<a_dim>(0),
                               std::bitset<a_dim>(1<<i));
        this->get_reference()
                .find_boundary(std::bitset<i_dim-1>(),
                               std::bitset<a_dim>(1<<i),
                               std::bitset<a_dim>(0));
      }
    }
  }
  void do_split() {
    rect_counter<a_dim, i_dim, B>::do_split();
    for (unsigned int i=0; i < (1<< i_dim); i++) {
      this->get_reference().find_child(std::bitset<i_dim>(i));
    }
  }
};

template <CGAL::dim_idx a_dim, class B>
class rect_counter_all<a_dim, 0, B>
  : public rect_counter<a_dim, 0, B> {
};
```

To see how these two templates actually behave, we can instantiate them and randomly split rectangles with something like:

```
typedef CGAL::Spatial_subdivision_tree<4, rect_counter>
                                          counting_tree;
typedef CGAL::Spatial_subdivision_tree<4, rect_counter_all>
                                          counting_all_tree;
counting_tree t1;
counting_all_tree t2;
for (int i=0; i < 5; i++) {
 counting_tree::rect_reference<4> h1 = t1.root_reference();
 counting_all_tree::rect_reference<4> h2 = t2.root_reference();
 for (int j=0; j < 50; j++) {
  std::bitset<4> loc (rand() & ((1<<4) - 1) );
  if (!h1.is_split())
   h1.split();
```

```
  h1 = h1.find_child(std::bitset<4>(loc));
  if (!h2.is_split())
   h2.split();
  h2 = h2.find_child(std::bitset<4>(loc));
  }
}
```

The resulting counts are:

