Combinatorial Curve Reconstruction and the Efficient Exact Implementation of Geometric Algorithms

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Stefan Funke

Saarbrücken
31. Juli 2001

Datum des Kolloqiums: 30. Juli 2001

Dekan der naturwissenschaftlich-technischen Fakultät I:
Professor Dr. Rainer Schulze-Pillot-Ziemen

Gutachter:
Professor Dr. Kurt Mehlhorn, MPI für Informatik, Saarbrücken
Professor Dr. Stefan Näher, Universität Trier

Abstract

This thesis has two main parts. The first part deals with the problem of *curve reconstruction*. Given a finite sample set S from an unknown collection of curves Γ, the task is to compute the graph G(S,Γ) which has vertex set S and an edge between exactly those pairs of vertices that are adjacent on some curve in Γ. We present a purely combinatorial algorithm that solves the curve reconstruction problem in polynomial time. It is the first algorithm which provably handles collections of curves with corners and endpoints.

In the second part of this thesis, we will be concerned with the exact and efficient implementation of geometric algorithms. First, we develop a generalized filtering scheme to speed-up exact geometric computation and then discuss the design of an object-oriented kernel for geometric computation.

Kurzzusammenfassung

Diese Dissertation besteht aus zwei Teilen. Der erste Teil beschäftigt sich mit dem Problem der Kurvenrekonstruktion. Gegeben eine endliche Menge von Stichprobenpunkten S von einer Menge von unbekannten Kurven Γ, besteht die Aufgabe darin, den Graphen G(S,Γ) zu konstruieren, welcher die Knotenmenge S und Kanten zwischen genau den Knotenpaaren besitzt, welche auf einer der Kurven in Γ adjazent sind. Wir präsentieren einen rein kombinatorischen Algorithmus, der das Kurvenrekonstruktionsproblem in polynomieller Zeit löst. Es ist der erste Algorithmus, der beweisbar Mengen von Kurven rekonstruieren kann, wenn diese auch Ecken und Endpunkte beinhalten dürfen.

Der zweite Teil dieser Dissertation handelt von der exakten und effizienten Implementierung von Geometrischen Algorithmen. Wir entwickeln zunächst ein generalisiertes Filterschema, um exakte geometrische Berechnungen zu beschleunigen, und entwerfen dann das Design eines objektorientierten Kernels für geometrische Berechnungen.

Acknowledgments

This thesis couldn't have been written without the help of many people. First of all, I thank my advisor Prof. Dr. Kurt Mehlhorn. His guidance, advice, and encouragement have been invaluable throughout my master's and doctoral studies.

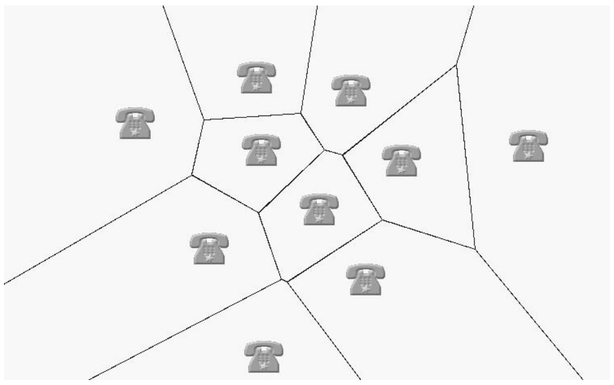
During the last three years I had many fruitful discussions with almost every member of our group. In particular, I am grateful to my coadvisor Edgar Ramos with whom I have spent hours discussing our work.

In addition, I am grateful to Ernst Althaus, Christoph Burnikel, Piotr Krysta, Stefan Näher, Stefan Schirra, Michael Seel, and Mark Ziegelmann for many interesting discussions (not only) concerning research related topics.

Contents

Preface

In this thesis we will present results in two areas which belong to the broad field of computational geometry.



Concepts of computational geometry concepts can be found in many places of everyday life. For example, imagine being in a big city and you are looking for a post office. There are several post offices in the city, but naturally you would prefer to know which one is closest to your current position. A city map which shows the exact position of each post office would be helpful, but still, there might be several post offices which are similarly close to your current position. To find exactly the post office which is closest, it would be nice if the map was subdivided into areas which have one particular post office as closest, like in the figure above. Now the question is, how do these areas look like and how can one compute them ?

Even though you probably do not really care if you go to a post office which is not the closest but almost as close, this problem describes a fundamental geometric concept, which has many applications. The subdivision of the map is a so-called Voronoi diagram. It can be used to model areas of influence of radio transmitters, guide robots and, even to describe and simulate the growth of crystals. There are many more geometric concepts which can be found in everyday life.

In the first part of this thesis we will theoretically investigate a problem from computational geometry, whereas the second part will be more concerned with some practical aspects of implementing geometric algorithms.

Combinatorial Curve Reconstruction

Given a set S of sample points from a collection of open and closed curves Γ, curve reconstruction is the problem of computing the graph $G(S, \Gamma)$, called the *correct reconstruction*, whose vertex set is S and which has an edge between two samples if and only if they are adjacent on a curve in Γ.

Obviously, it is not possible to correctly reconstruct a given curve from an arbitrary sample set from it. Therefore, some restrictions on the sample set are needed. Several Algorithms have been proposed that provably reconstruct certain classes of curves. In the first part of this thesis we present the first algorithm which provably reconstructs a collection of curves with corners and endpoints. This result has been published at the 12th Annual ACM-SIAM Symposium on Discrete Algorithms 2001 ([FR01]).

The Efficient Exact Implementation of Geometric Algorithms

When computer scientists design geometric algorithms, they usually assume the availability of exact arithmetic on real numbers (like we do for example in the first part of this thesis). Since no computer provides exact arithmetic on real numbers in hardware, programmers implementing these algorithms must find some substitution. Quite commonly, they resort to floating-point arithmetic due to its support by hard- and software. The resulting programs may not behave as expected, though. The roundoff errors accumulating during floating-point computation very often make the programs crash or produce inconsistent results. There are several ways to overcome this problem, first, one can design algorithms which explicitely deal with the problem of roundoff errors. Only very few such *robust* algorithms exist, mainly for very simple problems. The other possibility is the so-called *exact computation paradigm*, which advocates to give the implementer of a geometric algorithm the illusion of exact arithmetic on real numbers by providing exact number types and exact geometric predicates. Unfortunately, providing this 'illusion' has its cost which is usually considerably higher than pure floating-point arithmetic.

In the second part of this thesis, we will present two filtering concepts which reduce the overhead compared to floating-point arithmetic when adhering to the exact computation paradigm. These results have been published at the 11th Canadian Conference on Computational Geometry 1999 ([FMN99]) and the 16th Annual ACM Symposium on Computational Geometry 2000 ([FM00]).

Part I

Combinatorial Curve Reconstruction

Chapter 1

Reconstructing a Collection of Curves with Corners and Endpoints

1.1 Introduction

The problem of reconstructing a shape from a given finite set of points has attracted much attention in the literature during the last twenty years. Its importance arises from a wide area of applications, mainly in reverse engineering. For the most important problem, the reconstruction of a surface in the Euclidean space, many algorithms have been proposed that produce good approximations of the surfaces. The drawback of these algorithms is that they provide no guarantee for the correctness of the returned solution. Recently, these reconstruction problems have been investigated from a theoretical point of view. The results are algorithms that provably solve the reconstruction problem for a certain class of shapes. In the following we will give a formal specification of this problem, summarize our contribution to this problem and give an outline of the contents of this chapter.

Problem Description

In general a *shape* Γ is defined as a subset of the Euclidean space $\mathbf{R}^d$. Given a finite set of points S from an unknown shape Γ, called the *sample points*, the *shape reconstruction problem* asks for a shape $\Gamma' \supset S$ that approximates Γ.

We will only be concerned with reconstruction algorithms with *guaranteed performance*, i.e., algorithms that *provably* solve the reconstruction problem under certain assumptions on the shape Γ and the sample set S. Of course, we also need a notion of what the *correct reconstruction* is. In the case where the shape is a *curve*, i.e., a one–manifold, the correct solution can easily be defined as follows:

Definition 1 *The correct reconstruction, also called* polygonal reconstruction, *$G(S,\Gamma)$ of a sample set S with respect to a collection of non-intersecting curves Γ is defined as the graph G with vertex set S and an edge between two vertices if and only if they are adjacent in Γ.*

Already in the case of a *surface* in $\mathbf{R}^3$, i.e., a two–manifold, the definition of the correct reconstruction takes some deliberation. A possible answer was given by Amenta and

Bern [AB99].

While the problem of curve and surface reconstruction has been looked at by researchers in the computer vision and computer graphics community, only recently researchers in the area of computational geometry have started to work on the problem more intensely. They have proposed algorithms that provably solve the problem for certain classes of curves or surfaces, if they are appropriately sampled. Most of these algorithms required the shapes to be closed and smooth. Only for the case where the shape is a curve, algorithms are known which allow the shape to be open or non-smooth, i.e. differentiable in all but a finite number of points. But still, the problem of reconstructing a collection of possibly open *and* non-smooth curves has been open so far. For the reconstruction of surfaces in $\mathbf{R}^3$, current algorithms can only deal with smooth, closed surfaces; they are mostly based on the same ideas as the curve reconstruction algorithms in $\mathbf{R}^2$.

In this thesis, we address the general case of a collection of curves in $\mathbf{R}^2$ that are neither assumed to be closed nor smooth, i.e. we only require the curves to be differentiable in all but a finite number of points. Figure 1.1 gives an example. On the left the collection of curves Γ is shown, in the middle, a sample set S from Γ is visualized. The correct reconstruction $\mathrm{G}(S,\Gamma)$ of S with respect to Γ is shown on the right. The reader might say just by looking at this example, that it is obvious how to correctly reconstruct the sample points. The challenge is to design algorithms which do this 'obvious' job as good as a human being.

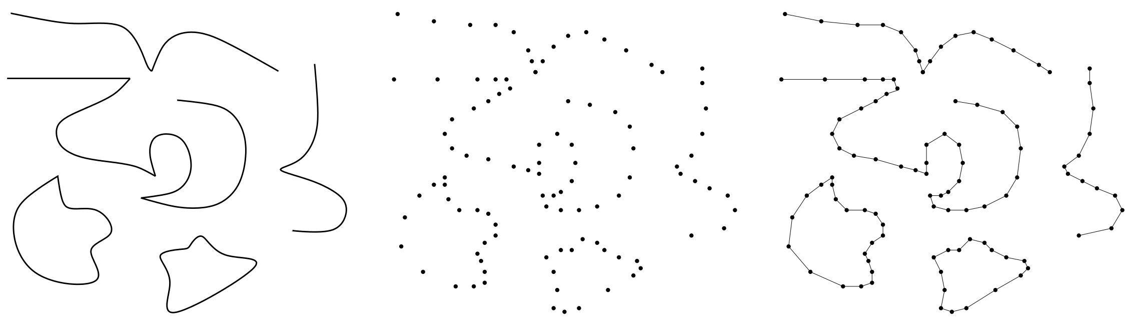


Figure 1.1: A collection of curves Γ, a sample S from Γ, and the correct reconstruction $\mathrm{G}(S,\Gamma)$

Practical Applications

The problem of reconstructing a surface from scattered sample points arises in many applications such as computer graphics, medical imaging, and cartography. Nevertheless there are applications where the underlying shape is a curve.

Computer Vision

Given a digital image of a scene with several objects, simple edge detection algorithms are used to select image pixels which are likely to belong to edges, often delimiting the boundaries of objects. Curve reconstruction algorithms then can be used to group these pixels into likely curves which then hopefully represent the shapes of the objects in the scene.

Surface Reconstruction from Layered Sample Sets

In some applications of surface reconstruction in $\mathbf{R}^3$, the sample sets are acquired in parallel slices, for example in computer tomography (CT) and magnetic resonance imaging (MRI). Each slice or layer is a sample from the shape that is the intersection of the scanned object with a plane. One approach to reconstruct the surface is first to reconstruct the shapes (=curves) in each layer and then connect adjacent layers appropriately. For example if two adjacent layers consist of a single curve each, the problem reduces to finding a cylindrical surface that connects the two polygons.

Plotting of Implicit Functions

In mathematics, some functions are given by the definition $f(x, y) = 0$. Such a definition is called an *implicit* definition. Many mathematical software packages support the plot of such implicit functions, if they can compute for every x all values y with $f(x, y) = 0$. The result of these plots is rather poor, if the function is self-intersecting or has sharp corners. To plot such an implicit function, one computes a large set of points on this function and reconstructs it with an appropriate algorithm.

Our Contribution

If the curve is closed, smooth, and uniformly sampled, several methods for the curve reconstruction problem are known to work ranging over minimum spanning trees [dFdMG95], α-shapes [BB97, EKS83], β-skeletons [KR85], and r-regular shapes [Att97]. A survey of these techniques appears in [Ede98]. The case of non-uniformly sampled closed curves was first treated successfully by Amenta, Bern and Eppstein [ABE98] and subsequently improved algorithms such as [DK99, Gol99] appeared. Open non-uniformly sampled curves were treated in [DMR99]. All the papers mentioned so far require the underlying curve to be smooth.

The first algorithm to treat one non-smooth curve was given by Giessen and later refined by Althaus/Mehlhorn. They showed that an open or closed non-smooth curve can be reconstructed using a TSP-tour. They were also able to show that an Integer-LP formulation of the problem can be solved in polynomial time. But still they failed for the general case where the sample set is taken from a *collection* of open and closed non-smooth curves. In this chapter we will present the following results:

- We give the first algorithm which provably can reconstruct a collection of non-smooth, open and closed curves Γ from a non-uniform sample set S. Our algorithm is completely combinatorial, does not rely on linear programming, and runs in polynomial time.

- In the proof of the correctness of our algorithm we introduce a sampling condition, different from those in the previous algorithms. We show that our sampling condition is implied by the other sampling conditions.

- As it was shown in [DMR99], allowing open curves but only giving a lower bound for the sampling density makes it impossible for an algorithm to compute *exactly* the correct reconstruction. So we prove that the output of our algorithm contains the correct reconstruction and that any extra edges in the output are justified by providing a collection of witness curves Γ' for which S is a valid sample and the output of our algorithm is exactly the correct reconstruction of S w.r.t. Γ'. Nevertheless, we also propose a slightly sharpened sampling condition, such that a modified version of our algorithm computes *exactly* the correct reconstruction. This is the first result of the type: *For any collection of curves with corners and endpoints, there exists a finite sample set such that it can be correctly reconstructed exactly by an algorithm.*

Additionally we have implemented a prototype of our algorithm that shows that it can be used in practice.

Outline

This chapter is organized as follows: In Sections 1.2 and 1.3 we start with some preliminaries and briefly review recent previous work on curve reconstruction. In Section 1.4, we discuss and present our choice of a sampling condition. Based on this sampling condition we develop our algorithm in Section 1.5 and prove its correctness in Section 1.6. Section 1.7 presents further results and variations of our algorithm and the sampling condition; in Section 1.8 we discuss the running time of our algorithm, its implementation, and present some experimental results. We conclude this chapter with some discussion and open problems.

1.2 Preliminaries

The following sections will heavily use terms, algorithms, and techniques from computational geometry, hence we first introduce the basic facts about Voronoi diagrams and Delaunay triangulations that are needed to understand the following presentation.

The results in this section are not original. They are presented only for the sake of completeness.

1.2.1 Convex Hulls and Triangulations

The *convex hull* of a set of points S is the minimal convex set of points covering S. Recall that a set X is called convex if for any two points p and q of X the entire line segment pq is contained in X.

A *triangulation* of a set of points S in the plane is a partition of the convex hull of S into triangles with three points of S as its vertices. Only if all points of a point set S are collinear, there is no triangulation of S.

Remark The convex hull $CH(S)$ of a set S of n points in the plane can be computed in $\mathrm{O}(n \log f)$, where f is the number of points in S which lie on the boundary of the convex hull. A triangulation of S can be obtained in $\mathrm{O}(n \log n)$.

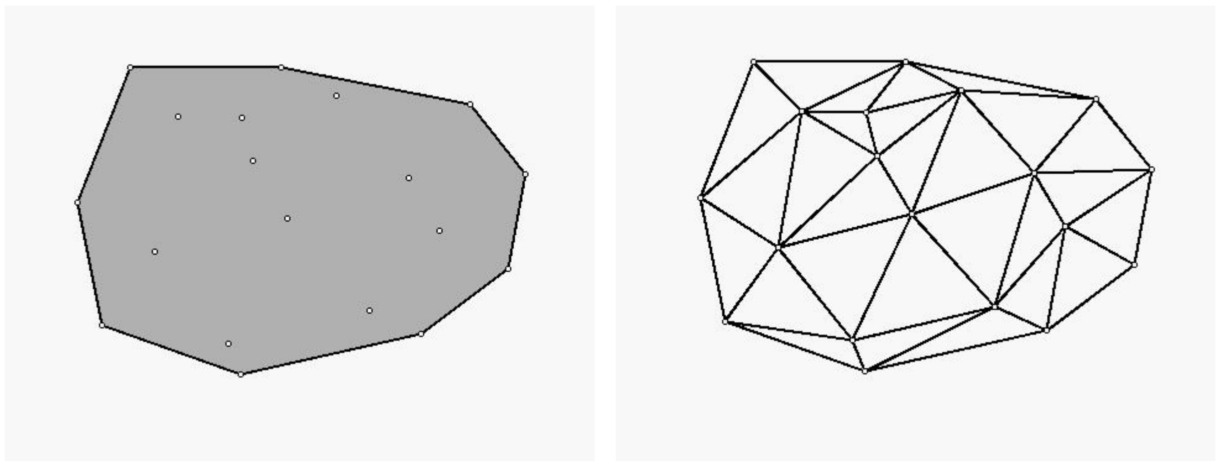


Figure 1.2: The convex hull and a triangulation of a point set

1.2.2 Delaunay Triangulations and Voronoi Diagrams

A triangulation of a point set S is called *Delaunay triangulation*$(\mathrm{DT}(S))$ if the interior of the minimum enclosing disk of any triangle in the triangulation contains no point of S. For example, the grey edges in Figure 1.3 connect the points in a Delaunay triangulation.

Interestingly one can show that the following alternative definition is equivalent to the one given above: A triangulation is a *Delaunay Triangulation* if for every edge $e = (p, q)$ there

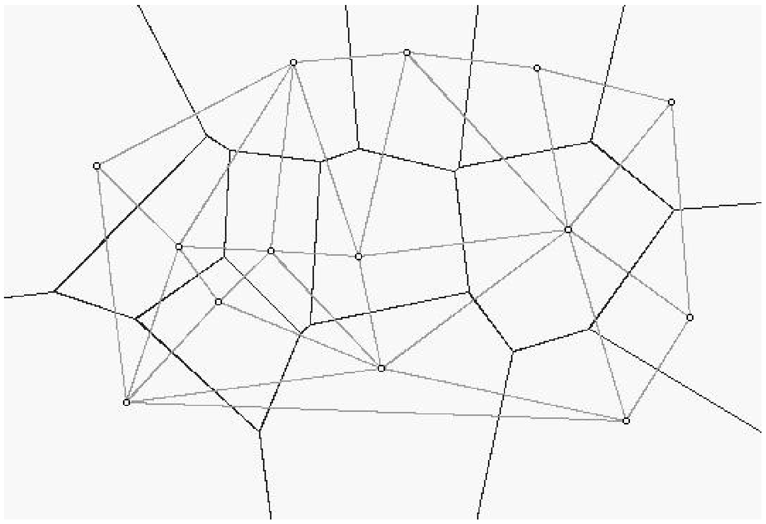


Figure 1.3: A Delaunay triangulation and the Voronoi diagram of a point set

exists a ball B with p and q on the boundary of B and no other points of S inside the ball.

Any triangulation of a point set can be turned into a Delaunay triangulation using several iterations of a local operation called *Delaunay flip*, see Figure 1.4. Starting from any triangulation of S, one inspects each edge $e = (p, r)$ and its adjacent triangles, here Δpqs and Δqrs. If r is in the circumcircle of Δpqs or p in the circumcircle of Δqrs – we say, e is not *locally Delaunay* – a flip of the edge (q, s) to (p, r) is performed, making this new edge locally Delaunay.

This procedure terminates when all edges are locally Delaunay. One can show that after at most $O(n^2)$ flip operations, all edges are locally Delaunay, and the resulting triangulation is a Delaunay triangulation.

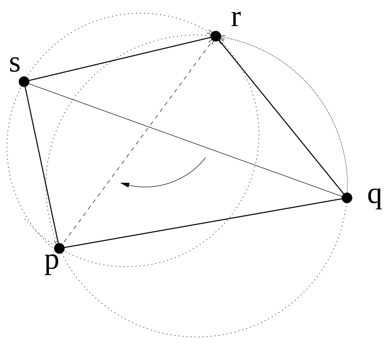


Figure 1.4: Delaunay Flip operation

Delaunay triangulations are not unique. Four cocircular points might introduce ambiguity as can be seen in Figure 1.5; either (p, r) or (s, q) can be part of a Delaunay triangulation. For the rest of this chapter, we will assume *general position* in a sense that no four points are cocircular. So for the point sets we are considering, there is only one unique Delaunay triangulation.

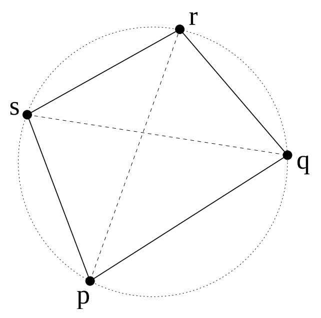


Figure 1.5: Cocircular points: ambiguous Delaunay triangulations

Remark The approach of first computing some triangulation of a point set S and then making it Delaunay by a sequence of flips can take $O(n^2)$ time in the worst case. There are other algorithms, though, which can compute the Delaunay triangulation of a point set in $O(n \log n)$.

Let S be a set of points in the plane. The *Voronoi diagram* $VD(S)$ of S is defined as the set of points in the plane with more than one nearest neighbor in S. $VD(S)$ consists of *Voronoi nodes* (which have three or more nearest point in S) and *Voronoi edges* (which have exactly two nearest points in S), which might be either segments, rays or lines (lines only occur in the degenerate case that all points in S are collinear). Voronoi edges do not intersect and only meet in Vornoi nodes. Each point $p \in S$ has a *Voronoi region* $VR(p)$ associated with it. $VR(p)$ might be unbounded but is delimited by Voronoi edges and Voronoi nodes. We call $VR(p)$ also a *face* of the Voronoi diagram. $VR(p)$ is a convex polygon and corresponds to the region which has p as closest point in S.

In Figure 1.3, the black edges show the Voronoi diagram of the point set. Voronoi diagrams and Delaunay diagrams are closely related structures. In fact, each one of them can be easily obtained from the other. For simplicity, we assume that the points in S are not collinear. We show how to obtain the Voronoi diagram $VD(S)$ from the Delaunay triangulation $DT(S)$ using the following duality transformation:

- The Voronoi nodes of $VD(S)$ are the centers of the circumcircles of the Delaunay triangles.

- Let $e = (p, q)$ be an edge of the Delaunay Diagram:

- If e is not an edge of the convex hull of S, then there is a Voronoi edge e' between the two Voronoi nodes created by the the two triangles which have e as one of their sides.
- If e is an edge of the convex hull, the Voronoi diagram has the ray, which is part of the bisector between p and q, starting at the Voronoi node defined by the triangle containing e.

Thus we have a node in VD(S) for every triangle in DT(S), an edge in VD(S) for every edge in DT(S), and a face in VD(S) for every node in DT(S). These are called the *dual node*, *dual edge*, and *dual face*, respectively. The reader is encouraged to check the duality transformation on the example in Figure 1.3, where both, the Voronoi diagram as well as the Delaunay triangulation of the point set are shown.

1.3 Previous Work

Many algorithms have been proposed for the curve reconstruction problem. In the following we will be only concerned with reconstruction algorithms that come with a guarantee, i.e. under certain assumptions on Γ and the sample set S taken from Γ, they output the correct reconstruction as defined in section 1.1.

The first algorithms like [dFdMG95], α-shapes [BB97, EKS83], β-skeleton [KR85], and r-regular shapes [Att97] are known to work if the curve is closed, smooth and *uniformly* sampled, i.e. the distance between two adjacent samples must be less than some constant, which is determined by the most detailed area of the curve. A survey on these techniques appears in [Ede98].

The case of non-uniformly sampled closed curves was first treated successfully by Amenta, Bern and Eppstein [AB99] and subsequently improved algorithms such as [DK99, Gol99] appeared. Open non-uniformly sampled curves were treated in [DMR99]. All these algorithms follow a "Delaunay Filtering" approach in a sense that they start from the Delaunay triangulation of the point set and locally decide for each edge whether it should remain in the reconstruction. A different approach was presented by Giesen [Gie99]. He showed that for a sufficiently dense sampling, the TSP (travelling salesperson) tour is the correct reconstruction for a single closed – not necessarily smooth – curve. Later Althaus and Mehlhorn [AM00] extended this result by showing that in this special instance of the TSP problem, the TSP tour can be computed in polynomial time using linear programming techniques.

We review some of the algorithms which work for non-uniform sample sets. Before we sketch these algorithms, we introduce some notations for curves and prove some basic facts.

1.3.1 Terminology and Basic Properties of Curves

We will identify a curve by an embedding: A *single open curve* is given by an *embedding* $\gamma : [0,1] \rightarrow \mathbf{R}^2$ and a *single closed curve* is given by an embedding $\gamma : S^2 \rightarrow \mathbf{R}^2$, where S^2 is the unit circle.

We call a single curve γ *semi-regular* or *non-smooth*, if at any point of the curve left and right tangents exist and only at a finite number of points they disagree. If the tangents always coincide, we say the curve is *regular* or *smooth*. A more rigorous definition of regularity can be found in [Gie99].

In the following we will also be concerned with a collection of curves Γ, which is a finite set of disjoint single semi-regular curves $\gamma_1, \ldots, \gamma_k$.

Figure 1.6 shows a closed smooth curve and an open non-smooth curve with disagreeing left and right tangents at c. We call c a *singularity* or *corner point* of the curve, and α the *corner angle*.

A *sample set* S of a collection of curves Γ is a finite set of points $s \in \Gamma$. Two points p_1 and p_2 are *adjacent* in Γ, if they both lie on some $\gamma_i \in \Gamma$, and there is a path from p_1 to p_2 on γ_i without visiting any sample $s \in S$.

As we have already mentioned, the *correct reconstruction* of a sample set S with respect to some Γ is defined as the graph $G(S, \Gamma)$ with with node set S and an edge between two

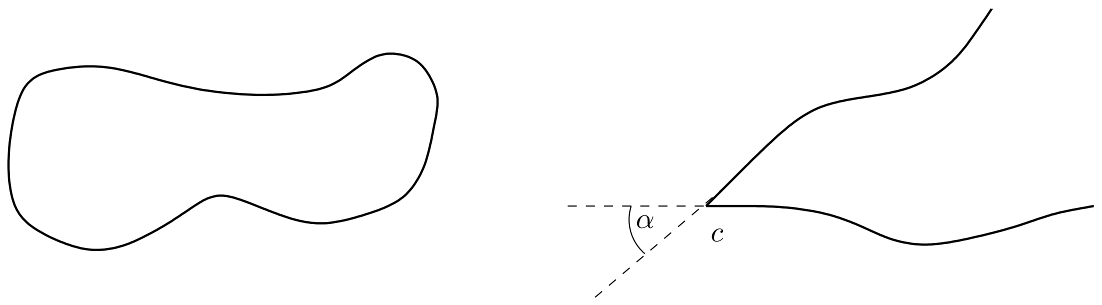


Figure 1.6: A closed smooth curve and an open non-smooth curve with corner angle α at c

vertices p and q if and only if p and q are adjacent in Γ.

1.3.2 Medial Axis and Local Feature Size

The *medial axis* M of a collection of Curves Γ is defined as the closure of all points having more than one nearest neighbour in Γ. See Figure 1.7 for an example of a curve and its medial axis.

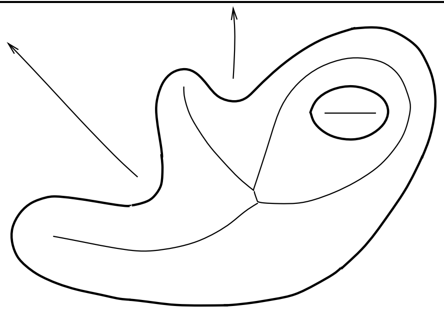


Figure 1.7: The light curves are the medial axis of the heavy curves. *Courtesy of N. Amenta, M. Bern, and D. Eppstein.*

Amenta, Bern and Eppstein [ABE98] use the medial axis to define a very elegant, non-uniform sampling condition. The non-uniformity comes from the fact, that in detailed areas of Γ, the medial axis is close to Γ, whereas in less detailed areas, the medial axis is far away from Γ.

They define the *local feature size* $\mathrm{lfs}(p)$ of a point $p \in \Gamma$ as the distance of p to the medial axis of Γ. For any $\epsilon > 0$, a sample set S is called a ϵ-sample – or ϵ-sampled –, if for every $p \in \Gamma$ there is a sample $s \in S$ with $dist(s,p) \leq \epsilon \mathrm{lfs}(p)$.

Remark Note that if there is a corner in a curve γ, the medial axis actually touches the curve in the corner point. Therefore, the local feature size near the corner point gets arbitrarily small, and no finite ϵ-sample exists for this γ. But as for now we will only be concerned with collections of *smooth* curves, this case never happens, and a finite ϵ-sample always exists.

For the further discussion we need the notion of a *curve Voronoi vertex* between two adjacent samples p and q which is defined as the intersection of the curve between p and q and the bisector of p and q (assuming that from every closed curve $\gamma \in \Gamma$ at least three samples were taken). We now summarize a few relations between the medial axis, the local feature size and the correct reconstruction. The first Lemma is a simple application of the triangle-inequality.

Lemma 1 $\mathrm{lfs}(p) \leq \mathrm{lfs}(q) + dist(p, q)$ *for all* $p, q \in \gamma$.

As direct consequence of this Lemma we can bound the length of a segment adjacent to some sample point $p \in \Gamma$ in terms of the local feature size in p.

Lemma 2 *The length of a segment* pq *of the polygonal reconstruction of* ϵ*-sampled collection of smooth curves* Γ *is at most* $(2\epsilon/(1-\epsilon))\mathrm{lfs}(p)$.

Proof. Let r be the curve Voronoi vertex of Γ between p and q. Then $dist(p, r) \leq \epsilon \mathrm{lfs}(r)$ and $dist(p, r) \geq dist(p, q)/2$. By Lemma 1 we have $\mathrm{lfs}(r) \leq dist(p, r) + \mathrm{lfs}(p)$. So $dist(p, r) \leq \epsilon(dist(p, r) + \mathrm{lfs}(p))$. Algebraic transformations lead to $dist(p, r) \leq \epsilon/(1-\epsilon)\mathrm{lfs}(p)$. Thus $dist(p, q) \leq 2\epsilon/(1-\epsilon)\mathrm{lfs}(p)$. $\square$

The following two Lemmas make the intuition precise that in detailed areas of the curve, the medial axis is close.

Lemma 3 *Let* B *be a ball so that* $B \cap \Gamma$ *is not connected. Then* B *contains a medial axis point.*

Proof. For a point x let x^* be a point on Γ with minimal distance to x. Let p and q be two points of Γ in different connected components with respect to B. Assume we are moving a point x from p to the center of B and then to q. In the beginning, the nearest point of Γ is p and at the end, the nearest point of Γ is q. Since $dist(x, x^*)$ is continuous, either the x^* remains in the component of p or there are at least two points on Γ with distance $dist(x, x^*)$ to x. This x is a medial axis point of Γ. $\square$

Lemma 4 *Let* S *be an* ϵ*-sample set from a collection of closed smooth curves* Γ*,* $\epsilon \leq 1$*, and* c *be the curve Voronoi vertex between two adjacent samples* p *and* q*. Then the open disk* D *centered at* c *touching* p *and* q *is empty of samples.*

Proof. According to Lemma 3, $D \cap \Gamma$ consists of one component only, otherwise D would contain a point of the medial axis. p and q are the points, where this component enters and leaves D without any samples between p and q on Γ. This proves the Lemma. $\square$

The next Lemma exhibits the connection between the Voronoi diagram and the medial axis, namely the property that the Voronoi vertices approximate the medial axis. One can even show that if the sampling density goes to infinity, the Voronoi vertices converge to the medial axis (this is only true in two dimensions, in three dimensions, Voronoi vertices can be arbitrarily far away from the medial axis).

Lemma 5 *Let S be some sample set from a collection of closed smooth curves Γ, D the maximal closed disk centered at a Voronoi vertex of VD(S) which contains no samples in its interior. Then D must contain a medial axis point.*

Proof. Assume first that in the neighbourhood of one of the samples $s \in S$ on the boundary of D, $\Gamma - s$ is contained completely in D. Then either $D \cap \Gamma$ is entirely contained in the boundary of D and the center of D is a point of the medial axis, or shrinking D around s will produce a smaller disk D_ρ, contained in D, with $D_\rho \cap \Gamma$ consisting of at least two connected components. By Lemma 3, D_ρ contains a point of the medial axis.

If there is no such s, then the intersection of F with B already consists of at least two connected components, and B contains a point of the medial axis by Lemma 3.

