TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Finite-Precision Approximation Techniques for Planar Arrangements of Line Segments

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Eli Packer

The research work for this thesis has been carried out at Tel-Aviv University
under the supervision of Prof. Dan Halperin

October 2002

Acknowledgments

My deepest appreciation to my supervisor Professor Dan Halperin, for his help, support and encouragement during this thesis, and for always being available and attentive to any question or advice needed. A special thank is given to all the CGAL team people (past and present) in Tel Aviv University and for Mr. Ariel Tankus for their valuable help. Finally, I would like to thank my family for their support.

Contents

Chapter 1

Introduction

In this chapter we present the problem of robustness in computational geometry in general and review work that has been done in this field. We focus on the subject of our research, *Finite-Precision Approximation* for arrangements of line segments in $\mathbb{R}^2$. We conclude this chapter with an outline of the thesis.

1.1 Robustness in Computational Geometry Algorithms

There are two major obstacles in making the implementation of computational geometry algorithms and data structures robust: the use of floating-point arithmetic and degeneracies (or near-degeneracies) in input data or that occur during intermediate computations. The two are closely related since floating-point arithmetic problems are often caused by degenerate input data. When using floating-point arithmetic, a degenerate case is induced not only by degenerate data, but also by close-to-degenerate data. Since floating-point arithmetic is imprecise, we cannot tell for sure whether a certain case is degenerate or close to be one. Thus we use the term *potential degeneracy*. In what follows we sometimes drop the word potential before the word degeneracy. We refer by *robustness* to the general goal of making geometric algorithms robust to the above obstacles, namely making the implementations reliable and insensitive to them. For more details on robustness see, e.g., [23, 33, 35, 42].

Many algorithms in computational geometry are designed and proven in a computational model that assumes exact arithmetic, while built-in number types are finite and thus imprecise. While the use of finite-precision arithmetic is often efficient, it is not robust when dealing with degenerate cases or near-degenerate ones. In such cases the primitives may lead to erroneous results. This problem is especially complicated for geometric algorithms because they operate on a mixture of numerical and combinatorial data, whose consistency might be lost when using limited precision arithmetic.

Exact arithmetic number types have been developed in recent years to cope with robustness in computational geometry [3, 29, 40]. They provide exact and more robust implementations but suffer from two major disadvantages. The first one is that they are more costly in time and space. The second is that certain primitives (such as trigonometric functions) are very hard to implement and are often not implemented at all.

There are adaptive evaluation schemes that try to speed up running time and maintain exactness at the same time. One form is the *floating-point filter* which applies exact arithmetic only when determining answers with floating-point is not possible [10, 17, 28, 29, 37].

Degeneracies occur when the algorithm needs a special treatment (for example the collinearity of three points). While most algorithms assume general position, namely assume that degeneracies do not occur, the problem of dealing with degeneracies is left to the implementor who finds that this problem is very complicated, and requires considerable resources. An effort to deal with degeneracies directly for certain problems is sometimes a viable solution [4, 15].

Another kind of effort to deal with degeneracies is by *symbolic perturbation*. The idea is to remove degenerate cases by replacing each coordinate of every input object by a polynomial in a sufficient small ε symbolically, while maintaining consistency of the input data [11, 12, 36, 41].

Heuristic Epsilon is another approach to coping with robustness issues. The idea is to treat the values whose difference is smaller than a small parameter ε as equal. It is very simple to implement, but suffers from the fact that the relation of equality is not transitive [22, 40].

Finite-Precision Approximation The focus of the thesis. A *Finite-Precision Approximation* is a class of preprocessing procedures that perturb the input data to make it more robust for the algorithm. The goal of most of them is to remove degeneracies whose identification is not definite. Thus the algorithms can in fact assume general position. Often such preprocessing procedures are used to convert the input data into a low-precision representation (such as Snap Rounding — see Chapter 2). A justification for this scheme is that most likely the input data are obtained by measuring real world objects which might be imprecise. Thus, if the perturbation is significantly smaller than the possible measurement errors, it should not effect the correctness of the algorithm. Unlike other techniques to deal with robustness issues, such as arbitrary precision number types, finite-precision approximation algorithms are designed only for certain types of objects. There are finite-precision approximation algorithms that may also change the type of the data. For example in Snap Rounding (see Chapter 2), a segment might be transformed into a polygonal chain. It requires that algorithms that follow can cope with the new type of objects. In the sequel we survey related work done in specific areas of finite-precision approximation. In Chapter 2 we describe work done on Snap Rounding and in Chapter 7 we refer

to work on Controlled Perturbation. Additional techniques that fall in this category appear, for example, in [31, 32, 39].

1.2 Software Libraries for Robust Geometric Computing

In order to support the robust use of geometric algorithms, several computational geometry groups have implemented robust geometric libraries. We focus on two which are closely related to the work of this thesis.

CGAL - the Computational Geometry Algorithm Library. CGAL is a collaborative effort of several academic institutes in Europe and Israel to develop a C++ software library of robust geometric data structures and algorithms [6, 14, 13]. The major goals of the library are robustness, generality, flexibility, efficiency and ease-of-use. The goals are achieved by applying both object-oriented programming and the generic programing paradigm. The algorithms we describe in this work have been implemented with a substantial use of CGAL capabilities — see Chapter 5 for details. Our Iterated Snap Rounding package has been completely "CGALized" as a part of the effort of supplying robust implementation of geometric algorithms. For additional experimental research which extensively uses the CGAL library see [15, 23, 26, 33, 34].

LEDA - the Library of Efficient Data Structures and Algorithms. A library of efficient data structures and algorithms and a platform for combinatorial and geometric computing on which application programs can be built [29, 30]. It supplies modules such as graph algorithms, geometric objects and algorithms or graphical I/O which cover a considerable part of combinatorial and geometric computing. Our implementations mainly use LEDA's arbitrary precision number types, graphical window and graphical output to a postscript file. These capabilities are used extensively by other CGAL implementations too.

1.3 Thesis Outline

In this thesis we present two algorithms to perturb arrangements of line segments in $\mathbb{R}^2$ in order to make them more robust for further manipulation. Line segments in $\mathbb{R}^2$ are the basis of a huge number of algorithms in computational geometry as well as other fields that deal with geometric data such as computer graphics, computer-aided geometric design, and more. Our perturbation algorithms are categorized as finite-precision approximation algorithms. Both of them serve as a preprocessing step for geometric algorithms. We implemented both algorithms and present experimental results obtained with the implementation. The first one, *Iterated Snap Rounding*, is

a variant of the well known *Snap Rounding* algorithm and the second one, *Controlled Perturbation of Line Segments*, is an instance of the *Controlled Perturbation* scheme, which follows other instances that deal with different kinds of geometric objects. Below we outline both new algorithms.

Iterated Snap Rounding. We point out that in a snap-rounded arrangement (see Chapter 2) the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding. We propose and analyze an augmented procedure, Iterated Snap Rounding, which rounds the arrangement such that each vertex is at least half-the-width-of-a-pixel away from any non-incident edge. Iterated snap rounding preserves the topology of the original arrangement in the same sense that the original scheme does. However, the guaranteed quality of the approximation degrades. Thus each scheme may be suitable in different situations. We describe an implementation of both schemes. In our implementation we substitute an intricate data structure for segment/pixel intersection that is used to obtain good worst-case resource bounds for Iterated Snap Rounding by a simple and effective data structure which is a cluster of kd-trees. A paper describing Iterated Snap Rounding was accepted for publication [24].

Controlled Perturbation of Line Segments. We present a perturbation scheme to overcome degeneracies and precision problems in computing an arrangement of line segments in $\mathbb{R}^2$. The idea behind this scheme is that the output set of line segments (*set*, for brevity) is built incrementally by inserting the segments to the set, each one in its turn, after possibly perturbing them in order to remove degeneracies that they induce. Thus the arrangement of the set that we build is degeneracy-free. The algorithm follows a scheme named Controlled Perturbation — see Chapter 7 for details.

The rest of the thesis is organized as follows. We divide it into two main parts: Iterated Snap Rounding and Controlled Perturbation of Line Segments. In the next chapter we describe the Snap Rounding scheme and the work that has been done in this area. In Chapter 3 we present our novel scheme which we call Iterated Snap Rounding. In Chapter 4 we present c-Oriented kd-Trees which constitute an efficient search structure we use in the implementation of Iterated Snap Rounding, and describe other implementation details in Chapter 5. In Chapter 6 we experimentally compare Snap Rounding and Iterated Snap Rounding. In Chapter 7 we describe the Controlled Perturbation scheme and the work that has been done in this area. In Chapter 8 we present Controlled Perturbation of line segments in $\mathbb{R}^2$ and describe implementation details in Chapter 9. In Chapter 10 we present experimental results for Controlled Perturbation of line segments. Concluding remarks and possible directions for future work are given in Chapter 12. In Apendix A we supply further technical details concerning the analysis of the Controlled Perturbation algorithm.

Part I

Iterated Snap Rounding

Chapter 2

Snap Rounding

Snap Rounding is a method that belongs to the family of finite-precision approximation of geometric structures. It converts an *arrangement* of line segments into a low-precision representation.

Given a finite collection $\mathcal{S}$ of segments in the plane, the arrangement of $\mathcal{S}$, denoted $\mathcal{A}(\mathcal{S})$, is the subdivision of the plane into vertices, edges, and faces induced by $\mathcal{S}$. A *vertex* of the arrangement is either a segment endpoint or the intersection of two or more segments. Given an arrangement of segments whose vertices are represented with arbitrary-precision coordinates, Snap Rounding (SR, for short) proceeds as follows [19, 27]. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement is replaced by the center of the hot pixel containing it and each edge e is replaced by the polygonal chain through the centers of the hot pixels met by e, in the same order as they are met by e. See Figure 2 for an illustration.

In the process, vertices and edges of the original arrangement may have collapsed. However, the rounded arrangement preserves certain topological properties of the original arrangement: The rounding can be viewed as a continuous process of deforming curves (the original segments into chains) such that no vertex of the arrangement ever crosses through a curve [21] (see Figure 2.2 for an illustration). The rounded version s' of an original segment s approximates s such that s' lies within the Minkowski sum of s and a pixel centered at the origin.

Related Work. Greene and Yao [20] were the first to propose a rounding scheme for polygonal subdivisions. They show that a simple rounding to the closest grid points violates topological properties and therefore a more sophisticated approach should be taken. They developed a method for perturbing lines slightly at grid points. They do that by introducing the notion of *hooks*. A hook is a vector from a point to its nearest grid point. The idea behind their method is that intersections between segments, as well as intersections between segments and hooks, are rounded. This rounding scheme

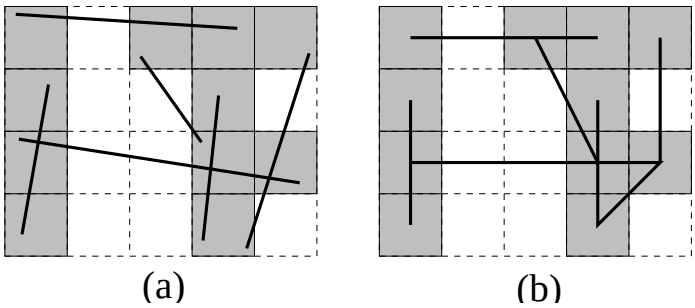


Figure 2.1: An arrangement of segments before (a) and after (b) snap rounding (hot pixels are shaded)

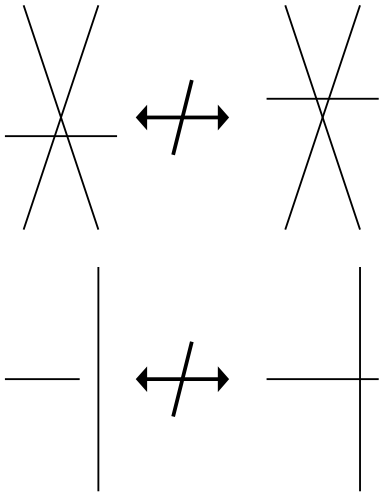


Figure 2.2: Two examples of topology violation that are ruled out in SR

is accomplished without violating many of the topological properties. This method provides a link between the continuous and the discrete domain. The problematic aspect is that the number of *links* of a polyline (namely the number of segments the polyline is composed of), which is the output for a segment, can be large. It can be much greater than the same number for SR since the idea of SR to break segments where they intersect hot pixels eliminates the extraneous intersections. The time complexity of the method is $O((n+k)\log n + h_i)$ where n is the number of input segments, k is the number of intersections among the input segments and h_i is the number of induced hooks.

Hobby [27] and Greene [19] proposed the SR paradigm. Hobby's algorithm [27] for SR is based on the Bentley-Ottmann sweep line algorithm for finding intersections of line segments (although other algorithms can be applied as well). During the sweep they round the arrangement by utilizing many properties of the hot pixels. The time complexity of the algorithm is $O((n+k)\log n + \sum_{h\in H}|h|)$ where n is the number of input segments, k is the number of intersections among the input segments, H is the set of hot pixels and $|h|$ is the number of segments intersecting a hot pixel h.

Guibas and Marimont [21] give a dynamic algorithm for snap rounding an arrangement of segments in the sense that segments can be inserted or removed dynamically from the SR representation. They do it by using ideas from Mulmuley's dynamic incremental construction algorithm of a point location structure based on trapezoidal decomposition of the arrangement, while maintaining (and producing) only the SR representation of the arrangement of the current segments. They also give elementary proofs of the topological properties maintained by SR.

Goodrich et al. [18] present an output sensitive algorithm for SR without first determining all the intersection pairs of segments in the input. The main idea is to improve the running time of the algorithm when many segments intersect in a hot pixel. Let b be the number of segments intersecting inside a pixel. The former methods had a overhead time of $\Omega(b^2)$ while here the overhead time is $O(b\log b)$. Thus the time complexity depends on the number of segments and the complexity inside hot pixels and it is $O(n\log n + \sum_{h\in H}|h|\log n)$ where the parameters are defined as above. They present two algorithms: the first one is deterministic with the above time complexity and the second one is randomized with the same expected running time. The first one is based on a plane sweep strategy with special treatment to hot pixels in order to find all the segments that intersect it. The second one is based on dynamically maintaining a trapezoidal decomposition of both segments and boundaries of hot pixels. They also extend SR to a set of line segments in $\mathbb{R}^3$ and give an output-sensitive algorithm to compute rounded arrangements. The idea is to round segments to voxels grid. Unlike snap rounding in $\mathbb{R}^2$, segments that almost intersect might induce a hot voxel and thus are rounded to its center. It is done by defining a connector to be the smallest segment connecting two given segments. Hot voxels are defined as the ones that contain segments' endpoints or connectors which are smaller than one unit. Then all segments are rounded to hot voxels.

Fortune [16] extends SR to three dimensions. The input to his algorithm is a

polyhedral subdivision P in $\mathbb{R}^3$ with a total of n facets. He shows that there is an embedding of the vertices, edges, and facets of P into a subdivision Q, where every vertex coordinate of Q is an integral multiple of $2^{-\lceil \log_2 n+2 \rceil}$. The embedding preserves or collapses vertical order on faces of P. The subdivision Q has $O(n^4)$ vertices in the worst case, and can be computed in the same time.

Chapter 3

Iterated Snap Rounding

3.1 The Distance Between a Vertex and a Non-Incident Edge

We first claim that degeneracies may be induced in the output of SR. The main motivation of the ISR algorithm that we present next, is to eliminate those degeneracies.

Consider the two segments s, t displayed in Figure 3.1 before and after SR. We denote the right endpoint of s' by s'_r. (Recall that u' is the rounded version of u.) After rounding, t' penetrates the hot pixel containing s'_r, but it does not pass through its center.

We can modify the input segment t so that t' becomes very close to s'_r: we move the left endpoint of t arbitrarily close to the top right corner of the pixel containing it. We vertically translate the right endpoint of t far downwards (outside its original pixel) —the farther down we translate it, the closer t' will be to s'_r.