```
t1 has 251 rectangles and 5 leaves of dimension 4
t1 has 0 rectangles and 0 leaves of dimension 3
t1 has 0 rectangles and 0 leaves of dimension 2
t1 has 0 rectangles and 0 leaves of dimension 1
t1 has 0 rectangles and 0 leaves of dimension 0
t2 has 3937 rectangles and 3691 leaves of dimension 4
t2 has 23584 rectangles and 21621 leaves of dimension 3
t2 has 52980 rectangles and 47106 leaves of dimension 2
t2 has 52898 rectangles and 45086 leaves of dimension 1
t2 has 15911 rectangles and 15911 leaves of dimension 0
```

4.2 Number Types and Reference Counting

There is heavy use of the the CGAL geometry primitives in the current code, and in most of the earlier prototypes. Especially before a system to cache convex hull computations was implemented, a very large fraction of the running time was spent performing orientation checks and similar geometric operations. Therefore, it is worthwhile to examine the performance of these operations.

CGAL supports several number types, and further supports some hybrid filtered kernels. The d-dimensional kernels do not yet have filtered modes, so the options are to use machine precision arithmetic, or to use some exact number system.

We need to find convex hulls of points which lie on the boundary of a box. This means that the positioning is very singular. It might be that many points will lie within an $(n-1)$-face. It was therefore not a surprise to find that the CGAL's convex hull implementation crashes when attempting these computations with machine floating point numbers. Several exact number types were tried, all based on representing rational numbers exactly. The best two are discussed below.

The first exact number type tried was the `Gmpq` class provided with CGAL. This is a wrapper

around a rational number representation provided by the GMP [10] library. It uses reference counting, and always reallocates on assignment. It was expected that the CGAL provided type would best match the CGAL geometric primitives.

Because of performance issues detailed below, the `mpq_class` provided directly by the GMP [10] was tried next. It is not reference-counted; every instance of `mpq_class` performs its own allocations. However, an expression template system is provided by the GMP library which reduces the number of temporaries created when evaluating and assigning expressions. As it turns out, `mpq_class` performed slightly better than `Gmpq`, apparently because of both this reduced temporary creation and because of the lack of reference count overhead.

While `mpq_class` was slightly better, profiler results for both codes were similar and showed serious room for improvement. The callgrind component of the Valgrind system [21] uses an instrumented processor emulator to give precise information on how much time is spent in individual calls. Typically in our tests, 20-25% of the running time was spent in calls to `malloc` made by the GMP library. Another 20-25% of the running time was spent on calls to `free`. A large fraction of the numbers used in these computations were small. In many cases coordinates values were fractions such as 13/32.

4.3 Convexity Avoidance Experiment

In an attempt to see what could be gained by avoiding convexity computations, an experimental H was devised which involves connecting the crossing midpoints together in a fairly ad-hoc and arbitrary manner. Since the exactness of computation was no longer an issue, it was possible to work in machine doubles. The result worked for all test functions that were produced, and it was much faster than the convexity-based approach. However, it is not yet clear how to prove termination with this H.

5
Results

As discussed in Chapter 4 several convexity-based implementations were developed. Depending on the shape being meshed, and where the subdivisions lie, the choice between H_V and $H_{V'}$ does change the appearance of the result. Figure 5.1 illustrates one example of this.

5.1 Examples in Three Dimensions

Spheres are a nature first function to mesh. Figure 5.1 shows that even for a sphere, H_V and $H_{V'}$ produce different results. Figure 5.2 shows the mesh of a tangle cube funtion $F(x) = 10 + \sum_{i=1}^{3} x^4 - 5x^2$. The code generates `.off` files which were then rendered by Geomview [9].

Table 5.1 gives some statistics from the examples. **Initial Subdivision** is the total number of boxes after the singularity covering algorithm. **Final Subdivision** is the number of boxes after the normal checking algorithm. **Vertexes** is the number of points needed by the faces. **Faces** is the number of faces in the result.

5.2 Examples in Four Dimensions

A major question when considering the output of higher-dimensional algorithms is how to visualize the result. Geomview 1.9.4 [9] defines and loads a format for a 4-dimensional `.off` mesh files. However, its display options are limited and for even medium sized meshes it is very difficult to make any confirmation of the mesh's structure.

One approach is to simplify the problem by taking the intersection of the higher dimensional mesh with a moving 3 dimensional hyperplanes. By doing this, we can find a sequence of cross sections. Figure 5.3 shows such a sequence of intersections with the mesh of a 4 dimensional sphere. Figure 5.4 shows a slice from a more interesting 4-dimensional polynomial.

Name	**Initial Subdivision**	**Final Subdivision**	**Vertexes**	**Faces**
Sphere	80	80	78	152
Tangle Cube	2992	9054	4337	8693

Table 5.1: Statistices from 3-dimensional test functions.

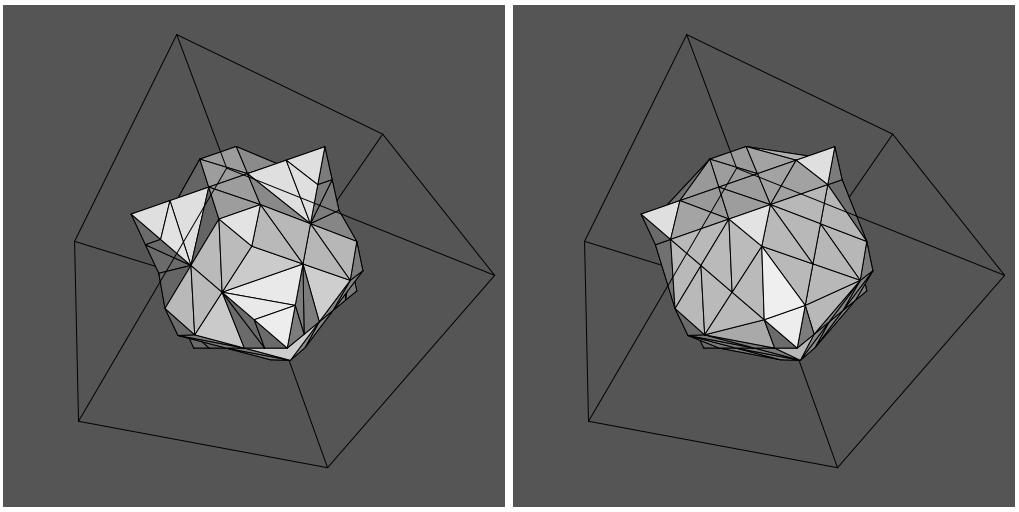


Figure 5.1: There can be a difference between H_V and $H_{V'}$. Here we mesh $F = x_i^2 - 1$, starting with an initial $B_0 = [-2, 2]^3$.

Name	**Initial Subdivision**	**Final Subdivision**	**Vertexes**	**Faces**
Sphere	390	390	264	1488
Tangle Tesseract	31824	159008	72576	440024

Table 5.2: Statistices from 4-dimensional test functions.

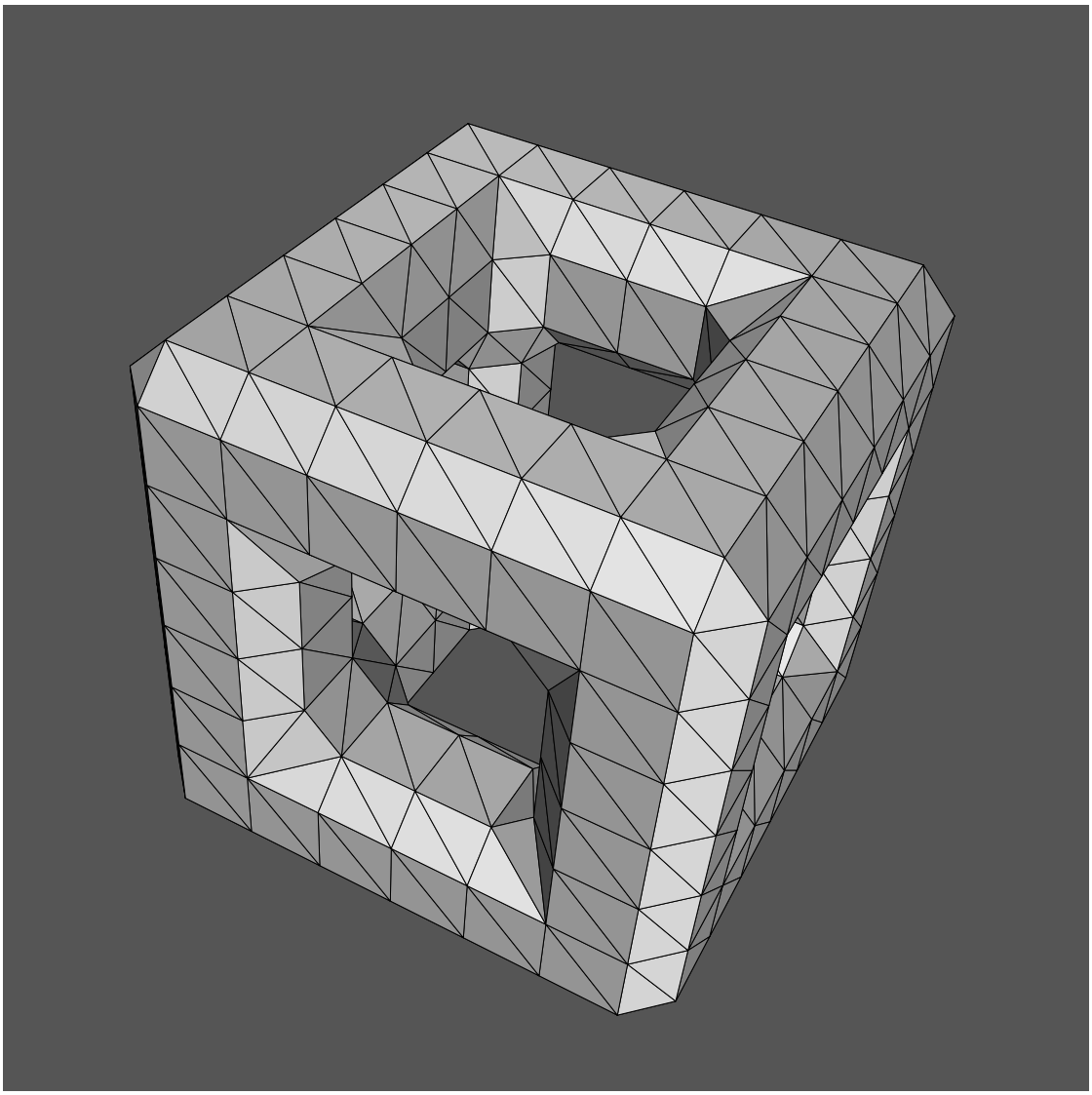


Figure 5.2: Mesh of a tangle cube $F(x) = 10 + \sum_{i=1}^{3} x_i^4 - 5x_i^2$

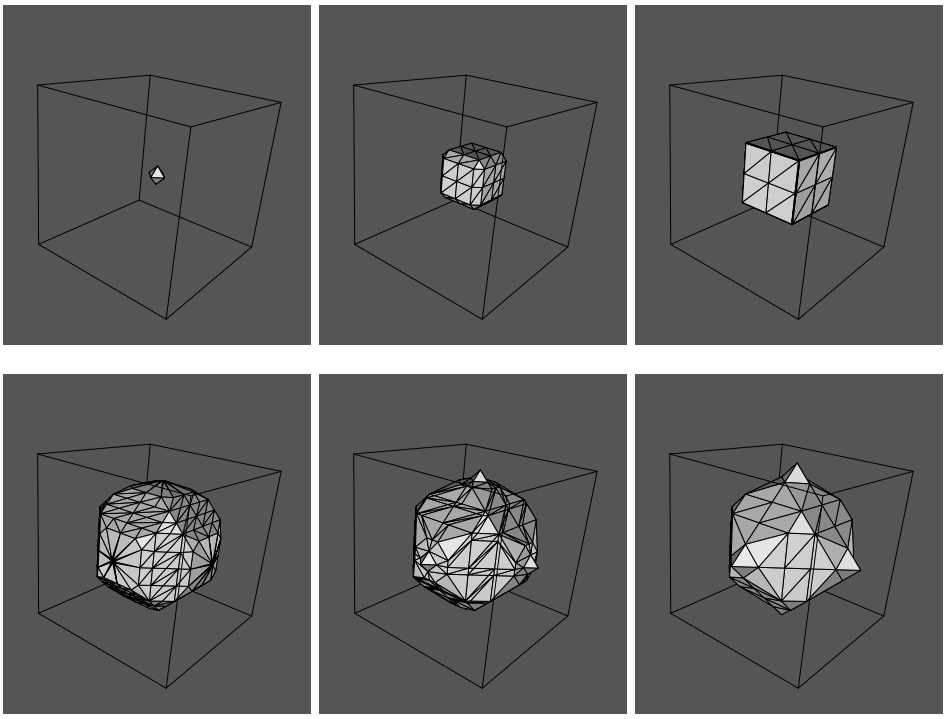


Figure 5.3: Slices of a 4 dimensional Sphere. Rendered with Geomview [9]

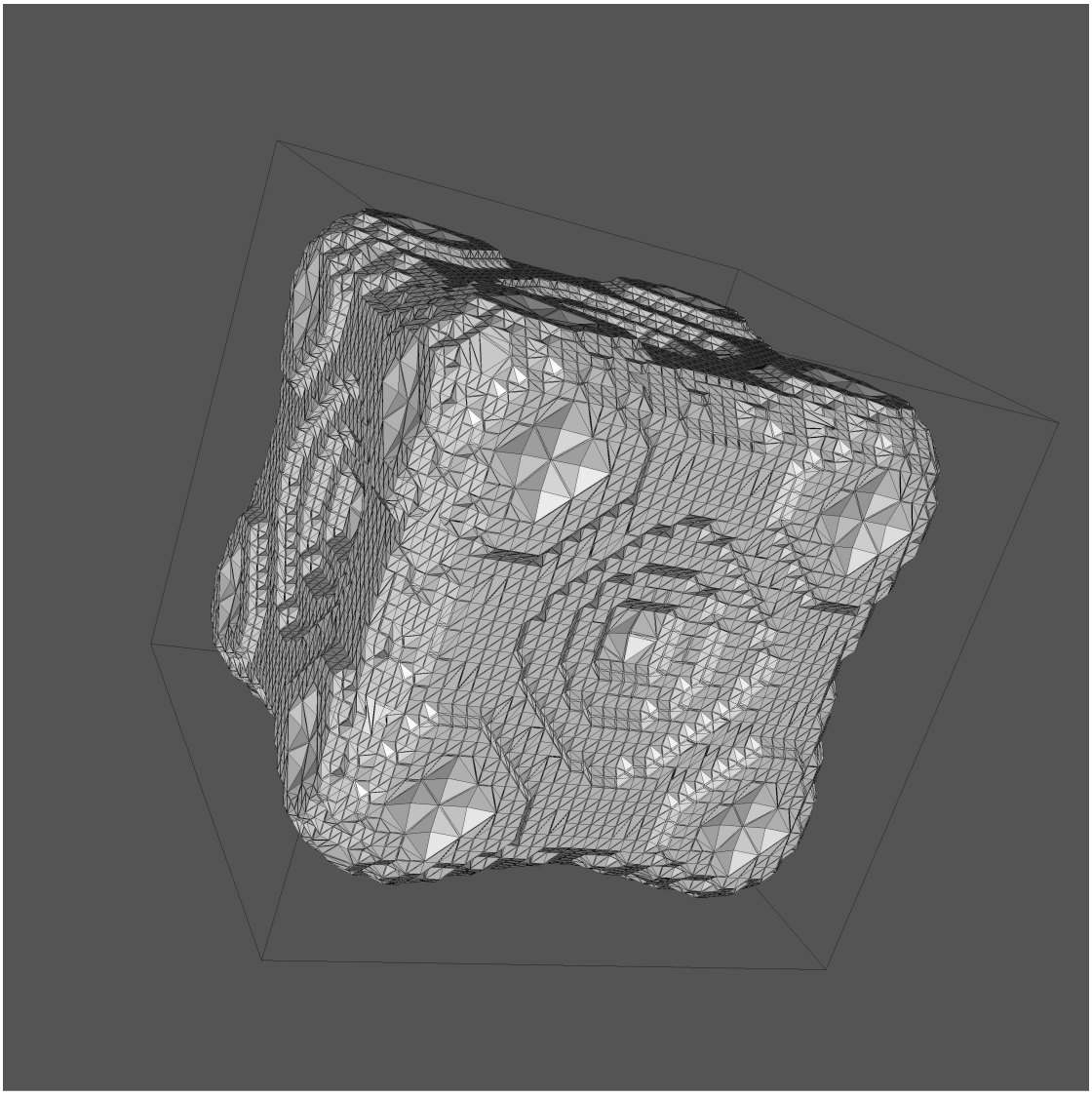


Figure 5.4: A rendering of a 3-dimensional slice of the 4-dimensional polynomial $F(x) = 10 + \sum_{i=1}^{3} x^4 - 5x^2$.

Conclusion

Theorem 2.1 gives a surprisingly simple test of topological accuracy. This test is new, and is usable to ensure topological accuracy in several different types of meshing algorithms. This has been demonstrated, and each of the demonstrations has lead to obvious additional projects. Below, we discuss this in more detail, and list both some open questions and some development projects related to this.

The convexity-based corner cutting of Section 3.3 requires a similar number of boxes as Vegter-Plantinga in 2 and 3 dimensions. Obviously, convex hull computation will never be as fast as a hard-coded table look-up, but if the convex hull computations are cached, a large fraction of that overhead goes away. Notice also that Theorem 2.1 does not particularly require convexity. It seems probable that there is an H which depends less on the interactions between the positions of the crossing midpoints. *Open problem:* Find an H which is faster to compute than H_V and which is guaranteed to terminate.

Contributing to the above mentioned performance challenges were the available geometry kernels, number systems, and their memory management. The best way to reduce the time spent performing memory management may be to switch to using some variation of a copying garbage collector. However, even without switching languages or trusting a pessimistic C++ collector, there is significant room for improvement. One could develop a rational number system with the following features:

1. Stores small numbers without a heap allocation, but dynamically perform such an allocation when necessary. "Small" could be defined by a template parameter, and operations could be designed to work between number with different local sizes.

2. Only perform one heap allocation to store a rational. Existing libraries create a multi-precision rational out of two multi-precision integers, each with its own heap allocation.

3. Share and reference count heap allocated numbers. Copy on write if necessary, but overwrite on assignment if the target has a count of one.

4. Use expression templates to minimize the number of temporaries created.

Open Development Project: Devise a C++ number type with most of the above characteristics. *Alternative:* Devise a geometry computation system in a high level language with good garbage collection support.

The Advancing Boxes Algorithm of Section 3.4 seems to have a lot of potential. It is purely an interval arithmetic algorithm, unlike Marching Cubes and Vegter-Plantinga it does not require exact computation of the sign of F. It allows us to approximate the normal vectors of $\{F = 0\}$ to arbitrary angular precision. *Open Development Project:* Implement the 2-dimensional Advancing Boxes Algorithm. *Open Problem:* Extend Advancing Boxes to work in higher dimensions.

A
Source Code

This appendix contains the verbatim source code used to generate our tangle tesseract example.

First, we have `meshTangleTess.cpp`, the top level file for our tangle cube example.

