Degree-Driven Design of Geometric Algorithms for Point Location, Proximity, and Volume Calculation

David L. Millman

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2012

Approved by:

Jack Snoeyink

David P. Griesheimer

Ming C. Lin

Dinesh Manocha

Chee K. Yap

© 2012
David L. Millman
ALL RIGHTS RESERVED

Abstract

DAVID L. MILLMAN: Degree-Driven Design of Geometric Algorithms for Point Location, Proximity, and Volume Calculation (Under the direction of Jack Snoeyink)

Correct implementation of published geometric algorithms is surprisingly difficult. Geometric algorithms are often designed for Real-RAM, a computational model that provides arbitrary precision arithmetic operations at unit cost. Actual commodity hardware provides only finite precision and may result in arithmetic errors. While the errors may seem small, if ignored, they may cause incorrect branching, which may cause an implementation to reach an undefined state, produce erroneous output, or crash. In 1999 Liotta, Preparata and Tamassia proposed that in addition to considering the resources of time and space, an algorithm designer should also consider the arithmetic precision necessary to guarantee a correct implementation. They called this design technique *degree-driven algorithm design*. Designers who consider the time, space, and precision for a problem up-front arrive at new solutions, gain further insight, and find simpler representations. In this thesis, I show that degree-driven design supports the development of new and robust geometric algorithms.

I demonstrate this claim via several new algorithms. For n point sites on a $U \times U$ grid I consider three problems. First, I show how to compute the nearest neighbor transform in $O(U^2)$ expected time, $O(U^2)$ space, and double precision. Second, I show how to create a data structure in $O(n \log Un)$ expected time, $O(n)$ expected space, and triple precision that supports $O(\log n)$ time and double precision post-office queries. Third, I show how to compute the Gabriel graph in $O(n^2)$ time, $O(n^2)$ space and double precision. For computing volumes of CSG models, I describe a framework that uses a minimal set of predicates that use at most five-fold precision.

The framework is over 500x faster and two orders of magnitude more accurate than a Monte Carlo volume calculation algorithm.

To my mom, dad, and sister.

Acknowledgments

I thank my fiancée Dr. Brittany Terese Fasy for her never ending love and support. Whether down the street or on the other side of the world, her encouragement could be heard all over. Thank you Brit!

I thank my adviser Dr. Jack Snoeyink for all that he has taught me while at UNC. Thank you for always pushing me to think more abstractly, write more clearly, and present more engagingly. I also thank my other committee members: Dr. David P. Griesheimer, Dr. Ming Lin, Dr. Dinesh Manocha, and Dr. Chee K. Yap for their suggestions, discussions, and time committed to serving on my committee.

The work in the dissertation work would not have been possible without the support of Department of Energy, Bettis Atomic Power Laboratory, National Science Foundation, Google, Lime Connect, and UNC-Chapel Hill. Two years of this research were performed under appointment of the Rickover Fellowship program in Nuclear Engineering sponsored by the Naval reactor division of the US Department of Energy. I thank all involved with the fellowship program and the technical and administrative support staff of UNC.

I thank my co-authors: Dr. Vicente H. F. Batista, Dr. Timothy Chan, Steven Love Dr. Brian Nease, Matt O'Meara, Dr. Sylvain Pion, Dr. Johannes Singler, and Clarence R. Willis. In particular, I thank co-author Vishal Verma for participating in our productive (and sometimes not so productive) research discussions.

I thank my friends for their limitless support. I thank my future family Joseph, Terese, Joey, Dana, and Devon Fasy for welcoming me into their family. I thank Darrell Bethea, Robbie

Cochran, Srinivas Krishnan, Alana Libonati, and Catie Welsh for making me remember that the world isn't just grid points and bisectors. There is no better way to end the week than with a trip to Bandido's or a walk to get coffee. I thank Doug McNamara for cheering so loudly for me.

Finally, I thank my parents: Dr. Ronald and Merri Millman, and sister, Lisa Millman, for their love and encouragement. They have always believed in me. I could never have done this without them.

Table of Contents

Abstract . . . iii

List of Figures . . . xii

List of Tables . . . xiv

1 Introduction . . . 1

1.1 Thesis Statement and Contributions . . . 3

2 Background . . . 6

2.1 Floating Point Representation . . . 6

2.2 Definitions of Correctness . . . 7

2.3 Previous Results in Degree-Driven Algorithm Design . . . 12

3 Analyzing the Precision of Predicates, Constructions and Algorithms . . . 16

4 Degrees of Predicates and Constructions . . . 24

4.1 Constructions . . . 25

4.1.1 $\texttt{Line}(a, b)$ . . . 27

4.1.2 $\texttt{Plane}(a, b, c)$ . . . 28

4.1.3 $\texttt{Bisector}(a, b)$ . . . 30

4.1.4 $\texttt{LineIntersection}(\ell, h)$ . . . 31

4.1.5 `VoronoiVertex`(a, b, c) . . . 32

4.2 Predicates and Simple Algorithms . . . 33

4.2.1 `PointOrdering`(a, b) . . . 33

4.2.2 `Orientation`(a, b, q) . . . 34

4.2.3 `Closer`(a, b, q) . . . 35

4.2.4 `SideOfABisector`(B_{ab}, q) . . . 36

4.2.5 `OrderOnLine`(B_{ab}, B_{cd}, ℓ) . . . 36

4.2.6 `OrderOnLine`(g, h, ℓ) . . . 37

4.2.7 `InCircle`(a, b, c, q) . . . 37

4.2.8 `IntersectInterior`$(\overline{ac}, \overline{bd})$ . . . 38

4.3 Checking Preconditions . . . 39

4.3.1 Equal Homogeneous Coordinates . . . 40

4.3.2 Preconditions for Points . . . 40

4.3.3 Preconditions for Lines . . . 41

5 Irreducibility of Polynomials . . . 42

5.1 Notation . . . 42

5.2 Basket Weaving Technique for Proving Irreducibility . . . 43

5.3 Polynomials from Determinants of Matrices with Independent Variables . . . 44

5.4 `PointOrdering`(a, b) is Irreducible . . . 46

5.5 `VoronoiVertex`(a, b, c) Cannot be Simplified . . . 47

5.6 `Orientation`(a, b, q) is Irreducible . . . 48

5.7 `SideOfABisector`(B_{ab}, q) and `Closer`(a, b, q) are Irreducible . . . 51

5.8 `OrderOnLine`(B_{ab}, B_{cd}, ℓ) is Irreducible . . . 52

5.9 `OrderOnLine`(l, h, ℓ) is Irreducible . . . 53

5.10 `InCircle`(a, b, c, q) is Irreducible . . . 53

6 Gabriel Graph . . . 56

6.1 Introduction . . . 56

6.2 Arrangements of Dual Lines . . . 57

6.3 Algorithm Description . . . 58

6.4 Conclusion . . . 62

7 Point Location . . . 63

7.1 Introduction . . . 63

7.2 Definitions and Notation . . . 65

7.2.1 The Voronoi Diagram and Grids . . . 65

7.2.2 Trapezoidation and Point Location . . . 68

7.3 Computing Point Location Structures with Triple Precision . . . 70

7.3.1 The Reduced-Precision Voronoi Diagram . . . 71

7.3.2 Constructing the Reduced-Precision Voronoi Diagram . . . 72

7.4 Computing Point Location Structures with Double Precision . . . 79

7.4.1 Definitions and Notation . . . 80

7.4.2 Constructing the D2-Voronoi Diagram . . . 84

7.5 Conclusion . . . 90

8 Nearest Neighbor Transform . . . 91

8.1 Preliminaries . . . 92

8.1.1 Previous work . . . 92

8.1.2 Definitions . . . 93

8.1.3 Problem transformations . . . 94

8.1.4 Geometric primitives . . . 95

8.2 Algorithm Description . . . 97

8.2.1 Discrete Upper Envelope 97
8.2.2 NNTrans Algorithm 102
8.3 Experiments 104
8.4 Conclusions 108

9 Volume Calculation 109

9.1 Model Representation 113
9.1.1 Primitives: Signed Quadric Surfaces 113
9.1.2 Component Hierarchy: Boolean Formulae 114
9.1.3 Problem Statement 116
9.2 Divide and Conquer Using Predicates 117
9.2.1 Surface/Axis-Aligned Box Classification 117
9.2.2 Component/Box Classification and Restriction 123
9.3 Volume Estimators 124
9.3.1 Error bounds 126
9.4 Experiments 127
9.5 Conclusions 130

10 Using Degree-driven Algorithm Design to Achieve EGC 132

10.1 EGC Technique 1: Static Analysis 133
10.2 EGC Technique 2: Software Implementations 133
10.3 EGC Technique 3: Arithmetic Filters 134
10.4 EGC Technique 4: Adaptive Evaluation 135
10.5 Conclusion 137

A Sign of a Sum 138

Bibliography 140

List of Figures

3.1 Intersection coordinate not lending itself to a floating point representation. 16

4.1 `Line`(a, b) produces the line though a and b. 27

4.2 `Plane`(a, b, c) produces the plane though a, b, and c. 28

4.3 `Bisector`(a, b) produces the perpendicular bisector $\overline{ab}$. 30

4.4 `LineIntersection`(ℓ, h) produces the intersection point of ℓ and h. 31

4.5 `VoronoiVertex`(a, b, c) produces the point equidistant to a, b, and c. 32

4.6 The point a precedes the point b. 33

4.7 The path from a to b to q forms a right turn. 34

4.8 The query point q is on the same side of the B_{ab} as b. 36

4.9 The bisector B_{ab} is above the bisector B_{cd} on the line ℓ. 36

4.10 The intersection of g and ℓ is left of the intersection of h and ℓ. 37

4.11 The query point q in inside the circumcircle of a, b and c. 37

4.12 The segments $\overline{ac}$ and $\overline{bd}$ intersect. 38

6.1 Gabriel graph of sites. 57

6.2 If site s_j kills (s, s_i) from the left, it also kills all edges right of s_i and left of s_j. . . 59

7.1 Trapezoidation of a small Voronoi diagram and a piece of the trapezoid graph. . . . 68

7.2 Voronoi vertices in a grid cell contracting to two rp-vertices. 72

7.3 Constructing the rp-Voronoi diagram. 73

7.5 The four cases per wall for bisector tracing. 75

7.4 A bisector entering a grid cell from the south. 75

7.6 Three views of intersecting bisectors. 80

7.7 Voronoi polygons, proxy trapezoidation, and Voronoi trapezoidation. 81
7.8 Illustration of `GCDWalk`. 85
7.9 New site s_i in conflict with Voronoi trapezoid τ. 87

8.1 Transforming sites in S to lines $L_Y(S)$. 95
8.2 Adding a set of lines to a DUE. 98
8.3 Constructing and updating the possible lists. 102
8.4 The three steps of the NNTrans algorithm. 103
8.5 Time-per-pixel box plots of 100 runs of `Usq` on varying densities and grid sizes. . . 106
8.6 Time-per-pixel for four implementations varying grid sizes and densities. 107
8.7 Time-per-pixel for four implementations on images from the MPEG7 data set. . . . 108

9.1 Hierarchy N_p-N-N_c, with comp. $C(N)$ striped purple and blue. 115
9.2 Intersection of a quadric with an axis-aligned plane. 117
9.3 Renderings of three test models: Cube, DrillCyl, and cPiped12. 127

List of Tables

4.1 Summary of primitive types. 26

4.2 Summary of constructions. 26

4.3 Summary of predicates. 27

9.1 Timings & num boxes evaluated when varying integrators, models, & tolerances. . 128

9.2 Timings and percentages of unit volume integrated by the different integrators. . . 131

Chapter 1

Introduction

The representation of numbers makes a difference in the ease of doing correct calculation. In *The Elements*, Euclid spoke of numbers as geometric measures and observed that the $\sqrt{2}$, and other numbers, could not be represented as a fractions of integer measures—that it was irrational. With computers we face the opposite question: how to represent geometry with numbers. Rather than using geometry to represent a number, we may, for example, want to calculate the intersection point of two lines and store the coordinates in memory. We then need to know how many bits of memory are sufficient to store the point accurately. Computers have brought new questions about designing and analyzing algorithms for solving problems that can be stated geometrically; these algorithms we call *geometric algorithms.*

The standard model of computation for analyzing geometric algorithms is the Real-RAM [109], which supports constant-time arithmetic operations on real numbers stored in a memory that can be indexed by integers. (Many bit operations or floor/ceiling are usually not supported directly, but must be implemented in software and so take non-constant time.) The Real-RAM has three unbounded quantities: the number of steps that an algorithm can take, the number of memory cells that it can use, and the size of the set of numbers that can be represented in a given cell. Because of increasing processor speeds, memory sizes, and bits to represent numbers it is appropriate to not place arbitrary limits on the Real-RAM. To ensure algorithms designed for the Real-RAM model

are practical, researchers seek to minimize the asymptotic growth of processor time and memory space as the problem sizes grow, and (with some famous exceptions) tacitly agree not to use the power afforded by unbounded precision in each cell. In *precision analysis* the number of bits used to represent numbers is analyzed and minimized as well. In this thesis I investigate designing geometric algorithms by considering all three: time, space, and precision.

My main study is predicates, which are tests that control branching in geometric algorithms. Segregating numerical computation on input coordinates into predicates not only enables precision analysis, but also increases the chances that an algorithm will be correctly implemented, since predicates can be separately tested, debugged, and put into libraries for reuse. In this thesis, I restrict my attention to *predicates* that test signs of polynomials in the input coordinates, and *constructions* that produce objects defined by rational polynomials in the input coordinates. Liotta, Preparata, and Tamassia suggested that we can bound the precision required by a geometric algorithm by analyzing the degree of predicates. Thus, the *degree* of my algorithms is the degree of their highest degree predicate.

In most efficient geometric algorithms it is common for predicates to take many times the precision of the input coordinates. To evaluate a degree 5 predicate on 32-bit input coordinates would require approximately 160 bits (about five times the number of bits for representing the input coordinates). As most machines only support 64-bit arithmetic, unless higher precision is simulated in software, for some inputs the test will be wrong.

We can use precision as a guideline for designing geometric algorithms, which Liotta, Preparata, and Tamassia call *degree-driven algorithm design*. Degree-driven algorithm design makes analysis easy to carry out yet general enough to have an impact. One can design an algorithm, determine how much precision it requires, implement the algorithm, and then specify a range of coordinate values for which the implementation guarantees a correct result.

1.1 Thesis Statement and Contributions

In this thesis, I show that degree-driven analysis supports the development of new and robust geometric algorithms. The contributions are summarized in the remainder of this section.

One of the early developments in computational geometry was the identification of predicates for various geometric tests. The design and implementation of algorithms were based on a limited set of predicates, rather than on numerical coordinates. Thus, we consider representations of primitives (such as points and lines) and predicates on these primitives. Chapter 3 describes how precision will be analyzed in this thesis and Chapter 4 derives and upper bounds the precision of many common predicates.

The easiest way to reduce the precision of a predicate is to factor its polynomial. It is often claimed that the polynomials in common predicates cannot be factored but is rarely shown and references are rarely provided. Chapter 5 shows that the polynomials from Chapter 4 cannot be factored and identifies some helpful techniques for showing that a polynomial cannot be factored.

The next four chapters support the thesis statement by using degree-driven analysis to give new and robust algorithms for the Gabriel graph, post-office queries, nearest neighbor transform and volume calculation of CSG models. Let me briefly define these problems and state the main results.

Gabriel Graph. First, given a set S with n point sites, an edge (s_i, s_j) with $s_i, s_j \in S$ is in the *Gabriel graph of* S if the edge maintains the Gabriel property, that is, the closed disk with diameter $\overline{s_i s_j}$ contains no points of S besides s_i and s_j. Testing if an edge has the Gabriel property can be degree 2. Thus, there is a simple brute force degree 2 algorithm for computing the Gabriel graph: construct all $O(n^2)$ edges, and for each edge e check all n sites to see if e has the Gabriel property. The brute force algorithm is $O(n^3)$ and degree 2. Chapter 7 shows how to compute the Gabriel graph in $O(n^2)$ time, $O(n^2)$ space, and degree 2. Along the way, we see that we can compute the trapezoidation of the arrangement of n lines in $O(n^2)$ time, $O(n^2)$ space, and degree 2.

Post-Office Queries. Second, given a query point q and a set S with n point sites, a *post-office query* returns the site of S closest to q. For $s_i, s_j \in S$, testing if q is closer to s_i or s_j is degree 2. Thus, there is a simple brute force linear time degree 2 algorithm for queries, test q against all sites. More efficient algorithms preprocess the point set to achieve $O(\log n)$ time queries. The standard algorithm [31] computes the Voronoi diagram of S, which takes $O(n \log n)$ time and degree at least 4; computes the trapezoid map of the Voronoi edges, which takes time $O(n \log n)$ and degree 5; and queries the map, which naively takes $O(\log n)$ time and degree 6 (however, it is easy to reduce such queries to degree 3). Chapter 7 presents a *reduced precision Voronoi diagram* that for n sites on a $U \times U$ grid, can be computed in $O(n \log Un)$ expected time, $O(n)$ expected space, and degree 3. The reduced precision Voronoi diagram supports $O(\log n)$ time and degree 2 post-office queries. The chapter concludes with the description of a degree 2 Voronoi diagram that can be constructed with degree 2 and supports $O(\log n)$ degree 2 post-office queries.

Nearest Neighbor Transform. Third, given a set of n sites S on a $U \times U$ grid of the pixels of an image, the *nearest neighbor transform* assigns to each pixel the closest site of S. Testing if a pixel is closer to s_i or s_j, with $s_i, s_j \in S$, is degree 2. Thus, there is a simple brute force $O(nU^2)$ time degree 2 algorithm for queries, which tests each pixel against all sites. More sophisticated algorithms have a running time independent of n. Chapter 8 shows how to compute the nearest neighbor transform with a randomized algorithm in $O(U^2)$ expected time, $O(U^2)$ space, and degree 2. A simpler $O(U^2 \log U)$ time, $O(U^2)$ space, and degree 2 deterministic algorithm is also presented. We compare prototype implementations of both algorithms with MATLAB's [91] compiled implementation of an $O(U^2)$ time, $O(U^2)$ space, and degree 3 algorithm. The experiments show that the clock time of the deterministic algorithm is about 3x faster than the randomized algorithm and that the clock times of the randomized and deterministic algorithms are 5–7x and 15–23x faster than MATLAB's.

Volume Calcualtion of CSG Models. Fourth, not all problems have low degree solutions. We should use degree-driven analysis to drive better decisions on selecting predicates, which informs algorithm design. Chapter 9 uses degree-driven algorithm design to improve accuracy and speed in computing the volumes of CSG models. For the types of models considered, testing if a point is inside a model M is degree 3. Thus, a simple Monte Carlo algorithm takes as input N sample points in B and the bounding box of M. Let h be the number of samples falling in M. The standard Monte Carlo algorithm approximates the volume as $\mathrm{Vol}(B) * h/N$. In practice, N needs to be very large. For example, when B is a unit cube N must be over 1.4 billion samples for a 10^{-4} relative error, which limits performance. Instead, one could intersect the surfaces defining the models, but computing the curves resulting from surface intersections without sophisticated software libraries is error prone and creates more structure than is necessary for calculating volumes. Chapter 9 describes a framework for computing volumes using a minimal set of predicates that are at most degree 5. My framework computes only the intersection curves of surfaces and axis aligned planes yet is over 500x faster and two orders of magnitude more accurate than the Monte Carlo algorithm.

Chapter 2

Background

A computer supports a subset of the reals called floating point numbers. In Section 2.1, I describe the IEEE standard for representing floating point numbers. Since a computer cannot represent all reals, designers (and implementers) of geometric algorithms may have different definitions of what it means for an algorithm to be correct (or have a correct implementation). In Section 2.2, I describe the two main paradigms to correctness, Topology-Oriented Computation [127] and Exact Geometric Computation [140]. One way to achieve the goals of exact geometric computation is to reduce the amount of precision required by an algorithm. Liotta, Preparata, and Tamassia [89] suggested a guideline called, degree-driven algorithm design for upper bounding the precision required by an algorithm. In Section 2.3, I survey the previous work in degree-driven algorithm design.

2.1 Floating Point Representation

In the IEEE 754 standard for binary floating point format a number is represented as a *sign* s, *exponent* e, and *mantissa* m[1]. The sign is 1 bit, and the exponent and mantissa are some fixed number of bits (dependent on representation and precision). In a *normalized* floating point representation, the highest order bit of the mantissa is assumed to be 1 and so it does not need to be

[1]The base is sometimes called the *radix*, the mantissa is sometimes called the *significand* and the exponent is sometimes called the *characteristic*.

explicitly represented. Often a *bias* is added to the exponent to simplify comparisons. The value of a normalized floating point number with bias E is $(-1)^s \times 1.m \times 2^{e-E}$.

This familiar representation was not always so common. In a 1998 interview[120], Kahan recalls some of the peculiarities of early implementations of floating point arithmetic. For example, two floating point values could test as not-equal, yet their difference could be zero. Kahan, along with Coonen and Stone proposed the IEEE Standard 754-1985, which is the basis for the "float" and "double" types in many high level programming languages such as C, C++, C#, and Java. In these languages, "float" is implemented by the IEEE 754-2008 single precision floating point format called *binary32* (in IEEE 754-1985 it was called *single*) with 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa. In fact, binary32 specifies a floating point number as $(-1)^s \times (1 + \Sigma_{i=1}^{23} b_i 2^{-i}) \times 2^{e-127}$.

The set of *binary32* numbers is finite, so only a subset of $\mathbb{R}$ can be represented exactly. The rest of the numbers must be approximated. For example, even a simple rational such as $1/3$ must be rounded to a number near $0.\overline{3}$. For arithmetic operations, the result of a floating point operation on floating point input is the same as the result of the true operation on the floating point input rounded to one unit in the last place. IEEE standard allows for multiple rounding modes. Of particular importance are "round up" and "round down", which round toward positive and negative infinity respectively. The two rounding modes are the basis for arithmetic filters, which are discussed further in Chapter 10.

2.2 Definitions of Correctness

Computational geometry textbooks and papers usually prove that an algorithm is correct in the Real-RAM model of computation [109]. In this model, arithmetic operations are exact and take unit time. In implementations on current technology, however, arithmetic operations are not exact, when too few arithmetic bits are available, *numerical errors* occur. Numerical errors can cause

predicates to be evaluated incorrectly and cause implementations to fail [44, 63].

One could naively rely on machine precision and replace comparisons against zero with comparisons against an ϵ-*tolerance*. Kettner *et al.* [75] investigate the errors introduced with floating point arithmetic using simple predicates, such as `Orientation`, and generate simple examples where an incremental convex hull algorithm fails. Their numerical experiments show that the errors of a floating point based orientation predicate have a complex and non-intuitive structure. Kettner also points out that ϵ-tolerances do not fix the problems raised by their investigation; by rounding non-zero values to zero, it only enlarges the complex error structure.

Instead, one could add steps to ensure correct results. Yap [141] calls the study of algorithms with running time dependent on the precision of the input or output, *numerical computational geometry*. Below I outline what I believe to be the two main paradigms of numerical computational geometry, *Topology-Oriented Computation* and *Exact Geometric Computation*.

Topology-Oriented Computation (TOC). Geometric algorithms often aim to compute structures for which topological properties may hold even if the geometry is perturbed. For example, the Voronoi diagram is a connected planar graph, that happens to have convex faces in its geometric embedding. Suighara [127, 128, 131] proposes that algorithms be designed to guarantee chosen topological properties (e.g., connectedness or planarity), even if the geometric embedding contains perturbations or numerical error (e.g., self-intersections or non-convexities). His Topology-Oriented Computation paradigm seeks to ensure that, at a minimum, geometric algorithms should fail gracefully as they reach the limit of the available precision.

Suighara defines TOC more formally [128], let P be a geometric problem in which X is the set of inputs to P and Y is the set of outputs. Let f be an algorithm for solving P and $\widetilde{f}$ be an implementation of f. If $\widetilde{f}$ is implemented with finite precision arithmetic, for $x \in X$ the output $\widetilde{f}(x)$ may not be the same as $f(x)$. Let $f_C(x)$ and $f_N(x)$ be the combinatorial and numerical outputs of f, respectively. Similarly, $\widetilde{f}_C(x)$ and $\widetilde{f}_N(x)$ are the combinatorial and numerical outputs

of the implementation $\widetilde{f}$.

For a concrete example, let P be the problem of computing the Voronoi diagram of sites with double precision floating point coordinates in 2D. An input x is n distinct sites with double precision floating point coordinates. The output is the Voronoi diagram of the n sites. Let f be Fortune's algorithm [43] and $\widetilde{f}$ be Brubeck's [17] implementation of Fortune's algorithm. For an input $x \in X$, $f_C(x)$ is a planar graph, and $f_N(x)$ the coordinates of the graph's vertices and the directions of the infinite edges. Brubeck reports that his implementation does not terminate for all input. But for all x for which $\widetilde{f}(x)$ terminates, $\widetilde{f}_C(x)$ and $\widetilde{f}_N(x)$ are the graph and the vertex coordinates reported by $\widetilde{f}$, respectively.

Suighara [128] defines the implementation $\widetilde{f}$ to be *robust* if $\widetilde{f}$ is defined for any $x \in X$. That is, for any $x \in X$, $\widetilde{f}(x)$ does not crash or go into an infinite loop. Note that the output $\widetilde{f}(x)$ may not be in Y. The implementation $\widetilde{f}$ is *topologically consistent* if for all $x \in X$, $\widetilde{f}_C(x) \in f_C(X)$. Note that this does not mean that $\widetilde{f}_C(x) = f_C(x)$ or even that $\widetilde{f}_C(x)$ is in the neighborhood of $f_C(x)$; simply that there is some input $y \in X$ such that $f(y) = \widetilde{f}(x)$.

Sugihara with others proposed robust and topologically-consistent algorithms for many problems including, solid modeling with planes [129], divide-and-conquer construction of Voronoi diagrams of points in 2D and 3D[107, 65, 66], incremental construction of the Voronoi diagrams of polygons [64], and gift-wrapping and divide-and-conquer constructions of convex hull in 3D [126, 102, 103]. TOC can create robust implementations. Sugihara and Iri [130] used TOC to create an algorithm for building the first Voronoi diagram of over a million sites with single precision arithmetic. (It should be noted that Isenburg *et al.* [67] used EGC, described in the next section, to build the first Voronoi diagram of over a billion sites with single precision input.) Held and Huber [56] used TOC to create the first floating point implementation for computing the Voronoi diagram of circular arcs (without discretizing or approximating the arcs).

A weakness of the TOC is that even for topologically consistent algorithms the output may be

in no way similar to the correct output. Consider the following topologically consistent algorithm for computing the Voronoi diagram. As a preprocess, select a point set x, and by hand compute the Voronoi vertices and edges for x. At run time, for any input y return the Voronoi vertices and edges for x. Moreover, few of the classic geometric algorithms designed using the Real-RAM model are robust, let alone topologically consistent. Thus, many of the classical geometric algorithms are not correct under the definitions of TOC.

