Approximation of Implicit Curves Using the Bad Edge Refinement Concept

Master Thesis

14th May 2003

Esther J. Moet

e.j.moet@student.rug.nl

RuG

Rijks*universiteit* Groningen

Master Thesis in Scientific Computing and Imaging
at the Department of Computing Science,
University of Groningen, The Netherlands

April 2003,
Esther Jantje Moet
e.j.moet@student.rug.nl

Contents

Abstract

Two-dimensional implicit curves are defined as the zero set of a function $f : \mathbb{R}^2 \to \mathbb{R}$. Several algorithms for piecewise linear approximations of regular implicit curves exist, usually without guaranteeing a correct topology. In this thesis, we develop and implement an algorithm that guarantees to find a topologically correct, polygonal approximation of a given implicit curve. This adaptive enumeration algorithm uses mesh refinement, based on the *bad edge concept*, to refine both quadtree and Delaunay triangulation meshes. This theoretical framework then defines the curve's topologically correct, piecewise linear approximation.

Acknowledgements

I would like to thank everyone who has been of assistance to me in completing my Master's thesis. In the first place, I would like to thank my supervisor, Gert Vegter, for his encouragement, guidance and assistance throughout my research. Furthermore, I would like to thank both Nico Kruithof and Simon Plantinga for their advise and helpful comments. Not in the last place, I appreciated Nico's endless patience on Friday afternoons.

Chapter 1

Introduction

This thesis is written as partial fulfillment of the requirements for the Master's Degree in Scientific Computing and Imaging at the University of Groningen, The Netherlands. We give a detailed overview of the theory, design, implementation, and results of this Master's research project. The subject of research is a method to create piecewise linear approximations of implicit curves.
In this first chapter, we describe the problem of approximating implicit curves, and introduce some definitions with respect to implicit curves. Next, the related work on this matter is described and finally, the project goals are stated.

1.1 Problem Description

There are three types of curves: explicit, implicit, and parametric curves. Explicit curves are the best-known curves of these three. These curves are of the form $y = f(x)$, the example below (figure 1.1) shows the graph of $y = x^2$, a parabola. For every x-value, there is exactly one possible y-value. To display such a curve, a series of y-values (belonging to sequential x-values) is computed and these points are connected.
In the parametric representation, curves are defined with an extra variable, generally called t. An example is $(x, y) = (cos(t), sin(t))$, which is the unit circle, shown on the right in figure 1.1. Displaying such a curve is performed by computing sequential $(x(t), y(t))$-coordinates by taking sequential t-values and connecting these points.

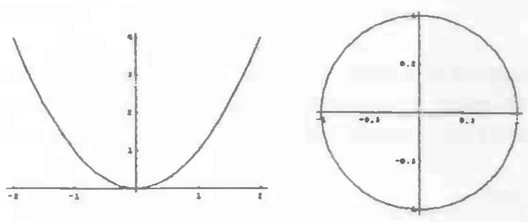

Figure 1.1: Examples of an explicit and an parametric/implicit curve. Left: $y = x^2$, right: $x^2+y^2-1 = 0$ or $(cos(t), sin(t))$.

Implicit curves on the other hand, are defined as the zero set of a function $f(x, y)$. These curves exists of the points for which $f(x, y)$ equals zero. In figure 1.1, $x^2 + y^2 - 1 = 0$ is shown, the unit circle again. The parametric representation $(cos(t), sin(t))$ thus is just an explicit way of describing this implicit curve. In an implicit curve, every x-value can correspond to multiple y-values, and the other way around (which means the curve is not injective or surjective). This is the reason why the explicit way of displaying a curve does not work for implicit curves. In 3D, $f(x, y, z) = 0$, the same holds for implicit surfaces. Approximation techniques have to be used to display these kind of curves or surfaces.

Implicit curves are common in physics and the medical world. They are also powerful for modelling purposes in Computer Graphics. In particular their 3D counterparts, implicit surfaces, can be generated in such a way that they model real-life objects very accurately, see figure 1.2.
This leads to the need of displaying these curves by precise approximations that are guaranteed to be *topologically correct*. We give an exact definition of a curve's approximation being topologically correct later, after some necessary terminology is introduced.

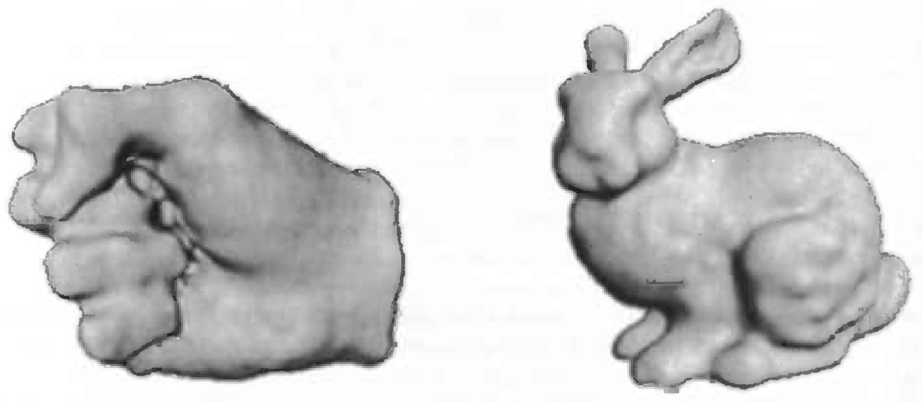
Figure 1.2: Two examples of solid modelling with implicit surfaces [1].

1.2 Implicit Curves

Implicit curves in 2D and surfaces in 3D are defined by equations of the type

$$\begin{aligned} f(x,y) &= 0 \text{ (in 2D) and} \\ f(x,y,z) &= 0 \text{ (in 3D),} \end{aligned}$$

where f is a real-valued C^2 function on $\mathbb{R}^2$ or $\mathbb{R}^3$, respectively. A C^2 function can be differentiated at least two times, which implies $f_{xy} = f_{yx}$, and has continuous derivatives. Figure 1.3 shows some examples of implicit curves. Although we keep implicit surfaces in mind, the algorithms presented in thesis only consider the approximation of implicit curves, the 2D-case. We will come back to a higher dimension in chapters 6 and 7.
In every point $p = (x,y) \in \mathbb{R}^2$, and therefore in every point on the implicit curve, the *gradient* of f, written as ∇f, is defined as

$$\nabla f(p) = (\frac{\partial f}{\partial x}(p), \frac{\partial f}{\partial y}(p)).$$

The gradient $\nabla f(p)$ is a 2D-vector, which in every point p on the curve is normal to the curve.
As mentioned earlier, implicit curves cannot be displayed as easily as graphs of the form $y = f(x)$. This is mainly because they are not necessarily injective and surjective. Furthermore, the gradient can turn 180 degrees over a small distance, which can result in complex curves. Finally, the curve can consist of multiple connected components, closed or open. An implicit curve can have singular points, also called singularities. These are the points $p = (x,y) \in \mathbb{R}^2$ that satisfy

$$\nabla f(p) = (\tfrac{\partial f}{\partial x}(p), \tfrac{\partial f}{\partial y}(p)) = (0,0),$$

which are the extreme points if the 3D surface of $z = f(x,y)$ is considered. In this thesis, we only consider regular (non-singular) implicit curves. The curves belonging to this special group do not pass through any of their singular points and thus are not self-intersecting.

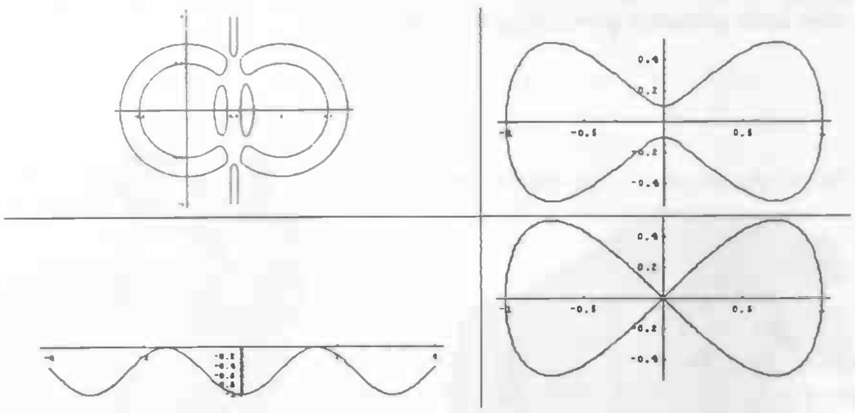

Figure 1.3: Four examples of implicit curves, displayed by *Mathematica* using the *ImplicitPlot* package. The first one, a good example of the modelling possibilities of implicit curves, is defined by the equation $f(x,y) = ((x-1)^2+y^2-0.5)\cdot((x-1)^2+y^2-0.25)\cdot(x^2+y^2-0.5)\cdot(x^2+y^2-0.25)\cdot(x-0.46)\cdot(x-0.54)+0.000001=0$. The second one is defined by $f(x,y) = x^2(1-x)(1+x)-y^2+0.01=0$. The third curve, which is not polynomial, is defined by the equation $f(x,y) = cos^2(x)+y=0$. The last one is a singular, self-intersecting curve, defined by the same equation as the second one, minus the 0.01 term.

When is an approximation of an implicit curve topologically correct? Let $f(x,y)$ be the curve we want to display the zero set of. Let the implicit curve, $c = f^{-1}(0)$, be a (regular) curve, contained in a box B. A *homeomorphism* $h : B \to B$ is a continuous bijection with a continuous inverse. Note that if B is compact, every continuous bijection has a continuous inverse.

Definition 1 *A curve c' is a* topologically correct approximation *of c if there is a homeomorphism $h : B \to B$ such that $h(c) = c'$.*

Figure 1.4 illustrates this definition with an example of a curve and its approximation. Intuitively, this means that a topologically correct approximation has the same number of connected components as the implicit curve and these components do only intersect themselves if the corresponding component of the curve does.

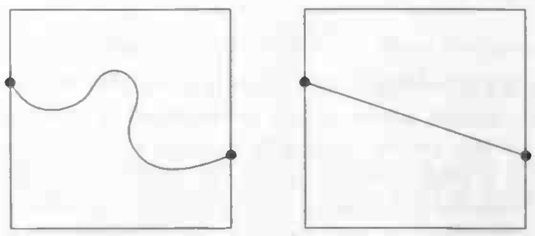

Figure 1.4: The straight line segment on the right is a topologically correct approximation of the curve on the left.

1.3 Related Work

As mentioned in the previous section, several algorithms for piecewise linear approximation of implicit curves exist, usually without guaranteeing a correct topology. The methods that have been proposed until now can be subdivided into two groups: continuation methods and adaptive enumeration methods. We discuss some examples of these methods and their (dis)advantages.

1.3.1 Continuation Methods

The first group of methods is based on continuation, or following, of the curve. These methods are *curve-based*, as opposed to the adaptive enumeration methods, which are discussed later and are *grid-based*. A continuation algorithm starts with a (set of) seed point(s) on the curve and computes more sample points of the approximation while following the curve. Eventually, connecting these sample points forms a piecewise linear approximation. We describe a few of these methods in short here.
A first continuation method is based on **Bresenham's algorithm** for displaying a certain function of the form $y = f(x)$. Bresenham's original algorithm decides which pixels of a pixel map should be set for a reasonable display of f [2]. This decision between pixels is based on the sign of $y - f(x)$ on either side of the curve. On the curve itself $y - f(x)$ equals zero, but its two half-planes correspond to $y - f(x) < 0$ and $y - f(x) > 0$. Aken and Novak adapted this algorithm to decide which pixels should be set in order to display a given implicit curve reasonably [3].
Both Aron [4] and Taubin [5] proposed **special robust algorithms for algebraic curves** that are defined by polynomial functions and therefore have special properties. For example, it is possible to know the maximum number of singular points beforehand. Aron and Taubin take advantage of these properties in their algorithms. In this way, specific algorithms can be devised that are not generally applicable, but are very useful (fast, efficient and robust) for this special type of curve.
Dobkin et al. propose a different approach which is not based on approximating the curve, but on first approximating the function f by interpolation (for example, with Lagrange or Hermite interpolation or with splines) and then drawing that curve exactly. This approach results in **polygonal lines** [6].
However, the best-known continuation method is the so-called **predictor corrector path following** method, among others described by Bremer and Hughes [7]. In this algorithm, the continuation of the curve is performed by following the tangent for one given time step. After this, via projection along the gradient, the next sample point is corrected to be positioned on the curve. This process is illustrated in figure 1.5.

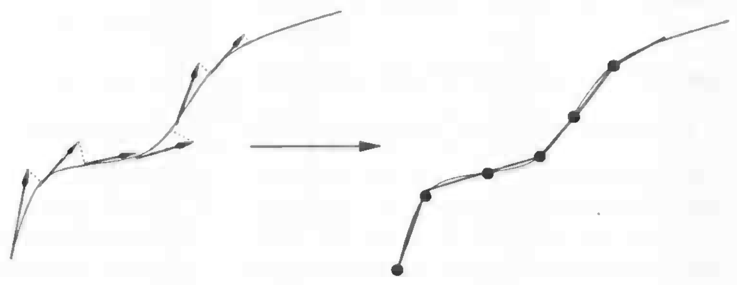

Figure 1.5: Illustration of the predictor corrector path following process.

Unfortunately, this group of methods has a few difficulties. Generally, the number of connected components of the curve is unknown beforehand. To display all these components, a seed point should be available on each one of them. But if the number of components is unknown, how can you make sure this is the case? Another difficulty can occur when the time step is taken too large. If the tangent is followed to compute the next sample point, this point can wrongly lie on a different component of the curve. This is illustrated in figure 1.6. The second group of approximation techniques tries to solve these problems.

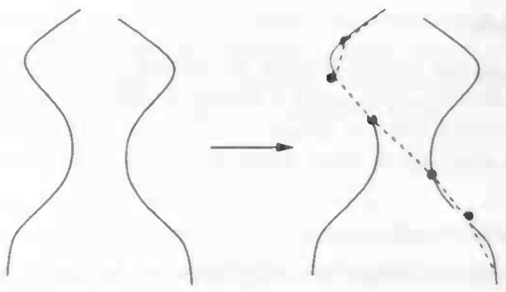

Figure 1.6: By taking the time step too big, the next sample point can wrongly lie on a different component of the curve.

1.3.2 Adaptive Enumeration Methods

The second group of approximation methods consists of the so-called adaptive enumeration methods, which are grid-based. The algorithm presented in this thesis belongs to this group. Adaptive enumeration methods generally perform three steps that are described in short here.
The first step is the generation of an initial mesh within some bounding box. Most of the methods in this group use quadtree meshes or triangulated meshes. For a description of these meshes, see chapters 2 and 3. The initial mesh generation is usually governed by a precondition about the mesh that should hold before the next step can be performed.
This next (and most important) step of the adaptive enumeration methods is the refinement of the mesh. This step is performed to remove ambiguities with respect to the approximation of the curve within each mesh cell. The refinement step is performed until in all grid cells, some criterion is satisfied. This criterion can be based on heuristics or a theoretical framework. However, it always tries to guarantee that in a cell satisfying this criterion, the curve's approximation is uniquely defined. In general, this results in a more detailed grid in the neighbourhoud of the curve, hence the *adaptive* in the group's name (see figure 1.7).

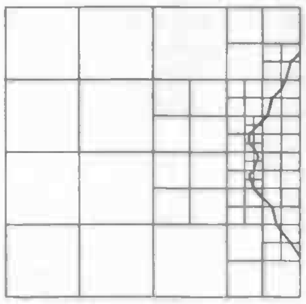

Figure 1.7: Illustration of the adaptive enumeration process. Near the curve, the grid is refined until some criterion is satisfied and the curve's approximation is defined uniquely.