```
#define CGAL_DISABLE_ROUNDING_MATH_CHECK

#include <CGAL/Gmpq.h>
#include <CGAL/gmpxx.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
//Hack to work around cgal's decision to define a max...
//the boost interval library gets upset with Gmpq otherwise.
namespace CGAL {
  Gmpq max(const Gmpq &a, const Gmpq &b) {
    return std::max(a,b);
  }
}


#include "MeshIntersection.h++"
#include "CompMeshOutput.h++"
#include "CGALHullFinder.h++"
#include "Mesher.h++"
#include "TestFunctions.h++"

#include <fstream>
#include <iostream>
#include <iomanip>
#include <sstream>
#include <set>

template <int length>
std::string intToString(int value) {
  std::stringstream buff;
  std::string r;

  buff << std::setfill('0') << std::setw(length) << value;
  buff >> r;
  return r;
}

#ifndef DIM
#define DIM 4
#endif
```

```
int main(int argc, char** argv) {
  typedef mpq_class Numeric;
  typedef CGAL::Cartesian_d<Numeric> K;

  typedef tangleCubeFunction<DIM, Numeric > fn;

  typedef CGALHullFinder<K, fn, DIM, true> hull_finder;

  fn f;

  Mesher<fn, K, DIM, hull_finder::info>
    mesher(f, Numeric(-4), Numeric(4));
  std::cout << '\n';
  int r = mesher.initialize_unknown();

  std::cout << r <<'\n';

  r = mesher.split_all();
  std::cout << r <<'\n';

  PointMesh<Numeric, DIM> pm;
  mesher.populateMesh(pm);
  std::cout << "Point: " << pm.points.size() << " Faces: " << pm.
      faces.size() << '\n';

#ifndef NO_OUTPUT
  MeshIntersectionComputer<Numeric, 4> mip(pm);

  for (int i=-4000; i <= 4000; i+=20) {
    PointMesh<Numeric, 3> slice;
    mip.intersectPlane(slice, Numeric(i)/Numeric(1000));

    std::cout << "\nSlice number: " << i+4000 << "\n";
    std::cout << "Distinct Corners: " << slice.points.size() <<"\n";
    std::cout << "Distinct Faces: " << slice.faces.size() << "\n";
    std::ofstream file;
    std::string filename = "/tmp/TangleTessSlices/4TangleSlice"+
        intToString<5>(i+4000)+".off";
    file.open(filename.c_str());
    writeOff<Numeric>(file, slice);
    file.close();
  }
#endif //NO_OUTPUT
}
```

`Mesher.h++` contains the generic top level meshing code:

```
#pragma once

#include "CGAL/Spatial_subdivision_tree.h"
#include "Ordinals.h++"
#include "PointMesh.h++"


template <typename FunctionType,
          typename K,
          dim_ord dim,
          template <dim_ord a_dim_,
                    dim_ord i_dim_,
                    class B_>
          class D_>
class Mesher {
  typedef typename CGAL::Spatial_subdivision_tree<dim, D_> tree_type;
  typedef typename tree_type::template rect<dim> cube;
  typedef typename tree_type::template rect_reference<dim> cube_ref;

  typedef typename K::RT Numeric;

  FunctionType &f;
  tree_type tree;
  Numeric low;
  Numeric high;
  LocationInterval<Numeric> inter;

  typedef std::set<cube_ref> set_type;
  set_type unknown;
  set_type valid;

  class ref_collector {
  public:
    std::set<cube_ref> & collection;
    ref_collector(std::set<cube_ref > & collection_) :
      collection(collection_) {
    }
    bool visit(cube_ref h) {
      if(h.get_rect().getSignInfo() == SK_KNOWN_MIXED_SIGN) {
        collection.insert(h);
      }
      return false;
    }
  };

  class face_collector {
    PointMesh<Numeric, dim> &pm;
```

```
  public:
    face_collector(PointMesh<Numeric, dim> &pm_) :
      pm(pm_) {
    }
    void operator()(const typename cube::face_desc &f,
                    const Point<Numeric, dim> &,
                    const Point<Numeric, dim> &){
      pm.add_face(f.face);
    }
  };

  class ref_mover {
  public:
    std::set<cube_ref> & destination;
    std::set<cube_ref> & source;
    ref_mover(std::set<cube_ref > & destination_,
                  std::set<cube_ref > & source_) :
      destination(destination_), source(source_) {
    }
    bool visit(cube_ref h) {
      if(h.get_rect().getSignInfo() == SK_KNOWN_MIXED_SIGN) {
        source.erase(h);
        h.get_rect().clear_hull();
        destination.insert(h);
      }
      assert(source.find(h) == source.end());
      return false;
    }
  };
public:
  Mesher(FunctionType &f_, Numeric low_, Numeric high_)
    : f(f_), low(low_), high(high_), inter(low, high){
    cube_ref root_ref = tree.root_reference();
    cube & root = root_ref.get_rect();

    root.f = &f;
    for (unsigned int i=0; i < dim; i++) {
      root.location[i] = &inter;
    }
    for (unsigned int i=0; i < dim; i++) {
      root_ref.find_boundary(std::bitset<dim-1>(),
                             std::bitset<dim>(1<<i),
                             std::bitset<dim>(0) ).get_rect().border
                                = true;
      root_ref.find_boundary(std::bitset<dim-1>(),
                             std::bitset<dim>(0),
                             std::bitset<dim>(1<<i) ).get_rect().
```

```
                                    border = true;
    }
    root.post_init();

  }

  int initialize_unknown () {
    cube_ref root_ref = tree.root_reference();
    cube & root = root_ref.get_rect();

    ref_collector c(unknown);
    root.get_reference().visit_leaves(c);
    return unknown.size();
  }

  void process_one_unknown () {
    assert(unknown.size() > 0);
    cube_ref r = *unknown.begin();
    unknown.erase(unknown.begin());

    if (r.get_rect().normal_check_ok()) {
      valid.insert(r);
    } else {
      ref_collector c(unknown);

      r.split();
      r.visit_leaves(c);

      ref_mover m(unknown, valid);
      r.visit_all_neighbors(m);
    }
  }

  inline int split_all() {
    while (unknown.size() != 0)
      process_one_unknown();
    return valid.size();
  }
public:
  void populateMesh(PointMesh<Numeric, dim> &pm) {
    face_collector fc(pm);
    for (typename set_type::iterator
           i = valid.begin();
           i != valid.end();
           ++i) {

      cube_ref r = *i;
```

```
      r.get_rect().visit_ext_faces_s(fc);
    }
  }
};
```

PointMesh.h++ implements a rather basic sort of triangle soup mesh with shared vertexes. This sort of mesh maps well to .off files.

```
#pragma once
#include <list>
#include "Simplex.h++"

template <class numeric, dim_ord dim>
class PointMesh {
public:
  typedef Point<numeric, dim> point_type;
  std::vector<point_type > points;
  std::map<point_type, unsigned int> point_idx_map;
  class face_type {
  public:
    unsigned int indices[dim];
  };

  std::list<face_type> faces;
  typedef typename std::list<face_type>::const_iterator face_iter;

  unsigned int get_make_point(const point_type &p) {
    typename std::map<point_type, unsigned int>::iterator i =
      point_idx_map.find(p);
    if (i  != point_idx_map.end() )
      return i->second;

    unsigned int r = points.size();
    points.push_back(p);
    assert(points[r] == p);
    point_idx_map.insert(std::make_pair(p, r));
    return r;
  }

  void add_face(Simplex<numeric, dim, dim-1> face) {
    typename std::list<face_type>::iterator n = faces.insert(faces.
        end(), face_type());
    for (unsigned int i=0; i < dim; i++) {
      n->indices[i] = get_make_point(face.getPoint(i));
    }
  }
```

```
  template <dim_ord x, dim_ord y>
  void rotate (numeric s) {
    assert(x >=0 && y>=0 && dim > 0 && x != y);
    assert(x < dim && y < dim);
    assert(s <= numeric(1) && s >= numeric(-1));

    numeric c = 1 - s*s;
    for (unsigned int i = 0; i < points.size(); i++) {
      numeric temp[dim];
      point_type p = points[i];

      for(unsigned int j=0 ; j < dim; j++) {
        if (j==x) {
          temp[j] = c * p.getCoord(x) + s * p.getCoord(y);
        }
        else if(j==y) {
          temp[j] = c * p.getCoord(y) - s * p.getCoord(x);
        }
        else {
          temp[j] = p.getCoord(j);
        }
      }
      points[i] = point_type(temp);
    }
  }
};
```

Simplex.h++ implements a simplex data structure. Key points are that it uses reference counting and has an order making it suitable to use with the C++ standard library containers.