Exact Geometric Computation (EGC). Geometric computing is part combinatorial, for example traversing an embedded graph, and part numeric, for example determining if two segments intersect. Yap [140] observes that the interplay between the numerical and the combinatorial is what causes geometric algorithms to be difficult to implement. His Exact Geometric Computation paradigm dictates that an algorithm's control flow should be independent of the machine on which the implementation is run; in particular, it should be the same as if the algorithm was implemented with real arithmetic.

The strength of EGC is that it cleanly separates the algorithm design process from the number types used in an implementation. By abstracting the number type away from the programmer, debugging and implementation become easier. Indeed, EGC's strength can be seen in its acceptance in software development. Well-known open-source software libraries CGAL [2] and LEDA [19] both have geometric kernels that support EGC.

The weakness of EGC is pointed out originally by Yap [140] and revisited by Held and Mann [57]. The easiest way to implement EGC is to use software to simulate rational or real arithmetic. When done blindly, simulating with software can be slow. Karasick *et al.* report that directly using rational arithmetic caused a $10,000$x slowdown [72]. However, with the knowledge that the important computation is the signs of polynomials and a bit of work, the slowdown can be reduced to below 10x. In Chapter 10, I outline the major techniques used for implementing EGC.

EGC does not specify how to output the coordinates of a construction, however, sometimes

approximate coordinates are needed. Without care, the approximate results may introduce fatal errors. For example, consider computing the convex hull of a set of segment intersection points and outputting its polygonal boundary as a list of vertex coordinates. The rounded coordinates may cause the boundary of the convex hull to be non-convex. *Geometric rounding* [32, 49, 51, 55, 58, 60] investigates how to round a construction's coordinates while maintaining the important geometric and topological properties.

Perhaps more seriously is that if number types are abstracted too early in the design process, an algorithm may perform unnecessary constructions. Let's see an example that takes a bit to explain. A *cascading construction* is a phenomenon where constructions are used to create more constructions[2]. A common industrial example of cascading constructions occurs in polyhedral modeling systems. Consider a system (similar to Maya [1]) in which the polyhedra are represented by intersecting a set of half-planes. A common operation is to output the boundary of the polyhedra. Often the boundary is represented as: a set of vertices, defined by coordinates; a set of polygonal faces, defined by an ordered list of pointers to vertices; and adjacencies between faces.

It is common to build the boundary incrementally. Assume that we have constructed the boundary of P_{i-1} defined by the first $i-1$ planes. When adding plane p_i, find all vertices outside of the constraints implied by p_i. Throw away all faces outside of p_i and split all faces intersected by p_i. Splitting faces introduces new vertices, update the vertices and the topology of the polyhedra. Often, in implementation, due to numerical errors, the resulting set of vertices defining a polygon are not coplanar as they would be in Real-RAM. Thus, a face may not have a clearly defined normal.

Suighara and Iri [129] observed that the cascading constructions are unnecessary, and are in fact a source of error. They observed that vertices do not need to be represented by coordinates. Instead, they can be represented by triples of planes defining them. Burnstein and Fussell [8] used

---

[2] The classic example of a cascading construction is the *Pentagon* problem, proposed by Dobkin and Silver [35], where a smaller inverted pentagon is constructed by intersecting the pentagon's diagonals.

this observation and some additional predicate evaluation techniques to build a modeling system that is faster than CGAL [2], which uses exact arithmetic, and is more stable than Maya, which uses floating point arithmetic.

Burnstein and Fussell's implementation is fast and stable because they avoid high precision tests used by other implementations. In particular, they use low precision predicates to reduce the need for simulating real arithmetic. Thus, when designing an algorithm (or implementing a pre-existing algorithm) it is worth paying attention to how much precision is used.

This thesis explores Liotta, Preparata, and Tamassia's proposed *degree-driven algorithm design* [89] guideline. This guideline seeks to optimize (and balance) running time, space, and precision simultaneously. Once we know how much precision is required by an algorithm, creating an efficient implementation that follows EGC becomes easier. The EGC implementation techniques described in Chapter 10 are still relevant. But degree-driven algorithm design allows a programmer to make more informed decisions on when the techniques are needed.

2.3 Previous Results in Degree-Driven Algorithm Design

The most complete study of degree-driven design was carried out for segment intersection problems [10, 11, 21, 90]. Boissonnat and Preparata [10] describe three problems for a set of n line segments defined by their endpoints with single precision coordinates:

P1 report all pairs of intersecting segments;

P2 construct the arrangement of the segments;

P3 construct the trapezoid graph of the segments.

They also describe variants in which the input segments are divided into two disjoint sets, and each segment is colored with the set in which they belong. The colors are often red and blue, so the colored variants are often called red-blue intersection problems.

For n segments we can have $\Theta(n^2)$ intersections. A trivial algorithm with worst-case optimal running time is to check all pairs using a double precision segment intersection test. However, more interesting algorithms consider the number of intersections k. Boissonnat and Preparata show that a degree 3 variant of the degree 5, $O((n+k)\log n)$ time Bentley-Ottmann sweep line algorithm [7] solves P1 in $O((n+k)\log n)$ time. They also show that P1 for red and blue segments with only bichromatic intersections can be solved with degree 2.

Chan [21], and Boissonnat and Snoeyink [11], investigate degree-driven algorithm design by using a restricted set of predicates and abstract the above/below test on curve segments (whose degree is dependent on the complexity of the carrier of the curve). Boissonnat and Snoeyink consider segment intersection problems on a set of pseudo-segments, which are x-monotone segments such that any pair have at most one point in their intersection. They show that when limited to the three tests: ordering of endpoints; checking if an endpoint, in the vertical slab defined by a segment, is above or below the segment; and testing if two curves intersect; P1 is lower bounded by $\Omega(n\sqrt{k})$. They also show that for segments, Balaban's [6] degree 3, $O(n \log n + k)$, algorithm can be modified to degree 2 with a slight loss of efficiency, running in $O(n \log^2 n + k \log n)$ time[3]. Finally, they show that even with restricted predicates, the red-blue intersection problem (with pseudo-segments) can be solved in $O(n \log n + k)$ time.

As touched upon in almost all the literature on degree-driven algorithms for segment intersections, but most clearly stated by Mantler and Snoeyink [90], P2 requires four-fold precision, and P3 requires five-fold precision. Mantler and Snoeyink consider P2 for red and blue segments. They show that the topology of arrangement (but not the coordinates of the intersections) can be computed using a sweep line algorithm that runs in optimal $O(n \log n + k)$ time using $O(n)$ space and degree 2.

---

[3]Boissonnat and Snoeyink's degree 2 version of Balaban's algorithm appears to be the first example of a time-precision trade off.

Researchers also consider point location queries. Given a set of n point sites where the coordinates of the sites and query points are b-bit integers. Let $U = 2^b$, and the set of representable points from which sites and queries are picked is a $U \times U$ grid, sometimes referred to as a universe of size U [25].

Liotta, Preparata and Tamassia [89] describe a degree 6 algorithm for rounding a trapezoidation of the Voronoi diagram to a degree 1 data structure capable of reporting nearest neighbor queries in $O(\log n)$ time with degree 2. Unfortunately, they still have to construct the trapezoidation. Millman and Snoeyink [99] consider how to construct Liotta's structure with a degree 3 randomized algorithm in a size U universe in time $O(n \log Un)$. We will see this algorithm in Chapter 7. Using a different approach, Millman and Snoeyink [100] describe a degree 2 construction of a degree 1 data structure supporting degree 2 logarithmic time point location queries. We further explore this construction in Chapter 7.

To compute the nearest neighbor for all query points in a universe of size U with degree 2, we could simply test all query points against all sites to achieve an $O(nU^2)$ algorithm. Millman *et al.* [98] show how to compute all nearest neighbor queries using degree 2 predicates with an expected time $O(U^2)$ algorithm, which we encounter in Chapter 8.

The additively weighted Voronoi diagram is a generalization off the Voronoi diagram in which a site s is defined by a center c and a weight w and the distance between a point p in the plane and s is $\|p - c\| - w$. Karavelas and Yvinec [73] proposed a degree 14 algorithm for computing the additively weighted Voronoi diagram. In previous work [96], I showed that the predicates presented by Karavelas and Yvinec could be simplified to degree 6. The new predicates, implemented in CGAL [2], resulted in a 39–66% speed up in predicate evaluation and a 10-20% reduction in the number of calls to software arithmetic for nearly degenerate inputs.

The Gabriel graph of a size n point set S is an embedded graph. The vertices are the points of S and the edges e_i have the property that the circle centered at the midpoint of e_i with diameter length

of e_i contains no other points of S. We can compute the Gabriel graph in $O(n^3)$ time, and degree 2 by comparing all triples. The classic result of Matula and Sokal [92] shows how to construct the Gabriel graph from the Delaunay triangulation in $O(n)$ time using the observation that a Delaunay edge is in the Gabriel graph if and only if it intersects the Voronoi edge to which it is the dual. Liotta [88] shows that Matula and Sokal's algorithm is degree 6, but provides a degree 2 algorithm accomplishing the same result. However, Liotta's algorithm must first use degree 4 to compute the Delaunay triangulation. Millman and Verma [101] showed how to compute the Gabriel graph directly, albeit slowly, in $O(n^2)$ time and degree 2. We will see this algorithm in Chapter 6.

Chapter 3

Analyzing the Precision of Predicates, Constructions and Algorithms

This thesis explores designing geometric algorithms and seeks to optimize the running time, space and precision. Analysis of algorithms in terms of time and space is commonly studied in a standard algorithms classes; see [30]. Analyzing the precision of an algorithm, however, is not so well known. Thus, the purpose of this chapter is to define terms that are central to the thesis and familiarize the reader with guidelines for upper bounding the precision required by an algorithm. I illustrate the important definitions and analysis on the example problem:

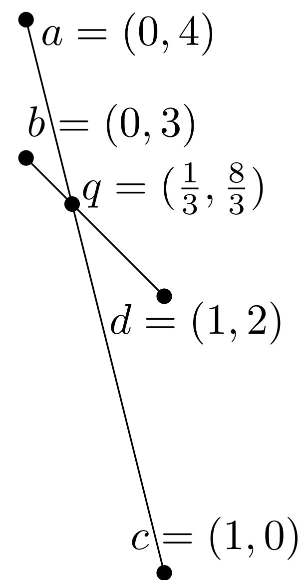


Figure 3.1: Two segments with an intersection coordinate that does not lend itself to a floating point representation.

DoSegsIntersect Given two, two-dimensional line-segments, $\overline{ac}$ and $\overline{bd}$, defined by their endpoints, determine if the interior of the segments intersect. Assume that no three endpoints are collinear.

We assume that no three points are collinear so that we may avoid unnecessary details that detract from the purpose of this chapter. Full details for intersecting segments are given in Chapter 4 where we remove the assumption.

In this thesis, I assume that the inputs to a geometric algorithm are a set of numeric coordinates scaled to b-bit integers and combinatorial relationships between the coordinates. Sometimes, I refer to the precision of the input coordinates as *degree 1* or *single precision*. In *DoSegsIntersect*, the numerical coordinates are the x and y values of the segment endpoints, and the combinatorial relationships are the pairing of values into points and the pairing of points into segments.

For $U = 2^b$, it is sometimes helpful to consider the coordinates as integers in $\{0, \ldots, U-1\}$. In such a case, when the input are points are planar, we can imagine them as laying on a $U \times U$ integer grid which we call $\mathbb{U}$. We assume that the input is scaled to an integer grid because as Chan and Pătraşcu [25] point out, the floating point plane is the union of integer grids with different scalings around the origin. The problems in this thesis are translation independent. Thus, the ability to have more precision for input near the origin is not particularly helpful.

A *predicate* is the sign of a multivariate polynomial with variables from the input coordinates. A positive, negative, or zero result can be interpreted geometrically. Often, the interpretation is used in algorithms to make branching decisions. Consider the predicate:

$\texttt{Orientation}(a, b, c)$**:** Given single precision coordinate values of 2D points a, b, and c,

$$\texttt{Orientation}(a, b, c) = \operatorname{sign}(b_x c_y - b_x a_y - a_x c_y - c_x b_y + c_x a_y + a_x b_y). \tag{3.1}$$

The predicate's positive, negative, or zero result has the geometric interpretation that the path from a to b to c makes a left turn, makes a right turn, or a, b and c are collinear, respectively. For the segments shown in Figure 3.1, $\texttt{Orientation}(a, c, d)$ is a left hand turn, $\texttt{Orientation}(a, c, b)$ is a right hand turn, and $\texttt{Orientation}(a, c, q)$, with q not rounded, is collinear.

The *degree of a predicate* is the degree of the polynomial. Thus, `Orientation` is degree 2. *Degree notation* is a helpful bookkeeping device for analyzing the degree of polynomials where we represent a degree k polynomial as $\textcircled{k}$. When it is clear from the context we may drop the sign function. Thus, the predicate $\texttt{Orientation}(a, c, b) = \textcircled{2}$.

Boissonnat and Preparata [10], define the degree of a predicate as the maximum degree of its irreducible factors. Even though I show that most of the predicates used in this thesis are irreducible, I omit the irreducibility condition. As we will see in Chapter 5, showing that a multivariate polynomial is irreducible is cumbersome (and not necessary for proving upper bounds). I hope that by omitting the irreducibility condition and simplifying precision analysis, degree-driven algorithm design can become of interest to a larger audience.

By removing the irreducibility condition the difference in degree may be substantial. For example, for variables $X = \{x_1, \ldots, x_n\}$, the polynomial formed by expanding the determinant of the Vandermonde matrix

$$V_1(X) = \begin{vmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{vmatrix}$$

has polynomial degree $n(n-1)/2$. The polynomial can be rewritten as a product of linear factors [136], $V_2(X) = \prod_{1 \le i < j \le n}(x_j - x_i)$. This factorization means that the predicate, $\mathrm{sign}(V_2(X))$, is degree 1. If we didn't know of the factorization, however, the predicate $\mathrm{sign}(V_1(X))$ would be degree $n(n-1)/2$.

The degree of a predicate serves as the basis for analyzing precision. Before proceeding, it is worthwhile to discuss the relationship between the degree and precision. Consider evaluating a predicate. The variables are from the input, thus they can be represented by b-bit integers. Since,

in this thesis, the coefficients of monomials in a predicate are small constants (usually 2 or 4), a monomial of degree k can be evaluated with $bk+O(1)$ bits. Moreover, in this thesis, the number of variables in a predicate is small, say s. For s variables, the number of monomials in a polynomial is $O(s^k)$ and a polynomial of degree k can be evaluated with $bk + O(1) + k \log_2 s$ bits. Thus, the calculation of a predicate can be carried out in $k(b + O(1))$ bits. The degree k can be thought of as the leading term, determining the required precision. Just as we ignore constants in time and space analysis, we ignore carry bits from addition and coefficients in this analysis. In Chapter 10 we discuss implementation techniques that can sometimes be used to reduce the number of bits.

A *construction* is an operation that produces a new object from the coordinate values of the input. Consider the construction producing the intersection point q of lines $\overleftrightarrow{ac}$ and $\overleftrightarrow{bd}$.

`Intersect`(a, c, b, d) takes the single precision coordinate values of four two-dimensional points a, c, b, and d and produces the coordinates of the intersection point of non-parallel lines $\overleftrightarrow{ac}$ and $\overleftrightarrow{bd}$. The construction returns the point q, whose coordinates are:

$$q_x = \frac{\begin{vmatrix} a_x c_y - c_x a_y & a_x - c_x \\ b_x d_y - d_x b_y & b_x - d_x \end{vmatrix}}{\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - d_x & b_y - d_y \end{vmatrix}}, \qquad q_y = \frac{\begin{vmatrix} a_x c_y - c_x a_y & a_y - c_y \\ b_x d_y - d_x b_y & b_y - d_y \end{vmatrix}}{\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - d_x & b_y - d_y \end{vmatrix}}. \tag{3.2}$$

The *degree of a construction* is the highest degree of the polynomials used to represent the output of the construction. Thus, `Intersect` produces coordinates of degree 3 over 2, written in degree notation as ③ / ② . Note that the degree of the construction may be the ratio of two degrees. We use degree to upper bound the number of bits required for evaluating predicates. If a predicate compares the rational coordinates of a construction, for analysis, the numerator and denominator must be kept separate so that we may clear fractions.

We can extend degree notation to the operations $\{+, \times\}$. Let f and g be polynomials with coefficients in $\mathbb{R}$. One can show that

$$\deg(fg) = \deg(f) + \deg(g) \qquad \text{and} \qquad \deg(f) + \deg(g) \leq \max(\deg(f), \deg(g)).$$

The equality $\deg(f) + \deg(g) = \max(\deg(f), \deg(g))$ holds when adding f and g does not cause all highest degree monomials to cancel. In this thesis, degree notation may only be used with operations when all highest degree monomials do not to cancel. Define the operations $\{+, \times\}$ for $\textcircled{a}$, $\textcircled{b}$, $\textcircled{c}$, and $\textcircled{d}$ as:

$$\begin{aligned}
\textcircled{a} + \textcircled{b} &= \boxed{\max(a, b)} \\
\textcircled{a} \times \textcircled{b} &= \boxed{a + b} \\
\frac{\textcircled{a}}{\textcircled{b}} + \frac{\textcircled{c}}{\textcircled{d}} &= \frac{\boxed{\max(a + d, c + b)}}{\boxed{b + d}} \\
\frac{\textcircled{a}}{\textcircled{b}} \times \frac{\textcircled{c}}{\textcircled{d}} &= \frac{\boxed{a + c}}{\boxed{b + d}}.
\end{aligned}$$

Next, I present a typical solution for *DoSegsIntersect* reducing it to a previously solved problem. If precision is a concern then I would not suggest using this solution. It is, however, useful for illustration purposes. In Algorithm 1, `SegsIntByConstruction` solves *DoSegsIntersect* by constructing the intersection point q of the lines through our segments and testing properties of q.

The *degree of an algorithm* is the maximum degree of its predicates and constructions. Let's see an example of precision analysis carried out `SegsIntByConstruction`. The algorithm tests if $\overleftrightarrow{ac}$ is parallel to $\overleftrightarrow{bd}$, in line 1. As each slope is degree 1 over 1, by clearing fractions, we check if the lines are parallel with degree 2. That is, we test if the predicate

$$\text{sign}\left((a_y - c_y)(b_x - d_x) - (a_x - c_x)(b_y - d_y)\right) = \textcircled{2}$$

Algorithm 1 `SegsIntByConstruction`(a, c, b, d): Determine if $\overline{ac}$ and $\overline{bd}$ intersect; if so return INTERSECT, if not return NOINTERSECT

Require: no three points are collinear
1: **if** $\overleftrightarrow{ac} \parallel \overleftrightarrow{bd}$ **then**
2:     **return** NOINTERSECT
3: **end if**
4: Point $q = \texttt{Intersect}(a, c, b, d)$ /* See Equation 3.2 */
5: Real $t_1 = (q_x - a_x)/(c_x - a_x)$
6: Real $t_2 = (q_x - b_x)/(d_x - b_x)$
7: **if** $t_1 \in (0, 1)$ & $t_2 \in (0, 1)$ **then**
8:     **return** INTERSECT
9: **else**
10:     **return** NOINTERSECT
11: **end if**

is zero. The `Intersect` construction, in line 4, computes the Cartesian coordinates of q, which we saw before are degree 3 over 2. Solving for t_1 and t_2, in lines 5 and 6 respectively, is degree 3 over 3. That is,

$$t_1 = \frac{q_x - a_x}{c_x - a_x} = \frac{③/② - ①}{①} = \frac{③/② - ②/②}{①} = \frac{(③ - ②)/②}{①} = \frac{③/②}{①} = \frac{③}{③},$$

and similarly for t_2. Next, we see that t_1 and t_2 are in $(0, 1)$, in line 7, with degree 3. Let $t_1 = n/d$. Both n and d are degree 3. We test if $t_1 \in (0, 1)$ by checking if:

$$(\text{sign}(n) < 0 \text{ and } \text{sign}(d) < 0 \text{ and } \text{sign}(n - d) > 0) \text{ or}$$
$$(\text{sign}(n) > 0 \text{ and } \text{sign}(d) > 0 \text{ and } \text{sign}(n - d) < 0).$$

As n and d are degree 3, $\text{sign}(n) = ③$, $\text{sign}(d) = ③$, and $\text{sign}(n - d) = ③$. Therefore, checking if $t_1 \in (0, 1)$ is degree 3. Similarly, checking if $t_2 \in (0, 1)$ is degree 3. Since the highest degree predicate of the algorithm is degree 3, `SegsIntByConstruction` is degree 3.

In an implementation of the typical solution, the coordinates of q would often be represented in floating point. Floating point values are a subset of $\mathbb{R}$, thus most real values must be rounded (see Section 2.1 for more on floating point representation) For a real value x, let $\texttt{fl}(x)$ be its floating point representation. For a geometric object o defined by coordinate values, $\texttt{fl}(o)$ is the geometric object induced by applying `fl` to the coordinate values of o. For example, the point $\texttt{fl}(q)$ has coordinate values $(\texttt{fl}(1/3), \texttt{fl}(8/3))$. Sometimes the difference between q and $\texttt{fl}(q)$ is negligible. In Figure 3.1, however, $\texttt{fl}(q)$ does not lie on either segment.

Instead, consider an algorithm that avoids the possibly erroneous construction of an intersection point. The algorithm uses the observation that $\overline{ac}$ intersects $\overline{bd}$ if and only if a and c are on opposite sides of $\overline{bd}$ and b and d are on opposite sides of $\overline{ac}$.

In Algorithm 2, `SegsIntByOrientation`, we use `Orientation` to determine if the two segments intersect by checking if the endpoints of $\overline{bd}$ are on opposite sides of $\overleftrightarrow{ac}$ and the endpoints of $\overline{ac}$ are on opposite sides of $\overleftrightarrow{bd}$.

Algorithm 2 `SegsIntByOrientation`(a, c, b, d): Determine if $\overline{ac}$ and $\overline{bd}$ intersect; if so return INTERSECT, if not return NOINTERSECT

Require: no three points are collinear

1: **if** $\texttt{Orientation}(a, c, b) \neq \texttt{Orientation}(a, c, d)$
    and $\texttt{Orientation}(b, d, a) \neq \texttt{Orientation}(b, d, c)$ **then**
2: **return** INTERSECT
3: **else**
4: **return** NOINTERSECT
5: **end if**

The `SegsIntByOrientation` algorithm avoids the construction of an intersection coordinate and arrives at a simpler algorithm. It uses the predicate in Equation 3.1, which is degree 2. The four `Orientation` predicates, in line 1, evaluate only Equation 3.1, so it too is degree 2. Thus, `SegsIntByOrientation` is degree 2.

This discussion shows that we can solve *DoSegsIntersect* with a degree 3 or degree 2 algorithm where both are constant time. Thus, from the stand point of precision I would suggest

`SegsIntByOrientation`. Picking an algorithm to solve a problem is not always so straightforward. As we will see in the following chapters, there appears to be a trade off between time and precision just as there is with time and space.

Chapter 4

Degrees of Predicates and Constructions

Geometric primitives, such as lines and points, may be represented by numbers in the computer in several ways. For example, a line can be specified as the coordinates of two points, as a slope and intercept, or as three coefficients of a homogeneous equation $ax + by + cw = 0$. Even for points there are choices of Cartesian, polar, or signed homogeneous coordinates [124, 125].

In this chapter, I define geometric predicates and constructions that are needed by the algorithms throughout the thesis. I focus on representations of primitives for which the predicates and constructions evaluate polynomials; in the next chapter, I show that the polynomials for the predicates that are defined here are irreducible. This chapter concludes with a discussion of verifying the preconditions for the predicates and construction. The results of both chapters, 4 and 5, are summarized in Tables 4.1 to 4.3.

Table 4.1 names representations of points and lines that the algorithms use. Signed homogeneous coordinates [124, 125] unify the representations. In short, a tuple (w, x, y) with $w \neq 0$ represents the Cartesian point $(x/|w|, y/|w|)$ with a spin that is clockwise if $w < 0$ and counter-clockwise if $w > 0$. A tuple with $w = 0$ represents the point at infinity in the direction (x, y). We call two points (or lines) a and b *distinct* if there is no $\lambda \in \mathbb{R} \setminus \{0\}$ such that $a = \lambda b$. Unless stated otherwise, points of $\mathbb{U}$ have positive spin. We keep track of the degrees of the elements of the tuple using degree notation, defined in Chapter 3, where we

denote a coordinate of degree k as $\textcircled{k}$.

For example, Cartesian point (x, y) has homogeneous representation $(1, x, y)$, which we write as ($\textcircled{0}$, $\textcircled{1}$, $\textcircled{1}$) because we care primarily about the degrees of the input coefficients in our analysis. Similarly, an input line specified by the coefficients homogeneous equation $(c, a, b) \cdot (w, x, y)^T = 0$ is ($\textcircled{1}$, $\textcircled{1}$, $\textcircled{1}$), while an input line specified in slope/intercept form is ($\textcircled{1}$, $\textcircled{1}$, $\textcircled{0}$).

Table 4.2 summarizes the degrees of constructions of points and lines that are named and derived in the indicated sections of this chapter. For example, the first line in the table tells us that the construction $\texttt{Line}(a, b)$ takes as input two StdPoints a and b and produces a line $\textcircled{1}\, x +\ \textcircled{1}\, y +\ \textcircled{2}\ = 0$ or in homogeneous form ($\textcircled{2}$, $\textcircled{1}$, $\textcircled{1}$). The derivation for `Line` is found in Section 4.1.1.

Table 4.3 summarizes the degrees of the predicates on points and lines. For example, the first three rows in the table tells us that $\texttt{PointOrdering}(a, b)$ for two StdPoints is degree 1, for a StdPoint and a Voronoi vertex is degree 3 (regardless of whether the Voronoi vertex is a or b), and for two Voronoi vertices is degree 5. The derivations are given in Section 4.2.1 and irreducibility is shown in Section 5.4.

4.1 Constructions

A geometric construction produces a new geometric object from the input. Here we consider the degrees of some well-known and useful constructions that appear regularly throughout this thesis[1].

[1] All constructions in this section have at least one precondition, such as the input points need to be distinct. When input does not conform to preconditions, the implementer may want an assertion or an exception to identify errors early; the user may prefer NaN, which propagates as specified by IEEE, so they know not to trust the results. From my experience implementing degree-driven algorithms in which static analysis tells us that the predicates are exact, it is better to throw an exception. Every time a construction in my prototype implementations experience a case where the precondition was not met, it was because of a programming bug, or an invalid input into an algorithm.

Table 4.1: Summary of point and line types that are input to predicates and constructions.

Name	Description
StdPoint	Point with degree 1 Cartesian coords or (⓪ , ① , ①) homogeneous coords.
StdLine	Line in std form with degree 1 coeffs. or (① , ① , ①) homogeneous coords.
StdSeg	Segment defined by two StdPoints.
SlopeLine	Line in slope intercept form or (① , ① , ⓪) in homogeneous coords.
VertLine	Vertical line defined by a degree 1 coords.
Bisector	Bisector constructed from two StdPoints.
LineInter	Output point of the construction `LineIntersection`$(\mathrm{StdLine}, \mathrm{StdLine})$.
BisectorInter	Output point of the construction `LineIntersection`$(\mathrm{Bisector}, \mathrm{Bisector})$.