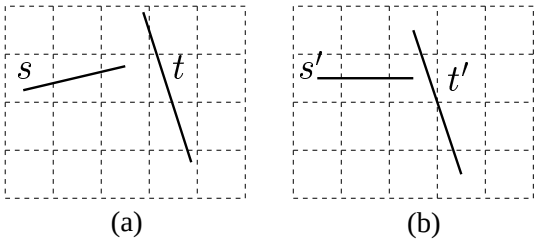


Figure 3.1: A vertex becomes very close to a non-incident edge after (b) snap rounding

We cannot make t' *arbitrarily* close to s'_r. If they are not incident then there is a lower bound on the distance between them. This distance, however, can be rather small. Let b denote the number of bits in the representation of the vertex coordinates of the output chains of SR. We tile a bounding square of the arrangement with $2^b \times 2^b$ unit pixels. In this setting the distance between t' and s'_r can be made as small as

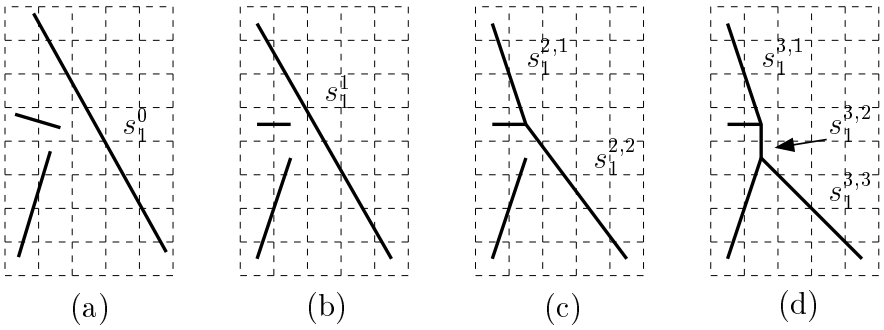


Figure 3.2: Iterated snap rounding for the input (a) results in (d)

$1/\sqrt{(2^b-1)^2+1} \approx 2^{-b}$.

One could argue that although SR produces near-degenerate output, it is still possible, during the rounding process, to determine the correct topology of the rounded arrangement in the hot pixel containing s'_r. However, this requires that the output of SR should include additional information beyond the simple listing of polygonal chains specified by their rounded vertices, making it more cumbersome to use and further manipulate.

3.2 Algorithm

We augment SR to eliminate the near-degeneracies mentioned above. Our procedure, which we call *iterated snap rounding* (ISR, for short), produces a rounded arrangement where an original segment is substituted by a polygonal chain each vertex of which is at least 1/2 a unit distant from any non-incident edge.

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be the collection of input segments whose arrangement we wish to round. Recall that a pixel is hot if and only if it contains a vertex of the input arrangement. Let H denote the set of hot pixels induced by $\mathcal{A}(S)$.

Our goal is to create chains out of the input segments such that a chain that passes through a hot pixel is re-routed to pass through the pixel's center. The difficulty is that once we reroute a chain it may have entered other hot pixels and we need to further reroute it, and so on.

Our rounding algorithm consists of two stages. In a preprocessing stage we compute the hot pixels (by finding all the vertices of the arrangement) and prepare a segment intersection search structure D on the hot pixels to answer queries of the following type: Given a segment s, report the hot pixels that s intersects. In the second stage we operate a recursive procedure, REROUTE, on each input segment. We postpone the algorithmic details of the preprocessing stage to the next sections and concentrate here on the rerouting stage.

REROUTE is a "depth-first" procedure. As we show below, the rerouting that we

propose does not add more hot pixels, so whenever we refer to the set of hot pixels we mean H. The input to REROUTE is a segment $s \in \mathcal{S}$. The output is a polygonal chain s^* which approximates s. Whenever s^* passes through a hot pixel, it passes through its center. See Figure 3.2 for an illustration.

We next describe the ISR algorithm. The routine REROUTE will produce an output chain s_i^* in the global parameter OUTPUT_CHAIN as an ordered list of links. If a segment is contained inside a single pixel, the chain degenerates to a single point.

ISR
Input: a set $\mathcal{S}$ of n segments
Output: a set $\mathcal{S}^*$ of n polygonal chains; initially $\mathcal{S}^* = \emptyset$
/* stage 1: preprocessing */
1. compute the set H of hot pixels
2. construct a segment intersection search structure D on H
/* stage 2: rerouting */
3. **for** each input segment $s \in \mathcal{S}$
4. initialize OUTPUT_CHAIN to be empty
5. REROUTE(s)
6. add OUTPUT_CHAIN to $\mathcal{S}^*$
7. **end for**

REROUTE(s)
/* s is the input segment with endpoints p and q */
1. query D to find H_s, the set of hot pixels intersected by s
2. **if** H_s contains a single hot pixel /* s is entirely inside a pixel */
3. **then** add the center of the hot pixel containing s to OUTPUT_CHAIN
4. **else**
5. let $m_1, m_2, \ldots, m_r$ be the centers of the r hot pixels in H_s in the order of the intersection along s
6. **if** ($r = 2$ **and** p, q are centers of pixels)
7. **then** add the link $\overline{m_1 m_2}$ to OUTPUT_CHAIN
8. **else**
9. **for** $i = 1$ to $r - 1$
10. REROUTE($\overline{m_i m_{i+1}}$)

We next discuss the properties of the procedure.

We fix an orientation for each input segment and its induced chains: it is oriented in lexicographically increasing order of its vertices. Thus, a non-vertical segment for example is oriented from its left endpoint to its right endpoint. (The orientation of a chain is well defined since, as is easily verified, a chain is (weakly) x-monotone and (weakly) y-monotone.) We represent the operation of REROUTE on a segment s_i as a tree T_i. The root contains s_i. The leaves of the tree contain the output polygonal chain s_i^*, one link in each node, ordered from left to right where the first

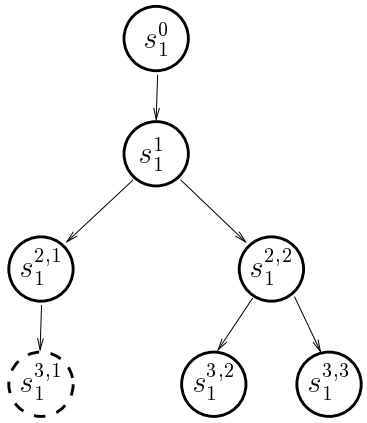


Figure 3.3: The tree T_1 corresponding to REROUTE(s_1) for s_1 of Figure 3.2. Nodes denoted by full-line circles contain segments with which we query the structure D. The dashed-line circle denotes a node containing an exact copy of the segment of its parent.

link is in the leftmost leaf. Each internal node ν together with its children represent one application of REROUTE (without recurrence): the segment s of ν, which passes through the hot pixels with centers $m_1, m_2, \ldots, m_r$, is transformed into the links $\overline{m_q m_{q+1}}, q = 1, \ldots, r-1$ which are placed in the children of ν ordered from left to right to preserve the orientation of the chain. We denote all the segments in the nodes at the jth level from left to right by $s_i^{j,1}, s_i^{j,2}, \ldots, s_i^{j,l_{i,j}}$, where $l_{i,j}$ denotes the number of nodes at this level. We denote the chain consisting of all the links at level j ordered from left to right by s_i^j. Thus $s_i^0 = s_i$. We denote by k_i the depth of the tree for s_i, and let $k := \max_{i=1}^n k_i$. For notational convenience, if a leaf λ is at level $k_\lambda < k$ then we add a linear path of $k_i - k$ artificial nodes descending from λ and all containing the same link that λ contains (we denote it differently at any level according to the level). See Figure 3.3 for an illustration of the tree T_1 corresponding to segment s_1 of Figure 3.2. We denote by $s(\nu)$ the segment (or link) that is contained in the node ν.

The next lemma gives an alternative view of ISR.

Lemma 3.1 *Given a set of segments $\mathcal{S}$, the output of ISR is equivalent to the final output of a finite series of applications of SR starting with $\mathcal{S}$, where the output of one SR is the input to the next SR.*

Proof: Once we determine the hot pixels H, snap rounding an input segment s (i.e., by the standard SR) can be done independently of the other segments. That is, the information necessary for rounding is in H. Notice that the chains $s_i^1, i = 1, \ldots, n$ are the result of applying SR to the original input segments $\mathcal{S}$.

The crucial observation is that SR does not create new hot pixels. It can break a segment into two segments that meet at the center of an existing hot pixel, but it cannot create a new endpoint nor a new intersection point (with another segment)

which are not at the center of an existing hot pixel—this would violate the topology preservation properties of SR [21].

It follows that with the same set H of hot pixels, the chains $s_i^{j+1}, i = 1, \ldots, n$ are the result of applying SR to the links in the chains $s_i^j, i = 1, \ldots, n$, and so on.

The process terminates when the link in each leaf of the tree has its endpoints in the center of hot pixels and it does not cross any other hot pixel besides the hot pixels that contain its endpoints.

The tree continues to grow beyond level j only as long as for at least one node ν in level j when we query with $s(\nu)$ we discover a new hot pixel through which $s(\nu)$ passes. We claim that a hot pixel is not discovered more than once per tree. This is so since, as already mentioned, each chain s_i^j is (weakly) x-monotone and (weakly) y-monotone. Since there are at most $O(n^2)$ hot pixels, the process will stop after a finite number of steps. □

The lemma's algorithmic interpretation is inefficient, but it is useful for proving some of the following properties.

Corollary 3.2 *ISR preserves the topology of the arrangement of the input segments in the same sense that SR does.*

Proof: The topological properties that are preserved by SR can be summarized by viewing SR as a continuous process of deforming curves (the original segments into chains) such that no vertex of the arrangement ever crosses through a curve [21]. Since SR does not create new vertices, the assertion follows from Lemma 3.1. □

Lemma 3.3 *(i) If an output chain of ISR passes through a hot pixel then it passes through its center.*
(ii) In the output chains each vertex is at least 1/2 a unit away from any non-incident segment.

Proof: Claim (i) follows from the definition of the procedure REROUTE. Since all the vertices of the rounded arrangement are centers of hot pixels, claim (ii) is an immediate consequence of (i). □

A drawback of ISR is that an output chain s_i^* can be farther away from the original segment s_i compared with the chain produced for the same input segment by SR. Recall that k_i denotes the depth of the recursion of REROUTE(s_i).

Lemma 3.4 *A final chain s_i^* lies in the Minkowski sum of s_i and a square of side size k_i centered at the origin.*

Proof: In SR, a rounded segment s' lies inside the Minkowski sum of the input segment s and a unit square centered at the origin. Since ISR is equivalent to k_i applications of SR, the claim follows. □

This deviation may be acceptable in situations where the pixel size is sufficiently small or when $k := \max_{i=1}^{n} k_i$ is small.

3.3 Algorithmic Details and Complexity Analysis

Let I denote the number of intersection points of segments in the original arrangement $\mathcal{A}(\mathcal{S})$. We first compute the set H of hot pixels. For that we use an algorithm for segment intersection. This could be done with a plane sweep algorithm, or more efficiently in $O(I + n \log n)$ time by more involved algorithms [2, 7]. To compute the hot pixels, the algorithm should also be given a pixel's width w and a point p that will be assigned the coordinate $(0, 0)$. The plane will be tiled with pixels that we will consider to be of unit width, and their centers will have integer coordinates. We denote the number of hot pixels by N. Notice that N is at most $O(n + I)$.

Remark. One could alternatively detect the hot pixels by the SR algorithm of Goodrich et al. [18] and thus get rid of the dependence of the running time of the algorithm on the number of intersections I. Notice however that for this step alone (namely for detecting the hot pixels) and for certain inputs (e.g., the input depicted in Figure 3.4 and described below) this alternative is costly.

Next we prepare the data structure D on the hot pixels H to answer segment intersection queries. We construct a multi-level partition tree [1] on the vertical boundary segments of the hot pixels, and an analogous tree for the horizontal boundary segments. The partition trees report the segments intersected by a query segment s from which we deduce the hot pixels intersected by s. Each tree requires $O(M^{1+\varepsilon})$ preprocessing time when allowed M units of storage for $N \leq M \leq N^2$. A query takes $O(N^{1+\varepsilon}/\sqrt{M} + g)$ time, where g is the number of hot pixels found [1].

How many times do we query the structure D for segment intersection?

Lemma 3.5 *If an output chain s_i^* consists of l_i links then during* REROUTE*(s_i) the structure D is queried at most $2l_i$ times.*

Proof: During REROUTE(s_i) when we query with a link (line 1 of REROUTE) either we do not find new hot pixels (new for the rounded version of s_i) in which case we charge the query to the link which is then a link of the final chain, or we charge it to the *first* new hot pixel (recall that we assigned an orientation to each segment and to each link). Each final link is charged exactly once and each vertex of the final chain is charged at most once, besides the last vertex which is never charged. The bound

follows. □

Let L denote the overall number of links in all the chains output by ISR. We summarize the performance bounds of ISR in the following theorem.

Theorem 3.6 *Given an arrangement of n segments with I intersection points, the iterated snap rounding algorithm requires $O(n \log n + I + L^{2/3}N^{2/3+\varepsilon} + L)$ time for any $\varepsilon > 0$ and $O(n + N + L^{2/3}N^{2/3+\varepsilon})$ working storage , where N is the number of hot pixels (which is at most $2n + I$) and L is the overall number of links in the chains produced by the algorithm.*

Proof: To find the intersections of the input segments we use Balaban's algorithm which requires $O(n \log n + I)$ time and $O(n)$ working storage. When an intersection is found we simply keep its corresponding hot pixel. For constructing and querying the multi-level partition trees (by Lemma 3.5 we perform at most $2L$ queries overall) we use a standard trick that balances between the preprocessing time and the overall query time, and does not require that we know the number of queries in advance. See, e.g., [8]. □

Next we discuss combinatorial bounds on the maximum complexity of the rounded arrangements. Interestingly, as shown next, there is no difference between the maximum asymptotic complexity of the rounded arrangements between SR and ISR.

Theorem 3.7[1] *Given an arrangement of n segments in the plane, in its rounded version: (i) the maximum number of hot pixels through which a single output chain passes is $\Theta(n^2)$, and (ii) the maximum overall number of incidences between output chains and hot pixels is $\Theta(n^3)$. (iii) The maximum number of segments in the rounded arrangement (namely without counting multiplicities) is $\Theta(n^2)$, and if the input segments induce N hot pixels then this number is $\Theta(N)$. All these bounds apply both to SR and to ISR.*

Proof: The upper bounds in claims (i) and (ii) are obvious. To see that these bounds are tight consider the following construction (see Figure 3.4). We take $n/2$ long horizontal segments spanning a row of $n^2/4$ pixels. Next we take $n/2$ short, slightly slanted segments, each spanning $n/2$ pixels such that overall each pixel in the row is intersected by exactly one short segment. The short segments are slanted such that in each pixel that they cross they intersect exactly one of the long segments. Each pixel in the row is now a hot pixel, and each of the long segments crosses all the hot pixels. The rounding obtained with both SR and ISR is the same.

[1] The slanted version of our horizontal construction was suggested to us by Olivier Devillers. Claim (iii) is due to Mark de Berg.

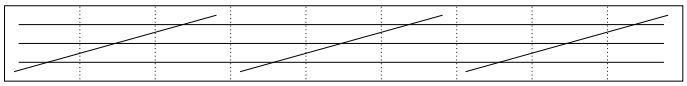


Figure 3.4: $\Theta(n)$ chains in the rounded arrangement are each incident to $\Theta(n^2)$ hot pixels

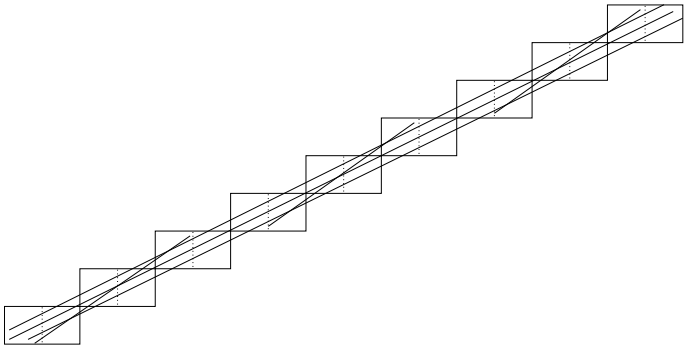


Figure 3.5: The slanted version yields $\Theta(n)$ rounded segments with $\Theta(n^2)$ links each

The construction yields a degenerate rounded arrangement. Each of the output chains is in fact a horizontal line segment. This construction can be slanted so that each rounded version of a long segment is a chain with "true" $\Omega(n^2)$ links. In the slanted version we use $n^2/2$ pixels arranged in $n^2/4$ rows. In each row at least one pixel is hot. See Figure 3.5 for an illustration.