Finally, when the grid is sufficiently refined and all cells satisfy the given criterion, the approximation of the implicit curve on this grid is determined. Often, this is a set of points or line segments, which together form a piecewise linear approximation of the curve. However, the approximation can also be a set of intersected grid cells that will be highlighted to display the curve. Either way, the result is a series or set of some sort, hence the *enumeration* part of the group's name. The main difference between the methods described here, is the refinement criterion.
The first adaptive enumeration method is proposed by Suffern [8]. The criterion he uses is based on heuristics that indicate when a curve's approximation within a grid cell is uniquely defined. Because

heuristics cannot guarantee anything about the approximation, Suffern developed another algorithm. Together with Fackerell, a new criterion was devised that uses **interval arithmetic** [9]. Affine arithmetic is a variant of interval arithmetic with which even tight interval bounds can be achieved [10]. The refinement criterion of this method is based on characteristics of the interval of the gradient value in a mesh cell.
Snyder proposed a very general **parametrizability** criterion, including an example in which he uses this criterion to approximate implicit curves [11]. The parametrizability criterion states that a grid cell should not be subdivided if the gradient of f can be parameterized in this cell in some direction. This implies that the gradient differs by at most 180 degrees and the curve is the graph of a function in this cell.
The adaptive enumeration methods have a few advantages. First of all, these methods are adaptive. They are more detailed near the curve, which is necessary to eliminate the problem in figure 1.6. Therefore, they give good results for more implicit curves than the continuation methods. Indeed, some of these methods can be proven to give topologically correct results, which of course depends upon the criterion that is used. This precision must unfortunately be paid for by computational inefficiency. In some of the mentioned methods, for example the one by Suffern and Fackerell, the interval-Newton method is applied to a great extent (maybe even four times per grid cell) which of course is very costly. Unfortunately, it is almost impossible to state anything about the complexity of any approximation algorithm, because these are seldom mentioned.

1.4 Topological Correctness

For some grid-based methods, topological correctness can be proven. This possibility depends on the refinement criterion that is used. We define a requirement that a mesh should satisfy after refinement with a given criterion, to be able to guarantee topological correctness of the approximation computed on this mesh:

Requirement 1 *For each element (cell) c of the mesh, the intersection $f^{-1}(0) \bigcap c$ of the curve with the cell is either empty or a topological segment (note that a point is a degenerate case of a topological segment).*

If requirement 1 holds and the intersection of a mesh cell with $f^{-1}(0)$ is non-empty, the straight line segment connecting the two points $f^{-1}(0) \bigcap \partial c$ is a topologically correct approximation of $f^{-1}(0) \bigcap c$.

Definition 2 *The* approximation of *$f^{-1}(0)$ is defined as the set of straight line segments that connect the points $f^{-1}(0) \bigcap \partial c$ of each cell c of the mesh.*

These line segments are ordered in such a way that they form a piecewise linear approximation.

Claim 1 *Requirement 1 implies topological correctness of the approximation of f.*

The proof of this claim uses the same techniques as [12]. Since it is completely standard, we omit further details. Concluding: if every mesh cell c has a topologically correct approximation in the straight line segment connecting the points $f^{-1}(0) \bigcap \partial c$, the entire mesh has one in the ordered set of these straight line segments.

1.5 Project Goals

In this project, we develop and implement an algorithm that guarantees to find all connected components of the implicit curve within a given bounding box. In addition, the resulting piecewise linear approximation of each component will be topologically correct. Thus, the algorithm guarantees to approximate the implicit curve correctly, but obviously only within the given bounding box.
This algorithm is an adaptive enumeration technique, it refines a mesh until a certain criterion is satisfied by all mesh elements. This criterion is based on the bad edge concept, which is described in detail in chapter 4. With this criterion, the need for any Newton root-finding method, especially the costly interval-Newton method, is eliminated.

The goal of this Master's research project is **to design and implement an adaptive enumeration algorithm to compute topologically correct, piecewise linear approximations of implicit curves**. We use both a quadtree mesh (squares) and a Delaunay triangulated mesh (triangles) and compare results. To realize this goal, we divide the algorithm into three steps:

1. Generate an initial mesh over the given bounding box;
2. Refine this mesh to remove ambiguities in the topology of the implicit curve's approximation;
3. Compute the intersection of the curve with the mesh, which satisfies requirement 1 of section 1.4, and display the polygon that is the result.

This algorithm is described in detail in chapter 5. The three steps are illustrated in figure 1.8, where the quadtree mesh is used. For the implementation of this algorithm, we use C++, the common computational geometry library CGAL [13] and the interval library `filib++` [14].

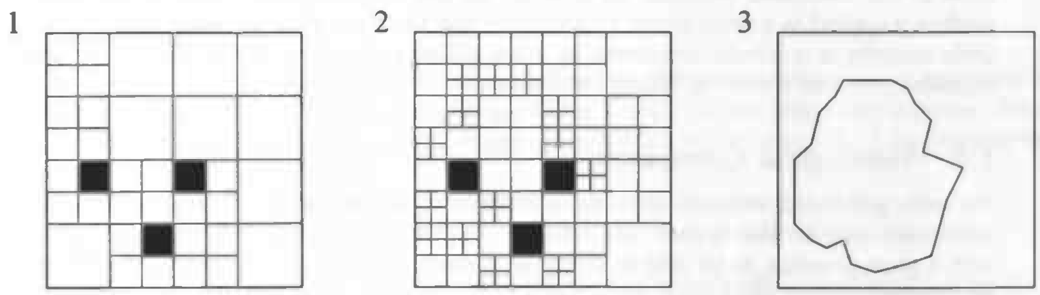

Figure 1.8: The three steps of polygonizing an implicit curve (this example illustrates the quadtree method).

In chapters 2 and 3, we look at the underlying theory of the quadtree mesh and Delaunay triangulated mesh used in this algorithm. Chapter 4 introduces the bad edge concept, which is the basis of this algorithm. We discuss the implementation of the three steps of the algorithm in chapter 5. In chapter 6, we discuss results and make a comparison of the two methods. Finally, chapter 7 discusses the functionality that could be added to the program in the future and the extension of the algorithm to 3D.

Chapter 2

Quadtrees and Refinement

To approximate an implicit curve, we need a mesh covering some user-defined bounding box. This mesh will be refined until topological correctness of the approximation is guaranteed, as described in requirement 1 of section 1.4. In this chapter, we discuss the first of the two meshes and the way it can be refined: quadtrees and quadtree refinement.

2.1 Quadtree

A *quadtree* (QT) is a geometrical subdivision of the plane into a hierarchical tree of squares which do not necessarily have the same size [15]. Each square is either a leaf of the tree, or an interior node. Interior nodes are split into four, hence *quad*tree, equal-sized children in the four compass directions north-west (NW), north-east (NE), south-west (SW), and south-east (SE).
A quadtree node has four neighbours in the four cartesian directions; a *neighbour* is a square of the same size, sharing a side. If the node is on the boundary of the quadtree, there is one or more direction in which it has no neighbour. If there is no square of the same size in some direction, the neighbour in that direction of the parent is the child's neighbour as well. A *corner* of a quadtree node is one of the four vertices of its square face. The horizontal and vertical sides that bound a face are called *edges*. A side can consist of multiple edges; this happens if the neighbouring node on that side is subdivided. When a quadtree is *balanced*, any side of a non-split leaf node has at most two edges. This balance restriction implies that every mesh cell differs at most a factor two in size with any of its four neighbours. This property is very useful for various operations or algorithms, for example neighbour finding.
The root of a quadtree mesh is some given bounding box. This root can be refined by subdivision, which means that each leaf node that does not satisfy a given refinement criterion (see section 1.3.2) is subdivided into four children. This subdivision of leaf nodes is repeated until all leaf nodes of the quadtree satisfy the given criterion. When a node is subdivided, the midpoint of the square is a new vertex of the subdivision. This midpoint is a corner of all newly created leaf nodes.

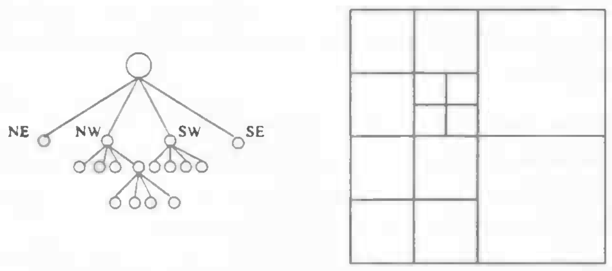

Figure 2.1: An example of an unbalanced quadtree and the corresponding subdivision. This tree is not balanced because the north-east child of the quadtree root is not split, but the south-east child of its western neighbour is.

The main advantage of quadtrees is their simple extension to higher dimensions, for example the octree structure in 3D. By extending squares to cubes, four children to eight, and four neighbours to six, quadtrees can be generalized to octrees. Whether algorithms using quadtrees (for example, point location or approximating implicit curves) can be generalized to 3D depends on the nature of that particular algorithm.
Another advantage of quadtrees is the ease of many operations, for example point location and refinement. Because of the symmetry of the squares in the subdivision and the axisymmetry of the edges, these operations are almost trivial to understand and implement.
The last two advantages of the quadtree are also important if the corresponding mesh is used for numerical simulations. The first is the possibility for a mesh to have cells of much varying sizes over small distances. This prevents unwanted large amounts of nodes on parts of the mesh were a detailed subdivision is not necessary. Of course, maintaining the balance restriction will not be beneficial to this favourable property, yet it does not completely eliminate this. Secondly, all angles in a quadtree mesh are 90°, which prevents numerical problems (like computing with very small angles) and thus enhances the quality of simulation results.

2.2 Data Structure

Unfortunately, the currently available stable release of the C++ computational geometry algorithms library CGAL does not contain a data structure for quadtrees. That's why we create one ourselves. First, we set up a list of requirements, then make a design for the structure. In section 5.1.1 we describe the implementation of this data structure.

2.2.1 Requirements

We formulate a list of requirements with respect to the implicit curve's approximation algorithm for a quadtree data structure. The need for these different requirements becomes clear in section 5.1, when the role of the quadtree mesh in our algorithm is explained in detail.

Requirement 1	A quadtree can be generated by recursively subdividing nodes until some criterion is satisfied by all leaf nodes
Requirement 2	The 2D data structure can easily be extended to a 3D octree data structure
Requirement 3	From the planar subdivision, given by the leafs of the quadtree, a Delaunay triangulation can be constructed
Requirement 4	Tree traversal is possible, easy and efficient
Requirement 5	The structure is easily passed on to CGAL to be displayed

Requirements 1, 3, 4 and 5 are considered in section 2.2.3. Requirements 2, 3, 4 and 5 will guide the design below.

2.2.2 Data Structure Design

We now design what the data structure for a quadtree mesh should look like, using the requirements from the previous section.

1. The root of the tree is the cell that represents the bounding box
2. Each internal node in the tree represents a subdivided cell of the planar subdivision
3. Each leaf of the tree is an undivided cell
4. Every node or leaf contains the following information:
    - Cartesian coordinates of the north-west (NW) corner point
    - Cartesian coordinates of the south-east (SE) corner point
    - Pointer to the parent cell (which is null for the root node)
    - Pointers to all 4 child cells (which are null for leaf nodes)
    - Compass direction of this node, as seen from its parent (NW, NE, SE or SW)

- Flag indicating whether this node is allowed to be subdivided
- Flag indicating whether this node is subdivided
- Level of this node in the tree

Pointers to neighbouring cells are not strictly necessary, because the lookup of neighbours can also be implemented by a routine that finds the desired neighbour only when it is actually asked for (see section 2.2.3). If the tree is balanced, neighbour finding is possible in constant time. This reduces the difficulty in keeping all pointers up-to-date when for example refining the tree.
The above setup can easily be extended to 3D (**requirement 2**) by replacing the four pointers to child cells by eight (plus the four cartesian directions to eight). The classification of tree nodes into root, internal node and leaf remains usable in 3D. By keeping the tree balanced and having the coordinates of the NW and SE corner as information of a node, a triangulation can be made easily (**requirement 3**). The same holds for the passing of the structure to CGAL (**requirement 5**), since a node can simply be displayed by four CGAL line segments. By keeping pointers to parent and children, traversal of the tree is possible (**requirement 4**), as well as neighbour searching

2.2.3 Operations Design

Besides the actual structure that contains the data which describes a quadtree, some operations on this structure should be available. To fulfill requirements 1, 3, 4 and 5, the following operations should be implemented:

1. Subdivide a leaf node into four child nodes, turning this node into an interior one.
2. Refine a quadtree mesh by subdividing leaf nodes until all leaf nodes satisfy a given criterion.
3. Get the neighbouring node in one of the four cardinal directions.
4. Get the square that bounds a leaf node.

Of course also the neighbours in the other cartesian directions than north (as in method 3) should be retrievable.
For **requirement 1**, the first two methods are needed. To be able to convert the quadtree mesh to a triangulation later (**requirement 3**), operation 2 should maintain the balance restriction of the quadtree. Operation 3 (together with parent and child pointers) gives opportunity for traversal (**requirement 4**). The last method is needed for displaying this quadtree node (**requirement 5**).

2.3 Refinement Method

As described in section 1.3, an adaptive enumeration technique refines a mesh to obtain an approximation of an implicit curve. As long as there is a mesh cell that does not satisfy the given refinement criterion, that cell is split into four child nodes. The refinement method of the quadtree mesh is based on the good/bad- qualification of edges mentioned in chapter 4. Section 4.2 describes the precise criterion that has to be satisfied by every mesh cell after refinement.

Refining a quadtree takes the following steps:

1 - List all leaf nodes of the current quadtree in list L

2 - While L is not empty, pop a leaf node and perform steps 3 & 4:

  3 - Check if the given criterion is satisfied by the current node

  4 - If not so, split node and add children to L whilst maintaining the balance restriction

All these steps are straightforward, except for step **3**. The details of this check are explained in section 5.1.2.

Chapter 3

Delaunay Triangulations and Refinement

The second type of mesh that is used to approximate implicit curves, is the Delaunay triangulation. In this chapter, we describe this type of mesh and the way it can be refined.

3.1 Delaunay Triangulation

A *triangulation* of a point set P is a planar subdivision of the convex hull of P into triangles with vertices in P. In other words, the triangulation $T(P)$ is a maximal planar straight line graph, where the *vertices* of the graph are the points of P. Each *face*, or *facet*, is a triangle. The three sides of a triangle are called *edges*. The *circumcircle* of a triangle is the unique smallest circle that passes through all three vertices of the triangle.
A triangulation is called a *Delaunay triangulation* (DT) if and only if the circumcircle of any triangle in the triangulation does not contain any vertex in its interior ([16], see figure 3.1).

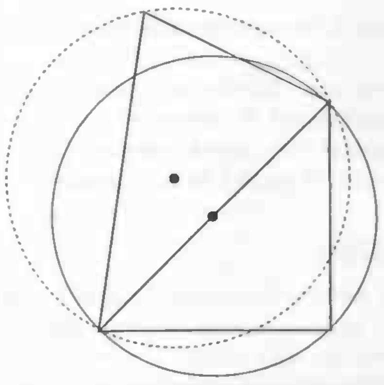

Figure 3.1: This triangulation, consisting of two triangles, satisfies the Delaunay property. Both triangles have a circumcircle that does not contain any other vertices.

A *constrained Delaunay triangulation* (CDT) is a triangulation where certain edges are *constrained*. These constrained edges (or *constraints*) are forced to be present in the triangulation, after which the triangulation tries to be 'as much Delaunay as possible'. Constraints are usually (part of) the input of a refinement algorithm, the so-called *Planar Straight Line Graph* (PSLG). As these constrained edges do not have to be Delaunay edges, the triangles of a constrained Delaunay triangulation do not necessarily fulfill the empty circumcircle property. However, they fulfill a weaker *constrained Delaunay property*. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is a CDT if and only if the circumcircle of any face encloses no vertex visible from the interior of the face [13].