$\square$

1.3.3 The Algorithms of Amenta, Bern, and Eppstein

After defining the concept of *local feature size* and ϵ-sample, Amenta, Bern, and Eppstein in [ABE98] present two algorithms – CRUST and β-CRUST – which they prove to correctly reconstruct a collection of closed smooth curves Γ if the sample set S is an ϵ-sample for $\epsilon < 0.252$ in case of the CRUST And $\epsilon < 0.297$ in case of the β-CRUST.

The β-CRUST Algorithm

We will first present the β-CRUST, an algorithm which was known before to be correct for uniform sample sets. It works as follows:

1. Compute the Delaunay Triangulation T of S.

2. Keep an edge $e = (p, q)$ of T iff the two closed balls of radius $\beta \cdot |e|/2$ touching p and q – also called β-balls – are empty of other samples.

See Figure 1.8 for the β-balls of an edge (p, q).

In the following we will prove that for $\beta = 2$, β-CRUST computes the correct reconstruction if S is a 1/3-sample from Γ.

Lemma 6 *Let γ be the angle between an edge $e = (p, q)$ and the tangents of its β-balls in p or q. Then $\gamma = \arcsin \frac{1}{\beta}$.*

Proof. Consider the edge e and and one of its β-balls as in the left picture of Figure 1.9. We have $\cos \alpha = \cos(\frac{\pi}{2} - \gamma) = \frac{|e|/2}{\beta \cdot |e|/2} = 1/\beta$. Hence $\gamma = \arcsin \frac{1}{\beta}$. $\square$

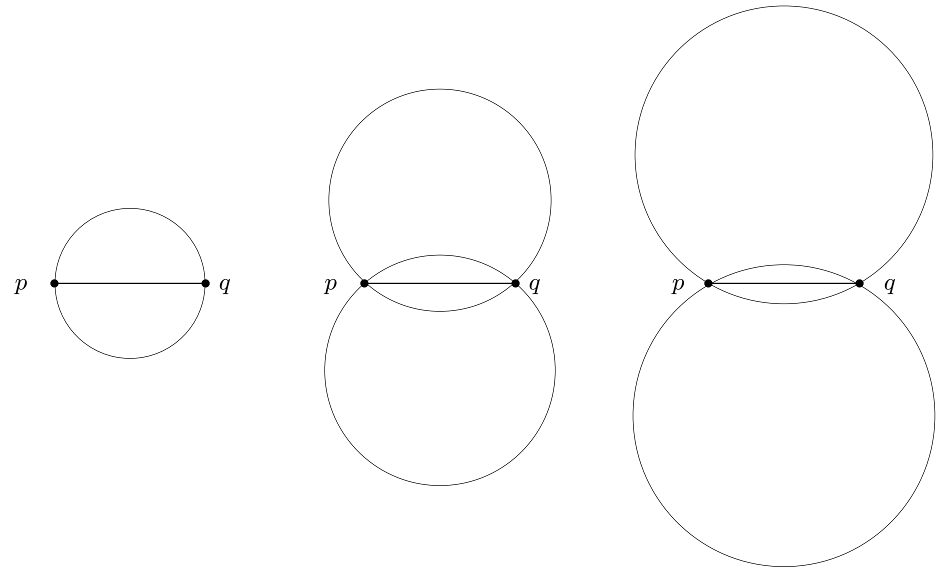


Figure 1.8: β-balls for an edge (p, q) and $\beta = 1, 3/2, 2$

.

Lemma 7 *Let o, p, q be three points in the plane such that both edges $e_1 = (o, p)$ and $e_2 = (p, q)$ have empty β-balls. Then we have that the angle between e_1 and e_2 is larger than $\pi - 2 \cdot \arcsin \frac{1}{\beta}$.*

Proof. Consider the ball B of radius r touching o,p and q as in the right picture of Figure 1.9. Clearly, $r \geq \beta \cdot |e_1|/2$ and $r \geq \beta \cdot |e_2|/2$, as both e_1 and e_2 have empty β-balls. Hence we know that for $i = 1, 2$: $\gamma_i \leq \arcsin \frac{1}{\beta}$ and therefore $\alpha = \pi - \gamma_1 - \gamma_2 \geq \pi - 2 \cdot \arcsin \frac{1}{\beta}$. □

Lemma 8 *If S is ϵ-sampled from a collection Γ of smooth closed curves, then the β-CRUST contains all edges of the correct reconstruction for $\epsilon \leq 1/(2\beta - 1)$ and $\epsilon \leq \cos \arcsin \frac{1}{\beta}$.*

Proof. Let us consider an edge $e = (p, q)$ in the correct reconstruction $\mathrm{G}(S, \Gamma)$.

Assume now that an edge e of the correct reconstruction is not part of the output of the β-CRUST, i.e. at least one of the two balls B_1, B_2 centered at b_1,b_2 of radius $r = \beta \cdot |e|/2$ touching p and q is not empty of other samples. Let c be the curve Voronoi vertex between p and q, B_c the ball centered at c touching p and q. According to Lemma 4 B_c has no samples in its interior.

We will distinguish two cases. First assume that c lies on the segment $b_1 b_2$, as in the left picture of Figure 1.10. We show that when moving the center c towards either b_1 or b_2, no other sample can enter the ball B_c (always centered at c and touching p and q) hence contradicting the assumption that the interior of $B_1 \cup B_2$ is non-empty. Assume otherwise, i.e. at some point, before reaching either b_1 or b_2, B_c touches a third sample x with a radius of r'. According to Lemma 5 we know that inside B_c there must be a medial axis point. But as c is still between b_1 and b_2, $r' < \beta \cdot |e|/2$, and hence $\mathrm{lfs}(p) < 2 \cdot r' < \beta \cdot |e|$. On the

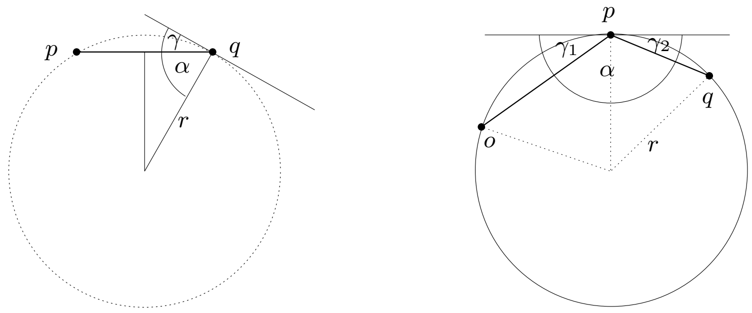


Figure 1.9: Properties of edges and their β-balls.

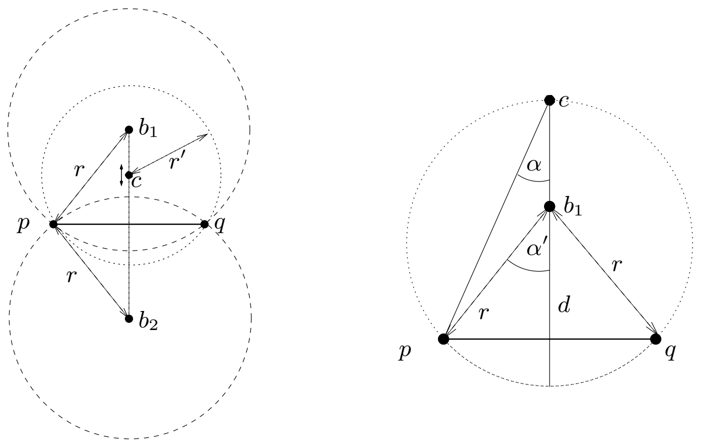


Figure 1.10: Emptiness of β-balls

other hand, according to Lemma 2 we have $|e| \leq \frac{2\epsilon}{1-\epsilon}\mathrm{lfs}(p)$, which yields a contradiction for $\epsilon \leq \frac{1}{2\beta-1}$.

Now assume w.l.o.g. that c is on the bisector of p and q *above* b_1, as in the right picture of Figure 1.10. But then we know that there must be a medial axis point inside the ball touching p,q and c (as we could simply place a sample at c), i.e. the local feature size of c can be at most the diameter d of this ball B_c. Let α be the angle between the bisector of p and q and pc. As c lies above b_1, we know that $\alpha < \alpha = \arcsin \frac{1}{\beta}$. and hence $d = dist(p,c)/\cos\alpha < dist(p,c)/\cos\arcsin\frac{1}{\beta}$. If S is really ϵ-sampled, $d(p,c) < \epsilon \cdot d$ or $d > d(p,c)/\epsilon$ which gives a contradiction for $\epsilon < \cos\arcsin\frac{1}{\beta}$. ◻

Lemma 9 *For $\beta \geq 2$, every sample has at most two edges in the β-CRUST.*

Proof. The angle between two edges in the β-CRUST is at least $\pi - 2 \cdot \arcsin\frac{1}{\beta}$ according to Lemma 7. For $\beta \geq 2$, this angle is larger than $2 \cdot \pi/3$. ◻

So we have shown that in the output of β-CRUST, each edge of the correct reconstruction has survived, and each sample has at most two adjacent edges. As each sample has also two edges in the correct reconstruction, those have to be identical and we get the following corollary:

Corollary 1 *For $\beta = 2$, the β-CRUST computes the correct reconstruction $G(S, \Gamma)$ of a ϵ-sampled set S with respect to a collection of closed smooth curves Γ if $\epsilon \leq \frac{1}{3}$.*

The CRUST Algorithm

The second algorithm – CRUST – which Amenta, Bern, and Eppstein present in their paper [ABE98] is slightly different. In some sense it reflects more directly the property of the sampling condition that the medial axis should be far away from an edge relative to its length. They prove that CRUST correctly reconstructs a collection of closed smooth curves Γ if the sample set S is an ϵ-sample for $\epsilon < 0.252$. It works as follows:

1. Compute the Voronoi diagram V of the sample points.
2. Compute the Delaunay Triangulation T of $V \cup S$.
3. Select all edges of T which connect two sample points.

Intuitively, their algorithm makes use of the fact that for a sufficiently dense sample set S, i.e. for a sufficiently small ϵ, the vertices of the Voronoi diagram of S approximate the medial axis (see Lemma 5). They only keep those edges of the Delaunay triangulation of S, for which there exists a ball touching the endpoints and empty both of samples and Voronoi vertices. It turns out that that these edges indeed are the correct reconstruction of S with respect to Γ.

For sake of simplicity we only prove the correctness for a slightly weaker bound of $\epsilon = 0.23$. The proof again proceeds in two steps, we first prove that all edges of the correct reconstruction are selected and then that no edge which is not part of the correct reconstruction is selected.

Lemma 10 *If S is an $1/3$–sample of a collection smooth closed curves Γ, the CRUST algorithm selects all edges of the correct reconstruction.*

Proof. We prove that for $\epsilon \leq 1/3$, each edge $e = (p, q)$ of the correct reconstruction has a ball touching p and q which is empty of Voronoi vertices of the Voronoi diagram of S.

Consider the curve Voronoi vertex c between p and q as in Figure 1.11. We claim that this ball is empty of Voronoi vertices. Assume otherwise, i.e. there exists a Voronoi vertex x inside the curve Voronoi disc D_c centered at c. We know that D_c has radius $r < \epsilon \cdot \mathrm{lfs}(c)$, therefore $d(p, x) < 2 \cdot \epsilon \cdot \mathrm{lfs}(c)$, i.e. the maximal empty ball D_x around x has radius at most $R = 2 \cdot \epsilon \cdot \mathrm{lfs}(c)$. According to Lemma 5, there must be medial axis point inside D_x. So we have $\mathrm{lfs}(c) < r + R < 3 \cdot \epsilon \cdot \mathrm{lfs}(c)$, which gives a contradiction for $\epsilon \leq 1/3$.

□

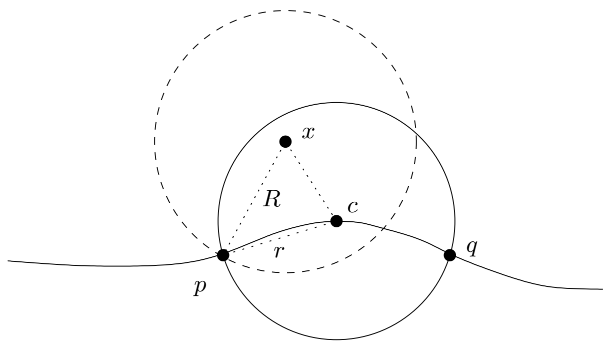


Figure 1.11: Construction of a ball empty of Voronoi vertices

Lemma 11 *If S is a 0.23–sample of a smooth closed curve γ, the CRUST algorithm returns no edges that are not in the correct reconstruction.*

Proof. Assume there exists an edge st which is not part of the correct reconstruction, but which has ball B touching s and t empty both of other samples and Voronoi vertices as in Figure 1.12. Let s_1, s_2 be the neighbours of s in the correct reconstruction.

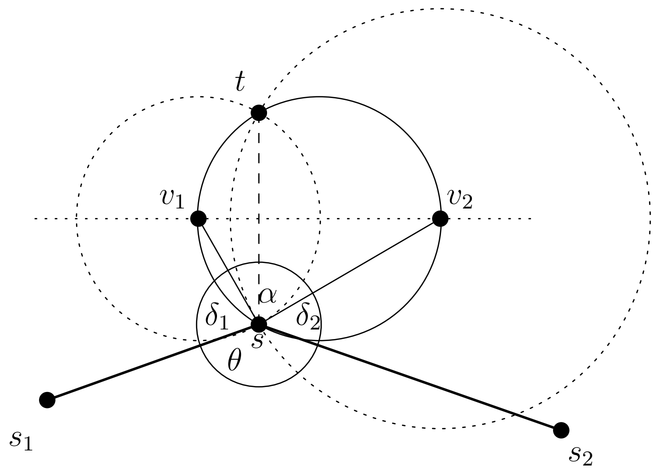


Figure 1.12: There is no ball empty of samples and Voronoi vertices!

Consider the intersection points v_1 and v_2 of B with the perpendicular bisector between s and t. Observe that the balls centered at v_1 and v_2 touching s and t are empty of samples, as B was empty of Voronoi vertices.

We will lower bound the angles around s. According to Lemma 7, $\theta > \pi - 2 \cdot \arcsin 1/\beta$. Furthermore, $\alpha = \pi/2$ as s lies on the circle with equator v_1v_2. It remains to give lower bounds for δ_1 and δ_2. First observe that the edges s_1s and ss_2 have empty β-balls on both sides (for which β is determined by the choice of ϵ). Look at Figure 1.13. s_1s has two β-balls centered at b_1 and b_2. Therefore we know that b_1b_2 is part of the boundary of the Voronoi cell of s. Furthermore $d(v_1, s_1) \geq d(v_1, s)$ and hence we know that v_1 lies to the right of the bisector between s_1 and s. As Voronoi cells are convex, v_1 cannot lie inside the triangle Δsmb_1. So γ is a lower bound for δ_1, i.e. $\delta_i \geq \gamma = \pi/2 - \arcsin 1/\beta$ according to Lemma 6.

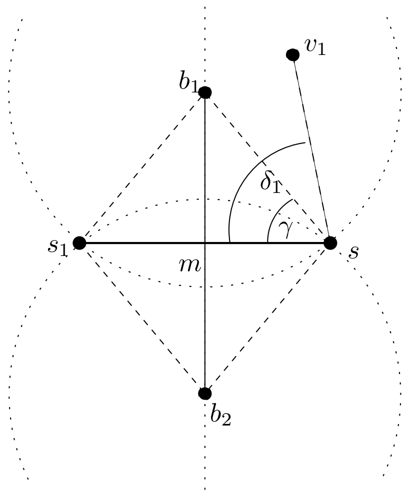


Figure 1.13: β-balls imply large δ_i!

So summing up the angles around s we have

$$\alpha + \theta + \delta_1 + \delta_2 > \pi - 2 \cdot \arcsin 1/\beta + \pi/2 + 2 \cdot (\pi/2 - \arcsin 1/\beta)$$

$$= 2 \cdot \pi + \pi/2 - 4 \cdot \arcsin 1/\beta > 2 \cdot \pi$$

for $\beta > 2.62$ or equivalently (Lemma 8) $\epsilon < 0.23$. $\square$

1.3.4 Variations on the CRUST algorithm

Following the work of Amenta, Bern, and Eppstein, other algorithms were presented which also use the framework of medial axis and local feature size.

Gold and Snoeyink [Gol99] observed that it suffices to select each Delaunay edge for which there exists a touching ball empty of samples and which does not contain the endpoints of its dual Voronoi edge. Note that our proof for the CRUST algorithm also applies to the Gold-Snoeyink algorithm. Gold and Snoeyink also provided an improved analysis showing that their algorithm works for $\epsilon \approx 0.42$.

An algorithm rather different from the ones mentioned so far was given by Dey and Kumar in [DK99]. It is basically a 'connect-to-the-nearest-neighbor' approach. What is interesting about it, is the fact that they could prove its correctness in the framework of the local feature size sampling condition. It works as follows:

1. For each sample s insert an edge to its nearest neighbour $n(p)$.

2. Each sample s which has only one adjacent edge to $n(s)$, picks the nearest neighbor $n'(s)$ which forms an angle of more than $\pi/2$ with $sn(s)$.

They show that this simple algorithm computes the correct reconstruction for a collection of smooth closed curves if $\epsilon \leq 1/3$. Althaus in [Alt01] showed that for a slightly modified version of the algorithm (picking $n(s)$ in a smaller cone), the algorithm computes the correct reconstruction for $\epsilon \leq 0.5$.

1.3.5 The Algorithm of Dey, Mehlhorn, and Ramos

The first algorithm that comes with a guaranteee for a collection of *open* and closed curves was given by Dey, Mehlhorn, and Ramos in [DMR99]. Here we will restrict to a quick summary of their result and discuss the problems which arise when allowing open curves.

When allowing open curves it is not possible for an algorithm to compute exactly the correct reconstruction $G(S,\Gamma)$ without assuming any condition additional to the local feature size sampling condition. Look at the example in Figure 1.14. Assume we have a valid sample set S for the *open* circular arc from p to q in the left picture. Unfortunately, this sample set S is also valid for the circle in the right picture, if we use the sampling condition with respect to the local feature size. So it is impossible for an algorithm to know whether to connect p and q or not unless it knows the original collection of Curves Γ where S is sampled from.

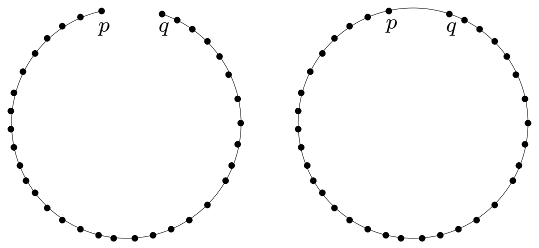


Figure 1.14: Connect p and q or not ?

So what kind of guarantee can we expect from an algorithm which claims to reconstruct open and closed curves ? First, we can expect that the output of the algorithm *at least* contains all edges which are in the correct reconstruction. But as we have seen, we cannot guarantee that there are no additional edges in the output. So the idea is to make sure that each additional edge can be *justified*. How do we do that ? We only return edges for which there exists a collection of open and closed curves Γ', also called *witness curves*, such that S is a valid sample set (with a slightly weakened sampling condition) for Γ' and the output is the correct reconstruction $G(S,\Gamma')$ of S with respect to Γ'. Note that in this case we have $G(S,\Gamma') \supseteq G(S,\Gamma)$.

As mentioned, the sampling condition for which S is a valid sample set for Γ' is weaker than the sampling condition which guarantees that all edges of $G(S,\Gamma)$ are part of $G(S,\Gamma')$. So the algorithm of Dey, Mehlhorn, and Ramos has a parameter ρ to adjust the required and the guaranteed sampling density. The exact result is stated as follows:

Theorem 1 (Dey, Mehlhorn, Ramos (99)) *Let S be a set of points in the plane and $\rho < 1$. The reconstruction G and the curve Γ' returned by the algorithm of Dey, Mehlhorn, and Ramos with parameter ρ satisfy the following two properties:*

- *If $\rho < 1/2$ and S is a $(\rho/8)$–sample from a collection of open and closed curves Γ then the correct reconstruction $G(S, \Gamma)$ of S w.r.t. Γ is a subset of G.*
- *Let $c = 13.35$. If $\rho < 1/8$ then S is a $(c\rho)$–sample from Γ' and $G = G(S, \Gamma')$, i.e. G is the polygonal reconstruction of S w.r.t. Γ'.*

As the algorithm has to justify each edge of the output, it has to be more strict in when pruning edges. While the CRUST variants required only the existence of *one* touching ball empty both of Voronoi vertices and other samples for an edge $e = (p, q)$ to 'survive', the algorithm by Dey, Mehlhorn, and Ramos has to make sure that in the close neighbourhood of e there are *no* Voronoi vertices at all (since a close Voronoi vertex implies a point of the medial axis nearby according to Lemma 5) and no endpoints of the curve (as this also induces a nearby point of the medial axis with arguments similar to the ones used in Lemma 3).

More precisely, their algorithm removes edges until for each edge (p, q), the ball of radius $dist(p, q)/\rho$ centered at the midpoint of pq is empty of Voronoi vertices of S and the ball centered at the midpoint of (p, q) with radius $dist(p, q)/(4\rho)$ contains no node with degree less than 2.

The proof techniques to show that the edges of the correct reconstruction have these properties are very similar to the ones we have seen in this section. The last step of their algorithm is the construction the collection of witness curves Γ'.

1.3.6 The TSP-based algorithms by Giessen and Althaus/Mehlhorn

So far, we have always assumed the existence of a valid sample set with respect to the local feature size criterion. But as briefly mentioned, this framework does not work if Γ contains non-smooth curves, as in this case, the medial axis actually passes through the points of non-smoothness – the corner points, demanding an infinite sampling density near the corners.

Giesen in [Gie99] obtained the first result for *one* closed non-smooth curve Γ [1]. He showed that for a sufficiently dense sample set from Γ, the *Travelling Salesman tour* of the sample set S is the correct reconstruction of S w.r.t. Γ.

Later Althaus [Alt01] and Althaus/Mehlhorn [AM00] improved this result in several ways:

- they provide a non-uniform sampling condition similar to the one using the concept of local feature size, for which the travelling salesman tour is the correct reconstruction
- they show that the travelling salesman tour can also be used to reconstruct one open curve with endpoints or several closed curves
- they show that this special instance of the travelling salesman problem can be solved in polynomial time if S is a valid sample set

[1] Actually Giesen excluded the case that in a corner left and right tangent make an angle of π.

Their algorithm is based on techniques from linear programming, so we will not go into further details. We only remark that the general case where the task is to reconstruct a collection of curves with corners *and* endpoints was still not solved.

1.4 Sampling Conditions

The problem of curve reconstruction as stated in the introduction only postulates that we connect all samples which are adjacent on the original curve. The sampling condition with respect to the medial axis does more, though. It also makes sure that the correct reconstruction does not miss a single feature or 'wiggle' of the original curve. So it even guarantees that we get a faithful *approximation* to the curve. The question is whether we really want that. See for example Figure 1.15.

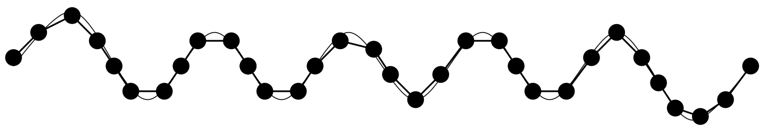


Figure 1.15: Wiggling Curve sampled according to the local feature size

If we apply the medial axis sampling condition, we are forced to sample this curve very densely as the 'wiggling' implies a very small local feature size for any point of the curve, and the correct reconstruction of the sample set is a pretty good approximation of the original curve not missing a single 'wiggle'. But if the only thing that we want is connecting the right samples (as stated in our definition of the curve reconstruction problem), a far less dense sampling should also do, like in Figure 1.16.

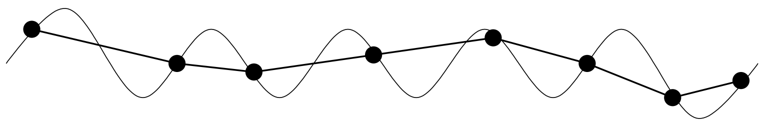


Figure 1.16: Wiggling Curve with a sparser sampling

Of course, now the correct reconstruction does not approximate the curve as with the higher sampling density, but nevertheless, a correct reconstruction should be possible as well.

So what we propose is the following: The decision of whether a given sample set S from a collection of curves Γ is valid should not be stated with respect to the original curve, but rather with respect to the correct reconstruction $G(S, \Gamma)$ of this sample set. Such a sampling condition would allow to 'skip' details of a curve if this does not affect the possibility to correctly reconstruct the curve (in the sense of our original definition of the curve reconstruction problem).

As mentioned in the introduction, one problem with a sampling condition with respect to the medial axis is the fact that for curves with corners, the medial axis passes through the corners, hence requiring a infinitely dense sampling near corners. This can be fixed by relaxing the sampling condition within controlled areas around the corners as in [Alt01, Gie99]. We will use the same idea for our sampling condition in the next section.

We believe that a sampling condition expressed with respect to the correct reconstruction $G(S,\Gamma)$ is more sensible than a sampling condition expressed directly with respect to the curve (and the medial axis). However, since the medial axis sampling condition has become quite standard in recent work on curve reconstruction, we will also show that our sampling condition is implied by the medial axis condition, *i.e.* all sample sets of a curve that are valid with respect to some medial axis sampling condition are also valid with respect to our sampling condition, i.e. our sampling condition is strictly weaker.

1.4.1 Our Sampling Condition

Our sampling condition describes how the correct reconstruction $G(S,\Gamma)$ of a sample set S with respect to a collection Γ of open and closed curves (possibly with corners) must look like to guarantee certain properties of the output of our algorithm.

Let us first consider a collection of open and closed curves without corners. Our sampling condition is as follows:

Sampling Condition for Smooth Areas: A sample set S for a collection of open and closed smooth curves Γ is valid, if for every edge $e = (p,q)$ of the correct reconstruction $G(S,\Gamma)$ the following holds (also see Figure 1.17):

1. the two closed balls B_1, B_2 of radius $r_1 = \beta \cdot |e|/2$ touching p and q are empty of other samples
2. all samples within the ball B_3 of radius $r_3 = f_{diam} \cdot |e|$ centered at the midpoint of e are connected to e in a chain that makes a total turn of less than $\theta_{ball} < \pi/2$ on each side of e. We call B_3 the *diametral ball of* e, though it is not 'diametral' in the common sense.

Basically, part one of our sampling condition resembles the properties of the β-CRUST, whereas part two expresses the additional constraint that in the neighbourhood of an edge (where the size of the neighbourhood is relative to the length of the edge), the correct reconstruction is a relatively straight polygonal chain. This also resembles the condition checked in [DMR99].

As with the medial axis sampling condition, the problems arise near corners. To solve them, we first have to identify areas near a corner and then define a weaker sampling condition for edges of the correct reconstruction which are completely contained in such a corner region. In the following we need the concept of a *cone of slope*. For a curve segment s the *cone of slope* of s is defined as the set of all lines through the origin which have the same slope as the tangent at some point in the interior of s, see Figure 1.18.

Now we can identify the corner areas:

Identification of corner areas: For each corner grow a ball around the corner point as long as:

1. the ball intersects the curves in Γ in two smooth curve segments, which we refer to as the *legs* of the corner, each with an endpoint in the corner and the other on the boundary of the ball

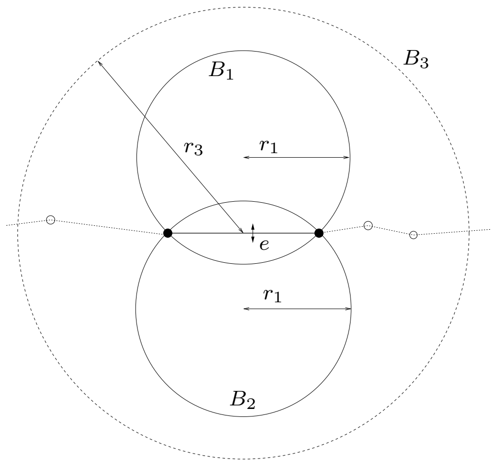


Figure 1.17: Sampling Condition for smooth areas
The sampling condition in B_3 implies that the samples left of e must be connected in decreasing order of their x-coordinates.

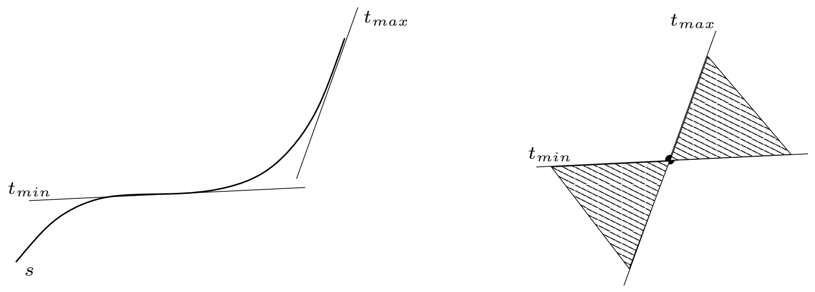


Figure 1.18: Cone of slope for a curve segment s.

2. on each leg of the corner the tangent varies by at most θ_{slope}
3. the cones of slope for each leg do not intersect

We call the maximal ball obtained by this procedure the *unshrunken corner ball* of this corner. It is then shrunken by a factor of f_{shrink} to obtain the *shrunken corner ball*. The area within the shrunken ball defines a *corner area*.

Intuitively, part one and two make sure that the curve segments ending in a corner are relatively flat, part three excludes cases like in Figure 1.19, whereas the shrinking process ensures that corner areas are well separated from each other and other detailed areas of Γ.

For a shrunken or unshrunken corner ball, the two legs ending at the corner partition the corner ball into two regions. We call the region which contains the angle at the corner point less than π the *inside* of the corner, the other region is called the *outside* of the corner.

Now we have to state a weaker sampling condition for these corner areas. First we drop

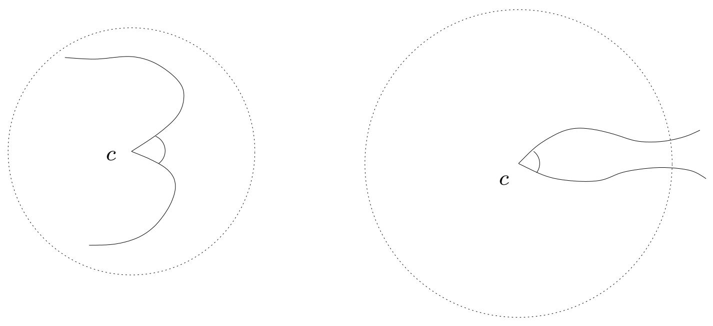


Figure 1.19: Excluded cases when growing the corner ball.

the second part of the sampling condition for smooth areas of the curve, and allow samples on the other corner leg in the β-ball touching an edge from 'inside'. Additionally, we add a condition that makes sure that we can somehow locally decide to which leg a sample belongs to. Otherwise it seems difficult to decide locally whether there is a corner or not as it can be seen in Figure 1.20 (unless, we make further assumptions, e.g. that there are no open curves).

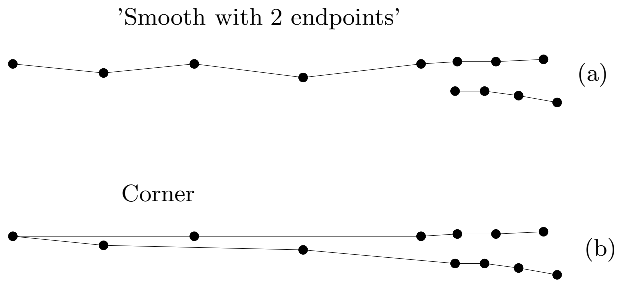


Figure 1.20: How to connect these samples ?

So our sampling condition for edges of the correct reconstruction *completely* inside a corner region is stated as follows:

Sampling Condition for Corner Areas: Let $e = (p, q)$ be an edge of the correct reconstruction completely inside a corner region, then we postulate (also see Figure 1.21):

1. the closed ball B_1 of radius $\beta \cdot |e|/2$ touching p and q from the 'outside' of the corner is empty of other samples
2. the closed ball B_2 of radius $\beta \cdot |e|/2$ touching p and q from the 'inside' of the corner may only contain other samples of the opposite leg ending in that corner (with the exception of the edge connecting the last two samples of each leg, whose inner β-ball may contain samples of both legs) which are inside the unshrunken corner ball
3. the turn between e and its adjacent edges in the correct reconstruction must be less than θ_{turn} (again with the exception of the edge connecting the last two samples

of each leg, but including the 'virtual' edges connecting those last two samples to the actual corner point)

4. the closed ball b_1 of radius $\beta_{low} \cdot |e|/2$ touching p and q from the 'outside' of the corner is empty of other samples

Intuitively, parts one, two, and three of this sampling condition ensure that the chains of the correct reconstruction that correspond to the legs in a corner area are rather straight, whereas part four makes sure that edges of different legs are well separated from each other unlike in Figure 1.20,(b).