Table 4.2: Summary of constructions of points and lines.

Name	Input	Output	Derivation
`Line`(a, b)	StdPoints	Line: (② , ① , ①)	4.1.1
`Plane`(a, b, c)	StdPoints	Plane: (③ , ② , ② , ②)	4.1.2
`Bisector`(a, b)	StdPoints	Line: (② , ① , ①)	4.1.3
`LineIntersection`(ℓ, h)	StdLines	Point: (② , ② , ②)	4.1.4
	Bisectors	Point: (② , ③ , ③)	
	SlopeLines	Point: (① , ① , ②)	
`VoronoiVertex`(a, b, c)	StdPoints	Point: (② , ③ , ③)	4.1.5

Table 4.3: Summery of predicates.

Name	Input	Degree	Derivation	Irreducible
$\texttt{PointOrdering}(a, b)$	StdPoints	1	4.2.1	5.4
	StdPoint and VorVert	3		
	VorVerts	5		
$\texttt{Orientation}(a, b, q)$	3 StdPoints	2	4.2.2	5.6
	2 StdPoints and 1 VorVert	4		
	1 StdPoints and 2 VorVert	6		
	3 VorVert	8		
$\texttt{Closer}(a, b, q)$	StdPoints	2	4.2.3	5.7
$\texttt{SideOfABisector}(B_{ab}, q)$	Bisector and a StdPoint	2	4.2.4	5.7
$\texttt{OrderOnLine}(B_{ab}, B_{cd}, \ell)$	Bisectors and a VertLine	3	4.2.5	5.8
$\texttt{OrderOnLine}(g, h, \ell)$	SlopeLines and a VertLine	2	4.2.6	5.9
$\texttt{InCircle}(a, b, c, q)$	StdPoints	4	4.2.7	5.10
$\texttt{IntersectInterior}(\overline{ac}, \overline{bd})$	StdSegs	2	4.2.8	-

A construction is preformed on coordinates of the input. Recall that we assume that input coordinates are degree 1, which are b-bit integers. For a universe of size $U = 2^b$, we can think of the set of all points with degree 1 Cartesian coordinates as lying on a $U \times U$ grid $\mathbb{U}$.

4.1.1 $\texttt{Line}(a, b)$

Often, the line defined though two distinct points $a, b \in \mathbb{U}$, depicted in Figure 4.1, is $\{ta + (1 - t)b \mid \forall t \in \mathbb{R}\}$. When $a_x \neq b_x$, we represent the line by algebraically expanding the equation $\frac{y-a_y}{x-a_x} = \frac{b_y-a_y}{b_x-a_x}$, arriving at the standard and slope-intercept forms,

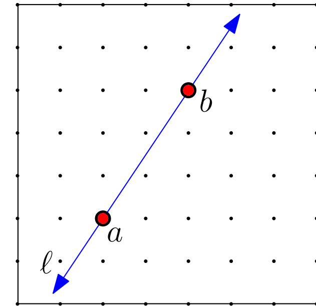


Figure 4.1: Given $a, b \in \mathbb{U}$ in red, construction $\texttt{Line}(a, b)$ produces the line ℓ in blue, through a and b.

$$\text{standard form:} \quad 0 = (a_y - b_y)x + (b_x - a_x)y + (a_x b_y - a_y b_x)$$

$$\text{slope-intercept form:} \quad y = \frac{b_y - a_y}{b_x - a_x}x + \frac{(a_y b_x - a_x b_y)}{b_x - a_x}.$$

Writing just the degrees of the coefficients, the line in

standard form is $0 = ① x + ① y + ②$ and in slope-intercept form is $y = ① / ① x + ② / ①$. The fraction means that the slope of the line through two points is a rational polynomial of degree 1 over 1 and the y-intercept is degree 2 over 1. This line representation, however, does not support orientation.

When orientation matters, we use the definition of a line from oriented projective geometry [125, Chapter 5.1]. For two distinct points a and b, the line from a to b is defined by the join $a \vee b$ (see [125, Chapter 5.4] for more on the join). The line from a to b can be written as a determinant equation:

$$\begin{vmatrix} a_w & a_x & a_y \\ b_w & b_x & b_y \\ w & x & y \end{vmatrix} = (a_x b_y - a_y b_x)w - (a_w b_y - b_w a_y)x + (a_w b_x - b_w a_x)y = 0$$

For a homogeneous coordinate $q = (q_w, q_x, q_y)$ let $\neg q = (-q_w, q_x, q_y)$, that is the spin of the coordinate is reversed. One can derive two useful identities about oriented lines. First that $a \vee b = \neg(b \vee a)$, which is interpreted geometrically to mean that the line from a to b is the same as the line from b to a with the opposite orientation. Second that $a \vee b = \neg b \vee a = b \vee \neg a$. (See [125, Chapters 3 and 5] for more on geometric interpretations and models of oriented projective geometry.)

4.1.2 `Plane`(a, b, c)

Three distinct points $a, b, c \in \mathbb{U}$ define a plane, which for some coefficients W, X, Y, and Z consists of all points $\{(x, y, z) \in \mathbb{R} \mid Xx + Yy + Zz + W = 0\}$. Construction `Plane`$(a, b, c)$ returns homogeneous coordinates (W, X, Y, Z) of the plane though a, b, and c, depicted in Figure 4.2.

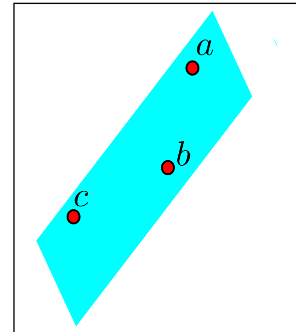


Figure 4.2: Given $a, b, c \in \mathbb{U}$ in red, the construction `Plane`(a, b, c) produces the plane, in blue, through a and b and c.

The coefficients can be derived by solving the system of equations

$$a_xX + a_yY + a_zZ = -W$$
$$b_xX + b_yY + b_zZ = -W$$
$$c_xX + c_yY + c_zZ = -W.$$

Let $D = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix}$. When $D \neq 0$, we can use Cramer's rule [135] to get

$$X = \frac{-W}{D}\begin{vmatrix} 1 & a_y & a_z \\ 1 & b_y & b_z \\ 1 & c_y & c_z \end{vmatrix} \qquad Y = \frac{-W}{D}\begin{vmatrix} a_x & 1 & a_z \\ b_x & 1 & b_z \\ c_x & 1 & c_z \end{vmatrix} \qquad Z = \frac{-W}{D}\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}.$$

Setting $W = D$ and applying elementary column operations, we find the coefficients,

$$W = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix} \qquad X = -\begin{vmatrix} 1 & a_y & a_z \\ 1 & b_y & b_z \\ 1 & c_y & c_z \end{vmatrix} \qquad Y = \begin{vmatrix} 1 & a_x & a_z \\ 1 & b_x & b_z \\ 1 & c_x & c_z \end{vmatrix} \qquad Z = -\begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & c_x & c_y \end{vmatrix}.$$

When $D = 0$, the plane passes through the origin. Thus, $W = 0$ and X, Y, and Z are the normal vector n of the plane, where $n = (a - b) \times (c - b)$.

Writing just the degree of the coefficients for input points in $\mathbb{U}$ is ② $x+$ ② $y+$ ② $z+$ ③ $=$ 0. The normal vector of the plane has degree (② , ② , ②). This representation does not support orientation.

For when orientation matters, we use the definition from oriented projective geometry [125, Chapter 5.2]. For three distinct points a, b and c the plane contains the line $a \vee b$ and the point c

which is the plane $a \vee b \vee c$. The plane can be written as a determinant equation:

$$\begin{vmatrix} a_w & a_x & a_y & a_z \\ b_w & b_x & b_y & b_z \\ c_w & c_x & c_y & c_z \\ w & x & y & z \end{vmatrix} = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix} w - \begin{vmatrix} a_w & a_y & a_z \\ b_w & b_y & b_z \\ c_w & c_y & c_z \end{vmatrix} x + \begin{vmatrix} a_w & a_x & a_z \\ b_w & b_x & b_z \\ b_w & c_x & c_z \end{vmatrix} y - \begin{vmatrix} a_w & a_x & a_y \\ b_w & b_x & b_y \\ c_w & c_x & c_y \end{vmatrix} z = 0$$

4.1.3 `Bisector`(a, b)

The *bisector* of distinct point sites $a, b \in \mathbb{U}$, depicted in Figure 4.3, is the locus of points equidistant to a and b, and is the perpendicular bisector of segment $\overline{ab}$. We sometimes write the bisector as B_{ab}. When the two sites defining the bisector are subscripted, as in a_1 and a_2, we may abbreviate the bisector as B_{12}.

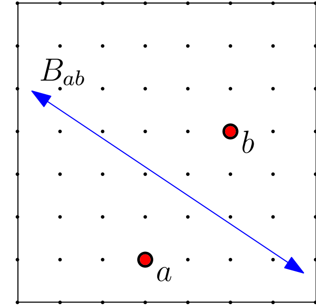


Figure 4.3: Given point sites $a, b \in \mathbb{U}$ the construction `Bisector`(a, b) produces the perpendicular bisector of $\overline{ab}$.

We derive an expression for B_{ab} by observing that it is the set of points $\{q \in \mathbb{R}^2 \mid \|q-a\|^2 = \|q-b\|^2\}$. Expanding this equation, we get the standard and slope-intercept forms of the equation for B_{ab} as

$$\text{standard form:} \qquad 0 = 2(a_x - b_x)x + 2(a_y - b_y)y + b_x^2 + b_y^2 - a_x^2 - a_y^2$$

$$\text{slope-intercept form:} \qquad y = \frac{(a_x - b_x)}{(b_y - a_y)}x + \frac{1}{2}\frac{b_x^2 + b_y^2 - a_x^2 - a_y^2}{(b_y - a_y)},$$

Observe that writing just the degree of the coefficients, the standard from of the bisector is $0 = \textcircled{1}\, x + \textcircled{1}\, y + \textcircled{2}$ and the y-intercept form is $y = \frac{\textcircled{1}}{\textcircled{1}} x + \frac{\textcircled{2}}{\textcircled{1}}$.

4.1.4 `LineIntersection`(ℓ, h)

The construction `LineIntersection`(ℓ, h) takes two distinct lines ℓ and h and computes the homogeneous coordinates of their intersection point q, as depicted in Figure 4.4. For Cartesian coordinates the output may be "parallel" rather than a point.

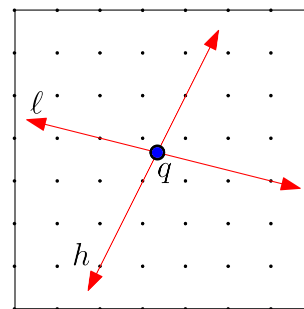


Figure 4.4: Given two non-parallel lines ℓ and h, in red, `LineIntersection`(ℓ, h) constructs the coordinates of q, in blue, the intersection point of ℓ and h.

We warm up with lines in slope intercept form and then consider lines in standard form. Given lines $\ell : y = \ell_m x + \ell_b$ and $h : y = h_m x + h_b$ the coordinates of the intersection point q of ℓ and h are

$$q = (\ell_m - h_m, h_b - \ell_b, \ell_m h_b - \ell_b h_m)$$

When $\ell_m = h_m$ the two lines are parallel, thus $q_w = 0$. When the coefficients of ℓ and h are degree 1, in Cartesian, the x-coordinate is degree 1 over 1 and the y-coordinate is degree 2 over 1, or homogeneous coordinates, (① , ① , ②). Many geometric algorithms analyzed with the Real-RAM model of computation have the property that processing left to right is the same as processing from top to bottom. However, as we will see in Section 4.2.1, this property does not always hold for lines in slope intercept form.

Next we consider lines ℓ and h, specified by homogeneous coordinates, i.e., $\ell = (\ell_w, \ell_x, \ell_y)$. The intersection point q of the lines ℓ and h is the solution to the linear equations

$$\ell_x x + \ell_y y + \ell_w = 0$$

$$h_x x + h_y y + h_w = 0.$$

The solution gives us the homogeneous coordinates

$$q = (\begin{vmatrix} \ell_x & \ell_y \\ h_x & h_y \end{vmatrix}, -\begin{vmatrix} \ell_w & \ell_y \\ h_w & h_y \end{vmatrix}, \begin{vmatrix} \ell_w & \ell_x \\ h_w & h_x \end{vmatrix}) \tag{4.1}$$

Next, we analyze the polynomials of the output points for three specific forms of input lines. When ℓ and h are degree 1, the lines have the form ① $x+$ ① $y+$ ① . Their intersection has homogeneous coordinates of the form (② , ② , ②). When ℓ and h are defined by a pair of points, Section 4.1.1 says that they have the form ① $x+$ ① $y+$ ② . Their intersection has homogeneous coordinates of the form (② , ③ , ③). When ℓ is a degree 1 line and a h is defined by a pair of points their intersection has homogeneous coordinates of the form (② , ③ , ③). When ℓ and h are defined by two bisectors (described in Section 4.1.3) and the bisectors share a common defining point, the intersection is particularly important. It is the same as computing a vertex of the Voronoi diagram, which the topic of Chapter 7 and Chapter 8. Thus, we focus on constructing a Voronoi vertex in the next section.

4.1.5 `VoronoiVertex`(a, b, c)

A *Voronoi vertex* v, constructed from distinct points $a, b, c \in \mathbb{U}$, is the point equidistant to a, b, and c, i.e., it is the center of the circle through a, b and c. As depicted in Figure 4.5 we can compute the coordinates of v by computing the intersection point of B_{ab} and B_{ac}. We derive the degree of a Voronoi vertex by combining the analysis on the degree of a bisector in Section 4.1.3 and line intersections in Section 4.1.4.

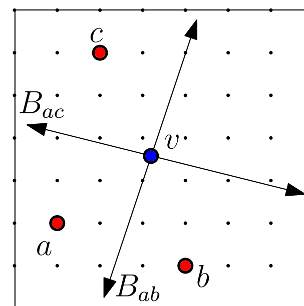


Figure 4.5: Given three non-collinear points $a, b, c \in \mathbb{U}$, in red, the construction `VoronoiVertex`(a, b, c), constructs the coordinates of the point v which is equidistant to a, b, and c.

Let $\ell = B_{ab}$ and $h = B_{ac}$ and recall that in Section 4.1.3 we saw that a bisector B can be written in standard form as $0 = B_x x +$

$B_y y + B_w = \textcircled{1}\, x + \textcircled{1}\, y + \textcircled{2}$. Equation 4.1 gave the homogeneous coordinates of the intersection point q, which, using degree notation, is

$$q = \left(\begin{vmatrix} \textcircled{1} & \textcircled{1} \\ \textcircled{1} & \textcircled{1} \end{vmatrix}, - \begin{vmatrix} \textcircled{2} & \textcircled{1} \\ \textcircled{2} & \textcircled{1} \end{vmatrix}, \begin{vmatrix} \textcircled{2} & \textcircled{1} \\ \textcircled{2} & \textcircled{1} \end{vmatrix} \right) = (\textcircled{2}, \textcircled{3}, \textcircled{3})$$

Thus, each Cartesian coordinate of a Voronoi vertex is degree 3 over 2 or in homogeneous form $(\textcircled{2}, \textcircled{3}, \textcircled{3})$.

4.2 Predicates and Simple Algorithms

Next we derive the degrees of common predicates and simple algorithms used throughout this thesis. We will no longer consider all possible point representations (e.g., point from $\mathbb{U}$, intersection point of two lines in standard form with degree 1 coordinates, etc.), but only those representations relevant in this thesis.

During implementation it is often convenient to use a perturbation scheme [3, 39, 41, 42, 114, 119] so that predicates are only positive or negative. In this section, we derive predicates without perturbation. Specific perturbation schemes are discussed in later sections in the context of specific algorithms.

4.2.1 `PointOrdering`(a, b)

We compare two distinct points a and b by lexicographic order of their Cartesian coordinates. The point $a \prec b$ if and only if $a_x < b_x$ or ($a_x = b_x$ and $a_y < b_y$). In Figure 4.6 the point a precedes the point b. Given a and b in homogeneous coordinates where $a = (a_w, a_x, a_y)$ and $b = (b_w, b_x, b_y)$ and a and b have the same spin

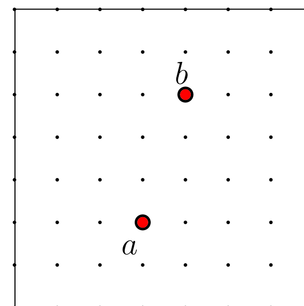


Figure 4.6: The point a precedes the point b.

$(a_w, b_w > 0$ or $a_w, b_w < 0)$, we determine the ordering by clearing fractions and subtracting to get the predicates $\texttt{OrderX}(a_w, a_x, b_w, b_x) = \text{sign}(a_w b_x - a_x b_w)$ and $\texttt{OrderY}(a_w, a_y, b_w, b_y) = \text{sign}(a_w b_y - a_y b_w)$.

We can rewrite the degrees of `Order` and `OrderY` as

$$\deg(\texttt{OrderX}(a_w, a_x, b_w, b_x)) = \max\{\deg(a_w) + \deg(b_x), \deg(a_x) + \deg(b_w)\}$$
$$\deg(\texttt{OrderY}(a_w, a_y, b_w, b_y)) = \max\{\deg(a_w) + \deg(b_y), \deg(a_y) + \deg(b_w)\}.$$

We determine the degree for four cases by plugging coordinates into the equations above. First, ordering two points in $\mathbb{U}$ is degree 1 since points of $\mathbb{U}$ have coordinates of degree (⓪ , ① , ①). Second, ordering the intersection points of two pairs of lines in standard form with degree 1 coefficients is degree 4 since the intersection points have coordinates of degree (② , ② , ②). Third, ordering a point of $\mathbb{U}$ and a Voronoi vertex is degree 3 since Voronoi vertices have coordinates of degree (② , ③ , ③). Fourth, ordering two Voronoi vertices is degree 5.

Points described in the previous paragraph have the same degree for their x- and y-coordinates; *asymmetric points* have Cartesian coordinates of different degrees. For example, the intersection of two lines in slope-intercept form, $y =$ ① $x +$ ① , has coordinates of degree (① , ① , ②). We can order the x-coordinate with degree 2, but the y-coordinate is degree 3. We will see in Chapter 6 that asymmetric points may lead us to select one sweep direction over another or one decomposition direction over another.

4.2.2 $\texttt{Orientation}(a, b, q)$

Given three distinct points a, b, q we determine if the straight line path from a to b to q forms an orientation that is clockwise, counter clockwise, or the three points are collinear. In Section 4.1.1, from

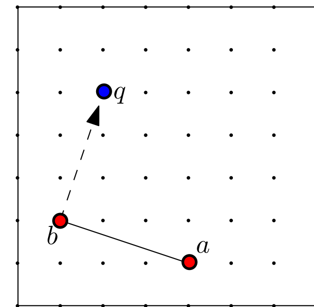


Figure 4.7: The path from a to b to q forms a right turn (clockwise orientation).

two points $a = (a_w, a_x, a_y)$ and $b = (b_w, b_x, b_y)$, we constructed an oriented line $a \vee b$. Using the line from a to b we can derive the predicate by plugging in the coordinates of $q = (q_w, q_x, q_y)$

$$\texttt{Orientation}(a, b, q) = \operatorname{sign}\left(\begin{vmatrix} a_w & a_x & a_y \\ b_w & b_x & b_y \\ q_w & q_x & q_y \end{vmatrix}\right)$$
$$= \operatorname{sign}\big((a_x b_y - a_y b_x) - (a_w b_y - b_w a_y) q_x + (a_w b_x - b_w a_x) q_y\big).$$

When all three points have positive spin ($a_w, b_w, q_w > 0$), we can interpret the positive, negative, and zero as path from a to b to q taking a left turn, right turn, and following a straight line, respectively.

Next, we analyze the polynomial for combinations of input points. When $a, b, q \in \mathbb{U}$ the points have the coordinate degree (⓪ , ① , ①), and the predicate is degree 2. When one point is a Voronoi vertex, Section 4.1.5 says that it has coordinates degree (② , ③ , ③), and the predicate is degree 4. In fact, the degree goes up by 2 for each Voronoi vertex: it takes degree 6 to determine the orientation of a point in $\mathbb{U}$ relative to a segment between two Voronoi vertices and degree 8 to determine orientation for three arbitrary Voronoi vertices.

4.2.3 $\texttt{Closer}(a, b, q)$

Given three distinct points $a, b, q \in \mathbb{U}$ we can test if a query point q is closer to a or b with degree 2 by comparing squared distances of Cartesian coordinates,

$$\operatorname{sign}(\|q - a\|^2 - \|q - b\|^2) = \operatorname{sign}(a_x^2 + a_y^2 - 2q_x a_x - 2q_y a_y - b_x^2 - b_y^2 + 2q_x b_x + 2q_y b_y) = \text{②} .$$

4.2.4 SideOfABisector(B_{ab}, q)

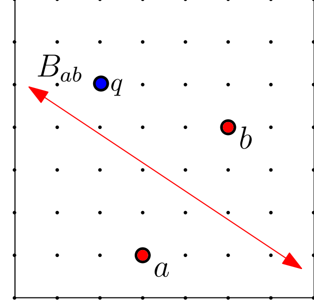


Figure 4.8: The query point q is on the same side of the B_{ab} as b.

Recall from Section 4.1.3 that the bisector of two distinct points $a, b \in \mathbb{U}$, written B_{ab}, is the locus of points equidistant to a and b. Bisector B_{ab} partitions the points of $\mathbb{U}$ into points on the same side as a, same side as b, and points on B_{ab}. The predicate SideOfABisector(B_{ab}, q) returns the partition containing a query point q. In Figure 4.8 the query point q is on the same side of B_{ab} as b.

Since the points on the same side as a are closer to a and on the same side as b are closer to b, we can use the same predicate as Closer and reinterpret the sign. Therefore, SideOfABisector is degree 2.

4.2.5 OrderOnLine(B_{ab}, B_{cd}, ℓ)

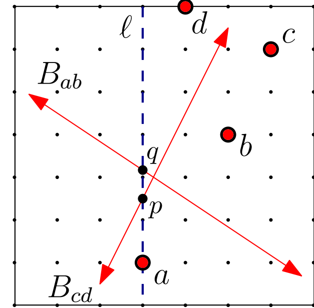


Figure 4.9: The bisector B_{ab} is above the bisector B_{cd} on the line ℓ.

Two non-vertical bisectors B_{ab} and B_{cd} intersect the vertical line $\ell : x = l$ with B_{ab} above, below or at the same point as B_{cd} on ℓ. In Figure 4.9, the line B_{ab} is above the line B_{cd} on the vertical line ℓ. We determine the ordering of B_{ab} and B_{cd} on ℓ by comparing y-coordinates of the intersection of B_{ab} and B_{cd} with ℓ.

Let q be the intersection of B_{ab} and ℓ. By plugging l into the slope-intercept form of B_{ab}, derived in Section 4.1.3, we get that

$$q_y = \frac{a_x^2 + a_y^2 + 2la_x - b_x^2 - b_y^2 - 2lb_x}{2(b_y - a_y)} = \frac{②}{①}.$$

Similarly, let p be the intersection of B_{cd} and ℓ. Comparing q_y and p_y, and clearing fractions gives a degree 3 test.

4.2.6 `OrderOnLine`(g, h, ℓ)

For three distinct lines in slope-intercept form $g : y = g_m x + g_b$ and $h : y = h_m x + h_b$, and $\ell : y = \ell_m x + \ell_b$ such that $g_m \neq \ell_m$ and $h_m \neq \ell_m$, let $q = g \cap \ell$ and $p = h \cap \ell$. We determine if q is to the left of, right of or incident to p by comparing the x-coordinates of p and q. In Figure 4.10, $h \cap \ell$ is left of $g \cap \ell$. Recall from Section 4.1.4 that p_x and q_x are rational coordinates of degree 1 over 1. Thus, the polynomial is degree 2. In Section 4.2.1, we saw ordering the y-coordinates of degree 1 lines in slope-intercept form is degree 3, thus, we avoid it.

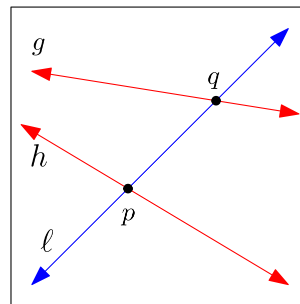


Figure 4.10: The intersection of g and ℓ is left of the intersection of h and ℓ.

4.2.7 `InCircle`(a, b, c, q)

Given three non-collinear sites, $a, b, c \in \mathbb{U}$ and a query point q also in $\mathbb{U}$, `InCircle` determines whether q is inside the circumcircle of $a, b, c \in \mathbb{U}$. In Figure 4.11, the query point q is inside the circumcircle of a, b, and c. This could be done by comparing the squared distances from `VoronoiVertex`(a, b, c) to a and to q, which gives a degree 6 polynomial when you clear fractions, but that polynomial factors to give the degree 4 determinant,

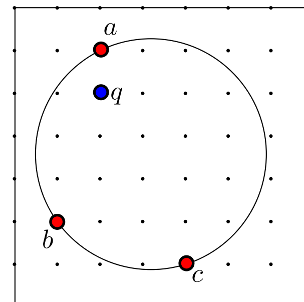


Figure 4.11: The query point q in inside the circumcircle of a, b and c.

$$\begin{vmatrix} 1 & a_x & a_y & a_x^2 + a_y^2 \\ 1 & b_x & b_y & b_x^2 + b_y^2 \\ 1 & c_x & c_y & c_x^2 + c_y^2 \\ 1 & q_x & q_y & q_x^2 + q_y^2 \end{vmatrix} = \begin{vmatrix} ⓪ & ① & ① & ② \\ ⓪ & ① & ① & ② \\ ⓪ & ① & ① & ② \\ ⓪ & ① & ① & ② \end{vmatrix} = ④ \,.$$

One may wonder if this polynomial can in fact be factored further. We see in Section 5.10 that in fact it cannot; it is irreducible.

Another way to arrive at this predicate is to consider the lifting map $M : \mathbb{R}^2 \to \mathbb{R}^3$ with $M(x, y) = (x, y, x^2 + y^2)$. The three lifted points $a' = M(a)$, $b' = M(b)$ and $c' = M(c)$ lie on a plane P in $\mathbb{R}^3$ that intersects the paraboloid $Q : x^2 + y^2 - z = 0$. The intersection curve of P and Q projects down onto the xy-plane as the circle though a, b and c; the points inside the circle lift to one side of P and the points outside lift to the other. Thus, an alternate way to derive this predicate is to write P as described in Section 4.1.2. Plugging in the lifted point q' into the equation of P arrives at the same predicate.

4.2.8 `IntersectInterior`($\overline{ac}, \overline{bd}$)

In Chapter 3 we saw two algorithms for determining if segments intersect, assuming that no three points were collinear. Here, I describe a degree 2 algorithm for checking if the interior of the segments intersect without the non-collinearity assumption.

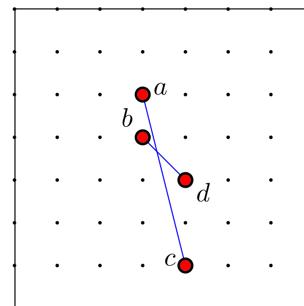


Figure 4.12: The interior of segments $\overline{ac}$ and $\overline{bd}$ intersect.