Insertion of a new vertex in a CDT can lead to a situation called *encroachment*. This denotes that the vertex we try to insert lies in the circumcircle of a constrained edge. If this point were inserted, it would violate the constrained Delaunay property. What action has to be taken in such a situation to still insert a new vertex, is described in section 3.2.
To avoid numerical difficulties and approximation errors when a grid is used for numerical simulations, a constraint can be put on the minimal size of any angle of a triangle in the triangulation. Bearing this in mind, Delaunay triangulations have some favourable properties [17]. The most important property is that they maximize the minimum angle among all possible triangulations of that particular point set. They minimize the radius of the maximal circumcircle as well. Because of these properties, the Delaunay triangulation has been very popular in mesh generation for many years.
Triangulations can be generalized to higher dimensions, for example to their 3D counterparts tetrahedrizations. Unfortunately, not all algorithms using triangulations can. For example, the flipping algorithm, which creates a Delaunay triangulation from an arbitrary triangulation by flipping edges that do not locally satisfy the Delaunay property [18], is not applicable in 3D. It can get stuck in a local optimum (with respect to the minimum angle), which does not have to lead to a global optimum. Unlike the 2D-algorithm, in which a local optimum always leads to a global optimum.

3.2 Refinement Method

Delaunay triangulations can be refined to satisfy a given criterion in various ways. We describe an adapted version of Ruppert's refinement algorithm for 2D quality mesh generation [19] in short here. Ruppert's algorithm is probably the first theoretically guaranteed meshing algorithm to be truly satisfactory in practice [17]. It allows the density of triangles to vary quickly over short distances and the algorithm is accompanied by a theoretical framework with which Ruppert proves that the algorithm produces meshes that are both nicely graded (a small feature in one part of the mesh does not unreasonably reduce the edge lengths at places sufficiently far away) and size-optimal (the number of mesh elements is within a constant factor of optimal in size) [20].
Our variant of Ruppert's algorithm starts with an initial constrained Delaunay triangulation of input vertices and constraints. This initial triangulation is described in section 5.2.1. It refines this mesh by inserting additional vertices (also called Steiner points) until all triangles satisfy the given quality constraint. Normally, this quality constraint only involves the size of the angles of a triangle, but in this case it is extended to the good/bad- qualification introduced in chapter 4. Section 4.2 describes the precise criterion that has to be satisfied by the mesh after refinement.
The algorithm keeps a list of triangles that do not satisfy the refinement criterion. The algorithm handles triangles from this list until it is empty (while updating it along the way). Handling a triangle boils down to the insertion of a new vertex, which is performed as follows [21]:

- Insertion of the circumcenter of the triangle is attempted. This fails if the new vertex encroaches upon a constrained edge.
- If the above fails, the midpoint of the edge upon which the circumcenter encroached is inserted.

This process of either splitting a triangle or subdividing an edge is illustrated in figure 3.2

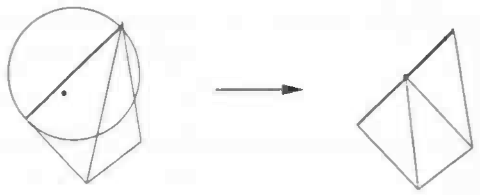

Figure 3.2: The circumcenter of the right triangle can not be inserted, because it encroaches upon the constrained edge (thick). Because of this encroachment, the midpoint of that edge is inserted as the new vertex.

Using this refinement method, the maximal size of the triangles will decrease and the minimal angle will increase. If the refinement criterion is constructed in such a way that the refinement will terminate if the minimal angle becomes large enough and the size of the triangles small enough, the algorithm will eventually terminate and all triangles will satisfy the given criterion. This is discussed in more detail in section 3.2. In figure 3.3, an example of the refinement of a triangulation is illustrated.

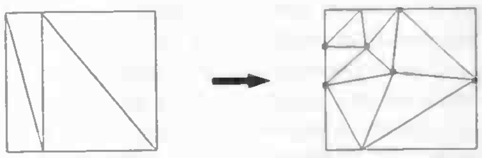

Figure 3.3: A triangulation is refined by triangle splitting or segment subdivision. The vertices that are inserted in this process are indicated by small squares.

3.3 Data Structure

Luckily, the C++ library CGAL already contains a data structure for a constrained Delaunay triangulation. We use a CDT instead of just a DT to force the bounding box of the mesh to be in the triangulation and thus keep the original geometry of the mesh. The data structure restores the constrained Delaunay property automatically when a new vertex is inserted. It does not check for encroachment, so we implement this ourselves (see section 5.2.2). The CDT data structure is defined in the header file `Constrained_Delaunay_triangulation_2.h`. Among others, operations are available for insertion and removal of vertices, insertion of constraints, queries about the triangulation, faces, or edges, and point location (the variable t has the CDT type `CGAL::Constrained_Delaunay_triangulation_2`):

```
1: Vertex_handle  t.insert (Point p)
2: void           t.remove(Vertex_handle v)
3: void           t.insert_constraint(Point a, Point b)
4: int            t.number_of_vertices ()
5: int            t.number_of_faces ()
6: bool           t.is_face (Vertex_handle v1,
                             Vertex_handle v2,
                             Vertex_handle v3)
7: bool           t.is_infinite ( Vertex_handle v)
8: bool           t.is_infinite ( Face_handle f)
9: bool           t.is_infinite ( Edge e)
10:Face_handle    t.locate (Point query)
```

These operations make our the refining algorithm a lot easier to implement. The implementation is described in detail in section 5.2.2.

Chapter 4

Bad Edge Concept

This chapter introduces the *bad edge concept*, which is the basis of the algorithms presented in this thesis. This concept guides the design of the precondition for mesh refinement and the criterion a mesh should satisfy after refinement to guarantee topological correctness of the approximation of the implicit curve over this mesh.

4.1 Bad Edges

The gradient of a C^2 function $f(x, y)$, $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$, is defined in every point of $\mathbb{R}^2$. If a point lies on the implicit curve, the gradient in this point is normal to the tangent of the curve in that point. By looking at the gradient $\bigtriangledown f$, an edge can be qualified as being good or bad ([21], *relint* is the relative interior of a line segment):

Definition 3 *An edge e of the mesh is called* bad *if there is a point p in relint(e) such that $\nabla f(p) \perp e$.*

An edge that is not bad, is called *good*. If an edge is good, the number of intersections it has with $f^{-1}(0)$ is either 0 or 1. A bad edge, on the other hand, can be intersected by $f^{-1}(0)$ an unknown number of times, because the gradient changes direction (possibly once or more). This observation is illustrated in figure 4.1. Because we do not want to use interval-Newton to find all intersections of an edge, we decide to compute intersections of an edge with $f^{-1}(0)$ only when this edge is good. The intersection of an implicit curve with a good edge can be approximated by linear interpolation, to even eliminate the use of the normal Newton method.

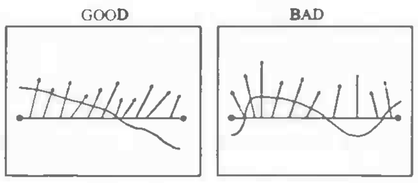

Figure 4.1: An edge is bad if the gradient of the function f at some point on the edge is perpendicular to it. This means that in that particular point, the isoline of f in that point is parallel to the edge, and thus the gradient may change its direction.

Using this definition, we also qualify a face of a planar subdivision to be good or bad:

Definition 4 *A face s is said to be* good *if one of the following conditions hold:*

1. *s does not intersect $f^{-1}(0)$*
2. *s intersects $f^{-1}(0)$, contains no singular point, and does not have two adjacent bad edges.*

This definition includes the condition 'contains no singular point' to prevent small components of the curve from lying entirely inside a face, see figure 4.2. Any closed component of an implicit curve has at least one singular point inside, hence this condition. Such a face should be refined until the face that contains this singular point no longer intersects the curve. This is the reason why we only approximate regular curves. Why a face cannot have two adjacent bad edges is explained in the proof of theorem 4.1

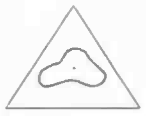

Figure 4.2: A face with an entire closed component of an implicit curve inside it.

A face is called *bad* if it is not good. To guarantee topological correctness of the approximation of an implicit curve over a certain mesh, requirement 1 from section 1.4 should be fulfilled by this mesh. We show how the bad edge concept can be incorporated, considering this requirement, in the refinement of triangulations.

Definition 5 *A bounded line segment is called* positive *(negative) if the value of f is positive (negative) at both endpoints.*

Theorem 1 *If a triangulated mesh has only good triangles, requirement 1 of section 1.4 is fulfilled and thus the approximation of an implicit curve $f^{-1}(0)$ from definition 2 is a topologically correct approximation.*

Proof In a triangulation with only good triangles, a face that intersects the implicit curve has at most one bad edge. We distinguish two cases:

- If a face has no bad edge at all, as illustrated at the left of figure 4.3, the intersection of this face with $f^{-1}(0)$ is either a point or a topological segment (which can be approximated by a straight line segment). Now, the intersection points of the curve with the edges uniquely define the way the curve passes through this face (definition 2). Within the face, the curve can be different from the straight line, but topologically, the approximation is correct [21].
- If a face has one bad edge (middle of figure 4.3), two faces meeting at the bad edge can be combined along this common bad edge to form a larger face (a quadrangle). Note that this larger face may be concave. This quadrangle has only good edges on the outside, and is a good face itself that has a uniquely defined (topologically correct) approximation of $f^{-1}(0)$. The proof of this statement is based on the fact that such a quadrangle can not have one positive and one negative diagonal [21]. For illustration of this idea, see figure 4.3.

□

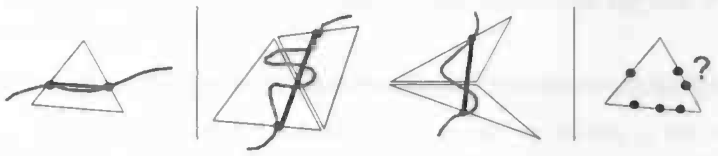

Figure 4.3: Four examples of possible situations for a triangle (from left to right: no bad edges, one bad edge (convex), one bad edge (concave), and two adjacent bad edges).

If a face would have two (or more) adjacent bad edges (at the right of figure 4.3), then these edges could all be intersected by $f^{-1}(0)$ an arbitrarily large number of times. The approximation of $f^{-1}(0)$ within this face is then not uniquely defined, because the correct way of connecting the (possibly many) intersections of the implicit curve with the edges is not known. Because the intersection of $f^{-1}(0)$ with this face is not guaranteed to be at most a single topological line segment, requirement 1 of section 1.4 would not hold.
Concluding, if we use a triangulation in which every face is good, the approximation of the implicit curve can be determined uniquely from definition 2.

4.2 Preconditions and Refinement Criteria

The purpose of the refinement step of the approximation algorithm has become clear: after refinement, the mesh should not have any bad faces in order to determine a topologically correct approximation of the implicit curve at hand. How do we accomplish this?

4.2.1 Delaunay triangulations

Let's consider Delaunay triangulated meshes first. The refinement method should remove all bad triangles from the mesh; triangles which intersect the curve, should not contain a singular point, or have two adjacent bad edges. We choose to do this in two steps. We first generate an initial mesh that isolates the singular points from the curve and then refine the mesh to remove all triangles with two or more adjacent bad edges. For reasons that become clear in section 5.2.2, we also remove all skinny triangles. A triangle is *skinny* if it has an angle below 20.2 degrees. In this way, the following precondition is put on the refinement step:

Precondition 1 *The initial Delaunay triangulation does not have any triangle that contains a singular point and intersects $f^{-1}(0)$ as well.*

The refinement criterion for the Delaunay triangulation method now can be stated:

Criterion 1 *The Delaunay triangulation must be refined until every triangle has all its angles above 20.2 °, and all triangles that intersect $f^{-1}(0)$ have at most one bad edge.*

4.2.2 Quadtrees

Because we defined the bad edge concept for triangles, a conversion step to create a triangulation is added in the quadtree method. To fulfill requirement 1 of section 1.4, this triangulation should have no bad or skinny triangles. Because the angles of a quadtree node are all 90° and the quadtree is balanced, skinny angles do not occur anyway (see section 5.1.2). Again, a precondition is put on the refinement step (when the mesh still is a quadtree).

Precondition 2 *The initial quadtree mesh does not have any square that contains a singular point and intersects $f^{-1}(0)$ as well.*

We define the *converted triangulation* as the result of conversion of the refined quadtree to a Delaunay triangulation. The refinement method for the quadtree mesh has to make sure that this converted triangulation does not contain any skinny and bad triangles.

Criterion 2 *The quadtree mesh must be refined until every triangle (of the converted triangulation) intersecting $f^{-1}(0)$ has at most one bad edge.*

The removal of bad or skinny faces is performed by face splitting. For quadtrees, this boils down to insertion of the midpoint of a node, resulting in four new child nodes. For Delaunay triangulations, face splitting is described in section 3.2.

4.3 Interval Arithmetic

The preconditions and criteria mentioned in the previous sections, describe some predicates for which a check should be available on edges and faces. These checks are (faces can be both triangles and squares):

1. Whether an edge is good or bad
2. Whether an edge intersects $f^{-1}(0)$
3. Whether a face contains a singular point
4. Whether a face intersects $f^{-1}(0)$

These four checks are implemented using interval arithmetic. An interval is a range of reals. For example, the intervals of x and y-coordinates over the bounding box of an edge or face are

$$\begin{aligned} I_x &= \{t \mid x_{min} \leq t \leq x_{max}\} \text{ and} \\ I_y &= \{t \mid y_{min} \leq t \leq y_{max}\}, \end{aligned}$$

where x_{min} is the smallest x-coordinate on of the bounding box (x_{max}, y_{min}, and y_{max} are analogous). Operations can be performed on these intervals. The operation is performed for all elements of the cartesian product and all outcomes are contained in the resulting interval. Addition, for example, of $[-2,2]$ and $[1,4]$ results in $[-1,6]$. Multiplication works analogously: $[-2,2] \times [1,4] = [-8,8]$. A special case of operation is the square of an interval, which always is positive: $[-5,5]^2 = [0,25]$. The interval of the function value of a function $f(x,y)$ over both the x and y interval is defined as

$$f(I_x, I_y) = \{f(s,t) \mid (s,t) \in I_x \times I_y\} = f(I_x \times I_y).$$

This is illustrated in figure 4.4 for a function f with only an x-parameter.

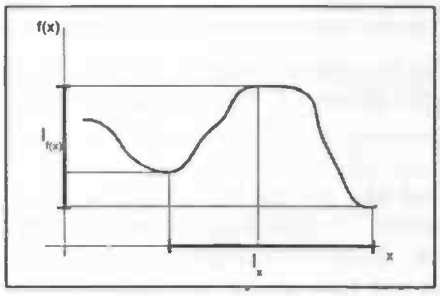

Figure 4.4: Given an interval of x-coordinates, the interval of the function value $f(x)$ can be computed.

All four needed interval checks examine whether zero is contained in one or two intervals. These intervals are the ranges of the value of a certain function over a given line segment or face. All checks can be implemented in the same fashion:

1 - Compute the interval of a function value over the bounding box of the line segment or face

2 - Check whether zero is contained in this interval. If it is not, return this as the result. If it is:

  3 - Split the box and do the check again for its subboxes

  4 - If the box is split some maximal number of times and the computed interval still contains zero, return that zero is contained in the interval

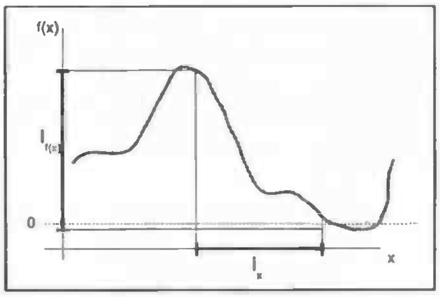

Figure 4.5: An interval can give a false positive for zero-containment when it has spacious bounds. In this figure, zero is contained in the computed interval $I_{f(x)}$, but not in the actual interval $f(I_x)$.

The subdivision of the bounding box if zero is contained in the computed interval is necessary because of possible spacious interval bounds. These spacious bounds can be the result of the implementation of the interval operations. This problem is illustrated in figure 4.5.