```
#pragma once

#include <vector>
#include <map>
#include <algorithm>
#include <iostream>
#include <assert.h>
#include <set>
#include "Point.h++"
#include "Vector.h++"
#include "FixedLengthSort.h++"

template <class numeric, dim_ord a_dim, dim_ord s_dim>
class Simplex;

template <class numeric, dim_ord a_dim, dim_ord s_dim>
std::ostream& operator<<(std::ostream&,
```

```
                        const  Simplex<numeric, a_dim, s_dim> &);

template <class numeric, dim_ord a_dim, dim_ord s_dim>
class Simplex {
public:
  typedef Point<numeric, a_dim> point_type;
private:
    class PointSort {
    public:
      bool operator()(const Point<numeric, a_dim> &A,const Point<
          numeric, a_dim> &B) {
        return A < B;
      }
    };

  class Impl {
  private:
    point_type points[s_dim + 1];
  public:
    Impl(const point_type my_points[s_dim+1], bool ordered) {
      if (ordered) {
        for (dim_ord i=0; i < s_dim+1; i++) {
          points[i] = my_points[i];
          if (i > 0) {
            assert(points[i-1]<=points[i]);
            if(points[i-1] == points[i]) {
              assert(false);
            }
          }
        }
      } else {
        point_type temp_points[s_dim +1];
        for (dim_ord i=0; i < s_dim+1; i++) {
          temp_points[i] = my_points[i];
        }
        FixedLengthSorter<point_type, s_dim+1, PointSort >::sort(
            temp_points, PointSort());
        for (dim_ord i=0; i < s_dim+1; i++) {
          points[i] = temp_points[i];
          if (i > 0) {
            assert(points[i-1] <= points[i]);
            if(points[i-1] == points[i]) {
              assert(false);
            }
          }
        }
```

```
      }
    }
    inline point_type getPoint(dim_ord i) const {
      return points[i];
    }
    inline bool operator==(const Impl &o) {
      for (dim_ord i=0; i < s_dim+1; i++) {
        if (points[i] != o.points[i]) {
          return false;
        }
      }
      return true;
    }
    inline bool operator<(const Impl &o) const {
      for (dim_ord i=0; i < s_dim+1; i++) {
        if (points[i] < o.points[i]) {
          return true;
        }
        if (o.points[i] < points[i]) {
          return false;
        }
      }
      return false;
    }

    inline bool contains(const point_type &p) const {
      for (dim_ord i=0; i <s_dim+1; i++) {
        if (points[i] == p)
          return true;
      }
      return false;
    }
  };

  boost::shared_ptr<Impl> impl;
#ifndef NDEBUG
  bool init;
#endif
public:
  typedef Simplex<numeric, a_dim, s_dim> my_type;

  Simplex(const point_type my_pts[s_dim+1], bool ordered = false) :
    impl(new Impl(my_pts, ordered)){
#ifndef NDEBUG
    init = true;
#endif
  }
```

```
  Simplex() {
#ifndef NDEBUG
    init = false;
#endif
  }

  Simplex(const Simplex<numeric, a_dim, s_dim - 1> &s,
          const point_type &p) {

    point_type pts[s_dim+1];
    for (unsigned int i=0; i < s_dim; i++) {
      pts[i] = s.getPoint(i);
    }
    pts[s_dim] = p;
    std::sort(pts, pts + s_dim + 1, PointSort() );
    boost::shared_ptr<Impl> impl_ (new Impl(pts, false));
    impl = impl_;
#ifndef NDEBUG
    init = true;
#endif
  }

  Simplex(const std::set<point_type >&s) {
    assert (s.size() == s_dim + 1);
    point_type pts[s_dim+1];
    typename std::set<point_type >::const_iterator i = s.begin();
    for (unsigned int j=0; j < s_dim+1; j++) {
      pts[j] = *i;
      i++;
    }
    boost::shared_ptr<Impl> impl_ (new Impl(pts, true));
    impl = impl_;
#ifndef NDEBUG
    init = true;
#endif
  }

  Simplex(const my_type &p) :
    impl(p.impl) {
#ifndef NDEBUG
    init = p.init;
#endif
  }

  point_type getPoint(dim_ord i) const {
#ifndef NDEBUG
```

```
    assert(init);
#endif
    assert (i < s_dim +1);
    return impl->getPoint(i);
  }

  bool operator==(const my_type &o) const {
#ifndef NDEBUG
    assert(init && o.init);
#endif
    return impl == o.impl || *impl == *impl;
  }

  bool operator!=(const my_type &o) const {
#ifndef NDEBUG
    assert(init && o.init);
#endif
    return !(*this == o);
  }

  bool operator<(const my_type &o) const {
#ifndef NDEBUG
    assert(init && o.init);
#endif
    if (impl == o.impl) return false;
    return *impl < *o.impl;
  }

  my_type &operator =(const my_type &o){
#ifndef NDEBUG
    assert(o.init);
    init = true;
#endif
    impl = o.impl;
    return *this;
  }

  Simplex<numeric, a_dim, s_dim - 1> boundary(dim_ord i) const {
    assert(i < s_dim+1);
    point_type points[s_dim];
    for (dim_ord j=0; j < i; j++) {
      points[j] = getPoint(j);
    }
    for (dim_ord j=i; j < s_dim; j++) {
      points[j] = getPoint(j+1);
    }
    return Simplex<numeric, a_dim, s_dim - 1> (points, true);
```

```
  }

  bool contains(const point_type &p) const {
    assert(init);
    return impl->contains(p);
  }

  friend std::ostream& operator<<(std::ostream& os, const Simplex<
      numeric, a_dim, s_dim>& r){
    os << "<" << r.impl->getPoint(0);

    for (unsigned int i=1; i < s_dim+1; i++) {
      os << "," << r.impl->getPoint(i);
    }

    os<< ">";
    return os;
  }

  Vector<numeric, a_dim> find_unsigned_normal(const numeric tolerance
      ) const {
    assert(s_dim == a_dim -1);
    Vector<numeric, a_dim> basis[a_dim];
    for (dim_ord i=1; i < a_dim; i++) {
      basis[i-1] = Vector<numeric, a_dim> (impl->getPoint(0), impl->
          getPoint(i)).unit();
    }
    Vector<numeric, a_dim>::complete_basis(basis, tolerance);
    return basis[a_dim-1];
  }

};
```

Point.h++ implements a point data structure. Key aspects are that it uses reference counting and has an order making it suitable to use with the C++ standard library containers.

```
#pragma once

#include "Ordinals.h++"
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <assert.h>


template <class numeric, dim_ord dim>
class Point;
```

```
template <class numeric, dim_ord dim>
std::ostream& operator<<(std::ostream&,
                         const Point<numeric, dim> &);

template <class numeric, dim_ord dim>
class Point {
private:

  class Impl {
  private:
    numeric coords[dim];
  public:
    Impl(numeric my_coords[dim]) {
      for (dim_ord i=0; i < dim; i++) {
        coords[i] = my_coords[i];
      }
    }
    template<typename iter>
    Impl(iter begin, const iter end) {
      for(dim_ord i=0; i < dim; i++){
        coords[i] = *begin;
        begin++;
      }
      assert(end == begin);
    }
    inline const numeric &getCoord(dim_ord i) const {
      return coords[i];
    }
    inline bool operator==(const Impl &o) {
      for (dim_ord i=0; i < dim; i++) {
        if (coords[i] != o.coords[i]) {
          return false;
        }
      }
      return true;
    }
    inline bool operator<(const Impl &o) {
      for (dim_ord i=0; i < dim; i++) {
        if (coords[i] < o.coords[i]) {
          return true;
        }
        if (coords[i] > o.coords[i]) {
          return false;
        }
      }
      return false;
    }
```

```
    inline bool operator>(const Impl &o) {
       for (dim_ord i=0; i < dim; i++) {
         if (coords[i] > o.coords[i]) {
           return true;
         }
         if (coords[i] < o.coords[i]) {
           return false;
         }
       }
       return false;
     }

   };

   boost::shared_ptr<Impl> impl;
#ifndef NDEBUG
   bool init;
#endif
public:

   typedef Point<numeric, dim> my_type;

   Point(numeric my_coords[dim]) :
     impl(new Impl(my_coords)){
#ifndef NDEBUG
     init = true;
#endif
   }

   template <typename iter>
   Point(iter begin, iter end) :
     impl(new Impl(begin, end)) {
#ifndef NDEBUG
     init = true;
#endif
   }