Finally, we ignore the chains, and we ask how complex can the rounded arrangement be, that is, we ignore multiplicities (overlap) of chains. Obviously, the rounded arrangement can have $\Omega(n^2)$ complexity. But this is also an upper bound since the (rounded) arrangement has N vertices and it is a planar graph. Therefore the number of edges can be at most $O(N)$. N can be at most $O(n^2)$. Again, our arguments do not depend on how the rounding was done (by SR or ISR). □

We conclude this section with a worst-case tight bound on the distance between an original segment and its output chain produced by ISR.

Theorem 3.8 *The maximum distance between an input segment and its output chain is* $\Theta(n^2)$.

Proof: Recall that n is the number of segments in the input. Let d be the distance between a certain input segment and its output chain. Since there are at most $O(n^2)$ hot pixels and each one may add at most $\sqrt{2}/2$ units to d, the upper bound follows. To see that this bound is tight, consider Figure 3.6(a)[2]. There are $n-1$ segments arranged vertically similar to the construction in Figure 3.4, inducing $\Theta(n^2)$ hot pixels.

[2]A similar idea that is improved by this example was suggested by Shai Hirsh.

We refer to this construction as $\mathcal{W}$. The segments of $\mathcal{W}$ together with the segment we add below compose the input. The center of the lowest hot pixel in $\mathcal{W}$ is at $(0,0)$ while the highest one is at $(0,a)$ where $a = \Theta(n^2)$ and even. We add a segment with integer coordinates $s = ((0,a),(b,0))$ (meaning that the endpoints of s are $(0,a)$ and $(b,0)$) where $b = \frac{a}{2}$. There are no other segments in the input besides the ones involved in $\mathcal{W}$ and s. Thus there are no hot pixels lying to the right of $\mathcal{W}$ beside the one centered at $(b,0)$. Let s^* be the output chain for s. Notice that for each c, $b \leq c \leq a$, the segment $((0,c),(b,0))$ intersects the hot pixel whose center is at $(0,c-1)$. Therefore during the process of ISR, s^* will slide down $\mathcal{W}$, each time penetrating more hot pixels from below. The work on s^* stops when reaching the hot pixel whose center is at $(0,b-1)$. Thus s^* will be composed of the chain $(0,a),(0,a-1)\ldots(0,b),(0,b-1),(b,0)$ (see Figure 3.6 for an illustration). It is easy to verify that the distance from $(0,b-1)$ (which is a vertex of s^*) to s is $\Omega(n^2)$. The claim follows. □

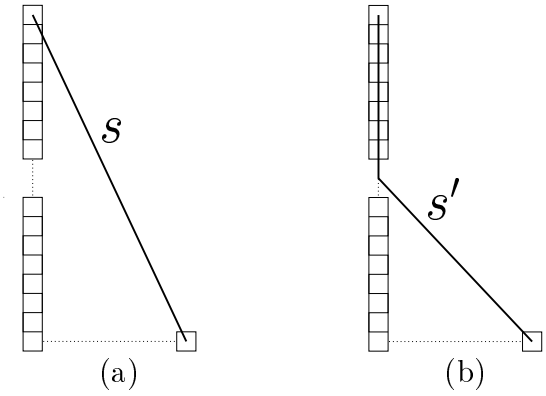


Figure 3.6: An input example for which the maximum distance between the input (a) and the output chain of ISR (b) is $\Omega(n^2)$

Chapter 4

c-Oriented Kd-Trees

In our implementation we use a plane sweep algorithm to find the intersections between segments in $\mathcal{S}$ and thus we identify the hot pixels. The non-trivial part to implement is the search structure D with which we answer segment/pixel intersection queries. In the theoretical analysis we use partition trees for D, as these lead to asymptotically good worst-case complexity. In practice, (multi-level) partition trees are difficult to implement. Instead, we implemented a data structure consisting of several kd-trees. Next we explain the details.

Observation 4.1 *A segment s intersects a pixel p of width w, if and only if the Minkowski sum of s with a pixel of width w centered at the origin contains the center of p.*

We could use Observation 4.1 in order to answer segment intersection queries in the following way: build a range search structure on the centers of the hot pixels. Let s be the query segment and $M(s)$ be its Minkowski sum with a pixel centered at the origin. Then query the structure with the range $M(s)$. Unfortunately, the known data structures for this type of queries are similar to the multi-level data structures that we have used in Chapter 3.

Instead we use kd-trees as an approximation of this scheme. A kd-tree answers range queries for axis-parallel rectangles [9]. Its guaranteed worst-case query time is far from optimal but it is practically efficient. A trivial solution would be to query with the axis-parallel bounding box of $M(s)$, which we denote by $B(s)$; see Figure 4.1. This may not be sufficiently satisfactory since the area of $B(s)$, which we denote by $|B(s)|$, may be much larger than the area of $M(s)$.

If we rotate the plane together with $M(s)$ the (area of the) axis-parallel bounding box changes whereas $M(s)$ remains fixed. The difference between the bounding boxes for two different rotations can be huge. Our goal is to produce a number of rotated copies of the set of centers of hot pixels so that for each query segment s there will be one rotation for which the area of the bounding box is not too much different from

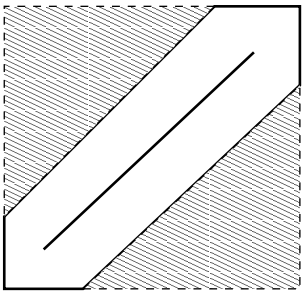


Figure 4.1: The bounding box of the Minkowski sum of a segment with a pixel centered at the origin. The shaded area is the redundant range.

the area of $M(s)$. Notice that if a segment s is rotated by $\pi/2$ radians, the size of the relevant bounding box remains the same. Since the determination of which rotation to choose is dependent only on the size of the respective bounding box, the range of rotations should be the half-open interval $[0 : \pi/2)$.

We construct a collection of kd-trees each serving as a range search structure for a rotated copy of the centers of hot pixels. We call this cluster *c-oriented kd-trees*. Let c be a positive integer and let $\alpha_i := (i-1)\frac{\pi}{2c}$ for $1 \leq i \leq c$. The structure consists of c kd-trees such that the i-th kd-tree, denoted by kd_i, has the input points rotated by α_i. Let $R_i(s)$ be the segment s rotated by α_i. For each query with segment s we do the following: for each $kd_i, 1 \leq i \leq c$, we compute $|B(R_i(s))|$. Let $1 \leq h \leq c$ be the serial number of the kd-tree for which $|B(R_h(s))| = \min_{i=1}^{c} |B(R_i(s))|$. Then we use the h-th kd-tree to answer the query with the segment s rotated by α_h. Finally, we discard all the points for which the segment does not intersect the respective hot pixels.

We next discuss a few important issues regarding the implementation and usage of this structure.

Exact rotations. We used exact arithmetic to implement ISR. Unfortunately, the available exact arithmetic number types do not support the calculations of *sines* and *cosines* which are necessary for calculating rotations. Instead we use only angles for which the sines and cosines can be expressed as rational numbers with small enumerator and denominator [5]. We keep an array $\mathcal{Z}$ of approximations to the *sines* of integer *degree* angles between $0 - 89$. We emphasize that once we fix an angle β we have the exact *sine* and *cosine* of β. What we cannot do is obtain the exact values of the trigonometric functions of a prescribed arbitrary angle. Since our choice of rotation angles is heuristic to begin with, the precise angle is immaterial, and the angle we use is never more than one degree off the prescribed angle. Moreover, there are techniques to achieve better approximations [5], but we prefer not to use them because of performance reasons.

How big should c be? There are advantages and drawbacks in using few kd-tress, say even one kd-tree compared to using many. When using one kd-tree, we are prone to get many false points in the range queries, resulting in more time to filter out the results. When using many kd-trees, we need to invest time in their construction and a little more time per query to find the best rotation. Our experiments show that in many cases a small number of trees suffices. Consider for example the numerical table "different number of kd-trees" in Figure 6.1. (The rounding example in this figure as well as the other examples are explained in detail in Chapter 6; here we only refer to the number of kd-trees used in their computation.) The first column shows how many kd-trees were used and the last column shows how much time the overall rerouting stage took compared with the time when using only one kd-tree (the full legend is given in Table 10.1). The best performance is obtained when we use 7 kd-trees. The time savings in this case is 17% over using a single kd-tree. The analogous table in the next example (Figure 6.2) shows that in that example there is no benefit in using more than one kd-tree.[1] In the next paragraph we present a heuristic improvement of the number of kd-trees. However, we leave the computation of the best number of kd-trees together with the best rotation angles of each one for further research.

Skipping kd_i's. Since c should be small, we expect most of the links of a certain input segment to have the same rotation as the input segment, since they should all have nearby slopes. Let J_i be the number of input segments that are rotated by α_i. If J_i is very small, it is not effective to create the respective kd-tree. Thus we fix a lower limit τ, and construct a kd-tree kd_i only if $J_i \geq \tau$. Obviously τ should be a function of c, and be sufficiently small to ensure that at least one kd-tree will be constructed. We chose to use $\tau = \frac{n}{2c}$. In the examples of Figures 6.1 and 6.2 J_i is always greater than $\frac{n}{2c}$. In other examples, such as geographic data, not all c trees are always constructed—in Figure 6.3, when the algorithm is given $c > 7$ it chooses to skip some of the kd_i's. In this example, using more than one kd-tree is wasteful since the map is relatively sparse, most of the segments are relatively small compared to the whole map and the bounding box of their Minkowski sum with a unit pixel does not intersect many hot pixels centers.

[1] The running time indicated in the tables is in seconds while using arbitrary precision rational arithmetic. The pixel size in the first example is 1 and in the second example is 15.

Chapter 5

Implementation Details

We implemented ISR in C++, using many capabilities of CGAL [6]. The package defines a C++ *class* to work on [38]. The programmer uses our package by creating instances of the class. The implementation is generic in the sense that each class is templated with a number type of which the data are composed. The user of our software chooses which number type to apply with the *template* mechanism of the C++ language.

The main input of the package is a set of line segments while the output is a set of polygonal chains.

The user can choose the output format. It can be either a text file describing the output or a graphic window in which both the input and the output are drawn (the graphic window is the LEDA window [29]).

Except for CGAL capabilities that we explicitly mention, we applied other CGAL elements such as geometric predicates, points, segment, vector and intersection operations.

The package supports both ISR and SR. It is up to the user to decide which one to apply. The way to convert the ISR algorithm to SR is simply to constrain the recursion depth of the Reroute routine to one (see Chapter 3), meaning that the output polygonal chains are determined immediately by the hot pixels that the original segments intersect.

ISR and SR are conveniently implemented with an exact number type, otherwise the topology of the input line segments may be violated. We implemented ISR with the Leda rational number type [29]. It is possible that under certain assumptions, SR and ISR may be implemented with finite-precision arithmetic.

Recall that we use the c-oriented kd-trees as our search structure (see Chapter 4). As a first step for creating the c-oriented kd-trees, we have to find the hot pixels. This is done by applying a plane sweep algorithm [6]. For that we use the plane sweep package of CGAL. Recall that the c-oriented kd-trees are composed of several kd-trees. We use the kd-tree package of CGAL to implement that. The user can

choose the number of trees to use.

The ISR package has become a part of CGAL.

Chapter 6

Rounding Examples: SR vs. ISR

To give the flavor of how the output of ISR differs from that of SR we present the rounding results for three input examples; see Figures 6.1, 6.2, and 6.3. For each example we display the input, the SR result and the ISR result. Then we zoom in on a specific area of interest in these three drawings—an area where the rounding schemes differ noticeably. A square near a drawing represents the actual pixel size used for rounding. Then we provide two tables of statistics. The first one refers to the best number of kd-trees as related to the discussion in the previous section. The second table summarizes the differences in the rounding for different pixel sizes. The abbreviations we use in these two tables are explained in Table 6.1. The *deviation* of a chain from its inducing segment s is the maximal distance of a point on the chain from s.

6.1 Congestion Data

The data contains 200 segments with $18,674$ intersections; see Figure 6.1. (For clarity, the pictures in Figure 6.1 depict a similar example with only 100 segments.) The bottom left part of the arrangement is zoomed in.

Both rounding schemes will collapse thin triangles that have two corners close by. However, not allowing proximity between vertices and non-incident edges, ISR collapses 'skinny' faces of the arrangement that SR does not (see the bottom of the zoomed-in area), for example triangles that have one corner close to the middle of the opposite edge.

For pixel size 1, SR and ISR are very different and the number of vertices that are less than half a unit away from a non-incident edge in the SR output is in the hundreds. The average deviation in ISR in this example is never more than 2.5 times that of the corresponding SR output. For pixel size greater than 1 the average deviation of a chain in ISR is almost the same as in SR. However, for pixel size smaller than 1, the average deviation is larger in the ISR output than in the SR output.

Abbreviation	Explanation
inkd	input number of kd-trees
nkd	actual number of kd-trees created
nfhp	overall number of false hot pixels in all the queries
tt	total time relative to using one tree
md	maximum deviation over all chains
ad	average deviation
mnv	maximum number of vertices in an output chain
anv	average number of vertices in an output chain
mdvs	minimum distance between a vertex and a non-incident edge
ncvs	number of pairs of a vertex and a non-incident edge that are less than half the width of a pixel apart
ps	pixel size
nhp	number of hot pixels

Table 6.1: Abbreviations

In terms of combinatorial complexity the results are similar and the average number of vertices per chain is roughly the same in both outputs. This is a phenomenon we have observed in all our experiments.

6.2 Triangulation Data

Figure 6.2 shows a set of input points (courtesy of Jack Snoeyink) and a triangulation of this set. The triangulation consists of 906 segments. The zoomed in pictures show a part of the triangulation for which there is considerable difference between SR and ISR.

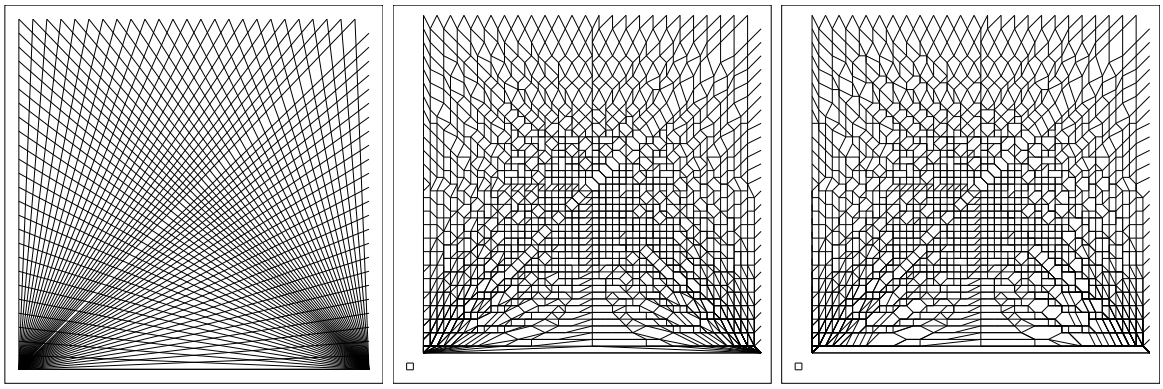
Again ISR collapses thin polygons that SR does not collapse. The second table in Figure 6.2 shows that in this case the average deviation of a chain in both schemes does not differ by much. The maximum deviation in ISR is always less than twice the pixel width. Here also the average number of links per chain is almost the same for the output of SR and ISR.

6.3 Geographic Data

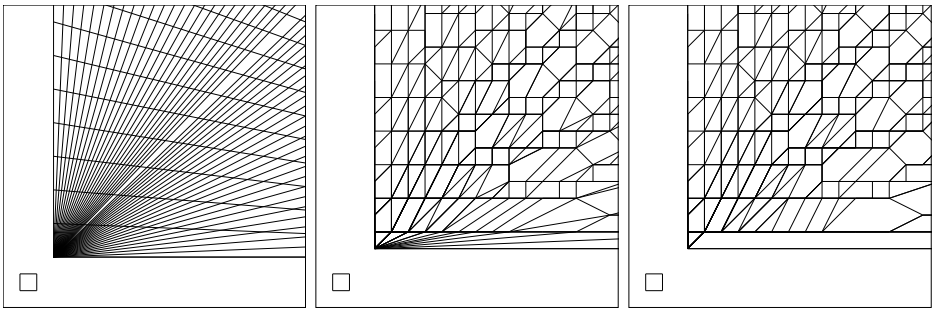
We ran both schemes on several geographic maps of countries and cities which are less cluttered than the examples above. The experiments for this type of data typically show little difference between the SR and ISR results. Figure 6.3 depicts the result for a map of the USA. The data contains 486 segments intersecting only at endpoints.