Given segments $\overline{ac}$ and $\overline{bd}$ with endpoints in $\mathbb{U}$, and $a \neq c$ and $b \neq d$, the intersection of the interiors of the segments is empty, a point, or an open segment. Recall that when no three points defining segments are collinear we can determine if the interiors of two segments intersect by checking if the endpoints of $\overline{bd}$ are on opposite sides of $\overleftrightarrow{ac}$ and the endpoints of $\overline{ac}$ are on opposite sides of $\overleftrightarrow{bd}$. The checks use `Orientation`, the degree 2 predicate from Section 4.2.2.

To handle collinear points we one additional predicate. Given a segment $\overline{rs}$ and a query point q with r, s and q collinear and $r \neq s$, `On`($\overline{rs}, q$) determines whether q is in the interior of $\overline{rs}$. The

dot product of the vectors $(r - q)$ and $(s - q)$ is negative when q is in the interior,

$$On(\overline{rs}, q) = \text{sign}((r_x - q_x)(s_x - q_x) + (r_y - q_y)(s_y - q_y)) = \textcircled{2} .$$

Now, we can give an algorithm for testing an intersection:

Algorithm 3 `IntersectInterior`$(\overline{ac}, \overline{bd})$: Determine if the interiors of $\overline{ac}$ and $\overline{bd}$ intersect; if so return INTERSECT, if not return NOINTERSECT

1: orientACB = `Orientation`(a, c, b)
2: orientACD = `Orientation`$(a, c.d)$
3: orientBDA = `Orientation`(b, d, a)
4: orientBDC = `Orientation`(b, d, c)
5: allCollinear = orientACB == 0 & orientACD == 0 & orientBDA == 0 & orientBDC == 0
6: anyCollinear = orientACB == 0 | orientACD == 0 | orientBDA == 0 | orientBDC == 0
7: **if** allCollinear & (`On`$(\overline{ac}, b)$<0 | `On`$(\overline{ac}, d)$<0 | `On`$(\overline{bd}, a)$<0 | `On`$(\overline{bd}, c)$<0) **then**
8: **return** INTERSECT
9: **else if** !anyCollinear & orientACB != orientACD & orientBDA != orientBDC **then**
10: **return** INTERSECT
11: **else**
12: **return** NOINTERSECT
13: **end if**

Algorithm 3 first does four orientation tests and checks if all and any of the points are collinear. When all the points are collinear the intersection is either an open segment or empty. When no points are collinear the orientation tests suffice.

Since the coordinates defining the input segments are in $\mathbb{U}$, Section 4.2.2 tells us that the four `Orientation` predicates are degree 2 and, as `On` is also degree 2, `SegmentIntersect` is degree 2.

4.3 Checking Preconditions

In an implementation, it is a good idea to verify that the input satisfies the preconditions. In this section, I describe how to detect when the preconditions, described in this chapter, fail.

4.3.1 Equal Homogeneous Coordinates

Many of the preconditions are simply checking if two homogeneous coordinates are *distinct*, that is, if one is a scaling of the other. Two homogeneous coordinates $a = (a_w, a_x, a_y)$ and $b = (b_w, b_x, b_y)$ are distinct when

$$\operatorname{sign}(a_x b_w - b_x a_w) \neq 0 \qquad \text{or} \qquad \operatorname{sign}(a_y b_w - b_y a_w) \neq 0. \tag{4.2}$$

4.3.2 Preconditions for Points

Many predicates and construction had the precondition that two points a and b are distinct. Below are three cases for when a and b are points of $\mathbb{U}$ or Voronoi vertices.

When $a, b \in \mathbb{U}$ they have coordinate degrees (⓪ , ① , ①). By Equation 4.2 testing if they are distinct is degree 1.

When b is a Voronoi vertex, Section 4.1.5 says that b has coordinate degrees (② , ③ , ③). By Equation 4.2, testing if a and b are distinct is degree 3, but we can do better. The Voronoi vertex b is equidistant to three non-collinear points $c, d, e \in \mathbb{U}$. If $\|c - a\|^2 = \|d - a\|^2 = \|d - a\|^2$ then a and b are not distinct. Since we are comparing squared distances of points in $\mathbb{U}$ only, testing if a Voronoi vertex and a point of $\mathbb{U}$ are distinct is degree 2.

When both a and b are Voronoi vertices, by Equation 4.2, testing if a and b are distinct is degree 5.

To summarize, we can determine if two points a and b are distinct:

- in degree 1, when $a, b \in \mathbb{U}$
- in degree 2, when $a \in \mathbb{U}$ and b is a Voronoi vertex
- in degree 5, when a and b are a Voronoi vertices.

4.3.3 Preconditions for Lines

Many predicates and constructions had the precondition that two lines ℓ and h are distinct. Below, are three cases for when ℓ and h are defined by a standard form equation with degree 1 coefficients, a pair of points in $\mathbb{U}$, and slope-intercept form with degree 1 coefficients.

When ℓ and h are defined by a standard form equation with degree 1 coefficients they have coordinate degrees (① , ① , ①). By Equation 4.2, testing if they are distinct is degree 2.

When ℓ and h are defined by pairs of points, Section 4.1.1 says that ℓ has coordinate degree (② , ① , ①). By Equation 4.2, testing if ℓ and h are distinct is degree 3, but we can do better. The line h is defined by two points $a, b \in \mathbb{U}$. If a and b lie on ℓ then ℓ and h are not distinct. Since we can test if a is on ℓ by checking if $\text{sign}(\ell \cdot a) = 0$, testing if two lines defined by points of $\mathbb{U}$ are distinct is degree 2.

When ℓ and h are in slope-intercept form they have coordinate degrees $y =$ ① $x +$ ① . By comparing slope and y-intercept, testing they are distinct is degree 1.

To summarize, we can determine if two lines ℓ and h are distinct:

- in degree 2, when lines are in standard form with degree 1 coefficients
- in degree 2, when lines are defined by pairs of points from $\mathbb{U}$
- in degree 1, when lines are in slope-intercept form with degree 1 coefficients.

The `OrderOnLine`(B_{ab}, B_{cd}, ℓ) predicate has the precondition that bisectors B_{ab} and B_{cd} are non-vertical. We can check that B_{ab} is non-vertical by checking if the y-coordinates of a and b differ (similarly for B_{cd}). Since $a, b \in \mathbb{U}$, testing if the bisectors are vertical is degree 1.

Chapter 5

Irreducibility of Polynomials

In Chapter 4, we saw geometric predicates and constructions and analyzed their precision. A natural question is, can any of the polynomials in the predicates be factored to produce a lower degree predicate? Conversely, are they irreducible? In this chapter, I show that they are. Note that unless otherwise specified, irreducibility is with respect to coefficients in $\mathbb{R}$.

First, in Section 5.1, I extend the "circle" notation used for degree analysis. The extended notation helps us discuss irreducibility, especially in some of the longer proofs (for example `InCircle` in Section 5.10). There are not many general techniques for showing that a multivariate polynomial is irreducible; Section 5.2-5.3 highlights a few techniques and lemmas that I found to be particularly helpful. In the rest of the sections, I show that the polynomials in the predicates presented in Chapter 4 are irreducible.

5.1 Notation

First, I define some notation and terminology helpful for this chapter. I use the polynomial $P(x, y, z, w) = xw - yz$ as a running example. When the number of variables in a polynomial is large it is convenient to abbreviate the set of variables as X. For example, we would write $X = \{x, y, z, w\}$, and $P(x, y, z, w)$ as $P(X)$.

We have represented a degree k multivariate polynomial as $\textcircled{k}$, for example, $P(X) = \textcircled{2}$.

We may be concerned with the degree of a polynomial with respect to some subset Y of its variables, that is, when we treat the variables not in Y as constants. If the degree of the polynomial is k with respect to Y we may represent the polynomial as $\textcircled{k}_Y$. For example, $P(X) = \textcircled{1}_w - \textcircled{0}_w$, and if $Y = \{w, z\}$ then $P(X) = \textcircled{1}_Y - \textcircled{1}_Y$.

5.2 Basket Weaving Technique for Proving Irreducibility

In general, it is difficult to show that an arbitrary multivariate polynomial is irreducible. Eisenstein's criterion is commonly used to show that polynomials in one variable with integral coefficients are irreducible over the rationals [36, 133]. One can show that a polynomial $p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ is irreducible by finding a prime $q \in \mathbb{Z}$ such that q divides a_i, for all i, but q^2 does not divide a_0. Applying the criterion is simple, however, there isn't as nice a criterion for multivariate polynomials that arise in geometric predicates.

Given a polynomial $P(X)$ that we wish show is irreducible, if P has no squared terms the following observation is often helpful:

Observation 1. For a set of variables X, when polynomial $P(X) = f(X)g(X)$ does not contain any squared terms, if a variable y of X is in f then the coefficient of any monomial with y in g must be zero.

This follows from the fact that a polynomial with coefficients in $\mathbb{R}$ in any number of variables is an integral domain [36, 133].

We use the observation and our main proof technique, which I call "basket weaving", to show:

Example 2. The degree 2 polynomial $P(x, y, z, w) = xw - yz$ is irreducible.

Proof. First, assume for contradiction that P has a factorization. Since P is degree 2 we have

$$P(x, y, z, w) = f(x, y, z, w)g(x, y, z, w) = \textcircled{1}\ \textcircled{1}\ .$$

Next, without loss of generality, assume that x is a monomial with non-zero coefficient in f and shuffle around terms. For $A \in \mathbb{R} \setminus \{0\}$ and polynomial f_1, rewrite

$$P(x, y, z, w) = xw - yz = (Ax + f_1(y, z, w))g(x, y, z, w).$$

By Observation 1 we know that the coefficient of any monomial with x as a variable in g is zero. Moreover, as P contains monomial xw, the coefficient of w in g must be non-zero (and zero in f). Thus, for $B \in \mathbb{R} \setminus \{0\}$ and polynomial g_1. We rewrite P as,

$$P(x, y, z, w) = (Ax + f_1(y, z))(Bw + g_1(y, z)).$$

Finally, we derive a contradiction. The coefficients of xy, xz, wy and wz in P are zero, and f_1 and g_1 are unit. This implies that the coefficient of yz is zero, which is a contradiction. □

5.3 Polynomials from Determinants of Matrices with Independent Variables

It is interesting to note that many of the polynomials in Chapter 4 are written as determinants. Next, I prove a helpful lemma: a polynomial that can be written as the determinant of a matrix of independent variables is irreducible over $\mathbb{R}$. Note that this remains true when the matrix is symmetric, although we will not have occasion to use that in this chapter. The 1907 textbook of Bôcher [9] contains results on polynomials from determinants, but delays in obtaining this book lead me to my own proofs (that differed from Bôcher's), which I include here. Given a matrix M of variables, we say that M is a matrix of *independent variables* if each entry of the matrix is unique.

Lemma 3. The polynomial obtained by expanding the $N \times N$ determinant of a matrix of independent variables is an irreducible polynomial of degree N.

Proof. Let D_N be the $N \times N$ dimensional matrix of independent variables x_{ij}, and $P(D_N) = |D_N|$ be the polynomial obtained by taking the determinant of D_N, and let D_N^{ij} be $(N-1) \times (N-1)$ minor matrix obtained by deleting the i-th row and j-th column of D_N. I show that $P(D_N)$ is irreducible by induction.

I start with some detail on the base case to clarify what is meant by a matrix with independent variables and to provide a simple example of the notation. Consider the 2×2 dimensional matrix with independent entries:

$$D_2 = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}.$$

In words, no variable is repeated in this matrix. The polynomial $P(D_N) = x_{11}x_{22} - x_{21}x_{12}$ is irreducible over $\mathbb{R}$ as we saw in Example 2.

Assume that the polynomial corresponding to the determinant of any $(N-1) \times (N-1)$ dimensional matrix with independent entries is irreducible and consider the rewriting of the Laplace expansion of $P(D_N)$ along the first row of D_N,

$$P(D_N) = x_{11}P(D_N^{11}) + \sum_{j=2}^{N}(-1)^{1+j}x_{1j}P(D_N^{1j})$$

The entries of each D_N^{1i} for $i = 1, 2, \ldots, N$ are independent, thus, each $P(D_N^{1i})$ is irreducible. As the entries of D_N are independent, the degree of $P(D_N)$ with respect to x_{11} is 1. Thus, if $P(D_N)$ has a nontrivial factorization, $P(D_N^{11})$ must divide each $P(D_N^{1j})$, for $j = 2, 3, \ldots, N$, which is not the case. In $P(D_N^{1j})$, the coefficient of any monomial with x_{jj} is zero, however, it is non-zero in $P(D_N^{11})$, thus $P(D_N^{11})$ does not divide any of the $P(D_N^{1j})$, let alone all of them. □

5.4 $\texttt{PointOrdering}(a, b)$ is Irreducible

We start off with a simple irreducibility argument that the polynomials described in Section 4.2.1 for the point orderings of two points a and b are irreducible. Consider the case where the x-coordinates of a and b differ (the arguments for the y-coordinate when $a_x = b_x$ are similar).

First, consider when points a and b are from $\mathbb{U}$. The polynomial $a_x - b_x = ①$ cannot be factored. Next, consider point a defined by the intersection of lines $\ell^{(a)}$ and $h^{(a)}$, and b defined by $\ell^{(b)}$ and $h^{(b)}$. All the lines are distinct and in standard form with degree 1 coordinates, e.g., line $\ell^{(a)}$ is defined by points satisfying the equation $\ell_x^{(a)} x + \ell_y^{(a)} y + \ell_w^{(a)} = 0$. The points a and b have homogeneous coordinates of the form (② , ② , ②) and the corresponding polynomial is $P(a_x, a_w, b_x, b_w) = \mathrm{sign}(a_x/a_w - b_x/b_w) = \mathrm{sign}(a_x b_w - b_x a_w)$, which is degree 4. Next, I show that P is irreducible.

As the lines defining a and b are unrelated, no cancellation will occur from multiplying $a_x b_w$ or $b_x a_w$. Thus, the only way to achieve a sub-degree 4 test is if there is some cancellation between a_x and a_w. Recall that

$$a_x = -\begin{vmatrix} \ell_w & \ell_y \\ h_w & h_y \end{vmatrix} \qquad \text{and} \qquad a_w = \begin{vmatrix} \ell_x & \ell_y \\ h_x & h_y \end{vmatrix}.$$

By Lemma 3 both a_x and a_w are irreducible. Moreover, there is no unit k such that $a_x = k a_w$. Thus, a_x and a_w are relatively prime, and so P is irreducible and degree 4.

A similarly structured argument shows that `PointOrdering` for Voronoi vertices and segment intersections are irreducible and degree 5. The main difference is that it is more involved to show that there is no cancellation when representing the input point in Cartesian coordinates; see Section 5.5. We summarize the irreducibility results for point ordering, described in this section, as follows:

Lemma 4. For points a and b, the polynomial in `PointOrdering(a,b)` is irreducible and

- degree 1 when $a, b \in \mathbb{U}$;
- degree 3 when $a \in U$ and b is the intersection of two lines in standard form with degree 1 coefficients or a Voronoi vertex defined by three points of $\mathbb{U}$;
- degree 4 when a and b are intersections of two lines in standard form with degree 1 coefficients;
- degree 5 when a and b are either Voronoi vertices, defined by three points of $\mathbb{U}$, or the intersection of segments with endpoints in $\mathbb{U}$.

5.5 `VoronoiVertex`(a, b, c) Cannot be Simplified

Next, I show that the coordinates produced by the construction `VoronoiVertex` cannot be simplified; that is, the numerator and denominator are both irreducible and relatively prime. Recall that for three points $a, b, c \in \mathbb{U}$ with a, b, and c distinct, the construction `VoronoiVertex`(a, b, c) produces the coordinates of the point q equidistant to a, b, and c. Point q has Cartesian coordinates of degree 3 over 2, which we derived by intersecting the bisectors of a and b, and c and d:

$$q_x = \frac{\begin{vmatrix} b_x^2 + b_y^2 - a_x^2 - a_y^2 & 2(a_x - b_x) \\ d_x^2 + d_y^2 - c_x^2 - c_y^2 & 2(c_x - d_x) \end{vmatrix}}{\begin{vmatrix} 2(a_x - b_x) & 2(a_y - b_y) \\ 2(c_x - d_x) & 2(c_y - d_y) \end{vmatrix}} \quad \text{and} \quad q_y = \frac{\begin{vmatrix} b_x^2 + b_y^2 - a_x^2 - a_y^2 & 2(a_y - b_y) \\ d_x^2 + d_y^2 - c_x^2 - c_y^2 & 2(c_y - d_y) \end{vmatrix}}{\begin{vmatrix} 2(a_x - b_x) & 2(a_y - b_y) \\ 2(c_x - d_x) & 2(c_y - d_y) \end{vmatrix}}.$$

I'll sketch the proof that q_x cannot be simplified, a similar argument can be applied to q_y. Consider the denominator of q_x. Each entry is a difference of unrelated variables. Replace each difference by a variables and apply Lemma 3 to show that the denominator is irreducible. Consider

the numerator of q_x. Let P be the polynomial obtained by expanding the numerator of $q_x/2$, and let $X = \{a_x, a_y, b_x, b_y, c_x, c_y, d_x, d_y\}$ be the set of variables in P. We can rewrite P as

$$\begin{aligned}P(X) &= (c_y - d_y)(b_x^2 + b_y^2 - a_x^2 - a_y^2) - (a_y - b_y)(d_x^2 + d_y^2 - c_x^2 - c_y^2) \\ &= g_2(c_y, d_y)f_1(a_x, a_y, b_x, b_y) - f_2(a_y, b_y)g_1(c_x, c_y, d_x, dy)\end{aligned}$$

Expanding P results in no additive cancellation and f_1 and g_1 are irreducible. Thus, for P to factor, f_2 would need to divide f_1 (and by the symmetry of the polynomial g_2 would divide g_1), which is not the case. So, the numerator is irreducible.

Since the numerator and denominator are both irreducible, the only hope for cancellation is if the denominator divides the numerator. However, this is not the case, as their degrees are relatively prime. Thus,

Lemma 5. For points $a, b, c \in \mathbb{U}$, the construction `VoronoiVertex` produces a point whose Cartesian coordinates are degree 3 over 2 where the numerator and denominator are irreducible and relatively prime.

5.6 `Orientation`(a, b, q) is Irreducible

Recall that the polynomial corresponding to the `Orientation` predicate for point in $\mathbb{U}$ described in Section 4.2.2,

$$P(a_x, a_y, b_x, b_y, q_x, q_y) = \begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & q_x & q_y \end{vmatrix} = b_x q_y - b_x a_y - a_x q_y - q_x b_y + q_x a_y + a_x b_y$$

Next, I use a basket weaving argument to show that P is irreducible by contradiction. Assume that P can be written as the product of two degree 1 polynomials f and g, and rewrite these polynomials

until we arrive at a contradiction. Similar to the proof of Example 2, I often use Observation 1.

Assume for the sake of deriving a contradiction that P can be factored into two polynomials f and g both of degree 1. Without loss of generality, let b_x be a monomial of f. For $A \in \mathbb{R} \setminus \{0\}$ and polynomial f_1 of degree 1, we can rewrite,

$$\begin{aligned} P(a_x, a_y, b_x, b_y, q_x, q_y) &= b_x q_y - b_x a_y - a_x q_y - q_x b_y + q_x a_y + a_x b_y \\ &= f(a_x, a_y, b_x, b_y, q_x, q_y) g(a_x, a_y, b_x, b_y, q_x, q_y) \\ &= (A b_x + f_1(a_x, a_y, b_y, q_x, q_y)) g(a_x, a_y, b_x, b_y, q_x, q_y). \end{aligned}$$

By Observation 1, the coefficient of b_x in g is zero. Moreover, as P contains monomials $b_x q_y$ and $b_x a_y$, the coefficients of q_y and a_y in g must be non-zero. Thus, for $B, C \in \mathbb{R} \setminus \{0\}$ and polynomial g_1 of degree 1, we can again rewrite

$$P(a_x, a_y, b_x, b_y, q_x, q_y) = (A b_x + f_1(a_x, a_y, b_y, q_x, q_y))(B q_y + C a_y + g_1(a_x, b_y, q_x)).$$

By applying Observation 1 again, we see that the coefficients of q_y and a_y in f_1 are zero, and as $q_x a_y$ is a monomial of P the coefficient of q_x in f_1 must be non-zero. By applying Observation 1 one last time, we see that the coefficient of q_x in g_1 must be zero. Thus, for $D \in \mathbb{R}$ and polynomials f_2 and g_2 of degree 1, we can rewrite

$$\begin{aligned} P(a_x, a_y, b_x, b_y, q_x, q_y) &= (A b_x + D q_x + f_2(a_x, b_y))(B q_y + C a_y + g_2(a_x, b_y)) \\ &= D B q_x q_y + h(a_x, a_y, b_x, b_y, q_x, q_y), \end{aligned}$$

with h a polynomial of degree 2 without monomial $q_x q_y$. However, as both D and B are non-zero, the right hand side of the equation has monomial $q_x q_y$ with non-zero coefficient DB, which is absent from the left hand side, providing a contradiction. This implies that the polynomial in

`Orientation` for point of $\mathbb{U}$ is an irreducible polynomial of degree 2.

In Section 4.2.2 we also saw that for homogeneous coordinates a, b, and q, we have the polynomial obtained by expanding the determinant

$$Q(a_w, a_x, a_y, b_w, b_x, b_y, q_w, q_x, q_y) = \begin{vmatrix} a_w & a_x & a_y \\ b_w & b_x & b_y \\ q_w & q_x & q_y \end{vmatrix}.$$

When the coordinates of a, b, and q are degree 1, by Lemma 3 the polynomial is irreducible. Provided that there is no cancellation within the homogeneous coordinate (for example, a Voronoi vertex), by slightly modifying the irreducibility argument (dependent on the form of the variables), we get that these polynomials are irreducible as well.

We summarize the above discussion with the following lemma:

Lemma 6. For points a, b and q, the polynomial in `Orientation` is irreducible and:

- degree 2 when $a, b, q \in \mathbb{U}$,
- degree 4 when a is a Voronoi vertex defined by three points of $\mathbb{U}$ and $b, q \in \mathbb{U}$,
- degree 6 when a and b are Voronoi vertices defined by points of $\mathbb{U}$ and $q \in \mathbb{U}$,
- degree 8 when a, b, q are Voronoi vertices defined by points of $\mathbb{U}$.

5.7 $\texttt{SideOfABisector}(B_{ab}, q)$ and $\texttt{Closer}(a, b, q)$ are Irreducible

Recall from Section 4.2.4 that for three points $a, b, q \in \mathbb{U}$, predicates $\texttt{SideOfABisector}(B_{ab}, q)$ and $\texttt{Closer}(a, b, q)$ are based off a degree 2 squared distance comparison,

$$\text{sign}(\|q - a\|^2 - \|q - b\|^2) = \text{sign}(a_x^2 + a_y^2 - 2q_xa_x - 2q_ya_y - b_x^2 - b_y^2 + 2q_xb_x + 2q_yb_y)$$

Let $X = \{a_x, a_y, b_x, b_y\}$ and $Q = \{q_x, q_y\}$, we can rewrite the polynomial of the above predicate as

$$P(X, Q) = 2(b_x - a_x)q_x + 2(b_y - a_y)q_y + a_x^2 + a_y^2 - b_x^2 - b_y^2.$$

Assume for the sake of deriving a contradiction that P can be factored as $P(X, Q) = \textcircled{1}\ \textcircled{1}$. As P is linear with respect to q_x and q_y and $\mathbb{R}[x]$ is an integral domain, $P(X, Q) = \textcircled{1}_Q\ \textcircled{0}_Q$. Let f and g be polynomials such that $P(X, Q) = f(X, Q)g(X)$.

The degree of g with respect to a_y is 0 or 1. If the degree were 0 then the non-zero coefficient of a_y^2 would cause the degree of f to be 2, thus, it is 1. We can apply the same analysis to a_x, b_x, and b_y to show that $P(X, Q) = f(X, Q)g(X)$ where the degree of a_x, b_x and b_y in f and g is one. However, this factorization implies that the coefficient of monomial q_xa_y in P is non-zero, which is a contradiction. Thus, P is irreducible, and therefore:

Lemma 7. For points a, b and q, the polynomial in $\texttt{SideOfABisector}$ and $\texttt{Closer}$ is irreducible and degree 2.

5.8 `OrderOnLine`(B_{ab}, B_{cd}, ℓ) is Irreducible

Recall from Section 4.2.5 that the for four points $a, b, c, d \in \mathbb{U}$ and a vertical line ℓ defined by a degree 1 coordinate l, the `OrderOnLine` test compares the degree 2 over 1, y-coordinates of $B_{ab} \cap \ell$ and $B_{cd} \cap \ell$, which has the form

$$B_{ab} \cap \ell = \frac{2(b_x - a_x)l + a_x^2 + a_y^2 - b_x^2 - b_y^2}{2(a_y - b_y)} = \frac{g(a_x, a_y, b_x, b_y, l)}{f(a_y, b_y)} = \frac{g_1}{f_1}$$
$$B_{cd} \cap \ell = \frac{2(d_x - c_x)l + c_x^2 + c_y^2 - d_x^2 - d_y^2}{2(c_y - d_y)} = \frac{g(c_x, c_y, d_x, d_y, l)}{f(c_y, d_y)} = \frac{g_2}{f_2}$$

I show that `OrderOnLine` for bisectors is irreducible and degree 3 by showing that the polynomial, $P = g_1 f_2 - g_2 f_1$ is irreducible.

First consider P when one point is shared between the two bisectors. Let p, q and r be these three points with p shared, and call this polynomial $\widehat{P}$. If we show that $\widehat{P}$ is an irreducible polynomial of degree 3, then, we get that P is irreducible, because if not, $\widehat{P}$ would factor.

Note that $P(p_x, p_y, q_x, q_y, p_x, p_y, r_x, r_y, l)$ can be interpreted geometrically as testing if $v =$ `VoronoiVertex`(p, q, r) lies to the right or left of a vertical line ℓ. Recall that Lemma 5 tells us that the Cartesian coordinates of a Voronoi vertex are rational degree 3 over 2, the numerator and denominator are irreducible and relatively prime. Thus, we can rewrite $\widehat{P} = v_x - v_w l =$ ③ $-$ ② ① . As v_x and v_w are relatively prime and irreducible, and l is different from the set of variables that define v_x and v_w, $\widehat{P}$ is irreducible. Therefore:

Lemma 8. For two bisectors defined by three or four points of $\mathbb{U}$ the polynomial in `OrderOnLine` is irreducible and degree 3.

5.9 `OrderOnLine`(l, h, ℓ) is Irreducible

Recall from Section 4.2.6 that for three lines, l, h, and ℓ, in slope intercept form, defined by degree 1 coefficients, we determine if $p = l \cap \ell$ is left, right or incident to $q = h \cap \ell$ with a degree 2 test that compares coordinates of a point with homogeneous coordinates of the form (① , ① , ②). The degree 2 follows from the comparison of the x-coordinates of the intersection points in Cartesian form.