   Point() {
#ifndef NDEBUG
     init = false;
#endif
   }

   Point(const Point<numeric, dim> &p) :
     impl(p.impl) {
#ifndef NDEBUG
```

```
    init = p.init;
#endif
  }

  const numeric &getCoord(dim_ord i) const {
#ifndef NDEBUG
    assert(init);
#endif
    assert(i < dim);
    return impl->getCoord(i);
  }

  class coord_iter {
  public:
    const my_type &p;
    int i;
    coord_iter(const my_type &p_, int i_) : p(p_), i(i_) {
    }
    coord_iter operator ++(int){
      i+=1;
      return coord_iter(p, i-1);
    }
    coord_iter &operator ++(){
      i++;
      return *this;
    }
    const numeric &operator*() {
      return p.getCoord(i);
    }
    bool operator==(const coord_iter &o) {
      assert(p == o.p);
      return i== o.i;
    }
    bool operator!=(const coord_iter &o) {
      assert(p == o.p);
      return i != o.i;
    }
  };

  coord_iter coord_begin() const {
    return coord_iter(*this, 0);
  }
  coord_iter coord_end() const {
    return coord_iter(*this, dim);
  }

  bool operator==(const my_type &o) const {
```

```
#ifndef NDEBUG
    assert(init && o.init);
#endif
    return impl == o.impl || (*impl) == (*o.impl);
  }

  bool operator!=(const my_type &o) const {
#ifndef NDEBUG
    assert(init && o.init);
#endif
    return !((*this) == o);
  }

  bool operator<(const my_type &o) const {
#ifndef NDEBUG
    assert(init && o.init);
#endif
    if (impl == o.impl) return false;
    return (*impl) < (*o.impl);
  }

  bool operator>=(const my_type &o) const {
    return !(*this < o);
  }

  bool operator>(const my_type &o) const {
#ifndef NDEBUG
    assert(init && o.init);
#endif
    if (impl == o.impl) return false;
    return (*impl) > (*o.impl);
  }

  bool operator<=(const my_type &o) const {
    return !(*this > o);
  }

  my_type &operator =(const my_type &o){
#ifndef NDEBUG
    assert(o.init);
    init = true;
#endif
    impl = o.impl;
    return *this;
  }

  friend std::ostream& operator<<(std::ostream& os, const Point<
```

```
      numeric, dim>& r){
    os << "[" << r.impl->getCoord(0);

    for (unsigned int i=1; i < dim; i++) {
      os << "," << r.impl->getCoord(i);
    }

    os<< "]";
    return os;
  }
};
```

CGALHullFinder.h++ uses the CGAL n-dimensional convex hull routine to find H_V or H'_V depending on the convexity_flag template parameter. This is the only file that would need to be changed in order to use a different convex hull algorithm.

```
#pragma once

#include "SimpleHull.h++"
#include "CrudeHull.h++"


#include <CGAL/basic.h>
#include <CGAL/Origin.h>
#include <CGAL/Convex_hull_d.h>
#include <CGAL/Cartesian_d.h>
#include <CGAL/enum.h>

template <typename K, typename FunctionType,
          dim_ord a_dim, bool convexity_flag>
class CGALHullFinder {
  typedef typename K::RT Numeric;

  typedef SimpleHull<K, FunctionType, a_dim, convexity_flag>
     simp_hull;

public:
  template <dim_ord a_dim_, dim_ord i_dim, class B>
  class info : public simp_hull::template data<a_dim_, i_dim, B> {
  };

  template <class B>
  class info <a_dim, a_dim, B> : public simp_hull::template data<
     a_dim, a_dim, B>{
  public:
    typedef typename simp_hull::template data<a_dim, a_dim, B> Base;
```

```
typedef typename Base::rect_top rt;
typedef typename Base::face_desc face_desc;
typedef CGAL::Point_d<K> cgal_point_type;
typedef Point<Numeric, a_dim> point_type;

//Visits all of the faces of the convex hull of the crossings.
//Except, when the convex hull only has dimension n-1, we try
//adding an interior corning and treating it as a crossing.  We
//return true if everything went fine and we found all the needed
//faces, false if we had trouble resolving the n-1 dimensional
//case.
template <typename visitor>
static bool visit_faces(const std::set<point_type> &crossings,
                const std::set<point_type> &interior_corners,
                const std::set<point_type> &exterior_corners,
                visitor &v){
  typedef CGAL::Convex_hull_d<K> hull_type;
  hull_type hull(a_dim);

  typename std::set<point_type>::const_iterator Si;
  for (Si = crossings.begin(); Si != crossings.end(); ++Si) {
    hull.insert(cgal_point_type(a_dim, Si->coord_begin(),Si->
        coord_end()));
  }

  if ((unsigned int) hull.current_dimension() < a_dim - 1) return
      true;


  if (hull.current_dimension() == a_dim -1) {
    const point_type &p=*(interior_corners.begin());
    const cgal_point_type p2 = cgal_point_type(a_dim,
                                               p.coord_begin(),
                                               p.coord_end());
    if (!  hull.is_dimension_jump(p2) ){
      return false;
    }
    hull.insert(p2);
  }

  typename hull_type::Facet_iterator i;

  for (i = hull.facets_begin(); i!= hull.facets_end(); ++i) {
    point_type pts[a_dim];

    for(unsigned int j=0; j < a_dim; j++) {
      CGAL::Point_d<K> p = hull.point_of_facet(i, j);
```

```
          pts[j] = point_type(p.cartesian_begin(), p.cartesian_end())
              ;
        }
        v(face_desc(hull.hyperplane_supporting(i),
                    Simplex<Numeric, a_dim, a_dim-1>(pts) ));
      }
      return true;
    }
  };
};
```

SimpleHull.h++ implements some of the functionality required by Mesher.h++ in a simple way.

```
#pragma once

#include <CGAL/basic.h>
#include <CGAL/Origin.h>
#include <CGAL/Cartesian_d.h>
#include <CGAL/enum.h>

#include "Simplex.h++"
#include "Vector.h++"
#include "FunctionInfo.h++"

template <typename K, typename FunctionType,
          dim_ord a_dim, bool convexity_flag>
class SimpleHull {
  typedef FunctionInfo<K, FunctionType, a_dim, convexity_flag> fi;


public:
  typedef typename K::RT Numeric;
  typedef typename K::FT Num;

  template <dim_ord a_dim_, dim_ord i_dim, class B>
  class data : public fi::template info <a_dim_, i_dim, B> {
  };

  template <class B>
  class data <a_dim, a_dim, B>
    : public fi::template info <a_dim, a_dim, B> {


  public:
    typedef typename fi::template info <a_dim, a_dim, B> Base;
    typedef typename Base::rect_top rt;
    typedef typename Base::outer_top ot;
```

```
typedef typename Base::rect_reference_top ref_top;

typedef typename ot::template rect<0> rect_point;

typedef typename Base::face_desc face_descr;

typedef CGAL::Point_d<K> cgal_point_type;
typedef Point<Numeric, a_dim> point_type;

class point_collector {
public:
  std::set<point_type> & interior_corners;
  std::set<point_type> & exterior_corners;
  std::set<point_type> & crossings;
  point_collector(std::set<point_type> &interior_corners_,
                  std::set<point_type> &exterior_corners_,
                  std::set<point_type> &crossings_):
    interior_corners(interior_corners_),
    exterior_corners(exterior_corners_),
    crossings(crossings_) {
  }
  bool visit(typename ot::template rect_reference<1> &r) {
    std::bitset<a_dim> un_collapsed = CGAL::inverse<a_dim>(r.
       collapsed_bits());
    rect_point &a=r.find_boundary(std::bitset<0>(),
                                  un_collapsed,
                                  std::bitset<a_dim>(0)).get_rect
                                      ();

    rect_point &b=r.find_boundary(std::bitset<0>(),
                                  std::bitset<a_dim>(0),
                                  un_collapsed).get_rect();
    if (a.is_positive()==convexity_flag) {
      exterior_corners.insert(a.point);
    } else {
      interior_corners.insert(a.point);
    }

    if (b.is_positive()==convexity_flag) {
      exterior_corners.insert(b.point);
    } else {
      interior_corners.insert(b.point);
    }

    if (a.is_positive() != b.is_positive() ) {
      crossings.insert(r.get_rect().get_midpoint() );
    }
```

```
    return false;
  }

  bool visit(typename ot::template rect_reference<a_dim - 1> &r)
     {
    std::bitset<a_dim> c = r.collapsed_bits();
    for (unsigned int i=0; i < a_dim; i++) {
      if (! c[i]) {
        std::bitset<a_dim> to_collapse(c);
        to_collapse[i] = true;
        to_collapse.flip();
        CGAL::partition_lister<a_dim> segments(to_collapse);
        do {
          r.find_boundary(std::bitset<1>(),
                          segments.get_part1(),
                          segments.get_part2()).visit_leaves(*
                             this);
        } while (segments.next());
      }
    }
    return false;
  }
};

template <class base_visitor>
class face_checker {
  base_visitor &v;
  const std::list<cgal_point_type> &interior_corners;
  const point_type high;
  const point_type low;
public:
  std::list<face_descr> faces;
  face_checker(base_visitor &v_,
               const std::list<cgal_point_type> &
                  interior_corners_,
               point_type high_,
               point_type low_)
    : v(v_), interior_corners(interior_corners_),
      high(high_), low(low_){
  }
  void operator()(const face_descr &f){
    f.validate();
    bool ext_face = true;
    typename std::list<cgal_point_type>::const_iterator k;
    for(k=interior_corners.begin(); k != interior_corners.end();k
       ++) {
      if (!f.plane.has_on_negative_side(*k)) {
```