The second table in Figure 6.3 shows the difference of using SR and ISR. In most

of the tests, there are occasional cases in which the distance between a vertex and a non-incident segment is shorter than half the size of a pixel. Thus there are differences between the SR and the ISR output. These differences are however minor. In the ISR output the maximum deviation is no more than twice that of the SR output. The average deviation in both the SR and ISR output is similar.



Input SR output ISR output



Input zoom in SR output zoom in ISR output zoom in

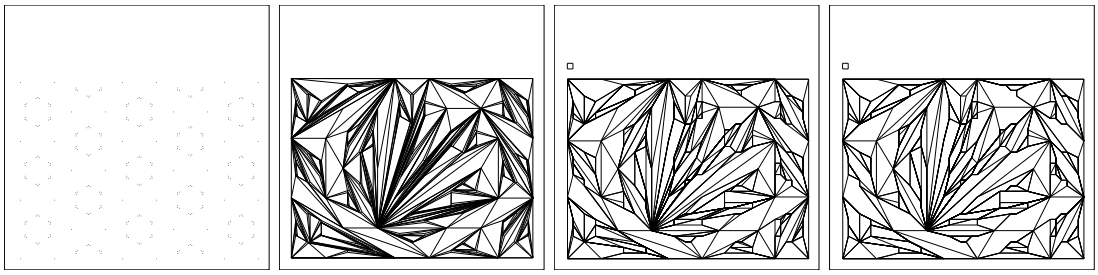
inkd	nkd	nfhp	tt
1	1	613477	100% = 213.2 s
2	2	513551	87.2%
3	3	474997	83.6%
4	4	478749	84%
5	5	479507	84.3%
6	6	463025	83.4%
7	7	456882	83%
8	8	456269	84%
9	9	455334	84.8%
10	10	456196	86.3%

Different number of kd-trees

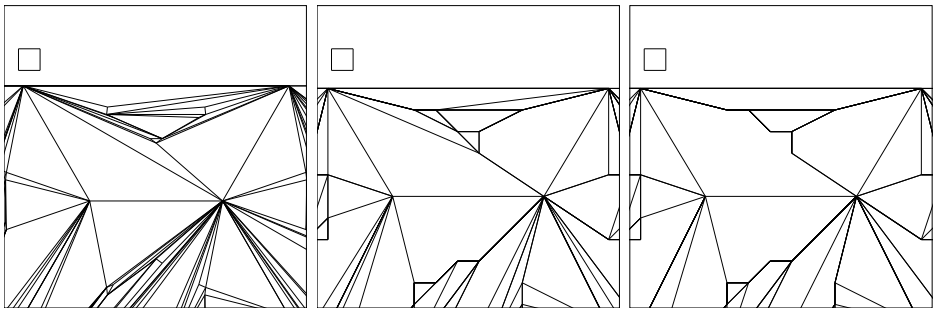
		isr						sr					
ps	nhp	md	ad	mnv	anv	mdvs	ncvs	md	ad	mnv	anv	mdvs	ncvs
0.125	8488	1.01	0.19	120	90.96	0.08	0	0.09	0.09	106	87.95	0.04	17
0.25	8261	1.5	0.41	124	94.15	0.15	0	0.17	0.17	112	89.16	0.06	58
0.5	7711	1.68	0.67	135	97.9	0.28	0	0.35	0.35	126	91.66	0.08	135
1	6003	1.58	0.99	154	101.85	0.55	0	0.71	0.71	153	95.99	0.07	328
2	2538	1.51	1.41	101	72.9	1.26	0	1.41	1.41	101	72.87	0.88	3
3	1143	2.12	1.84	67	49.1	2.12	0	2.12	1.84	67	49.09	1.34	1
4	673	2.82	2.7	51	37.56	2.82	0	2.82	2.7	51	37.56	2.82	0
5	439	3.53	3.32	41	30.31	3.53	0	3.53	3.32	41	30.3	2.23	1
10	120	7.07	6.6	21	15.58	7.07	0	7.07	6.6	21	15.58	7.07	0

ISR and SR comparison (n = 200)

Figure 6.1: Congestion data



Input points | Input triangulation | SR output | ISR output



Input zoom in | SR output zoom in | ISR output zoom in

inkd	nkd	nfhp	tt
1	1	4872	100% = 15.4 s
2	2	4789	103.2%
3	3	4852	102.6%
4	4	4597	101.9%
5	5	4487	102.6%
6	6	4349	103.2%
7	7	4349	102.6%
8	8	4399	102.6%
9	9	4419	103.2%
10	10	4358	102.6%

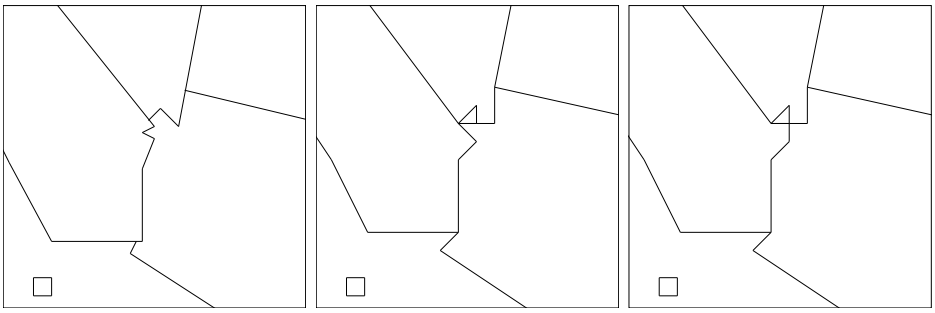
Different number of kd-trees

		isr						sr					
ps	nhp	md	ad	mnv	anv	mdvs	ncvs	md	ad	mnv	anv	mdvs	ncvs
2	306	2.231	0.825	6	2.219	1.223	0	1.341	0.812	6	2.198	0.318	9
5	300	9.804	2.691	7	2.625	3.14	0	3.494	2.442	6	2.48	0.741	50
10	249	17.194	5.18	9	2.761	5.368	0	7.028	4.847	7	2.637	1.414	45
15	195	22.088	6.985	10	2.75	9.486	0	10.559	6.512	10	2.622	2.631	67
20	162	32.207	7.614	9	2.621	11.767	0	13.914	7.19	8	2.532	4.85	45

ISR and SR comparison

Figure 6.2: Triangulation data

Input SR output ISR output



Input zoom in SR output zoom in ISR output zoom in

inkd	nkd	nfhp	tt
1	1	293	100% = 9.11 s
2	2	306	102%
3	3	302	103.1%
4	4	284	103.8%
5	5	293	105%
6	6	275	106%
7	7	260	106.8%
8	6	269	106.1%
9	8	272	107.9%
10	8	253	107.9%

Different number of kd-trees

		isr						sr					
ps	nhp	md	ad	mnv	anv	mdvs	ncvs	md	ad	mnv	anv	mdvs	ncvs
0.125	486	0.097	0.088	4	2.098	0.111	0	0.088	0.088	4	2.096	0.045	1
0.25	485	0.353	0.177	5	2.113	0.196	0	0.176	0.176	5	2.107	0.039	2
0.5	480	0.392	0.353	4	2.104	0.377	0	0.353	0.353	4	2.100	0.039	2
1	475	1.414	0.715	5	2.137	0.569	0	0.707	0.707	5	2.115	0.196	3
2	432	2.236	1.063	5	2.137	1.264	0	1.414	1.043	5	2.102	0.392	9
3	379	3.807	1.353	5	2.037	2.121	0	2.121	1.336	5	2.020	1.341	2
4	338	3.333	1.764	6	1.991	2.828	0	2.828	1.758	5	1.983	1.264	2
5	299	4.735	2.124	5	1.897	3.535	0	3.535	2.110	4	1.884	1.581	3
10	177	10.606	3.732	5	1.615	7.071	0	7.071	3.696	5	1.602	7.071	0

ISR and SR comparison

Figure 6.3: Geographic data

Part II

Controlled Perturbation of Line Segments

Chapter 7

Controlled Perturbation

Controlled Perturbation is another kind of finite-precision approximation technique. More precisely, it is a framework whose details need to be worked out for different kinds of objects. The name of this scheme is *Controlled* Perturbation since it is controlled in two aspects. First, by determining the size of the perturbation, we control the running time of the perturbation scheme and set a tradeoff between the magnitude of the perturbation and the efficiency of the computation. Second, after each object is processed, the perturbation algorithm guarantees that it induces no degeneracies. Generally, the scheme proceeds as follows.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of input objects. Each object $s \in S$ is inserted in its turn. Possibly, each $s \in S$ is further divided into $m \geq 1$ parts, $Q(s) = q_1, q_2 \cdots q_m$, ordered in a specific order determined by the algorithm. Then each part $q \in Q(s)$ is inserted in its turn. In each insertion we check if q induces degeneracies with the already inserted objects. (In what follows "an object induces degeneracies" always refers to "with the already inserted objects".) If not, it is inserted. Otherwise we have to perturb q. We next describe how to do so in our algorithm (other algorithms of this scheme apply similar ideas). We define the set $Q(s)$ such that each $q \in Q(s)$ has a unique vertex, v_q, which is perturbed if necessary in order to remove degeneracies. We perturb v_q in the following way. We define a disc C, centered at v_q with a radius r. r is computed such that when picking up a point p randomly inside C, we are guaranteed that with a reasonable probability, φ (for example $\frac{1}{2}$), if we place v_q at p then the object q attached to v_q will not induce any degeneracies. We pick up a placement for v_q randomly inside C and check if any degeneracy is induced. If not, q is inserted. Otherwise we continue choosing placements for v_q in the same way until we find a degeneracy-free placement. Since the probability to induce no degeneracies is φ, we find a degeneracy-free perturbation after $\frac{1}{\varphi}$ trials on the average. The selection of the parameter φ is a tradeoff between the size of the perturbation and the efficiency of the computation. The larger φ is, the bigger the perturbation magnitude is, but the probability of finding degenerate-free placement is greater, thus less trials on the average should suffice. We set $\varphi = \frac{1}{2}$ in our implementation.

Related Work. Halperin and Shelton [25] were the first to propose a Controlled Perturbation algorithm. Their goal was the elimination of degeneracies induced by collections of spheres in $\mathbb{R}^3$. They described a software package for computing and manipulating the subdivision of a sphere and for computing the boundary surface of the union of spheres. Their implementation was a component in a package aimed to support geometric queries on molecular models. The excuse for the perturbation is that the model is approximate to begin with. The time complexity of their method is linear in the number of spheres in the input.

It was followed by Raab in [33, 34] who proposed a Controlled Perturbation algorithm to eliminate degeneracies induced by polyhedral surfaces in $\mathbb{R}^3$. The motivation was to create a robust model for swept volume applications. A swept volume is defined as the geometric space occupied by an object moving along a trajectory in a given time interval. Her swept volume application computes the boundary of a collection of three-dimensional polyhedra and employs vertical decomposition as its final step.

Chapter 8

Controlled Perturbation of Line Segments

8.1 Introduction

In this chapter, we propose a Controlled Perturbation algorithm for arrangement of line segments in $\mathbb{R}^2$ (CPLS for short). CPLS follows the framework described in Chapter 7. While this framework has been applied for collection of spheres [25] and polyhedral surfaces [34], we propose a novel scheme of the framework, for arrangements of line segments in $\mathbb{R}^2$.

The idea of CPLS is to perturb the above arrangement into a robust representation for further manipulation. This is done by eliminating degeneracies induced in the arrangement. Degeneracies are eliminated by slightly perturbing some of the line segments inducing them, creating degeneracy-free data.

8.2 Preliminaries and Key Ideas

We use the following notation throughout the chapter. $S = \{s_1, s_2, \ldots, s_n\}$ is the set of input line segments ordered arbitrarily. S_i denotes the set of the first i segments of S. For each $s_i \in S$ we denote its endpoints by p_i and q_i (we relate to p_i as the first endpoint, and to q_i as the second one; this choice is arbitrary). As we show later, the perturbation of each $s \in S$ has two phases. In the first one p_i is possibly perturbed. We denote by p_i' the result of the first phase on p_i, either perturbed or not. We denote by s_i' the segment $\overline{p_i'q_i}$. In the second phase, q_i is possibly perturbed. We denote by q_i' the result of the second phase on q_i, either perturbed or not. We denote by s_i'' the segment $\overline{p_i'q_i'}$. s_i'' is the CPLS output for s_i. Let $S_i'' = \{s_1'', s_2'' \ldots s_i''\}$ and $A = S_n''$. Then A is the output of CPLS, namely the set of the output segments produced by CPLS.

The goal of CPLS is to eliminate degeneracies induced in arrangements of line segments so that the algorithms that manipulate the input further will be robust. In order to define a degeneracy formally, we use a *resolution parameter*, $\varepsilon_0 > 0$, which is another input of the algorithm. Two features are degenerate if they are not ε_0-away from each other (i.e, the distance between them is less than ε_0). We need to use other two artificial resolution parameters which cannot be smaller than ε_0. They are denoted by ρ_1 and ρ_2. The idea is that when perturbing a segment s_i, ρ_1 is the resolution parameter of p'_i (more precisely, we demand that a disc centered at p'_i with a radius ρ_1 is empty) and ρ_2 is the resolution parameter of q'_i and s''_i (more precisely, we demand that a disc centered at q'_i with a radius ρ_2 is empty and that the distance between s''_i and any vertex induced by S''_{i-1} is at least ρ_2). We need two different resolution parameters for the endpoints since as we show below, the work on p'_i is different from the work on q'_i. They also differ from ε_0 since ε_0 is used for a certain degeneracy which must have smaller perturbation magnitude than ρ_1 and ρ_2 — see Appendix A.2.4. We discuss the exact relations among the resolution parameters in Section 8.7 and Appendix A.2.4.

In order to eliminate degeneracies we use the following perturbation process. We order the line segments arbitrarily and possibly perturb each one in its turn. For each line segment s_i, we possibly perturb p_i and q_i several times. If p_i induces degeneracies, it is perturbed in order to find a placement in which it does not induce any degeneracy. The perturbation of q_i is different since its goal is not only to eliminate the degeneracies induced by q_i, but also to eliminate the ones induced by the whole s'_i. If q_i or s'_i induce degeneracies, we perturb q_i until all degeneracies are eliminated. Once the work on an endpoint is done, its placement is determined and it is never perturbed again. Each endpoint is perturbed inside a disc whose center is the original endpoint and whose radius is called a *perturbation radius*. Since the goals of the perturbations of p_i and q_i are different, different perturbation radii are used for each one. We denote the perturbation radii by δ_1 and δ_2 for p_i and q_i respectively. The radii are determined such that the probability that a placement of an endpoint induces a degeneracy is no more than $\frac{1}{2}$. We choose a placement for the endpoint at random (inside the disc) until no degeneracy is induced. Since the probability to induce degeneracies after the perturbation is no more than $\frac{1}{2}$, after no more than 2 perturbations on the average we find a degenerate-free placement. The determination of the values of δ_1, δ_2, ρ_1 and ρ_2 are technical (and tedious) and hence postponed to Appendix A.

Previous Controlled Perturbation algorithms [25, 34] applied optimization techniques in order to make the work and the outcome of the algorithm more efficient. We implemented these optimizations and describe how we apply them in our algorithm — see Section 8.5.

A critical decision when designing a geometric algorithm is whether to use finite-precision arithmetic or exact arithmetic. We describe the advantages and disadvantages of using each one and explain why we choose to implement CPLS with finite-precision arithmetic. Other Controlled Perturbation algorithms [25, 34] used

finite-precision arithmetic as well.

Throughout the algorithm we use the following atomic operations (we assume that each operation takes $O(1)$ time):

- Finding an intersection between two line segments.
- Finding the distance between a segment and a point.
- Picking up a random point inside a disc.

We expect the perturbation radii and the resolution parameters to be much smaller than the length of the input line segments. Otherwise CPLS is not *acceptable* for the input since line segments may be perturbed significantly. In that case, the user should refrain from using CPLS and resort to other fixed-precision approximation schemes. We next give a formal definition of this issue.

Definition 8.1 *CPLS is considered λ-acceptable for an input set S of segments and for a parameter λ if and only if $\delta/L \leq \lambda$ where δ is the largest perturbation radius and L is the length of the longest input line segment in S.*

Note that the biggest perturbation radius is bigger than any resolution parameter (see Equation A.6 and Theorem A.4). Thus if the perturbation is λ-acceptable, then for each resolution parameter ε, $\varepsilon/L \leq \lambda$ holds. We get that CPLS is λ-acceptable if any resolution parameter and perturbation radius is at least $\frac{1}{\lambda}$ times smaller than the longest input segment. Thus the perturbation magnitudes will be relatively small, provided that λ is small enough. We arbitrarily choose $\lambda = \frac{1}{10}$ in our implementation. Our experiments have shown that with a reasonable input resolution parameter and input that is not extremely congested, CPLS is found λ-acceptable. The above definition is crucial for both the CPLS algorithm and its analysis, as we show below.