One may find such a sampling condition rather artificial, and in fact, it is possible to formulate a more "natural condition", if we make the amount of "wiggling" (expressed in the angles θ_{turn} and θ_{slope}) dependent on the actual angle at the corner point like it is done in [AM00, Gie99]. The reason why we did not follow this approach is the following: Our algorithm at the end can only output edges which it can justify by presenting the appropriate witness curve (unlike the TSP-based reconstructions of [AM00, Gie99], as they do not treat the general case with several open curves). If we had set θ_{turn} and θ_{slope} to a fraction of the actual corner angle, this would mean for small corner angles that the incoming legs must be basically straight lines, which is not very likely to occur in practice. So in theory the algorithm would reconstruct corners, but in practice it would fail completely. As it turns out, with our choice of sampling condition, many corners are detected in practice.

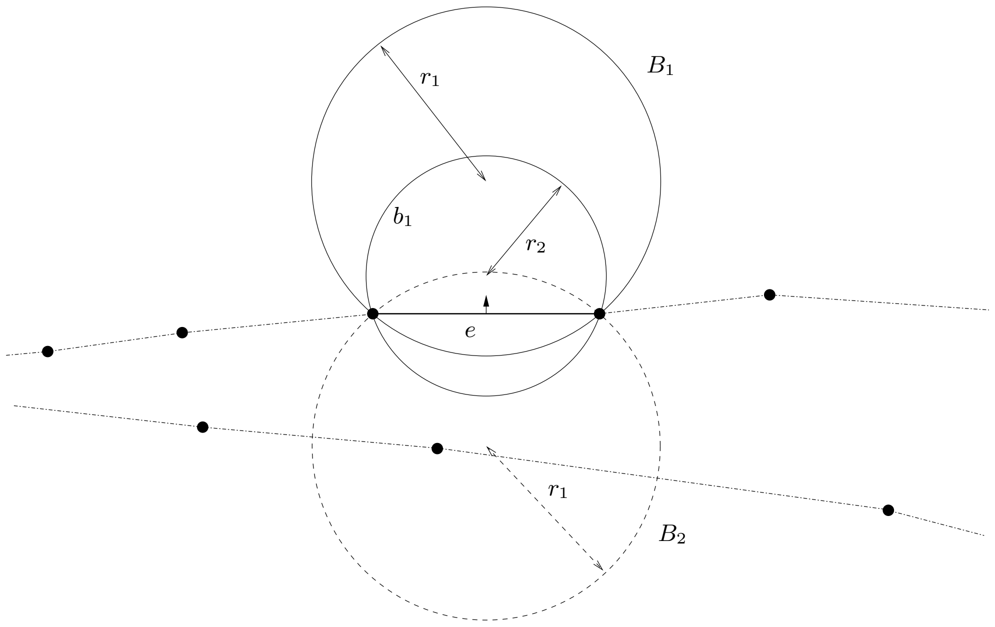


Figure 1.21: Sampling Condition for corner areas

We can now summarize our sampling condition.

General Sampling Condition: A sampling S for a collection of open and closed curves Γ

(possibly with corners) is valid if for any edge $e \in G(S, \Gamma)$

- Sampling Condition (smooth area) holds if e is (at least partly) outside a corner region
- Sampling Condition (corner area) holds if e is completely inside a corner region
- Any smooth component of the correct reconstruction consists of at least 3 samples

Figure 1.22 illustrates the unshrunken and shrunken corner balls in a collection of curves with corners and endpoints. Inside the solid circles, the sampling condition for corner areas must hold.

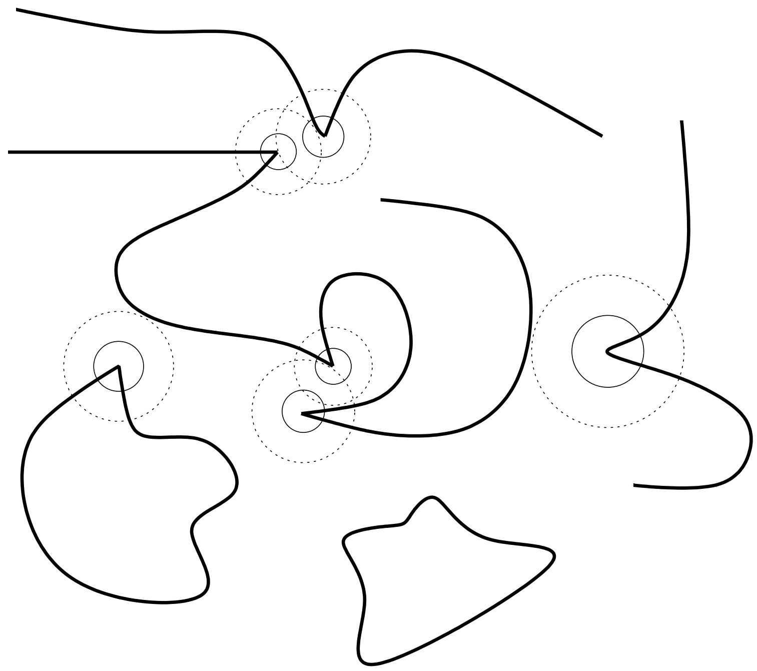


Figure 1.22: Corner balls for a collection of curves with corners and endpoints

Before we proceed, we list some notation and conventions, and make some basic observations that will be used throughout the paper.

Notation/Conventions

corner sample: We call the last samples of each leg *corner samples*. If there is a sample close to the actual corner, we may say that there is only one corner sample as this sample fits into both legs.

corner spanning edge: We call the edge connecting the last samples of each leg *corner spanning edge*.

smooth/corner area: We call the area in the shrunken corner ball *corner area*, the rest is called *smooth area*.

smooth/corner edge: We call an edge which lies completely inside a corner area a *corner edge*, the other edges are *smooth edges*

red/blue edge: We say an edge e is *red*, if it has two empty β-balls; we say e is *blue*, if it is not red, but has empty β- and β_{low}-balls on one side. (Remark: clearly all smooth edges must be red, but some of the corner edges and even some edges that do not belong to the correct reconstruction might also be red.)

In the figures, we draw a small normal arrow at the midpoint of an edge to indicate that it has β- and β_{low}-balls on that side. So every red edge has two such arrows, whereas every blue edge has only one arrow, as can be seen in Figures 1.17 and 1.21.

θ_{beta}**,** θ_{low}**:** To simplify notation we write θ_{beta} for $\arcsin \frac{1}{\beta}$ and θ_{low} for $\arcsin \frac{1}{\beta_{low}}$

Observations

Intersection of β_{low}**-balls:** Observe that for any blue or red edge, the intersection of the two touching β_{low}-balls is empty of samples, for $1 \leq \beta_{low} < \beta$.

Monotonicity of corner legs: Observe that from any point on one leg of a corner, the Euclidean distance to the other points on that leg grows when moving away from that point on the leg, if $\theta_{slope} < 90^0$.

Red chains: Observe that for $\beta \geq \sqrt{2}$ there cannot be a sample which has more than three adjacent red edges, as the angle between two adjacent edges must be larger than $\pi - 2 \cdot \theta_{beta}$ according to Lemma 7. For $\beta \geq 2$, there can be at most 2 adjacent red edges, i.e. the β-CRUST outputs a collection of chains.

Turn of smooth edges: We also call the turn angle between to adjacent edges the *turn* of these edges. For the turn between smooth edges we can give even a stronger bound than the one implied by the edges being red. Due to the sampling condition, we know that if $f_{diam} \geq 3/2$, the turn between two smooth edges is at most θ_{ball}, or in other words, the angle between two adjacent smooth edges is at least $\pi - \theta_{ball}$.

1.4.2 Medial axis sampling condition implies our sampling condition

In the following we will show that our sampling condition is weaker than one possible sampling condition based on the local feature size criterion, i.e. every collection of curves Γ which satisfies this sampling condition w.r.t. the local feature size also satisfies our sampling condition.

First we identify the corner areas in the same way as for our sampling condition but with $\theta_{slope} = \theta_{turn}$. We then define a new local feature size $\text{lfs}'(p)$ for any point p on the curve as follows:

- if p is not in any shrunken corner ball, $\mathrm{lfs}'(p) = \mathrm{lfs}(p)$, where $\mathrm{lfs}(p)$ denotes the distance to the medial axis of Γ
- if p is contained in a shrunken corner ball, we set $\mathrm{lfs}'(p)$ to the distance from p to the medial axis of the collection of curves obtained by removing the leg not containing p within the corner ball [2].

The sampling condition is then stated as follows:

Sampling Condition w.r.t. the medial axis :

- for every point p on the curves there must be a sample within distance $\epsilon \cdot \mathrm{lfs}'(p)$
- for any edge $e = (p, q)$ of the correct reconstruction inside the shrunken corner ball, the angle determined by e at some other sample t inside the corner ball is less than $\theta_{angle} = \pi - \theta_{low}$
- every component of the collection of curves must contain at least 3 samples

Before we can prove that this sampling condition w.r.t. the local feature size is stronger than our sampling condition, we need to prove a small auxiliary lemma.

Lemma 12 *Let s, t, u be three points in $\mathbf{R}^2$, $\pi/2 \leq \angle stu = \alpha \leq \alpha^*$. Then we have for the radius r of the circumcircle through s, t, u: $r \leq \frac{d(s,u)}{2 \cdot \cos(\alpha^*/2)}$.*

Proof. Look at the picture in Figure 1.23. When moving t on the circular arc between s and u, the angle at t remains constant according to Thales' theorem. So it suffices to consider the case where t lies on the bisectore between s and u. But then it is easy to see that $\cos(\alpha/2) = \frac{d(t,u)}{2 \cdot r}$ or $r = \frac{d(t,u)}{2 \cdot \cos(\alpha/2)}$. But as he have $\alpha \geq \pi/2$, clearly $d(t, u) < d(s, u)$ and hence $r \leq \frac{d(s,u)}{2 \cdot \cos(\alpha^*/2)}$.

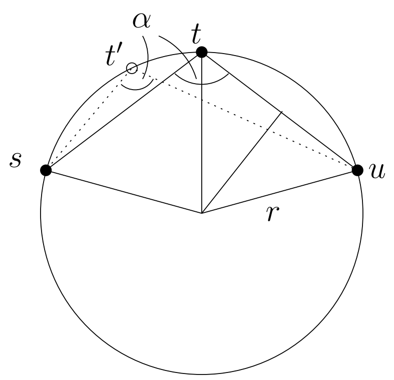


Figure 1.23: An application of Thales' theorem.

□

[2] We define the local feature size of the corner point as the minimum distance to either medial axis

Lemma 13 *Any sample set S valid with respect to the medial axis sampling condition is valid with respect to our sampling condition for*

$$\epsilon \le MIN(\frac{1}{4f_{diam} + \frac{2f_{diam}+1}{\sin(\theta_{ball}/4)} + 1}, \frac{1}{2\beta - 1})$$

Proof. We start with the sampling condition for smooth areas.

1. According to Lemma 8 every smooth edge has empty β-balls for $\epsilon \le 1/(2\beta - 1)$.

2. Now assume that although S is an ϵ-sample, the chain left or right of e connecting the samples inside the ball B_3 of radius $f_{diam} \cdot |e|$ centered at the midpoint of e makes a turn of more than θ_{ball}. We will show in the following that this implies the existence of three samples s, t, u in the diametral ball which make an angle of at most $\pi - \theta_{ball}/2$ at t. According to Lemma 12 this implies a Voronoi vertex at distance less than $d_v = f_{diam} \cdot |e| + \frac{(f_{diam}+1/2)\cdot|e|}{(2\cdot\sin(\theta_{ball}/4))}$ from p or q. But this Voronoi vertex implies that for p or q we have a local feature size less than $2 \cdot d_v$ (Lemma 5). Hence for For $\epsilon \le \frac{1}{4\cdot\frac{d_v}{|e|}+1}$ we get a contradiction to Lemma 2.

   Assume w.l.o.g. in the chain right of e there exist two edges $e_1 = (o, p)$ and $e_2 = (q, r)$ (one of them might be e itself) such that the angle α at the intersection x of the supporting lines of e_1 and e_2 is larger than θ_{ball}. Let e_1, e_2 be such that α is maximized. If one of the angles $\angle opq, \angle opr, \angle oqr, \angle pqr$ is smaller than $\pi - \theta_{ball}/2$ we are already done. Otherwise, e_1 and e_2 lie mutually in their cones with opening angle $2 \cdot \theta_{ball}$ as in Figure 1.24. Look at the triangle Δpxq. The angle at x inside the triangle is $\pi - \alpha$, but

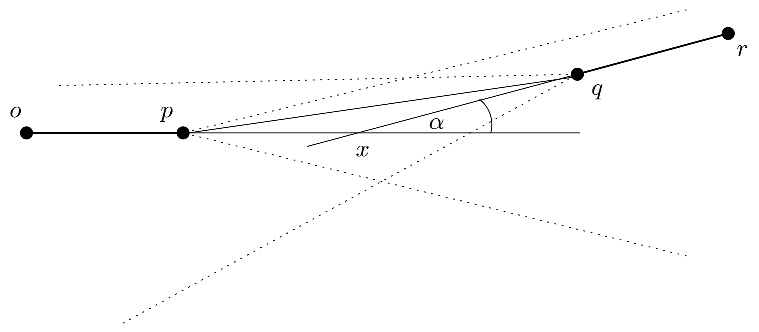


Figure 1.24: Finding s, t, u

   as the sum of the angles in triangle is π, either the angle at p or the angle at q must be larger than $\alpha/2$. As we have $\alpha > \theta_{ball}$, we obtain three samples – either o, p, q or p, q, r which make an angle smaller than $\pi - \theta_{ball}/2$.

Now we have a look at the sampling condition for corner areas.

1. According to Lemma 8, there exist empty β-balls touching edges of the corner from 'outside'.

2. When removing the other leg, the edges of the corner have empty β-balls on both sides, so when adding the other leg again, only samples from that leg can enter the β-ball touching from 'inside'.

3. Follows from the fact that we have identified the corner areas with $\theta_{slope} = \theta_{turn}$.

4. Assume an edge $e = (p, q)$ has no empty β_{low}-ball touching from 'outside'. Assume that 'inside' the corner is below e. Then it is clear that there can be no samples inside the β_{low}-ball above e as e has an empty β-ball touching from outside; so we concentrate on the part below e. But then notice that according to Thales' theorem for all points x on the boundary of the β_{low}-ball the angle at $\angle pxq$ is exactly $\pi - \arcsin 1/\beta_{low}$. For points *inside* the β_{low}-ball this angle is *larger*, a contradiction to the sampling condition.

$\square$

1.5 The algorithm

The main idea of our algorithm is that we first detect the edges that can be justified as being 'smooth'. Then starting from these edges we explore potential corner areas, possibly removing some of the edges previously discovered as smooth.

The following is a high-level description of our algorithm:

1. Compute the Delaunay triangulation $DT(S)$.
2. Extract the β-CRUST from $DT(S)$ and colour all edges of the β-CRUST *red*, which belong to red chains of length at least 3. Of the remaining edges colour those *blue* that have at least on one side empty β- and β_{low}-balls.
3. Let T be the set of samples which are adjacent to exactly *one* red edge
4. As long as there are elements in T, remove one of them and start exploration of a potential corner. If successful, add this corner to the set M of detected corners.
5. Remove interfering corners from M
6. Remove some of the red edges if they interfere with the corners in M to get $H(S)$.
7. Construct a collection of witness curves Γ' from $H(S)$ by adding small 'caps' and corners points

Of course, the most interesting part is how to actually explore potential corners. From now on, we assume that a corner is represented as two sequences corresponding to the upper and lower leg ending in that corner.

1.5.1 Exploring a corner

The idea of the corner exploration is that we consider a sample $s \in T$ as starting point of a potential corner. s is adjacent to exactly one red edge e_s, and we first try to continue this red edge with a blue edge. As we do not know the orientation of e_s, i.e. we do not know where the 'outside' of the potential corner is, we simply try both possibilities.

For this step and the steps to follow, a crucial part is how to determine the next edge in a leg. This is done using the following procedure (also look at Figure 1.25):

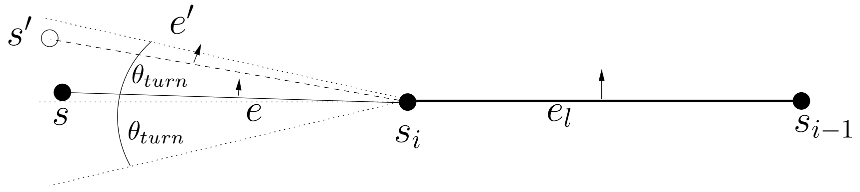


Figure 1.25: Picking the next edge of the potential corner

FindNextEdge(s_i, s_{i-1}): Let $e_l = (s_i, s_{i-1})$ be the last edge detected in one leg of a corner. Assume we are also given an orientation, i.e. we know where the 'outside' of the corner is (which implies that there are empty β- and β_{low}-balls on that side).

Let M be the set of all blue or red edges $e' = (s', s_i)$ which make a turn of less than θ_{turn} with e_l and which have empty β_{low}- and β-balls on the same side as the orientation of e_l.

As next edge $e = (s, s_i)$, pick the shortest among the edges in M.

Having found this adjacent blue edge e_b, we use the fact that this blue edge builds a Delaunay triangle with a sample o on the opposite leg (follows from the sampling condition). If o has no adjacent red edge, we abort the exploration, otherwise we pick one of the (potentially two) red edges adjacent to o – let's call it e_o which does not contain a sample already part of e_s or e_b. Then we follow both potential corner legs until finding the

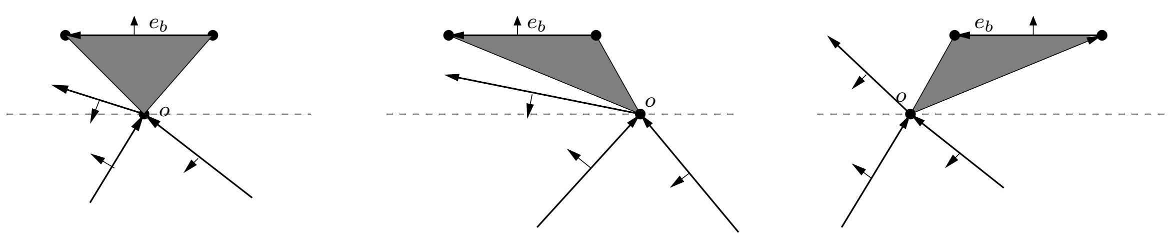


Figure 1.26: Choosing the orientation and direction of e_o

corresponding corner. The orientation of e_o, i.e. where with respect to e_o the 'outside' of the corner is, and the direction of e_o, i.e. in which direction the corner lies, is determined by the relative position of e_o with respect to e_b and o. Look at Figure 1.26 for the possible orientations and directions of e_b. Assume e_o is above the parallel line to e_b through o. If e_o lies to the left of the triangle formed by e_b and o, e_o points from o towards the corner point, if it lies to the right, this cannot be the start of a corner exploration. If e_o lies below the parallel line to e_b through o, o is 'on the way' from e_o to the corner point. The 'outside' of the corner is always to the left of e_o. There might be cases, where only one of the (potentially two) red edges adjacent to o leads to a correct corner exploration, so we try both possibilities[3].

Following the legs actually works step by step. We first determine for each of the two current reconstructed legs whether there exist potential continuation edges (using the **Find-NextEdge** procedure). We take the shorter of them and then verify whether the things reconstructed so far justify a corner. If so, we add it to the temporary set M' but nevertheless continue to follow the candidate legs until either

- for both sequences no continuations are found, or
- the two legs meet, or

[3]There is one case, where one of the red edges leads to an incorrect corner exploration, namely when one of them is a corner spanning edge.

- the angle between any two segments in one candidate leg is larger than θ_{slope}
- the cones of slope of the two legs intersect

When this procedure terminates, we take all explored corners in M' and add them to the set M of all potential corners (with all edges of the grown corner ball as determined in the verification stage which we will describe in the following).

1.5.1.1 Verifying a corner

The task of the verification stage, which is called after every continuation step, is to check whether the connections found so far make up a justifiable corner. To do this, we compute a tentative corner point and check whether there exists a large enough ball to cover all of the blue edges in our current reconstruction which is empty of other samples (samples that are not part of the corner to be verified).

Let s_u and s_l be the last samples in the upper and lower legs reconstructed so far, and let e_u and e_l be the corresponding last edges. To determine a tentative corner, we consider the two cones of angle $2 \cdot \theta_{turn}$ at s_u and s_l w.r.t. e_u and e_l. The tentative corner point C is determined as (in this order) – also see Figure 1.27:

- if s_u is contained in the other cone, set $C = s_u$ (and vice versa)
- if the cones do not intersect, the corner verification has failed
- if the cones intersect, take the 'inner' intersection point of their boundaries

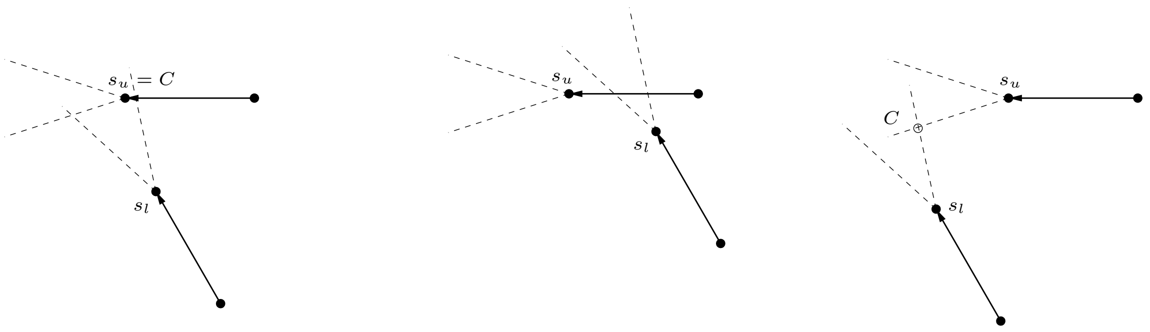


Figure 1.27: Determining the tentative corner point C.

We then determine the maximum distance D from C to any point in one of the sequences found so far which has a blue adjacent edge. We call the ball centered at C with radius D the *ungrown corner ball.* We extend the two candidate legs by red edges (if such exist) as long as the cone of slope (including the edges from the tentative corner point C to the last samples of the candidate legs) has angle less than $\theta_{slope} + \theta_{turn}$ in each leg and as long as the furthest point of each leg has distance less than $f_{grow} \cdot D$ from C. If no such red edges can be found, the verification fails, otherwise, we consider the ball around C with radius $f_{grow} \cdot D$ –

this ball is called the *grown corner ball.* If it contains other samples than the one present in the two sequences, the verification fails, otherwise we check that all blue edges only contain in their inner β_{low}-balls samples of the opposite leg within the grown ball.

1.5.2 Removal of interfering corners

Our algorithm by now has produced a collection of possible corners (represented as the two reconstructed legs ending in that corner) which might possibly interfere with each other, where we say that two corners interfere with each other, if the overlay of the corresponding graphs has a degree 3 vertex. We distinguish two kinds of interference: *overlap* and *intersection.* Two corners

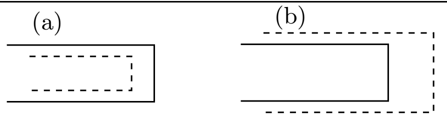


Figure 1.28: Cases of agreeing overlap

- *overlap* If the degree 3 vertices are only caused by (at most two) corner spanning edges which cross the interior of the other corner. We also distinguish between:

  agreeing overlap: if both corners point into the same direction, see Figure 1.28 for a schematic outline of these cases.

  disagreeing overlap: if the corners do not point into the same direction. We can exclude this case as the cones of slope cannot be non-intersecting in both directions.

- *intersect* if they interfere but do not overlap

We first get rid of the intersecting corners by just deleting any pair of corners that intersects each other. For the remaining overlapping ones, we always delete the corner that starts inside the other one, so in Figure 1.28, the dashed corners would be deleted. If two overlapping corners start at the same points, we delete the shorter of the two.

1.5.3 Removal of interfering red edges

At that stage we have identified a set of potential corners, but some of them might interfere with red edges found in the first step, i.e. some red edges might touch or cross a potential corner causing a degree 3 vertex. We will prove later on that these red edges cannot be part of the correct reconstruction, so we simply delete them.

1.6 Correctness of the algorithm

First we have to prove is that if a collection of curves Γ is sampled according to our sampling condition, then the correct reconstruction is part of the output graph $H(S)$ of our algorithm ('Good edges are captured'). It might produce some more edges, though, so the second thing we have to prove is the existence of a collection of curves Γ' for which S is a valid sampling (with a slightly weaker sampling condition) and $H(S)$ is the correct reconstruction of S with respect to Γ' ('Captured edges are good') like in [DMR99]. So Γ' is in some sense a certificate for the reasonability of each edge our algorithm has constructed.

Choosing appropriate parameters for the sampling condition and in the algorithm, for example

$\beta = 2 \Leftrightarrow \theta_{beta} = 30^0$	$\beta_{low} = \frac{2}{\sqrt{3}} \Leftrightarrow \theta_{low} \approx 60$
$\theta_{turn} = 10^o$	$\theta_{slope} = 30^o$
$\theta_{ball} = 30^o$	$f_{diam} \approx 2.84$
$f_{shrink} \approx 4.71$	$f_{grow} \approx 1.86$

we obtain our two main Theorems:

Theorem 2 *Let Γ be a collection of open or closed curves possibly with corners, S be a set of samples of that collection meeting our sampling condition. Then each edge of the correct reconstruction $G(S, \Gamma)$ is present in the graph H returned by our algorithm with the only possible exception of edges spanning a corner of a curve.*

Theorem 3 *For any input sample set S, our algorithm returns a graph H and a collection of curves Γ' such that S is a valid sample set for Γ' with:*

$\beta' = \beta$	$\beta'_{low} = \beta_{low}$
$\theta'_{turn} = \theta_{turn}$	$\theta'_{slope} = \theta_{slope} + \theta_{turn}$
$f'_{diam} = \frac{1}{2}$	$\theta'_{ball} = \theta_{ball}$
$f'_{shrink} = \frac{f_{grow}}{\sqrt{1-\frac{1}{\beta^2}}}$	

and $H = H(S, \Gamma')$, i.e. H is the correct reconstruction of S with respect to Γ'.

Remarks

One might wonder why Theorem 2 excludes corner spanning edges of $G(S, \Gamma)$ in the guarantee for the output of our algorithm. One reason for this is, that if the two legs of a corner are sampled so densely that there is no blue edge, our algorithm cannot find a starting point for the corner exploration, so it finds all edges except for the corner spanning edge, see Figure 1.29. Observe that this also makes sense, because if the two legs are very densely sampled, it may well be the case that the original curve does not have a corner there but just two endpoints. We will need this 'conservative' behaviour of our algorithm later on when we modify the

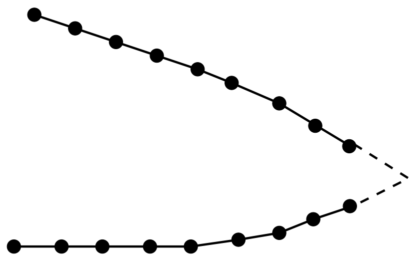


Figure 1.29: Densly sampled legs of a potential corner

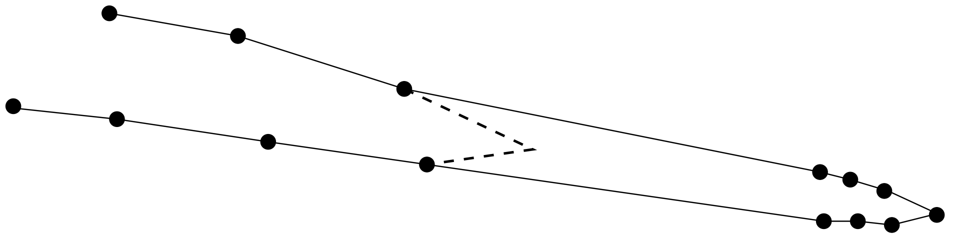


Figure 1.30: 'Extended' corner

sampling condition and our algorithm to get a result of the type: *For every collection of curves with corners and endpoints, there exists a sampling such that our algorithm outputs exactly the correct reconstruction.*

The other reason for the corner spanning edge not being detected is that our algorithm managed to extend the corner by further samples (which either form themselves a real corner or some smooth part), see Figure 1.30. Here the actual corner would be the dashed one, but the algorithm was able to justify the extended corner which also includes a group of additional samples (which, of course, have to be outside the unshrunken corner ball of the correct corner). Note that this can only happen for very small corner angles and is due to our relaxed sampling condition which uses a constant for θ_{turn} independent of the angle at the actual corner. Later we will show how to exclude that case as well without sacrificing the constant θ_{turn} angle.

1.6.1 Good edges are captured (Theorem 2)

Let $G(S, \Gamma)$ be the correct reconstruction of S with respect to a collection of curves Γ. We will show in the following that if S is a valid sampling of Γ, then every edge of of $G(S, \Gamma)$ will be detected and 'survive' throughout the course of the algorithm, and therefore be present in the output H of our algorithm (with exception of some corner spanning edges).

The following lemma does not require proof:

Lemma 14 *Smooth edges are detected and colored red by the algorithm after the first 2 steps.*

What we have to prove now is that smooth edges will not be killed later on because of interference with a potential corner:

Lemma 15 *A smooth edge cannot:*

1. *'touch' a (wrong) corner from outside at a sample which is not a corner sample*

2. *'touch' a corner at a corner sample*

3. *'cross' an incorrectly detected corner*

if the following holds:

- $\theta_{turn} \leq \pi - 4 \cdot \theta_{beta}$
- $f_{diam} \geq 3/2$
- $\theta_{ball} \leq \theta_{low}$
- $f_{grow} \geq 1 + \frac{2}{f_{diam} - \frac{1}{2}}$
- $\theta_{slope} \leq \pi - 2 \cdot \theta_{beta}$
- $\theta_{ball} \leq \frac{\pi}{2} - \frac{\theta_{turn}}{2}$
- $\theta_{turn} \leq \theta_{low} - \theta_{ball} - 2 \cdot \arctan \frac{1}{2 \cdot f_{diam}}$

Proof.

1. Consider the three adjacent samples s_0, s_1, s_2 on one leg ending in a (wrong!) corner detected by the algorithm as in Figure 1.31. Assume the smooth edge $e = (p, s_1)$ touches the corner at s_1 from outside. e must have empty β-balls on both sides. Hence we know that $\alpha_1 > \pi - 2 \cdot \theta_{beta}$ and $\alpha_2 > \pi - 2 \cdot \theta_{beta}$. As the turn between (s_0, s_1) and s_1, s_2 is at most θ_{turn}, this leads to a contradiction for $\theta_{turn} \leq \pi - 4 \cdot \theta_{beta}$.

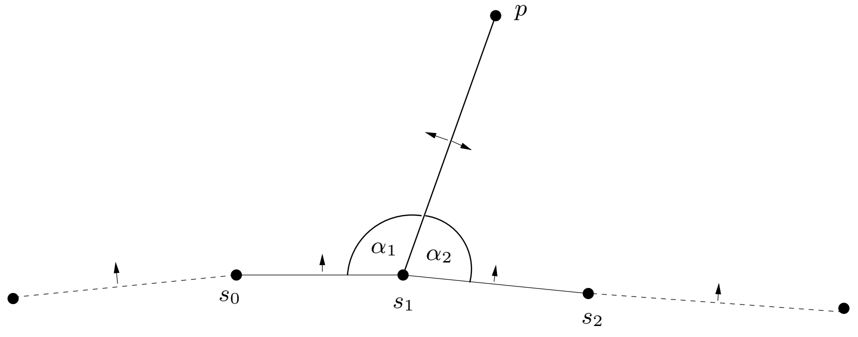


Figure 1.31: Smooth edge touching wrong corner from outside at non-corner sample

2. Let $e = (x, o)$ be a smooth edge touching an incorrectly detected corner c at a corner sample o as in Figure 1.32. Let q be the next sample on the same (wrong) leg as o, p the first sample on the other leg ending in c. Furthermore let r be the radius of the ungrown corner ball of the incorrect corner. Clearly, $|e| \geq (f_{grow} - 1) \cdot r$. Hence for $f_{grow} \geq 1 + \frac{2}{f_{diam} - \frac{1}{2}}$ and $f_{diam} \geq 3/2$, the ungrown ball is completely contained in the diametral ball around e, in particular p and q are both contained in the diametral ball.

It remains to show that it is not possible that they are connected in a single chain within that center ball. But this can easily be seen by considering the triangle $\triangle opq$. Due to the β_{low}-balls, we have $\alpha, \gamma < \pi - \theta_{low}$. Hence for $\theta_{ball} \leq \theta_{low}$ it is obvious that there is no way that o, p, q are connected within the center ball in one single 'flat' chain. As if the chain first visits p, it would have to make a turn of at least θ_{low} to get to q and vice versa.