For the interval computations, we use the C++ library `filib++` [14]. This library provides a variety of interval arithmetic operations, from simple addition and substraction to the more complicated $arcsin$ and $sinh$, as well as set theoretic and relational operations. To get any use out of this library, we have to implement the interval variant of the function f and its first-order derivatives. It should be possible to use various test curves, therefore we introduce a function identifier `fid` that is a parameter to the routine that computes f and its derivatives. The implementation of all test curves' functions is placed in these routines and based on `fid`, the correct function is evaluated.

```
Interval fInterval(int fid,   Interval x, Interval y);
Interval fxInterval(int fid,  Interval x, Interval y);
Interval fyInterval(int fid,  Interval x, Interval y);
```

The second and fourth checks, whether an edge or face intersects the curve, both examine whether zero is contained in an interval of f over I_x and I_y. The third check essentially does the same for both f_x and f_y over the intervals I_x and I_y. The bad edge check is a bit different than the other three. It checks whether the gradient of this function is perpendicular to the given line segment. If two vectors are perpendicular, their dot product is zero. We define $\Delta x = x_b - x_a$, $\Delta y = y_b - y_a$,

$$\begin{aligned} I_{\Delta x} &= \{t \mid 0 \leq t \leq \Delta x\} \text{ , and} \\ I_{\Delta y} &= \{t \mid 0 \leq t \leq \Delta y\}. \end{aligned}$$

To check whether an edge e is bad, the dot product $I_{dot} = (\Delta x, \Delta y).(f_x(x,y), f_y(x,y))$ is computed over the bounding box of e. If the gradient and edge are perpendicular in some point on the edge, I_{dot} contains zero. The four different interval checks are listed in table 4.1.

The interval checks for f and its derivatives all compute the interval of the function value over a rectangular bounding box, also when the interval of f is needed merely over a line segment or a triangle. The line segment (or edge) checks differ from the face checks in the way the bounding box is subdivided and the check is made for the subboxes (because of the possible spacious bounds). For both checks, the bounding box is split into four equal subboxes, but for an edge, only two of these are checked (the two intersected by the line segment). In the face check, all four subboxes are checked. This results in the following methods:

Check	Interval Check(s)
Good/bad edge	$0 \in (I_{\Delta x}, I_{\Delta y}).(f_x(I_x, I_y), f_y(I_x, I_y))$
Intersected edge	$0 \in f(I_x, I_y)$
Singular point in face	$0 \in f_x(I_x, I_y)$ and $0 \in f_y(I_x, I_y)$
Intersected face	$0 \in f(I_x, I_y)$

Table 4.1: Four different checks on intervals.

```
1: bool segmentIsBad(int fid, Point p0, Point p1, int Nsteps);
2: bool segmentContainsZero(int fid, Point p0, Point p1, int Nsteps);
3: bool boxContainsSP(int fid, Point nw, Point se, int Nsteps);
4: bool boxContainsZero(int fid, Point nw, Point se, int Nsteps);
```

where `fid` is the function identifier, `p0` and `p1` are the endpoints of the checked line segment, `nw` and `se` are the north-west and south-east corner of the checked bounding box, and `Nsteps` is the maximal number of subdivisions of the bounding box. Note that for good edges, the method `segmentContainsZero` can immediately return true if the sign of f in the two end point is different, because f is monotonous in a good edge.

4.4 Linear Interpolation

After the mesh is refined, the approximation of the implicit curve can be determined from definition 2. We only compute the intersections of good edges with $f^{-1}(0)$, to avoid inefficient operations like interval-Newton. A good edge has the nice property that it only intersects $f^{-1}(0)$ once. Therefore, we do not compute this intersection exactly, but approximate it with linear interpolation. Note that the approximation then remains topologically correct, because there still is a homeomorphism between the intersection of the cell with $f^{-1}(0)$ and the approximation.
Linear interpolation uses the parameterization of an edge and the function value in its two vertices, to approximate the point on this edge where the function value is zero. The edge between the two vertices $p_a = (x_a, y_a)$ and $p_b = (x_b, y_b)$ can be parameterized in s as follows:

$$(x_s, y_s) = (1-s)(x_a, y_a) + s(x_b, y_b)$$

If on a good edge the sign of $f(p_a)$ and $f(p_b)$ is different, somewhere between $s_0 = 0$ and $s_1 = 1$, $f(s) = f(x_s, y_s)$ must be zero. The point where $f(s)$ is zero, the root s^*, can be approximated by:

$$s^* = \frac{-f(p_a)}{f(p_b) - f(p_a)}.$$

The approximation of an intersection of $^{-1}(0)$ with an edge is illustrated in figure 4.6.

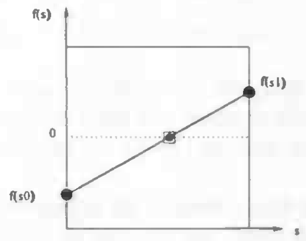

Figure 4.6: Approximation of the intersection point of the implicit curve with a good edge by linear interpolation.

Chapter 5

Approximating an Implicit Curve

In this chapter, the two algorithms for approximating implicit curves are described step by step. Using the four preconditions and criteria discussed in section 4.2, the three main steps of the algorithms are discussed: initial mesh generation, mesh refinement and curve approximation.

- As described in the section 4.2, an initial mesh must be constructed that isolates singular points from the curve to be approximated. This means that no grid cell of the initial mesh intersects the curve and contains one or more singular points at the same time. Note that this is possible because the implicit curves we work with are regular and thus are not self-intersecting.

- After the initial mesh is generated by isolating the singular points from the curve, we refine the mesh to remove all skinny and bad faces. In the end, the mesh will have a unique, topologically correct approximation of the implicit curve, because requirement 1 of section 1.4 is fulfilled.

- When all faces of the mesh are good, all topological ambiguities have been removed from the mesh and the polygonal approximation of the curve can be determined from definition 2. We discuss the implementation of computing the approximation for both (refined) meshes. Linear interpolation, described in section 4.4, is used for approximating intersections of good edges with $f^{-1}(0)$.

For illustration purposes, we show the (intermediate) results for the function f of equation (5.1), also displayed in figure 5.1 (with the `ImplicitPlot`-package of *Mathematica*). This curve has singular points in $(-\frac{1}{2}\sqrt{2},0)$, $(0,0)$ and $(\frac{1}{2}\sqrt{2},0)$ and a bounding box with north-west and south-east corner points $(-2,2)$ and $(2,-2)$ respectively, is used.

$$f(x,y) = x^2(1-x)(1+x) - y^2 + 0.01 = x^2 - x^4 - y^2 + 0.01 \tag{5.1}$$

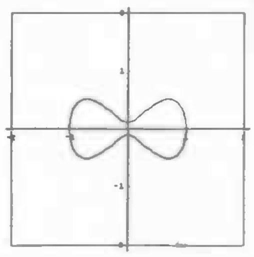

Figure 5.1: Function f inside its chosen bounding box.

5.1 Quadtree Method

5.1.1 Initial Mesh Generation

In the quadtree method, we generate a balanced quadtree as the initial mesh. The midpoints of leaf nodes in a quadtree are the only new vertices that can be inserted to refine the mesh. We start with an empty quadtree that is just a root node, with the corresponding square being the user-defined bounding box. Next, we isolate the singular points by recursively subdividing the quadtree nodes that contain at least one singular point an intersect the curve, using the interval checks mentioned in section 4.3. This subdivision is repeated until all quadtree nodes that contain one or more singular points, do not intersect the implicit curve $f^{-1}(0)$ and the singular points are thus separated from the curve. Meanwhile, we keep the quadtree balanced, so we do not need a separate method to do this afterwards.

Data structure

Before we can generate the initial quadtree mesh, we have to implement the data structure to save the quadtree in. This data structure is implemented in the file `quadtree.h`. The design of section 2.2.2 is followed closely. The header file of the class `Quadtree` is constructed as:

```
class Quadtree {
 private:
    Point nw,se;
    Quadtree * parent;
    Quadtree * child[4];
    int subdiv;
    int pardir;
    int consin;
    int level;
 public:
    ...
}
```

The flag indicating whether a node is allowed to be subdivided is `consin`, short for 'contains singular point'; faces in the initial mesh that contain at least one singular point but not intersect the curve, are already good and do not need to be subdivided any further (not even to keep the quadtree balanced). The data structure is now ready to be extended with some extra functionality (besides the obvious get and set methods).

Operations

The operations that have to be implemented are described in section 2.2.3. Most of these methods are very straightforward. The definition of these operations is placed in `quadtree.h` and the implementation in `quadtree.C`.

```
class Quadtree {
 private:
    ...
 public:
    ...
    Quadtree(Quadtree *pn, Point p1, Point p2, int pd, int l);
    ...
    void subDivide();
    ...
    void      isolateSingularPoints();
    ...
    Quadtree* getNorthNeighbour();
    ...
    CSegment  getNorthSegment();
    ...
};
```

For the isolation of the singular points in the method `isolateSingularPoints()`, the subdivision method `subDivide()` and interval checks `boxContainsSP()` and `boxContainsZero()` are used to subdivide the nodes that contain one or more singular points until these containing nodes do not intersect the implicit curve. The implementation of the balance restriction and the neighbour-finding method is taken from the algorithms in [16]. For displaying the quadnode in `CGAL`, the methods `getNorthSegment()` and its analogous routines are implemented. These operations are employed in the mesh generation method as follows:

```
generateInitialQTMesh(fid,xmin,xmax,ymin,ymax):

  INPUT - Function identifier fid and corner points of bounding box xmin, xmax, ymin, and ymax
  OUTPUT - Initial quadtree mesh (root) that isolates the singular points from the implicit curve
  1 - Quadtree root = createRootNode(xmin,xmax,xmin,xmax);
  2 - isolateSingularPoints(int fid):
        3 - Create a list L of all leaf nodes, and for every element of this list, compute bf0 = boxContainsZero
            and bsp = boxContainsSP
        4 - If b0 and bsp are both true, subdivide node and add new child nodes to L.
            If only bsp is true, set consin of this node to true.
        5 - If the node has been subdivided, restore the balance restriction.
```

Figure 5.2 displays the outcome of this method for the function f of equation (5.1):

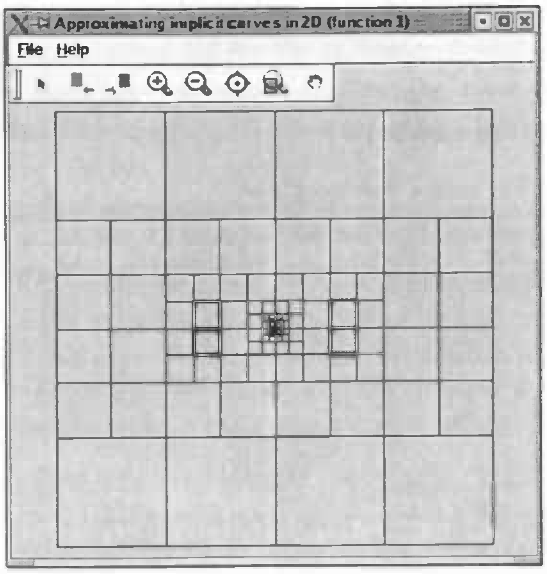

Figure 5.2: Initial quadtree that isolates the singular points $(-\frac{1}{2}\sqrt{2},0)$, $(0,0)$ and $(\frac{1}{2}\sqrt{2},0)$. The quadtree nodes containing a singular point are displayed with thick borders.

5.1.2 Mesh Refinement

In section 4 the refinement criterion for the quadtree mesh is defined; criterion 4.2.2 implies that a quadtree node is labelled *bad* if one of the triangles it consists of (in the converted triangulation) is bad. We now describe how this converted triangulation is constructed. The conversion of a quadtree node to triangles is based on which sides of a node, and thus which neighbours, are split. In a balanced quadtree, each side is split at most once. This leads to the five conversion situations shown in figure 5.3.

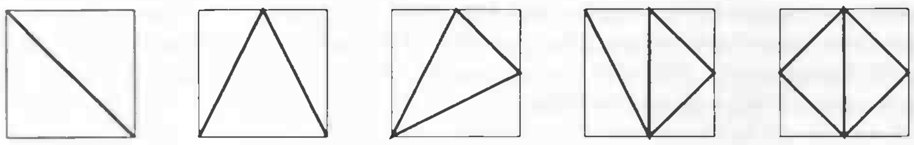
Figure 5.3: The five situations of converting a quadtree node to triangles.

Note that the minimal angle created in this way is 26.6°, which is automatically the minimal angle of the entire triangulation. Hence, there will be no skinny triangles in the resulting triangulation at all.
The quadtree refinement algorithm, described in section 2.3, can be implemented almost directly using criterion 4.2.2.
We need a data structure to keep a list of all quadtree nodes that still have to be checked for refinement. We choose a binary search tree (BST) because of the efficient average insertion, deletion and query running time. Every quadtree node has a unique identifying key, which is used for ordering in the binary search tree. The `refineTree()`-method in `quadtree.C` now is constructed:

```
refineQTMesh(fid,qt):
 INPUT - Integer fid identifying the function f and the initial quadtree qt
 OUTPUT - Refined quadtree mesh qt without bad faces
 1 - Create a binary search tree leaflist containing all leaf nodes of qt
 2 - while(! leaflist→isEmpty())
     3 - Pop next leaf node c from leaflist
     4 - Compute the interval of the function value fi over c
     5 - if ((!c→containsSing()) && (fi.contains(0)))
         6 - toBeSplit = check if node should be split to maintain balance restriction
         7 - if not toBeSplit, then toBeSplit = check, based on split edges, if the conversion
             of this node to a triangulation will contain a bad triangle
         8 - if toBeSplit then subdivide c and add its new children to leaflist
```

In figure 5.4, two meshes are shown: the initial quadtree created by `generateInitialQTMesh()` and a refined quadtree, which is the result of `refineQTMesh(fid)`. The quadtree is not symmetrical (as the function f of equation 5.1 is) because the refinement is based on the asymmetrical conversion of figure 5.3.

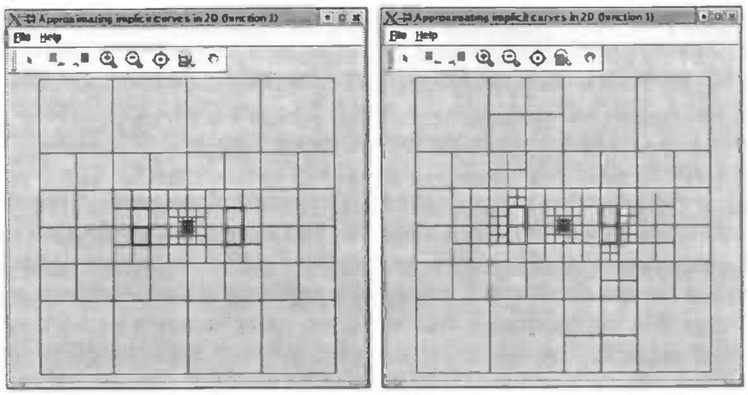

Figure 5.4: An example of an initial quadtree mesh (top left) that has been refined (top right).

5.1.3 Computing the Approximation

Before we start approximating the intersection of the implicit curve with the mesh, the quadtree is converted to a Delaunay triangulation. To do this, we create an empty constrained Delaunay triangulation and force all quadtree edges to be present in the resulting triangulation:

```
convertQTtoDT(qt):
  INPUT - Refined quadtree qt
  OUTPUT - Constrained Delaunay triangulation cdt
  1 - Initialize cdt as an empty constrained Delaunay triangulation
  2 - Create a list $L$ of all leaf nodes of the quadtree qt
  3 - For every element in $L$:
        4 - Insert the edges that the triangulation of this node has, according to figure 5.3 as constraints
            into cdt
```

The result of this conversion is a triangulation in which all triangles are good, see figure 5.5 for an example.