Discussion: CPLS vs. SR and ISR. SR (see Chapter 2), as well as ISR (see Chapter 3), have basically the same goal as CPLS, but the results of SR and ISR compared with CPLS are quite different. Both make the vertices of the original arrangement well separated. In SR and ISR all vertices inside a certain pixel are collapsed to the center of it, possibly introducing new degeneracies. The situation is different in CPLS. Here vertices of the original arrangement are perturbed to make their distance not less than a given threshold. In that sense, the results are somewhat opposite to SR and ISR. An advantage of SR and ISR over CPLS is that they preserve certain topological features while CPLS does not. On the other hand, an advantage of CPLS over SR and ISR is that the output type is maintained (line segments) while SR and ISR transform segments into polygonal chains. While SR has the property that an output chain is very close to its original segment this is not the case for ISR and CPLS

where the distance between an original segment and its output depends on the input segments and the parameters of the algorithm. In SR the distance between a vertex and a non-incident edge can be extremely small inducing potential degeneracies. This is not the case for ISR and CPLS. While ISR and SR can maintain planar subdivisions, CPLS is constrained to work with segments. The above discussion implies that CPLS provides another scheme to create a robust approximation of an arrangement of line segments in $\mathbb{R}^2$ beyond the well known SR and our ISR. Each scheme may be suitable in different situations.

8.3 The Degeneracies

Recall that a vertex of an arrangement of line segments is either an endpoint, e, or an intersection point i of two segments. Let s be a segment. Three types of degeneracies are possible in an arrangement of line segments:

D_1 : *endpoint - line segment* It takes place when the distance between e and s (where e is not an endpoint of s) is smaller than a given threshold.

D_2 : *intersection - line segment* It takes place when the distance between i and s (where s is not one of the segments that induce i) is smaller than a given threshold.

D_3 : *two endpoints* It takes place when s is short enough such that its endpoints induce degeneracies.

8.4 Algorithm

As we described earlier, each line segment is processed in two phases, one for each endpoint. Next we explain the details of each phase. For brevity we omit the special case of s_1 which involves only perturbations due to degeneracies of type D_3 in which only q_1 might be perturbed such that it is sufficiently far from $p'_1 = p_1$.

First phase. In this phase p_i is possibly perturbed. The possible degeneracies in this case are of type D_1. We first check whether no $s''_j \in S''_{i-1}$ induces a degeneracy with p_i. If so, we set $p'_i = p_i$. Otherwise we have to perturb p_i. The perturbation is done as follows. We perturb p_i randomly in a disc centered at p_i with a radius δ_1, giving p'_i. We check if p'_i induces degeneracies of type D_1. If not, the work on p'_i is done. Otherwise, we continue choosing placements at random inside the same disc centered at p_i until we find a placement for which p'_i induces no degeneracies. Recall that δ_1, the radius of the disc, is determined such that the probability that a placement of an endpoint induces degeneracies is no more than $\frac{1}{2}$. In Appendix A we show that this holds for any $\delta_1 \geq \frac{8mR\rho_2}{\pi}$, where m is the maximum number of line segments that were inserted into A by the time that a certain $s \in S$ is inserted, which can be very close to s or intersect it (we describe how to estimate m in Section 8.6),

R is the ratio $\frac{\rho_1}{\rho_2}$ (the size of ρ_1 is determined in this way, namely we set ρ_1 to be $R\rho_2$; we give the details in Section 8.7), and ρ_2 is the resolution parameter for the second phase (see Theorem A.4).

Second phase. In this phase, s_i'' is inserted into A. Since the location of p_i' is already determined, only q_i may be perturbed such that none of the following types of degeneracies arise:

D_3 between p_i' and q_i'.

For each $s_j'' \in S_{i-1}''$:
D_1 induced by either p_j' or q_j' and s_i''.
D_1 induced by q_i' and s_j''.

For each $s_j'', s_k'' \in S_{i-1}'', j \neq k$ (the order of j and k is not important):
if s_j'' intersects s_k'', D_2 induced by this intersection and s_i''.

For each $s_j'', s_k'' \in S_{i-1}'', j \neq k$ (the order of j and k is important):
if s_j'' intersects s_i'', D_2 induced by this intersection and s_k''.

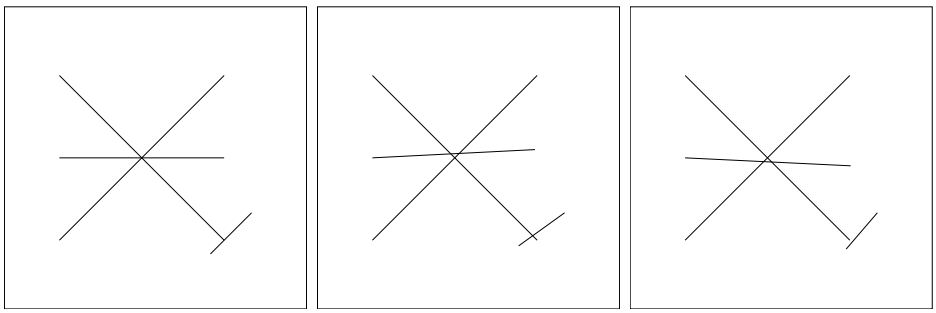
We check if s_i' induces degeneracies. If not, $q_i' = q_i$ and s_i' is inserted into A. Otherwise, we perturb q_i inside a disc centered at q_i with a radius δ_2 in the same manner as done for p_i above. Recall that δ_2 is determined such that the probability that a placement of an endpoint induces a degeneracy is no more than $\frac{1}{2}$. In Appendix A we show that this holds for any $\delta_2 \geq \frac{4\rho_2}{\pi}\left(\frac{m(m+3)(L+\frac{8mR\rho_2}{\pi})}{\rho_2\sqrt{R^2-1}} + 2(m+1)\right)$, where m, R and ρ_2 are defined as in the first phase and L is the length of the largest line segment in S.

Figure 8.1 depicts two results of Controlled Perturbation of an arrangement of four line segments. Notice that both degeneracies of types D_1 and D_2 are eliminated, but the topology of the original arrangement may not be preserved.

The complexity of the procedure is analyzed in Sections 8.5, 8.6, 8.7 and in Appendix A, and is summarized in Theorem 8.8.

8.5 Optimizations

We describe two typical optimization techniques to improve the quality of the output and the performance of the algorithm. These techniques were previously applied in the context of Controlled Perturbation [25, 34]. The first one deals with a useful technique to find a group of candidate segments for the degeneracies tests. By that we improve the performance of CPLS since we prevent many possibly redundant tests



(a) (b) (c)

Figure 8.1: An arrangement of line segments (a) and two different results of CPLS (b),(c)

of degeneracies. The second one deals with reducing the perturbation magnitudes. By achieving smaller perturbations we improve the quality of the output since the output line segments are closer to the original ones. Our CPLS implementation uses these techniques and we base our analysis on them.

8.5.1 Tiling the Plane

Let $s_i \in S$ be a segment that is currently perturbed. We need to find the segments that induce degeneracies with s_i. We could check all the segments in S_{i-1} for that but in practice we can do better. Let $U(s_i) \subseteq S_{i-1}$ be the set of segments that may induce degeneracies with s_i, after possibly perturbing the segments. Then it suffices to work only with segments of $U(s_i)$. We next describe a way to find a superset of $U(s_i)$, which may still be much smaller than S_{i-1}.

Let τ be the smallest magnitude that satisfies the following condition: if the distance between two line segments is equal or greater than τ, then there is no possibility that a degeneracy which is a result of both is induced. In Section 8.6 we show that $\tau = 2\max(\delta_1, \delta_2) + \rho_1$. We use the following definition throughout this section.

Definition 8.2 *Let o_1 and o_2 be two geometric objects and $d(o_1, o_2)$ be the minimal distance between them. We say that two objects are μ-close if $d(o_1, o_2) < \mu$.*

Let $V(s_i) = \{s \in S | s_i$ and s are τ-close$\}$. The following clearly holds:

Observation 8.3 $U(s_i) \subseteq V(s_i)$.

Recall that L is the maximum length of a segment in S. We tile the plane with a grid of squares, H, whose edge length is L such that the point $(0, 0)$ is a vertex of the tiling. We keep the squares that are used (as described below) in a balanced

binary tree. Let $H_{s_i} \subseteq H$ be the squares that are intersected by s_i or are τ-close to s_i. Let $S(H_{s_i}) \subseteq S_{i-1}$ be the set of segments that intersect H_{s_i} or are τ-close to H_{s_i}. Obviously each segment $s \in V(s_i)$ intersects at least one square $h \in H_{s_i}$ or is τ-close to it but possibly other segments that are not τ-close to s may also intersect H_{s_i}. We get the following.

Observation 8.4 $V(s_i) \subseteq S(H_{s_i})$.

From observations 8.3 and 8.4 we get that $U(s_i) \subseteq S(H_{s_i})$. Thus the tests for degeneracies with s_i can be restricted to the segments of $S(H_{s_i})$. Now the problem is restricted to finding $S(H_{s_i})$. We describe next how we do that. Then the work for finding $S(H_{s_i})$ for all the segments is done as a preprocessing step when all the parameters necessary for computing the resolution parameters are determined (more precisely after the work described in Sections 8.6 and 8.7). During the perturbation process, each segment s_i points to $S(H_{s_i})$ and we use the segments of $S(H_{s_i})$, after being possibly perturbed, when testing for degeneracies with S_i.

Notice that if CPLS is λ-acceptable then $\tau \leq 3\lambda L$. Then, if CPLS is λ-acceptable, for each λ we can bound the number of squares which s_i intersects or that are τ-close to s_i by a constant (recall that the edge length of the square is L). Recall that we choose $\lambda = \frac{1}{10}$. With that value the bound of the number of such grid squares is 7. It follows that if we denote by w the maximum possible number of segments in each square, then $|S(H_{s_i})| = O(w)$. The algorithm proceed as follows. We first find the square $h_1 \in H$ that contains p_i. We insert h_1 into H_{s_i}. Then we check each one of the 8 neighboring squares of h_1 to see if s_i intersects it or if s_i is τ-close to it. Let $H_1 \subseteq H$ be the group of squares that satisfy this condition. Each $h \in H_1$ is inserted into H_{s_i}. Then we apply the same procedure we applied to h_1 to each $h \in H_1$ and possibly find another set $H_2 \subseteq H$ of squares. Then each $h \in H_2$ is inserted to H_{s_i}. We do not need to search for neighbors of the squares in H_2 since by that time H_{s_i} is complete. The reason is that other squares are too far to intersect or be τ-close to s_i. Obviously we take care not to insert a square twice.

For each $h \in H$ let $S(h)$ be the list of segments intersecting h or that are τ-close to h ($S(h)$ is built incrementally in the process we describe below). For each $h \in H$ being inserted to H_{s_i}, we insert each $s \in S(h)$ to $S(H_{s_i})$ and insert s_i to $S(h)$. We need to avoid inserting the same segment twice, thus the data structure that holds $S(H_{s_i})$ is implemented as a balanced binary tree. By the time we finish the above algorithm, we are guaranteed that $S(H_{s_i})$ is complete.

The work we described above is for finding the segments that can induce degeneracies with s_i, thus appropriate when inserting q_i. When inserting p_i the work is a little different since we need to find the segments that may induce degeneracies with p_i. The only difference is that instead of finding H_{s_i}, we have to find $H_{p_i} \subseteq H$ which are the squares that are ζ-close to p_i where $\zeta = \max(\delta_1, \delta_2) + \delta_1 + \rho_1$ (the difference from τ is that we regard the perturbation of a vertex p_i and not of an entire segment). The following claim is similar to the one we described above for s_i: If CPLS is λ-

acceptable, then $|S(H_{p_i})| = O(w)$. Thus the asymptotic complexity of the procedure does not change.

Another important point is to estimate the number of degeneracies that are tested when processing a certain line segment. This estimation effects the complexity of the work. For example, when testing degeneracies with intersections of line segments, we expect that the number of degeneracies to be tested would be quadratic in the number of line segments that might induce degeneracies. We denote this quantity by $\Psi(w)$. We postpone the various estimations of $\Psi(w)$ to Appendix A.

The following theorem summarizes the discussion above.

Theorem 8.5 *Finding the set of segments that are tested with the currently inserted segment for degeneracies takes $O(n(\log n + w \log w))$ preprocessing time and $O(wn)$ working storage for all the inserted segments together, where w is the maximum possible number of segments in each square.*

Proof: Recall that n is the number of segments in the input. For a segment s_i, finding and inserting squares to the squares data structure takes $O(\log n)$ time. Since $S(h)$ and $S(H_{s_i})$ are implemented as a list and a balanced binary tree respectively, the work on them takes $O(w \log w)$ time. Together the total time for n segments is $O(n(\log n + w \log w))$. There are at most $O(n)$ squares, each holds at most $O(w)$ segments. For each segment inserted we build the tree $S(H_{s_i})$ with $O(w)$ nodes. Thus the working storage is $O(nw)$.

□

8.5.2 Reducing the Perturbation Magnitude

If a vertex must be perturbed then we would like to move it as little as possible. Since our algorithm does not find the smallest perturbation that removes the degeneracies, which is a very complicated and time consuming task, we take another strategy. The discussion below is relevant both to the first and the second phases. As mentioned in Section 8.4, we check if there is a need to perform perturbations to eliminate degeneracies. If there is, we perturb the vertex randomly inside a disc whose center is the original placement of the vertex in order to find a degeneracy-free placement. The size of the disc (namely its radius δ) is sufficiently large so that no more than two perturbation trials on the average are needed. Since δ is an upper bound which fits extreme cases of congested areas such that the various forbidden placements do not overlap, there is a great chance that a smaller radius is sufficient. Thus the optimization is carried out as follows. We begin with a smaller radius, r (we choose r to be 10 times bigger than the largest resolution parameter, ρ_1 — see Lemma A.2 and the details following it), making a few random trials in order to find a degeneracy-free placement (let c be a parameter we fix for the number of such trials). If we find one, we are done. Otherwise we double r, making at most c additional random trials inside

the larger disc. We continue this way until we find a degeneracy-free placement. If we reach a point in which $r \geq \delta$, we set r to be δ, and continue with the same r until we find a degeneracy-free placement. With δ as the radius, we need at most two trials on the average to get a degeneracy-free placement. The reason for stopping at this radius is that we do not want the perturbation to be larger than δ in order to constrain the perturbation magnitude and to satisfy the perturbation analysis. We next give an upper bound on the expected running time of this procedure.

Theorem 8.6 *Finding a degeneracy-free placement for any vertex takes* $O(c\Psi(w)\log\frac{\delta}{\rho_1})$ *expected time.*

Proof: Let κ be the largest integer that satisfies $10 * 2^{\kappa}\rho_1 < \delta$. Then the series of the radius sizes is $\{10\rho_1, 20\rho_1, \ldots, 10 * 2^{\kappa}\rho_1, \delta\}$, where for each one c trials are carried out except for the last one, in which a small constant number of trials are carried out. Thus the number of trials is $O(c\log\frac{\delta}{\rho_1})$. Since each trial involves $O(\Psi(w))$ tests (recall that $\Psi(w)$ is a function that determines the number of potential degeneracies that might be involved as a function of w), the total expected time is $O(c\Psi(w)\log\frac{\delta}{\rho_1})$. □

8.6 Computing τ and m

This section describes a preprocessing step of the perturbation algorithm. We defined τ as the smallest magnitude such that if the distance between two line segments is equal or larger than τ then there is no possibility that a degeneracy which is a result of both is induced. We start by evaluating τ.

Two segments cannot induce a degeneracy if after they are perturbed, their distance is equal to or greater than some resolution parameter. The largest resolution parameter we encounter is ρ_1 (see Appendix A). We conclude that:

$$\tau = 2\max(\delta_1, \delta_2) + \rho_1.$$

We are now ready to give a formal definition of the parameter m which was mentioned above. We define m to be the maximum number of line segments inserted by the time a certain segment $s \in S$ is inserted, which may induce degeneracies with s, namely the segments that are τ-close to s. Thus m strongly depends on τ which in turn depends on δ_1,δ_2 and ρ_1. On the other hand, according to Appendix A, δ_1, δ_2 and ρ_1 depend on m. Therefore we need to find a way to determine the values of these parameters.