```
        ext_face = false;
        break;
      }
    }
    if (ext_face) {
      v(f, high, low);
      faces.push_front(f);
    }
  }
};

template <class base_visitor>
class face_collector {
  base_visitor &v;
  std::list<face_descr> & faces;
public:
  face_collector(base_visitor &v_,
                 std::list<face_descr> &faces_ )
    : v(v_), faces(faces_) {
  }
  void operator()(const face_descr &f){
    faces.push_front(f);
    v(f);
  }
};

typedef typename ot::template rect_handle<a_dim-1> face_handle;

template <typename visitor>
bool visit_ext_faces (const std::set<point_type> &crossings,
                      const std::set<point_type> &
                         interior_corners,
                      const std::set<point_type> &
                         exterior_corners,
                      point_type highs,
                      point_type lows,
                      visitor &v){
  for (unsigned int i=0; i < a_dim; i++) {
    assert(highs.getCoord(i) > lows.getCoord(i));
  }

  typename std::list<cgal_point_type> interior_list;

  typename std::set<point_type>::iterator Si;
  for (Si = interior_corners.begin();
       Si != interior_corners.end();
       ++Si) {
```

```
    interior_list.push_front(cgal_point_type(a_dim,
                                             Si->coord_begin(),
                                             Si->coord_end() ) );
  }

  face_checker<visitor> col(v, interior_list, highs, lows);

  bool r = rt::visit_faces(crossings,
                           interior_corners,
                           exterior_corners,
                           col);
  if (!r)
    return false;

  for (Si= crossings.begin(); Si != crossings.end(); ++Si) {
    bool found_match = false;
    cgal_point_type P (a_dim, Si->coord_begin(), Si->coord_end())
        ;
    for(typename std::list<face_descr>::const_iterator Fi=col.
        faces.begin();
        Fi != col.faces.end();
        ++Fi) {
      if (Fi->plane.has_on_boundary(P)) {
        found_match=true;
        break;
      }
    }
    if (!found_match) {
      return false;
    }
  }
  return true;
};

template <typename visitor>
bool visit_ext_faces_s(visitor &v){

  std::set<point_type > exterior_corners;
  std::set<point_type > interior_corners;
  std::set<point_type> pts;

  point_collector pc(interior_corners, exterior_corners, pts);

  for (unsigned int i=0; i < a_dim; i++) {
    this->get_reference().find_boundary(std::bitset<a_dim-1>(),
                                        std::bitset<a_dim>(1<<i),
                                        std::bitset<a_dim>(0))
```

```
      .visit_leaves(pc);

    this->get_reference().find_boundary(std::bitset<a_dim-1>(),
                                        std::bitset<a_dim>(0),
                                        std::bitset<a_dim>(1<<i))
      .visit_leaves(pc);
  }

  if (interior_corners.size() ==0 ||
      exterior_corners.size() ==0) {
    return true;
  }

  return (static_cast< rt& > (*this))
    .visit_ext_faces (pts,
                      interior_corners,
                      exterior_corners,
                      this->get_reference()
                      .find_boundary(std::bitset<0>(),
                                     std::bitset<a_dim>((1<<a_dim
                                         )-1),
                                     std::bitset<a_dim>(0))
                      .get_rect().point,
                      this->get_reference()
                      .find_boundary(std::bitset<0>(),
                                     std::bitset<a_dim>(0),
                                     std::bitset<a_dim>((1<<a_dim
                                         )-1))
                      .get_rect().point,
                      v);
}

void clear_hull() {
  //e_faces_known = false;
  //e_faces.clear();
}

void init_as_root() {
   Base::init_as_root();
   //e_faces_known = false;
}

void init_as_child(typename B::rect_top & parent,
                   std::bitset<a_dim> pos){
  Base::init_as_child(parent, pos);
  //e_faces_known = false;
}
```

```
  };
};
```

FunctionInfo.h++ keeps tract of what information is known about the signs of boxes. In addition, it contains some of the logic required to perform normal vector checking and other related logic.

```
#pragma once

#include <boost/numeric/interval.hpp>
#include "CGAL/Spatial_subdivision_tree.h"
#include "CGAL/Cartesian_d.h"

#include "LocationData.h++"
#include "Point.h++"
#include "Simplex.h++"

//Represents what we know about the sign f within a region.
//KNOWN_MIXED_SIGN doesn't guarantee that f is actualy mixed, rather
//just that the interval evaulation of f on the square is mixed.

enum signKnowledge{ SK_UNKNOWN_SIGN, SK_POSITIVE_SIGN,
                    SK_NONPOSITIVE_SIGN, SK_KNOWN_MIXED_SIGN };

enum gradKnowledge{ UNKNOWN_GRAD, NONZERO_GRAD, KNOWN_MAYBE_ZERO };

inline signKnowledge subsetKnowledge(signKnowledge s) {
  if (s == SK_KNOWN_MIXED_SIGN)
    return SK_UNKNOWN_SIGN;
  return s;
}

template <dim_ord len>
std::bitset<len> inverse(std::bitset<len> r) {
  r.flip();
  return r;
}

template <dim_ord len>
std::bitset<len> insertZero(dim_ord loc, std::bitset<len-1> s){
  std::bitset<len> r (s.to_ulong());
  r[loc] = false;
  for (unsigned int i=loc+1; i < len; i++) {
    r[i] = s[i-1];
  }
  return r;
```

```
}

template <typename K, typename FunctionType, dim_ord a_dim, bool
    convexity_flag>
class FunctionInfo {
public:
  typedef typename K::RT Numeric;
  typedef typename K::FT Num;
  typedef LocationData<Numeric> loc_type;

  template <dim_ord a_dim_, dim_ord i_dim, class B>
  class info : public loc_type::template data <a_dim_, i_dim, B> {
  };

  template <class B>
  class info <a_dim, a_dim, B>
     : public loc_type::template data <a_dim, a_dim, B>  {
    typedef typename loc_type::template data <a_dim, a_dim, B> base;
    typedef typename base::outer_top ot;

  public:
    signKnowledge sign;
    gradKnowledge grad;
    FunctionType *f;


    class face_desc {
    public:
      CGAL::Hyperplane_d<K> plane;
      Simplex<Numeric, a_dim, a_dim-1> face;
      face_desc(const CGAL::Hyperplane_d<K > &plane_,
                const Simplex<Numeric, a_dim, a_dim-1> &face_) :
        plane(plane_), face(face_) {
      }
      face_desc(){
      }
      void validate()const {
        for (unsigned int i=0; i < a_dim; i++) {
          assert(plane.has_on_boundary(CGAL::Point_d<K> (a_dim, face.
              getPoint(i).coord_begin(),
                                                          face.
                                                              getPoint(
                                                              i).
                                                              coord_end
                                                              ())) );
        }
      }
```

```
};

class norm_check_visitor {

  typedef typename boost::numeric::interval<Num> b_inter;
public:
  bool found_bad_norm;
  const b_inter *grad_comps;
  const Num zero;
  const std::bitset<a_dim> p_borders;
  const std::bitset<a_dim> n_borders;
  norm_check_visitor(const b_inter *grad_comps_,
                     std::bitset<a_dim> p_borders_,
                     std::bitset<a_dim> n_borders_)
    : found_bad_norm(false),
      grad_comps(grad_comps_),
      zero(0),
      p_borders(p_borders_),
      n_borders(n_borders_) {
  }
  void operator()(const face_desc&f,
                  const Point<Numeric, a_dim> & large,
                  const Point<Numeric, a_dim> & small ) {
    if (!found_bad_norm) {
      CGAL::Vector_d<K > norm = convexity_flag ? f.plane.
         orthogonal_vector()
                                              : - f.plane.
                                                  orthogonal_vector
                                                  ();
      using namespace boost::numeric;
      using namespace boost::numeric::interval_lib;

      b_inter sum(Num(0));
      if (p_borders.any() ||
          n_borders.any() ) {
        Numeric zero(0);
        for (unsigned int j=0; j < a_dim; j++) {
          bool b_hit = false;
          assert(large.getCoord(j) > small.getCoord(j));
          if (j > 0) {
            assert( (large.getCoord(j) - small.getCoord(j)) ==
                    (large.getCoord(j-1) - small.getCoord(j-1)) )
                       ;
          }
          for (unsigned int k=0; k < a_dim; k++) {
            assert(large.getCoord(j) >=  f.face.getPoint(k).
```

```
            getCoord(j));
          assert(small.getCoord(j) <=  f.face.getPoint(k).
            getCoord(j));

          if (p_borders[j] && f.face.getPoint(k).getCoord(j) ==
              large.getCoord(j))
            b_hit = true;
          if (n_borders[j] && f.face.getPoint(k).getCoord(j) ==
              small.getCoord(j))
            b_hit = true;
        }
        if (b_hit) {
          sum += (hull(grad_comps[j],zero) * norm.cartesian(j))
            ;
        } else {
          sum += (grad_comps[j] * norm.cartesian(j));
        }
      }
      if (! cergt(sum, zero)) {
        found_bad_norm = true;
      }
    } else {
      for (unsigned int j=0; j < a_dim; j++) {
        sum += (grad_comps[j] * norm.cartesian(j));
      }
      if (! cergt(sum, zero)) {
        found_bad_norm = true;
      }
    }
  }
}
};

bool normal_check_ok() {

  using namespace boost::numeric;
  using namespace boost::numeric::interval_lib;

  typedef interval<Num> b_inter;

  b_inter loc[a_dim];

  std::bitset<a_dim> p_bds(0);
  std::bitset<a_dim> n_bds(0);

  const std::bitset<a_dim> zero(0);
  for (unsigned int i=0; i < a_dim; i++) {
```

```
    loc[i] = b_inter(Num(this->get_interval(i).get_left_coord()),
                     Num(this->get_interval(i).get_right_coord())
                         );
    std::bitset<a_dim> dir(1<<i);
    p_bds[i] = this->get_reference().find_boundary(std::bitset<
        a_dim-1>(),
                                                   dir,zero)
      .get_rect().border;
    n_bds[i] = this->get_reference().find_boundary(std::bitset<
        a_dim-1>(),
                                                   zero,dir)
      .get_rect().border;
  }
  b_inter grad_comps[a_dim];
  f->evalGrad(loc, grad_comps);

  norm_check_visitor v(grad_comps,
                       p_bds,
                       n_bds );


  if (!(static_cast < typename base::rect_top &> (*this)).
      visit_ext_faces_s(v))
    return false;

  return (!v.found_bad_norm);
}

signKnowledge getSignInfo() {
  if (sign != SK_UNKNOWN_SIGN)
    return sign;

  assert(f != 0);
  using namespace boost::numeric;
  using namespace boost::numeric::interval_lib;


  typedef interval<Num> b_inter;
  b_inter loc[a_dim];
  for (unsigned int i=0; i < a_dim; i++) {
    loc[i] = b_inter(this->get_interval(i).get_left_coord(),
                     this->get_interval(i).get_right_coord() );
  }
  const Num zero(0);
  b_inter res = f->eval(loc);
  if (cerle(res, zero)) {
    sign = SK_NONPOSITIVE_SIGN;
```