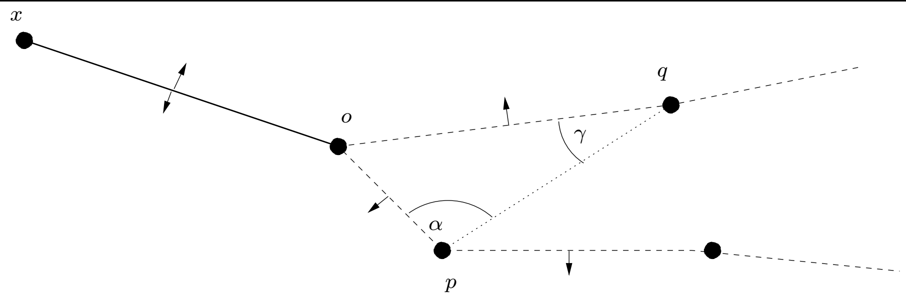


Figure 1.32: Smooth edge touching wrong corner at corner sample

3. We consider two adjacent smooth edges $e_1 = (p, q)$ and $e_2 = (q, r)$. If one of them touches the corner from outside, we are already done and can use the proofs of the first two cases. Otherwise, these two edges either jump from one side of the corner to the other and back, or one of them overlaps with the corner and the other crosses the corner.

   (a) Assume p and r lie on the same leg whereas q lies on the opposite leg as in Figure 1.33. First assume that there is no sample between p and r. But then we know that due to the β_{low}-ball around (p, r), $\delta < \pi - \theta_{low}$. On the other hand, the turn between (p, q) and (q, r) can be at most θ_{ball} if $f_{diam} \geq 3/2$, i.e. $\delta > \pi - \theta_{ball}$, which gives a contradiction for $\theta_{low} \geq \theta_{ball}$.

   If there are samples between p and r, though, we claim that there is no way that p and r can be linked by edges which make turn of less than θ_{slope} in total. To see that, let p_1 be the sample adjacent to p on the way to r. Clearly $\alpha < \theta_{beta}$, as well as $\gamma < \theta_{beta}$. Hence to get from p_1 to r one would need a turn larger than $\pi - 2 \cdot \theta_{beta}$ which gives a contradiction for $\theta_{slope} \leq \pi - 2 \cdot \theta_{beta}$.

   (b) Assume that e_1 overlaps with the (incorrectly) detected corner, and e_2 crosses the corner. The sample s which is adjacent to q in the (wrong) corner, must be outside the β-balls around e_2 as well as within a cone of width $2 \cdot \theta_{turn}$. Look at Figure 1.34, especially at the triangle Δqsr and the angle $\delta = \angle srq$. Due to the sampling condition we know that $\alpha \leq \theta_{ball} + \theta_{turn}$.

   We want to show that δ must be large and hence contradicting the empty β_{low}-balls for $e' = (q, s)$. To get to a contradiction, consider the diametral ball of radius $f_{diam} \cdot |e_2|$ around e_2.

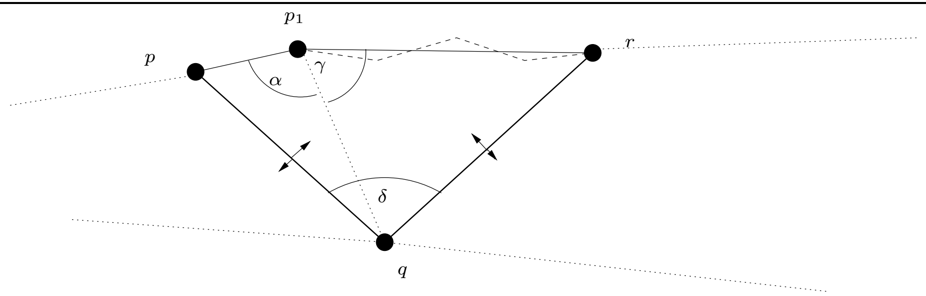


Figure 1.33: Smooth edges zig-zag-crossing wrong corner

Assume first s lies within the diametral ball around e_2 of radius $f_{diam} \cdot |e_2|$. Then either α or δ must be larger than $\pi - \theta_{ball}$. In the latter case, if $\theta_{low} \geq \theta_{ball}$, we have a contradiction as due to the β_{low}-ball around e', $\delta < \pi - \theta_{low}$ must hold. In the former case where α is very large, we would get a contradiction to the maximum turn angle for $\theta_{ball} \leq \frac{\pi - \theta_{turn}}{2}$.
Now assume that s lies outside the diametral ball. Then we have $\gamma \leq 2 \cdot \arctan \frac{1}{2 \cdot f_{diam}}$. As we also know that $\alpha \leq \theta_{ball} + \theta_{turn}$, we get $\delta \geq \pi - \theta_{ball} - \theta_{turn} - 2 \cdot \arctan \frac{1}{2 \cdot f_{diam}}$, which leads to a contradiction for $\theta_{turn} \leq \theta_{low} - \theta_{ball} - 2 \cdot \arctan \frac{1}{2 \cdot f_{diam}}$.

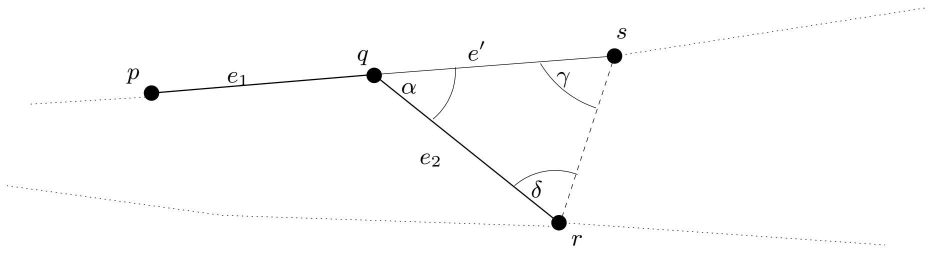


Figure 1.34: Smooth edges overlapping and crossing wrong corner

$\square$

So now we know that every smooth edge of the correct reconstruction will survive the stages of our algorithm and hence be present in the output. Let us now consider the non-smooth edges of one particular corner. We will show that there is a canonical element $s \in S$ such that if the algorithm starts a corner exploration from s, it will detect a potential corner which covers all the edges of the real corner we are considering.

To prove this we first have to state a small lemma which implies that if we are given a correct part of the sequence of samples on either leg (together with an orientation where the 'outside' is), our procedure `FindNextEdge()` will find the next edge of this leg (if it exists).

Lemma 16 *Let $e = (s_1, s_0)$ be an edge of the correct reconstruction within a corner area, oriented such that the 'outside' of the corner lies to the left of $\overrightarrow{s_1 s_0}$, and let s be the other* *sample which s_1 is adjacent to. Assuming that s lies on the same leg as s_1 and s_0, then there is no other sample s' such that the following conditions all hold at the same time:*

- *the turn-angle between e and $e' = (s', s_1)$ is less than θ_{turn}*
- *$d(s_1, s') \leq d(s_1, s)$*
- *there is an empty β-ball to the left of $\overrightarrow{s', s_1}$*

if we have $\theta_{turn} \leq \frac{\theta_{low} - \theta_{beta}}{2}$

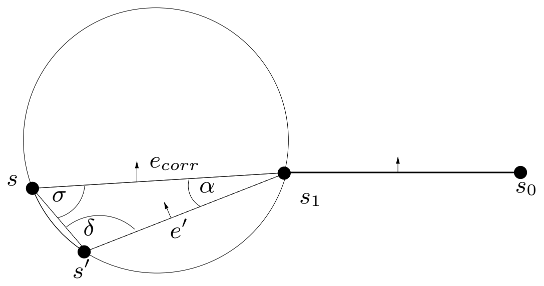


Figure 1.35: Only one correct edge is possible

Proof. Look at Figure 1.35 and assume the contrary. Let $e' = (s', s_1)$ be the 'wrong' edge, $e_{corr} = (s, s_1)$ the 'correct' edge. Note that s' must be on the other leg ending in that corner, if it had been on the same leg or not part of this corner area, we would have $e' > e_{corr}$. We consider the triangle $\triangle s_1 s s'$. Clearly $\alpha = \angle s' s_1 s \leq 2 \cdot \theta_{turn}$, as both e_{corr} and e' make a turn-angle of less than θ_{turn} with e. Let $\delta = \angle s s' s_1$ and $\sigma = \angle s' s s_1$. We have $\sigma = \pi - \alpha - \delta$, and $\sigma \leq \frac{\pi}{2}$, as $|e'| \leq |e_{corr}|$. We want to compute an upper bound for the ratio $\frac{r}{|e'|}$ where r is the radius of the circumcircle of $\triangle s_1 s s'$. A simple observation shows $\frac{r}{|e'|} = \frac{1}{2 \cdot \sin \sigma}$. To get a contradiction we have to show that $\beta > \frac{1}{\sin \sigma}$, or in other words $\sigma > \theta_{beta}$ (i.e. the ball around e' is not large enough). If we make sure that $\delta < \pi - \theta_{beta} - 2 \cdot \theta_{turn}$ we are done.

For a bound on this δ observe that due to the sampling condition, $\delta < \pi - \theta_{low}$. Hence we get as final condition $\theta_{turn} \leq \frac{\theta_{low} - \theta_{beta}}{2}$ □

Now we know that if we have somehow managed to find the right 'start' of the corner, i.e. correctly determined the start of the two legs ending in that corner, there will be a time during the algorithm's execution where exactly the correct edges of the corner have been detected (except for the corner spanning edge). This can be easily seen by the fact that we always extend with the shorter continuation edge and therefore first all edges within the corner area are picked before connecting to outside the corner area.

To show that for every corner there exists a good starting point for the exploration, follow both legs of the corner from the intersection of the unshrunken corner ball to the corner point until hitting the first Delaunay edge crossing the inside of the corner which is part of a triangle with a blue edge of the correct reconstruction. The red edge adjacent to this blue edge is clearly a good starting point for the corner exploration. We call this the *canonical corner exploration.*

Now we have to prove that at the point when exactly the correct edges have been detected, the verification test will pass.

Lemma 17 *At the time when the correct edges of a 'real' corner have been detected, the verification test will pass for* $f_{grow} \leq (f_{shrink} - 1)/2$.

Proof. Consider a real corner with corner point C and its shrunken corner ball of radius R. Clearly the tentative corner c computed by the algorithm lies inside the shrunken corner ball and the radius r of the ungrown corner ball around the tentative corner c (which has to cover all blue edges in the two legs) is at most $2 \cdot R$. As the shrunken corner ball has been shrunken by a factor of f_{shrink}, the samples which are not part of the two chains making a total turn of at most θ_{slope}, have distance at least $f_{shrink} \cdot R$ from C and hence distance at least $(f_{shrink} - 1) \cdot R \geq (f_{shrink} - 1) \cdot r/2$ from c. So we must have $f_{grow} \leq (f_{shrink} - 1)/2$. $\square$

So now we know that for each corner of the correct reconstruction, there is at least one reconstruction (namely the correct reconstruction) in M which covers all edges of that corner. It remains to show that for every corner, the correct reconstruction or an 'extension' of it covering all edges survives the next stages. For this we will first make a simple observation.

Observation[**Detour**] Let $e = (p, q)$ be a red or blue edge. Then there is no path in the Delaunay triangulation $DT(S)$ of a point set S from p to q which makes a turn of less than θ_{slope} for $\theta_{slope} \leq \theta_{low}$.

Proof. Let p_2, p_2 be the samples which build triangles with e in $DT(S)$. As e is a red or blue edge, we know that the intersection of the two β_{low}-balls touching p and q is empty of samples. Hence for the angles α_1, α_2 at p_1 and p_2 in the respective Delaunay triangles we have $\alpha_i \leq \pi - \theta_{low}$. So any chain going from p to q would have to get around the quadrilateral pp_1qp_2, which requires a turn of at least θ_{low}. $\square$

Lemma 18 *The reconstruction of a correct corner cannot be intersected by the reconstruction of a 'wrong' corner for*

- $\theta_{turn} \leq \frac{\theta_{low} - \theta_{beta}}{2}$
- $\theta_{turn} \leq \frac{\theta_{low}}{4}$
- $\theta_{slope} \leq \theta_{low} - 3 \cdot \theta_{turn}$

Proof. We restrict to the area of the shrunken corner ball, as outside the shrunken ball all edges of the correct corner must be 'smooth' and hence the proofs for 'smooth edges are not intersected by wrong corners' can be used. We will distinguish several cases, always assuming that if we are in one particular case, all the cases before did not apply:

1. Consider the case that a 'wrong' edge e' touches at a sample p of either leg of the correct reconstruction but *not* a corner sample. Clearly, e' cannot come from another sample of the same leg as this would be a detour. So it has to come either from outside the unshrunken corner ball or from the other leg crossing the inside of the corner.

   - Assume e' comes from 'outside'. Due to the shrinking process, e' has length at least $(f_{shrink} - 1)$-times the length of any edge in the shrunken corner ball. Let q and s be the adjacent samples of p in the correct reconstruction. Look at Figure 1.36. q must be adjacent to p in the wrong corner reconstruction as it has to be contained in the corner ball of the wrong corner but any indirect way to p would be a detour. Note that if p was a corner sample, it would be impossible to cover all samples within the corner ball of the correct corner with the other branch. So

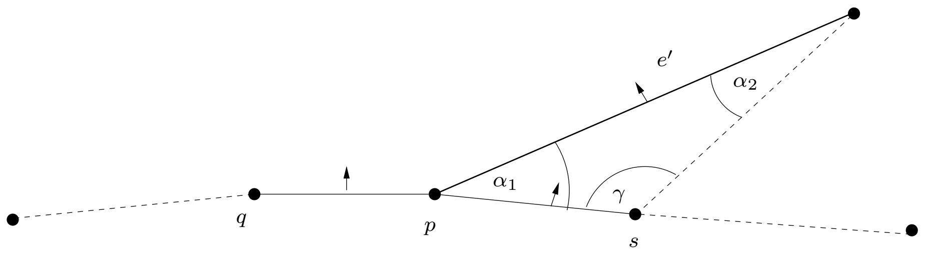


Figure 1.36: Wrong edge touching corner inbetween

     the turn angle between (q, p) and e' is less than θ_{turn}, and hence $\alpha_1 \leq 2 \cdot \theta_{turn}$. On the other hand, $\alpha_2 < \theta_{beta}$ due to the β-ball touching (p, s) from outside. So $\gamma > \pi - \theta_{beta} - 2 \cdot \theta_{turn}$. But with $\gamma < \pi - \theta_{low}$ due to the β_{low}-ball around e', we get a contradiction for $\theta_{turn} \leq \frac{\theta_{low} - \theta_{beta}}{2}$.

   - Consider the case that $e' = (p, q)$ crosses inside the correct corner like in Figure 1.37, but e' is not a corner spanning sample, i.e. not both q and p are corner samples of the wrong corner [4]. Let s and o be the adjacent samples of p in the correct corner, r and t the adjacent samples of q, respectively.

     Clearly e' cannot be the only edge of the wrong corner, so p or q must have another adjacent edge. First assume that q has another adjacent edge e''. Clearly e'' cannot end in some sample outside the corner ball (covered by the previous case). On the other hand it cannot touch any sample of the other branch from 'outside' due to the cone of slopes constraint in the sampling condition. So q has to be connected to either r or t or some sample on the opposite leg left of p (right of p is not possible due to the same argument as in Figure 1.33).

     If q is not a corner sample of the wrong corner and is connected to t or some sample on the opposite leg, clearly $\alpha \leq 2 \cdot \theta_{turn}$. And as e' has a β-ball to the outside, $\delta < \theta_{beta}$. Hence $\gamma > \pi - 2 \cdot \theta_{turn} - \theta_{beta}$ which gives a contradiction for

[4] If e' is a corner spanning sample of the wrong corner, we have the case of agreeing or disagreeing overlap, which is treated separately.

$\theta_{turn} \leq \frac{\theta_{low}-\theta_{beta}}{2}$, as we should have $\gamma > \pi - \theta_{low}$ due to the β_{low}-ball around (q, r). Now assume q is not a corner sample and is connected to r. If p is also not a corner sample, it has to be connected to s and we get $\sigma \leq 2 \cdot \theta_{turn}$ and $\tau + \kappa \leq 2 \cdot \theta_{turn}$. So of course, $\kappa \leq 2 \cdot \theta_{turn}$. But then we have $\chi \geq \pi - 4 \cdot \theta_{turn}$, which contradicts to $\chi < \pi - \theta_{low}$ due to the β_{low}-ball around e' when choosing $\theta_{turn} \leq \frac{\theta_{low}}{4}$. So in this case, p has to be a corner sample and we have the a symmetric case of the following.

W.l.o.g. we can assume that q is a corner sample. So p must have another adjacent sample, which cannot be on the same side as q, as we could use the same argument as for the smooth ZigZag case. On the other hand it cannot connect to any sample other than s or o as we would then have a detour. But as p is not a corner sample (because q is and we assume that not both are corner samples), it must be adjacent to s. So $\sigma \leq 2 \cdot \theta_{turn}$. We also know that $\kappa < \pi - \theta_{low}$ and hence $\chi > \theta_{low} - 2 \cdot \theta_{turn}$ and $\epsilon < \pi - \theta_{low} + 3 \cdot \theta_{turn}$.

For $\theta_{turn} \leq \frac{\theta_{low}-\theta_{beta}}{2}$ and because of $\chi > \theta_{low} - 2 \cdot \theta_{turn}$, we know that o must be contained in the other β-ball of e', hence o must be part of the opposite leg of e' (in the wrong corner). Clearly o must have two adjacent edges in the wrong corner, both of which have to be in the angle between uo and oq which has angle ϵ (due to the detour and visibility properties). Let u' be the sample adjacent to o which is further away from q. The edges of that leg must make a path from u' to q without turning more than θ_{slope} to the left (note that this holds even if there is another corner sample on the way to q). But this leads to a contradiction for $\theta_{slope} \leq \theta_{low} - 3 \cdot \theta_{turn}$.

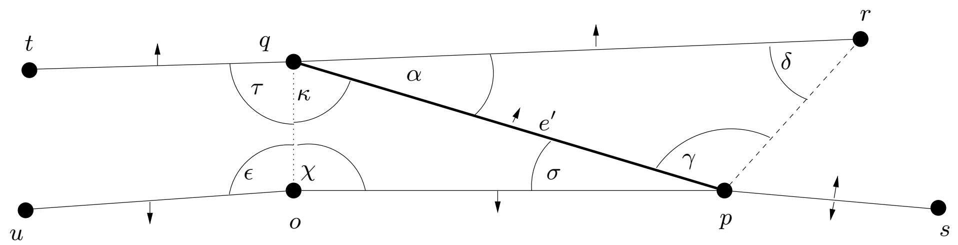


Figure 1.37: Wrong edge crossing corner inside

2. Consider the case that exactly one edge touches the correct corner from outside at a corner sample. When looking at the shrunken corner ball of the correct corner, it is clear that it must be completely contained in the grown corner ball of the wrong corner, so all samples must be part of the 'wrong' corner. On the other hand the intersection of a corner with any ball must have an even number of intersections on the boundary of the ball, which is not the case here.

3. Consider the case that two edges touch the correct corner from outside at the same corner sample c. Look at Figure 1.38. Again, observe that the whole shrunken corner

ball must be contained in the grown corner ball of the wrong corner, hence the other samples in the correct corner must all be connected in one single leg. Let p and q be the samples adjacent to c in the correct corner reconstruction, p on the same leg as c. It is easy to see that the wrong corner must connect the samples exactly the same way as the correct reconstruction of the corner, but with a shortcut from p to q leaving out c as this is already used. Any other case would imply a detour which cannot happen. Looking at the triangle $\triangle cpq$, it is then obvious that the angles at p and q must be $\leq 2 \cdot \theta_{turn}$. On the other hand, as p and q are connected by an edge, the angle at c must be less than $\pi - \theta_{low}$, giving a contradiction for $\theta_{turn} \leq \frac{\theta_{low}}{4}$

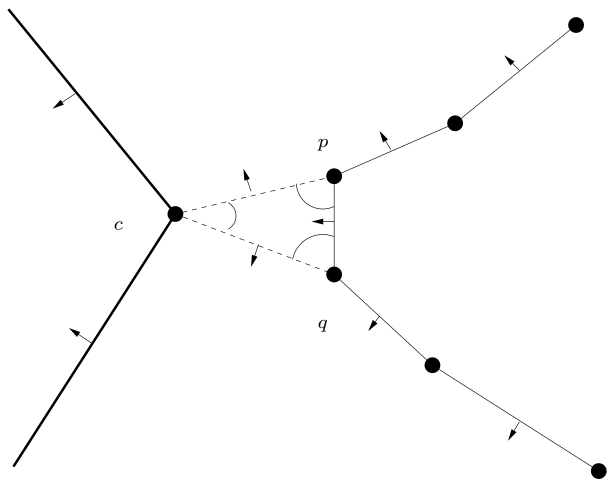


Figure 1.38: Two long edges touching at one corner sample

4. Consider the case that two edges touch the correct corner from outside at two different corner samples c_1 and c_2. Two cases can be distinguished:

   - c_1 and c_2 are not adjacent in the 'wrong' corner. But then clearly, c_1 and c_2 cannot be part of the same leg of the wrong corner, as this 'other' path from c_1 to c_2 would be a detour.

     Now look at how the branch going through c_1 continues:

     – if it follows the upper leg of the correct corner until leaving, the other branch must be covered by the other leg and we have an overlap case

     – if it jumps somewhere 'outside', we have already covered that case

     – if it jumps inside from some sample s_1 to the other leg, it cannot jump back to a sample s_2 on the same leg and adjacent to s_1 as we would have a detour then. If s_2 is on the same leg as s_1 but not adjacent to s_1, we would have 'separated' some samples between s_1 and s_2 which cannot be connected to c_2. So after jumping to the other side, the leg has to follow the other leg until leaving the corner ball, hence separating the rest of the upper leg from c_2 again.

- c_1 and c_2 are adjacent in the wrong corner. Let d_1, d_2 be the next samples on each of the legs, as in Figure 1.39. As no detour is allowed, the other leg of the wrong corner must follow exactly the correct corner, but with a shortcut (d_1, d_2) as c_1 and c_2 are already covered. Look at the triangle $\Delta d_2 d_1 c_1$. For the angles at d_1 and d_2 we get upper bounds of $2 \cdot \theta_{turn}$. On the other hand, as d_1, d_2 are connected by an edge with β- and β_{low}-balls, the angle at c_1 must be smaller than $\pi - \theta_{low}$, giving a contradiction for $\theta_{turn} \leq \frac{\theta_{low}}{4}$.

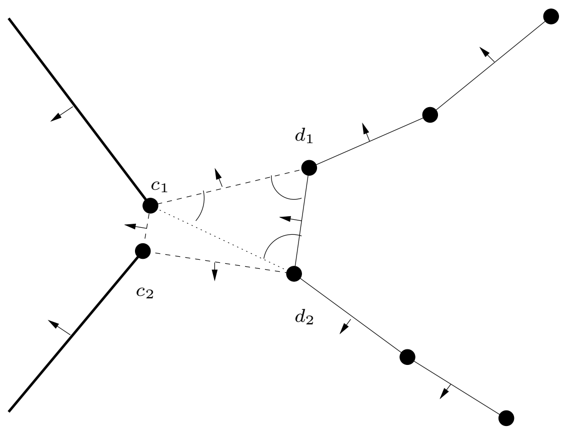


Figure 1.39: Two long edges touching at two different corner samples

$\square$

Lemma 19 *At least one corner exploration covering (at least) all edges of a corner in the correct reconstruction survives all stages of the algorithm if we have $f_{shrink} \geq 2$.*

Proof. We have already proven that the canonical corner exploration cannot be intersected by a wrong corner, so it definitely survives the 'removal of intersecting corners' stage. It remains to show that if a corner exploration E_1 covering all edges of a real corner is killed by another corner exploration E_2 in the 'removal of overlapping corners' stage, E_2 also covers all the edges of the real corner. Clearly case (a) in Figure 1.28 fulfills our requirement. So we have to look at case (b). Assume the dashed – 'good' – corner exploration E_1 covers all edges of the correct corner, but is killed by the other – 'bad' – corner exploration E_2 which does not cover all edges of the correct corner.

Observe, that E_2 has to start outside the unshrunken corner ball of the correct corner, as all edges which are in the unshrunken, but not in the shrunken corner ball of the correct corner have to be red and a corner exploration can only start at blue edges. Furthermore, the corner spanning edge must be inside the shrunken corner ball of the correct corner (and therefore also in the ungrown corner ball of E_1), as otherwise the 'smooth edge cannot touch wrong corner' case applies. So the radius of the ungrown corner ball of E_2 must be at least $(f_{shrink} - 1) \cdot r$, where r is the radius of the ungrown corner ball of the correct corner.

So for $f_{shrink} \geq 2$, all samples of the correct corner must be contained in the ungrown corner ball of E_2, which gives a contradiction. The case where the two overlapping corners start at the same points and the 'shorter' one is deleted does not require any discussion. $\square$

To sum up, we have proven that every edge of the correct reconstruction is detected by the algorithm and survives till the end. So 'all good edges are captured'.

1.6.2 Captured edges are good (Theorem 3)

Basically almost all statements of Theorem 3 follow directly from the algorithm. The only statement that requires a proof is the statement that for every corner, the ball of radius $r \cdot \sqrt{1 - \frac{1}{\beta^2}}$ does not intersect any segments of the output graph which are not part of the two legs ending in that corner, where r is the radius of the grown corner ball. But this can be easily seen, as the distance of any edge e to the center of the grown corner ball must be greater than $r \cdot \sqrt{1 - \frac{1}{\beta^2}}$, since this e must have an empty β-ball on that side, hence the ball of radius $r \cdot \sqrt{1 - \frac{1}{\beta^2}}$ cannot intersect any of these segments.

1.6.2.1 Construction of a collection of Witness Curves Γ'

As our sampling condition does not directly refer to the curvature of the curve, we can construct witness curves by simply taking the polygonal reconstruction computed by our algorithm and adding very small 'caps' at every sample which is adjacent to two non-corner spanning edges. Corner spanning edges are replaced by two edges to the corner point estimated by the algorithm.

Using the same idea as in [DMR99], one could prune the output of our algorithm even further and then construct curves as in [DMR99], which are then witness for the sampling condition with respect to the medial axis, though we have not elaborated this further.

1.6.3 Setting the Parameters for the Algorithm and the Sampling Condition

The proofs of this section have led to the following constraints on the constants of the sampling condition and the algorithm:

$$\theta_{turn} \leq \pi - 4 \cdot \theta_{beta}$$
$$\theta_{turn} \leq \frac{\theta_{low}}{4}$$
$$\theta_{slope} \leq \pi - 2 \cdot \theta_{beta}$$
$$f_{diam} \geq \frac{3}{2}$$
$$f_{grow} \geq 1 + \frac{2}{f_{diam} - \frac{1}{2}}$$
$$f_{shrink} \geq 2$$
$$\theta_{turn} \leq \theta_{low} - \theta_{ball} - 2 \cdot \arctan \frac{1}{2 \cdot f_{diam}}$$
$$\theta_{turn} \leq \frac{\theta_{low} - \theta_{beta}}{2}$$
$$\theta_{low} \geq \theta_{ball}$$
$$\theta_{slope} \leq \theta_{low} - 3 \cdot \theta_{turn}$$
$$\theta_{ball} \leq \frac{\pi - \theta_{turn}}{2}$$
$$f_{grow} \leq \frac{f_{shrink} - 1}{2}$$

In the following we compute a possible setting of this parameters which we consider reasonable. β should preferably be chosen as small as possible to keep the required sampling density especially in the smooth parts low. Its value should be at least 2, though, as we want to make sure that no sample is adjacent to more than 2 red edges, hence we fix $\beta = 2$, i.e. $\theta_{beta} = 30^o$. β_{low} must be chosen smaller than β, but as large as possible to be not too restrictive in the sampling condition for the corner areas. Let us fix $\beta_{low} = \frac{2}{\sqrt{3}}$, i.e. $\theta_{low} = 60^o$.

With this choice of β and β_{low} we get an upper bound of 15^o for θ_{turn}. But as we need some 'slack' for the other parameters as well, we choose $\theta_{turn} = 10^o$ and with that information we choose $\theta_{slope} = 15^o$. It remains to determine values for θ_{ball}, f_{diam}, f_{grow} and f_{shrink}. A balanced choice for these parameters could be $\theta_{ball} = 30^o$, $f_{diam} \approx 2.84$, $f_{grow} \approx 1.86$, $f_{shrink} \approx 4.71$

So finally we end up with the following choice of parameter values:

$\beta = 2 \Leftrightarrow \theta_{beta} = 30^o$	$\beta_{low} = \frac{2}{\sqrt{3}} \Leftrightarrow \theta_{low} = 60^o$
$\theta_{turn} = 10^o$	$\theta_{slope} = 30^o$
$\theta_{ball} = 30^o$	$f_{diam} \approx 2.84$
$f_{shrink} \approx 4.71$	$f_{grow} \approx 1.86$

The reader is encouraged to check that these values satisfy all the constraints.

1.7 Further Results

1.7.1 How to obtain *exactly* the correct reconstruction ?

The algorithm as outlined so far guarantees that all edges of the correct reconstruction G of the original collection of curves Γ are present in the output H of our algorithm, but there might be additional edges in the output, though our algorithm can 'justify' each of them.

The ultimate goal, of course, is to find a sampling condition and an algorithm which for any collection Γ of curves with endpoints and corners guarantees that the output of the algorithm is exactly the correct reconstruction $G(S, \Gamma)$, if the sample set S conforms to this sampling condition.

To reach this goal, one first has to restrict the possible curves allowed in Γ. Given any sample set S, it is always possible that the original collection of curves Γ consists exactly of those sample points, i.e. each curve in Γ is degenerate in a sense that it only consists of a single point. There is no way the algorithm can detect this. So it seems reasonable to restrict the curves allowed in Γ to be non-degenerate, i.e. each curve must consist of more than just one point.

The second difficulty arises in cases as sketched in Figure 1.40. With the current sampling condition and algorithm, the algorithm would connect p and q even if they are endpoints in the correct reconstruction, provided there are no samples in the neighbourhood which are inside the β-balls of (p, q).

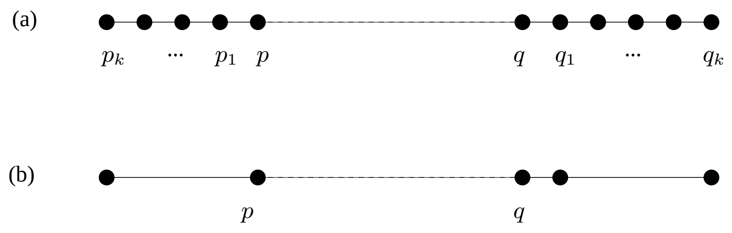


Figure 1.40: Connect p and q or not ?

Note that in Figure 1.40,(a) it really seems reasonable *not* to connect p and q as the edge (p, q) would be very long compared to the other edges in that chain. On the other hand, one should definitely not reject an edge (p, q) just because let's say q's other adjacent edge is very short as shown in Figure 1.40,(b).

So what we propose is some kind of local uniformity condition. The length of an edge shouldn't be much longer than the longest edge in some specified neighbourhood, where neighbourhood describes nearby edges in the same chain as e. Clearly the neighborhood shouldn't be too small as just one very short edge should not prevent a long edge like in Figure 1.40,(b). Hence our local uniformity condition is defined as follows:

Local Uniformity Condition (I): Let $e = (p, q)$ be an edge in a potential reconstruction of a sample set. Furthermore let $q_1, q_2, q_3, \ldots q_k$ be the samples in the chain when following the chain adjacent to q, and let p_1, p_2, p_3, $\ldots q_k$ the samples adjacent to p.

We say e is not reasonable due to oversampling, if either

- $q_1, \ldots q_k$ exist and $\forall i = 1 \ldots (k-1)$ we have $|e| > f_{stretch} \cdot d(q_i, q_{i-1})$,
- $p_1, \ldots p_k$ exist and $\forall i = 1 \ldots (k-1)$ we have $|e| > f_{stretch} \cdot d(p_i, p_{i-1})$,

This means for one particular edge e to be reasonable, the longest edge amongst its k neighbours to either side must have length at least $|e|/f_{stretch}$.

For the whole thing to make sense, of course we need to require that each component of the curve has at least $2k+1$ samples taken from it:

Minimum Sampling Condition: At least $2k+1$ samples must be taken from every single curve in the collection of curves Γ.

It should be clear how to modify the algorithm to reject edges which are unreasonable due to local non-uniformity. In following we give an alternative local uniformity condition which is more in the spirit of the sampling conditions we have met so far:

Local Uniformity Condition (II): For any edge e of the correct reconstruction, the balls centered at the midpoint of e with radius r, $|e|/2 \leq r \leq |e|/2 + |e| \cdot (k-1) + f_{stretch} \cdot |e|$ must intersect the correct reconstruction in one component.

Lemma 20 *If the local uniformity condition (II) is fulfilled, no open ends of the correct reconstruction are closed by our modified algorithm.*

Proof. Consider a gap closing edge $g = (p, q)$ and let $e = (p_i, p_{i-1})$, $i \leq k$ be the longest edge amongst its neighbours. As g is not discarded by our modified algorithm, we know that $d(p_{i-1}, q) > |e| \cdot (k - 1 + f_{stretch})$, and $d(p_{i-1}, p) \leq (k-1) \cdot |e|$. Therefore $d(p, q) > f_{stretch} \cdot |e|$ and our algorithm would have discarded edge g. $\square$

Another problem is the fact that our algorithm might capture a corner spanning edge, if the samples on both legs are so dense that there is no blue edge which could trigger a corner exploration (see Figure 1.29). We can circumvent this by postulating:

Corner Triggering: For any corner there must be a blue edge in the correct reconstruction.