Let $\Delta = \{\delta_1, \delta_2\}$. Since the smaller the value of m is, the smaller the perturbation magnitudes are, we want to find the smallest m such that together with the values of

Δ there are indeed no more than m segments which are τ-close to a certain one. We propose three approaches:

I We begin with $m = 1$, compute the values of the parameters in Δ and check if there are at most $m = 1$ segments τ-close to any segment $s \in S$ with the appropriate values of Δ. If this is the case, we set m to 1 and the values of the parameters in Δ accordingly. Otherwise, we increment m by one and do the above again. We continue with this scheme until there are at most m segments τ-close to a certain segment $s \in S$ with the appropriate values of Δ. Obviously we are guaranteed to stop when we reach $m = n$. We use the tiling technique as described in Section 8.5.1 to find the potentially close segments.

Complexity. We have at the worst case n tests for a valid m. From Theorem 8.5 each one takes $O(n(\log n + w \log w))$ time. Thus the total time complexity is $O(n^2(\log n + w \log w))$. The working storage is $O(nw)$ as in Theorem 8.5.

The problem is that this approach increases the asymptotic time complexity of the algorithm significantly.

II We simply let $m = n$ and calculate the values of the parameters in Δ.

Complexity. There is a constant number of computations. Thus the total time is $O(1)$.

In this way, the parameters of Δ get upper bound values which are obviously valid for applying CPLS. Nevertheless, this approach is problematic since although we save time, we may get very big perturbation radii which degrade the quality of the output.

III We compute the parameters of Δ when $m = n$. The values we get are valid upper bounds since m is upper-bounded by n. With these values we find what is the number of segments which are τ-close to a certain segment $s \in S$ and set it to m. Obviously, this value is an upper bound of the real m because the real m is supposed to be found with parameters which are less than or equal to the ones we get here. Now we recalculate the values of the parameters in Δ according to the newly found m. We use the tiling technique here too.

Complexity. The same as the tiling technique since it is applied once: $O(n(\log n + w \log w))$ time and $O(nw)$ working storage.

This approach has a much better time complexity than the first one. Although we can get larger perturbation radii, our experiments have shown that they are still relatively small. In that sense, it is better than the second approach. Thus we use this approach throughout the article and in our implementation. All the complexity calculations are effected by this approach.

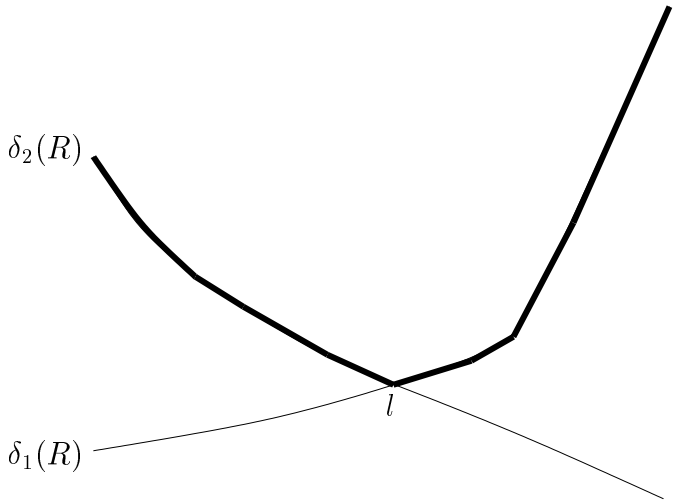


Figure 8.2: δ_1 and δ_2 as functions of R. The thick line is $G_\delta(R)$

8.7 Approximating the Best Ratio (R)

This section describes yet another preprocessing step. Thus it is strongly related to the procedure that is described in Section 8.6. Recall that R determines the ratio of the two resolution parameters ρ_1 and ρ_2. A nice and simple approach would be to set R to be the one that makes the maximum of the perturbations radii minimal. For simplicity we choose R to be an integer. It has to be bigger than 1 so the square root in the inequality defining δ_2 is real (see Theorem A.4). Notice from the values of δ_1,δ_2 and ρ_2 in Theorem A.4 that if all the parameters but R are constant, $\delta_1(R)$ is an increasing monotone function and $\delta_2(R)$ is a decreasing monotone one.

Let $G_\delta(R) = max(\delta_1(R), \delta_2(R))$. We get that $G_\delta(R)$ has one local minimum, l (see Figure 8.2). We are interested in finding l since it is exactly the value for which the maximum between the perturbations radii is minimal. More precisely, for convenience we search only for integers bigger than 1. Let l' denote the local minimum for integers. We find l' by applying a variation of binary search with R, while R ranges from 1 to some very big value (denoted by M). M can be the maximum integer, the maximum double, or other constants depending on the architecture (we chose it to be the maximum double). For each R being checked, we make an iteration of approach III in order to find the values of the various parameters. When the binary search is over, R is set to l'. The exception is the following case. Although M has a very big value, there is a possibility that $l' > M$. In that case R is set to M. Except for this extreme case, the deviation from the optimal R is small. The following theorem summarizes the preprocessing work of CPLS:

Theorem 8.7 *The preprocessing work of CPLS takes* $O(n \log M(\log n + w \log w))$ *time and* $O(nw)$ *working storage.*

Proof: Since we make a binary search over M, each time applying the approach III, the time bound follows. The working storage is not effected by the search. □

8.8 Discussion: Exact Arithmetic vs. Finite-Precision Arithmetic

An important discussion concerning the implementation of geometric algorithms is whether to use finite-precision or exact arithmetic. By using exact arithmetic, we are guaranteed to get accurate results. Unfortunately, by using the available exact arithmetic number types which support square root operation (as CPLS requires), we get a huge time and space overhead. On the other hand, under certain assumptions, CPLS can use finite-precision arithmetic and still produce valid results. The assumption is that the resolution parameter ε_0 is chosen sufficiently big such that degeneracies are not induced due to the errors with the machine precision. (We are currently investigating this issue to determine the relation between ε_0 and the machine precision.) We implemented our software with floating-point arithmetic. Earlier Controlled Perturbation [25, 34] also rely on this assumption when using floating-point arithmetic.

The idea behind our scheme is that since the output has a finite-precision representation and has no degeneracies, following manipulations that use finite-precision arithmetic can safely use it. The same assumption mentioned above regarding the resolution parameter holds here too, namely that the resolution parameter is chosen sufficiently big so that degeneracies are not induced due to the errors with the machine precision.

We next point out another problem which may arise when using finite-precision arithmetic. This problem is relevant to the earlier Controlled Perturbation algorithms as well.

Points inside forbidden loci. Recall that the radius of the perturbation disc is chosen such that the area of the disc is at least twice bigger than the sum of the areas of all possible forbidden loci (namely, those that induce degeneracies). Thus when picking up a point inside the disc, we have a probability of at least $\frac{1}{2}$ to be outside the forbidden loci. If we use finite-precision arithmetic, it is not obvious that this is the case. Consider the example in Figure 8.3. The point p has to be perturbed inside the disc C where the resolution parameter is ε. Since we use finite-precision arithmetic, we have a grid of points, K, to pick up randomly inside C (the black points in Figure 8.3). The forbidden loci are induced by the five strings crossing C (denoted by Z). These are the shaded rectangles in Figure 8.3. Note that the width of each $z \in Z$ is 2ε. Let $Y = Z \cap C$. Therefore, Y is the forbidden loci. The strings are placed such that they cover all the points of K. Thus we never get a degenerate-free

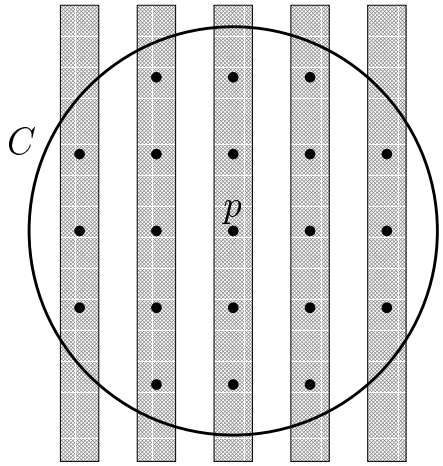


Figure 8.3: A case where all the grid points are inside the forbidden loci

placement. Even if not all the points are inside forbidden loci, the probability to get a degenerate-free placement can be significantly less that $\frac{1}{2}$, resulting in a hard work to find one.

From the example in Figure 8.3 we can conclude that if there are m segments inducing these strings and the perturbation radius is δ, then $\delta \approx 2m\varepsilon$. However, the real magnitudes of δ_1 and δ_2 are usually much bigger than $2m\varepsilon$ and therefore this example is not realistic. Here the practical assumption is that ε is sufficiently larger than the machine precision and hence situations as the one depicted in Figure 8.3 will be ruled out. Our experiments confirm that this problem does not arise.

8.9 The Main Theorem

We conclude this chapter by giving the main theorem of CPLS.

Theorem 8.8 *Given n input line segments and a resolution parameter ε_0, a valid perturbation of the line segments can be computed in $O(n \log M(\log n + w \log w) + nwc(\log \frac{\delta_1}{\rho_1} + w \log \frac{\delta_2}{\rho_1}))$ expected time and $O(nw)$ working storage, and the output size is n, where the parameters are described below. M is a big constant that can be expressed in the number type that is used. w is the maximum number of line segments in a grid square as described in Section 8.5.1. c is the number of trials done in order to find a degenerate-free locus before enlarging the radius of the perturbation disc. $\rho_1 = R\rho_2$ is the biggest resolution parameter related to the first perturbation phase where $\rho_2 = \sqrt{\frac{(1+2\lambda)\varepsilon_0 L}{\sqrt{1-\frac{\lambda^2}{4}}}}$ is related to the second perturbation phase, R is simply the ratio $\frac{\rho_1}{\rho_2}$, L is the length of the longest input line segment in the input and λ is the acceptance parameter of CPLS. δ_1 and δ_2 are the perturbation radii for the first and second vertices of a line segment respectively. The values of δ_1 and δ_2 are chosen such*

that $\delta_1 \geq \frac{8mR\rho_2}{\pi}$ *and* $\delta_2 \geq \frac{4\rho_2}{\pi}\left(\frac{m(m+3)(L+\frac{8mR\rho_2}{\pi})}{\rho_2\sqrt{R^2-1}} + 2(m+1)\right)$ *where* m *is the maximum number of line segments that were inserted into the output by the time that a certain* $s \in S$ *is inserted, which can be very close to* s *or intersect it (See Appendix A for more details on these magnitudes.)*

Proof: The complexities above are the results of summing up the complexities in Theorems 8.6, 8.7, A.1 and A.3. The output has $O(n)$ size since each input line segment contributes one, possibly perturbed, line segment to the output. □

Chapter 9

Implementation Details

The implementation details we described that are not specific to ISR (see Chapter 5) are relevant here too. We next describe more implementation details of the CPLS package.

The main input and the main output of the package are sets of line segments. The Controlled Perturbation is a framework for several types of algorithms (see Chapter 7). Thus we have designed our package more generally for Controlled Perturbations in $\mathbb{R}^2$. The main class is a general frame to perform Controlled Perturbation algorithms. It is separated from the type of objects that use it. Its knowledge is the Controlled Perturbation framework (see Chapter 7) and the optimizations (see Section 8.5). In order to use it, it has to be *templated* with the information of the object, its perturbation details, and as other classes of CGAL [6], the arithmetic number type. The object and its perturbation are separated in order to allow different perturbation schemes for the same type of objects. The advantage of our design is that once the Controlled Perturbation frame class is implemented, it is suitable for other kinds of objects and perturbations. Using this frame, we built the CPLS package by templating both line segments and our perturbation algorithm. Future Controlled Perturbation algorithms in $\mathbb{R}^2$ will be able to apply the same frame class, making the work easier to implement and support. This was exactly the case when we developed and implemented Controlled Perturbation algorithms for both arrangements of polygonal lines and arrangements of polygons (which are extensions not described in this thesis). We used floating-point arithmetic for implementing CPLS.

Chapter 10

Experimental Results

We present two kinds of experimental results obtained with the CPLS package. For each one we give quantitative magnitudes that indicate the quality of CPLS.

We choose ε_0 such that the other resolution parameters (ρ_1 and ρ_2) will be sufficiently large compared with the resolution of the standard double (floating-point) number type. Notice that the only computation involving ε_0 is when we determine ρ_2. We calculated ρ_2 using LEDA's bigfloat number type[1] [30]. We observed and confirmed that the computed value ρ_2 was the same when using either bigfloat or double. The rest of the computation was carried out with the standard machine double.

The abbreviations we use in this section are explained in Table 10.1.

10.1 Congestion Data

The input for this example is similar to the one we used in Section 6. The difference is that this one has only 100 segments with 2150 intersections. The segments' bounding box lower left corner is $(0, 0)$ and upper right corner is $(100, 100)$. For clarity, the pictures in Figure 10.1 depict a similar example with resolution 1e-15 so that the perturbations are sufficiently large to be visible. In order to show the differences between the input and the output more clearly, the bottom left part of the example is zoomed in.

δ_1 and δ_2, the number of perturbed vertices and the actual size of the perturbations are bigger as we use a bigger ε_0. The actual average and maximum perturbations are much smaller than δ_1 and δ_2. It shows that the optimization that we describe in Section 8.5.2 reduces the perturbation magnitudes significantly. The average number of trials to find a valid perturbation for a vertex does not exceed 3.476.

[1] LEDA's bigfloat mimics floating-point representation with user-fixed mantissa length (we set it to 100) and arbitrary length exponent.

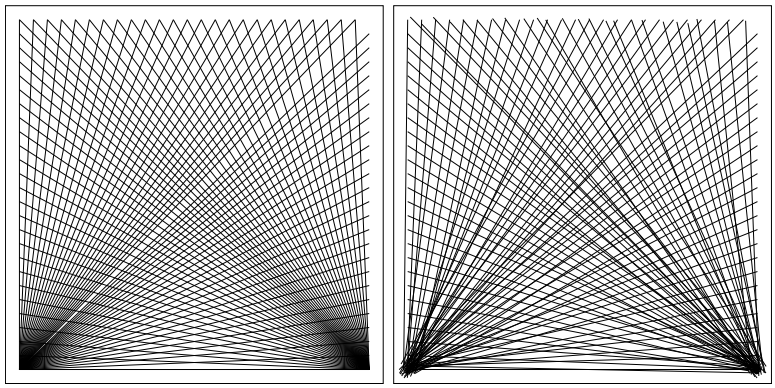
Abbreviation	Explanation
n	number of input line segments
nvp	average number of perturbed vertices
ap	average perturbation
mp	maximum perturbation
ant	average number of trials to find a valid perturbation for a vertex

Table 10.1: Abbreviations

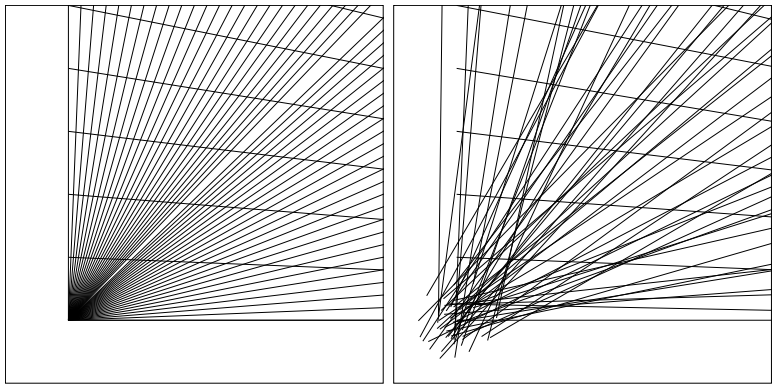
10.2 Random Data

The input for this example is a set of line segments whose coordinates are chosen randomly inside a bounding rectangle whose lower left corner is $(0, 0)$ and upper right corner is $(1, 1)$. We divided the experiments into four different numbers of segments: 100, 200, 300 and 400. For each one we created a random set and tested each one several times, each time with a different resolution parameter. For clarity the pictures in Figure 10.2 depict a similar example with only 50 segments in a bigger resolution parameter (2e-9) so that the perturbations are sufficiently large to be visible. In order to provide a nice example with visible degeneracies elimination, the coordinates of three segments of the input were determined by us instead of being randomly chosen. The input-zoom-in picture shows a part of the set in which these three line segments induce degeneracies with other line segments. The degeneracies and their elimination, as shown in the output-zoom-in picture, are clearly visible.