Furthermore, recall from Section 4.1.4 that:

$$p_x = \frac{l_b - \ell_b}{\ell_m - l_m} \qquad \text{and} \qquad q_x = \frac{h_b - \ell_b}{\ell_m - h_m}$$

and for $X = \{l_m, l_b, h_m, h_b, \ell_m, \ell_b\}$, the polynomial in `OrderOnLine` for X-intersections is

$$P(X) = (l_b - \ell_b)(\ell_m - h_m) - (\ell_m - l_m)(h_b - \ell_b) = \begin{vmatrix} (l_b - \ell_b) & (\ell_m - l_m) \\ (h_b - \ell_b) & (\ell_m - h_m) \end{vmatrix} = \begin{vmatrix} 1 & \ell_b & -\ell_m \\ 1 & l_b & -l_m \\ 1 & h_b & -h_m \end{vmatrix}.$$

This familiar form was encountered in the polynomial in the `Orientation` predicate for points from $\mathbb{U}$ (the only difference is now, the third column is negative). In Section 5.6 we showed that this polynomial is irreducible. We can use a similar argument here, thus,

Lemma 9. For three lines in slope intercept form with degree 1 coefficients, the polynomial in `OrderOnLine` for x-intersections is irreducible and degree 1.

5.10 `InCircle`(a, b, c, q) is Irreducible

Recall that the polynomial corresponding to the `InCircle` predicate, described in Section 4.2.7. Next, I show that this polynomial is irreducible.

Let $X = \{a_x, a_y, b_x, b_y, c_x, c_y\}$ and $Q = \{q_x, q_y\}$. We can write the polynomial as,

$$P(X,Q) \quad = \quad \begin{vmatrix} 1 & a_x & a_y & a_x^2 + a_y^2 \\ 1 & b_x & b_y & b_x^2 + b_y^2 \\ 1 & c_x & c_y & c_x^2 + c_y^2 \\ 1 & q_x & q_y & q_x^2 + q_y^2 \end{vmatrix}.$$

We proceed by showing that if P can be factored, it must reduce to two polynomials of the form $P(X,Q) = f(Q)g(X)$, i.e., the coefficient of any monomial of g with q_x or q_y as a variable is zero and similarly for the coefficients of X in f. Then I show that such a factoring does not exist. I begin by expanding P by minors in the last row,

$$P(X,Q) = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 \\ b_x & q_y & q_x^2 + q_y^2 \\ c_x & c_y & c_x^2 + c_y^2 \end{vmatrix} - \begin{vmatrix} 1 & a_y & a_x^2 + a_y^2 \\ 1 & b_y & q_x^2 + q_y^2 \\ 1 & c_y & c_x^2 + c_y^2 \end{vmatrix} q_x + \begin{vmatrix} 1 & a_x & a_x^2 + a_y^2 \\ 1 & b_x & q_x^2 + q_y^2 \\ 1 & c_x & c_x^2 + c_y^2 \end{vmatrix} q_y - \begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & q_y \\ 1 & c_x & c_y \end{vmatrix} (q_x^2 + q_y^2) \tag{5.1}$$

Observe that the degree of P with respect to Q is 2, so we can write P as $P_X(Q) = \textcircled{2}_Q$. The polynomial has two factorizations with respect to Q, $P_X(Q) = \textcircled{2}_Q \textcircled{0}_Q$ or $P_X(Q) = \textcircled{1}_Q \textcircled{1}_Q$. Notice that if P_Q is two linear factors in Q, then the zero set of P_X is interpreted geometrically as a pair of lines. However, the set of points in which `InCircle` evaluates to zero is a circle. Therefore, if P_X factors, it does so into degree 2 and degree 0 polynomials with variables of Q.

Assume for the sake of deriving a contradiction that P factors. Then, by the previous discussion, $P(X,Q) = f(X,Q)g(Q)$, with neither f nor g a unit. Next, I show that if P factors then the coefficients of X in f are zero in three steps: (s1) show that f and g are degree 2 or 0 with respect to any single coefficient; (s2) show that all the coefficients of X must be in the same polynomial; and (s3) show that polynomial must be g.

First, let $Y = X \setminus \{a_y\}$, and consider the degree of f and g with respect to a_y. Since the power of any variable in P is at most 2, we have the degree of g with respect to a_y is 0, 1 or 2. Consider the monomials of P with $a_y q_x$ to any power. They can be written as $\alpha_1(Y)a_y^2 q_x$, $\alpha_2(Y)a_y q_x$, and $\alpha_3(Y)a_y q_x^2$. If the $f = \text{①}_{a_y}$ then $g = \text{①}\ (a_y)$, which erroneously implies that $q_y a_y$ is a monomial of P. Apply this same argument to each monomial of X and get (s1).

Second, let $O(X)$ be the coefficient of q_x^2 in Equation 5.1. Notice that O is the irreducible polynomial from Lemma 6. Thus, the variables of X must appear in the same polynomial. The preceding statement combined with (s1) gives (s2).

Third, if the coefficients of X are non-zero in f then we have that $P(X, Q) = f(X, Q)g()$. However, this implies that g is a unit, which violates our initial assumption. Therefore, we have (s3); that is, $P = f(Q)g(X)$. However, this implies that $O(X)$ must divide every coefficient of Q in Equation 5.1. In particular, it must divide $\alpha(X)$, the coefficient of q_x in P. But, consider the monomial $a_y b_x$, it has a non-zero coefficient in $O(X)$ and a zero coefficient in $\alpha(X)$. Thus, as $\mathbb{R}[X]$ is an integral domain and $O(X)$ does not divide $\alpha(X)$, we have a contradiction. Therefore:

Lemma 10. The polynomial in `InCircle` is irreducible and degree 4.

Chapter 6

Gabriel Graph

In this chapter I give an example of how adding the requirement that the degree of an algorithm be low can lead us to redesigning algorithms and data structures. I describe an algorithm for computing the Gabriel graph in double precision, designed with Vishal Verma [101]. We cannot achieve the efficiency of the usual approach that obtains the Gabriel graph as a subset of the Voronoi diagram, since computing that requires degree at least 4, but we are able to achieve degree 2 with better than the brute force running time.

6.1 Introduction

Given a finite set of sites S, an edge (s_i, s_j) with $s_i, s_j \in S$ is in the *Gabriel graph of* S if the edge maintains the Gabriel property, that is, the closed disk with diameter $\overline{s_i s_j}$ contains no points of S besides s_i and s_j. In Figure 6.1 the left image depicts the Gabriel graph of seven sites. In the right image, edge (a, b) is in the Gabriel graph because only a and b are in the closed disk with diameter $\overline{ab}$. The edge (e, f) is not in the Gabriel graph because the closed disk with diameter $\overline{ef}$ contains g. It is know that the Gabriel graph [48] is a subgraph of the Delaunay triangulation. Matula and Sokal [92] showed how to compute the Gabriel graph directly from the Delaunay triangulation in time proportionate to the number of sites in S.

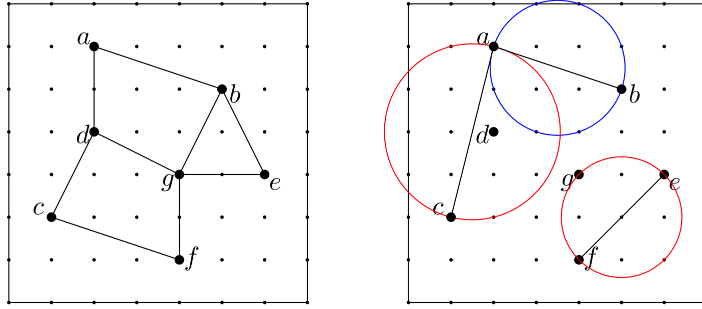


Figure 6.1: *Left*: The Gabriel graph of seven sites S. *Right*: The closed disk with diameter $\overline{ab}$, with boundary depicted in blue, contains no sites of $S \setminus \{a, b\}$ so edge (a, b) is in the Gabriel graph. The closed disks with diameter $\overline{ac}$ and $\overline{ef}$, with boundaries depicted in red, contain sites d and g, respectively, thus edges (a, c) and (e, f) are not in the Gabriel graph.

Computing the Delaunay triangulation requires four times the precision of the input coordinates, and Matula and Sokal's Gabriel graph algorithm uses six-fold precision. Liotta [88] showed how to implement Matula and Sokal's algorithm using only double (two-fold) precision, however, it still requires four-fold precision for computing the Delaunay triangulation. A natural question that follows is, can we compute the Gabriel graph with only double precision?

The answer is yes! In Section 6.3 we show that we can compute the Gabriel graph with double precision (albeit rather slowly). In the next section we describe how to compute the arrangement of lines that are the duals of points in double precision, which will be important for the Gabriel graph construction.

6.2 Arrangements of Dual Lines

Recall the point/line duality [31] that maps a point $p = (p_x, p_y)$ to a line $p^* := (y = p_x x - p_y)$ and a line $l := y = mx + b$ to a point $l^* := (m, -b)$. The set S^* is the set of lines dual to the set of sites of S. We notate the arrangement of the lines in S^* as $A(S^*)$ and the Gabriel graph of S as $G(S)$.

It is known that for a set of line segments defined by their endpoints, computing an arrangement requires four times the input precision and computing its trapezoidation requires five times the

input precision [90]. In this section, we show that for a set of lines defined as duals of points double precision suffices for computing an arrangement and its trapezoidation.

We begin by observing that for non-parallel lines p^* and q^*, the x-coordinate of the point $\ell^* = p^* \cap q^*$ is the slope of line $\ell = \overleftrightarrow{pq}$, which is $(p_y - q_y)/(p_x - q_x)$.

Observation 11. The x-coordinate of the intersection of two dual lines is represented by a rational polynomial of degree 1 over 1.

For three dual lines, p^*, q^*, and r^*, where p^* and q^* intersect r^*, by Observation 11 and clearing fractions, we compare the x-ordering of the intersection points with degree 2. We call this the `OrderOnALine`(p^*, q^*, r^*) predicate.

By using the `OrderOnALine` predicate in an incremental construction of an arrangement (such as [31, Chapter 8.3]) we achieve a degree 2 construction. Furthermore, we can use the `OrderOnALine` predicate to add the verticals into the arrangement and get its trapezoidation.

Lemma 12. For n dual lines S^*, we can compute the arrangement of S^* and its trapezoidation in $O(n^2)$ time and degree 2.

6.3 Algorithm Description

Next, I describe how to construct the Gabriel graph in $O(n^2)$ using degree 2. I begin by defining the predicates used in our construction. Let $D(p, q)$ be the closed disk with $\overline{pq}$ as the diameter. A site s, distinct from p and q, *kills* the edge (p, q) if s lies in $D(p, q)$. Let m be the midpoint of $\overline{pq}$. The degree 2 predicate `IsKiller`(p, q, s) compares the squared distances between m and s and m and p to determine if s kills edge (p, q).

Given the arrangement $A(S^*)$ and a site $s_i \in S$ we compute the circular orderings of the sites in $S \setminus \{s_i\}$ around s_i so that we may efficiently determine the edges of the Gabriel graph incident to s_i. Consider the line s_i^*, each vertex $v_j \in A(S^*)$ that lies on s_i^* corresponds to a line through s_i

and some other site $s_j \in S$. As mentioned in Section 6.2, the slope of $\overleftrightarrow{s_i s_j}$ is the x-coordinate of v_j, thus, by walking along s_i^* in $A(S^*)$ we find a set of lines, through s_i ordered by slope, which gives the circular ordering of the sites of $S \setminus \{s_i\}$ around s_i.

Constructing a circular ordering for a site is a purely topological operation on the arrangement $A(S^*)$ and uses degree 0. For each site s, computing the circular ordering takes time proportional to the number of vertices that lie on s^*, which is $O(n)$.

Lemma 13. Given $A(S^*)$, for site $s \in S$, we can compute the circular ordering of the sites in $S \setminus \{s\}$ around s in $O(n)$ time and degree 0.

Once we have computed a circular ordering of $S \setminus \{s\}$ around each $s \in S$, in $O(n)$ time we can compute the Gabriel edges incident at s. The key idea behind this step is captured in Lemma 14.

We number the circularly ordered sites in $S \setminus \{s\}$ in a counter clockwise manner starting with any $s_0 \in S \setminus \{s\}$, i.e., $s\vec{s_{i+1}}$ is the first ray counter clockwise from $s\vec{s}_i$ at s. Let $D_l(s, s_i)$ denote the closed semicircular disk that has $\overline{ss_i}$ as the diameter and lies to the left of $s\vec{s}_i$. The site s_j kills the edge (s, s_i) *from the left* if and only if s_j lies in $D_l(s, s_i)$. Then,

Lemma 14. If s_j lies in $D_l(s, s_i)$ and $\forall k \in \{i, i+1, \ldots, j-1\}$, $s_k \notin D_l(s, s_i)$, then s_j also lies in $D_l(s, s_k)$, $\forall k \in \{i, i+1, \ldots, j-1\}$

Intuitively, the lemma says that if s_j is the first site (in a counter clockwise sense) that kills (s, s_i) from the left, then s_j kills all $(s, s_k), i \leq k \leq j-1$, from the left. Figure 6.2 gives a brief idea of the lemma and the proof.

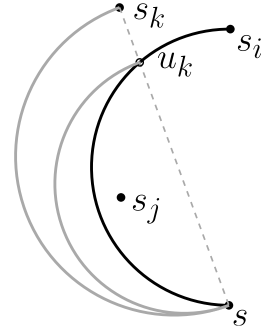


Figure 6.2: The site s_j lies in the part of $D_l(s, s_i)$ that is to the left of $s\vec{s}_k$. This part of $D_l(s, s_i)$ is a subset of $D_l(s, u_k)$, which itself is a subset of $D_l(s, s_k)$.

Proof. Since $i \leq k \leq j-1$ and $s_k \notin D_l(s, s_i)$, the segment $\overline{ss_k}$ intersects the circular part of the boundary of $D_l(s, s_i)$ at some point u_k. For every point $p \in D_l(s, u_k)$, angle $\angle sps_k \geq \angle spu_k > \pi/2$. Thus p also lies in $D_l(s, s_k)$. Hence $D_l(s, u_k) \subset D_l(s, s_k)$.

Let H be the closed half plane that lies on left of the line $s\vec{u}_k$. For every point $p \in D_l(s, s_i) \cap H$, angle $\angle spu_k \geq \angle sps_i > \pi/2$. Thus $(D_l(s, s_i) \cap H) \subset D_l(s, u_k)$. Using this with the subset relation from the previous paragraph we have $(D_l(s, s_i) \cap H) \subset D_l(s, s_k)$. Since s_j lies in $(D_l(s, s_i) \cap H)$ it also lies in $D_l(s, s_k)$. $\square$

Let L be the circularly doubly linked list of sites of $S \setminus \{s\}$ ordered counter clockwise around s. Algorithm 4 efficiently identifies sites $s' \in L$ such that the edge (s, s') is killed from the left by some site in L. Such edges are marked *dead* by the algorithm. A similar algorithm is used to identify the sites s'' such that the edge (s, s'') is killed from the right. The edges that are killed neither from left nor right belong to the Gabriel graph.

We now give a brief overview of Algorithm 4. Given a site $u \in L$, define `left_victims`(u) as a subset of $S \setminus \{s\}$ such that for each site $v \in$ `left_victims`(u), u is the first (when walking left along the list L) site to kill the edge (s, v) from left. Lemma 14 says that the set `left_victims`(u) is contained in a continuous sublist of L that starts on the right of u and only contains sites w such that (s, w) is killed from left by u. The lemma is used in the inner while loop of Algorithm 4 to find a sublist L_u such that: (a) each site in L_u has a killer in $(u \cup L_u)$; and (b) the union of the `left_victims` of the sites in $(u \cup L_u)$ is a subset of L_u. Due to (a), we know that the sites in L_u can be killed from left and hence they are marked $dead$. Due to (b), for any remaining site $v \in L \setminus L_u$, if the edge (s, v) is killed from the left then v belongs to the set of `left_victims` of some site in $L \setminus L_u$. Thus, to find the remaining `left_victims`, we process the smaller list $L \setminus L_u$.

Testing if $killer \in D_l(s, current)$ uses degree 2 predicates `isKiller` and `Orientation`, thus, the above algorithm runs in $O(|L|)$ time and is degree 2.

Lemma 15. Given the circular ordering of $S \setminus \{s\}$ around s, in $O(n)$ time and degree 2, we can find the Gabriel edges incident at s.

For completeness, we describe the three steps for constructing the Gabriel graph of a set of n

Algorithm 4 `KillFromLeft`(L, s): Give a set of sites L circularly ordered around s Delete the sites of L that are killed from the left.

```
1:  Initialize the unseen values of each site in L to true
2:  Initialize the dead values of each site in L to false
3:  u = any site in L
4:  // Invar: Any site in L killed from left is in the set of left_victims of another site in L.
5:  while u→unseen do
6:     u →unseen = false
7:     killer = u
8:     current = u → right
9:     // The while loop computes L_u
10:    // Invar: Sites right of u & left of cur are killed from left by u or a site between u and cur.
11:    // Invar: All left_victims of sites between u and killer are found.
12:    while killer≠current do
13:       if killer ∈ D_l(s, current) then
14:          current →dead = true
15:          current = current → right
16:       else
17:          killer = killer → right
18:       end if
19:    end while
20:    L_u = the sublist of L, that is to the right of u and left of current
21:    Delete L_u from L
22:    u = u →left
23: end while
```

sites S with degree 2. First, compute $A(S^*)$, which by Lemma 12 takes $O(n^2)$ time and degree 2. Second, for each site $s_i \in S$ compute the circular ordering of the sites of $S \setminus \{s_i\}$, which in total, by Lemma 13, takes $O(n^2)$ and degree 0. Third, for each site $s_i \in S$, use the circular orderings to compute the set of Gabriel edges in which s_i is a member, which in total, by Lemma 15, takes $O(n^2)$ and degree 2.

Corollary 16. We can compute the Gabriel graph in $O(n^2)$ time using degree 2.

6.4 Conclusion

In this chapter, we showed how to construct the Gabriel graph of n sites in $O(n^2)$ time and degree 2. Even though an $O(n^2)$ construction for the Gabriel graph is too slow for practical applications, Corollary 16 tells us that we can at least compute the Gabriel graph with degree 2 and do better than brute force. In contrast, we simply cannot compute the Delaunay triangulation with degree 2. A natural next question (not addressed here) is can we compute the Gabriel graph in sub-quadratic time with degree 2?

Chapter 7

Point Location

This chapter follows Liotta, Preparata, and Tamassia in modifying point location structures for proximity queries so that they correctly perform queries from the grid $\mathbb{U}$ in degree 2. Queries from off the grid need not be answered correctly. Liotta, Preparata, and Tamassia built their structure by rounding the results of a degree 5 computation of a Voronoi diagram; I give two new structures that can be built more directly by incremental construction of degree 3 and of degree 2. The degree 3 construction is efficient as a randomized incremental construction (RIC). The degree 2 construction, despite our earlier claim [100] to the contrary, does not fit the usual RIC analysis, because we cannot give a constant bound on the number of sites to define a rightmost grid point of a cell. We conjecture that it is $\Theta(\log U)$.

7.1 Introduction

Computing the topological structure of the Voronoi diagram of n sites requires degree 4. We saw in Chapter 4 that Voronoi vertices of sites with integer coordinates have rational coordinates of degree 3 over 2, and that testing whether a query point is above or below a segment joining two Voronoi vertices is degree 6.

Liotta *et al.* [89] developed an *implicit Voronoi diagram* to answer Post Office queries for n

sites on a $U \times U$ grid – finding the closest site to a query in $O(\log n)$ time and $O(n)$ space using only degree 2. Unfortunately, their algorithm must compute the entire Voronoi diagram before rounding it to their structure. They also sort oronoi vertices by y coordinate, which requires degree 5.

In this section we introduce the *Reduced Precision Voronoi diagram*, or RP-Voronoi for short, that supports proximity queries, but is computed incrementally using at most triple precision in $O(n \log(nU))$ expected time. From our structure we can obtain the structure of Liotta *et al.* in linear time. A natural next question is, can we use only degree 2 to build a point location query structure? We show that it can be done by introducing a structure, called the *D2-Voronoi*, and provide an algorithm for its construction. We conjecture the time and space bounds for the construction. It seems unlikely that one could obtain Liotta *et al.*'s structure with only degree 2.

To appreciate the challenge in reducing the construction to triple and double precision, note that most algorithms compute Voronoi diagrams by computing the Delaunay triangulation, which uses the degree 4 `InCircle` test to verify that three points define an empty circle [53]. Our algorithms compute the closest sites only at grid points, which is not enough information to obtain the dual Delaunay triangulation.

Fortune's sweep algorithm [43], computes the Voronoi directly, but uses a degree 6 predicate that orders two circumcircles by their extreme points. The notion of abstract Voronoi diagrams [82] reduces Voronoi computation to ordering bisectors around Voronoi vertices, which again uses `InCircle`. With degree 3, however, we can still order bisectors around a grid cell, which is essential to our construction. With degree 2 we cannot even determine this order. In fact, we cannot even determine if two Voronoi cells are adjacent, which we see in Section 7.4 adds complexity to a degree 2 construction.

7.2 Definitions and Notation

The Voronoi diagram is well known in computational geometry and many optimal algorithms have been proposed [31, 43, 54, 121]. Most are designed for a RealRAM or another computational model in which coordinate computations may be carried out to arbitrary precision, allowing the computer to work with exact Euclidean geometry. As we are concerned with the precision of input, we start with a restricted definition of the Voronoi diagram and remind the reader of some of its properties.

7.2.1 The Voronoi Diagram and Grids

Assume that we are given a set of n *sites*, $S = \{s_1, s_2, \ldots, s_n\} \subset \mathbb{U}$. By assumption, S lay in a bounded rectangle in the integer grid. The distance metric is Euclidean.

The Voronoi diagram, $\mathrm{VoD}(S)$, is the partition of our bounded rectangle into maximally connected regions with the same set of closest sites. Regions with one closest site are called *Voronoi regions*, regions with two closest sites are called *Voronoi edges*, and regions with three or more closest sites are called *Voronoi vertices*. The Voronoi cell $\mathrm{VR}_S(s_i)$ is the closure of the region of s_i; that is, $\mathrm{VR}_S(s_i) = \{x \in \mathbb{R}^2 \mid \|x - s_i\| \leq \|x - s_j\|,\ \forall s_j \in S\}$. Note that we suppress the S and write $\mathrm{VR}(s_i)$ when S is clear from the context.

Let B_{ij} denote the locus of points equidistant to s_i and s_j, the perpendicular bisector of the segment $\overline{s_i s_j}$. Note that as we are interested in a particular region, we can clip bisectors and Voronoi edges to finite segments. Building a point location structure on top of the Voronoi diagram gives the classical solution to the *Post office problem*; after preprocessing the sites, you can determine the closest site to a query point, q, in $O(\log n)$ time.

Note that a line bisecting two grid points might not hit any points of the grid $\mathbb{U}$. In building our structures, it is convenient to use the half-integer grid $\mathbb{U}_2 = \frac{1}{2}[2U+1]^2$ and the continuous square $\mathcal{U} = [1/2, U + 1/2]^2 \subset \mathbb{R}^2$. That way, at least the midpoint between two sites $(s_i + s_j)/2$ is on

the half-integer grid. In this section, we abuse definitions and say that a point is *single precision* to mean that it is on the grid $\mathbb{U}_2$.

In Chapter 4 we derived the degree of many constructions and predicates. Here, we recall one basic construction and four predicates for Voronoi diagrams.

`VoronoiVertex`(a, b, c)**:** A Voronoi vertex constructed from grid point sites $a, b, c \in \mathbb{U}$ has coordinates of degree 3 over degree 2.

`PointOrdering`(a, b)**:** Compare points by lexicographic order, so that a precedes b if and only if $a_x < b_x$ or ($a_x = b_x$ and $a_y < b_y$). Comparing grid points $a, b \in \mathbb{U}_2$ is degree 1, comparing a grid point with a Voronoi vertex is degree 3, and comparing two Voronoi vertices is degree 5. When a precedes b we may write $a \prec b$.

`Closer`(s_i, s_j, q) **or** `SideOfABisector`(B_{ij}, q)**:** A *bisector* $B_{ij} = \{p \in \mathbb{R}^2 : \|p - s_i\| = \|p - s_j\|\}$ is the line equidistant to two sites. Determining if a point q is on a bisector, or determining the closer of the two sites, is degree two—simply compare squared distances.

`InCircle`(a, b, c, q)**:** The basic predicate for Voronoi computation [31], `InCircle` determines whether q is inside the circumcircle of $a, b, c \in \mathbb{U}$. This could be done by comparing the squared distances from `VoronoiVertex`(a, b, c) to a and to q, which gives a degree 6 polynomial when you clear fractions, but that polynomial can factor to give the `InCircle` predicate, which is degree 4.

`Orientation`(a, b, c)**:** The orientation predicate, which reports if the path a, b, c makes a right turn, left turn, or goes straight, is a determinant on the homogeneous coordinates of the inputs. It is a degree 2 test on grid points, which is the only way we will use it. Degree goes up by 2 for each Voronoi vertex: it takes degree 8 to determine orientation for three arbitrary Voronoi vertices, and degree-6 to determine the orientation of a grid point relative to an edge of a triangulation of a Voronoi diagram.

Aurenhammer [4] surveys properties of the Voronoi diagram under the Euclidean metric. We

use the following:

- Bisectors are straight lines.
- A Voronoi edge on a cell boundary $\mathrm{VR}(s_i)$ and $\mathrm{VR}(s_j)$ lies on the bisector of s_i and s_j.
- A Voronoi cell is the intersection of closed half planes; thus, Voronoi cells are convex.
- A site s_i is contained in its cell, $s_i \in \mathrm{VR}(s_i)$.

The following properties of the Voronoi diagram are also known, but we state them in a form that will be helpful for our constructions.

Lemma 17. The order in which Voronoi cells intersect a line ℓ is the same as the order of the corresponding sites orthogonal projection onto ℓ.

For our degree 3 incremental construction we will need to decide if a new cell intersects a horizontal or a vertical segment. A corollary of Lemma 17 will give us a convenient way of making this decision.

Corollary 18. Without loss of generality, consider a horizontal segment σ and the set of sites $\mathcal{S}$ whose Voronoi cells intersect σ. Let q be a new site, and let s_i, s_j be the sites of S whose projections onto σ form the smallest interval containing the projection of q; site s_i or s_j can be taken as infinite if no finite interval exists. To determine if $\mathrm{VR}_{\mathcal{S}}(q)$ intersects σ, it suffices to test if an endpoint of σ or the intersection of $\sigma \cap B_{ij}$ is closer to q than to both s_i and s_j.