Observe that it is very simple to generate such a blue edge by placing a sample close to the actual corner point of the corner. Either both edges connecting the formerly last samples of either leg to this additional sample are red then we are done anyway, or one of them is blue, so it triggers the corner exploration.

Furthermore our algorithm might extend a corner further than it is supposed to do as we have seen in Figure 1.30. This can also be avoided by placing a sample point close to the actual corner point which is then chosen by both legs and terminates the corner exploration (or by using the local uniformity condition).

So with this additional sampling conditions and a slightly restricted definition of curves, we get the following result:

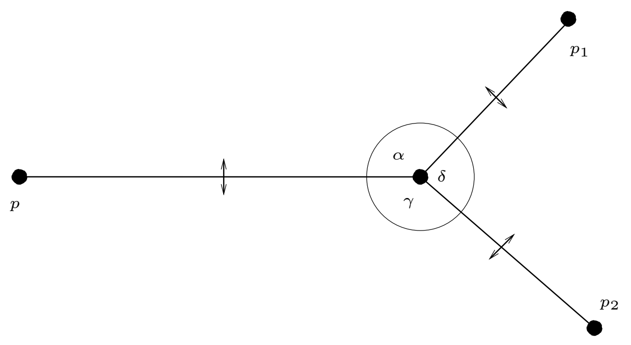


Figure 1.41: Sample with 3 red adjacent edges

Theorem 4 *For every collection of curves Γ there exists a finite sample set such that our modified algorithm exactly returns the correct reconstruction.*

1.7.2 Tuning the Parameters for best Practical Performance

In our experiments it turned out that the most limiting factor of our algorithm when it comes to sparse sample sets is the fact that not enough red edges are detected. So tweaking the algorithm to allow a smaller β-value would improve the performance considerably in practice. With a small modification of the algorithm this is possible without losing the theoretical guarantees. The main problem is that if we allow smaller values for β, we cannot be sure that all samples have at most 2 adjacent red edges. For example for $\sqrt{2} \leq \beta < 2$, there might be samples with three adjacent red edges. Fortunately we can clean-up the β-skeleton after step 2 of our algorithm easily by uncoloring the longest red edge for each degree-3-sample. Then we continue with the algorithm as described above. It remains to show that this clean-up procedure does not uncolor any smooth edges:

Lemma 21 *Smooth edges are detected and colored red by the algorithm after the first 3 steps for $\beta \geq \sqrt{2}$ and $f_{diam} \geq \frac{3}{2}$ and $\theta_{ball} < \frac{\pi}{2} - \theta_{beta}$.*

Proof. Clearly all smooth edges are colored red in step 2. Now assume that a 'smooth' edge $e = (p, q)$ is uncolored in step 3 of the algorithm, i.e. it is adjacent to a sample q which has two other red edges $e_1 = (q, p_1)$, $e_2 = (q, p_2)$, where $|e| > |e_1|, |e_2|$, as in Figure 1.41. Note that for $f_{diam} > \frac{3}{2}$, p_1 and p_2 are contained in the diametral ball of e of radius $|e| \cdot f_{diam} \geq 2$. Due to the sampling condition for smooth areas we know that $\alpha = \angle pqp_1 \geq \pi - \theta_{ball}$ as well as $\gamma = \angle pqp_2 \geq \pi - \theta_{ball}$. Hence $\delta = \angle p_2qp_1 < 2 \cdot \theta_{ball}$. On the other hand, because of the empty β-balls around e_1 and e_2, we know that $\delta > \pi - 2 \cdot \theta_{beta}$, which leads to a contradiction for $\theta_{ball} < \frac{\pi}{2} - \theta_{beta}$. $\square$

With this little modification we can choose $\beta \approx 1.55$, which is equivalent to $\theta_{beta} \approx 40^o$, but still keep the values for all other parameters.

Another parameter that strongly influences practical performance is f_{grow}. It should be chosen as small as possible to keep the area where the curve must remain 'flat' around the corner to a minimum. Note that f_{grow} can be decreased quite freely, if in turn f_{diam} is increased. (Note that f_{diam} only appears in the sampling condition but not in the algorithm !) But just to make sure that at least the ball covering all blue edges of a detected corner does not intersect any other parts of the curve, we want $f_{grow} \geq \frac{1}{\sqrt{1-\frac{1}{\beta^2}}} = \frac{2}{\sqrt{3}}$, and hence $f_{shrink} = \frac{4}{\sqrt{3}} + 1$ and $f_{diam} = \frac{2\cdot\sqrt{3}}{2-\sqrt{3}} + \frac{1}{2} \approx 13.43$. This choice of f_{diam} also allows us to set $\theta_{ball} \approx 45^o$.

1.7.3 Related work

Concurrently with our conference submission of this paper [FR01], Dey and Wenger submitted a paper to the same conference in which they present an algorithm based on their previous work in [DW00]. Their improved algorithm can reconstruct a collection of closed curves with corners (no open curves). They also use the idea of first detecting allegedly 'smooth' areas and then exploring potential corner areas. While we have described our algorithm mostly in terms of empty β-balls, they use what they call the 'ratio condition' which denotes the ratio between the lengths of Delaunay edges and their dual Voronoi edges. In fact these concepts are quite the same. The conditions for which they guarantee correct exploration of corners differ from ours.

In [BP01], Boyer and Petitjean have also sketched a sampling condition and an algorithm which refers more to properties of the correct reconstruction rather than the original curve. They do not provide a rigourous proof of correctness, though.

1.8 Running Time and Experimental Results

We haven't attempted to optimize the running time of our algorithm. Assuming that of the $n = n_s + n_c$ samples we have n_c samples in corner regions and n_s samples in regular parts, we can obtain the following bounds with a very naïve implementation:

- Computation of the Delaunay Triangulation: $\mathrm{O}(n \cdot \log n)$
- Colouring the red and blue edges: $\mathrm{O}(n)$
- At most $4 \cdot n_c$ corner exploration can happen, each of which requires at most n steps. After each of these steps there is a corner verification step which costs at most $\mathrm{O}(n)$, so we obtain a total running time for corner explorations and verification of: $\mathrm{O}(n_c \cdot n^2)$
- detecting interfering corners and red edges can be done in time: $\mathrm{O}(n)$

So the overall running time for a very naïve implementation is dominated by the corner explorations and verifications which is $\mathrm{O}(n_c \cdot n^2)$.

Note though that the estimation of cost $\mathrm{O}(n)$ for each verification step is very pessimistic, as in most cases, when the verification stage has to verify a corner explorations of two sequences $q_k q_{k-1} \dots q_1$ and $p_l p_{l-1} \dots p_1$, the potential 'next picks' q_{k+1} or p_{l+1} are contained in the corner ball around the potential corner, so the verification test can be aborted after just one comparison. In fact, to design an input which requires a full $O(n)$ verification after every extension of the sequences would require that the distance between the samples grows geometrically. So in practice, one would expect a running time of $O(n_c \cdot n)$.

We have implemented a prototype of our algorithm (currently without the local uniformity condition) using LEDA [MN00], which seems to behave pretty well in practice, even when using the parameters determined in section 1.6.3.

Compared to for example the CRUST algorithm by Amenta, Bern, Eppstein [ABE98] or the Dey-Kumar algorithm [DK99], it did a much better job in connecting corners as can be seen in Figures 1.42, 1.43, 1.45.

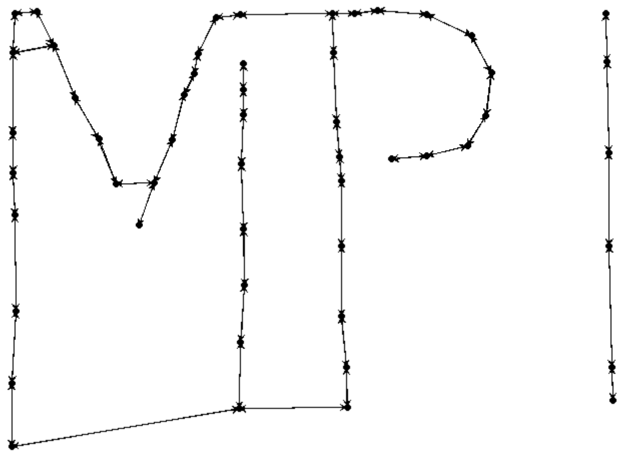


Figure 1.42: The Crust algorithm

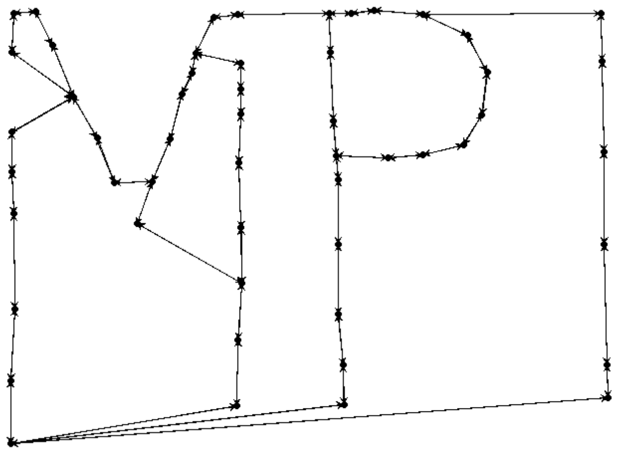


Figure 1.43: The Dey-Kumar algorithm

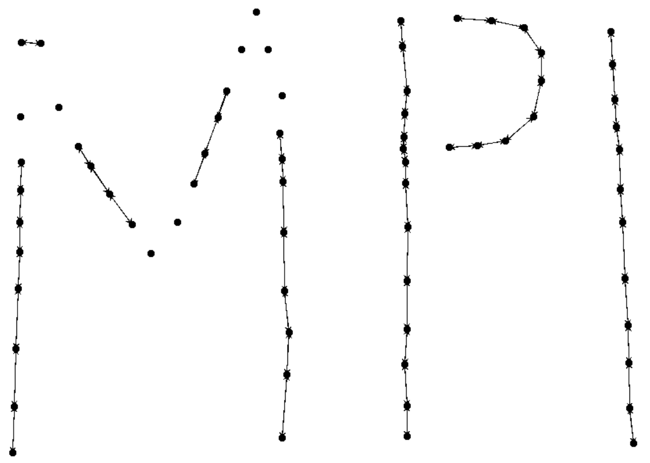


Figure 1.44: The Dey-Mehlhorn-Ramos algorithm

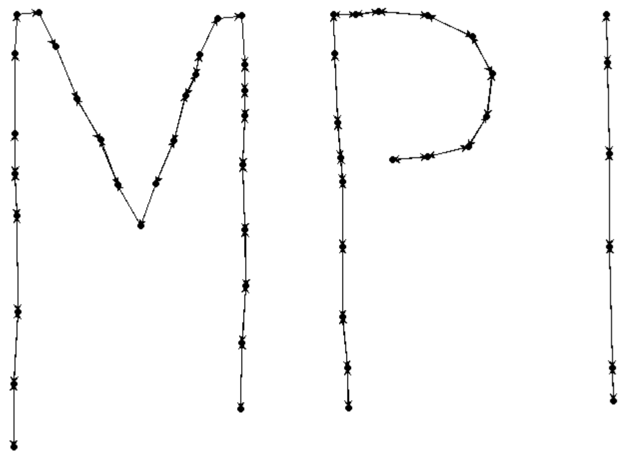


Figure 1.45: Our algorithm

1.9 Discussion and Outlook

We presented the first algorithm for curve reconstruction which provably can handle a collection of curves with corners and endpoints and also introduced a new sampling condition which is not as restrictive as the sampling conditions based on the medial axis. In practice our algorithms seems to perform quite well compared to existing algorithms like the ones mentioned in the introduction. As a more theoretical result we also proved that for any collection of curves with corners and endpoints, there exists a finite sample set from that collection for which a slight modified version of our algorithm outputs exactly the correct reconstruction.

In two dimensions, the problem of reconstructing open and closed curves with branching points is still open. But the really interesting problem is the reconstruction of surfaces in $\mathbf{R}^3$.

In three dimensions, as of now algorithms with a guarantee only exist for closed smooth manifolds. These algorithms like the algorithms in $\mathbf{R}^2$ are based on a Delaunay filtering approach, so as a first step, they all compute the Delaunay triangulation of the sample set. Unfortunately the complexity of a Delaunay triangulation in $\mathbf{R}^3$ can be $\theta(n^2)$. So even though the complexity of the surface to be reconstructed is $O(n)$, all these algorithms have an inherent running time of $\theta(n^2)$. Recently we have developed a modification of one of these algorithms which only locally computes part of the Delaunay triangulation. With an additional sampling condition similar to the local uniformity condition, we could prove that the surface reconstruction problem can be solved in $\mathrm{O}(n \log n)$ time for smooth closed manifolds. Still, it would be nice to get rid of this additional condition and, of course, to provably reconstruct open surfaces or even surfaces with sharp ridges and corners. The main idea of our algorithm, first to detect parts of the curve that are likely to be smooth and then explore potential corners might also apply there.

Part II

The Efficient and Exact Implementation of Geometric Algorithms

Chapter 2

The Exact Computation Paradigm

2.1 Introduction

Geometric algorithms are usually designed and proven to be correct in a computational model that assues exact computation over the real numbers. Since no computer provides exact arithmetic on real numbers in hardware, programmers must find some substitution when implementing these algorithms. Quite commonly, they resort to fast finite precision arithmetic due to its support by hard- and software as well as its convenient use. For some problems and restricted sets of input data, this approach works well, but in many implementations the effects of squeezing the infinite set of real numbers into the finite set of floating-point numbers can cause catastrophic errors in practice.

There are several ways geometric algorithms may misbehave when exact arithmetic is replaced by floating-point arithmetic. In the best case, they produce quite usable results in spite of some incorrect decisions, but most algorithms do not; they either produce completely inconsistent results, crash or loop.

To give you an idea how easily a simple predicate can be decided incorrectly when replacing exact arithmetic by finite precision computation, look at the example in Figure 2.1:

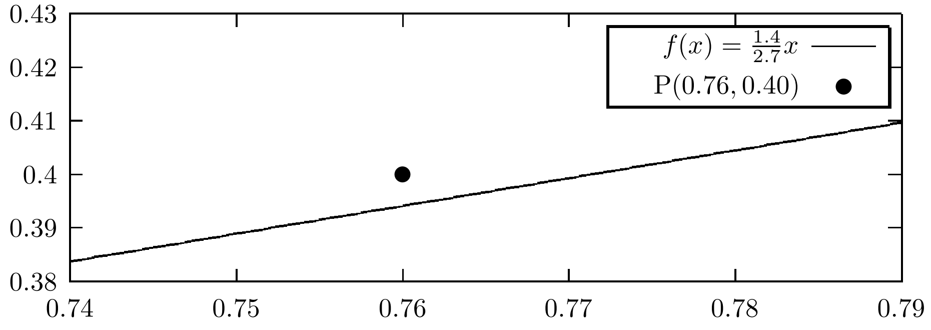


Figure 2.1: Point P is clearly above line f

Consider a line f given by the equation $y = f(x) = 1.4 \cdot x/2.7$. What we are interested in, is the position of the point $P(0.76/0.40)$ with respect to f, i.e. does P lie above or below the line f. This test occurs in almost any geometric algorithm and is called the *sidedness predicate*. Using exact arithmetic, it is not hard to see that P actually lies *above* f as

$$f(P_x) = f(0,76) = 1.4 \cdot 0.76/2.7 < 0.40 = P_y$$

Now assume we are restricted to a floating-point system with base 10, mantissa length 2, and rounding to nearest, i.e. after an arithmetic operation the result is always rounded to two significant digits. Let's do the calculation with this restricted precision. $\odot, \oslash$ denote the floating-point counterparts of multiplication and division.

$$f(P_x) = f(0.76) = 1.4 \odot 0.76 \oslash 2.7 = 1.1 \oslash 2.7 = 0.41 > 0.40 = P_y$$

And hence we conclude that P is *below* f which is clearly wrong !

Conditionals like this are the most critical parts in a program because they determine the flow of control. If in every test the same decision is made as if all computations would have been done over the reals, the algorithm is always in a state equivalent to its theoretical counterpart. But still, if some predicates are decided incorrectly, why is this such a big problem ? It might only produce a slightly perturbed output. The problem is, that conditionals are usually not independent. So if due to roundoff errors a conditional is decided incorrectly, this might contradict some other conditionals already decided or going to be decided in the future. Algorithms are usually not designed to cope with such inconsistencies, so they crash, loop or produce garbage output.

The exact computation paradigm

There are two obvious approaches for solving the precision problem. The first is to change the model of computation: design *robust* algorithms that can deal with imprecise computation, i.e. algorithms that can cope with inconsistencies incurred by incorrect predicate decisions. Unfortunately, there is no generic way to derive such algorithms, and only for a small number of (easy) problems, such algorithms have been developed. The second approach is the *exact computation paradigm* ([YD95]); it advocates to guarantee correctness of the implementation by ensuring that every single predicate is evaluated correctly. This is achieved by providing software-based exact number types and hence exact predicate implementations.

As we have seen, the evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. So the naive way to compute the sign of an expression is to compute the value of the expression (using exact arithmetic) and to read off the sign from the value. Unfortunately, computing these expressions using exact arithmetic has its cost, which is considerably higher then pure floating-point arithmetic; an overhead of a factor of 10-100 can be expected.

Floating-point filters

A much more efficient technique is the use of *floating-point filters* [FV96, KLN91]. A floating-point filter computes an approximate value of an expression and a bound for the maximal deviation from the true value. If the error bound is smaller than the absolute value of the approximation, approximation and exact value have the same sign and hence the sign of the approximation may be returned. In this way the true sign can be obtained quickly. The advocates of floating-point filters claim that filters at the predicate level realize the exact computation paradigm at little cost; the running time is claimed to be no more than twice the running time of a pure floating-point implementation.

Of course, this statement is only true if the floating-point filter always succeeds in deciding the predicate, and the floating-point filter mechanism can be applied for the whole computation. Very often these conditions are not met in practice, though.

First, the given geometric setting might be such that very often the floating-point filter fails (usually due to degenerate or nearly-degenerate configurations like 3 (almost) collinear points, 4 (almost) cocircular points, etc.). In these cases, we will always have to resort to exact arithmetic with its implied overhead. And even in the cases where the floating-point filter succeeds in deciding the predicate, it would be desirable to reduce the overhead compared to double arithmetic even further.

Secondly, if the algorithms construct new geometric objects, the floating-point filtering concept cannot be used directly. What is the problem with the construction of new objects? In most geometric kernels like the ones of the popular libraries CGAL [CGA99] and LEDA [MN00], points (and all other geometric objects) are represented by their coordinates (either cartesian or homogeneous) and hence the construction of an object requires the *exact* computation of its coordinates. This has two undesirable consequences. On one hand the running time increases dramatically when operating with constructed objects due to the increased numerical complexity, but maybe even more importantly, the space consumption grows rapidly. So in practice, users either write their own routines to allow for filtering on construction level (which can be supported by tools like our EXPCOMP expression compiler [BFS98]) or use rounding schemes after every construction to keep the numerical complexity and space requirements low, see [For99] for example. Note though, that these rounding schemes actually require a proof that they do not affect the final result considerably. These proofs are non-trivial and usually cannot be generalized. So each application has to be considered separately.

Both these problems we have actually experienced in our implementation work and therefore tried to come up with solutions.

Our Contribution

Of course, the ultimate challenge is to achieve exact geometric computation at the cost of floating-point arithmetic. Our results are a step into this direction:

- We develop the concept of *structural filtering* which is a generalization of floating-point filtering. Structural filtering can reduce the overhead compared to pure floating-point arithmetic further by allowing some predicates to err, without sacrificing the guarantee for an exact outcome. Structural filtering views the execution of an algorithm as a sequence of steps and applies filtering at the level of steps. A step can be anything between a simple predicate and the execution of the whole algorithm.

As an interesting subresult on its own, we exhibit differences in the behaviour of sorting algorithms under erroneous comparisons.

- We present the design of a geometric kernel called LOOK (Lazy Object Oriented Kernel for geometric computation), which is the first of its kind that makes use of filtering techniques not only on the predicates level but also on the level of geometric constructions. The main idea is to represent geometric objects not by their coordinates, but by the geometric operation that produced them and compute coordinate representations only on demand in a lazy fashion.

  We thoroughly discuss all issues arising in the design of such a kernel, in particular the choice of numerical representation and space consumption.

Both of these approaches have been successfully tested and used in the framework of the LEDA and CGAL libraries.

Outline

In Section 2.2 we first present some preliminaries and discuss previous work which we will build upon. Chapter 3 will introduce the new concept of structural filtering whereas in Chapter 4 we present the design of LOOK, our new kernel for geometric computation.

2.2 Preliminaries and Previous Work

In this section we will briefly review the state of the art in software packages for exact arithmetic and filtering techniques.

2.2.1 Exact Number Types in LEDA

There exist several software packages providing means for exact computation like the Gnu Multiprecision library GMP [Gra96], the CORE library [KLPY99], and the number types in LEDA (Library of Efficient Datatypes and Algorithms) [MN00], which we will concentrate on in the following.

In LEDA, the programmer has access to three C++ classes for integer, rational and algebraic computation. They can be used very similarly to the standard C/C++ number datatypes due to operator overloading, but also have additional functions for conversion, sign determination etc.

Integer Arithmetic

The LEDA type integer represents integer numbers of arbitrary size. Amongst others the arithmetic operations `+,-,*` are available for that datatype. LEDA integers guarantee exactness in a sense that each operation is performed numerically exactly.

An integer is essentially implemented by a vector of unsigned long numbers with the sign and the size stored separately.

The use of LEDA integers induces quite an overhead compared to double arithmetic, which increases with increasing bit-lengths of the input values, since every operation is performed numerically exactly.

Rational Arithmetic

A rational is implemented by two integer numbers – one for the numerator and one for the denominator, where the sign of the numerator determines the sign of the rational.

As for LEDA integers, the use of LEDA rationals induces quite an overhead compared to double arithmetic, which also increases with increasing bit-lengths of nominator and denominator.

Algebraic Arithmetic

The LEDA type real represents algebraic numbers. Amongst others the arithmetic operations `+,-,*,/,sqrt` are available for the datatype. Unlike LEDA integers or rationals, LEDA reals do not guarantee exactness in a sense that each operation is performed numerically exactly, but if a sign of a real is asked by calling the `sign(real &)` function, it is guaranteed that this result is correct.

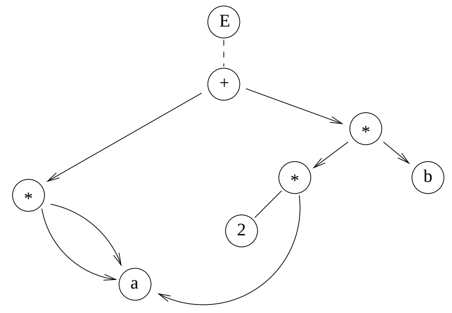


Figure 2.2: Expression Graph for $E = a^2 + 2 \cdot a \cdot b$

A real expression E is represented by its expression graph $\mathcal{G}(E)$ and a double approximation $\widetilde{e}$ with error bound err such that

$$|e - \widetilde{e}| \leq err$$

$\mathcal{G}(E)$ is a directed acyclic graph with a single root. The root is associated with E itself, and inner nodes correspond uniquely to the arithmetical operations that define E. The input values of E are contained in the leaves of its expression graph. Figure 2.2 shows an example. When the sign of a real number x needs to be determined, the datatype first computes a number q_x – called *separation bound* for x –, such that $|x| < q_x$ implies $x = 0$. Using bigfloat arithmetic — LEDA bigfloats are a generalization of floating-point numbers with exponents and mantissa of arbitrary length according to the IEEE standard [IEE85] —, the datatype then computes successively intervals of decreasing length that include x, until the interval does not contain zero (then the sign is obvious) or else until the length of the interval is smaller than q_x (then the sign must be zero).

Two shortcuts are used to speed up the computation of the sign. Firstly, as long as the double approximation $\widetilde{x}$ is known to be exact, i.e. $\widetilde{x} = x$, no expression graph is constructed and only the (exact) double approximation $\widetilde{x}$ represents x.

Secondly, if the double approximation $\widetilde{x}$ suffices to determine the sign of x i.e. $0 \notin [\widetilde{x} - err, \widetilde{x} + err]$, it is returned and no bigfloat arithmetic is used. This technique is called a floating-point filter and we will discuss several variants in the next subsection.

As to be expected, the overhead of LEDA reals is quite high compared to double arithmetic. The construction of the expression graph and the computation of the error bound has its costs. Even if large parts of the computation do not require the construction of an expression graph due to the double arithmetic still being exact, some overhead is imposed by the tests for the exactness of the double results.

2.2.2 Deriving Error Bounds for Floating-Point Computations

In this section we will have a closer look at floating-point arithmetic. If the floating-point arithmetic on a machine complies to the IEEE standard, one can guarantee an error bound for the error occurring in *one single* operation: Let $x = x_1 \mathrm{op} x_2$ be the *exact* outcome of an arithmetic operation on two floating-point numbers x_1, x_2, $\widetilde{x} = x_1 \mathrm{fop} x_2$ the result under floating-point arithmetic. Then the IEEE standard guarantess that $|\widetilde{x} - x| \leq 2^{-(p+1)}$ where p is the mantissa length of the floating-point representation ($p = 52$ for the C/C++ type double).

But as we evaluate complex expressions involving more than one operator, errors are propagated in some way from the "earlier stages" of computation to the final result. Assuming we have evaluated a complex expression e with floating-point arithmetic to $\widetilde{e}$, what we then want is an upper bound *err* for the error of this value, i.e. something like:

$$|e - \widetilde{e}| \leq err$$

We briefly present several techniques that can be used to compute this error bound. For a more detailed description of these techniques and their proofs see [BMS96, BBP98, Fun97].

2.2.2.1 Fully-Dynamic error analysis

Fully-dynamic error analysis means that the error bound is computed completely at run-time, and hence can make use of the actual values of the expressions. We present two schemes for fully-dynamic error analysis:

Relative Error Bounds In [BMS96], relative error bounds are derived for floating-point computations, i.e. for every expression e and its floating-point evaluation $\widetilde{e}$, an ϵ_e is calculated such that the following is true at all times:

$$|\widetilde{e} - e| \leq \epsilon_e \cdot |\widetilde{e}|$$

The following shows a list of the formulas for inductively computing the ϵ_e-values always assuming that the for the operands x, y the invariant is fulfilled:

$$\epsilon_{x \pm y} = 2^{-p-1} + |\frac{\widetilde{x}}{\widetilde{e}}| \cdot \epsilon_x + |\frac{\widetilde{y}}{\widetilde{e}}| \cdot \epsilon_y$$

$$\epsilon_{x \cdot y} = (\epsilon_x + \epsilon_y + \epsilon_x \cdot \epsilon_y) \cdot (1 + 2^{-p-1}) + 2^{-p-1}$$

$$\epsilon_{x/y} = 2^{-p-1} + \frac{\epsilon_x + \epsilon_y}{1 - \epsilon_y} \cdot (1 + 2^{-p-1})$$

$$\epsilon_{\sqrt{x}} = eps + \epsilon_x \cdot (1 + 2^{-p-1})$$

For more detailed information and the proofs see [BMS96]. Note though that

- The relative errors must be computed dynamically at runtime, since computing the relative error of an addition requires the actual values of the floating-point approximations.

- The overhead compared to simple floating-point evaluation is quite high. For example, the addition $x + y$ requires – apart from the computation $x + y$ itself – 2 additions, 2 multiplications, 2 divisions and 2 absolute values for computing the new relative error.

These error bounds are used as built-in floating-point filter of the LEDA type real to filter out easy tests which do not require the arbitrary precision calculation.

Interval arithmetic More recently, Burnikel, Brönnimann, and Pion in [BBP98] presented another fully-dynamic scheme for computing error bounds, which was previously known in the numerical analysis community but not used in the context of computational geometry before. Their approach is based on the possibility to switch rounding modes if the floating-point arithmetic on a machine complies with the IEEE standard. A value x is represented by an interval $[\widetilde{x}] = [\widetilde{x_l}, \widetilde{x_u}]$. Assuming x and y are represented by intervals $[\widetilde{x}]$ and $[\widetilde{x}]$, the following rules are used to compute the interval resulting from an arithmetic operation:

$$[\widetilde{x}] + [\widetilde{y}] = [\widetilde{x_l} + \widetilde{y_l}, \widetilde{x_u} + \widetilde{y_u}]$$

$$[\widetilde{x}] - [\widetilde{y}] = [\widetilde{x_l} - \widetilde{y_u}, \widetilde{x_u} - \widetilde{y_l}]$$

$$[\widetilde{x}] \cdot [\widetilde{y}] = [\min\{\widetilde{x_l}\widetilde{y_l}, \widetilde{x_l}\widetilde{y_u}, \widetilde{x_u}\widetilde{y_l}, \widetilde{x_u}\widetilde{y_u}\}, \max\{\widetilde{x_l}\widetilde{y_l}, \widetilde{x_l}\widetilde{y_u}, \widetilde{x_u}\widetilde{y_l}, \widetilde{x_u}\widetilde{y_u}\}]$$

$$[\widetilde{x}]/[\widetilde{y}] = \begin{cases} [\widetilde{x}] \cdot [1/\widetilde{y_u}, 1/\widetilde{y_l}]] & , \quad 0 \notin [\widetilde{y}] \\ \mathbf{R} & , \quad \text{otherwise} \end{cases}$$

$$[\widetilde{x}]^{1/2} = \begin{cases} [\widetilde{x_l}^{1/2}, \widetilde{x_u}^{1/2}] & , \quad 0 \notin [\widetilde{y}] \\ \mathbf{R} & , \quad \text{otherwise} \end{cases}$$

If one of the intervals is infinite, we set the resulting interval to $\mathbf{R} = [-\infty, \infty]$. Since the computed intervals $[\widetilde{x}]$ in general have bounds x_l, x_u which are not exactly representable by a floating-point number, we always round downwards (resp. upwards) to obtain an interval $[\widetilde{x_l}, \widetilde{x_u}]$ that encloses the 'real' interval and is exactly representable by floating-point upper and lower bounds. This rounding can be cheaply implemented by switching the rounding mode of the IEEE floating-point unit. Unfortunately, not all currently used platforms adhere to the IEEE floating-point standard.

Again we note, though, that the overhead compared to pure floating-point arithmetic is rather high. In case of the multiplication, 4 times more operations have to be performed, not even counting the comparisons to determine the maximum and minimum for lower and upper bounds. Furthermore, switching the rounding-modes on most architectures is a very costly operations. Still, at the cost of this higher overhead, interval arithmetic gives the best possible error bounds.

2.2.2.2 Semi-static Error analysis

As we have seen, fully-dynamic error analysis has the drawback of implying a rather large overhead during runtime.

This suggests dividing the computation of the error bounds in a static part and a dynamic part. The static part can be precomputed before runtime without any knowledge of the actual

expr. e	approx. $\widetilde{e}$	supremum $\widetilde{e_{sup}}$	index ind_e
$x+y$	$\widetilde{x} \oplus \widetilde{y}$	$\widetilde{x_{sup}} \oplus \widetilde{y_{sup}}$	$1 + MAX(ind_x, ind_y)$
$x-y$	$\widetilde{x} \ominus \widetilde{y}$	$\widetilde{x_{sup}} \oplus \widetilde{y_{sup}}$	$1 + MAX(ind_x, ind_y)$
$x \cdot y$	$\widetilde{x} \odot \widetilde{y}$	$\widetilde{x_{sup}} \odot \widetilde{y_{sup}}$	$1 + ind_x + ind_y$
x/y	$\widetilde{x} \oslash \widetilde{y}$	$\frac{(\lvert\widetilde{x}\rvert \oslash \lvert\widetilde{y}\rvert) \oplus (\widetilde{x_{sup}} \oslash \widetilde{y_{sup}})}{(\lvert\widetilde{y}\rvert \oslash \widetilde{y_{sup}}) \ominus (ind_y + 1) \cdot 2^{-p}}$	$1 + MAX(ind_x, ind_y + 1)$
$x^{\frac{1}{2}}$	$\sqrt{\widetilde{x}}$	$\begin{cases} (\widetilde{x_{sup}} \oslash \widetilde{x}) \odot \sqrt{\widetilde{x}} & if\ \widetilde{x} > 0 \\ \sqrt{\widetilde{x_{sup}}} \odot 2^{\frac{p}{2}} & if\ \widetilde{x} = 0 \end{cases}$	$1 + ind_x$

Table 2.1: Rules for computing approximations, suprema and indices.

values of the expressions, whereas the dynamic part is computed during runtime, but with hopefully much less operations than the error calculations we have seen in the last paragraph. We will review the semi-static scheme which we have presented in [BFS98]. For sake of simplicity we neglect the problems of underflow and refer the reader to [Fun97] for a complete discussion.

For every expression e we not only compute the floating-point approximation $\widetilde{e}$ but also an upper bound $\widetilde{e_{sup}}$ for $|\widetilde{e}|$, called the supremum of $\widetilde{e}$, and an integer ind_e – the index of e–, such that the following bound for the absolute error of the floating-point approximation is true at all times:

$$|\widetilde{e} - e| \leq \widetilde{e_{sup}} \cdot ind_e \cdot 2^{-p} \tag{2.1}$$

An input value x exactly representable by a double has the floating-point approximation $\widetilde{x} = x$, the supremum $\widetilde{x_{sup}} = |\widetilde{x}|$ and the index 0. An input value not exactly representable by a double has the floating-point approximation $\widetilde{x} = round(x)$, the supremum $\widetilde{x_{sup}} = |\widetilde{x}| = |round(x)|$ and the index 1.