We tested different number of line segments to see the effect of the density on the results. When we increase the number of line segments, ρ_2 does not change while ρ_1, δ_1, δ_2, the number of perturbed vertices and the actual average and maximum perturbations become bigger. As in the previous example, the actual average and maximum perturbations are much smaller than δ_1 and δ_2. The effect of changing the resolution parameter is also similar to the previous example. The average number of trials to find a valid perturbation for a vertex does not exceed 2.01.



Input Output

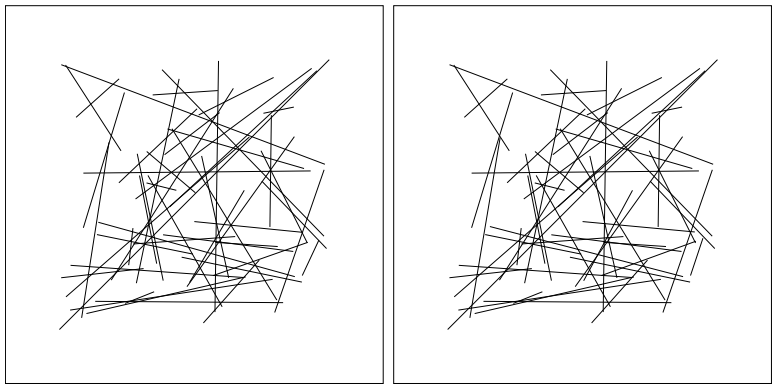


Input zoom in Output zoom in

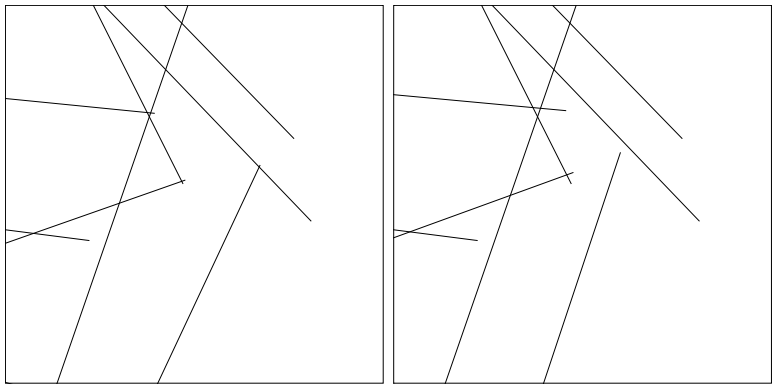
ε_0	ρ_1	ρ_2	δ_1	δ_2	nvp	ap	mp	ant
1e-18	9.384e-3	1.201e-8	2.365	2.365	149	0.092	0.348	3.476
1e-20	2.95e-3	1.201e-9	0.743	0.743	149	0.028	0.115	3.268
1e-22	9.315e-4	1.201e-10	0.234	0.234	149	9.114e-3	0.0346	3.241
1e-24	2.943e-4	1.201e-11	0.074	0.074	149	2.866e-3	0.011	3.093
1e-26	9.308e-5	1.201e-12	0.023	0.023	149	9.019e-4	2.904e-3	3.073

Statistics (n = 100)

Figure 10.1: Congestion data



Input Output



Input zoom in Output zoom in

n	ε_0	ρ_1	ρ_2	δ_1	δ_2	nvp	ap	mp	ant
100	1e-17	5.483e-4	3.798e-9	0.138	0.138	1	3.338e-3	3.338e-3	1
100	1e-18	3.021e-4	1.201e-9	0.076	0.076	1	1.986e-3	1.986e-3	1
100	1e-19	1.679e-4	3.798e-10	0.042	0.042	1	1.367e-3	1.367e-3	1
100	1e-20	9.384e-5	1.201e-10	0.023	0.023	1	7.109e-4	7.109e-4	1
200	1e-17	8.401e-4	3.798e-9	0.425	0.425	21	6.009e-3	8.349e-3	1.428
200	1e-18	4.46e-4	1.201e-9	0.226	0.226	12	3.493e-3	4.459e-3	1.416
200	1e-19	2.428e-4	3.798e-10	0.123	0.123	7	1.657e-3	2.403e-3	1
200	1e-20	1.34e-4	1.201e-10	0.067	0.067	5	7.78e-4	1.229e-3	1.2
300	1e-17	1.145e-3	3.798e-9	0.871	0.871	50	7.679e-3	0.021	1.74
300	1e-18	5.801e-4	1.201e-9	0.441	0.441	29	3.736e-3	5.633e-3	1.31
300	1e-19	3.074e-4	3.798e-10	0.234	0.234	16	1.944e-3	2.97e-3	1.062
300	1e-20	1.672e-4	1.201e-10	0.127	0.127	9	1.202e-3	1.518e-3	1.222
400	1e-17	1.496e-3	3.798e-9	1.52	1.52	114	0.01	0.298	2.01
400	1e-18	7.196e-4	1.201e-9	0.731	0.731	60	4.652e-3	7.923e-3	1.683
400	1e-19	3.696e-4	3.798e-10	0.375	0.375	39	2.565e-3	6.739e-3	1.384
400	1e-20	1.974e-4	1.201e-10	0.2	0.2	19	1.455e-3	1.973e-3	1.421

Statistics

Figure 10.2: Random data

Chapter 11

Implementation Details

We implemented both ISR and CPLS with C++, using many capabilities of CGAL [6]. Each package defines a C++ *class* to work on [38]. The programmer uses our packages by creating instances of these classes. The implementation is generic in the sense that each class is templated with a number type with which the data are composed of. The user of our software chooses which number type to apply with the *template* mechanism of the C++ language.

The main input of the applications is a set of line segments while the outputs are a set of polygonal chains of segments for ISR and a set of the perturbed segments for CPLS.

The user can choose the output format. It can be either a text file describing the arrangements in the output or a graphic window in which both the input and the output are drawn (the graphic window is the LEDA window [29]).

Except for CGAL capabilities that we explicitly mention, we applied other CGAL elements such as geometric predicates, points, segment, vector and intersection operations.

We next discuss implementation details of each package.

ISR. The ISR package supports both ISR and SR. It is up to the user to decide which one to apply. Generally speaking, the way to convert the ISR algorithm to SR is simply to constrain the recursion depth of the Reroute routine to one (see Chapter 3), meaning that the output polygonal chains are determined immediately by the hot pixels that the original segments intersect.

Since, as presented, ISR and SR must be implemented with an exact number type, we implemented the package with the Leda rational number type [29].

Recall that we use the c-oriented kd-trees as our search structure (see Chapter 5) for ISR. As a first step for creating the c-oriented kd-trees, we have to find the hot pixels. This is done by applying a plane sweep algorithm [6]. Recall that the c-

oriented kd-trees are composed of several kd-trees. We use the kd-tree package of CGAL to implement that. The user has the ability to choose the number of trees to use.

The ISR package has become a part of CGAL.

CPLS. The Controlled Perturbation is a framework for several kinds of algorithms (see Chapter 7). Thus we have designed our package more generally for Controlled Perturbations in $\mathbb{R}^2$. The main class is a general frame to perform Controlled Perturbation algorithms. It is separated from the kind of objects that use it. Its knowledge is the the Controlled Perturbation frame and the optimizations for reducing perturbation size and the tiling of the plane (see Chapter 8). In order to use it, it has to be *templated* with the information of the object, its perturbation details, and as other classes of CGAL, the arithmetic number type. The object and its perturbation are separated in order to allow different perturbation schemes for the same kind of objects. The advantage of our design is that once the Controlled Perturbation frame class is implemented, it is suitable for other kinds of objects and perturbations. Using this frame, we built the CPLS package by templating both line segments and our perturbation algorithm. Future Controlled Perturbation algorithms in $\mathbb{R}^2$ will be able to apply the same frame class, making the work easier to implement and support. This was exactly the case when we developed and implemented Controlled Perturbation algorithms for both arrangements of polygonal lines and arrangements of polygons (which are beyond the scope of this thesis).

We used floating-point arithmetic for implementing CPLS.

Chapter 12

Conclusion

We presented two types of finite-precision approximation techniques for arrangements of segments in the plane. The goal of these techniques is to create robust data for further manipulation of the input. Each technique may be suitable in different situations. We implemented both techniques and presented experimental results obtained with our implementation.

12.1 Iterated Snap Rounding

We presented an augmented Snap Rounding procedure which rounds an arbitrary precision arrangement of segments in $\mathbb{R}^2$ with the advantage that each vertex in the rounded arrangement is at least half a unit away from any non-incident edge. The new scheme makes the rounded arrangement more robust for further manipulation with limited precision arithmetic than the output that the standard Snap Rounding algorithm produces. We have proved that the maximum distance between an original segment and its output chain is $\Theta(n^2)$ in the worst case. On the other hand, many examples have demonstrated a very small deviation, no more than a small constant number of pixels. We believe that real-world data behave in this way and not like pathological examples such as the one we used to prove the lower bound. We implemented ISR using exact arithmetic.

We propose several directions for further research: (1) Can detecting all the hot pixels through which an output chain passes be done more efficiently? (2) Extend the scheme to non-linear curves. (3) The rounded arrangement can have at most $O(n^2)$ segments, whereas our algorithm (as well as the known algorithms for SR) may produce $\Omega(n^3)$ output links. The task here is to devise an output sensitive algorithm where the output size is the size of the rounded arrangement and not the overall complexity of the chains. (4) Improve the heuristics for choosing the directions of the kd-trees. (5) Find a scheme that controls both the distance of a vertex and a non-incident edge and the maximum perturbation magnitude.

12.2 Controlled Perturbation of Line Segments

We presented an algorithm that eliminates degeneracies from an arrangement of line segments by perturbing the endpoints of the input segments slightly. Thus making the rounded arrangement more robust for further manipulation. We implemented the algorithm using floating-point arithmetic. Our experimental results have generated relatively very small perturbations.

We have recently also developed algorithms for Controlled Perturbations of both arrangements of polygonal lines and arrangements of polygons. We implemented both of them and achieved good results. We intend to report on these Controlled Perturbations algorithms in a separate report.

We propose several directions for further research: (1) In our work we assumed that we are given a sufficiently large ε so that computing with floating-point arithmetic can be carried out safely. To fill up the gap here one needs to determine, given the specific arithmetic precision, what is the smallest $\varepsilon > 0$ with which all the computations in CPLS can be done safely. (2) Recall that the perturbations of the first and second endpoints are different. The result is that it is possible that the output of CPLS changes if we change the order of the endpoints of some of the line segments. Another effect on the result of CPLS can be induced by changing the order of insertion of the line segments in the input since the perturbation of a certain line segment depends on the segments that precede it. The task here is to devise a way for determining good orders both of the line segments in the input and of the endpoints along the line segments. Good orders would be ones with which there is a considerable probability that the actual perturbation magnitudes would be smaller than the ones achieved with random orders. (3) Apply the Controlled Perturbation scheme to other kinds of objects.

Appendix A

Computing δ_1 and δ_2

In this appendix we derive upper bounds on δ_1 and δ_2, the perturbation radii of the first and second endpoints of a line segment respectively.

We remind the reader of the notation introduced in Chapter 8. The same notation is used throughout the appendix. $S = \{s_1, s_2, \ldots, s_n\}$ are the input line segments ordered arbitrarily. We denote by $s_i \in S$ the segment that is currently perturbed. We denote its endpoints by p_i and q_i (we relate to p_i as the first endpoint, and to q_i as the second one; the order is arbitrary). Let p'_i be the result of the first phase on p_i, either perturbed or not. Let s'_i be the segment $\overline{p'_i q_i}$. Let q'_i be the result of the second phase on q_i, either perturbed or not. Let s''_i be the segment $\overline{p'_i q'_i}$. Let $S''_i = \{s''_1, s''_2, \ldots, s''_i\}$ be the set of the first i perturbed segments. Let $P'_i = \{p'_1, \ldots, p'_i, q'_1, \ldots, q'_i\}$. Let $A = \{s'' | s \in S\}$. As we mentioned in Section 8.2, A is the output of CPLS. We denote by m the maximum number of line segments inserted by the time any segment s is inserted, that can induce degeneracies with s. w is the maximum number of line segments that intersect or are τ-close to a grid square as described in Section 8.5.1.

A.1 First Phase: Computing δ_1

Recall that δ_1 is the perturbation radius of the first endpoint and ρ_1 is the resolution parameter in this case. Each $s'' \in S''_{i-1}$ defines a forbidden placement for p'_i. This placement is the Minkowski sum of s'' and a disc centered at the origin with a radius ρ_1. It is easy to show that the maximum area which it can cut from the perturbation disc is when s'' passes through p_i and intersects the perturbation disc twice. This area is bounded by a rectangle whose area is $2\rho_1 \times 2\delta_1$ (rectangle *abcd* in Figure A.1). There is an upper bound of m segments defining such loci — see Section 8.6. The area of the perturbation disc has to be at least twice bigger than the sum of the areas of all the forbidden loci. Since the perturbation disc area is $\pi\delta_1^2$ we get:

$$\pi\delta_1^2 \geq 8m\rho_1\delta_1$$

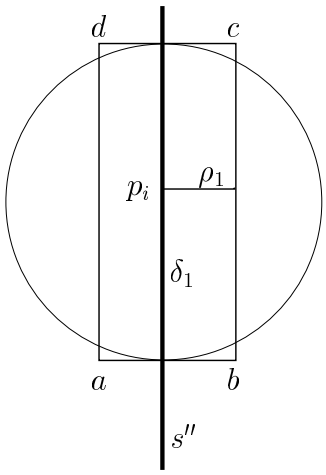


Figure A.1: Forbidden loci induced by $s'' \in S''_{i-1}$

$$\delta_1 \geq \frac{8m\rho_1}{\pi} \tag{A.1}$$

Recall that w is the maximum possible number of segments in each square of the tiling (see Section 8.5.1) and c is a parameter we fix for the number of perturbation trials we make before enlarging the perturbation radius (see Section 8.5.2). The next theorem summarizes the time complexity of the first phase.

Theorem A.1 *The first phase for all the segments together takes $O(nwc \log \frac{\delta_1}{\rho_1})$ expected time.*

Proof: Picking up a random point inside the disc takes $O(1)$ time. Each test involves a computation of a random point in a disc and the distance between a segment and an endpoint ($O(1)$ time). There is an upper bound of $O(w)$ segments to test (see Section 8.5.1). According to Theorem 8.6, in the worst case we have $O(c \log \frac{\delta_1}{\rho_1})$ trials, each one consists of at most $4w$ tests. We get that $\Psi = O(w)$. Thus the first phase for all the segments together takes $O(nwc \log \frac{\delta_1}{\rho_1})$ expected time. □

A.2 Second Phase: Computing δ_2

As noted in Section 8.4, there are several different cases of degeneracies in the second phase. Each one of them induces forbidden loci. In Sections A.2.1-A.2.4 we describe these cases. Each test of each of these cases takes $O(1)$ time. In Section A.2.5 we compute the value of δ_2 and the complexity of the second phase.

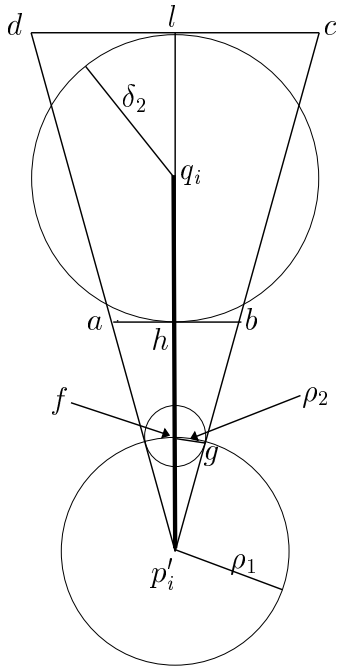


Figure A.2: Forbidden loci of an endpoint/intersection

A.2.1 Computing Forbidden Loci Induced by P'_{i-1} and s_i

Recall that ρ_2 denotes the resolution parameter used in this case. Here, s''_i must not penetrate the disc of radius ρ_2 centered at the points of P'_{i-1}. This is demonstrated in Figure A.2, where s'_i is the thick line whose first endpoint, p'_i, has already been perturbed, and f is an already inserted endpoint which must be at least ρ_2 away from s''_i in order not to induce degeneracies (we explain later why we place f at the intersection between s'_i and the disc of radius ρ_1 around p'_i). In order to prevent s''_i from penetrating the disc with radius ρ_2 around f, q'_i must not be located inside the wedge dp'_ic. It defines a trapezoid which bounds the forbidden loci (trapezoid $abcd$ in Figure A.2). This trapezoid is maximal when f is located on s'_i and on the disc with the radius ρ_1 around p'_i (this disc does not contain anything but s'_i since p_i has already been successfully perturbed) and when s'_i's length is maximal ($L + \delta_1$, where L is the length of the longest input segment and δ_1 is the maximum perturbation that p_i could have been perturbed in the first phase). This explains our choice where to place f.