Proof. Assume that q is above σ and that we would like to determine if $\mathrm{VR}_{\mathcal{S}}(q)$ appears below σ. Let $x_i < x_j$, and $c_i, c_j \in \sigma$ be points in the cells of s_i and s_j respectively. Consider the point q' that has the same x coordinate as q, but is raised to infinity. Now, lower q' continuously, computing the Voronoi diagram of $\sigma \cup \{q'\}$ until the cell of q' intersects σ, and let $c_{q'}$ be this intersection point. Lemma 17 tells us that $c_i < c_{q'} < c_j$. In addition, $c_{q'}$ must be on B_{ij}, otherwise $c_{q'}$ would be in the middle of the cell of s_i or s_j, causing the cell to be non-convex. In fact, $c_{q'}$ is the point equidistant

to s_i, q' and s_j. Now, if q' is above q then the point equidistant to s_i, q and s_j must be below $c_{q'}$ so the cell of q must intersect σ. Alternatively, if q' is below q then the point equidistant to s_i, q and s_j is above $c_{q'}$ so the cell of q is completely above σ and therefore, their intersection is empty. □

Next, we see that a Voronoi diagram intersected with a convex region free of sites is a collection of trees.

Lemma 19. Given a set of sites S and a convex region R containing no sites of S; the edges and vertices of the $\mathrm{VoD}(S)$ in R form a forest.

Proof. If $R \cap \mathrm{VoD}(S)$ contained a cycle then R would contain a Voronoi cell, and therefore a site. Since R contains no sites, we conclude that $R \cap \mathrm{VoD}(S)$ does not contain a cycle. □

7.2.2 Trapezoidation and Point Location

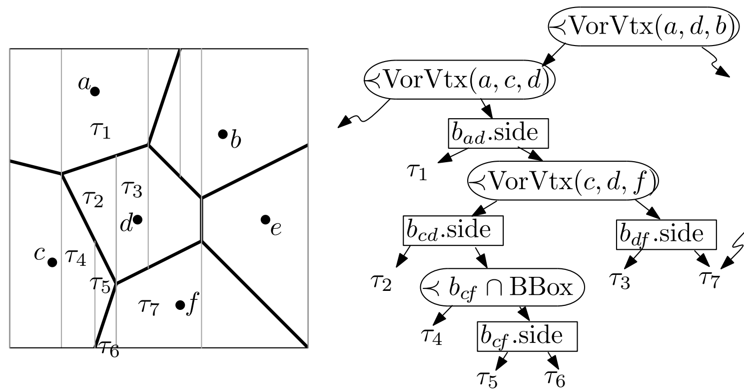


Figure 7.1: Trapezoidation of a small Voronoi diagram, and a piece of the trapezoid graph for the strip with labeled faces, $\tau_1, \ldots, \tau_7$.

Any planar subdivision with line segments can be decomposed into trapezoids by making vertical cuts at the endpoints of each segment that extend until reaching another segment. We define three different, but related, trapezoidations in Section 7.4.1. Each trapezoid τ will have *top* and *bottom* lines, and a *left* and *right* point, with $\tau.left \prec \tau.right$ and both points on or between the

top and *bottom* lines. Each trapezoid may have up to four neighbors, reached by crossing the vertical lines above or below $\tau.left$ and $\tau.right$, although if the subdivision is into monotone regions, as in Figure 7.1, then each trapezoid has at most two neighbors. When $\tau.left$ and $\tau.right$ have the same x coordinate, we can still think of τ as a trapezoid since the lexicographic order on points used by $\prec$ is consistent with skewing the points slightly, or rotating the vertical direction counter-clockwise. For example, the bisector of d and e in Figure 7.1 is the bottom edge for an infinitesimally thin trapezoid in the Voronoi cell of d and the top edge for an infinitesimally thin trapezoid in the Voronoi cell of e.

Trapezoid graphs [31, ch. 6] support point location queries, which identify the trapezoid containing a query point q. The *trapezoid graph* is a directed acyclic graph (DAG) with three types of nodes:

x**-node:** Stores a point v; Uses `PointOrdering` to check if a query $q \prec v$. If both q and v are grid points on $\mathbb{U}_2$, this test is degree 1; if v is a Voronoi vertex, this test is degree 3.

y**-node:** Given a query point q and a line ℓ, determine if q is above, below, or on ℓ. If ℓ is a bisector of two sites, or if ℓ is defined by two grid points on $\mathbb{U}$, then this test is degree 2. On the other hand, if ℓ is defined by two arbitrary Voronoi vertices, this test is degree 6.

leaf node: Leaf nodes represent trapezoids, for which the containing region is known.

Figure 7.1 shows a portion of the trapezoid graph for seven trapezoids in a small Voronoi diagram.

Liotta, Preparata, and Tamassia [89] achieved a degree-two post-office query by building, what they called the *implicit Voronoi diagram*. The implicit Voronoi diagram is a rounded trapezoid graph for the Voronoi diagram. They build the trapezoid graph in the straightforward way, depicted in Figure 7.1, which is to use Voronoi vertices (and intersections of bisectors with the bounding box) at x-nodes and bisectors at y nodes. Liotta *et al.* improve this by two simple observation: first, since a Voronoi edge between $\mathrm{VR}(s_i)$ and $\mathrm{VR}(s_j)$ lay on B_{ij} one can use `SideOfABisector` to get a degree 2 y-node test. Second, since the queries are grid points, rounding non-integer x

nodes to half-integers reduces the degree of that test from 3 to 1 without changing any test results. They still used a degree 5 algorithm to compute the Voronoi diagram and compare y-coordinates of Voronoi vertices when building the trapezoidation.

7.3 Computing Point Location Structures with Triple Precision

In the following sections we define and show how to construct the reduced precision Voronoi diagram using degree 3. Before going into the details, we will need one more construction based on the `OrderOnLine` predicate, defined in Chapter 4:

`OrderOnLine`(B_{12}, B_{34}, ℓ)**:** determines whether two non-vertical bisectors B_{12} and B_{34} intersect a vertical line ℓ with B_{12} above, below or at the same point as B_{34} on ℓ and is degree 3.

We use the `OrderOnLine` predicate in a construction that allows us to identify the grid cells containing a Voronoi vertex. Observe that given bisector B_{12} and a non-vertical segment $\sigma \subset B_{34}$ with left and right endpoints on horizontal gridlines, ℓ_l and ℓ_r, we can determine if σ intersects B_{12}. The order of B_{12} and B_{34} on ℓ_l and ℓ_r is different if and only if B_{12} intersects σ. Thus, two calls to `OrderOnLine` suffice. Given that B_{12} and σ intersect, we can binary search for the grid cell containing the intersection in $O(\log U)$ and degree 3. This construction is important so let's name it:

Lemma 20. Given two bisectors B_{12} and B_{34} of sites from $\mathbb{U}$ and two grid lines ℓ_l and ℓ_r, the construction `GridCell`$(B_{12}, B_{34}, \ell_l, \ell_r)$ returns if B_{12} and B_{34} intersects between ℓ_l and ℓ_r in constant time and degree 3. When they do intersect between ℓ_l and ℓ_r, the construction returns the grid cell containing the intersection $O(\log U)$ time and degree 3.

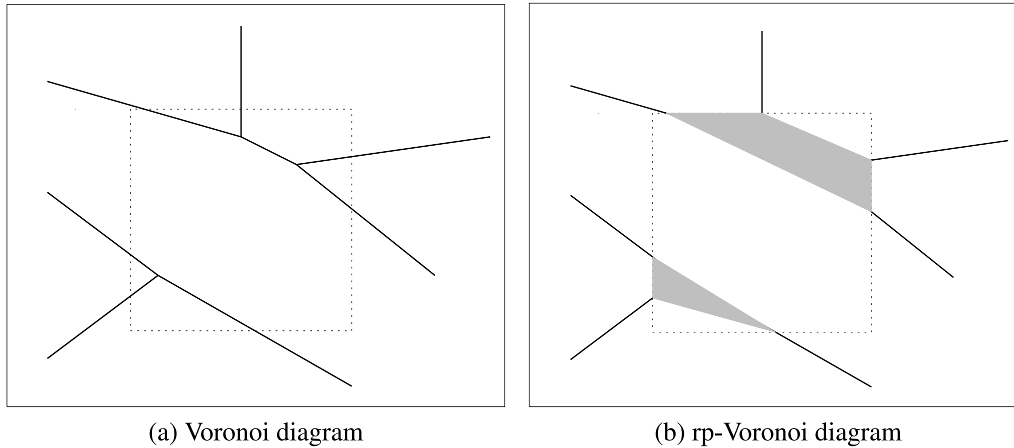
7.3.1 The Reduced-Precision Voronoi Diagram

Given a set of n sites $S = \{s_1, s_2, \ldots, s_n\}$ with degree 1 coordinates, we define a reduced-precision Voronoi diagram, or rp-Voronoi for short, that is intermediate between the true Voronoi diagram $\mathrm{VoD}(S)$ and the implicit diagram of Liotta *et al.* [89]. Because we use predicates of at most degree three, we cannot know exactly how bisectors intersect inside of a grid cell. The predicates of the previous section, however, do provide full information at the grid cell borders. This low level of information inside of a grid cell gives us a "fuzzy" picture of Voronoi vertices that we contract to *rp-vertices*. Since we do, however, know precise information at the grid boundaries we maintain *rp-edges* that keep the same edge ordering as Voronoi edges entering the grid cell. In this way we keep enough control of the Voronoi vertices to perform constructions efficiently; in contrast, the implicit diagram rounds Voronoi vertices off their defining edges.

Let us consider the integer grid $\mathbb{U}$ as a partition into *grid cells* of the form $[i, i+1) \times [j, j+1)$ for integers i, j. The *rp-Voronoi* $\widehat{\mathrm{V}}(S)$ is the graph with *rp-vertices* and *rp-edges* defined by contracting every edge of the Voronoi diagram $\mathrm{VoD}(S)$ that lies entirely inside some grid cell.

Figure 7.2(a) depicts and example grid cell $G \in \mathbb{U}$ and shows the intersection $G \cap \mathrm{VoD}(S)$, which by Lemma 19 is a forest. In the graph structure of the rp-Voronoi, $\widehat{\mathrm{V}}(S)$, we therefore contract each tree of the forest to an rp-vertex. Edges that leave the grid cell are preserved as rp-edges. Notice that the planar embedding of the Voronoi $\mathrm{VoD}(S)$ gives a natural planar embedding of $\widehat{\mathrm{V}}(S)$ in which the ordering of edges entering a grid cell is preserved as the ordering of edges around the rp-vertex. We find it useful to depict these rp-vertices as the convex hulls of the intersections of rp-edges, as in Figure 7.2(b). Although we never actually compute these convex hulls they bound the locations where the tree of $\mathrm{VoD}(S)$ can lie.

Each rp-vertex v maintains the grid cell G_v containing v, and a list of its incident rp-edges in counter-clockwise order by their entry to G_v. Each rp-edge e stores the generator sites s_1 and s_2 of the corresponding Voronoi diagram edge, and pointers to its location in the lists of its two



(a) Voronoi diagram (b) rp-Voronoi diagram

Figure 7.2: The Voronoi vertices in a grid cell on the left contract to two rp-vertices, depicted as gray convex polygons on the right. The ordering of the edges entering the grid cell is maintained in both diagrams.

rp-vertices. Standard data structures, like the doubly-connected edge list [31] or quad-edge [53], allow us to maintain the order in the planar subdivision represented by $\widehat{\mathrm{V}}(S)$.

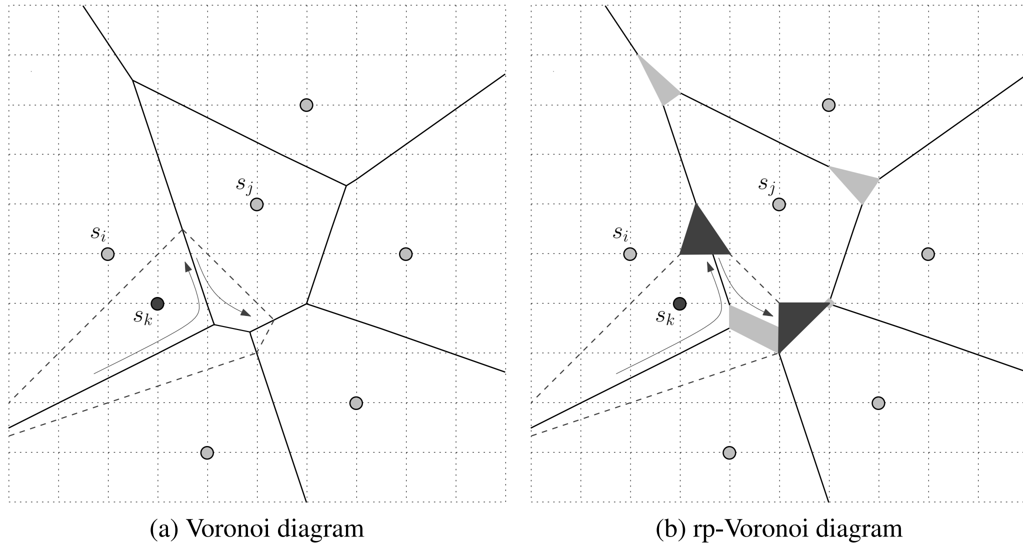
Finally we define the *boundary of the rp-region* of s to be the alternating sequence of rp-edges storing site s and the connecting rp-vertices that form a cycle. We call the *rp-region* all the points enclosed in this cycle, and the *rp-cell* the union of the rp-region with its boundary.

Observation 21. The number of rp-vertices and rp-edges in the rp-Voronoi diagram of S is less than or equal to the number of Voronoi vertices and Voronoi edges in the Voronoi diagram of S respectively.

As we will show in Section 7.3.2, we can retrieve the implicit Voronoi diagram once we have constructed the rp-Voronoi.

7.3.2 Constructing the Reduced-Precision Voronoi Diagram

Next we describe how to construct the rp-Voronoi, analyze the expected time and space, describe how to use the rp-Voronoi for point location and show how to convert the rp-Voronoi to the implicit Voronoi diagram.



(a) Voronoi diagram (b) rp-Voronoi diagram

Figure 7.3: The cell for the new dark gray site s_k is "carved" out of the diagram of light gray sites. The traced bisectors are emphasized with dotted lines, and the tree walk is shown with gray arrows.

We create the rp-Voronoi by a randomized incremental construction [31] that parallels Sugihara and Iri's algorithm [130]: inserting a new site by "carving" out the new cell from the previous diagram. Inserting a new site invalidates a sub-graph of the Voronoi diagram, referred to as the *conflict region*. Sugihara and Iri made the observation that the conflict region is a tree, and that by walking the tree we identify the invalid sub-graph.

Specifically, their method constructs a Voronoi diagram of the first $k-1$ sites and then inserts site s_k. To start carving, the site s_i closest to s_k is identified and the bisector B_{ik} is traced until it enters the neighboring Voronoi cell, $\mathrm{VR}(s_j)$. The bisector B_{jk} is then traced, and the process continues until it returns back to $\mathrm{VR}(s_i)$. The tracing process requires the identification of the next bisector intersection with a Voronoi edge. Sugihara and Iri do this by walking around the cell of s_j on the side of the new site s_k until the next intersection is found (see Figure 7.3a).

Our method does the same computation, but since we restrict ourselves to degree three, it is too costly to compute and compare coordinates of bisector intersections.

Incremental Construction

We initialize the rp-Voronoi diagram with two sites s_1 and s_2, and use their bisector B_{12} to split the initial region of interest (the grid) by using a binary search.

Now, assume that we have already constructed the rp-Voronoi of $k-1$ sites S_{k-1} and that we would like to insert site s_k. The *rp-Voronoi Update Procedure* takes as input the rp-Voronoi of S_{k-1} and a new input site s_k and returns the rp-Voronoi of $S_{k-1} \cup \{s_k\}$.

rp-Voronoi Update Procedure: We sketch the procedure in this paragraph and then fill in the details in the remainder of the section. We first locate the site $s_i \in S_{k-1}$ closest to s_k, and proceed in two steps. We find the subgraph T that consists of the set of rp-vertices and rp-edges that are no longer part of the rp-Voronoi of $S_{k-1} \cup \{s_k\}$. In the Voronoi diagram, the conflict region is a tree and the sum of all conflict region sizes is linear in expectation. In the rp-Voronoi we walk a subset T of the edges of this tree, and their vertices; once we have identified this subset, we maintain our data structure in time proportional to its size, which is therefore also linear (see Figure 7.3b).

To identify T we start by tracing out the s_i, s_k bisector B_{ik}. We walk around the boundary of the region of s_i until we find the grid cell G containing the intersection of B_{ik} and the boundary of the rp-region of s_i. As in Sugihara and Iri's algorithm, we would like to pick the next bisector to trace, thus, allowing us to continue our tree walk. To *pick the next bisector* for the rp-Voronoi with limited precision there are two cases: one simple and the other interesting.

In the simple case, B_{ik} intersects the rp-edge e that stores sites s_i and s_j. This intersection is determined by applying the `GridCell` construction. We switch to the s_k, s_j bisector and continue building T by walking around the boundary of the region of s_j on the side of s_k.

In the more interesting case, B_{ik} intersects an rp-vertex v. The intersection is determined by first checking if B_{ik} passes through the grid cell containing v. If it does, we compare the order of B_{ik} and the two bisectors defining the edges incident on v of the rp-cell of s_k.

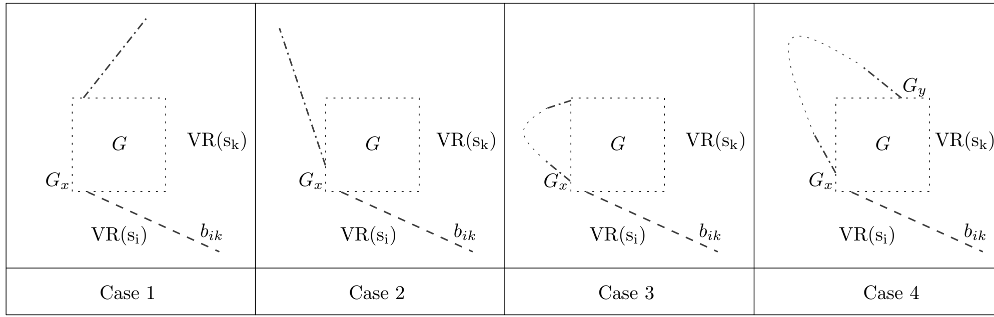


Figure 7.5: Grid cell G with grid walls G_x and G_y as west and north walls respectively. A bisector enters G through the south wall by tracing the s_i, s_k bisector B_{ik}, in dashed gray. In alternating dashed and dotted gray are the four cases per wall for bisector tracing.

Let G_N, G_E, G_S and G_W be the north, east, south and west grid walls, respectively, of G, and without loss of generality, assume that we have entered G from the south, with s_i below the B_{ik} bisector (see Figure 7.4).

Since Voronoi cells are convex, the new cell of s_k can intersect each of the grid cell boundary walls at most twice. This gives us four cases for how the traced bisectors of the new Voronoi cell enter and exit a grid wall G_x of G (see Figure 7.5). Traced bisectors of the new Voronoi cell,

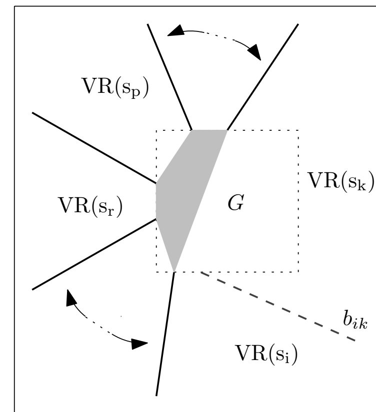


Figure 7.4: We enter grid cell G from the south while walking the tree of the new site s_k along the s_i, s_k bisector B_{ik} in dashed gray. The projections of sites s_p and s_r onto the west grid line are directly above and below the projection of s_k onto the west grid line respectively.

$(c1)$ do not exit through G_x.

$(c2)$ exit through G_x and do not return to G.

$(c3)$ exit through G_x and return through G_x.

$(c4)$ exit through G_x and return through a different grid wall G_y.

First, we determine if the Voronoi cell $\mathrm{VR}(s_i)$ pokes out of the G_W grid wall. We find the two sites s_p and s_r whose Voronoi cells intersect G_W and whose y coordinate is directly above and

below the y coordinate of s_k, respectively. We then determine if $\mathrm{VR}(s_i)$ pokes out by applying Corollary 18.

If $\mathrm{VR}(s_k)$ does not poke out of G_W (case 1) we repeat the process with G_N, followed by G_E. If $\mathrm{VR}(s_k)$ does poke out then there are some points in the $\mathrm{VR}(s_r)$ that are now in the $\mathrm{VR}(s_k)$. Since no site has an empty cell there must be a Voronoi edge $e \in \mathrm{VR}(s_k)$ that is a subset of the s_r, s_k bisector. We trace B_{rk}, following the tree, towards B_{ik} until we return back to G. Now, we have identified the next bisector to trace for our tree walk. In addition, we just walked backwards through a subtree of T, and we continue the procedure by tracing B_{rk} in the opposite direction as before.

The other three cases are determined by continuing the walk. Case 3 corresponds to the walk returning back to the grid cell through the same cell wall it exited. Case 4 occurs when the walk returns back to the grid cell, but through a different grid wall. This allows us to determine cases 3 and 4 that can cause multiple rp-vertices to occur in one grid cell.

We continue this process until we have completed the cycle, identified T and the new rp-vertices and rp-edges. We update the rp-Voronoi of S_{k-1} to get the rp-Voronoi of $S_{k-1} \cup \{s_k\}$.

Note that if a Voronoi vertex v is outside our region of interest we do not need to identify the grid cell containing v since it will not be used for proximity queries. However, to continue with our tree walk we can apply Corollary 18 similar to the case where a bisector intersects an rp-vertex.

Analysis

Point location is accomplished by the standard method of maintaining the construction history [54] allowing for point location in expected $O(\log n)$ time. To achieve a degree two algorithm we use grid cells in x-nodes and `SideOfABisector` for y-nodes, much like the structure in [89]. The incremental construction described above relies on `SideOfABisector`, which we saw in Section 7.2 was degree 2, and `GridCell`, which we saw in Lemma 20 was degree 3.

As explained by Observation 21, the rp-Voronoi and Voronoi diagram have the same combinatorial complexity. The update procedure creates at most as many rp-vertices as Voronoi vertices. As shown by [31, 54] the number of Voronoi vertices created is expected linear throughout the algorithm. Furthermore, the tree walk touches only edges that are modified, and the number of modified edges is constant in expectation [54].

However, we must pay two additional charges in each update. First there is an extra $O(\log U)$ charge for finding bisector segment intersections. Secondly, we must pay an $O(\log n)$ to find the upper and lower neighbors when a bisector intersects an rp-vertex. So we have shown the expected time to insert a new site into the reduced-precision Voronoi diagram of size n is $O(\log n + \log U)$ and that this insertion can be done with degree 3. We conclude:

Theorem 22. We can construct a reduced-precision Voronoi diagram of n sites laying on a $U \times U$ grid with a degree three algorithm in expected $O(n \log(Un))$ time.

Reduced-precision Voronoi to Implicit Voronoi

Next we describe how to convert the reduced-precision Voronoi to the implicit Voronoi of [89], described in Section 7.3.1. An rp-vertex corresponds to a tree T, of Voronoi vertices and Voronoi edges. Some of the Voronoi vertices of T may be on grid lines and the implicit Voronoi would assign these vertices integer coordinates. To create the implicit Voronoi we must separate these vertices from an rp-vertex.

Theorem 23. We can convert the reduced-precision Voronoi diagram to the implicit Voronoi diagram in $O(n)$ time with degree three.

Proof. We say that rp-edges e_{12} and e_{34} separating the s_1, s_2 and s_3, s_4 rp-regions respectively, have a *common intersection* if e_{12} and e_{34} intersect ℓ, and B_{12}, B_{34} and ℓ all pass through the same point. For each rp-vertex v we perform the following procedure. First, we retrieve the grid cell G containing v. Next, we walk around the rp-edges $E_v = \{e_1, e_2, \ldots, e_k\}$ incident on v and group

the edges of E_v that have a common intersection. Each group is labeled E_i where i is the lowest number of the edge set. Without loss of generality assume that no E_i contains e_1 and e_k unless it contains $\{e_1, \ldots, e_k\}$. Since only adjacent rp-edges can intersect on grid lines if there are m edges in E_i then $E_i = \{e_i, \ldots, e_{i+m-1}\}$.

Next, we remove each edge set E_i from v and attach them to a new vertex v_i'. This creates a "gap" in v. We create edge $e = (v_i', v)$ and insert e between the first and last edge of E_i for v_i', and in the "gap" of v. We finalize the construction of e by storing its generator sites.

Now, we assign geometric information to v_i' where the integral component of v_i' is computed similar to the `OrderOnLine`, and we snap non-integral component to the half grid point. Once this process is complete all Voronoi vertices of T that are on grid lines are separated and assigned the same geometric information as the implicit Voronoi diagram. Finally we set the geometric information of v by snapping both components of v to the half grid point contained in G.

Since the number of vertices is linear in n for the Voronoi, the rp-Voronoi, and the implicit Voronoi diagrams, and each edge is considered a constant number of times to compute a single intersection with a grid line, this process takes $O(n)$ time. Furthermore, each of the computations of the procedure are derived from the `OrderOnLine` predicate. Thus, this algorithm is degree three. □

From Theorem 22 and 23 we conclude,

Corollary 24. The implicit Voronoi diagram of n sites can be constructed in expected $O(n \log(Un))$ time with degree three.

7.4 Computing Point Location Structures with Double Precision

In the previous section we saw how to build a point location structure using degree 3. In this section, we define a point location query data structure that can be computed with degree 2. The section concludes with some conjectures on the time and space complexity of the construction. Despite our earlier claim [100], the construction does not fit the usual RIC analysis, because we cannot give a constant bound on the number of sites to define a rightmost grid point of a cell. We conjecture that it is $\Theta(\log U)$. Without the bound, however, the construction is $O(n^2 \log U)$.

To appreciate the challenge in reducing the construction to degree 2 note that both Sugihara and Iri's incremental construction [130], which avoids geometric tests in favor of topological inference (but still relies on the degree 4 `InCircle` predicate) and our RP-Voronoi construction, from the previous section (which uses degree 3), are built incrementally. Both algorithms must walk through the current diagram. Walking does not seem to work without a notion of adjacency of cells, which seems difficult to achieve with only degree 2.

Moreover, with degree 2 we cannot even determine the grid cell containing a Voronoi vertex. For the RP-Voronoi we determined the grid cell containing a Voronoi vertex by identifying the grid cell in which bisectors changed order. Unfortunately, ordering bisectors along vertical and horizontal grid lines is degree 3. Figure 7.6 depicts three view of the intersection of two bisectors in Real-RAM, and when limited to degree 3 and degree 2 predicates.

In this section, we define a trapezoidation of approximate Voronoi cells that is suitable for incremental construction and use a history DAG (directed acyclic graph) rather than a walk to find which trapezoids need to be updated when a new site is inserted. We develop a new implicit Voronoi diagram structure that we are able to compute by an incremental construction using only double precision.