The index ind_e may be computed statically whereas $\widetilde{e}$ and $\widetilde{e_{sup}}$ must be computed at runtime using the inductively given rules in table 2.1. $+, -, \cdot, /, .^{\frac{1}{2}}$ denote exact addition, subtraction, multiplication, division, and square root, whereas $\oplus, \ominus, \odot, /, \sqrt{}$ denote their floating-point counterparts.

We see that the computation of the supremum is quite similar to the computation of the floating-point approximation itself and therefore the implied overhead is reasonably small. In case of $+, -, \cdot$, only twice as many operations are needed, three times as many in case of $\sqrt{}$. The division has a considerably higher overhead but its use often can be avoided in practice.

This scheme for error analysis is similar to the one given in [Bur96] but refined in some respects. In contrast to [Bur96], our scheme unrestrictedly supports divisions and square roots.

2.2.2.3 Fully static error analysis

If the input data consists of integer values of a bound bit-length and $+, -, \cdot$ are the only operators used, an upper bound for $\widetilde{e_{sup}}$ can be determined statically such that no overhead for the computation of the supremum occurs at runtime . (This is even possible for non-integer input values where the bit-length of $\lceil |e| \rceil$ is bounded.) An upper bound for the bit-length of the supremum can be inductively computed using the following formulas:

$$bitlen_{\pm} = 1 + MAX(bitlen_{op_1} + bitlen_{op_2})$$

$$bitlen_{\cdot} = bitlen_{op_1} + bitlen_{op_2}$$

So there is no overhead at runtime for computing the error bound $ind_e \cdot 2^{bitlene_{sup}} \cdot 2^{-p}$.

2.2.3 Filtering of Geometric Predicates

As we have seen with these schemes for computing error bounds of floating-point calculations, there is always a tradeoff between tightness of the error bound and the run-time overhead implied by the computation of the error bound. This suggests the following evaluation strategy for the sign evaluation of an arithmetic expression e:

1. evaluate e using double arithmetic to $\widetilde{e}$
2. check the sign of $\widetilde{e}$ with the fully-static determined error bound (if available); only if this fails, continue
3. compute the semi-static error bound and check the sign of $\widetilde{e}$ with that; only if this fails, continue
4. compute the fully-dynamic error bound and check the sign of $\widetilde{e}$ with that; only if this fails continue
5. evaluate e using exact arithmetic to obtain the sign of e

In this way, easy instances are always decided in early stages of this cascaded evaluation and the implied overhead is reasonably small. The generation of such cascaded evaluation schemes can even be automated by tools like EXPCOMP [Fun97, BFS98] which we have developed.

The overhead observed in practical applications when comparing such an exact but filtered implementation with a (not necessarily reliable) floating-point implementation very often is around a factor of 2 (only measuring the time to evaluate the predicates and not taking into account the time spent on the combinatorial part of the algorithm). It gets worse, though, if

- the input data exhibits a lot of (near-)degeneracy, i.e. many of the critical expressions end up with a value of zero or close to zero, which makes it much harder for the filter stages to decide the predicate, of course.
- the predicates do not only operate on input data but on geometric objects constructed during the course of the algorithm; remember that if all filter stages fail, the last stage of the predicate evaluation falls back to exact arithmetic assuming that the input data is available in an exact representation. So in these cases the filtering only takes place in the predicate evaluation but not in the constructions.

In the next two Chapters we will present results which try to remedy these drawbacks.

Chapter 3

Structural Filtering – a New Approach to Filtering

3.1 Filtering Strategies

The topic of this section is a general discussion of filtering strategies. We view the execution of an algorithm as a sequence of steps. A step may be anything from the execution of a single instruction over the execution of a large subprogram to the execution of the entire program. If every step of an algorithm produces the correct result, the entire computation will produce the correct result.

The execution of a step consists of the evaluation of conditionals (predicates) and the execution of the straight-line code between the conditionals. The simplest way to ensure the correct execution of a step is to guarantee that all conditionals in the step are evaluated correctly.

An alternative way to ensure the correct execution of a step is to allow errors in the evaluation of the conditionals, to check at the end of the step whether the step performed correctly, and, if not, to repair the errors made. Of course, this approach is only viable if the "unsafe" execution of a step is faster than its "safe" execution, if the correctness check is simple, if errors occur rarely, and if the repair is simple. Observe that there are four "ifs" in the preceding sentence. We will show that there are many situations where the answer to all four ifs is yes.

We start by refining our view of the execution of an algorithm. We view algorithms as manipulating an underlying data structure and distinguish between search and update steps. Update steps are pieces of code that may change the underlying data structure and search steps are pieces of code that do not change the underlying data structure but are otherwise arbitrary. Structural filtering applies to search steps. It does not modify update steps. Thus the underlying data structure stays correct. We give three examples to illustrate the concepts.

1. Any algorithm falls under the paradigm if we call the value of all program variables the underlying data structure, the evaluation of each predicate[1] in a conditional a search

[1]We assume that predicates in conditionals have no side-effects, a minor restriction. In geometric programs

step (the step "searches" for the value of the expression), and call the straight-line pieces of code between conditionals update steps.

2. Consider a dictionary implementation based on a balanced tree. The tree constitutes the data structure manipulated by the algorithm. An insert operation consists of a search step, which determines the position in the tree at which the new key is to be added, followed by an update step, which adds the key to the tree.

3. Consider an incremental algorithm for constructing Delaunay diagrams. The data structure is the current Delaunay triangulation and a search structure for locating points in the triangulation. An insertion of a new point consists of a search step, which locates the triangle of the current triangulation containing the new point, and an update step which inserts the point, performs flips to construct the new Delaunay triangulation, and modifies the search structure.

We postulated that a search step does not change the underlying data structure. A search step computes information (= the value of a predicate, a position in a tree, a triangle in a triangulation) which the subsequent update step uses to perform changes on the data structure. A search step evaluates some number of predicates. We assume that a predicate can be evaluated in two ways; the expensive way guarantees the correct value and the cheap way will usually give the correct result, but may err. In this general discussion we make no assumption about when a cheap comparison errs. In the context of geometric programs a cheap evaluation of a predicate is the evaluation with floating-point arithmetic, and an expensive evaluation is the evaluation with exact arithmetic (maybe with a floating-point filter).

The safe way to perform a step is to use only expensive predicate evaluations. Assume now that we use cheap predicate evaluations instead. The following observation is trivial but powerful. *If a search step amounts to a walk in an acyclic graph where predicate evaluations are used to determine the edges to be followed, then a search step will always terminate.* In our three examples above the search is a walk in an acyclic graph[2].

The search step, if executed with cheap predicates, may not end in the right sink of the acyclic graph. We postulate that it is easy to check whether the correct sink is reached. In our first example, the check amounts to the error-bound computation in the floating point evaluation of the underlying arithmetic expression, in our second example, the check amounts to the (exact) comparison with the two neighboring elements, and in the third example, the check amounts to orientation tests with three sides of a triangle.

If the search step ends in the correct sink of the search graph, we are done at this point. If the check reveals an error, we still have to find the correct sink. There is a generic way of reaching the correct sink. Repeat the search with expensive predicate evaluations. Observe that this is possible because we postulated that a search step does not change the underlying data structure. In our first example, the generic strategy amounts to an evaluation with

the predicates in conditionals are typically the evaluation of the sign of an arithmetic expression.

[2]In the first example the graph is a tree with three nodes. In the root the boolean expression is evaluated and the two children correspond to true and false.

exact arithmetic. In the two other examples, there are better ways to correct the error. In the second example, we may walk along the leaves of the tree and in the third example, we may use a walk through the triangulation.

Let us summarize. Structural filtering applies to search steps. If the search step amounts to the walk in an acyclic graph then it can be performed with cheap comparisons without the danger of looping. An error in the search step can always be corrected by redoing the search with expensive comparisons. Better strategies may exist and we gave two examples. The verification of the search step is problem dependent. With the generic solution to error correction, *only* the verification requires additional programming.

What can we hope to gain by structural filtering? The cost of an update step is unchanged. The cost of a search step is its cost when executed with cheap comparisons, plus the cost of the check, plus the cost of the repair. Structural filtering is particularly useful if the search steps dominate the running time of the algorithm. This is the case for our second and third example and, more generally, for many incremental constructions in geometry. In an insertion into a tree, the search step has cost $O(\log n)$ and the update step has cost $O(1)$. The same holds true for randomized incremental algorithms for convex hulls, Delaunay triangulations, Voronoi diagrams, and many other problems.

There is a second phenomenon which is exploited by structural filtering. Predicate evaluations may be redundant. There may be several paths to the correct sink and hence errors in predicates may be corrected by later predicates. Figure 3.1 illustrates the phenomenon for our third example.

We will next compare structural filtering with filtering on the predicate level and filtering on the algorithm level.

Filtering on Predicate level

Filtering at the predicate level amounts to evaluate all predicates correctly, but to do so in a clever way. The evaluation of a predicate amounts to the computation of the sign of an arithmetic expression. Predicate filtering computes the sign in three stages: In stage one the expression is evaluated using floating-point arithmetic, in stage two an error bound for the floating-point computation is computed, and in stage three the expression is evaluated with exact arithmetic, if the error bound does not suffice to conclude that the sign computed in stage one is the correct sign. The cheap evaluation of the predicate uses only stage one. The implementation of predicate filters is discussed in [BFS98] and [MN00]. The efficacy of floating-point filters is discussed experimentally in [FV96, MN00, Sch99] and theoretically in [DP99].

Let us consider the extreme cases. If the floating-point computation always computes the correct sign, the cheap evaluation never errs and saves the computation of the error bound. The computation of the error bound has typically about the same cost as the computation of the sign and hence a cheap comparison has about half the cost of an expensive comparison. Thus we may expect that structural filtering can make significant savings; we should not expect to see a factor of two since the search step has to do some work outside the predicate evaluations and since structural filtering has to verify the result of the search.

If the floating-point computation never computes the correct sign, predicate filtering always has to resort to exact arithmetic. Since the cost of exact arithmetic is significantly larger than the cost of floating-point arithmetic (around 10-100 times the cost; see [Sch99], for example), stage three will dominate the cost of an expensive predicate evaluation and a cheap comparison is much cheaper than an expensive comparison. Thus, even with the generic repair technique, the cost of structural filtering is not much larger than the cost of predicate filtering; observe that the cost of the search step with cheap predicates will be much smaller than the cost of the search with expensive predicates.

The advantage of predicate filtering is its genericity. Once "filtered" versions of the predicates are available, all algorithms using them benefit. There is no change required in an algorithm to switch from unfiltered predicates to filtered predicates. Moreover, the techniques for writing filtered predicates are well developed and even software supported [BFS98].

The disadvantage of predicate filtering is the fact that the error-bound computation is always made. Structural filtering avoids it at the cost of the verification of the search step.

Filtering on Algorithm level

While the filters on predicate level work on the level of the most basic (low-level) operations of an algorithm, filters on algorithm level work on the highest level possible. Here the idea is: compute with floating-point arithmetic, check the result, and repair, if necessary, to get the exact result.

There are two problems with filtering at the algorithm level. First, the design of robust algorithms using only floating-point arithmetic is a difficult task even if robustness only means that the program should always run to completion. The papers [FM91, Mil88, SOI90] illustrate the difficulty of designing robust algorithms. Second, the repair step is non-trivial if the floating-point algorithm does not come with a strong guarantee of what it computes. The purpose of restricting filtering to the search steps is precisely to guarantee that errors in predicate evaluations do not corrupt the data structure. Only the paper [KW98] discusses filtering at the algorithm level and the repair step. The main disadvantage of filtering at the algorithm level is that there are no widely applicable techniques for obtaining robust floating-point implementations.

Of course, filtering at the algorithm level approach also has its advantages. If no cheap evaluation errs, the result will be correct, and the only additional cost is the cost of checking.

In the following sections we will try to give an idea of our new filtering scheme on some simple examples.

3.2 Sorting

In the following we examine how simple sorting algorithms behave when comparisons are allowed to err. How difficult the repair step is, depends on that behaviour. Surprisingly we will show that there *are* differences between sorting algorithms when analyzing them with respect to robustness against erroneous comparisons.

We consider the problem of sorting a set $S = \{x_1, \ldots, x_n\}$ from a linearly ordered universe. Our algorithms may use *cheap* and *expensive* comparisons. An expensive comparison always gives the correct result whereas a cheap comparison may err in a comparison of x_i and x_j, if $|rank(x_i) - rank(x_j)| \leq k$, where $rank(x)$ is the number of elements in S that are smaller than x.

As a measure for the quality of the outcome $x_{s(1)}, \ldots, x_{s(n)}$ of a sorting algorithm, we count the number of inversions, i.e.:

$$I = |\{(i,j) : i < j, x_{s(i)} > x_{s(j)}\}|$$

Lemma 22 *Any sorting algorithm using cheap comparisons only may produce a result with $I = ((k-1) \cdot n)/2$ inversions.*

Proof. Let $x_1, \ldots, x_n$ be the elements to be sorted (in increasing order). Group them into n/k groups $G_0, G_1, \ldots, G_{n/k-1}$ of adjacent elements, i.e., $G_i = \{x_{k \cdot i+1}, \ldots, x_{k \cdot i+k}\}$. An algorithm cannot distinguish between the elements in one group and hence may output them in decreasing order even if all comparisons between elements of distinct groups are correct. Each group then contributes $(k \cdot (k-1))/2$ inversions. □

An immediate consequence of this lemma is the following corollary:

Corollary 2 *In our model, any sorting algorithm requires $\Omega(n \cdot \log k)$ expensive comparisons to exactly sort a sequence of n elements.*

Proof. We only need to observe that $O(k \cdot \log k)$ expensive comparisons are needed for each group of size k to obtain a correct result. □

An (almost) sorted sequence containing I inversions can be sorted using (2,4)-finger search trees with $O(n \cdot \log(2 + I/n))$ expensive comparisons or using insertion sort with $O(n + I)$ expensive comparisons ([Meh84]). Hence, if we can prove a sorting algorithm to produce O$(k \cdot n)$ inversions when using cheap comparisons only, we can combine this algorithm with (2,4)-finger search trees to an exact sorting algorithm which is optimal with respect to the number of expensive comparisons.

Now we turn to actual sorting algorithms. In particular, we will we will examine mergesort, quicksort and heapsort (according to the description in [CLR90])when executed with cheap comparisons. It turns out, that quicksort is optimal whereas mergesort is suboptimal. Heapsort may be optimal, but we can only prove a suboptimal bound.

3.2.1 Mergesort

Lemma 23 *Mergesort with cheap comparisons produces a result with at most $k \cdot n \cdot \log n$ inversions.*

Proof. We show that for a (by mergesort possibly incorrectly sorted) list $x_1 x_2 x_3 \ldots x_n$ and elements x_i, x_j, $i < j$, we have $rank(x_i) \leq rank(x_j) + k \cdot \log n$. The lemma follows immediately.
We use induction on the number of merging levels. Level 0 with $n = 1$ is trivial. Now assume we have two lists $x_1 x_2 \ldots x_{n/2}$ and $x_{n/2+1} \ldots x_n$ which we want to merge. Consider w.l.o.g. an element x_j from the first list. By induction hypothesis, all elements x_i, $i < j$ have rank at most $rank(x_j) + k \cdot \log n/2$. So the largest element of the second list that is moved to the result list before x_j can have at most rank $k + rank(x_j) + k \cdot \log n/2 = rank(x_j) + k \cdot \log n$.
$\square$

Lemma 24 *For $k = 1$ mergesort may produce $\Omega(n \cdot \log n)$ inversions (with cheap comparisons).*

Proof. Let $x_1 x_2 \ldots x_n$ be the result sequence of mergesort. The idea of the proof is that we construct an input for mergesort and the outcome of all comparisons such that there are l disjoint subsequences of length $d \approx \frac{n}{l}$, where each of these subsequences is decreasing. Hence we get about $d^2 \cdot l$ inversions in the resulting sequence. For $l = n/\log n$ and $d = \log(n/\log n)$ this is $\Omega(n \cdot \log n)$. Note, that only comparisons of elements may err whose ranks differ by one.
We construct the input recursively. Let L be the set of sequences $\{L_1, L_2, \ldots, L_l\}$ where $L_i = \{x_{i_1}, x_{i_2} \ldots, x_{i_d}\}$ with $x_{i_j} = x_{i_{j-1}} - 1$ for $j = 2 \ldots d$. And for all $i \neq j$, $L_i \cap L_j = \emptyset$. We now look at the complete binary tree representing the computation of mergesort.
Starting at the root, we distribute the contents of the sequences to the subtrees. From each sequence L_i we send the first element to one subtree and the remaining sequence to the other subtree.
More formally, each node v with children v_{left}, v_{right} is given a set of sequences $S_v = \{L_1^v, L_2^v, \ldots, L_m^v\}$ and a set of "processed" elements E_v. For the *root* we have $S_{root} = L$ and $E_{root} = \emptyset$. Intuitively, E_v are the elements to be distributed amongst the leaves of the subtree rooted at v.
The procedure for a node v works as follows: first we partition the set E_v into two sets of equal size $E_v = E_{v_{left}} \uplus E_{v_{right}}$. We send the heads of the first $m/2$ sequences to the left child node, i.e., $E_{v_{left}} := E_{v_{left}} \cup \{head(L_i^v) | i = 1 \ldots m/2\}$ and the tails to the right child node, i.e., $S_{v_{right}} := \{tail(L_i^v) | i = 1 \ldots m/2\}$. The same the other way around with the second half of the sequences, i.e., $E_{v_{right}} := E_{v_{right}} \cup \{head(L_i^v) | i = (m/2) + 1 \ldots m\}$ and $S_{v_{left}} := \{tail(L_i^v) | i = (m/2) + 1 \ldots m\}$.

It is easy to see that the number of elements in E_v of a node v on level k is $e_k = k \cdot l/2^k$, the number of elements in S_v of the same node v is $s_k = l/2^k$. Our construction goes through as long as $e_k, s_k \geq 2$. Hence for a given l, the upper bound d for k is given by $d = \log l$. So we can choose $l = n/\log n$ and $d = \log(n/\log n)$. As $d < \log n$, our construction ends a few levels above the leaves. We then distribute the elements of each E_v arbitrarily among the leaves of the subtree rooted at v and assign arbitrary values to the still unoccupied leaves.

It remains to show that each of these sequences in L appears in the resulting sequence of mergesort in reverse (i.e. decreasing) order. This can be easily seen by induction on the merge steps where such a sequence "participates" with some of its elements.

Let us consider a sequence L_i. When we merge sequences s_1, s_2, some elements $S \subset L_i$ may be present in s_1 or s_2. If so, exactly one, the largest element x_1 of S is in one sequence – let's say w.l.o.g. in s_1 – and all the rest of S, i.e. $x_2, x_3, \ldots, x_{d'}$ ($x_i = x_{i-1} - 1$ for $i = 2 \ldots d'$), is in s_2 and by induction hypothesis in reverse order. As we assume that elements of different sequences L_i, L_j are compared correctly, the elements of S present in s_2 are not interleaved with elements of other sequences L_j. Again, as elements of different sequences are compared exactly, there will be a point in the merging process of s_1 and s_2 where x_1 is compared with x_2. This comparison may err since $x_1 = x_2 + 1$ and hence x_1 is moved to the result sequence before x_2, i.e. S ends up in reverse order in the result sequence of this merging step.

□

For $k = 1$, our upper and lower bound have the same order. We leave it as an open problem to prove a lower bound for $k > 1$. The lower bound shows that mergesort is not optimal. The running-time is clearly not affected by the use of erroneous comparisons.

3.2.2 Quicksort

Lemma 25 *Quicksort (with cheap comparisons) produces a list with at most $2 \cdot k \cdot n$ inversions.*

Proof. We show that for a fixed element y, the rank of an element x right of y in the result of quicksort is greater than $rank(y) - 2k$. This implies that the number of such pairs (y, x) where $x < y$ is at most $2k$.

If $x < y$, but x ends up to the right of y then there must be a node z at which y is routed to the left or $y = z$ and x is routed to the right or $x = z$. The element z is either smaller than x, equal to x, lies between x and y, is equal to y, or is larger than y.

In the first case the comparison between z and y is incorrect and hence the ranks of z and y differ by at most k. Since x lies between z and y the ranks z and y differ by at most k. The last case is symmetric.

In the second case the comparison between y and x is incorrect and hence the ranks of x and y differ by at most k. The next to last case is symmetric.

In the third case the comparisons between x and z and between y and z are incorrect and hence the rank of either element differs by at most k from the rank of z. Thus the rank of x and y differs by at most $2k$.

□

This lemma shows that quicksort is optimal up to a constant factor with respect to robustness against imprecision of the comparison operation.

It is not obvious that the expected number of comparisons of quicksort is still $O(n \log n)$. The standard argument is that the rank of the root is a random integer in $\{1, \ldots, n\}$ and hence we get balanced subproblems. This argument does not hold any longer since comparisons may be incorrect. The argument is basically correct as long as the number of elements in a subset is much larger than k, say larger than $5k$. Once a subset is smaller than $5k$ the depth of the resulting tree is at most $5k$ and hence the depth of the entire tree is $O(k+\log n)$. The number of cheap comparison required by quicksort is therefore $O(n \cdot k + n \log n)$. We next improve the bound to $O(n \cdot \log n)$

Consider the following directed graph on S. We have an arc from x to y if x is declared smaller than y by a cheap comparison. The indegree of a node is then the number of elements that are declared smaller and the outdegree of a node is the number of elements that are declared larger. For each node the sum of the indegree and the outdegree is equal to $n-1$. The total indegree is equal to the total outdegree; both are equal to $n(n-1)/2$, the number of arcs.

The claim is that in any such graph the number of "middle" elements, i.e., those elements which have their indegree as well as their outdegree bounded by $7n/8$ is at least a fixed fraction of the elements. Here is a proof.

Partition S into sets A, B, and C, where A contains all elements whose outdegree is at least $7n/8$, C contains all elements whose indegree is at least $7n/8$, and B contains the remaining elements. For an element in B the indegree and the outdegree are bounded by $7n/8$.

Lemma 26 $|B| \geq n/10$.

Proof. Assume that $|B| < n/10$. Also assume that $|A| \geq |C|$. Then $|A| \geq (n - n/10)/2 = 9 \cdot n/20$ and hence $|B| + |C| \leq 11 \cdot n/20$. Each $x \in A$ has an outdegree of at least $7 \cdot n/8$; at most $11 \cdot n/20$ of its outgoing edges can end in $B \cup C$ and hence at least $(7/8 - 11/20) \cdot n > n/8$ edges have to end in A. Since every node in A has more than $n/8$ outgoing edges to nodes in A there must be at least one node in A whose indegree is larger than $n/8$, a contradiction to the definition of A.

$\square$

The Lemma above shows that at least $n/10$ elements are good splitters and hence the recursion depth of quicksort is $O(\log n)$ with high probability; see [MR95]. Thus quicksort uses $O(n \log n)$ cheap comparisons with high probability.

3.2.3 Heapsort

Lemma 27 *In our model starting with a correct heap, heapsort (with cheap comparisons) produces a result with at most* $2 \cdot k \cdot n \cdot \log n$ *inversions.*

Proof. Heapsort operates in phases. In each phase it outputs the root of the heap, moves the key of a leaf into the root and lets the element sink down to its correct position by a sequence of downheap operations. We show that at the beginning of each phase and for each node n and its children c_i, $i = 1, 2$:

$$rank(key[n]) - rank(key[c_i]) \leq 2 \cdot k$$

where $key[x]$ denotes the key stored at node x. It follows that the maximum rank of an element within the heap is $rank(key[root]) + 2 \cdot k \cdot \log n$, and hence each phase can create at most $2 \cdot k \cdot \log n$ inversions. The lemma follows.

Let n be a node in the tree, c_1, c_2 its children, p its parent and $key'[x]$, $x \in \{p, n, c_1, c_2\}$ the key stored at x after a downheap operation on node n.

We show that after a downheap operation on node n,

$$rank(key'[p]) - rank(key'[n]) \leq 2 \cdot k$$

$$rank(key'[n]) - rank(key'[c_i]) \leq 2 \cdot k$$

and if there was a swap with child c_s, $s \in \{1, 2\}$

$$rank(key'[n]) \leq rank(key'[c_s]) + k$$

As the downheap operation before the current one has kept the above invariant, we know that $rank(key[p]) - rank(key[n]) \leq k$. We now compare $key[n]$ with $\min(key[c_1], key[c_2])$. If no swap happens, we know that $rank(key[n]) \leq rank(key[c_i]) + 2 \cdot k$ and the downheap operation stops.

If a swap happens with let's say c_1, we have: $rank(key'[n]) \leq rank(key'[c_1]) + k$ and $rank(key'[n]) \leq rank(key'[c_2]) + 2 \cdot k$. Hence also $rank(key'[p]) - rank(key'[n]) \leq 2 \cdot k$. The downheap operation continues with node c_1.

$\sqcup\!\!\!\!\sqcap$

A correct heap can be constructed with a linear number of expensive comparisons. Heap building with inexact comparisons also yields a heap which satisfies for any node n and its children c_1, c_2, $rank(key[n]) - rank(key[c_i]) \leq 2 \cdot k$.

Summary: We showed that quicksort is optimal in our model up to a constant factor, and that mergesort is suboptimal. For heapsort we leave the exact behaviour as an open question.

With a repair step – either finger search trees or insertion sort –, quicksort allows exact sorting of a sequence with $O(n \log k)$ (using finger search trees) or $O(k \cdot n)$ (using insertion sort) expensive comparisons. The former bound is optimal as we have proved in corollary 2.

Note that for this application, incorrect comparisons *always* require a repair later on. So we can only gain by saving the cost of computing the error bound and possibly some exact arithmetic computations where the error bound is too weak to prove the correctness of a (correct) floating-point result.

3.3 Searching

In a comparison based search structure which is a directed acyclic graph (e.g. a tree), we can use cheap comparisons during the location of a new point without taking the risk of looping. The only thing we have to make sure is that there is an easy way to get from a possibly incorrect result of the search to the correct result.

In the following we will consider binary search trees and a search structure for point location during the randomized incremental construction of the Delaunay Triangulation of points in the plane.

3.3.1 Binary Search on Trees followed by Linear Search through the Leaves

Consider a comparison based search structure for a linearly ordered set S of objects $x_1 < x_2 < \ldots x_n$. We use x_0 and x_{n+1} to denote the fictitious points $-\infty$ and $+\infty$. The search structure divides space into $2n+1$ cells, n cells corresponding to the points in S and $n+1$ cells for the open intervals between adjacent points in S. There is a natural linear order on the cells. Each cell is either a closed or an open interval. In the linear arrangement of the cells open and closed cells alternate and the extreme cells are open. The following lemma bounds the maximal "error" of a search in terms of the set of points whose comparison with the query point is erroneous. It assumes that all comparisons are between the query point and points in S. All comparison-based realizations of dictionaries have this property.

Lemma 28 *Consider a query point q and let i be such that $x_i < q < x_{i+1}$ or $x_i = q$. If the comparisons between q and x_j are correct for $|i-j| \geq k$ then the cell delivered by a search for q has distance at most $2k$ from the cell containing q.*

Proof. Assume that a search for q produces a cell C' different from C. We may assume w.l.o.g. that C' is to the left of C. Then q was compared with the right endpoint, say x_j, of C' and the outcome of this comparison was erroneous. There are at most the cells x_j, (x_j, x_{j+1}), …, x_i between C' and C. By our assumption we have $i - j < k$ and hence the distance between C' and C is at most $2k$.

□

Under the assumptions of the preceding Lemma the cost of a search for q is $O(\log n)$ cheap comparisons plus $O(k)$ expensive comparisons.

As for sorting, we remark that incorrect decisions always lead to a repair step at the end; so we only may gain by not having to compute the error bounds and possibly some exact arithmetic evaluations due to the error bound being too weak.

3.3.2 Point Location for Delaunay Triangulations

In the randomized incremental algorithm for computing the Delaunay triangulation of a set of points in the plane, a search structure is maintained to locate each new point to be inserted in the current triangulation. This is usually implemented as a history graph, which is a directed acyclic graph recording all insertions and flips executed in the algorithm so far. Again, we can perform all comparisons cheaply and still get to some sink corresponding to a triangle. Then we have to check whether the query point in fact lies inside this triangle. If not, we walk across one side of the current triangle whose inequality was violated to an adjacent triangle. We continue like that until we reach the correct triangle.

We remark that even if some comparisons are incorrect, the correct triangle may still be reached directly (see Figure 3.1). So the potential gain in running time is due to saving error bound computations as well as exact arithmetic evaluations of non-crucial predicates.

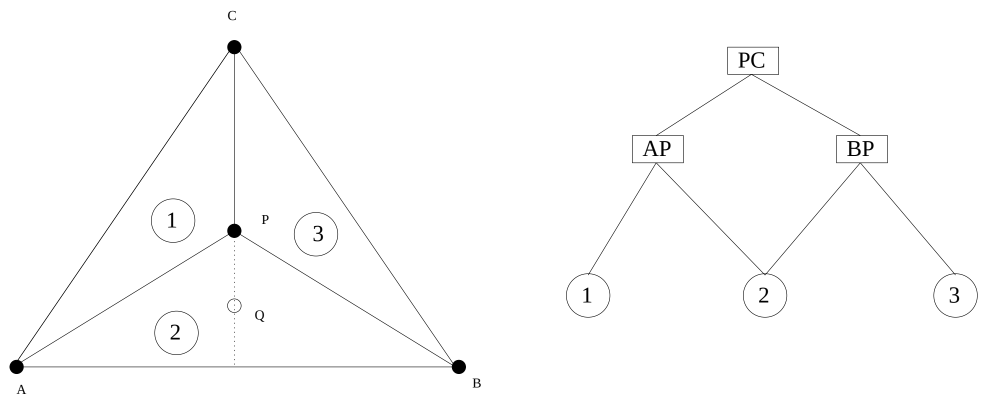


Figure 3.1: When locating Q, the orientation of Q w.r.t $\overrightarrow{PC}$ is not important.

3.4 Experimental Results

We performed two experiments to evaluate the benefits of structural filtering. In the first experiment we sorted points lexicographically and in the second experiment we computed the Delaunay triangulation of a set of points. For both experiments we used the rational geometry kernel of the LEDA system [MN00]. In this kernel, points (type `rat_point`) are represented by homogeneous coordinates of type `integer` (the arbitrary precision integer type of LEDA) and also by floating-point approximations of type `double`. The kernel uses a floating-point filter on the predicate level (see [MN00, Section 8.7]). An exact evaluation of a geometric predicate operates in three steps: (1) Compute the value using floating-point arithmetic, (2) compute an error bound, (3) if necessary, evaluate the predicate using integer arithmetic. A cheap evaluation performs only step (1).

LEDA, Version 3.8, provides means for easily implementing algorithms according to the "structural filtering" paradigm. Using a global flag (`rat_point::float_computation_only`), the programmer can tell the kernel to always take the sign of the floating-point computation (step (1)) when evaluating a predicate. Hence implementing a "cheap" locate procedure just means turning on this flag and performing the location procedure as usual. To check for the correctness of the outcome, the "exact" mode has to be switched on again, of course. Our experience shows that modifying existing implementations to make use of structural filtering usually is a matter of a few minutes, adding just a few lines of code.

The rational geometry kernel of LEDA can be used as a kernel traits class with the algorithms of CGAL and hence structural filtering is also available for programmers using CGAL [CGA99].

3.4.1 Sorting

Sorting a set of points lexicographically is a very common subroutine in many geometric algorithms. We have implemented a "structurally filtered" version of quicksort, i.e., after choosing the splitter, all elements are distributed to the left or right according to a possibly inexact floating-point comparison. A call of quicksort is still guaranteed to return a sorted sequence. This requires the use of a non-trivial conquer-step. The conquer-step is essentially insertion sort of the splitter and the "right" sequence until no swaps take place anymore. In the worst case, this requires $O(k^2)$ comparisons per recursion, but overall, the number of such comparisons is bound by $O(n \cdot k)$ as we have shown. In practice, this turned out to be more efficient than a "repair run" over the final result. Usually, only 2 (exact) comparisons are necessary (to check that the splitter is greater than the rightmost element of the left sequence and smaller than the leftmost element of the right sequence).

Observe, that repairing the final result could be regarded as a filter on "algorithm level", whereas repairing after every recursion is more the "intermediate level filter" we are advocating.

We have compared both implementations, the exact quicksort and the structurally filtered quicksort with a floating-point-only implementation. As input for all three implementations we chose randomly generated `rat_points`. The output was the sequence of points in lexi-

	$1 \cdot 10^5$	$2 \cdot 10^5$	$4 \cdot 10^5$	$8 \cdot 10^5$	$1.6 \cdot 10^6$
qs_exact	2.58	5.65	12.5	28.0	63.4
qs_repair	1.93	4.35	9.56	21.1	49.7
qs_float	1.80	4.05	8.94	19.8	47.2

Table 3.1: Quicksort: total running time in secs, $2 \cdot 10^5$ to $1.6 \cdot 10^6$ points

cographic order. Our experiments show an advantage of about 20–25 % compared to the "normal", exact version of quicksort, which is due to not having to compute the error bounds for most comparisons (see Table 3.4.1). Surprisingly, the version which uses *only* floating-point operations, does not perform twice as fast as the exact, predicate filtered version. This is probably due to cache and memory effects. So the version using structural filtering is only about 5–7 % slower than the floating-point version. See table 3.4.1 for our results.