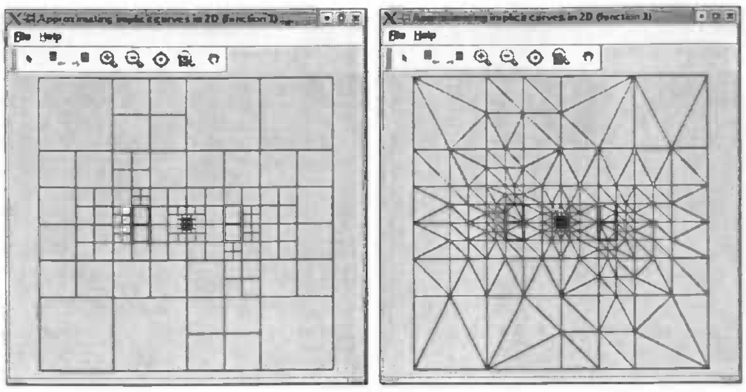

Figure 5.5: An example of a refined quadtree mesh (left) that has been converted to a constrained Delaunay triangulation. The squares containing a singular point are constrained and displayed with thick borders.

We now compute the intersection of $f^{-1}(0)$ with a triangulation with only good triangles. A good triangle that intersect the implicit curve $f^{-1}(0)$, has zero or one bad edges. A good triangle without a bad edge has an intersection with $f^{-1}(0)$ that is empty, a point or a segment. If a triangle has one bad edge, it shares this bad edge with the neighbour along that edge. Therefore, these two triangles together form a quadrangle which again has an intersection with $f^{-1}(0)$ that is empty, a point or a segment (see chapter 4).
We first look at the case when a triangle is on the boundary of the mesh. If this triangle has a bad edge on the boundary, the neighbouring face along this bad edge is the infinite face. Only the intersections of the two good edges of that particular triangle have to be checked. If there are two intersections, these points can be connected with a straight line segment. If there had been a neighbouring triangle on the bad edge side, the quadrangle that these two triangles would form intersected $f^{-1}(0)$ at most in a topological segment. Because this straight line segment already is contained in the first triangle, the neighbouring triangle would not be intersected by the curve (see the left of figure 5.6). If there is one intersection, we neglect it. It should have been connected with an intersection point on an edge of the neighbouring triangle sharing the bad edge. But because this neighbour is the infinite face and we do not want to compute the intersection of the curve with any bad edge, this one intersection is an end point of the approximation (see the right of figure 5.6).

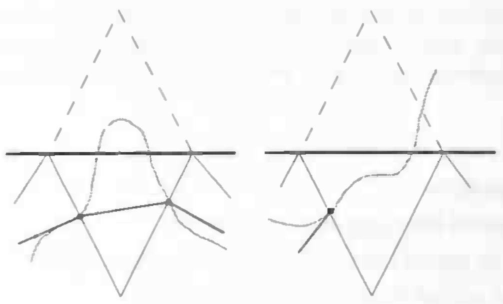
Figure 5.6: Approximation of the intersection of $f^{-1}(0)$ with a boundary mesh cell.

Now we look at the case when a triangle is in the interior of the triangulation. We just have to approximate the intersections of the good edges of each triangle with $f^{-1}(0)$, using linear interpolation. The action that has to be taken next, depends on the number of intersections that the edges of this face have with the implicit curve.

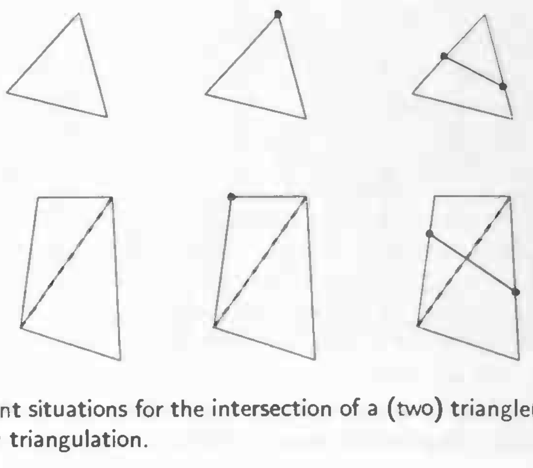
Figure 5.7: The six different situations for the intersection of a (two) triangle(s) with the implicit curve in a constrained Delaunay triangulation.

No more work has to be done when there are no intersections, see the left part of figure 5.7.
If there is one intersection, it can be ignored, because it can only be one of the corners of the triangle and that intersection will also be found in another triangle, see the middle part of figure 5.7. It is important to see that this intersection cannot be a tangent point of the curve in the interior of one of the edges. If an edge contains a tangent point of the curve, the gradient is perpendicular to that edge and thus the edge is bad. But if the edge is bad, we would not have been computing its intersection with the curve in the first place. Because a tangent point cannot occur if a single intersection is found, this has to be a corner point of the face.
If two intersections are found, these two can be connected, see the right part of figure 5.7. In figure 5.8 the approximation of function f of equation (5.1) is shown.

```
computeApproximation(fid,cdt):
  INPUT - Function identifier fid and constrained Delaunay triangulation cdt
  OUTPUT - Piecewise linear approximation $A$ of the implicit curve over the mesh cdt
  1 - Initialize two (empty) lists of straight line segments $L_s$ and $A$.
  2 - For every triangle in cdt:
      3 - If the bounding box of the triangle contains the function value zero:
          4 - If there is no bad edge, approximate the intersections of $f^{-1}(0)$ of the three good
              edges using linear interpolation. If two intersections ($p_0$ and $p_1$) are found, add the
              segment $(p_0, p_1)$ to $L_s$.
          5 - If there is a bad edge, approximate the intersections of $f^{-1}(0)$ of the two good edges
              of this triangle and the two edges of the neighbouring triangle along the bad edge
              using linear interpolation. If two intersections ($p_0$ and $p_1$) are found, add the segment
              $(p_0, p_1)$ to $L_s$.
  6 - Re-organize the line segments in $L_s$ to form a piecewise linear approximation, assign this sorted
      list to $A$ and display $A$ on the screen.
```

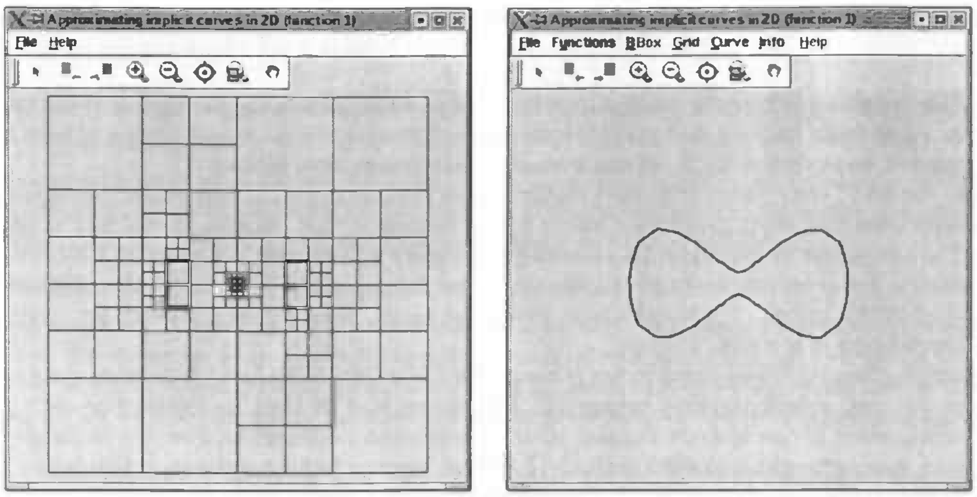
Approximating implicit curves in 2D (function 1)
File Help
Approximating implicit curves in 2D (function 1)
File Functions BBox Grid Curve Info Help

Figure 5.8: Example of a refined quadtree mesh and the computed approximation of an implicit curve, using this mesh. The squares containing a singular point are constrained and displayed with thick borders.

5.2 Delaunay Triangulation Method

5.2.1 Initial Mesh Generation

As mentioned in section 3.3, CGAL already has a constrained Delaunay triangulation data structure, which use in our implementation. We create an empty triangulation and insert the boundary segments of the bounding box as constrained edges. To prevent inserted points from lying outside the bounding box later, we create a nine times bigger bounding box (the original bounding box is copied eight times, surrounding itself) [22]. For clarity, only the original bounding box is displayed in figures throughout this thesis.
For the isolation of singular points, we create an imaginary rectangular mesh over the (original) bounding box. This rectangle can be split into two smaller rectangles, either in horizontal or vertical direction. This splitting is performed as long as a subrectangle both intersects the implicit curve and contains at least one singular point (these checks are performed with the interval checks from section 4.3). This process is illustrated in figure 5.9. The boundaries of all rectangles that (after subdivision) contain one or more singular points but do not intersect the implicit curve, are inserted as constraints in the triangulation. This ensures that these particular rectangles remain present in the triangulation, which prevents rearrangement of that area (analogous to the flag indicating whether a node is allowed to be refined in the quadtree method).

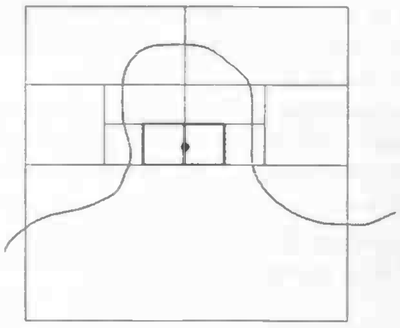

Figure 5.9: An imaginary rectangular mesh is recursively split until the rectangle in which (at least) one singular point lies, is no longer intersected by the implicit curve.

This imaginary rectangular mesh is stored in a list containing all rectangles that still need to be checked for subdivision, because they possibly both intersect the implicit curve and contain at least one singular point. Like in section 5.1.2, we use a binary search tree to store this list.
In the BST, every (internal or leaf) node contains the coordinates of two points, corresponding to the north-west and south-east corner points of the rectangle. The ordering in the BST is lexicographic. The comparison of two nodes (representing rectangles) is performed in two steps. First, the north-west corners are compared lexicographically (these are lexicographically 'smaller' than the corresponding south-east corners). If the north-west corners are equal, the south-east corners are compared, which are different in any case.
Initially, this BST only contains the rectangle that is the bounding box. As long as there are elements in the tree, a new rectangle is popped. This rectangle is checked for intersection with $f^{-1}(0)$ and containment of one or more singular point. If both checks evaluate to true, the rectangle is split and both new rectangles are added to the BST. If both evaluate to false, no more action has to be taken. If the rectangle contains at least one singular point but does not intersect the implicit curve, we achieved our goal: the singular point(s) is/are isolated from the curve in this rectangle.
The interval checking, and possible subdivision, is performed until the BST is empty. Eventually, the boundaries of all rectangles that do contain one or more singular point but do not intersect $f^{-1}(0)$, are inserted as constraints in the initial triangulation. The following algorithm is now constructed:

```
generateInitialCDTMesh(fid,xmin,xmax,ymin,ymax):
  INPUT - Function identifier fid, corner points of bounding box xmin, xmax, ymin and ymax
  OUTPUT - Initial constrained Delaunay mesh cdt that isolates the singular points from the implicit
      curve
  1 - Insert boundaries of the (nine times bigger and the original) bounding box as constrained edges
      of the triangulation dt to create a CDT
  2 - Initialize the list $L$ of rectangles to be subdivided with the single rectangle with coordinates
      (xmin,ymax) and (xmax,ymin)
  3 - while $L$ is not empty, repeat:
        4 - Pop a rectangle from $L$ and compute bf0 = boxContainsZero and bsp = boxContainsSP
        5 - If b0 and bsp are both true, subdivide rectangle and add two new rectangles to $L$. If only
            bsp is true, insert its boundary segments as constrained edges in cdt.
```

This gives the result of figure 5.10 for the same function f of equation (5.1) as was used in the quadtree examples.

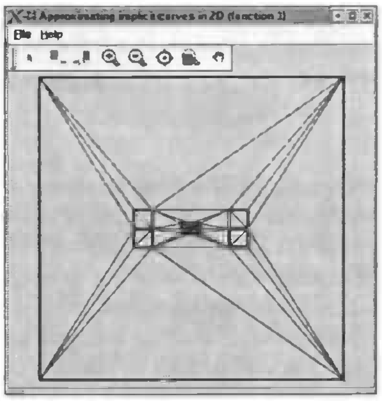

Figure 5.10: Initial triangulation that isolates the singular points $(-\frac{1}{2}\sqrt{2},0)$, $(0,0)$ and $(\frac{1}{2}\sqrt{2},0)$. The rectangles containing a singular point and the boundary of the original bounding box, are constrained edges and are displayed with thick borders.

5.2.2 Mesh Refinement

The implementation of the refining method for the Delaunay triangulated mesh follows the outlines of the algorithm described in chapter 3. First though, we discuss the termination of the algorithm and the implementation of the check for encroachment when a new vertex is inserted.

Termination

A triangle is *skinny* if it has an angle smaller than 20.2 degrees. By refining the mesh with Ruppert's algorithm, the minimum angle in the triangulation increases and in the end, no skinny triangles are left. This particular bound is necessary to ensure a circumradius-to-shortest edge ratio smaller than or equal to $\sqrt{2}$ for all triangles, which in turn is needed for termination of Ruppert's refinement algorithm [19].

When a triangle's area is very small and the angles between its edges are not too small, the gradient will not differ very much between the three edges. This implies that if the triangle is small enough and the angles are large enough, the gradient will not be perpendicular to two different edges. So, if our variant of Ruppert's refinement algorithm is performed, eventually no two adjacent edges will be bad anymore and the list of bad triangles will be empty. This is the reason for termination of our refinement algorithm [21].

Encroachment

In section 3.2, the encroachment principle was explained. A vertex, which is inserted in an existing triangulation, can encroach upon a constrained edge. Constrained edges are either boundary edges of the original, or nine times bigger, bounding box, or boundary edges of a rectangle that isolates one or more singular points from the implicit curve.
CGAL regrettably does not have an easy way to access all constrained edges of a triangulation. Therefore, to check for encroachment during the refinement method, we create a BST of constrained edges. The nodes in this BST all contain the coordinates of two points, representing the begin and end point of the constrained edge that is stored in that node. The same lexicographical ordering as in section 5.2.1 is used to compare nodes, although one adjustment is made. Because in this case there is no north-west or south-east points, the two points of one node are ordered lexicographically first, and then compared with the other points in that order. The function

```
handleInsertion(Delaunay &dt, BinarySTree* constr, TPoint tp)
```

performs insertion of a point `tp` in triangulation `dt`. The BST `constr` is the list containing all constrained edges of the triangulation as described above. This method handles the insertion according to the rules of section 3.2, dealing with encroachment. Thus, it tries to insert the point `tp`, but if it encroaches upon a constrained edge in `constr`, it subdivides this encroached edge into two subedges and `constr` is updated accordingly.

Algorithm

Because the refinement criterion consists of having no skinny as well as no bad triangles, the algorithm keeps two sets of triangles. We create two BST's, one of skinny triangles and one of bad ones.
In these two BST's, each node contains the coordinates of three points, corresponding to the triangle these three points form. The lexicographical ordering of section 5.2.1 can be extended and used for ordering nodes of three points as well. The algorithm handles triangles from the skinny list first, until that list is empty and then continues with the bad list, and updates both along the way. This update is done by checking all triangles that are incident to the new vertex (which is either the circumcircle of the split triangle or the midpoint of an encroached constrained edge). These incident faces are exactly those triangles of the triangulation that have been created when the vertex was inserted [22]. If such an incident triangle is skinny, it is inserted in the skinny-BST and if it is bad, in the bad-BST.

```
refineCDTMesh(fid,cdt,constr,xmin,xmax,ymin,ymax):
  INPUT - Integer fid identifying the function f, the constrained Delaunay mesh cdt, a set of
      constrained edges constr and the corner points of the bounding box xmin, xmax, ymin and
      ymax
  OUTPUT - Refined constrained Delaunay mesh cdt without bad or skinny faces
  1 - Create two binary search trees, bad and skinny, containing all faces (identified by its three
      vertices) that have to be refined
  2 - while either skinny or bad is not empty:
      3 - Get next three vertices (describing a face) from skinny, or it it is empty, from bad
      4 - If these vertices still form a face in the triangulation
          5 - Compute circumcenter cc of current face, do handleInsertion(cdt,bound,cc);
          6 - For all incident faces of the new vertex
              7 - Check if it is bad or skinny and add it to the corresponding binary search tree
```

In figure 5.11, an example of a result of `refineCDTMesh(...)` is shown, together with an initial mesh generated by `generateInitialCDTMesh(...)`.