Note that g in Figure A.2 is the place where the segment $\overline{p'_ic}$ is tangent to the disc centered at f. We denote by D the maximum area of a trapezoid $abcd$ (the forbidden loci). Next we compute its magnitude.

$$\begin{aligned} \Delta p'_i g f &\approx \Delta p'_i h b \approx \Delta p'_i l c \\ \frac{\rho_2}{|\overline{p'_i g}|} &= \frac{|\overline{ab}|/2}{L + \delta_1 - \delta_2} = \frac{|\overline{dc}|/2}{L + \delta_1 + \delta_2} \end{aligned}$$

$$\begin{aligned} |\overline{ab}| &= \frac{2\rho_2(L+\delta_1-\delta_2)}{|\overline{p_i'g}|} \\ |\overline{dc}| &= \frac{2\rho_2(L+\delta_1+\delta_2)}{|\overline{p_i'g}|} \\ D &= (|\overline{ab}|+|\overline{dc}|)\delta_2 \end{aligned}$$

$$D = \frac{4\rho_2\delta_2(L+\delta_1)}{\sqrt{\rho_1^2-\rho_2^2}} \tag{A.2}$$

We need to coordinate between ρ_1 and ρ_2 in order to compute δ_1 and δ_2 in terms of the input parameters. Let R be the ratio $\frac{\rho_1}{\rho_2}$ (we have described in Section 8.7 how to determine R). Then

$$\rho_1 = R\rho_2 \tag{A.3}$$

If ρ_2 is not much smaller than ρ_1 then in Figure A.2 $\angle dp_i'c$ is not very small. Thus the size of the trapezoid *abcd*, which is the forbidden loci in that case, may be unacceptably big. So we expect ρ_2 to be much smaller than ρ_1. We also cannot make it arbitrarily small because the bigger the R is, the bigger ρ_1 would be, resulting in a big perturbation for the first endpoint — see Inequality A.1. In Section 8.7 we propose a way to find an R for which the biggest perturbation radius is small.

We get that ρ_1 must be greater than ρ_2. Thus the square root in equation A.2 is real. Since there are $2m$ possible endpoints for this case of degeneracy, the total area of the forbidden loci in this case is bounded by:

$$F_1 = \frac{8m\rho_2\delta_2(L+\delta_1)}{\sqrt{\rho_1^2-\rho_2^2}} \tag{A.4}$$

A.2.2 Computing Forbidden Loci Induced by Intersections of Segments of S_{i-1}'' and s_i

The resolution parameter in this case is ρ_2 too. The effect of an intersection is the same as the effect of an endpoint as described in Section A.2.1. The only difference is that there are at most $\binom{m}{2} = \frac{m(m-1)}{2}$ such intersections. Thus the total size of the forbidden loci in this case is upper bounded as follows:

$$F_2 = \frac{2m(m-1)\rho_2\delta_2(L+\delta_1)}{\sqrt{\rho_1^2-\rho_2^2}} \tag{A.5}$$

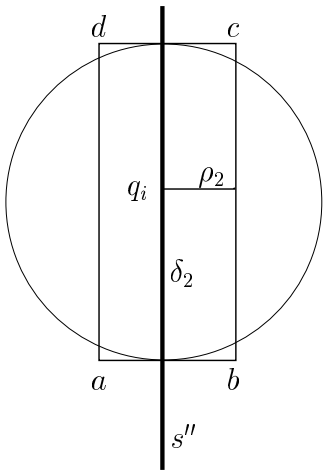


Figure A.3: Forbidden loci induced by q_i and $s'' \in S''_{i-1}$

A.2.3 Computing Forbidden Loci Induced by $S''_{i-1} \cup \{p'_i\}$ and q_i

First we discuss the forbidden loci induced by segments of S''_{i-1}. This case is similar to the first phase described in Section A.1, but this time we want the resolution parameter to be ρ_2, and the perturbation radius to be δ_2. As shown in Figure A.3, the size of the bounding rectangle is $2\rho_2 \times 2\delta_2 = 4\rho_2\delta_2$. The size of the forbidden loci for p'_i is $\pi\rho_2^2$ which is definitely smaller than $4\rho_2\delta_2$. Since we are interested in computing an upper bound on the area of the forbidden loci, we can bound it by $4\rho_2\delta_2$ and regard it as it was an above bounding rectangle in our analysis. Since there are at most $m+1$ objects that may induce degeneracies with q_i (m segments from S''_{i-1} and p'_i), the total size in this case is

$$F_3 = 4(m+1)\rho_2\delta_2 \tag{A.6}$$

A.2.4 A Lower Bound on the Distance Between an Intersection of s_i with an Already Inserted Segment and an Already Inserted Segment

Let s''_j and s''_k be two already inserted segments where s''_j intersects s''_i at a point f. We next argue that if all the degeneracies above are not induced after possibly perturbing s_i giving s''_i, then a degeneracy of type D_2 involving f and s''_k cannot arise as well. We do so, without loss of generality, by giving a lower bound on the distance between such intersection f and s''_k; we denote this lower bound by ρ_3. Assume that this type of degeneracy involves s''_i (perturbed to take care of the cases in phase 1 and in Sections A.2.1, A.2.2 and A.2.3), s''_j and s''_k. By that we show that this kind

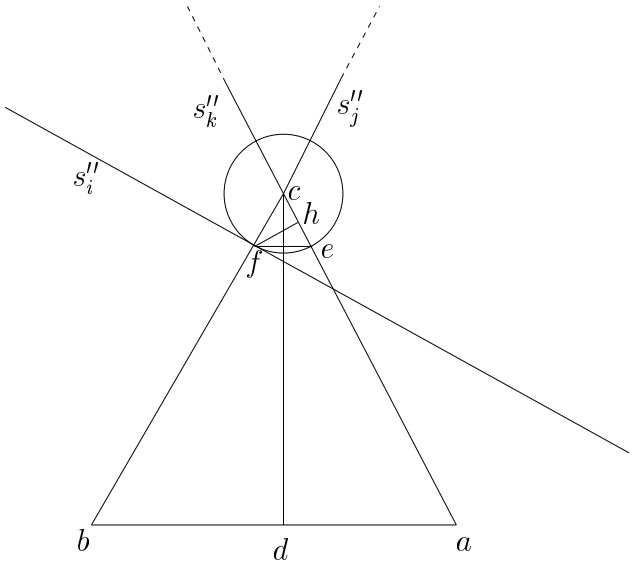


Figure A.4: Minimal distance when two segments of S_{i-1}'' intersect

of degeneracy is eliminated automatically after eliminating all the other degeneracies that take place in CPLS. Thus we can ignore this degeneracy when perturbing line segments although it effects the magnitudes of the resolution parameters and the perturbation radii.

We differentiate between two cases:

The first one is when s_j'' and s_k'' do not intersect each other. Since each one of their endpoints is at least ρ_1 far away from the other segment, this also holds for $f \in s_j''$ and s_k''. Therefore the lower bound in this case is $\rho_3 = \rho_1$. Thus no degeneracy may be induced in this case.

The second case is when s_j'' and s_k'' intersect. This case is demonstrated in Figure A.4. The two already inserted segments, s_j'' and s_k'' force s_i'' not to penetrate the disc C, centered at their intersection, c, with a radius ρ_2. Thus f, the intersection between s_i'' and s_j'', would be closest to s_k'' if it is placed on C. Moreover, the smaller the angle $\angle bca$ (denoted by α) is, the smaller ρ_3 is (in Figure A.4 it the size of $\overline{fh}$ - the distance from f to s_j''). α is minimal when s_j'' and s_k'' have a maximal length below their intersection (bounded by $L+\delta_1+\delta_2$) and when the distance between their lower endpoints (a and b are the endpoints in the figure) is minimal, bounded by ρ_2 in this case (the resolution parameter for the second endpoint). Under these conditions, we next compute a lower bound on the length of the segment $\overline{ef}$.

We get:

$$\frac{|\overline{ef}|}{\rho_2} = \frac{\rho_2}{L+\delta_1+\delta_2}$$

$$|\overline{ef}| = \frac{\rho_2^2}{L+\delta_1+\delta_2}$$

We assume that the perturbation is λ-acceptable according to Definition 8.1, otherwise CPLS would not be applied. Let σ denote any resolution parameter or perturbation radius. Then $\sigma \leq \lambda L$. Together with a simple trigonometric observation in Figure A.4, we get that

$$\begin{aligned}
cos(\frac{\pi}{2}-\frac{\alpha}{2}) &= \frac{\rho_2/2}{L+\delta_1+\delta_2} \\
cos^2(\frac{\pi}{2}-\frac{\alpha}{2}) &= \frac{\rho_2^2}{4(L+\delta_1+\delta_2)^2} \\
sin^2(\frac{\pi}{2}-\frac{\alpha}{2}) &= 1-\frac{\rho_2^2}{4(L+\delta_1+\delta_2)^2} \geq \\
1-\frac{(\lambda L)^2}{4(L+\delta_1+\delta_2)^2} &\geq 1-\frac{\lambda^2}{4} \\
sin(\frac{\pi}{2}-\frac{\alpha}{2}) &\geq \sqrt{1-\frac{\lambda^2}{4}} \\
sin(\frac{\pi}{2}-\frac{\alpha}{2}) &= \rho_3/|\overline{ef}| \\
\rho_3 &\geq |\overline{ef}|\sqrt{1-\frac{\lambda^2}{4}} \\
\rho_3 &\geq \frac{\rho_2^2\sqrt{1-\frac{\lambda^2}{4}}}{L+\delta_1+\delta_2} \\
\rho_3 &\geq \frac{\rho_2^2\sqrt{1-\frac{\lambda^2}{4}}}{(1+2\lambda)L}
\end{aligned}$$

Recall that we choose $\lambda = \frac{1}{10}$.

The next lemma argues that ρ_3 in this case is the smallest resolution parameter.

Lemma A.2 $\rho_3 < \rho_2 < \rho_1$

Proof: Since we fixed ρ_2 to be smaller than ρ_1, we only have to prove that $\rho_3 < \rho_2$. Consider Figure A.4: if CPLS is λ-acceptable, then $\angle bca$ is sufficiently small so that ρ_3 (the length of segment $\overline{fh}$) is smaller than ρ_2 (the length of segment $\overline{ce}$). The claims follows. □

We do not encounter other magnitudes of resolution parameters, thus ρ_3 should be the input resolution parameter. Then $\varepsilon_0 = \rho_3$ and we get that:

$$\varepsilon_0 \geq \frac{\rho_2^2\sqrt{1-\frac{\lambda^2}{4}}}{(1+2\lambda)L}$$

We need to compute the value of ρ_2 in terms of the input parameters. If we change the inequality above to an equation, we obtain an upper bound on ρ_2 which we use below. We get that

$$\rho_2 = \sqrt{\frac{(1+2\lambda)\varepsilon_0 L}{\sqrt{1-\frac{\lambda^2}{4}}}} \tag{A.7}$$

A.2.5 Computing δ_2

As in the first phase, we want the perturbation disc size to be at least twice bigger than the total area of all the forbidden loci. Then by using formulas A.4, A.5 and A.6, $\pi\delta_2^2 \geq 2(F_1 + F_2 + F_3)$ and we get that:

$$\delta_2 \geq \frac{4\rho_2}{\pi}\left(\frac{m(m+3)(L+\delta_1)}{\sqrt{\rho_1^2-\rho_2^2}} + 2(m+1)\right) \tag{A.8}$$

The next theorem summarizes the complexity of the second phase.

Theorem A.3 *The second phase for all the segments takes $O(nw^2c\log\frac{\delta_2}{\rho_1})$ expected time.*

Proof: Each test for degeneracies takes $O(1)$ time. Since we have an upper bound of $O(w)$ segments to check in each perturbation, $O(w)$ tests are done as described in Sections A.2.1 and A.2.3 while $O(w^2)$ tests are done as described in Section A.2.2 . We get that $\Psi(w) = O(w^2)$. According to Theorem 8.6, the second phase for all the segments together takes $O(nw^2c\log\frac{\delta_2}{\rho_1})$ expected time. □

A.3 Concluding Perturbation Radii

We conclude the Appendix with a theorem that summarizes the magnitudes of δ_1 and δ_2.

Theorem A.4 *The magnitudes of δ_1 and δ_2 are:*

$$\begin{aligned}\delta_1 &\geq \frac{8mR\rho_2}{\pi} \\ \delta_2 &\geq \frac{4\rho_2}{\pi}\left(\frac{m(m+3)(L+\frac{8mR\rho_2}{\pi})}{\rho_2\sqrt{R^2-1}} + 2(m+1)\right)\end{aligned}$$

where

$$\rho_2 = \sqrt{\frac{(1+2\lambda)\varepsilon_0 L}{\sqrt{1-\frac{\lambda^2}{4}}}}$$

Proof: The magnitudes are derived immediately from the Equations and Inequalities A.1, A.3, A.7 and A.8. □

Bibliography

[1] P. K. Agarwal and M. Sharir. Applications of a new space-partitioning technique. *Discrete Comput. Geom.*, 9:11–38, 1993.

[2] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 211–219, 1995.

[3] 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.

[4] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[5] J. Canny, B. R. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 251–260, 1992.

[6] *The CGAL User Manual, Version 2.4*, 2002. www.cgal.org.

[7] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.

[8] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algortihmica*, 12:30–53, 1994.

[9] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Heidelberg, Germany, 1997.

[10] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20:523–547, 1998.

[11] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[12] I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1–2):219–242, Sept. 1997.

[13] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. *Software - Practice and Experience*, 30:1167–1202, 2000.

[14] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[15] E. Flato. Robust and efficient construction of planar Minkowski sums. M.Sc. thesis, Dept. Comput. Sci., Tel Aviv University, Tel Aviv, Israel, 2000. http://www.cs.tau.ac.il/∼flato/thesis.

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

[17] 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.

[18] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[19] D. H. Greene. Integer line segment intersection. Unpublished Manuscript.

[20] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

[21] L. Guibas and D. Marimont. Rounding arrangements dynamically. *Internat. J. Comput. Geom. Appl.*, 8:157–176, 1998.

[22] L. J. Guibas. Implementing geometric algorithms robustly. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, pages 24–28, May 1996.

[23] D. Halperin. Robust geometric computing in motion. *In Algorithmic and Computational Robotics: New Dimensions (WAFR 2000)*, pages 9–22, 2001. To appear in *Int. J. of Robotics Research.*

[24] D. Halperin and E. Packer. Iterated snap rounding. *Computational Geometry: Theory and Applications*, 23(2):209–225, 2002.

[25] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comput. Geom. Theory Appl.*, 10:273–287, 1998.

[26] I. Hanniel. The design and implementation of planar arrangements of curves in CGAL. M.Sc. thesis, Dept. Comput. Sci., Tel Aviv University, Tel Aviv, Israel, 2000. http://www.math.tau.ac.il/∼hanniel/thesis.ps.

[27] J. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13:199–214, 1999.

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

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

[30] K. Mehlhorn, S. Näher, C. Uhrig, and M. Seel. *The LEDA User Manual, Version 4.1.* Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, 2000.

[31] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artif. Intell.*, 37:377–401, 1988.

[32] V. J. Milenkovic. Shortest path geometric rounding. *Algorithmica*, 27(1):57–86, 2000.

[33] S. Raab. Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. M.Sc. thesis, Dept. Comput. Sci., Bar Ilan University, Ramat Gan, Israel, 1999. http://www.math.tau.ac.il/∼raab/master_thesis.ps.

[34] S. Raab. Controlled perturbation of arrangements of polyhedral surfaces with application to swept volumes. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1999.

[35] S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[36] R. Seidel. The nature and meaning of perturbations in geometric computing. *Discrete Comput. Geom.*, 19:1–17, 1998.

[37] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[38] B. Stroustrup. *The C++ Programming Language.* 3rd edition, 1997.

[39] K. Sugihara. On finite-precision representations of geometric objects. *J. Comput. Syst. Sci.*, 39:236–247, 1989.

[40] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[41] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40(1):2–18, 1990.

[42] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, Boca Raton, FL, 1997.