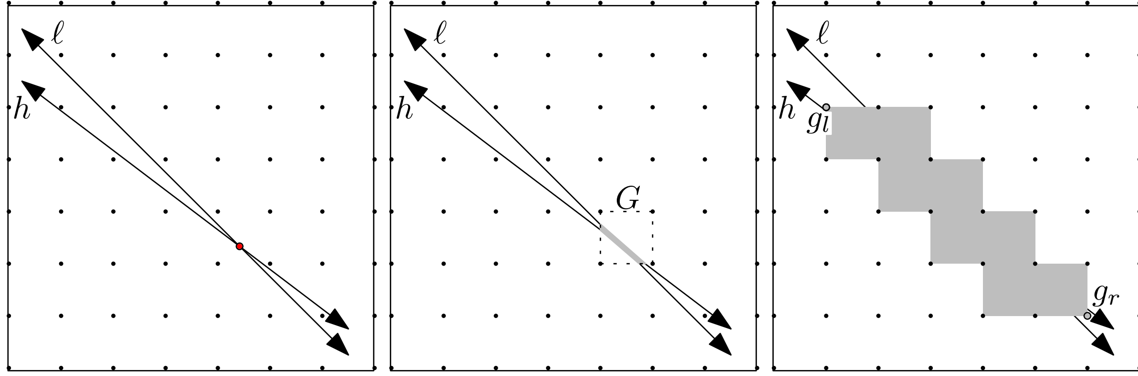


Figure 7.6: Three views of the intersection of two bisectors ℓ and h. *Left:* Depicts the intersection of ℓ and h in Real-RAM. We can think of the intersection point as the red marker on ℓ and h. *Middle:* Depicts the intersection of ℓ and h when restricted to degree 3. We can determine the grid cell G containing the intersection and bisector orderings around the grid cell. We can think of the intersection of ℓ and h as somewhere in the gray polygon in G. *Right:* Depicts the intersection of ℓ and h when restricted to degree 2. We can determine which side of a bisector a grid point lies and the set of grid cells that a bisector passes though. That tells us that ℓ is above (and h is below) grid point g_l, that ℓ is below (and h is above) grid point g_r, and that the intersection is somewhere in between g_l and g_r. We can think of the intersection of ℓ and h as somewhere in union of the grey boxes.

7.4.1 Definitions and Notation

Next we described the basic primitives for computing a degree 2 Voronoi diagram and three separate but related trapezoidations that will be helpful in the construction.

Voronoi Polygons on the Grid and Proxy Trapezoidation

Define the *Voronoi polygon* $C_S(s_i)$ to be the convex hull of the grid points of $\mathbb{U}_2$ in the Voronoi cell of site $s_i \in S$. We omit the subscript S when the set of sites is understood. Voronoi polygon $C(s_i)$ is contained in the Voronoi cell of s_i, and the containment may be proper. As illustrated in Figure 7.7, the set of all Voronoi polygons may leave *gaps* in $\mathcal{U}$, even though they cover all the grid points.

Define the *cell proxy* of s_i, denoted $P_S(s_i)$, to be the line segment from the minimum to the maximum point of $C_S(s_i)$, as illustrated in the middle of Figure 7.7. Because the n Voronoi

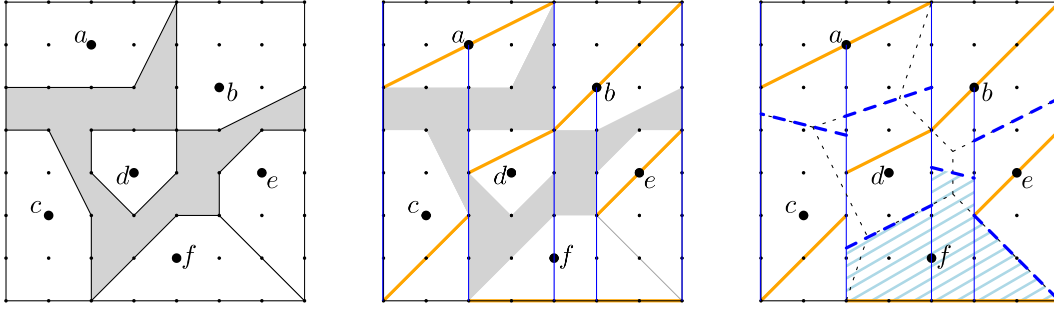


Figure 7.7: *Left:* Voronoi polygons of sites on a grid may leave gaps. *Middle:* Proxy trapezoidation of proxy segments, which connect the maximal and minimal points in a Voronoi polygon. The lexicographic order for points implies that grid points that appear on the vertical segments are actually in trapezoids to the left or right, unless they are proxy segment endpoints. Thus, f is in the trapezoid that contains d, a is in the trapezoid above the proxy for d, and there is an infinitesimally thin trapezoid between the endpoints of the proxies for c and d. *Right:* Splitting trapezoids of the proxy trapezoidation by bisectors gives the Voronoi trapezoidation.

polygons can actually have a total of $\Theta(n \log U)$ sides, we will often prefer to work with the n proxy segments instead. Define the *proxy trapezoidation* to be the vertical visibility map of all cell proxies, as shown in the same figure. Note that the proxy trapezoidation for a set of n sites S has at most $3n + 1$ trapezoids, and is canonical—it is completely determined by the sites.

In [24], we observed that if we can compute a proxy trapezoidation, then we can use it to answer post office queries using degree 2, because only the sites contributing the top and bottom proxies are candidates for the closest sites to the grid points inside a proxy trapezoid. (Note that the only grid point actually on a vertical side is the endpoint of a proxy segment that defines the side, and we already know the neighbor of a proxy – the lexicographic ordering used by $\prec$ implicitly perturbs grid points directly above a defining endpoint into the trapezoid to the right, and directly below a defining endpoint into the trapezoid on the left.)

Lemma 25. If $q \in \mathbb{U}$ is a query point in a trapezoid τ with $\tau.top = P(s_i)$ and $\tau.bottom = P(s_j)$ then q is in $C(s_i)$ or $C(s_j)$.

Proof. Suppose that some grid point $q \in \tau$ belongs in a different Voronoi polygon $C(s_k)$. Since the Voronoi polygons are convex and non-overlapping, the cell proxy $P(s_k)$ must pass through

trapezoid τ, which is a contradiction. □

Corollary 26. Given cell proxies $P_S(s_i)$ for all $s_i \in S$, one can build a point location structure that answers post office queries in $O(\log n)$ expected time and linear expected space.

Proof. We use the randomized incremental construction of the trapezoid graph of the proxy trapezoidation, as in [31, ch. 6]. A query determines the trapezoid τ that contains the query point q using degree 1 tests at x-nodes and degree 2 test at y-nodes. We can then use the degree 2 $Closer(B_{ij}, q)$ test to decide which of the two candidate sites is closer. □

Chapter 8 describes an algorithm for labeling each grid point with its closest site in $O(U^2)$ expected time, $O(U^2)$ space and degree 2. By slightly modifying the algorithm, one can produce all proxies in $O(U^2)$ expected time, $O(n)$ space and degree 2. Thus,

Theorem 27. There exists a degree 2 algorithm that produces a data structure supporting $O(\log n)$ time and degree 2 point location queries.

While Theorem 27 tells us that we can build the proxy trapezoidation with degree 2, it is slow. Perhaps the proxy trapezoidation could be built more quickly with a direct construction. In the remainder of the section, we describe a incremental construction and conjecture on running time of the construction.

Voronoi Trapezoidation and Conflicts

A few additional concepts will help us describe an incremental construction of the proxy trapezoidation. First, if we split each trapezoid of the proxy trapezoidation with the bisector of sites donating the proxies at top and bottom, then we obtain the Voronoi trapezoidation depicted at the right of Figure 7.7. This trapezoidation is also canonical, since it is derived from the proxy trapezoidation.

It is important to note that we don't actually compute y-coordinates of bisectors; the *top* and *bottom* pointers just point to line equations for proxies, bounding box sides, or bisectors, each of

which can be represented as a degree 2 polynomial. All *left* and *right* points are proxy endpoints or bounding box corners, and all are from $\mathbb{U}_2$. Thus, we observe:

Lemma 28. One can test if a grid point q is inside a given trapezoid τ of a Voronoi trapezoidation in constant time using degree 2.

In the *Voronoi trapezoidation* every trapezoid τ intersects some proxy $P(s_i)$ and either a bisector b_{ij} or a bounding box side; we set $\mathtt{site}(\tau) = s_i$ and $\mathtt{bisector}(\tau) = b_{ij}$ (or the bounding box side in the latter case).

The trapezoids with $\mathtt{site}(\tau) = s_i$ cover the Voronoi polygon $C(s_i)$ and therefore contain all grid points in the Voronoi cell of s_i. We can't really say how the covering trapezoids relate to the true Voronoi cell of s_i. They may miss portions of the cell on the left and right ends, because they all use the proxy $P(s_i)$ as top or bottom. As the striped trapezoids in Figure 7.7 show, they don't necessarily form a convex region and may contain more or less than the true Voronoi cell – all we guarantee is that they contain the grid points of $C(s_i)$. This means that it would be difficult to bound the work necessary to insert a new site s by walking through the Voronoi trapezoidation of existing sites.

Consider a subset of sites $R \subseteq S$. We say a site $s_k \notin R$ is *in conflict* with trapezoid τ of the Voronoi trapezoidation of R if and only if there exists a $g \in \tau \cap \mathbb{U}_2$ such that $\|g - s_k\| < \|g - \mathtt{site}(\tau)\|$, i.e., if some grid point in τ is closer to s_k than to $\mathtt{site}(\tau)$. We extend the conflict to the proxy trapezoid whose split produced τ, as well. We call g a *witness* to the conflict.

In the next section we will give a detailed description of how to maintain the proxy/Voronoi trapezoidations as sites are inserted. As in many randomized incremental constructions, we will find it useful to keep the history DAG of the proxy trapezoidation: as a new site is added, some old trapezoids will be deleted and their area filled with new trapezoids – the parents of a new trapezoid will be the minimal set of old trapezoids that cover all the grid points that it contains. This implies that we can trace conflicts through history:

Lemma 29. If site $s \in S$ is in conflict with a trapezoid τ of the history DAG, then it is in conflict with at least one parent of τ.

For sites S, let D2-Voronoi(S) denote the proxy trapezoidation with its history represented in a history DAG and the Voronoi trapezoidation implied by introducing bisectors in each trapezoid.

7.4.2 Constructing the D2-Voronoi Diagram

In this section we give the details of our degree 2 incremental construction. We assume that we are given a set of n sites S; let $S_i = \{s_1, s_2, \ldots, s_i\}$. We further assume that we have built the proxy trapezoidation and its history that constitute D2-Voronoi(S_{i-1}) and want to obtain D2-Voronoi(S_i) by inserting s_i.

After observing how incremental construction affects the Voronoi polygons and proxy segments, as defined in Section 7.4.1, we introduce a few more predicates and constructions for grid points in trapezoids. Finally, we describe the updates to the trapezoidation, and analyze the time and space.

Lemma 30. In an incremental construction, a Voronoi polygon on the grid can only shrink:

$$\forall s \in S_{i-1}, \quad C_{S_i}(s) \subseteq C_{S_{i-1}}(s),$$

and a proxy segment changes if and only if the new site s_i conflicts with at least one of its endpoints:

$$\forall s \in S_{i-1}, P_{S_i}(s) \neq P_{S_{i-1}}(s) \iff s_i \text{ conflicts with an endpoint of } P_{S_{i-1}}(s).$$

Predicates and Constructions

We will continue to use a model of computation in which arithmetic operations are constant-time. Computations do not use floor or integer division unless otherwise stated.

Lemma 31. (i) Determining if bisector B_{ij} intersects the vertical segment $\overline{rt}$ with $r, t \in \mathbb{U}_2$ takes degree 2 and constant time. (ii) Determining the shortest subsegment $\overline{ab} \subset \overline{rt}$ with $a, b \in \mathbb{U}_2$ that intersects B_{ij} takes degree 2 and $\log \|r - t\|$ time, or constant time if floor is allowed.

Proof. For (i), compare the result of `SideOfABisector` for B_{ij} with the two endpoints of $\overline{rt}$. (ii) can use floor, or can binary search with `SideOfABisector` and B_{ij}. □

Lemma 32. There is a degree 2 procedure `HullVertices`, taking $O(\log U)$ time, that computes the convex hull of the grid points, if any, that lie in a region bounded by a constant number of bisectors or lines defined by two grid points.

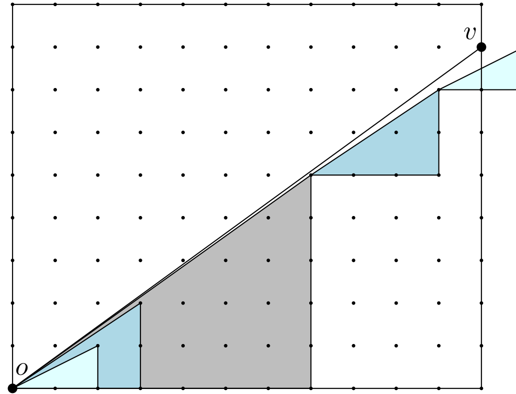


Figure 7.8: Each darker triangle's hypotenuse at left is an increasingly accurate rational representation of the slope of segment $\overline{ov}$; the `GCD walk` reuses these at right to compute the convex hull of the grid points below $\overline{ov}$.

Proof. Euclid's GCD algorithm, interpreted geometrically, gives the sequence of best rational approximations to an arbitrary slope, ordered by increasing denominator. Kahan and Snoeyink [68, Lemma 4.6] turned this into a procedure `HullVertices`(o, v) that takes the origin, an arbitrary point v, and computes the convex hull of the grid points in the bounding box of $\overline{ov}$ that are on or below $\overline{ov}$. Their procedure runs in $\Theta(\log(\|o - v\|))$ time and is degree 2; the highest degree predicate is the orientation test on grid points. The lower bound applies because the convex hull of grid points in the intersection of $O(1)$ half-planes may have this many vertices.

With a small modification, their procedure can start from a unit segment between grid points, from Lemma 31(ii), and follow a bisector within a region. We call this operation the `GCD walk`. In each step we evaluate the degree 2 predicates, `SideOfABisector` and `Orientation` for grid points. The walk takes time proportional to the log of the maximum side of the bounding box of the region in which it runs. This remains bounded by $O(\log U)$. □

This has two useful corollaries.

Corollary 33. Procedure $\texttt{InConflict}(\tau, s_i)$ determines if s_i is in conflict with trapezoid τ of the Voronoi trapezoidation in $O(\log U)$ time and degree 2.

Proof. Suppose that $s_u = \texttt{site}(\tau)$, as in Figure 7.9. We can use the `GCD walk` to determine the convex hull of grid points that are in τ between $\tau.\mathit{left}$ and $\tau.\mathit{right}$ and bounded by $\texttt{bisector}(\tau)$ and b_{ui}, taking $O(\log U)$ time and degree 2. □

For the next corollary, and as shown with a dash-dotted line in Figure 7.9, the proxy of any convex hull of grid points is the line segment from the minimum to the maximum under lexicographic order ($\prec$).

Corollary 34. Given a site s_i in conflict with a trapezoid τ of the Voronoi trapezoidation, the operation $\texttt{findProxies}(s_i, \tau)$ constructs, in $O(\log U)$ time and degree 2, the proxies for two convex hulls of the grid points in trapezoid τ of the Voronoi trapezoidation—those that are closer to $\texttt{site}(\tau)$, and those that are closer to s_i.

Incremental construction

Now, suppose that we have constructed D2-Voronoi(S_{i-1}). We update the proxy trapezoidation and its history as we insert the new site s_i using three steps. The first step, *Find Conflict*, identifies the set of trapezoids of D2-Voronoi(S_{i-1}) that are in conflict with s_i. The second step, *Proxy*

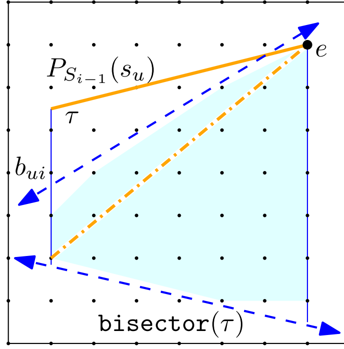


Figure 7.9: Suppose that new site s_i is in conflict with a Voronoi trapezoid τ, with $s_u = \texttt{site}(\tau)$. As drawn, $\tau.top$ is proxy $P_{S_{i-1}}(s_u)$ and $\tau.bottom = \texttt{bisector}(\tau)$. The new bisector b_{ui} cuts the shaded area from τ, which is the convex hull of the grid points that witness the conflict by being closer to s_i than s_u. The dash-dotted line depicts the proxy of s_i in τ.

Update, finds the new proxy $P_{S_i}(s_i)$ and updates all other proxies that changed on the insertion of s_i. The third step, *Trapezoidation Update* splits and merges proxy trapezoids to reflect these proxy changes, and updates the history, producing D2-Voronoi(S_i).

Find Conflict: We collect the trapezoids of the D2-Voronoi(S_{i-1}) that are in conflict with s_i by traversing the history DAG. Starting at the root, we visit only those trapezoids that are in conflict with s_i, or whose parent was in conflict with s_i. At each trapezoid, we test for conflict by Corollary 33, spending $O(\log U)$ time and degree 2. By Lemma 29, if there is a conflict, it persists all the way to the root, so we will find it.

Proxy Update: Adding s_i to S_{i-1} creates a new proxy $P_{S_i}(s_i)$ for s_i, and usually forces some old proxies to be updated, for example, proxy $P_{S_{i-1}}(s_u)$ in Figure 7.9. Thanks to Lemma 30, we can identify which proxies must be updated by testing their endpoints: In Figure 7.9, trapezoid τ is defined in part by endpoint e of proxy $P_{S_{i-1}}(s_u)$. Testing $\texttt{Closer}(s_i, s_u, e)$ identifies, in constant time and degree 2, whether this proxy must be updated. Moreover, we can find all proxies to update by testing only those that define trapezoids in conflict with s_i, since an endpoint that witnesses the need to update the proxy is also a grid point that witnesses a conflict with s_i.

For each Voronoi trapezoid τ in conflict with s_i, we can apply Corollary 34 to find the min and max grid points in the regions that we obtain if we split τ by the bisector of $\texttt{site}(\tau)$ and s_i. The min and max grid points identified from regions on the s_i side of the bisector become the endpoints of the new proxy $P_{S_i}(s_i)$. Similarly, for any proxy that is updated, we take the min and/or max points identified from the regions closer to the corresponding site. Thus, we spend $O(\log U)$ time and degree 2 on each trapezoid in conflict with s_i.

Trapezoidation Update: Having identified the updates to proxies, we must now update the trapezoidation and the history. We do this in four substeps.

First, we add all new proxy endpoints splitting the trapezoids that contain them. In the history, old trapezoids point to two new trapezoids that are leaves of the history DAG.

Second, we add the segment for the new proxy $P_{S_i}(s_i)$ by walking through the trapezoidation, splitting and merging trapezoids as needed. To update the history, briefly, if k trapezoids are crossed by the proxy, then for each trapezoid we create two leaf trapezoids. (Note that some of the trapezoids crossed may not have been found in the history since they may not contain a witnesses to a conflict. They did, however, have witnessed conflicts on both ends. This is important, because for an efficient algorithm we cannot afford to walk through all trapezoids looking for possible grid points to include in the Voronoi polygon $C_{S_i}(s_i)$.)

Third, we shorten the old proxies that need to be updated. Notice that trapezoids that simply replace $P_{S_{i-1}}(s)$ with a shorter $P_{S_i}(s)$ as the *top* or *bottom* change geometrically, but do not need to change their representations. Thus, this operation is like the inverse of adding a proxy in step two: we erase $P_{S_{i-1}}(s)$ from those trapezoids that do not intersect $P_{S_i}(s)$, merging and splitting and updating the history as needed.

Fourth, we merge any adjacent trapezoids whose left and right boundaries are defined by grid points that were proxy endpoints but are no longer. In the history this just redirects pointers to corresponding leaves.

Lemma 35. The leaves of the trapezoid graph corresponding to new trapezoids in the proxy trapezoidation of S_i are $O(1)$ deeper than the leaves for their parent trapezoids in the proxy trapezoidation of S_{i-1}.

Proof. Each update in substeps 1–3 adds a single level to the history DAG, and each trapezoid of the proxy trapezoidation can be involved in at most 4 updates (to *top*, *bottom*, *left*, and *right*). □

Analysis

In [100] we proposed a randomized incremental construction and attempted to use Mulmuley's general framework of stoppers and triggers (or definers and killers, as described in [31, ch. 9.3]) to show one can compute the D2-Voronoi using $O(n \log n \log U)$ time and $O(n)$ expected space. To get the bounds using the frameworks one needs to show that the number of sites defining a trapezoid is $O(1)$. While a trapezoid is defined by at most 4 proxy segments, it is not clear how one bounds the number of sites needed to define a proxy segment.

A more focused problem is to bound δ, the maximum number of sites needed to define the right-most point of a Voronoi polygon. An upper bound (that is far too loose to be useful) is $\delta < 4U$. If a grid point p were to be a right-most point, for each column there is at most one site above and below, for each row there is at most one site left and right. This would tell us that building the D2-Voronoi would take $O(Un \log n \log U)$ expected time. We conjecture that $\delta < \log U$ which would produce an expected running time of the form $O(n \log n \log^{\alpha} U)$ expected time for some $\alpha \geq 1$.

In the worst case, an insertion can create $O(n)$ new trapezoids. While it seems too pessimistic, without better analysis, building the D2-Voronoi takes $O(n^2 \log U)$ time and degree 2.

7.5 Conclusion

In this chapter, we showed how to construct for n sites the rp-Voronoi diagram in $O(n \log(Un))$ expected time and degree 3, and the D2-Voronoi in degree 2. Moreover, we saw how to use the rp-Voronoi and D2-Voronoi to execute point location queries in $O(\log n)$ and degree 2 and how to convert rp-Voronoi to the implicit Voronoi diagram in $O(n)$ and degree 3.

The rp-Voronoi and D2-Voronoi diagrams answer point location queries even though they do not even use sufficient precision to determine a Delaunay triangulation. Since we cannot compute the Delaunay triangulation with degree 2 or 3, can we compute a triangulation that is in some sense close to Delaunay? With degree 2 we can compute the convex hull and Gabriel graph, but which edges should we add to complete the triangulation? The dual of the rp-Voronoi diagram captures any bisector of the Voronoi diagram that intersects a grid line, and thus its dual captures corresponding Delaunay edges. Still, we would be left with polygonal holes in the induced subdivision, which could be triangulated, however, the properties of this triangulation are unknown.

Finally, and perhaps most importantly, can the analysis of the D2-Voronoi diagram be tightened to sub-quadratic time and degree 2.

Chapter 8

Nearest Neighbor Transform

The *nearest neighbor transform* of a black and white $U \times U$ image assigns to every pixel the index of the closest black pixel under the Euclidean metric; it is a discretized Voronoi diagram of the black pixels [13]. The closely related *distance transform* assigns to each pixel the distance to the closest black pixel. Both transforms have a long history of application in image processing [108].

Both can be computed in a straightforward manner, for each pixel, measure the distance to every black pixel and take the minimum. Similar algorithms are often used to generate discrete approximations to the continuous Voronoi diagram, especially with the assistance of graphics hardware [61]. It is a challenge, however, to compute the nearest neighbor transform in time that is bounded by the image size $O(U^2)$ only, and not by the number of black pixels; the algorithms that do this are based on computing the continuous Voronoi diagram and discretizing to a grid [13, 23, 93]. Computing the continuous Voronoi diagram, however, requires four times the input precision to guarantee correct, consistent output, and this is usually ignored in work on efficient geometric algorithms.

We observed [24] that it would be possible to apply Liotta *et al.*'s "degree-driven analysis of algorithms" [89] to derive an algorithm to compute the nearest neighbor transform in double precision. Here, we provide the full algorithm and show that our degree 2 algorithm takes expected $O(U^2)$ time and $O(U^2)$ space. We also describe a simpler degree 2 algorithm that takes

$O(U^2 \log U)$ time and $O(U^2)$ space but is faster for inputs of reasonable sized (from 256×256 to at least 8192×8192). In addition, we report on experiments with our prototype implementation.

In the next section, we survey previous work and precisely define our task. Then, after exploring predicates in Section 8.1.4, we give algorithm details in Section 8.2, and experiments in Section 8.3.

8.1 Preliminaries

We begin by reviewing the previous work on the nearest neighbor transform, defining some notation, and transforming the problem to that of computing an envelope of lines.

8.1.1 Previous work

Many researchers have developed algorithms for computing the nearest neighbor transform on various architectures, including serial and parallel CPUs and GPUs. However, the arithmetic precision of the algorithms are often overlooked, even though all algorithms need exact arithmetic to guarantee a correct output.

Breu *et al.* [13] proposed the first linear time algorithm, which uses divide and conquer to compute and discretize a 2D Voronoi diagram computation. If you analyze the arithmetic calculations of their geometric predicates, you find that their algorithm requires five times the input precision. They also presented a second algorithm that uses three times the input precision.

Both Chan and Maurer *et al.* proposed dimension reduction algorithms [22, 23, 93] that compute the nearest neighbor transform from one-dimensional Voronoi diagrams for each column and one-dimensional weighted Voronoi diagrams for each row. Their calculations can be modified to require only three times the input precision. By changing parts of the algorithm, we will see how to reduce that to double precision without affecting the asymptotic $O(U^2)$ bound on the expected running time.

Hoff and others [61] presented the earliest work on using the GPU to compute the nearest neighbor transform. Their algorithm is dependent on the number of black pixels in addition to the size of the image, and precision is determined by the resolution of the Z-buffer. Approximate GPU algorithms have been proposed [112, 113, 117], but these cannot guarantee an exact result. Cao *et al.* [20] adapted Maurer's algorithm to the GPU, focusing on efficient data structures that take advantage of the memory and the multi-threaded processing power of the GPU. Cao's implementation used triple precision, just like Maurer's.

8.1.2 Definitions

We assume that we are given the coordinates of n black pixels, or ***sites*** from a $U \times U$ grid, $\mathbb{U}$. Denote the sites $S = \{s_1, ..., s_n\}$, where each $s_i = (x_i, y_i)$ with $1 \leq x_i, y_i \leq U$. We will be a little lazy in notation, denoting the coordinates of other pixels $p \in \mathbb{U}$ as $p = (x_p, y_p)$.

In this chapter, we assume that the n sites S are listed in order of increasing x-coordinates, with ties broken by y-coordinates. That is, for all $i < j$, either $x_i < x_j$ or ($x_i = x_j$ and $y_i < y_j$). Note that this also forbids duplication of sites. We can create this ordered list easily from pixels in a given image in $O(U^2)$ time and $O(n)$ additional space, or from an unordered list by counting sort in $O(n + U)$ time and space.

We can formally define the problem of computing the nearest neighbor transform as follows:

Problem 36 (NNTrans-min)**.** For each pixel $q \in U^2$, find the site $s_i \in S$ such that, for all $j < i$, the distance $\|q - s_i\| < \|q - s_j\|$, and for all $j > i$, the distance $\|q - s_i\| \leq \|q - s_j\|$.

That is to say, for each pixel, we wish to finds the closest site, breaking ties by selecting the site with the lowest index.