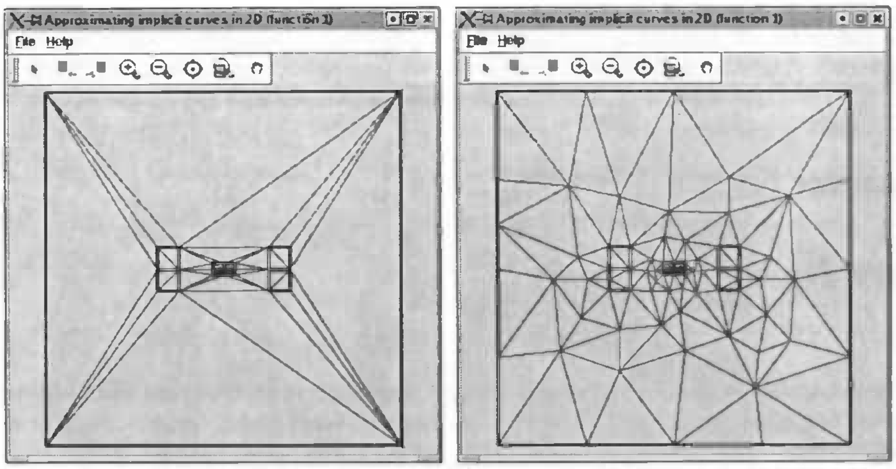

Figure 5.11: An example of an initial constrained Delaunay mesh (left) that has been refined (right). The rectangles containing a singular point and the boundary of the original bounding box are constrained edges and are displayed with thick borders.

5.2.3 Computing the Approximation

Essentially, the computation of the approximation in the Delaunay method is the same as in the quadtree method (after the conversion step), described in section 5.1.3. The result of the refinement step is again a triangulation in which all triangles are good. Thus, the approximation of $f^{-1}(0)$ is uniquely defined on this mesh. Therefore, the same approximation method can be used, with one tiny adjustment. In the quadtree method, a face that shares a bad (boundary) edge can be infinite. Because of the nine times bigger bounding box surrounding the original bounding box in the Delaunay method, this does not happen. Instead, a check is made whether the neighbouring triangle lies inside the original bounding box. If a neighbour lies outside the bounding box, it is treated in the same way as the infinite face is in the quadtree method. For an example result, see figure 5.12.

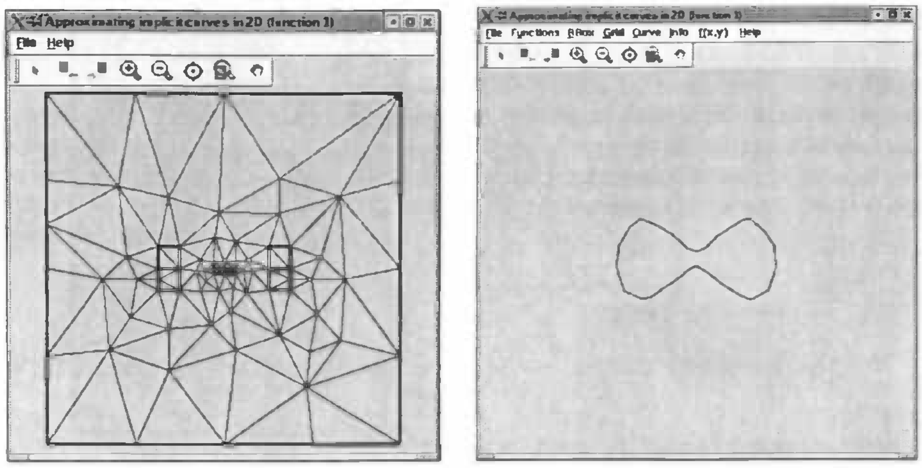
Approximating implicit curves in 2D (function 1)
File Help
File Functions BBox Grid Curve Info f(x,y) Help

Figure 5.12: Example of a refined Delaunay mesh and the computed approximation using this mesh. The rectangles containing a singular point and the boundary of the original bounding box, are constrained edges and displayed with thick borders.

Chapter 6

Results

In this chapter, we present the results of both the quadtree method and the Delaunay triangulation method, evaluate the results and compare the two methods based on several characteristics and benchmark results.

6.1 Quadtree Method

Below are some results for various implicit curves from a test set. On the left is the output *Mathematica* gave using the command `ImplicitPlot`, on the right is the output of our program. For more results of the quadtree method, see figure A-2 in appendix A.

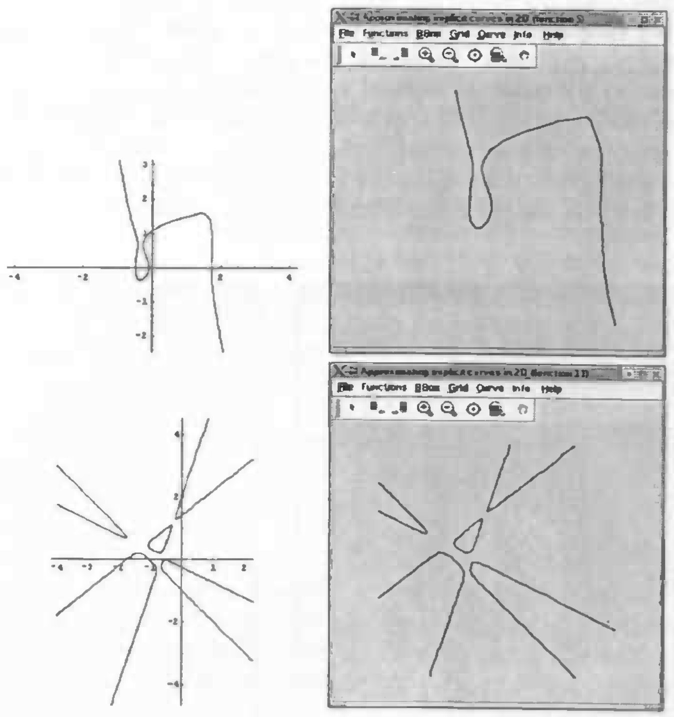

Figure 6.1: The results for two example implicit curves: $f(x,y) = x^4(1-x)^5 + x + y^2(1-y) + 0.1 = 0$ and $f(x,y) = -0.1 + (7+4x-5y)(-2-3x+y)(1+x+y)(0.5+x+2y) = 0$.

6.2 Delaunay Triangulation Method

Below are some examples of approximations, computed with the Delaunay method. For more results of the Delaunay method, see figure A-3 in appendix A.

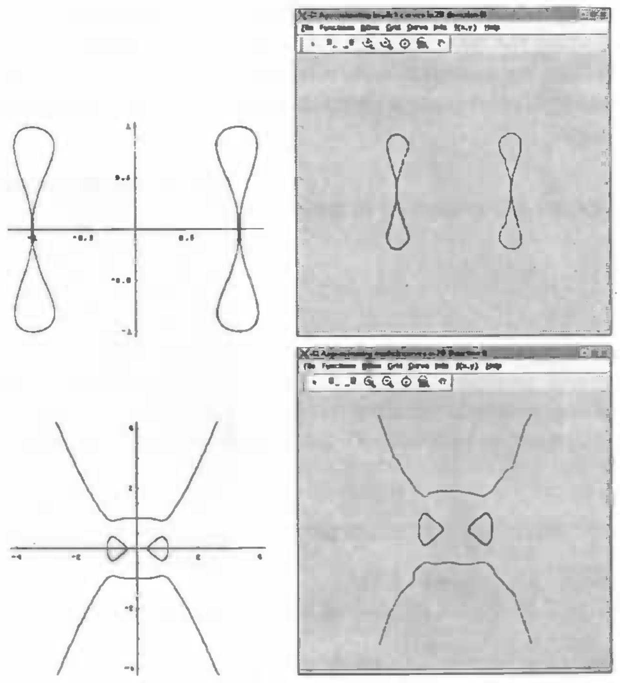

Figure 6.2: The results for two example implicit curves: $f(x,y) = ((1-x)(1+x))^2 - y^4(1-y)(1+y) - 0.0001 = 0$ and $f(x,y) = x^4(1-x)(1+x) - y^2(1-y)(1+y) - 0.01 = 0$.

6.3 Evaluation and Problems

If we compare the results of some input curves with the result *Mathematica* gave for the same curve, the difference that guaranteeing topological correctness makes, becomes clear. In the figure below, for two curves from the test set, number 10 and 12, the output of both the quadtree algorithm and *Mathematica* is shown (the result of the Delaunay method is topologically the same as the result of the quadtree method). For test curve 10, a small area of the Mathematica output is enlarged (see figure 6.3). In this area, the approximation of *Mathematica* is not topologically correct. Our two algorithms do not make this mistake.

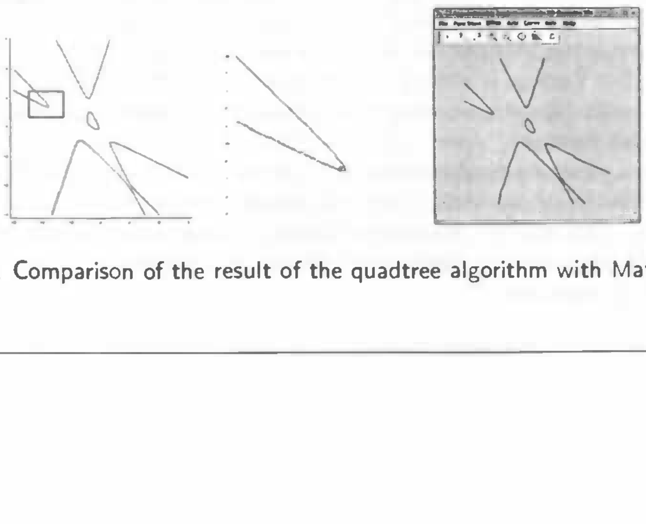

Figure 6.3: Comparison of the result of the quadtree algorithm with Mathematica.

A more severe error in the *Mathematica* approximation occurs with curve 12. During its computations, *Mathematica* gives the following warning:

```
Solve::"ifun": "Inverse functions are being used by Solve, so some solutions
may not be found."
```

As can be seen in figure 6.4, there is only one connected component in the output. When the same curve is displayed using the quadtree algorithm, the implicit curve proves to be quite different, which could be expected from the periodical nature of the equation $f(x,y) = cos(x)+sin(y)+0.01$. The only connected component *Mathematica*'s computations resulted in, is not even a connected component of the real implicit curve.

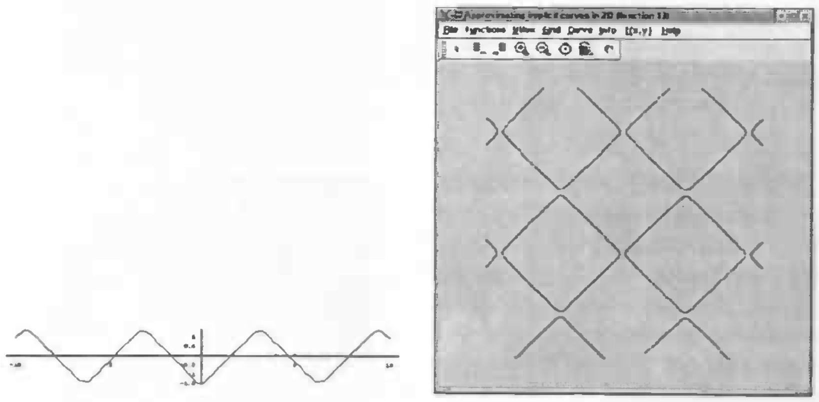

Figure 6.4: Comparison of the result of the quadtree algorithm with Mathematica.

So the results look rather good. Unfortunately, there are three implementation issues we would like to address here.
The first issue concerns the update of the two BST's (for skinny and bad triangles) that are maintained in the Delaunay refinement algorithm. When a new vertex is inserted in the (constrained Delaunay) triangulation, a number of new faces is created. In our implementation, we assume that all of these new faces are incident to the new vertex. But sometimes, a few skinny and bad triangles remain after the refinement step, which is an indication that our assumption may not be valid. We consider this highly improbable and suppose this is a yet undiscovered error in the implementation. We solved this issue for the time being by running the refinement repeatedly, until the number of vertices in the triangulation does not change anymore. This does not increase the execution time dramatically, because the number of bad and skinny triangles rapidly decreases.
Secondly, a similar problem occurs in the quadtree method. In the refinement algorithm, each quadtree node is either subdivided or remains undivided. This decision is based on whether or not one of the triangles it would consist of if it were converted to a CDT, is bad. This conversion to triangles depends on which of the sides of the node are split. A problem arises when a quadtree node is labelled to be good, therefore is not subdivided, but one of it's neighbours is subdivided later on in the refinement process. Because of the subdivision of its neighbour, the conversion to triangles changes and it could well be that one of the triangles the first node now will consist of is bad. This quadtree node should be subdivided, but because it already had its turn in the refinement process, it will remain undivided. Again, we solve this issue by running the refinement repeatedly, until the number of nodes in the quadtree remains the same.
Finally, for some particular combinations of a curve and a bounding box, the program crashes due to a `CGAL_precondition_violation`. This violation of a precondition occurs in an internal `CGAL` method and a long investigation of the constrained Delaunay triangulation data structure did not yield a proper way to solve this problem. By choosing a different bounding box, the implicit curve concerned can be displayed without problems.

6.4 Comparison of Methods

The results of the two methods are visually very similar. Of course, comparing the two methods on several other key points is interesting. We choose the following properties to compare the quadtree and Delaunay triangulation methods on:

1. Efficiency of computation (execution time)
2. Mesh size (number of mesh elements)
3. Simplicity of implementation
4. Extendability to 3D

Some benchmark results are shown first, which assist us in making the comparison on execution times and mesh sizes.

6.4.1 Benchmark Results

In table 6.1 below, for each of the twelve test curves from our test set (see figure A-1 in appendix A), the benchmark results are shown. For each curve, a suitable bounding box has been chosen and the mesh size and average execution time (in seconds) of both the quadtree and the Delaunay method are shown over ten executions of the entire algorithm. The number of singular points is computed with *Mathematica*.

Function	#Singular points	Bounding Box	Quadtree		Delaunay	
			#Elements	Ex. time	#Elements	Ex. time
1	3	9	258	0.0458	109	0.0964
2	4	9	312	0.0526	161	0.2352
3	2	400	217	0.0342	94	0.0985
4	1	9	64	0.0064	44	0.0267
5	4	16	1288	2.2140	542	5.4983
6	9	16	1088	0.2518	547	0.7589
7	3	16	1199	1.231	481	0.5362
8	9	36	2706	0.4786	1261	1.7468
9	3	1	1099	0.1988	406	0.4372
10	9	16	2389	0.9575	1172	3.2996
11	9	16	1841	0.7725	765	1.8857
12	6	36	77	0.0501	125	0.3830

Table 6.1: Benchmark results for eleven test curves.

For two bounding boxes, we visualize these benchmark results in a total of four charts, see figure 6.5. In these charts, the x-axis represents the test set of curves and the y-axis the execution time in seconds or the number of mesh elements respectively. Data for both the quadtree and Delaunay method are shown.