3.4.2 Randomized Incremental Delaunay Triangulation

We have implemented the randomized incremental algorithm for computing the Delaunay Triangulation of a point set in the plane using the LEDA rational geometry kernel. We call this version **dt_exact** in the following. Then we modified the search structure in our implementation to make use of structural filtering, i.e., we did the comparisons in the directed acyclic Delaunay graph using inexact floating-point comparisons and performed "walking" at the end to guarantee that we reach the correct triangle. We call this version **dt_search**.

Finally, a simple observation allowed us to even perform all incircle tests (which trigger "flips") inexactly. If we guarantee that a flip only takes place in a convex quadrilateral, we always have a valid triangulation. At the end of the algorithm we start the flipping algorithm to make sure that the triangulation we have computed is indeed the Delaunay triangulation. As in the version **dt_search**, we perform the point location with floating-point arithmetic only, followed by "walking". This version is called **dt_flip**.

Why do we hope for an improvement in running time compared to the **dt_exact** version? In the following we assume that floating-point arithmetic always gives the exact result and has cost 1 per predicate evaluation. We also assume that the floating-point filter always can decide the predicate but has cost 2 per predicate evaluation. This is a reasonable assumption on the overhead imposed by current floating-point filter schemes.

For the query structure, instead of $c \cdot \log n$ exact orientation tests – for some constant c –, we have $c \cdot \log n$ floating-point tests followed by three exact orientation tests to verify that we are in the correct triangle. Hence overall we may decrease our cost by $n \cdot ((c \cdot \log n) - 3)$.

For the incircle tests, things are not quite that good. The expected number of incircle tests is about $9 \cdot n$ during the algorithm. Hence the exact algorithm has to pay a cost of $18 \cdot n$. The modified algorithm where the incircle tests are first done in floating-point arithmetic only, has to pay a cost of $9 \cdot n$, but has to perform about $3 \cdot n$ exact incircle tests at the end, to check that the local Delaunay property is fulfilled. Hence overall we can only decrease our cost by $3 \cdot n$ which probably will be negligible.

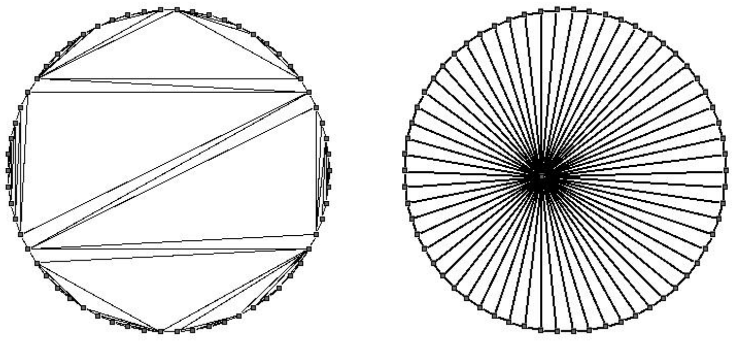


Figure 3.2: Incircle tests are not important, if a center point is inserted later on.

In both cases, though, a considerable gain in performance can be achieved if there were tests which required arbitrary precision when done exactly, but are not important for the outcome of the algorithm. An example for this phenomenon was given for the query structure in Figure 3.1. For the incircle tests, imagine that in the set of input points there is a subset of more than 3 points lying (almost) on a circle. As long as no point inside this circle is inserted, all tests involving triangles of 4 of these points are (nearly) degenerate and hence are hard to decide by the floating-point filter on predicate level. Nevertheless the outcome of any of these tests does not affect the final result at all as these edges are "flipped away" later-on when a point inside the circle is inserted (see Figure 3.2).

The results of our experiments can be found in Tables 3.2, 3.3, 3.4 and 3.5. As input data we used `rat_points` with homogeneous integer coordinates of different bit-lengths. As to be expected, for random inputs (Table 3.2), the **dt_search** version gains about 10-15 % in the overall running time against the **dt_exact** version, due to not having to compute the error bounds for most predicates. The **dt_flip** version, though, performs much worse since the additional check over all edges of the triangulation is rather expensive in that case, even if no flips take place. A similar result can be observed for input data on a grid (see Table 3.3), but here, the advantage of inexact search is even bigger than in the random case.

Looking at the location time only, we have a difference in running time of 20-29 % between the exact and "structurally filtered" search (see Table 3.4).

For points near a circle, the picture changes drastically (see Table 3.5). Here the **dt_flip** version performs much better than the two other versions, and since the dominating cost are the incircle tests (almost all of them are "difficult", i.e., require exact arithmetic) the **dt_exact** and **dt_search** version do not differ significantly in their running times. The **dt_flip** version performs more than 30 % better than the other two implementations, since there are many difficult tests during the algorithm which are not important for the final result. Note that this difference increases substantially (up to a factor of 3!) if we place one additional point for example in the center of the circle.

	32	40	52	80	100	128
dt_exact	194	195	192	197	194	198
dt_search	174	170	169	171	170	175
dt_flip	204	204	201	204	206	207

Table 3.2: Delaunay Triangulation: running time in secs; 400000 random points, 32–128 bit

	32	40	52	80	100	128
dt_exact	208	216	228	268	351	462
dt_search	177	188	197	233	314	402
dt_flip	216	232	246	290	591	645

Table 3.3: Delaunay Triangulation: running time in secs; 600x600 grid, 32–128 bits

	grid	random
dt_exact	90	86
dt_search	64	67

Table 3.4: Point location time in secs, 40bit, 600x600 grid and 400000 random points

	32	40	52	80	100	128
dt_exact	75.4	74.7	74.8	75.2	75.1	75.8
dt_search	73.0	72.8	73.0	73.3	73.1	72.0
dt_flip	48.2	48.3	48.4	47.7	48.3	48.5

Table 3.5: Delaunay Triang.: running time in secs; 100000 points near a circle, 32–128 bits

3.5 Discussion

We have presented a simple filtering scheme which can be used in addition to (or maybe instead of) the well-known predicate filtering when implementing geometric algorithms. The main idea is to allow predicate decisions to be erroneous but still guarantee a correct final result. Of course, this requires *some* predicates to be evaluated exactly. But the number of those predicates can be kept rather low as we have shown.

As we have seen in our experimental results, running time can be improved either due to fewer error bound computations (as in the example of quicksort), or due to exact computations saved because the result of the predicate is not important (Delaunay triangulation of points near a circle). The gain in performance varies from 20 % (quicksort and point location in Delaunay triangulation algorithm) to 30 % (inexact flipping during the insertions).

Our idea is generic in a sense that it can be applied to almost all algorithms whose operation can be divided into location and update procedures. *Structural filtering* addresses the location stage, which usually dominates the running time for incremental algorithms. The next chapter will focus on how to make the update stages more efficient and also deals with the efficient construction of geometric objects.

Chapter 4

LOOK – a Lazy Object-Oriented Kernel for Geometric Computation

4.1 Designing a Kernel for Exact Geometric Computation

Geometric Algorithms consist of different layers, the bottommost layer – the *kernel* – being basic objects and predicates, and the arithmetic used to do the computations. The next layer are basic algorithms and datastructures. Further layers convert data to different representations, to output, and from input. A geometric algorithms library provides components for the different layers. Ideally, the layers should only communicate using a well-defined interface, such that replacing one implementation of a layer by another should not affect the functionality of the whole system. While the geometric part of the LEDA library [MN00] still has a rather monolithic design, the CGAL library (Computational Geometry Algorithms Library [CGA99]) is completely modular in way that single layers can be replaced by alternative implementations.

In this Chapter we will be only concerned with the design of a *kernel* for a geometric algorithms library, i.e. the bottommost layer where basic objects like points, lines and predicates like the sidedness test are implemented. We will discuss the suitable representation, storage, and construction of geometric objects as well as the use of floating-point filter techniques.

In particular the last item will be discussed and treated extensively. The difficulty of integrating floating-point filter techniques in a kernel design lies in the fact that in "usual" kernel designs (e.g. LEDA or CGAL) geometric objects like points and lines are represented by their coordinates (either cartesian or homogeneous) and hence the construction of an object requires the *exact* computation of its coordinates. Otherwise there is no way we can guarantee the correct evaluation of each predicate, as in the worst case (i.e. when all filter stages fail), the exact representation of the involved objects must be available.

The exact computation of object coordinates has two undesirable consequences. On one hand the running time increases dramatically when operating with constructed objects due to the increased numerical complexity, but maybe even more importantly, the space consumption grows rapidly. So in practice, users either write their own routines to allow for filtering on construction level (which can be supported by tools like EXPCOMP [BFS98]) or use rounding schemes after every construction to keep the numerical complexity and space requirements low, see [For99] for example. Note though, that these rounding schemes actually require a proof that they do not affect the final result considerably. These proofs are non-trivial and usually cannot be generalized. So each application has to be considered separately.

In this Chapter we present a new kernel design called LOOK (Lazy Object-Oriented Kernel for geometric computation) which supports filtering also for geometric constructions. The main idea is that geometric objects are not represented by their coordinates, but by the geometric operation that produced them. Hence exact computation of the coordinates is still possible, but does not have to be performed on construction, and only if needed at all.

In section 4.2 we will briefly survey the features of LOOK which are fairly standard and match existing kernels for geometric computation. Section 4.3 is the core of this Chapter, as it discusses in detail the concepts of the implementation that allow for filtering on geometric object level. Section 4.4 shows how to use and extend the framework provided by LOOK. Finally, Section 4.5 gives extensive experimental results, which show the benefits of lazy-construction for nested computations.

4.2 Features of LOOK

Like the LEDA RatKernel [MN00], the standard geometric kernel in LEDA, which builds upon exact homogeneous integer representation, or the CGAL kernels, LOOK provides class representations for 2d points and may other 2d geometric objects that can be constructed from points, like circles, lines, segments, etc. In LOOK they are called `OPOINT, OCIRCLE, OLINE, OSEGMENT` , etc.

Similar to the CGAL kernels when parameterized with `leda_real` as representation type ([BFMS99]), LOOK also provides support for computation with algebraic geometric objects as they occur for example when intersecting circles. (Actually we only allow geometric objects created by a subset of the algebraic numbers, namely the numbers that can be generated as the result of an expression involving $+, -, \cdot, /$ and k-th root.)

The geometric constructions supported by LOOK currently include line (segment) intersection, computing the center of a triangle, intersection of circles, etc. Of course, all common geometric predicates like orientation test, incircle test etc. are also available.

Programmers can easily replace the default LEDA RatKernel in LEDA's geometric algorithms by LOOK. Due to the genericity of CGAL, it is also no problem to use LOOK as a kernel implementation for the algorithms in CGAL.

4.3 Implementation Concepts

What distinguishes our kernel from other existing kernels is the idea of 'bookkeeping' on object level. This approach was already suggested in [FV96] as 'lazy constructor evaluation', but so far if a programmer wanted to use it, he had to hand-code everything — from the object representations to the predicates. We have developed a tool called EXPCOMP [BFS98] which supports implementations with lazy constructor evaluation but it does not provide a general framework for geometric computation; the programmer still has to implement all predicates and constructions from scratch.

The main idea of our implementation is the following: when we construct a geometric object, we do not compute its coordinates exactly, but only their floating-point approximations and also store references to the objects which were involved in the construction. If later a more precise or exact coordinate representation of the object is required, we can compute this using this 'history' of the object. So with every geometric object we have a so called *object dependency graph* associated, which is a directed acyclic graph recording the 'construction' of the object. Every node of this directed acyclic graph corresponds to a geometric construction. Figure 4.1 shows the object dependency graph for a point object resulting from a line segment intersection of two segments $\overline{PQ}$ and $\overline{RS}$ followed by a centerpoint computation of the intersection point and S and T.

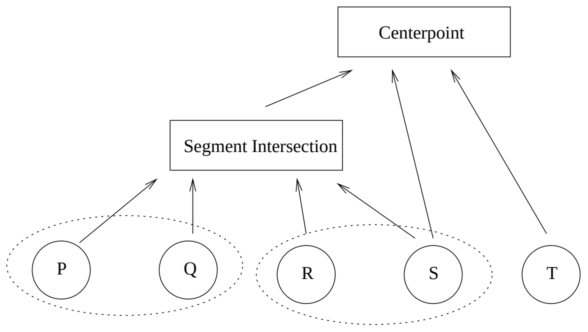


Figure 4.1: Object dependency graph for a segment intersection followed by a center computation

The same approach, but on the level of operators in arithmetic expressions is taken in the number type LEDA `real`. First, one only computes a floating-point approximation for the expression, but also builds an acyclic directed graph recording its 'construction' with the nodes corresponding to arithmetic operators. If the current floating-point approximation of an expression value does not suffice e.g. for a sign determination, one can recompute a better approximation using this 'history' information. But as it was already mentioned in [FV96] and can be seen in [BFMS99], this bookkeeping on arithmetic operator level has its cost. An overhead factor of around 10–50 compared to pure floating-point computation can be expected just for computing the floating-point approximation and storing the history information.

When performing 'bookkeeping' on object level, the overhead occurs only once for every geometric construction instead of every arithmetic operation, so we can expect a considerable gain in running time. But what can we expect when comparing with the present kernels for rational geometric objects like the LEDA RatKernel or the parametrized kernels of CGAL? Clearly there is a potential for improving the performance, too, as they *always* perform geometric constructions exactly using a datatype for exact/arbitrary precision arithmetic; only the predicate evaluations are tuned using floating-point filter mechanisms. Especially when we deal with deeply nested geometric constructions, where the exact constructions get very expensive, our approach should pay off.

When designing a geometric kernel, it is very important to wrap these techniques as transparently as possible to the programmer. In the following we will touch upon the main issues in our kernel implementation and how it is wrapped up. Note that the core object of our kernel is the point (we call it `OPOINT`), since all other geometric objects in our kernel are represented by tuples of points. This does not always seem very natural for example when a line is given by its line equation $ax + by = 1$, but observe that in this case, one can obtain a point-based representation of the same numerical complexity, namely the points $P_1(0, 1, b)$, $P_2(1, 0, a)$. Things get more difficult, though, when it comes for example to the representation of circles. They can be represented by three points on the circle, one centerpoint and one point on the circle, or by a centerpoint and the radius. Switching between these representations might increase the numerical complexity, so care has to be taken when choosing the most appropriate representation for the given application. Of course, one might consider adding other core objects to the kernel if the necessity arises.

4.3.1 Floating-point filters

The use of floating-point filters to speed up the exact evaluation of predicates has become standard when implementing geometric algorithms. If a predicate is expressed as the sign of an arithmetic expression, the idea is first to evaluate this expression using floating-point arithmetic but also compute an error bound to determine whether the outcome reliable.

According to the way this error bound is computed, we can classify floating-point filters into *static* (error bound is computed completely before runtime), *semi-static* (the error bound is partly computed before runtime) and *dynamic* (the error bound is computed completely at runtime). Obviously, static error bounds incur the least overhead at runtime but tend to be rather weak, whereas the dynamic bounds are typically pretty tight but incur a larger overhead. Examples for these techniques can be found in [BBP98, BFS98, DP99, FV96, KLN91]

As we are dealing with possibly nested computations, using static filters seems to be a bad idea because after few levels of computation their error bounds get very imprecise. So we decided to use dynamic filtering techniques for all constructions and predicates, but also use semi-static filters as the very first step in predicate evaluations. In the actual implementation, we use the new LEDA datatype `INTERVAL` which is the interval arithmetic implemetation from [BBP98] making use of the IEEE rounding modes to keep the overhead compared to floating-point computation relatively low (around a factor of 4–6) and – for some predicates – the preprocessing tool EXPCOMP [BFS98] which automatically generates semi-static filter code (which usually has an overhead of only about 1–2). For this purpose, we had to slightly modify the version of EXPCOMP presented in [BFS98] to make it compatible with the `INTERVAL` datatype.

4.3.2 Cartesian and Homogeneous Representations

When designing a geometric kernel, one can choose between a representation of the points in the plane using *cartesian* or *homogeneous* coordinates. With cartesian coordinates, a point is represented by two (integer, rational or algebraic) values p_x, p_y. With homogeneous coordinates, the point is represented by three values p_X, p_Y, p_W s.t. $p_x = p_X/p_W$ and $p_y = p_Y/p_W$.

One advantage of the homogeneous representation is the fact that if we restrict ourselves to rational objects, we can perform all computations using integer arithmetic without divisions. This is the approach taken in the LEDA RatKernel. There is one problem, though: if the computation is nested, the homogeneous coordinates tend to get very large. Consider the following example:

Given three points p, q, r represented using homogeneous integer coordinates of bitlength 16. Compute the center c of these points. The expressions for computing the homogeneous coordinates of c have degree 6, so the bitlength of c_X, c_Y,c_W will be around 96.

This becomes especially a problem if we use this homogeneous representation for the floating-point approximation of c as well. Since an incircle test expressed in homogeneous coordinates has a degree of 12, we would get a value of bitlength more than 1100! Remembering that a variable of type `double` according to the IEEE standard [IEE85] can have a value of at most 2^{1024}, we see that in these cases, a floating-point filter will *always* fail. Note that this failure is not due to geometric 'difficulty' but only due to the numerical representation.

For this reason we decided to compute the floating-point approximation of a point using cartesian coordinates first, and only if necessary, compute its homogeneous floating-point coordinates (as in some cases they allow for detection of degenerate cases with the floating-point filter, which is not as easy with the cartesian representation). With the cartesian representation, only predicates difficult in a *geometric* sense (i.e. almost or exactly collinear, cocircular, etc.) cannot be decided by the filter.

For the exact representations we decided to store homogeneous `leda_integer` coordinates and cartesian `leda_real` coordinates. Remember though, that – as for the homogeneous floating-point approximations – memory for these representations is only allocated if their value is actually requested and computed. We will go into more detail how this works in section 4.3.4.

Furthermore, we have ensured that arbitrary precision computations are performed using `leda_integer` arithmetic as long as possible. So only objects which have square roots involved in their construction may have `leda_real` representations.

4.3.3 Reference Counting and Handles

Before we define an actual object representation for our points, we have to think about some requirements. It is clear that the points we are dealing with may be defined by input data, but they also might be the result of some geometric construction. Nevertheless, when we use a point later on, we shouldn't have to care about that. Current geometry kernels solve this problem by simply computing the coordinates of the point using exact arithmetic; then there is no difference to a point defined by input data with the exactly those coordinates. But as we do not want to perform the constructions exactly, we have to think about something different.

A first approach would be to write an abstract base class `base_point` which defines the properties of a point. Then for every possible construction, we derive a specialized class from `base_point` . For example a class `doubleC_point` representing points initialized by cartesian double coordinates fills in the necessary functionality based on the double precision input data it is initialized with, another example would be a class `Intersection_Point` , also derived from `base_point` , which fills in the functionality using the data computed by the intersection of two lines.

With this scheme, we could write all predicates in terms of pointers or references to the abstract base class `base_point` and still plug-in pointers/references to different derived instances into the predicates. There are some caveats when using the pointer- or reference-based scheme, though, in particular one has to be very careful when allocating and deallocating memory for the objects.

One possible solution for this problem is the use of *reference counting*. For every distinct point object there exists exactly one *representative* and the programmer can access this representative only via a so called *handle*. If an assignment takes place (which is between handles, then), only the pointer to the representative is copied and a reference counter associated with it is incremented. The representative lives as long as it is referenced by at least one handle. The advantage of this scheme, apart from faster assignments of large objects, is the simplification in memory handling. And, of course, if the same object is copied frequently, the memory consumption is also much lower.

In our concrete implementation, we have a handle class `OPOINT` , which basically just contains a pointer of type `base_point` . `base_point` itself is the abstract base class for all point types. Figure 4.2 gives an overview of the class scheme in LOOK.

So if one wants to add a construction to LOOK (for example computing the midpoint of two points), one has to derive from the abstract base class `base_point` and implement the necessary virtual member functions. As we will see in section 4.4, these member functions basically describe how this midpoint is computed using different arithmetic datatypes.

4.3.4 Lazy Evaluation

4.3.4.1 Constructions

As mentioned, we use a lazy evaluation scheme for all geometric objects in our kernel. This implies that on construction of a geometric object, we only compute its floating-point approx-

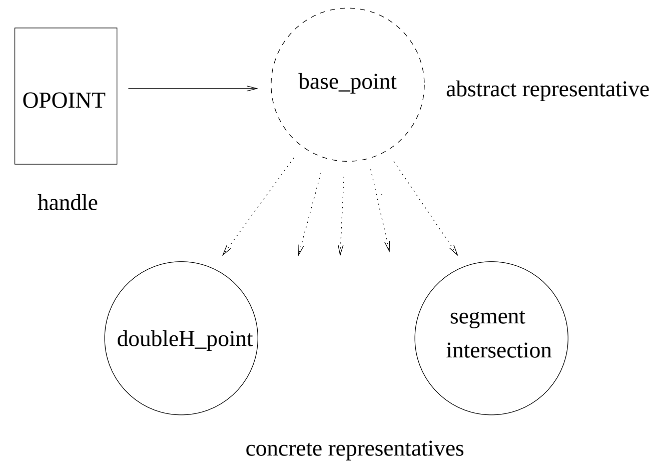


Figure 4.2: Object classes in LOOK

imation and store references to its defining objects. Only if later on, the exact coordinates as `leda_real` or `leda_integer` are requested, they are computed using the information about the defining objects. Of course, this usually triggers an exact evaluation of these defining objects as well.

To wrap the lazy-evaluation functionality in a transparent manner, we allow the programmer to access the coordinates of a geometric object only using member functions. The member function for the exact integer X-Coordinate looks as follows:

```
leda_integer base_point::X()
{
  if (!(Status&COMPUTED_HEXACT))
    SharpenHEXACT();
  return  Ext->_XCoord;
}
```

It is first checked, whether we already have computed the exact homogeneous coordinates, and if not, we compute them by calling the `SharpenHEXACT()` member function. `SharpenHEXACT()` again is a wrapper function:

```
void base_point::SharpenHEXACT()
{
  if (Ext==NULL)  // memory allocation necessary ?
    Ext=new ExtendedBlock;

  ComputeHExact();

  // sharpen homog. FP-APX
  Ext->_XCoordAPX=INTERVAL(Ext->_XCoord);
  Ext->_YCoordAPX=INTERVAL(Ext->_YCoord);
  Ext->_WCoordAPX=INTERVAL(Ext->_WCoord);

  // sharpen cart. FP-APX
  leda_rational xR(Ext->_XCoord,Ext->_WCoord),
                yR(Ext->_YCoord,Ext->_WCoord);
  __xCoordAPX=INTERVAL(xR);
  __yCoordAPX=INTERVAL(yR);

  Status|=(COMPUTED_HAPX|COMPUTED_HEXACT
           |COMPUTED_CAPX);
  Birthday++;
}
```

We first check if we have to allocate the memory for the extended datastructure, since by default we only store (and allocate memory for) the cartesian double approximation of an object. Then we call the actual function computing the coordinates and use this exact value to sharpen the floating-point approximations of the current object. Finally we increment the 'birthday' of the current object. Intuitively speaking, an object is "reborn" each time a more precise coordinate representation of it is computed. The next section will explain the purpose of that in more detail.

Note that the code for triggering the lazy evaluation mechanism is completely encapsulated in the `::X()` and `::SharpenHEXACT()` member functions. If a new class is derived from `base_point` , the programmer only has to provide the member functions which actually compute the coordinates (here: `ComputeHExact()`).

To sum it up, every point object derived from `base_point` in our kernel provides the following member functions for accessing the coordinates in different formats and types:

name	Type	Format
`::x() ::y()`	`leda_real`	cartesian
`::xAPX() ::yAPX()`	`INTERVAL`	cartesian
`::X() ::Y() ::W()`	`leda_integer`	homog.
`::XAPX() ::YAPX() ::WAPX()`	`INTERVAL`	homog.

Keep in mind that because of the lazy evaluation mechanism, the corresponding values are only computed when requested by a call to one of these member functions. Only the cartesian floating-point approximations are computed on instantiation to save a redirection in this case.

4.3.4.2 Predicates

Of course, the predicates must also know about the lazy-evaluation mechanism in the objects they are working on. We will briefly sketch how we implemented the very common orientation test. Note that this code fragment includes special statements which are preprocessed by EXPCOMP ([BFS98]) – a tool for automatically generating efficient floating-point filter code. In this example we use EXPCOMP to generate a first filter stage which uses a semi-static floating-point filter based on the cartesian representation.

```
int orientation(const OPOINT &a, const OPOINT &b,
                const OPOINT &c)
{
 int sgn_res=NO_IDEA;
 BEGIN_FILTER // 1st stage generated by EXPCOMP
 {
 DECLARE_ATTRIBUTES real_apx_type FOR a.x() a.y()
                          b.x() b.y() c.x() c.y();
   exact AX=a.x(); exact AY=a.y();
   exact BX=b.x(); exact BY=b.y();
   exact CX=c.x(); exact CY=c.y();
   exact D=(AX-BX)*(AY-CY)-(AY-BY)*(AX-CX);
   sgn_res=sign(D);
 }
 END_FILTER

 if (sgn_res==NO_IDEA)   // 2nd stage
 {
   INTERVAL AX=a.xAPX(); INTERVAL AY=a.yAPX();
   INTERVAL BX=b.xAPX(); INTERVAL BY=b.yAPX();
   INTERVAL CX=c.xAPX(); INTERVAL CY=c.yAPX();
   fpu::round_up();
   INTERVAL D=(AX-BX)*(AY-CY)-(AY-BY)*(AX-CX);
   sgn_res=msign(D);
   fpu::round_nearest();
 }
 if (sgn_res==NO_IDEA)  // 3rd stage
 {
  if ((a.RatType() && b.RatType() && c.RatType())
  {
     /*homogeneous test using           */
     /*interval-arithmetic              */
     /*accessing a.XAPX() .... c.WAPX() */
    if (sgn_res==NO_IDEA)
      /*homogeneous test using exact    */
      /*exact integer arithmetic        */
  }
  else                  // 4th stage
   sgn_res=sign((a.x()-b.x())*(a.y()-c.y())
              -(a.y()-b.y())*(a.x()-c.x()));
 }
 return sgn_res;
}
```

The evaluation strategy is as follows:

1. cartesian evaluation using a fast filter implementation generated by EXPCOMP (this accesses the `::xAPX()` and `::yAPX()` member functions)

2. cartesian evaluation using interval arithmetic (tighter error bounds, but slower; accesses the `::xAPX()` and `::yAPX()` member functions)

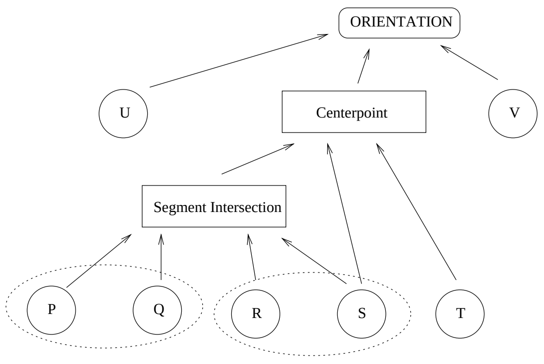


Figure 4.3: Object dependency graph for an orientation test

3. If the involved objects are all of rational type:
   (a) homogeneous evaluation using interval arithmetic (accesses the `::XAPX()` , `::YAPX()` and `::WAPX()` member functions)
   (b) homogeneous evaluation using exact LEDA `integer` arithmetic (accesses the `::X()` , `::Y()` and `::W()` member functions)
4. If they are of algebraic type:
   (a) cartesian evaluation using exact LEDA `real` arithmetic (accesses the `::x()` and `::y()` member functions)

So only if a predicate cannot be decided by the earlier stages, additional time and space is spent on a more accurate computation of the involved objects and the predicate itself.

4.3.5 Progressive Exact Evaluation

Using the lazy evaluation scheme, we first try to evaluate the predicate using floating-point arithmetic and if the outcome cannot be proved to be correct, we trigger an exact computation for all objects involved in that predicate (which in turn triggers exact computations of their 'defining' objects) and then evaluate the predicate using exact arithmetic. But can we do better?

Let us first extend the definition of the object dependency graph and also allow geometric predicates as nodes (actually we only allow them as the root of an odg). Consider the object dependency graph in figure 4.3. Let us define the depth of an object in the object dependency graph as the maximum length of a path from the root to that object.

So far, if the floating-point evaluation of the orientation test fails, exact arithmetic computation of all objects involved (P, Q, R, S, T, U, V, intersection point, centerpoint) is triggered. But it may help only to compute the coordinates of the objects on depth 2 (here: the intersection point) exactly, improve their floating-point approximations using the exact values and then recompute the floating-point approximations of the 'higher' objects (here: the centerpoint). Since we get better error bounds now, this may suffice to decide the orientation predicate. Hence, if the floating-point evaluation of a predicate fails, we proceed as follows:

1. determine the deepest node in the object dependency graph which has not been evaluated exactly; assume its depth is d; if $d = 1$, evaluate the predicate exactly
2. evaluate all objects with depth d exactly and improve their floating-point approximations
3. reevaluate the floating-point approximations of all objects with distance less than d to the root
4. if no decision could be made, start from 1. again

Note that actual 'exact evaluation' is possible only for rational objects. For algebraic objects, 'exact evaluation' means determining the value as a `leda_real` and 'improving the floating-point approximations' means evaluating the `leda_real` representation to double precision.

To allow for an efficient implementation, we store a *birthday* for every object. Objects built of input data have birthday 0. When a new geometric object is constructed (and its floating-point approximation is computed), it gets as birthday the sum of the birthdays of its defining objects. When the `leda_integer` or `leda_real` representations of the coordinates of an object are triggered and the floating-point approximation are improved accordingly, its birthday is incremented by one. Then, in the process of progressive exact evaluation, if an object realizes that the sum of the birthdays of its defining objects is greater than its own birthday, it knows that it can improve its floating-point approximation by just recomputing using the improved floating-point approximations of its defining objects.

We skip this additional code in the predicate evaluation, but it only adds a few lines to the code given for the orientation test.

4.3.6 Conservative Memory Management

So far our strategy is to perform exact computation only on demand, but once computed keep the result such that if later the same result is required, no recomputation is necessary. For some applications, though, where memory consumption is an issue, it may be more appropriate not to keep the exact results, i.e. after a predicate evaluation which possibly triggered a sequence of arbitrary precision computations, we discard the arbitrary precision results and only keep the refined floating-point approximations.

We have incorporated a simple toggle in the LOOK kernel to switch between discarding and keeping results computed using arbitrary precision arithmetic. If a predicate evaluation required arbitrary precision arithmetic and the 'discard' toggle is set, it recursively frees all extended blocks allocated during the evaluation of the predicate.

As it turns out, there are cases where discarding the computed results in a better runtime than keeping them. This is probably due to the cache getting less effective, if memory consumption is very high.

4.4 How to Use it as a Programmer

4.4.1 'Normal' Use

The nice thing about LOOK is the fact that it encapsulates and hides all of the above mechanisms from the programmer. If he does not want to introduce new geometric constructions or predicates, he can use the geometric object representations provided by LOOK in exactly the same manner as if he was using a geometric kernel of LEDA or CGAL.

4.4.2 Extending LOOK

Nevertheless, adding functionality to LOOK is not hard either. For instance adding additional predicates is very easy, given the examples already present in LOOK. They show how the different floating-point filter stages are combined to get best performance. For very efficient code, we also recommend using EXPCOMP [Fun97] for the first filter stage as it was done in case of the orientation predicate.

But even if she wants to extend LOOK beyond the geometric constructions already provided, the effort is not much more than writing these constructions for another kernel, for example the LEDA or CGAL kernels. Basically she only has to derive a new point class from the abstract class `base_point` and implement the member functions which actually compute the coordinates of the new point. With little more effort, she can also support the progressive exact evaluation code. In the following we will consider the simple example of line intersections.

4.4.2.1 Example: Intersection of Lines

For sake of simplicity, we neglect the case where the lines are parallel or identical. Of course, in the actual implementation these cases are treated as well.

We first derive a new class from the abstract representation class `base_point_rep`. An instance of this new class `line_intersection` upon instantiation stores the defining lines, determines its type (either rational or algebraic), triggers the computation of its approximated cartesian coordinates and stores its 'birthday' which is needed for the progressive exact evaluation mechanism.

```
class line_intersection : public base_point_rep
{
  OLINE L1, L2;

  public:
  line_intersection(const OLINE &l1,
                    const OLINE &l2)
  {  L1=l1;L2=l2; SharpenCAPX();
     if (L1.RatType() &&
         L2.RatType())
       Status|=RAT_TYPE;
     Birthday=L1.Birthday()+L2.Birthday();
  }
  private:
    virtual int ComputeHExact();
    virtual int ComputeHApprox();
    virtual int ComputeCExact();
    virtual int ComputeCApprox();

    virtual void Reincarnate();    //optional
    virtual int DeepestInexact();  //optional
    virtual void SharpenAtDepth(int d); //optional
};
```

The programmer only has to implement the virtual `ComputeXXX()` member functions as they are called from the lazy-evaluation mechanism of the parent `base_point_rep` class. Optionally, if the new class should also make use of the progressive exact evaluation scheme, implementations for the `Reincarnate()` , `DeepestInexact()` and `SharpenAtDepth()` member functions can be given, though these implementations are almost generic and only depend on the number and type of 'defining' objects.