```
    return sign;
  }
  if (cergt(res, zero)) {
    sign = SK_POSITIVE_SIGN;
    return sign;
  }

  sign = SK_KNOWN_MIXED_SIGN;
  return sign;
}

bool nonZeroGrad() {
  assert (f!= 0);
  if (grad == UNKNOWN_GRAD) {
    using namespace boost::numeric;
    using namespace boost::numeric::interval_lib;

    typedef interval<Num> b_inter;

    b_inter loc[a_dim];
    for (unsigned int i=0; i < a_dim; i++) {
      loc[i] = b_inter(this->get_interval(i).get_left_coord(),
                       this->get_interval(i).get_right_coord() );
    }
    b_inter grad_comps[a_dim];
    f->evalGrad(loc, grad_comps);
    const std::bitset<a_dim> zero(0);
    for (unsigned int i=0; i < a_dim; i++) {
      if (! zero_in(grad_comps[i]) ) {
        std::bitset<a_dim> dir(1<<i);
        if (!(this->get_reference().find_boundary(std::bitset<
            a_dim-1>(),
                                                  dir,zero)
              .get_rect().border
              ||
              this->get_reference().find_boundary(std::bitset<
                  a_dim-1>(),
                                                  zero,dir)
              .get_rect().border)) {
          grad = NONZERO_GRAD;
          break;
        }
      }
    }
    if (grad != NONZERO_GRAD)
      grad = KNOWN_MAYBE_ZERO;
  }
```

```
  if (grad == NONZERO_GRAD) {
    return true;
  }
  assert(grad == KNOWN_MAYBE_ZERO);
  return false;
}

template <typename collector>
void collect_crossings(collector &c) {
  for (unsigned int i=0; i < a_dim; i++) {
    std::bitset<a_dim> zero(0);
    std::bitset<a_dim> dir(1<<i);
    this->get_reference().find_boundary(std::bitset<a_dim-1>(),
                                        dir,zero)
      .get_rect().collect_crossings(c);
    this->get_reference().find_boundary(std::bitset<a_dim-1>(),
                                        zero,dir)
      .get_rect().collect_crossings(c);
  }
}

void init_as_root() {
  base::init_as_root();
  f = 0;
  sign = SK_UNKNOWN_SIGN;
  grad = UNKNOWN_GRAD;
}

void init_as_child(typename B::rect_top & parent,
                   std::bitset<a_dim> pos){
  base::init_as_child(parent, pos);

  sign = subsetKnowledge(parent.sign);
  f = parent.f;
  assert(f != 0);
  if (parent.grad == NONZERO_GRAD) {
    grad = NONZERO_GRAD;
  } else {
    assert(parent.grad == UNKNOWN_GRAD ||
           parent.grad == KNOWN_MAYBE_ZERO );
    grad = UNKNOWN_GRAD;
  }
}
void do_split() {
  base::do_split();
  for (unsigned int i =0; i < (1<< a_dim); i++) {
```

```
      this->get_reference().find_child(std::bitset<a_dim>(i));
    }
  }
  void post_init() {
    base::post_init();

    std::bitset<a_dim> zero(0);
    CGAL::nbit_pattern_lister<a_dim> faces(1);
    do {
      this->get_reference().find_boundary(std::bitset<a_dim-1>(),
                                          faces.current,zero);
      this->get_reference().find_boundary(std::bitset<a_dim-1>(),
                                          zero,faces.current);
      CGAL::partition_lister<a_dim> segments(CGAL::inverse<a_dim>(
          faces.current));
      do {
        this->get_reference().find_boundary(std::bitset<1>(),
                                            segments.get_part1(),
                                            segments.get_part2());
      } while (segments.next());
    } while (faces.next());


    if (! this->get_reference().is_split()) {
      signKnowledge s = this->getSignInfo();
      if (s==SK_KNOWN_MIXED_SIGN) {
        if (! this->nonZeroGrad() ) {
          this->get_reference().split();
        }
      }
    }
  }
};

template <class B>
class info <a_dim, a_dim -1, B>
  : public loc_type::template data <a_dim, a_dim-1, B> {
  typedef typename loc_type::template data <a_dim, a_dim-1, B> base
      ;
  typedef typename base::outer_top ot;
public:

  bool border;
  signKnowledge sign;
  FunctionType *f;

  template <dim_ord c_dim>
```

```
  void init_as_boundary(typename ot::template rect<c_dim> & init_by
      ,
                        std::bitset<a_dim> p_dims,
                        std::bitset<a_dim> m_dims) {
    base::init_as_boundary(init_by, p_dims, m_dims);

    sign = subsetKnowledge(init_by.sign);
    f = init_by.f;
    border = false;
    assert(f!= 0);
  }

  void init_as_child(typename B::rect_top & parent,
                     std::bitset<a_dim> pos){
    base::init_as_child(parent, pos);

    sign = subsetKnowledge(parent.sign);
    f = parent.f;
    border = parent.border;
    assert(f != 0);
  }

  void do_split() {
    base::do_split();
    for (unsigned int i=0; i < (1<< (a_dim-1)); i++) {
      this->get_reference().find_child(std::bitset<a_dim-1>(i));
    }
  }
};

template <class B>
class info <a_dim, 1, B> : public  loc_type::template data <a_dim,
    1, B> {
public:
  signKnowledge sign;
  FunctionType *f;
  typedef typename loc_type::template data <a_dim, 1, B> base;
  typedef typename base::outer_top ot;

  bool is_crossing() {
    assert(! this->get_reference().is_split());

    if (sign == SK_POSITIVE_SIGN ||
        sign == SK_NONPOSITIVE_SIGN)
      return false;

      typedef typename ot::template rect<0> vertex_type;
```

```
      std::bitset<a_dim> zero(0);
      std::bitset<a_dim> dir(this->get_reference().collapsed_bits()
          );
      dir.flip();
      vertex_type a = this->get_reference()
        .find_boundary(std::bitset<0>(), dir,zero).get_rect();
      vertex_type b = this->get_reference()
        .find_boundary(std::bitset<0>(), zero,dir).get_rect();

      return a.is_positive() != b.is_positive();
    }

    Point <Numeric, a_dim> get_midpoint() {
      assert(!this->get_reference().is_split());
      std::bitset<a_dim> dir(this->get_reference().collapsed_bits());
      Numeric coords[a_dim];
      for(unsigned int i=0; i < a_dim; i++) {
        coords[i] = dir[i]?this->get_coord(i)
          :(this->get_interval(i).get_midpoint());

      }
      return Point <Numeric, a_dim>(coords);
    }

    template <dim_ord c_dim>
    void init_as_boundary(typename ot::template rect<c_dim> & init_by
        ,
                          std::bitset<a_dim> p_dims,
                          std::bitset<a_dim> m_dims) {
      base::init_as_boundary(init_by, p_dims, m_dims);

      sign = subsetKnowledge(init_by.sign);
      f = init_by.f;
      assert(f!= 0);
    }

    void init_as_child(typename B::rect_top & parent,
                       std::bitset<a_dim> pos){
      base::init_as_child(parent, pos);

      sign = subsetKnowledge(parent.sign);
      f = parent.f;
      assert(f!=0);
    }

};
```

```
  template <class B>
  class info <a_dim, 0, B> : public  loc_type::template data <a_dim,
     0, B> {
  public:
    signKnowledge sign;
    FunctionType *f;
    typedef typename loc_type::template data <a_dim, 0, B> base;
    typedef typename base::outer_top ot;

    bool is_positive() {
      switch (sign) {
      case SK_KNOWN_MIXED_SIGN:
        assert(false);
      case SK_POSITIVE_SIGN:
        assert(f->is_positive(this->point));
        return true;
      case SK_NONPOSITIVE_SIGN:
        assert(!f->is_positive(this->point));
        return false;
      case SK_UNKNOWN_SIGN:
        break;
      }
      bool r = f->is_positive(this->point);
      sign = r ? SK_POSITIVE_SIGN : SK_NONPOSITIVE_SIGN;
      return r;
    }

    template <dim_ord c_dim>
    void init_as_boundary(typename ot::template rect<c_dim> & init_by
       ,
                          std::bitset<a_dim> p_dims,
                          std::bitset<a_dim> m_dims) {
      base::init_as_boundary(init_by, p_dims, m_dims);

      sign = subsetKnowledge(init_by.sign);
      f = init_by.f;
      assert(f!=0);
    }
  };
};
```

LocationData.h++ is used by FunctionInfo.h++ and the other higher layers to keep tract of the positions of the boxes.