6.4.2 Running Times and Mesh Size

From these charts, we make the immediate observation that averagely, the mesh size with the quadtree mesh has a size that is three times as big as the Delaunay mesh. This was expected because of the higher degree of freedom in vertex insertion in the Delaunay method, which results in faster termination of the refinement algorithm. A new vertex can be inserted at the circumcenter of a triangle, as well as at the midpoint of an encroached edge, as opposed to just at the center of a quadtree node. Furthermore, the balance restriction is maintained in the quadtree, which results in more mesh elements than strictly necessary.

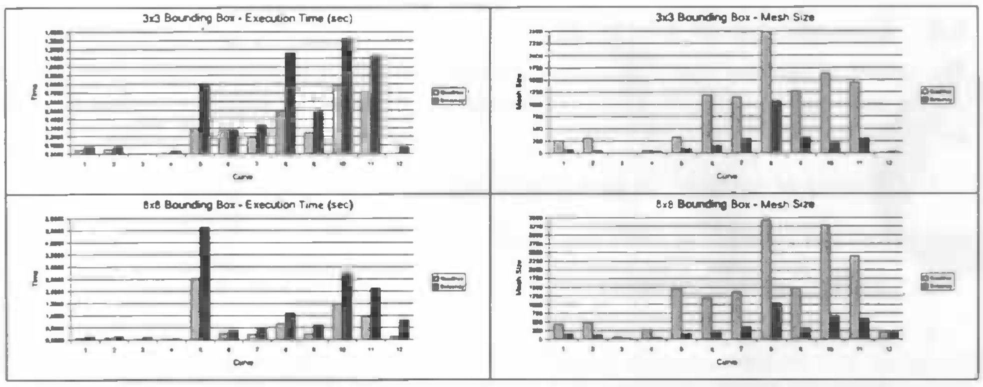

Figure 6.5: Charts of benchmark results for the 3x3 and 8x8 bounding boxes.

However, the quadtree method is averagely two times as fast as the Delaunay method. In the first place, this is due to the overhead in updating the constrained Delaunay triangulation when a vertex is inserted. Edge flipping is then performed to restore the constrained Delaunay property. Of course, this does not happen in the quadtree structure. But another implementation issue of the Delaunay method influences the higher execution time and is not a factor in the quadtree method.
We use a binary search tree that contains all constrained edges of the triangulation, which is used for the encroachment check. For every point that has to be inserted in the refinement process, the function `handleInsertion(..)` traverses this entire tree, which of course is very inefficient. A faster way to check for encroachment is therefore welcome. We discuss some thoughts on this.
The key to optimization is the fact that all constrained edges are axisymmetrical. A point that should be inserted can only encroach upon an edge that is intersected by either the horizontal or vertical line through this point. By restricting the encroachment check to be performed only on segments that cover the x or y-coordinate of the insertion point, the number of edges for which the circumcircle containment check is computed is significantly reduced. If this restriction is emitted from the program, the running time of the Delaunay algorithm aggravates with an average factor of 1.3 to 1.5, so it obviously is of reasonable influence on the execution time.

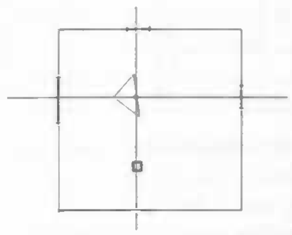

Figure 6.6: The circumcenter of the triangle the point for which insertion is attempted. This point can only encroach upon an edge that is intersected by either the horizontal or vertical line through this point.

The axisymmetry of the constrained edges can be used by keeping all constrained edges in an interval tree, described by [16], instead of in a binary search tree. This tree can retrieve all containing intervals for a certain point (in our case, horizontal or vertical line segments) with $O(n\ log\ n)$ building time and $O(log\ n)$ query time, using $O(n)$ storage. A binary search tree also uses $O(n)$ storage with $O(log\ n)$ query time, but finding the possibly encroached edges requires traversal of the entire tree, thus $O(n\ log\ n)$ time.

Also noteworthy is the long execution time of test curve 5. While the mesh that is used for approximating this curve does not have a considerably large number of elements, the execution time is substantially longer than with curves that produce meshes with about the same size. The equation describing this curve is

$$f(x,y) = x^4(1-x)^5 + x + y^2(1-y) + 0.1 = 0,$$

which becomes

$$f(x,y) = 0.1 + x + x^4 - 5x^5 + 10x^6 - 10x^7 + 5x^8 - x^9 + y^2 - y^3 = 0$$

after expansion. The long execution time can now be explained by the high powers of x in this equation. This equation is the only in the test set that has a ninth power of x or y in it and therefore the computation of f and its derivatives (especially in the interval checks) becomes very costly.
Another property of test curve 5 that can be of influence here, is the minimum distance of any singular point to the curve. Because the initial mesh is generated to isolate the singular points from the curve, this minimum distance greatly influences the execution time of the initial mesh generation. Furthermore, if this distance is very small, more tiny mesh cells are created in the neighbourhood of the curve than if the distance was bigger. The total amount of mesh elements may be equal, but the execution time can increase dramatically in the refinement step, because there are more cells intersected by $f^{-1}(0)$ and consequently, more mesh cells will be bad and more computation is needed. Because this is due to the nature of the bad edge concept, there is no real solution for this.

6.4.3 Implementation

The implementation of the quadtree method is simpler to produce and understand than the Delaunay, not in the last place because the latter uses the CGAL library extensively. In the implementation and understanding process, the programmer has to put a lot of research into the CGAL implementation of triangulations. However, there is no standard implementation available for the quadtree data structure. To produce the quadtree method, extra thought goes to designing and implementing this structure.
Because of the axisymmetry of the quadtree elements, a lot of geometric operations or queries are easy and efficient. The encroachment check and the computation of the circumcenter of a triangle in the Delaunay method both are much more complicated geometric primitives than any of those used in the quadtree method. The quadtree and Delaunay method have 2500 and 2100 lines of code, respectively, with 1000 lines of shared code.

6.4.4 Extendability to 3D

Both methods use a data structure that is extendable to three dimensions; quadtrees can be extended to octrees and a constrained Delaunay triangulation to a constrained Delaunay tetrahedrization (the term triangulation is also used in 3D). Examples of these structures are shown in figure 6.7. Mesh refinement is possible for both methods as well. Octree refinement inserts the midpoint of a cube, to subdivide it into eight new child nodes. A three-dimensional Delaunay refinement algorithm, based on Ruppert's algorithm in 2D, is proposed by Shewchuk [20] and an implementation is discussed in [23]. Like in 2D, mesh cells are split by inserting the circumcenter (the right of figure 6.7). There are some issues regarding this refinement. For example, slivers, tetrahedra with a low circumradius-to-shortest edge ratio, can have arbitrarily small dihedral angles. In the refinement process, they often are not split, because their small circumradii limit the likelihood that other vertices are inside their circumspheres [17]. However, respectable results have been achieved with 3D Delaunay refinement methods. Many researchers are working on this at the moment and we are positive that tetrahedral mesh refinement will be a common thing in the future.
Because the octree refinement would be based on whether the tetrahedra it consists of are bad, there should be a decomposition of a cube into tetrahedra available. Analogous to the decomposition of a square into triangles, this decomposition is based on whether the sides of the cube (square) are split. In figure 6.8 a possible decomposition is shown for the cases where zero, one or two neighbours are subdivided. For the first case, this is the Coxeter-Freudenthal decomposition [24],[25].

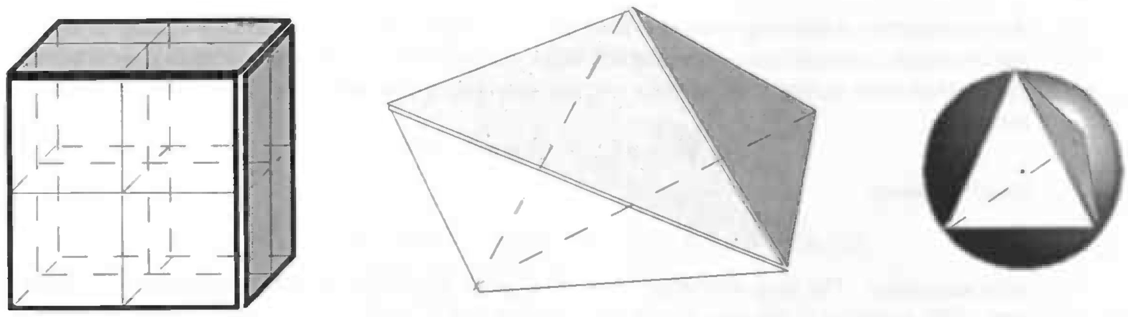

Figure 6.7: Left: an octree, middle: a tetrahedral mesh, right: the circumsphere and circumcenter of a tetrahedron.

In the other cases, one of the midpoints is connected with the four corners of the face opposite to it, and the midpoints of the split faces are all connected with each other. The faces of the cube are all triangulated into two triangles (if not split) or four (if split). The cases of three to six split faces are analogous, but rather complicated to visualize clearly in 2D.

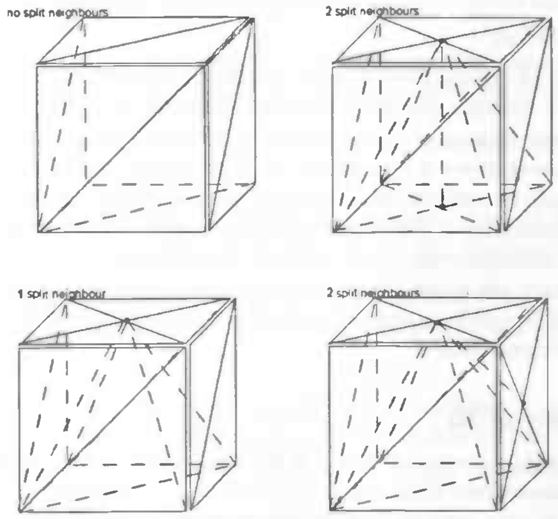

Figure 6.8: A possible decomposition of a cube, with zero, one, or two split neighbours, into tetrahedra. With zero or one neighbour, there is only one possible configuration. With two split neighbours, there are two, which by rotation and mirroring cover all possible configurations.

What we have accomplished so far, is that both algorithms have a tetrahedral mesh at its disposal, although in the octree case, this is an imaginary one (the tetrahedral mesh after conversion). This tetrahedral mesh has to fulfill the following requirement for topological correctness, which is analogous to requirement 1 of section 1.4 for the two-dimensional algorithm.

Requirement 2 *For each element (cell) c of the mesh, the intersection $f^{-1}(0) \bigcap c$ of the curve with the cell is either empty or a topological face (note that a point and a segment are a degenerate cases of a topological face).*

If a tetrahedral mesh satisfies requirement 2, the approximation of the implicit curve on this mesh can be defined:

Definition 6 *The* approximation of $f^{-1}(0)$ *is defined as the piecewise linear surface that connects the points* $f^{-1}(0) \bigcap \partial c$ *of each cell c of the mesh.*

The goal of the algorithm, for both methods, is to refine the 3D mesh until requirement 2 holds (in the octree case, after conversion to a tetrahedrization). The refinement criterion is again that a tetrahedron should be refined if it is skinny or bad. A cube should be subdivided if one of the tetrahedra it consists of is bad. The decomposition is chosen in a way to ensure that no resulting tetrahedron is skinny. When a tetrahedron is bad must still be defined, but has the same intuitive meaning as a bad triangle. To be able to define this precisely, the basis of the algorithm - the bad edge concept - has to be extended to 3D as well. We try to accomplish this in section 7.2.

Chapter 7

Conclusion and Future Work

7.1 Future Work

Our implementation now consists of six programs: for both the quadtree and the Delaunay method, there is a fully functional, interactive program, a demo program and a benchmarking program. The fully functional program allows the user to adjust two program parameters, describing the function f and the bounding box. Also, some visual parameters can be switched on or off; the grid, curve and the faces isolating the singular points can all be shown or hidden. The demo program performs the quadtree or Delaunay algorithm on a single function step by step, and the benchmarking program performs the algorithm many times (on different bounding boxes and for all test set curves) to compute average execution times.
Although these programs already give the user a lot of flexibility and the execution time is near optimal, it would be nice to add the following functionality to the program in the future.

1. Use of red-black trees or splay trees instead of binary search trees for optimal refinement running times;

2. Symbolic evaluation of the function f and its derivatives instead of hard coding them in the implementation, which does not allow for an easy way to add new test functions;

3. Extension of the combination of triangles along bad edges, which forms polygons (instead of just quadrangles), which have only good edges on the outside and do not contain any singular points (see figure 7.1).

Because the symbolic evaluation of f and its derivatives (and thus parsing of text input) is too elaborate for this thesis, but the current way of adding a curve is very unfriendly, we included a *Mathematica* notebook to orderly determine the code that should be added to the implementation for a new implicit curve.
Of course, the main extension we are interested in, is the approximation of implicit surfaces. We kept this extension to 3D in mind throughout this thesis. In the next section, we will investigate the possibilities to extend both algorithms to 3D, as announced in section 6.4.4.

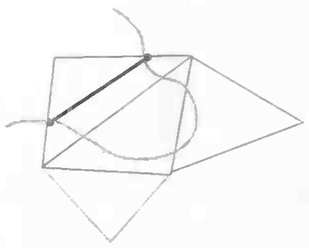

Figure 7.1: Combining more triangles around a bad triangle reduces the number of mesh elements

7.2 Extension to 3D

In section 6.4.4, we compared the quadtree and Delaunay methods on their extendability to 3D. The data structures and refinement algorithms for both methods proved to be extendable to 3D. We formulated requirement 2 in section 1.4, that has to be fulfilled by the mesh after refinement.
To be able to extend this approximation algorithm to 3D, we need to discuss the following issues:

- The bad edge concept: definitions of bad triangles, tetrahedra, and cubes
- Definition of the mesh refinement preconditions and criteria
- Definition of the curve's approximation after refinement

7.2.1 Bad Edge Concept

We start our extension attempt by defining bad edges and faces in the same way as in 2D. We are now using a C^2 function $f(x,y,z)$ with gradient $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$.

Definition 7 *An edge e of the mesh is called* bad *if there is a point p in relint(e) such that $\nabla f(p) \perp e$.*

The definition of a good face in 3D is exactly the same as in 2D: definition 4. Now we extend this definition to the third dimension:

Definition 8 *A volume v is said to be* good *if one of the following conditions hold:*

1. *v does not intersect $f^{-1}(0)$*
2. *v intersects $f^{-1}(0)$, contains no singular point, and does not have two adjacent bad faces.*

7.2.2 Preconditions and Refinement Criteria

The definitions from the previous section guide the preconditions and refinement criteria for the mesh refinement of both Delaunay tetrahedrizations and octrees. First, the Delaunay method is considered.

Precondition 3 *The initial Delaunay tetrahedral mesh does not have any tetrahedron that contains a singular point and intersects $f^{-1}(0)$ as well.*

The refinement criterion for the Delaunay tetrahedral mesh method now can be stated:

Criterion 3 *The Delaunay tetrahedral mesh must be refined until every tetrahedron intersecting $f^{-1}(0)$ has at most one bad face and no angles smaller than $20.2°$.*

The precondition and criterion for the octree can be adapted in the same way.

Precondition 4 *The initial octree mesh does not have any cube that contains a singular point and intersects $f^{-1}(0)$ as well.*

We define the *converted tetrahedral mesh* as the result of conversion of the octree to a Delaunay tetrahedral mesh after refinement. The refinement method for the octree mesh has to ensure that this converted tetrahedral mesh does not contain any skinny and bad tetrahedra.

Criterion 4 *The octree mesh must be refined until every tetrahedron (of the converted tetrahedral mesh) intersecting $f^{-1}(0)$ has at most one bad face and no angles below $20.2°$.*

7.2.3 Approximation of the Implicit Surface

With the preconditions and criteria for the mesh refinement available, we can construct a tetrahedral mesh in which all tetrahedra are good; they do not intersect the surface, or if they do, no singular point lies inside and they do not have two bad faces. On such a mesh, how can we define the approximation of $f^{-1}(0)$? This depends on its faces and edges being good or bad.
If the tetrahedron t has no bad faces or edges, this approximation is straightforward. Because of requirement 2 of section 1.4, the intersection of $f^{-1}(0)$ with t is either empty or a topological face. The approximation of the intersection of the implicit curve with t is then defined as the face with vertices $f^{-1}(0) \bigcap \partial t$ (see figure 7.2).