As we neglect the degenerate cases of identical or parallel lines, the implementation of our `IntersectLines()` [1] function is trivial. We just initialize a new `OPOINT` -handle with the representative of the line intersection:

```
int IntersectLines(const OLINE &l1,
                   const OLINE &l2,
                   OPOINT &s)
{
  s=OPOINT(new line_intersection(l1,l2));
  return 1;
}
```

Of course, what is really interesting now, are the implementations of the virtual `ComputeXXX()` member functions.

```
int line_intersection::ComputeCExact()
{
  OPOINT S1=L1.Source(), T1=L1.Target(),
         S2=L2.Source(), T2=L2.Target();

  leda_real dx1=T1.x()-S1.x();
  leda_real dy1=T1.y()-S1.y();
  leda_real dx2=T2.x()-S2.x();
  leda_real dy2=T2.y()-S2.y();
  leda_real w=dy1*dx2-dx1*dy2;

  leda_real c1=T1.x()*S1.y() -
               S1.x()*T1.y();
  leda_real c2=T2.x()*S2.y() -
               S2.x()*T2.y();
```

[1] the only thing of this section a 'normal' programmer will ever see

```
  _xCoord()=(c2*dx1-c1*dx2)/w;
  _yCoord()=(c2*dy1-c1*dy2)/w;
  return 1;
}
```

and

```
int line_intersection::ComputeCApprox()
{
  OPOINT S1=L1.Source(), T1=L1.Target(),
         S2=L2.Source(), T2=L2.Target();

  INTERVAL t1x=T1.xAPX(), t1y=T1.yAPX();
  INTERVAL s1x=S1.xAPX(), s1y=S1.yAPX();
  INTERVAL t2x=T2.xAPX(), t2y=T2.yAPX();
  INTERVAL s2x=S2.xAPX(), s2y=S2.yAPX();

  fpu::round_up();
  INTERVAL dx1=t1x-s1x, dy1=t1y-s1y;
  INTERVAL dx2=t2x-s2x, dy2=t2y-s2y;
  INTERVAL w=dy1*dx2-dx1*dy2;

  INTERVAL c1 = t1x*s1y - s1x*t1y;
  INTERVAL c2 = t2x*s2y - s2x*t2y;


  _xCoordAPX()=(c2*dx1-c1*dx2)/w;
  _yCoordAPX()=(c2*dy1-c1*dy2)/w;
  fpu::round_nearest();
  return 1;
}
```

The code for the pair `ComputeHExact() / ComputeHApprox()` is analogous. In fact, the code for computing the approximation is in most cases just a copy of the 'exact' code with the arbitrary precision number type replaced by the interval type.

Note that in the code for computing the approximations, we have to switch rounding mode of the floating-point unit before doing any calculations as the interval type `INTERVAL` relies on the correct IEEE rounding mode when determining upper and lower bounds of the intervals (there is also a version, where the switch of the rounding mode is done implicitly, but it is considerably slower as it has to be done before and after every arithmetic operation), see [BBP98] for more details.

By just providing the above code, points generated by intersection of lines can benefit from all advantages of our framework; in particular, an `OPOINT` constructed using `IntersectLines(...)` can be plugged into any predicate within the LOOK kernel. If the predicate turns out to be 'difficult' the exact evaluation of the intersection point is automatically triggered. The code allowing for that is all inherited from `class base_point_rep`, so the programmer does not have worry about that.

To be complete we state the missing code for the optional `Reincarnate()`, `DeepestInexact()` and `SharpenAtDepth()` member functions.

```
virtual int line_intersection::Reincarnate()
{
  if (Status&COMPUTED_HEXACT)
    return 0;
  L1.Reincarnate(); L2.Reincarnate();
  int NewBirthday=L1.Birthday()+L2.Birthday();

  if (NewBirthday>Birthday)
  {
    Birthday=NewBirthday;
    ComputeCApprox();
    return 1;
  }
  else return 0;
}

virtual int line_intersection::DeepestInexact()
{
  if (Status&COMPUTED_HEXACT) return 0;
  int l1=L1.DeepestInexact();
  int l2=L2.DeepestInexact();

  return MAX(l1,l2)+1;
}

virtual void line_intersection::SharpenAtDepth(int d)
{
  if (d==1) SharpenHEXACT();
  else if (d>1)
  {
    L1.SharpenAtDepth(d-1);
    L2.SharpenAtDepth(d-1);
  }
}
```

Just as a remark, the implementation of the `IntersectSegments()` function is trivial when reducing it to the line intersection:

```
int IntersectSegments(const OSEGMENT &s1,
      const OSEGMENT &s2, OPOINT & p)
{
  int s1_s, s1_t, s2_s, s2_t;
  s1_s=orientation(s1,s2.Source());
  s1_t=orientation(s1,s2.Target());
  s2_s=orientation(s2,s1.Source());
  s2_t=orientation(s2,s1.Target());

  if ((s1_s!=s1_t) && (s2_s!=s2_t))
  {
    p=OPOINT(new line_intersection(s1,s2));
    return 1;
  }
  return 0;
}
```

Note that the orientation tests called from this function are, of course, fully filtered.

4.5 Experiments

In this section we compare implementations of geometric algorithms using our LOOK kernel with implementations based on the LEDA RatKernel and the CGAL kernel. The test platform was a Sun UltraSparc 333 Mhz with 128 MB RAM running Solaris 2.7. We used `g++` 2.95.3, LEDA 4.0, and CGAL 2.2.

4.5.1 LOOK compared to LEDA's RatKernel

In our first example, we examine how different geometry kernels behave when we increase the nesting depth of geometric constructions.

We have tested seven different implementations of Dwyer's divide-and-conquer algorithm for computing the Delaunay triangulation of a set of points in the plane ([Dwy87]). First one using the floating-point kernel of LEDA (FPKernel), then two variants using the rational kernel of LEDA. In both variants, constructions are always performed using exact integer arithmetic; in the first variant (which is the one in the current LEDA version), though, predicates are filtered using a floating-point filter based on homogeneous coordinates, whereas the second variant additionally incorporates a floating-point filter based on cartesian coordinate representation to overcome the problem of overflow as discussed in section 4.3.2.

Finally there are four variants using the LOOK kernel. The first one keeps all results computed using arbitrary arithmetic and uses the progressive evaluation scheme (MaxMem/PEE); the second one also keeps all results but does not use the progressive evaluation scheme (MaxMem/no PEE). The third one deletes all results computed using arbitrary precision arithmetic after each predicate evaluation and uses the progressive evaluation scheme (MinMem/PEE) whereas the fourth variant neither keeps exact results nor uses the progressive evaluation scheme (MinMem/no PEE).

As a benchmark we iterated Voronoi diagram computations. Starting with a point set S_0, in iteration i we determined S_{i+1} from S_i by computing the Delaunay triangulation of S_i and adding the circumcenters of all triangles. By this procedure, the number of elements in S_i roughly tripled in each iteration. Note that about 2/3 of the elements in S_i are created in the $(i-1)$-th iteration, so most predicate evaluations involve data points constructed in the previous stage.

As S_0 we generated a set of N randomly distributed points on a 32bit integer grid such that the final iteration computes the Delaunay triangulation of about 8000 points. To keep the comparison between lazy construction in LOOK and the immediate exact arithmetic construction in the RatKernels fair, we always skipped the construction of the circumcenters in the last iteration. Tables 4.1 and 4.2 show the results for different numbers of iterations. We give both, the total running time and the time spent for constructing the circumcenters. Of course, the variants using LOOK use very little time for construction as they only store the involved objects and compute a floating-point approximation. But if later, during the computation of the Delaunay triangulation in the next iteration, the coordinates are requested in arbitrary precision arithmetic, the arbitrary precision computation is triggered.

We now look in more detail at the (MaxMem/PEE) instance where we initially start with

# of	Initial	FPKernel	RatKernel	
Iter.	N		cart.FP	no cart.FP
1	8100	(0/0.56)	(0/0.58)	(0/0.56)
2	2700	(0.09/0.76)	(0.42/1.11)	(0.43/1.22)
3	900	(0.11/0.81)	(0.75/1.84)	(0.83/73.1)
4	300	(0.11/0.85)	(3.63/103)	(3.95/836)
5	100	(0.12/0.80)	(33.6/1009)	(37.6/8065)

Table 4.1: Iterated Voronoi computations (1); time in secs. (construction/total)

# of	Initial	LOOK			
Iter.	N	MaxMem/PEE	MaxMem/no PEE	MinMem/PEE	MinMem/no PEE
1	8100	(0/1.12)	(0/1.11)	(0/1.21)	(0/1.26)
2	2700	(0.11/1.61)	(0.13/1.60	(0.12/1.91)	(0.14/1.98)
3	900	(0.15/2.07)	(0.16/2.03)	(0.14/2.91)	(0.14/2.99)
4	300	(0.15/3.93)	(0.14/7.67)	(0.16/6.51)	(0.14/12.0)
5	100	(0.15/24.23)	(0.17/51.8)	(0.16/21.6)	(0.15/78.5)

Table 4.2: Iterated Voronoi computations (2); time in secs. (construction/total)

100 points and perform 5 iterations of Voronoi computations. Table 4.3 shows how many of the extended blocks (which store the coordinates of type `leda_integer`) are allocated after the *i*-th iteration. Note that for the 'MinMem' schemes, the number of extended blocks will be at most 100 after every predicate evaluation. Of course, during a predicate evaluation some extended blocks have to be allocated temporarily.

Iter.	#Point Objects	# Extended blocks
2	299	0
3	848	245
4	2528	788
5	7562	4865

Table 4.3: Memory allocation

To compare the running time for 'easy' examples, we computed 2 iterations on a set of 10000 randomly generated points with 32bit integer coordinates. This computation is basically the *crust* computation as described in [ABE98]. See table 4.4 for the results.

It turns out, that if the computation does not involve more than one level of construction, LOOK is about 2–3 times slower than the (tuned) LEDA RatKernel. With increasing nesting depth of the constructions, LOOK performs better and better compared to the LEDA RatKernel.

One reason for that is the use of the lazy evaluation scheme. As can be seen in table 4.3, only some of the constructed points have been evaluated exactly during the algorithm, whereas the RatKernels *always* perform the constructions using exact arithmetic. If we get to higher nesting depths than 5, though, exact arithmetic evaluation of almost all point objects has been triggered. A closer examination of the results showed that this is due to degeneracies which lead to difficult predicate evaluations triggering the exact evaluations of constructions.

FPKernel	RatKernel(1)	RatKernel(2)	LOOK
3.06	5.47	4.47	6.68

Table 4.4: Crust computation of 10000 points

But nevertheless, the LOOK implementations are still much faster in these cases than the RatKernel implementations due to the improved filtering techniques (interval filters which yield closer error bounds than the semi-static filters used in the RatKernels).

Surprisingly, the LOOK variant using the memory saving scheme of deleting all arbitrary precision results after a predicate evaluation together with the progressive exact evaluation performs very well, especially with increasing nesting depth. Although the deletion of all intermediate arbitrary precision computations requires some recomputations, the reduced memory allocation seems to lead to a more efficient caching, so the overall running time is even better. Note that the progressive exact evaluation scheme helps considerably for both memory management schemes.

4.5.2 LOOK as a CGAL Kernel Traits Class

In the CGAL library [CGA99] all algorithms and data structures are generic in the sense that a programmer can plug-in any geometric kernel that meets certain requirements into the algorithms provided by CGAL. But CGAL also provides templates for geometric kernels where the user only has to plug-in a number type that is used for the coordinate representation. For example, by plugging-in the number type `leda_real` for exact arithmetic with algebraic numbers into the cartesian kernel template, we get a kernel for exact computation with algebraic geometric objects.

We have accommodated LOOK to be compatible with CGAL's algorithms by wrapping it in a so-called kernel-traits class. In the following we will compare the performance of the LOOK kernel with other kernels when plugged into the CGAL algorithms. We have tested the following kernels:

- `LOOK` the LOOK kernel as CGAL plugin
- `RatKernel` the LEDA RatKernel as CGAL plugin
- `C<leda_real>` the CGAL cartesian kernel with `leda_real` representation
- `C<double>` the CGAL cartesian kernel with `double` representation [2]

As the first example we computed the convex hull of 50000 random points with 32bit integer coordinates using the default convex hull algorithm of CGAL. Note that this computation does not involve any geometric constructions, so we expect the RatKernel to perform best of the exact kernels. The results can be seen in table 4.5.

[2]Note that this kernel does not guarantee exact computation.

RatKernel	LOOK	C<leda_real>	C<double>
0.38	1.00	3.02	0.16

Table 4.5: ConvexHull of 50000 32bit integer points

A more complex example is the computation of the convex hull of points which are not available as input data, but computed as the intersection of circles. Note that this computation involves square-roots and hence we cannot use the RatKernel for this experiment. Table 4.6 shows the result for the intersections of 500 circles (they have about 26000 intersection points). Also compare with the experimental results in [BFMS99].

	LOOK	C<leda_real>	C<double>
intersection	0.52	7.51	0.11
total	1.08	10.1	0.17

Table 4.6: ConvexHull of the intersections of 500 circles; cartesian integer coordinates

As we have seen in the previous section, we have to pay a little bit for the more involved filtering-techniques, so for very simple examples, where no constructions take place, we lose about a factor of 2–3 compared to the RatKernel, but are still 3 times faster than the cartesian `leda_real` kernel. If constructions take place, though, as in the example for the convex hull of circle intersections, we gain a factor of about 10 compared to the cartesian `leda_real` kernel. As we have predicted in our discussion in the previous sections, this is due to reducing the bookkeeping overhead from expression level to geometric construction level.

4.6 Discussion

We have presented LOOK, a lazy object-oriented kernel design for exact geometric computation. In contrast to previous kernels, LOOK supports various kinds of floating-point filter techniques both on predicate level as well as on construction level. If a problem involves many geometric constructions, LOOK performs about 3–40 times better than the LEDA RatKernel or CGAL kernels for exact computation.

The technique of bookkeeping on object level also allows for many evaluation strategies. If memory consumption is a big issue, one can keep this very low – even close to pure floating-point computation, as the arbitrary precision representation of at most one predicate evaluation is present in memory at any given time. To our very surprise, this approach of discarding all arbitrary precision results after a predicate evaluation performed quite well, especially for more complicated examples. So it seems as if memory allocation is not the main difficulty for deeply nested exact constructions.

Of course, LOOK is not a panacea for exact implementations of geometric algorithms. Although the advanced filtering techniques allow the decision of most predicates without resorting to exact arithmetic computations, the evaluation of really difficult predicates which require exact arithmetic, gets *very* expensive with increasing nesting depth. Here is a point where algorithmic changes may help. There is the idea to design algorithms with only low-degree predicates and thus reducing the numerical complexity, for example [BS99]. On the other hand one could try to reduce the number of arbitrary precision evaluations even further by allowing some of the predicates to err. Of course, as we want a correct final result, it all depends on *which* predicates we allow to err. In [FMN99] we show that a very simple but powerful idea can reduce the number of arbitrary precision evaluations considerably, in particular in (almost) degenerate cases. Part of our future work will be devoted to combining this approach with LOOK to improve performance for deeply nested computations.

Furthermore we want to increase the number of 'algebraic' constructions in our kernel, e.g. the constructions for Voronoi nodes in the Voronoi diagram of points and line segments. Of course it may be also interesting to apply these ideas to construct a kernel for higher-dimensional geometric computation.

Summary

In the first part of this thesis we investigate theoretically a problem from computational geometry. Given a finite set S of sample points from a collection of curves Γ, curve reconstruction is the problem of computing the graph $G(S, \Gamma)$, called the *correct reconstruction*, whose vertex set is S and which has an edge between two samples if and only if they are adjacent on a curve in Γ.

Obviously, it is not possible to correctly reconstruct a given collection of curves from an arbitrary sample set from it. Therefore, some restrictions on the sample set and/or the curves are needed. Several Algorithms have been proposed that provably solve this problem for certain classes of curves. The algorithms differ with respect to (a) whether they can deal with just a single curve or a collection of curves, (b) whether the samples might come only from closed or also from open curves, (c) whether they require non-uniform or uniform sampling (i.e. whether the required sampling density locally adapts to the level of detail or is determined for the whole curve by the most detailed part of the curve), and (d) whether the curves have to be smooth or corners are allowed, i.e. points where left and right tanget disagree.

We present an algorithm which is the most general to date and which under a non-uniform sampling condition provably reconstructs a collection of curves which might be open or closed and contain finitely many corners. Our algorithm is completely combinatorial and runs in polynomial time.

As most previous algorithms for the curve reconstruction problem, our algorithm identifies a subgraph of the Delaunay triangulation of the sample set as the correct reconstruction. The novelty of our approach lies in the fact that for reconstructing corners of a curve, we do not follow a simple filtering approach where for each edge it is locally decided whether it is part of the output or not, but we kind of "walk into" the corners from allegedly smooth areas of the curve. We have first presented this result at the 12th Annual ACM-SIAM Symposium on Discrete Algorithms 2001 ([FR01]).

In the second part of this thesis we are concerned with an issue arising when implementing geometric algorithms. When computer scientists design geometric algorithms, they usually assume the availability of exact arithmetic on real numbers (like we do for example in the first part of this thesis). Since no computer provides exact arithmetic on real numbers in hardware, programmers implementing these algorithms must find some substitution. Quite

commonly, they resort to floating-point arithmetic due to its support by hard- and software. The resulting programs may not behave as expected, though. The roundoff errors accumulating during floating-point computation very often make the programs crash or produce inconsistent results. There are several ways to overcome this problem, first, one can design algorithms which explicitely deal with the problem of roundoff errors. Only very few such *robust* algorithms exist, mainly for very simple problems. The other possibility is the so-called *exact computation paradigm*, which advocates to give the implementer of a geometric algorithm the illusion of exact arithmetic on real numbers by providing exact number types and exact geometric predicates. Unfortunately, providing this 'illusion' has its cost which is usually considerably higher than pure floating-point arithmetic.

The evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. So the naive way to compute the sign of an expression is to compute the value of the expression (using exact arithmetic) and to read off the sign from the value. This way we suffer an overhead of a factor between 10 and 100 compared to floating-point arithmetic. One way to speed up exact geometric predicates is the use of so-called floating-point filters. The idea is first to compute a floating-point approximation together with an error bound for the maximal deviation from the true value. If the error bound is smaller than the absolute value of the approximation, approximation and exact value have the same sign and hence the sign of the approximation may be returned. The advocates of floating-point filters claim that filters at the predicate level realize the exact computation paradigm at little cost; the running time is claimed to be no more than twice the running time of a pure floating-point implementation.

Of course, this statement is only true if the floating-point filter always succeeds in deciding the predicate, and the floating-point filter mechanism can be applied for the whole computation. Very often these conditions are not met in practice, though.

First we develop the concept of *structural filtering* which is a generalization of floating-point filtering. Structural filtering can reduce the overhead compared to pure floating-point arithmetic further by allowing some predicates to err, without sacrificing the guarantee for an exact outcome, of course. Structural filtering views the execution of an algorithm as a sequence of steps and applies filtering at the level of steps. A step can be anything between a simple predicate and the execution of the whole algorithm. As an interesting subresult not restricted to the computational geometry domain, we examine how sorting algorithms behave under erroneous comparisons. We show, for example, that quicksort stays an optimal sorting algorithm when comparisons may err, but mergesort becomes suboptimal. We have presented these results at the 11th Canadian Conference on Computational Geometry 1999 ([FMN99]).

Secondly we develop the design of a geometric kernel called LOOK (Lazy Object Oriented Kernel for geometric computation), which is the first of its kind that makes use of filtering techniques not only on the predicate level but also on the level of geometric constructions. The main idea is to represent geometric objects not by their coordinates, but by the geometric operation that produced them and compute coordinate representations only on demand in a lazy-fashion. We have presented this result at the 16th Annual ACM Symposium on Computational Geometry 2000 ([FM00]).

Zusammenfassung

Im ersten Teil dieser Arbeit untersuchen wir eine theoretische Fragestellung aus dem Gebiet der Algorithmischen Geometrie – das Problem der Kurvenrekonstruktion. Gegeben eine endliche Menge S von Stichprobenpunkten von einer Familie von Kurven Γ, besteht die Aufgabe darin, den Graphen $G(S, \Gamma)$ – auch die *korrekte Rekonstruktion* genannt – zu berechenen, dessen Knotenmenge S ist, und welcher eine Kante zwischen zwei Knoten genau dann besitzt, wenn diese auf einer der Kurven in Γ adjazent sind.

Es scheint unmöglich zu sein, die korrekte Rekonstruktion einer gegebenen Familie von Kurven für eine *beliebige* Stichprobenmenge zu berechnen; daher muß man gewisse Anforderungen sowohl an die Stichprobenmenge als auch an die Kurvenmenge stellen. Verschiedenste Algorithmen wurden in der Vergangenheit entwickelt, welche dieses Problem für gewisse Klassen von Kurven beweisbar lösen können. Die Algorithmen unterscheiden sich hinsichtlich (a) ob eine Menge von Kurven oder nur eine einzelne Kurve rekonstruiert werden kann, (b) ob nur geschlossene Kurven behandelt werden können, oder auch offene Kurven, (c) die Stichproben uniform sein müssen (d.h. ob die erforderliche Stichprobendichte sich lokal an die Detailliertheit der Kurve anpassen kann, oder ob die Dichte global für die ganze Kurve durch den detailliertesten Teil bestimmt wird) und (d) ob die Kurven "glatt" sein müssen oder ob Ecken erlaubt sind, d.h. Punkte, an denen keine Tangente existiert.

Wir präsentieren den bislang allgemeinsten Algorithmus, welcher für eine nicht uniforme Dichtebedingung der Stichproben beweisbar eine Familie von Kurven rekonstruiert, welche sowohl offene und geschlosse Kurven als auch endlich viele Ecken beinhalten darf. Unser Algorithmus ist kombinatorischer Natur und benötigt polynomielle Zeit.

Wie die meisten bisherigen Algorithmen zur Kurvenrekonstruktion identifiziert unser Algorithmus einen Teilgraphen der Delaunay Triangulierung der Stichprobenmenge als korrekte Rekonstruktion. Die Neuerung unseres Ansatzes besteht darin, Ecken dadurch zu erkennen, sich nicht nur auf lokale Betrachtungen zu beschränken, sondern die Ecken ausgehend von glatten Teilen der Kurve inkrementell zu "erschliessen". Dieses Resultat haben wir auch auf dem 12th Annual ACM-SIAM Symposium on Discrete Algorithms 2001 vorgestellt ([FR01]).

Im zweiten Teil der Arbeit beschäftigen wir uns mit einem Problem, welches häufig bei der Implementierung von geometrischen Algorithmen auftritt. Beim Entwurf geometrischer Algorithmen wird üblicherweise die Verfügbarkeit von exakter Arithmetik auf reellen Zahlen angenommen (wir haben diese Annahme z.B. im ersten Teil dieser Arbeit getroffen). Da allerdings kein Computer exakte Arithmetik auf reellen Zahlen durch Hardware unterstützt, müssen Programmierer, die diese Algorithmen implementieren wollen, sich anderweitig behelfen. Sehr häufig greifen sie auf Gleitkomma-Arithmetik zurück, zumeist wegen der guten Unterstützung durch Programmiersprachen und Hardware. Die so erstellten Programme verhalten sich allerdings häufig nicht wie erwartet. Rundungsfehler, die sich während den Gleitkomma-Rechnungen akkumulieren, lassen die Programme häufig abstürzen oder inkonsistente Resultate liefern. Es existieren verschiedene Ansätze, dieses Problem anzugehen; zum einen kann man Algorithmen a priori so entwerfen, daß sie mit dem Problem der Rundungsfehler umgehen können. Nur sehr wenige dieser *robusten* Algorithmen existieren, meist auch nur für sehr einfache Probleme. Die andere Möglichkeit ist das sogenannte *Exact Computation* Paradigma, welches dem Implementierer eines geometrischen Algorithmus die Illusion von exakter Arithmetik auf reellen Zahlen gibt, indem man exakte Zahlentypen und exakte geometrische Prädikate zur Verfügung stellt. Leider ist diese Illusion mit Kosten verbunden, die im Vergleich mit reiner Gleitkomma-Arithmetik bedeutend höher liegen.

Ein geometrisches Prdikat kann durch die Bestimmung des Vorzeichens eines arithmetischen Ausdruckes ausgewertet werden. Die naive Art und Weise, das Vorzeichen zu bestimmen, besteht darin, den Wert des Ausdrucks mittels exakter Arithmetik zu berechnen und das Vorzeichen vom Ergebnis abzulesen. Dieses Vorgehen dauert etwa 10 und 100 mal länger als dieselbe Rechnung in Gleitkomma-Arithmetik. Eine Möglichkeit, die Auswertung zu beschleunigen, besteht in der Benutzung sogenannter Gleitkomma-Filter. Die Idee ist folgende: man berechnet zuerst eine Gleitkomma-Approximation des Ausdruckes und eine Fehlerschranke für die maximale Abweichung vom echten Wert des Audrucks. Wenn die Fehlerschranke kleiner ist als der Wert der Approximation, ist das Vorzeichen des exakten Wertes bekannt. Nur für den Fall, daß der approximierte Wert unter der Fehlerschranke liegt, ist die Auswertung mittels exakter Arithmetik erforderlich. Von den Gleitkomma-Filtern wird behauptet, daß sie das exakte Rechnen ohne große Zusatzkosten ermöglichen. Die Laufzeiten würden sich verglichen zu einer reinen Gleitkomma-Implementierung nur etwa verdoppeln.

Diese Aussage gilt natürlich nur, wenn das Vorzeichen immer durch die Gleitkomma-Filterstufe bestimmt werden kann (d.h. keine Auswertung mit exakter Arithmetik nötig ist) und der Gleitkomma-Filtermechanismus bei allen arithmetischen Berechnungen zur Anwendung kommt. In der Praxis treffen diese Bedingungen allerdings oft nicht zu.

In diesem Teil der Arbeit entwickeln wir zunächst das Konzept der "Strukturellen Filterung", welches eine Verallgemeinerung des Gleitkomma-Filters ist. Diese neue Technik erlaubt es, den Geschwindigkeitsunterschied im Vergleich zu reiner Gleitkomma-Arithmetik noch weiter zu reduzieren, indem einige Prädikate falsch ausgewertet werden dürfen. Nichtsdestotrotz wird ein exaktes Endergebnis garantiert. Strukturelle Filterung betrachtet die Ausführung eines Algorithmus als eine Folge von Schritten und wendet Filterung auf der Ebene dieser Schritte an. Ein Schritt kann alles zwischen einer arithmetischen Operation und der Ausführung eines ganzen Algorithmus sein.

Als Teilresultat, welches nicht nur für unseren Kontext von Interesse ist, zeigen wir, wie sich Sortieralgorithmen bei fehlerhaften Vergleichsoperationen verhalten. Wir beweisen zum Beispiel, daß Quicksort ein optimaler Sortieralgorithmus bleibt, wenn Vergleichsoperationen fehlerhaft sein können, wohingegen Mergesort suboptimal wird. Wir haben dieses Resultat auf der 11th Canadian Conference on Computational Geometry 1999 ([FMN99]) vorgestellt.

Schließlich entwickeln wir das Design eines Geometriekernels, genannt LOOK (Lazy Object Oriented Kernel for geometric computation), welcher Gleitkomma-Filterungstechniken nicht nur auf der Ebene der Prädikate, sondern auch auf der Ebene der geometrischen Konstruktionen anwendet. Die Kernidee ist es, geometrische Objekte nicht durch ihre Koordinaten zu repräsentieren, sondern durch die Operationen, durch welche sie erzeugt wurden, und Koordinatenrepräsentationen nur bei Bedarf zu berechenen. Dieses Resultat haben wir auf dem 16th Annual ACM Symposium on Computational Geometry 2000 ([FM00]) vorgestellt.

Bibliography

[AB99] Nina Amenta and Marshall Bern. Surface reconstruction by Voronoi filtering. *Discrete Comput. Geom.*, 22(4):481–504, 1999.

[ABE98] Nina Amenta, Marshall Bern, and David Eppstein. The crust and the β-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing*, 60:125–135, 1998.

[Alt01] E. Althaus. *Curve Reconstruction and the Travelling Salesman Problem.* Phd thesis, Universitaet des Saarlandes, 2001.

[AM00] E. Althaus and K. Mehlhorn. Polynomial time TSP-based curve reconstruction. In *Proc. 11th ACM-SIAM Sympos. Discrete Algorithms*, pages 686–695, January 2000.

[Att97] D. Attali. r-regular shape reconstruction from unorganized points. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 248–253, 1997.

[BB97] Fausto Bernardini and Chandrajit L. Bajaj. Sampling and reconstructing manifolds using alpha–shapes. In *Proc. 9th Canad. Conf. Comput. Geom.*, pages 193–198, 1997.

[BBP98] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[BFMS99] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 341–350, 1999.

[BFS98] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 175–183, 1998.

[BMS96] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. The LEDA class “real” number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, Saarbrücken, 1996.

[BP01] E. Boyer and S. Petitjean. Regular and non-regular point sets: Properties and reconstruction. *Comput. Geom. Theory Appl.*, 19(2–3):101–126, 2001.

[BS99] J.-D. Boissonat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 370–379, 1999.

[Bur96] Christoph Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections.* PhD thesis, Universitaet des Saarlandes, 1996.

[CGA99] *The CGAL Reference Manual*, 1999. Release 2.0.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[dFdMG95] L. H. de Figueiredo and J. de Miranda Gomes. Computational morphology of curves. *Visual Comput.*, 11:105–112, 1995.

[DK99] T. K. Dey and P. Kumar. A simple provable algorithm for curve reconstruction. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 893–894, January 1999.

[DMR99] T. K. Dey, K. Mehlhorn, and E. A. Ramos. Curve reconstruction: Connecting dots with good reason. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 197–206, 1999.

[DP99] Olivier Devillers and Franco P. Preparata. Further results on arithmetic filters for geometric predicates. *Comput. Geom. Theory Appl.*, 13:141–148, 1999.

[DW00] T. K. Dey and R. Wenger. Reconstructing curves with sharp corners. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 233–241, 2000.

[Dwy87] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[Ede98] H. Edelsbrunner. Shape reconstruction with delaunay complex. In *Proc. 2nd Latin Amer. Sympos. Theoret. Informatics*, volume 1380 of *Lecture Notes Comput. Sci.*, pages 119–132. Springer-Verlag, 1998.

[EKS83] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans. Inform. Theory*, IT-29:551–559, 1983.

[FM91] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, June 1991.

[FM00] S. Funke and K. Mehlhorn. LOOK – a lazy object-oriented kernel for geometric computation. In *Proc. 16th Annu. ACM Symposium on Computational Geometry*, pages 00–00, 2000.

[FMN99] S. Funke, K. Mehlhorn, and S. Naeher. Structural filtering – a paradigm for efficient and exact geometric programs. In *Proc. 11th Canad. Conf. on Comput. Geom.*, 1999.

[For99] S. Fortune. Vertex-rounding a three-dimensional polyhedral subdivision. *Discrete Comput. Geom.*, 22(4):593–618, 1999.

[FR01] S. Funke and E. A. Ramos. Reconstructing a collection of curves with corners and endpoints. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 344–353, January 2001.

[Fun97] Stefan Funke. Exact arithmetic using cascaded computation. Master's thesis, Universität des Saarlandes, 1997.

[FV96] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[Gie99] J. Giesen. Curve reconstruction, the TSP, and Menger's theorem on length. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 207–216, 1999.

[Gol99] C. Gold. Crust and anti-crust: A one-step boundary and skeleton extraction algorithm. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 189–196, 1999.

[Gra96] Torbjörn Granlund. *GMP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, 1996. http://www.swox.com/gmp/.

[IEE85] *IEEE Standard for binary floating point arithmetic, ANSI/IEEE Std* $754-1985$. New York, NY, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[KLN91] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, January 1991.

[KLPY99] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee Yap. *The CORE Library Project*, 1.2 edition, 1999. http://www.cs.nyu.edu/exact/core/.

[KR85] D. G. Kirkpatrick and J. D. Radke. A framework for computational morphology. In G. T. Toussaint, editor, *Computational Geometry*, pages 217–248. North-Holland, Amsterdam, Netherlands, 1985.

[KW98] L. Kettner and E. Welzl. One sided error predicates in geometric computing. In Kurt Mehlhorn, editor, *Proc. 15th IFIP World Computer Congress, Fundamentals - Foundations of Computer Science*, pages 13–26, 1998.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[Mil88] V. Milenkovic. *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic.* Phd thesis, Carnegie Mellon University, 1988.

[MN00] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[Sch99] S. Schirra. A case study on the cost of geometric computing. In *Proc. 1st Workshop on Algorithm Engineering, LNCS 1619*, pages 156–176. Springer-Verlag, 1999.

[SOI90] K. Sugihara, Y. Ooishi, and T. Imai. Topology-oriented approach to robustness and its applications to several Voronoi-diagram algorithms. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 36–39, 1990.

[YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.