```
#pragma once

#include "CGAL/Spatial_subdivision_tree.h"
```

```
#include "Point.h++"

using CGAL::dim_idx;

template <typename Numeric>
class LocationInterval {
  Numeric &left;
  Numeric &right;
  Numeric midpoint;

  LocationInterval * left_interval;
  LocationInterval * right_interval;

public:

  LocationInterval(Numeric &left_, Numeric &right_)
  : left(left_), right(right_), midpoint((left_ + right_) / Numeric
      (2)) {
    left_interval = 0;
    right_interval =0;
  }
  ~LocationInterval() {
    if (left_interval != 0)
      delete left_interval;
    if (right_interval != 0)
      delete right_interval;
  }
   LocationInterval<Numeric> &get_left_interval() {
    if (left_interval == 0)
      left_interval = new LocationInterval<Numeric>(left, midpoint);
    return *left_interval;
  }
   LocationInterval<Numeric> &get_right_interval() {
    if (right_interval == 0)
      right_interval = new LocationInterval<Numeric>(midpoint, right)
          ;
    return *right_interval;
  }
  Numeric &get_left_coord()  {
    return left;
  }
  Numeric &get_right_coord()  {
    return right;
  }
  Numeric &get_midpoint() {
    return midpoint;
  }
```

```
};


template <typename Numeric>
class LocationData {
public:
  typedef LocationInterval<Numeric> inter;
  template <dim_idx a_dim, dim_idx i_dim, class B>
    class data : public B {
  public:
    typedef typename B::outer_top ot;
     void *location[a_dim];

     inter & get_interval(dim_idx idx) {
      assert(idx < a_dim);
      assert( ! this->get_reference().collapsed_bits()[idx]);
      assert( location[idx] != 0 );
      return *(static_cast< inter *>(location[idx]));
    }

     Numeric & get_coord(dim_idx idx) {
      assert(idx < a_dim);
      assert (this ->get_reference().collapsed_bits()[idx]);
      assert(  location[idx] != 0);
      return  *(static_cast< Numeric *>(location[idx]));
    }

    void init_as_child(typename B::rect_top & parent,
                       std::bitset<a_dim> pos){
      B::init_as_child(parent, pos);

      std::bitset<a_dim> collapsed = this->get_reference().
         collapsed_bits();

      for (unsigned int i=0; i < a_dim; i++) {
        if (collapsed[i]) {
          location[i] = parent.location[i];
        }else {
          location[i] = pos[i] ? static_cast<void *> (& parent.
              get_interval(i).get_right_interval())
                               : static_cast<void *> (& parent.
                                  get_interval(i).get_left_interval()
                                   );
        }
      }
    }
```

```
template <dim_idx c_dim>
void init_as_boundary(typename ot::template rect<c_dim> & init_by
    ,
                      std::bitset<a_dim> p_dims,
                      std::bitset<a_dim> m_dims) {
  B::init_as_boundary(init_by, p_dims, m_dims);


  std::bitset<a_dim> collapsed = this->get_reference().
     collapsed_bits();
  std::bitset<a_dim> c_collapsed = init_by.get_reference().
     collapsed_bits();

  assert((p_dims.to_ulong() & m_dims.to_ulong()) ==0);
  assert((p_dims.to_ulong() & c_collapsed.to_ulong()) == 0);
  assert((m_dims.to_ulong() & c_collapsed.to_ulong()) == 0);
  assert((m_dims.to_ulong() | p_dims.to_ulong() |  c_collapsed.
     to_ulong())
         == collapsed.to_ulong());

  for (unsigned int i=0; i < a_dim; i++) {
    if (c_collapsed[i]) {
      location[i] = init_by.location[i];
    } else {
      if (p_dims[i]) {
        assert(collapsed[i]);
        location[i] = static_cast< void *> (& init_by.
           get_interval(i).get_right_coord());
      } else {
        if (m_dims[i]) {
          assert(collapsed[i]);
          location[i] = static_cast< void *> (& init_by.
             get_interval(i).get_left_coord());
        } else {
          assert(!collapsed[i]);
          location[i] = static_cast< void *> (& init_by.
             get_interval(i));
        }
      }
    }
  }
}
bool contains(const Point<Numeric, a_dim> &p) {
  std::bitset<a_dim> collapsed = this->get_reference().
     collapsed_bits();

  for (unsigned int i=0; i < a_dim; i++) {
```

```
      if (collapsed[i]) {
        if (p.getCoord(i) < get_interval(i).get_left_coord() ||
            p.getCoord(i) > get_interval(i).get_right_coord() )
          return false;
      } else {
        if (p.getCoord(i) != get_coord(i))
          return false;
      }
    }
    return true;
  }

};

template <dim_idx a_dim, class B>
class data<a_dim, a_dim, B> : public B {
public:
  LocationInterval<Numeric> *location [a_dim];

   inter & get_interval(dim_idx idx) {
    assert(idx < a_dim);
    return *location[idx];
  }

  Numeric get_coord(dim_idx idx) {
    assert(false);
    return Numeric(0);
  }

  void init_as_child(typename B::rect_top & parent,
              std::bitset<a_dim> pos){
    B::init_as_child(parent, pos);

    for (unsigned int i=0; i < a_dim; i++) {
        location[i] = pos[i] ?  & parent.get_interval(i).
           get_right_interval()
                             : & parent.get_interval(i).
                               get_left_interval();

    }
  }

  bool contains(const Point<Numeric, a_dim> &p) {
    for (unsigned int i=0; i < a_dim; i++) {
      if (p.getCoord(i) < location[i]->get_left_coord() ||
          p.getCoord(i) > location[i]->get_right_coord() )
        return false;
```

```
    }
    return true;
  }

};

template <dim_idx a_dim, class B>
class data<a_dim, 0, B> : public B{
public:
  Point<Numeric,a_dim>  point;

  typedef typename B::outer_top ot;

  template <dim_idx c_dim>
  void init_as_boundary(typename ot::template rect<c_dim> & init_by
      ,
                        std::bitset<a_dim> p_dims,
                        std::bitset<a_dim> m_dims) {
    B::init_as_boundary(init_by, p_dims, m_dims);
    std::bitset<a_dim> c_collapsed = init_by.get_reference().
       collapsed_bits();

    Numeric coords [a_dim];
    for (unsigned int i=0; i < a_dim; i++) {
      if (c_collapsed[i]) {
        coords[i] = init_by.get_coord(i);
      } else {
        coords[i] = p_dims[i] ? init_by.get_interval(i).
           get_right_coord()
          : init_by.get_interval(i).get_left_coord();
      }
    }
    point = Point<Numeric, a_dim> (coords);
  }

  bool contains(const Point<Numeric, a_dim> &p) {
    return p == point;
  }
};
};
```

Bibliography

[1] The Boost C++ Libraries. `http://www.boost.org`.

[2] M. Burr, S.W. Choi, B. Galehouse, and C. Yap. Complete subdivision algorithms, II: Isotopic meshing of singular algebraic curves. In *Proc. Int'l Symp. Symbolic and Algebraic Computation (ISSAC'08)*, pages 87–94, 2008. Hagenberg, Austria. Jul 20-23, 2008.

[3] B.F. Caviness and J.R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Texts and monographs in Symbolic Computation. Springer, 1998.

[4] 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, Issued December 1, 1987. U.S. Patent Application Serial Number 4,710,876.

[5] Coddington and Levinson. *Theory of Ordinary Differential Equations*, chapter 1. Krieger Publishing Company, 1955.

[6] George E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings of the 2nd GI Conference on Automata Theory and Formal Languages*, pages 134–183, London, UK, 1975. Springer-Verlag.

[7] George E. Collins, Jeremy R. Johnson, and Werner Krandick. Interval arithmetic in cylindrical algebraic decomposition. *J. Symb. Comput.*, 34(2):145–157, 2002.

[8] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[9] Geomview. `http://www.geomview.org`.

[10] The GNU Multiple Precision Arithmatic Library. `http://www.gmp.org`.

[11] Youssef Jabri. *The Mountain Pass Theorem: Variants, Generalizations and Some Applications*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[12] Long Lin and Chee Yap. Adaptive isotopic approximation of nonsingular curves: the parametrizability and non-local isotopy approach. In *Proc. 25th ACM Symp. on Comp. Geometry*, pages 351–360, June 2009. Aarhus, Denmark, Jun 8-10, 2009.

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

[14] Ramon E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[15] Heinrich Müller and Michael Wehle. Visualization of implicit surfaces using adaptive tetrahedrizations. *Scientific Visualization Conference*, 0:243, 1997.

[16] Simon Plantinga and Gert Vegter. Isotopic approximation of implicit curves and surfaces. In Jean-Daniel Boissonnat and Pierre Alliez, editors, *Symposium on Geometry Processing*, volume 71 of *ACM International Conference Proceeding Series*, pages 251–260. Eurographics Association, 2004.

[17] Helmut Ratschek and Jon Rokne. *Computer Methods for the Range of Functions*. Horwood Publishing Limited, Chichester, West Sussex, UK, 1984.

[18] Fabrice Rouillier and Nathalie Revol. Multiple precision floating-point interval library. `http://perso.ens-lyon.fr/nathalie.revol/software.html`.

[19] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[20] John M. Snyder. Interval analysis for computer graphics. In James J. Thomas, editor, *SIGGRAPH*, pages 121–130. ACM, 1992.

[21] Valgrind. `http://www.valgrind.org`.

[22] Chee K. Yap. Robust geometric computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.