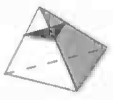

Figure 7.2: The approximation of an implicit surface within a tetrahedron in which all faces are good.

If t has a bad face (figure 7.3), it can be connected with its neighbour along that face (s) to form a larger volume with only good faces on the outside. The combined volume may be concave. The intersection of $f^{-1}(0)$ with this larger volume is then normally defined as the face with vertices $f^{-1}(0) \bigcap \partial(t \bigcap s)$. Note though, that not necessarily all edges on the outside of this volume are good and we do not compute the intersections with bad edges. Therefore, the approximation of the intersection of this tetrahedron with the implicit surface will not necessarily be a face that splits the tetrahedron in two, see the cross-section in figure 7.3. If several tetrahedra with this problem are adjacent and their approximations are joined, holes will appear in the surface.

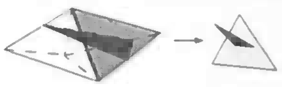

Figure 7.3: The approximation of an implicit surface within the union of two tetrahedra sharing a bad face (a cross-section of this face is shown as well).

If t has one or more bad edges, this face has to be connected with all the other tetrahedra sharing that bad edge, forming a larger volume with only good edges on the outside. In figure 7.4, this is illustrated by the union of three tetrahedra sharing an edge. Unfortunately, there can be arbitrarily many tetrahedra sharing the bad edge and the combination of these tetrahedra may become extremely complicated. Because we do not want to use interval Newton's method to compute the intersections of a bad edge with $f^{-1}(0)$, the combination of tetrahedra along bad edges is necessary. On the other hand, we want to keep this approximation algorithm as simple as possible. Therefore, we have to conclude that this leads to difficulties, and thus the bad edge concept, as proposed in this thesis, is not suitable for extension to 3D.

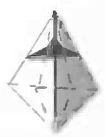

Figure 7.4: The approximation of an implicit surface within the union of three tetrahedra sharing a bad edge.

Of course, other criteria for mesh refinement, which are applicable in 3D, are conceivable. These could (and should) be the subject of future research.

References

[1] G. Turk and J.F. O'Brien. Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics (TOG)*, 21(4):855–873, 2002.

[2] J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[3] J.V. Aken and M. Novak. Curve-drawing algorithms for raster displays. *ACM Transactions on Graphics*, 2(147-169), 1985.

[4] D.S. Aron. Topologically reliable display of algebraic curves. *SIGGRAPH'83 Proceedings, Computer Graphics*, 17(3):219–227, 1983.

[5] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer Graphics and Applications*, 14(2):14–23, 1994.

[6] D.P. Dobkin et al. Contour tracing by piecewise linear approximation. *ACM Transactions on Graphics*, 9(4):389–423, 1990.

[7] D.J. Bremer and J.F. Hughes. Rapid approximate silhouette rendering of implicit surfaces. *Computer Vision, Graphics, and Image Processing*, 32:1–28, 1998.

[8] K.G. Suffern. Quadtree algorithm for contouring functions of two variables. *The Computer Journal*, 33(5):402–407, 1990.

[9] K.G. Suffern and E.D. Fackerell. Interval methods in computer graphics. *Computers & Graphics*, 15(3):331–340, 1991.

[10] A. de Cusatis Junior, L.H. de Figueiredo, and M. Gattass. Interval methods for ray casting implicit surfaces with affine arithmetic. *Proceedings of SIBGRAPI'99*, pages 65–71, 1999.

[11] J.M. Snyder. Interval analysis for computer graphics. *SIGGRAPH'92 Proceedings, Computer Graphics*, 26(2):121–130, 1992.

[12] H. Edelsbrunner and N. R. Shah. Triangulating topological spaces. *Int. J. on Comp. Geom.*, 7:365–378, 1997.

[13] Several universities in Europe & Israel. Computational Geometry Algorithms Library. *http://www.cgal.org*, 1998 - now.

[14] M. Lerch et al. Filib++ interval library. *http://www.math.uni-wuppertal.de/org/WRST/software/filib.html*, 2001.

[15] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *Proc. 31st Symp. Foundations of Computer Science*, 1:231–241, October 1990.

[16] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry - Algorithms and Applications.* Springer, second edition, 2000.

[17] J.R. Shewchuk. Lecture notes on Delaunay mesh generation. *University of Berkeley*, September 1999.

[18] R. Sibson. Locally equiangular triangulation. *The Computer Journal*, 21(2):65–70, 1992.

[19] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, 1995.

[20] J.R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. *14th Annual ACM Symposium on Computational Geometry*, pages 86–95, 1998.

[21] J.D. Boissonnat, D. Cohen-Steiner, and G. Vegter. Meshing implicit curves in the plane. Unpublished, December 2002.

[22] H. Edelsbrunner. *Geometry And Topology For Mesh Generation*. Cambridge Monographs on Applied & Computational Mathematics. Cambridge University Press, 2001.

[23] G.L. Miller, S.E. Pavy, and N.J. Walkington. Fully incremental 3D Delaunay refinement mesh generation. *Proceedings, 11th International Meshing Roundtable, Sandia National Laboratories*, pages 75–86, 2002.

[24] H. S. M. Coxeter. Discrete groups generated by reflections. *Ann. of Math.*, 6:13–29, 1934.

[25] H. Freudenthal. Simplizialzerlegungen von beschränkter flachheit. *Ann. of Math.*, 43:580–582, 1942.

Appendix A

Result Figures

1	$f(x,y) = x^2(1-x)(1+x) - y^2 + 0.01$
2	$f(x,y) = x(1-x)(1+x) + y(1-y)(1+y) + 0.1$
3	$f(x,y) = x^2 + 3xy + y^3 - 25$
4	$f(x,y) = x^2 + y^2 - 1$
5	$f(x,y) = x^4(1-x)^5 + x + y^2(1-y) + 0.1$
6	$f(x,y) = x^4(1-x)(1+x) - y^2(1-y)(1+y) - 0.01$
7	$f(x,y) = x^6(1-x)(1+x) - y^2 - 0.0001$
8	$f(x,y) = ((1-x)(1+x))^2 - y^4(1-y)(1+y) - 0.0001$
9	$f(x,y) = x^2 - xy + y^4 + 0.0001$
10	$f(x,y) = -0.1 + (-2-3x+y)(1+x+y)(2x+y)(0.5+x+2y)$
11	$f(x,y) = -0.1 + (7+4x-5y)(-2-3x+y)(1+x+y)(0.5+x+2y)$
12	$f(x,y) = cos(x) + sin(y) + 0.01$

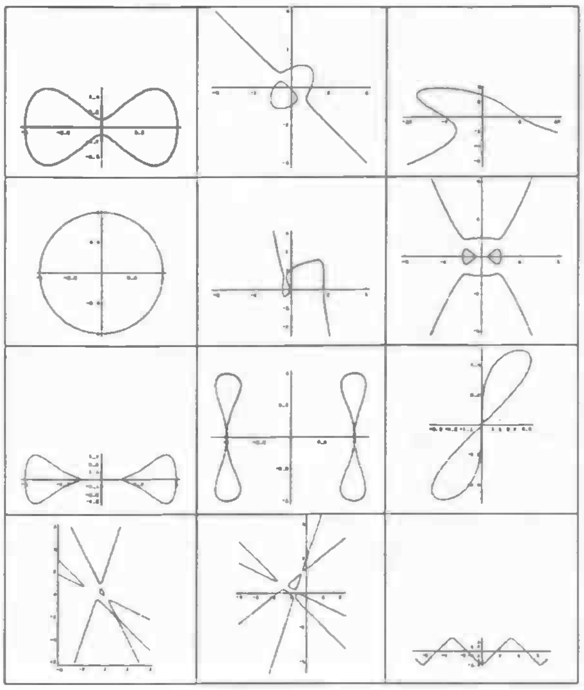

Figure A-1: The test set of functions of which the implicit curve, corresponding with $f(x,y) = 0$, will be approximated.

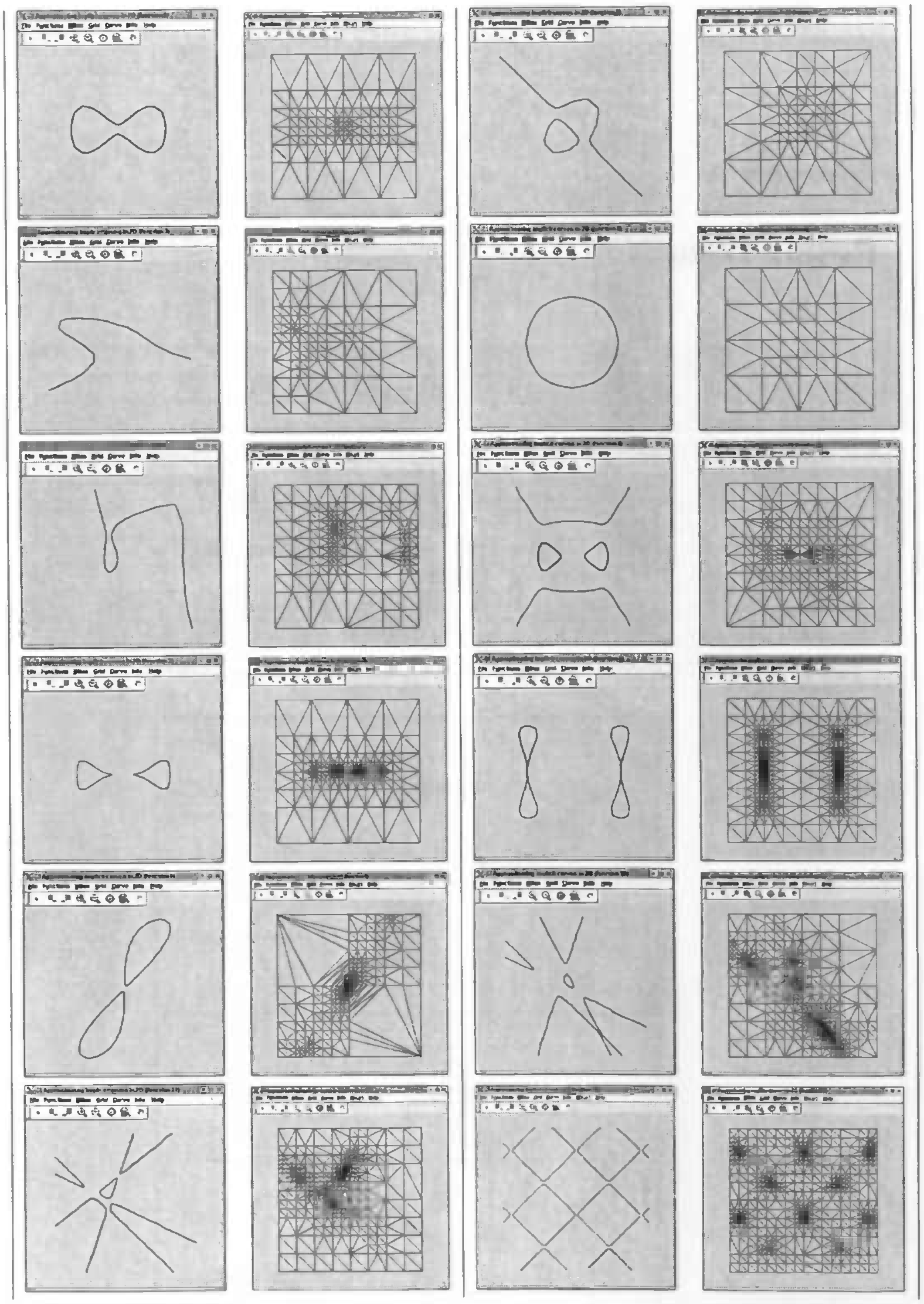

Figure A-2: Results of the quadtree method

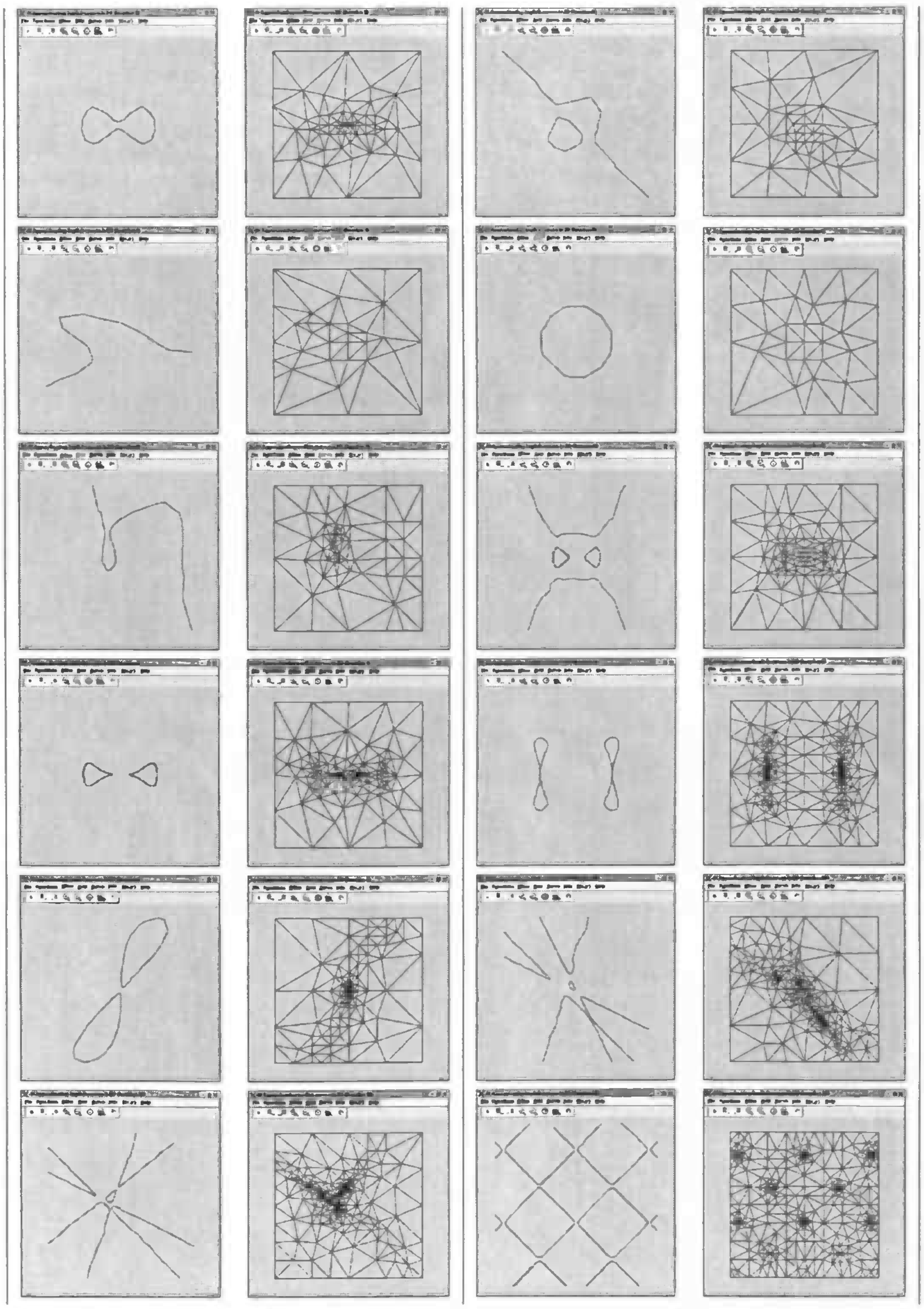

Figure A-3: Results of the Delaunay method