8.1.3 Problem transformations

We can use common transformations [16, 40] to turn the NNTrans-min problem into a series of problems on lines. First, we can work with squared distances when comparing distances from q to sites s_i and s_j, using any of the following equivalent inequalities for $j < i$: (For $i < j$, replace all strict inequalities, so $>$ becomes $\geq$ in the third line below.)

$$\|q - s_i\|^2 < \|q - s_j\|^2$$

$$q \cdot q - 2q \cdot s_i + s_i \cdot s_i < q \cdot q - 2q \cdot s_j + s_j \cdot s_j$$

$$2x_i x_q + 2y_i y_q - x_i^2 - y_i^2 > 2x_j x_q + 2y_j y_q - x_j^2 - y_j^2.$$

Thus, we can convert the NNTrans-min problem to an equivalent:

Problem 37 (NNTrans-max)**.** For each pixel q, find the site with lowest index $s_i \in S$ that achieves the maximum of $2x_i x_q + 2y_i y_q - x_i^2 - y_i^2$.

NNTrans-max can be solved one row at a time, by fixing the y-coordinate the equation that we wish to maximize looks like a line. For the given Y: Let $L_Y(s_i)$ map the site s_i to the line $\ell_i^Y : y = a_i x + b_i$ where the slope $a_i = 2x_i = $ ① and the intercept $b_i = 2y_i Y - x_i^2 - y_i^2 = $ ② . Let $L_Y = L_Y(S)$ denote the set of all lines obtained from sites S. Note that since S is sorted by x with ties broken by y, L_Y is sorted by slope, with ties broken by y-intercept.

Along a fixed row $y = Y$ of the grid, we can phrase the NNTtrans-max problem as determining the highest line at each pixel – the *discrete upper envelope (DUE)* of the lines.

Problem 38 (DUE-Y)**.** For a fixed $1 \leq Y \leq U$, and for each $1 \leq X \leq U$, find the smallest index of a line of L_Y with maximum y coordinate at $x = X$.

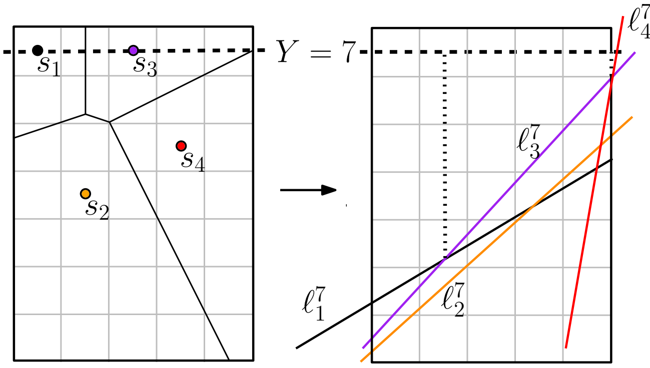


Figure 8.1: Example transformation from S to $L_Y(S)$ for $Y = 7$. *Left*: sites s_1 through s_4 are shown. *Right*: Each site s_i is transformed to the line ℓ_i^7. Observe that when ℓ_i^7 is the highest line at $x = X$, site s_i is the site with minimal distance to pixel at $(X, 7)$

The key to computing the Nearest Neighbor transform in $O(U^2)$ time and degree 2 will be solving DUE-Y in $O(U)$ time and degree 2. As shown in Figure 8.1, when ℓ_i^Y is the highest line at $x = X$, site s_i is the site with minimal distance to pixel at (X, Y). Observe that each line in L_Y has the form $y =$ ① $x +$ ② . In Section 8.1.4 we show how to compute the discrete upper envelope in $O(U)$ expected time with degree 2 and in Section 8.2 we use this construction to solve NNTrans-max in $O(U^2)$ expected time and degree 2.

8.1.4 Geometric primitives

Before diving into algorithms, we present some geometric primitives (predicates and constructions) and their precision. Most of these primitives are part of our algorithm; the few that are not are tempting options that we avoid because of their high precision requirement. Recall from Chapter 4

Observation 39. Given two sites, s_i and s_j and a query point q with $s_i, s_j, q \in \mathbb{U}$, the predicate $\texttt{Closer}(s_i, s_j, q)$ is degree 2.

In special geometric configurations this may simplify further. For example, if the sites are on the same vertical line so $x_i = x_j$, then there is a factor of $(y_i - y_j)$, so the computation need only determine the signs of two linear terms. In general, however, it is irreducible of degree 2 as we saw in Chapter 5.

Observation 40. Given two sites s_i and s_j on the same vertical line and a query point q with $s_i, s_j, q \in \mathbb{U}$ the predicate $\texttt{Closer1D}(s_i, s_j, q)$ is degree 1.

A common test in geometric algorithms is determining the orientation of three points, a, b, and c, by evaluating the sign of a determinate whose entries are the homogeneous coordinates of the points. The $\texttt{Orientation}(a, b, c)$ predicate is degree 2 on points with degree 1 coordinates. When the points have coordinate values of ($\textcircled{1}$, $\textcircled{2}$) (which is the form our points will take after transformations), the predicate has higher degree:

$$\begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & c_y \\ 1 & c_x & c_y \end{vmatrix} = \begin{vmatrix} \textcircled{0} & \textcircled{1} & \textcircled{2} \\ \textcircled{0} & \textcircled{1} & \textcircled{2} \\ \textcircled{0} & \textcircled{1} & \textcircled{2} \end{vmatrix} = \begin{vmatrix} \textcircled{1} - \textcircled{1} & \textcircled{2} - \textcircled{2} \\ \textcircled{1} - \textcircled{1} & \textcircled{2} - \textcircled{2} \end{vmatrix}$$

$$= \begin{vmatrix} \textcircled{1} & \textcircled{2} \\ \textcircled{1} & \textcircled{2} \end{vmatrix} = \textcircled{1}\ \textcircled{2} - \textcircled{1}\ \textcircled{2} = \textcircled{3}$$

Observation 41. Given three points a, b, and c with coordinate precision ($\textcircled{1}$, $\textcircled{2}$) the predicate $\texttt{Orientation}(a, b, c)$ is degree 3.

Consider two lines, $\ell : y = \ell_m x + \ell_b$ and $h : y = h_m x + h_b$, each with degree 1 slope and degree 2 y-intercept, i.e., $y = \textcircled{1}\ x + \textcircled{2}$. The construction $\texttt{Intersect}(\ell, h)$ returns the coordinates of the intersection of the two lines q, which are:

$$q_x = \frac{h_b - \ell_b}{\ell_m - h_m} = \frac{\textcircled{2} - \textcircled{2}}{\textcircled{1} - \textcircled{1}} = \frac{\textcircled{2}}{\textcircled{1}}$$
$$q_y = \frac{\ell_m h_b - \ell_b h_m}{\ell_m - h_m} = \frac{\textcircled{2}\ \textcircled{1} - \textcircled{1}\ \textcircled{2}}{\textcircled{1} - \textcircled{1}} = \frac{\textcircled{3}}{\textcircled{1}}$$

Observation 42. Given two lines ℓ and h of the form $y = \textcircled{1}\ x + \textcircled{2}$, the $\texttt{Intersect}(\ell, h)$ construction produces a point with rational x- and y-coordinates of ($\textcircled{2}$ / $\textcircled{1}$, $\textcircled{3}$ / $\textcircled{1}$).

As our goal is a degree 2 algorithm, the high degree primitives of Observations 41 and 42 are undesirable. Next, we describe the predicates that provide just enough information to compute a nearest neighbor transform with degree 2.

Given two points $a, b \in \mathbb{U}$, the predicate $\texttt{Before}(a, b)$ returns true if a is lexicographically before b. That is, if $a_x < b_x$ (or $a_x = b_x$ and $a_y < b_y$). This test compares degree 1 values only, therefore the predicate is degree 1.

Observation 43. Given two points $a, b \in \mathbb{U}$, the predicate $\texttt{Before}(a, b)$ is degree 1.

Given two lines, ℓ_1 and ℓ_2, of the form $y = \text{①}\, x + \text{②}$, and a degree 1 vertical line h defined by points satisfying the equation $x - x_0 = 0$, the predicate $\texttt{Above}(\ell_1, \ell_2, h)$ returns true if ℓ_1 is above ℓ_2 on the line h. We evaluate this predicate by plugging x_0 into the slope intercept form of the lines and comparing the result. Thus, we compare the evaluation of two degree 2 polynomials.

Observation 44. Given two lines ℓ_1 and ℓ_2 of the form $y = \text{①}\, x + \text{②}$ and a degree 1 vertical line h, the predicate $\texttt{Above}(\ell_1, \ell_2, h)$ is degree 2.

We can use the $\texttt{Above}$ predicate in a binary search for the vertical slab in which the ordering of two lines swap. When the vertical slab is a column width, we can return the column containing the lines intersection. Thus,

Lemma 45. Given two lines ℓ_1 and ℓ_2 of the form $y = \text{①}\, x + \text{②}$, the construction $\texttt{IntersectCol}(\ell_1, \ell_2)$ returns the column containing the intersection of the two lines in $O(\log U)$ time and degree 2.

8.2 Algorithm Description

We show first how to compute the discrete upper envelope of lines with double precision, then how to use this to compute the nearest neighbor transform.

8.2.1 Discrete Upper Envelope

Consider the discrete upper envelope (DUE) of a set of lines, which we represent as a minimal length sequence of tuples (s, i, j) where s is the highest line at x for all integers between i and j,

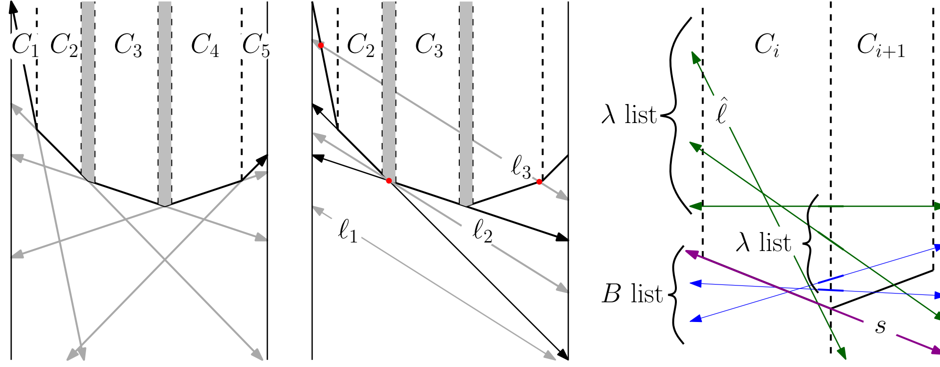


Figure 8.2: A DUE of m lines L is created by adding a random sample R of $m/2$ lines to the DUE of $L \setminus R$. ***Left***: The DUE contains cells C_1 through C_5 defined by lines shown in gray. Each cell is defined by (s, i, j), and can be visualized as the trapezoidal area above line s, between $x = i$, and $x = j$. Sometimes boundaries do not match up and we get a gap between two cells, for example, the gray gap between C_2 and C_3. ***Center***: Three examples of identifying the first cell intersected by identically sloped lines ℓ_i. Thanks to the slope ordering of an upper envelope, each ℓ_i would intersect cells C_2 or C_3 if it intersects any cells. Since we are computing the DUE, cells that would be completely inside a grid column, for example a cell formed by ℓ_2, are dropped. As ℓ_3 intersects C_2 and C_3 we test cells to the left until finding the leftmost intersected cell. That cell then adds ℓ_3 to its B list. ***Right***: Each cell C_i maintains three lists of lines: s, colored magenta, defines C_i; B, colored blue, are the lines intersecting C_i from the bottom; and λ, colored green, are the lines intersecting C_i from the left. We find cell C_{i+1}'s λ list by throwing away any line of $\ell \in \lambda \cup B$ that is below the defining line of C_{i+1} at its leftmost grid point, for example $\hat{\ell}$.

and ties in 'highest' are broken by lowest index. Each tuple (s, i, j) determines a cell consisting of all the points on or above s at $i \leq x \leq j$.

For any set of lines with the same slope, only the line with largest y-intercept appears on the upper envelope. Thus, in $O(n)$ time we can throw away any line that shares a slope but has smaller y-intercept value. We are left with a set L of $m = O(U)$ lines; for the remainder, we compute the discrete upper envelope of these lines.

We briefly sketch three algorithms for computing discrete upper envelopes, which, in increasing implementation complexity, are: `D3-DUE` is an $O(U)$ algorithm using a degree 3 `Orientation` predicate; `UlgU-DUE` replaces the predicate with `IntersectCol` to reduce to degree 2 at the cost of $\log U$ time for binary search; `U-DUE` achieves $O(U)$ expected time and degree 2 by randomization. We test all three algorithms in our experiments.

First, we sketch the `D3-DUE` procedure, which takes L and produces the upper envelope of L. It is known that computing the upper envelope of a set of lines is equivalent to computing the lower hull of a set of points sorted by x under a transformation that maps each line $y = ax + b$ to point $(a, -b)$ [74] . Linear time algorithms for computing the lower hull of a set of points sorted by x use the `Orientation` predicate [94]. For our input, the coefficients a and b are degree 1 and 2, respectively, so `Orientation` is degree 3 by Observation 41; any procedure that uses `Orientation` is at least degree 3. Thus, `D3-DUE` takes $O(U)$ time and degree 3.

Second, the `UlgU-DUE` procedure also takes $L = \{\ell_1, \ldots, \ell_m\}$, sorted by slope, and produces the discrete upper envelope directly from the lines. Even here we have an obstacle; in Observation 42 we saw that computing intersection points requires too much precision. Fortunately, for the discrete upper envelope it is enough to identify the grid column containing the intersection of a pair of lines, which is degree 2 and $O(\log U)$ time by Lemma 45. We use this as follows.

We maintain a stack to represent the prefix of the DUE computed so far, much like a convex hull algorithm. Instead of using an orientation test to decide to pop the stack on the insertion of a new line ℓ_k, we use the `Above` test to pop all cells (s, i, j) where ℓ_k is above s at i and at j. Let line ℓ be the line defining the last cell of the DUE, we find the beginning column of the cell for ℓ_k with `IntersectCol`(ℓ_k, ℓ). The procedure `UlgU-DUE` uses only the degree 2 predicates `Above` and `IntersectCol` and takes $O(U \log U)$ time.

Third, the `U-DUE` procedure solves the DUE-Y problem in expected $O(U)$ time and degree 2 by random sampling and one-sided recursion. We use a stable random partitioning of the set L of m lines to create a set R of $m/2$ lines that maintains slope ordering. In order to find the DUE of all m lines, we (recursively) find Q, the DUE of the lines not in R, and then 'add' the lines of R into Q.

Here is an overview of the input, output, and invariants for the 'adding' procedure in `U-DUE`, depicted in Figure 8.2:

Input: The sampled lines R, ordered by slope with no duplicate slopes, and Q, the DUE of rest of the lines, represented as an array of cells in order of increasing x. Each cell stores its defining line and leftmost grid point, the sentinel leftmost cell has left end at $-\infty$.

Output: The DUE of all the lines, in the same representation as an array of cells as the input.

Data Structure: Each cell is given the head of a slope-sorted linked list of lines that "start in the cell;" each line of R is put in the list for the first cell it intersects (by increasing x).

Processing: The new DUE is built by processing the old cells in increasing x, i.e., from left to right.

Invariant A: Three slope-sorted lists of lines contribute to the DUE inside a cell: s, the single line defining the cell; B, the list of lines that start in the cell; and λ, the list of lines that cross over from the previous cell.

Invariant B: The algorithm uses a stack to maintain the output DUE of the set of lines from the lists of Invariant A up to a chosen slope.

We initialize and maintain invariants as follows:

First, we identify, for each line ℓ of R, the cell that ℓ starts in. We do so in two sub-steps: first we identify a cell intersected by ℓ (if one should exist). Since R and the lines of an upper envelope are ordered by slope, we can do this by merging Q and R in $O(m)$ time. Next, we walk through cells to the left to find the first cell intersected by each line ℓ. We bound the expected time of this operation later. We do these sub-steps for the lines of R in reverse order, inserting lines at the head of the linked list associated with the cell, so that the lines starting in each cell remain in order of increasing slope.

Next, we process cells from left to right. To begin, any lines that start in the sentinel are checked to see if they continue into the first finite cell. Those that do not are discarded. This gives the three lists for the first cell.

In processing a cell, we maintain a stack to represent the DUE, as in `UlgU-DUE`. The only difference is that the binary searches of `IntersectCol` is over the boundary of the cell (not the entire grid).

Lemma 46. Given a set S of n lines sorted by slope of the form $y =$ ① $x+$ ② , we can compute the discrete upper envelope of S in $O(U + n)$ expected time.

Proof. As mentioned above, in $O(n)$ we produce L of size m, where each line has a unique slope. We upper bound the processing for a cell of width U_i intersected by m_i lines as $cm_i \log U_i$, where c is a constant. Clarkson and Shor's results on random sampling [28, Theorem 3.7] say that $E[\sum_i m_i^2] = O(m)$, so each cell intersects not too many lines on average. By the Cauchy-Schwartz inequality, and as $\sum_i U_i \leq U$, we upper bound the total cost of processing all cells:

$$\begin{aligned} E\left[\sum_i cm_i \log U_i\right] &\leq c\sqrt{\mathrm{E}\left[\sum_i m_i^2\right] \cdot \sum_i \log^2 U_i} \\ &\leq c\sqrt{O\left(m\right) \cdot m \log^2 \frac{U}{m}} \\ &= O\left(m \log \frac{U}{m}\right). \end{aligned}$$

We build DUEs recursively on random samples of half the lines, so the total expected time is bounded by the following recurrence:

$$\begin{aligned} T\left(m, U\right) &= T\left(\frac{m}{2}, U\right) + O\left(m\right) + O\left(\mathrm{E}\left[\sum_i m_i \log U_i\right]\right) \\ &= T\left(\frac{m}{2}, U\right) + O\left(m \log\left(\frac{U}{m}\right)\right) \implies \\ T\left(m, U\right) &= O\left(m \log\left(\frac{U}{m}\right)\right) \end{aligned}$$

Since $m \leq U$, we have $T(m, U) = O(U)$. Thus, in total it takes $O(n + U)$ time in expectation to compute the discrete upper envelope. □

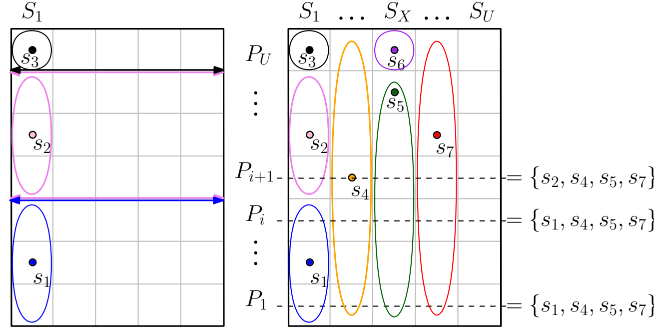


Figure 8.3: *Left*: Ovals drawn around sites touch the pixels in the column that are nearest neighbors in the 1D Voronoi diagram. The Voronoi diagram of sites in the same column S_X are horizontal lines. Thus, if we restrict attention to sites in S_X, all pixels a row have the same closest site. So, only one site per column can ever be the nearest neighbor to a pixel, and vertical comparisons suffice to find out which one is nearest$_X(Y)$, leaving us with only $O(U)$ sites to consider for each row. *Right*: Illustration of *possible lists* and their generation. Ovals show vertical nearest neighbors. After determining which sites are the highest in their column, (s_3, s_4, s_6, and s_7), we only need to perform a few tests for each site $s_i \in P_i$ to generate P_{i+1}. Each site s_i exhibits one of three cases. Case 1: s_i is the highest in its column, so it is in P_{i+1}. Case 2: s_i is still nearest$_X(Y)$. Case 3: s_{i+1} is now nearest$_X(Y)$ so it replaces s_i in P_{i+1}.

8.2.2 NNTrans Algorithm

Assume that we are given a set of n sites $S = \{s_1, \ldots, s_n\}$ on a $U \times U$ pixel grid. We compute the Nearest Neighbor transform of S by using DUE-Y in each row in order, which takes in $O(U^2)$ expected time, $O(U^2)$ space and degree 2. Our algorithm has the following three steps.

In the first step, if S is not already sorted, then we sort the sites by increasing x-coordinate (breaking ties by y-coordinate) using counting sort by y and then by x. For convenience, we assume that the sites of S are labeled in their sorted order.

Let S_X denote the set of sites of S with given x-coordinate value X – sites in a given grid column, as seen in Figure 8.3. Each S_X is a contiguous sublist of S, thanks to the sorting. The Voronoi diagram of just the sites in S_X is formed by the horizontal lines bisecting adjacent sites in S_X.

If we now consider a given grid row $y = Y$, then from each non-empty S_X at most one site can possibly contribute to the Voronoi diagram – for each S_X, let nearest$_X(Y)$ = the site of

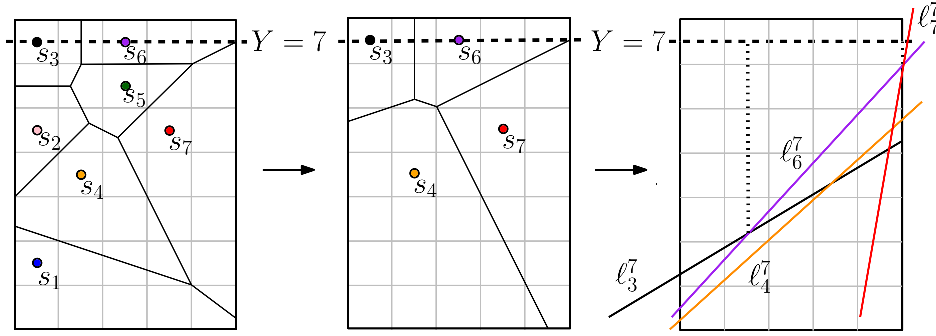


Figure 8.4: The three steps of the NNTrans algorithm: (Steps one and two show the Voronoi diagram of the sites to illustrate that it does not change along Y when sites are removed.) *Step one*: Sites are sorted by x-coordinate breaking ties by y-coordinate. *Step two*: For each row Y, sites are reduced to the possible list P_Y. *Step three*: Each site $s_i \in P_Y$ is transformed to the line ℓ_i^Y and we compute the DUE of the lines.

S_X nearest to (X, Y), breaking ties by lowest index (and $\emptyset$ when $S_X = \emptyset$). The *possible list*, $P_Y = \{\text{nearest}_X(Y) \mid \forall 1 \leq x \leq U\}$, contains the at most U sites that can be nearest neighbors of a pixel in row $y = Y$.

In the second step, depicted in Figure 8.3, we produce a data structure, which we call the *possible list generator*, that iterates from $Y = 1, \ldots, U$ producing each P_Y. This data structure simply maintains a pointer to the last site used in each ordered list S_X. These pointers advance as Y crosses the bisector between adjacent elements in each list S_X, generating all possible lists P_Y in sequence.

In the third step, we transform each possible list into lines, and compute the discrete upper envelope of the lines $L_Y(P_Y)$ from Problem 38 by Lemma 46. Each DUE-Y constructs one row of the Nearest Neighbor transform.

Theorem 47. Given n sites on a $U \times U$ pixel grid, we can compute the Nearest Neighbor transform of the sites in $O(U^2)$ expected time, $O(U^2)$ space, and degree 2.

Proof. The procedure to compute the Nearest Neighbor transform of the sites has three steps, shown in Figure 8.4.

The first step sorts the n sites with two counting sort in two passes over S and makes degree 1 comparisons of coordinates. Thus, the step uses $O(U)$ space, $O(n)$ time and degree 1.

The second step constructs the possible list generator. Each possible list contains at most U sites. Thus, the size of the data structure is $O(U)$. Since S is sorted we can initialize P_1 in $O(U)$ time by identifying adjacent sites with different x-coordinates, which is degree 1. To update from P_Y to P_{Y+1}, we perform at most U evaluations of the `Closer1D` predicate, which by Observation 40 is degree 1. Thus, initializing the possible list generator takes $O(U)$ time, $O(U)$ space and degree 1 and generating P_{Y+1} from P_Y takes $O(U)$ time and degree 1.

The third step generates the possible list P_Y for each $Y = 1, \ldots U$. The sites are transformed into lines $L_Y(P_Y)$ sorted by slope. Each line in L_Y has the form ① $x +$ ② and we compute the discrete upper envelope of L_Y. Since L_Y has at most U lines, by Lemma 46, processing each Y takes $O(U)$ expected time, $O(U)$ space and degree 2. Therefore, processing all possible sites takes $O(U^2)$ expected time, $O(U^2)$ space and degree 2. □

In the proof above we use the procedure `U-DUE` to compute the nearest neighbor transform in $O(U^2)$ time and degree 2. However, we could have instead used `UlogU-DUE` to get an $O(U^2 \log U)$ degree 2 algorithm or `Deg3-DUE` to get an $O(U^2)$ degree 3 algorithm. In the next section we will compare the experimental running times of implementations of the three algorithms and MATLAB's [91] implementation of Maurer's algorithm.

8.3 Experiments

In this section we investigate robustness and timings of four implementations of algorithms that compute the nearest neighbor transform of an image. We will see that the theoretical advantage of removing a $\log U$ factor is outweighed by the cost of generating and maintaining random samples. On the positive side, however, our degree 2 algorithms, which are guaranteed correct in double

precision, are notably faster than the implementation of Maurer's algorithm used to perform MATLAB's `bwdist` function.

Here are the four implementations:

`Usq` is our implementation of the expected $O(U^2)$ time and degree 2 algorithm, discussed in Section 8.2.2;

`UsqLgU` is our implementation of the $O(U^2 \log U)$ time and degree 2 algorithm, discussed at the end of Section 8.2.2;

`Deg3` is our implementation of the (U^2) and degree 3 algorithm, also discussed at the end of Section 8.2.2; and

`Maurer` is the MATLAB `bwdist` function, which is a compiled implementation of Maurer's algorithm (after version of 2009a).

Recall that `Usq` and `UsqLgU` are degree 2 and `Deg3` and Maurer's algorithm are degree 3.

Timings were taken on a 3.2GHz Intel Xeon processor with 12GB RAM running Ubuntu 10.04. Our implementations were written in C++, compiled with gcc toolchain and `bwdist` was run using MATLAB 7.0.10 (R2010a) [91].

In our timing experiments we run the four implementations on varying grid sizes and site densities. The tuple of an algorithm A, grid size U, and density δ form a *run*. For each run, we generate a set of sites S by selecting if a pixel contains a site uniformly at random with probability δ, then we execute A on input S for enough iterations so that it takes over one second and report the average run length of an iteration of the time of the run. For runs on algorithm `Usq`, the random number generator is reset to a fixed seed before each iteration to maintain the same execution path.

Our first experiment investigates timings and variance of `Usq`. The implementation was run on increasing grid sizes $\{265, 512, \ldots, 8192\}$, and densities $\{.01, .02, \ldots, .05\}$, and $\{.1, .2 \ldots, .5\}$. In Figure 8.5, we see a box and whiskers plot of the time-per-pixel for 100 runs of different seeds

and the medians of runs on the same grid size are connected with lines. We plot time-per-pixel as it allow detail to be seen at both small and large grid sizes.

The first observation about Figure 8.5 is that the boxes are small, about 2 microsecond width on average, indicating that Usq's expected $O(U^2)$ running time comes with a small variance. This is not surprising. The algorithm Usq makes U calls to a randomized algorithm with expected $O(U)$ time, which smooths out the variance.

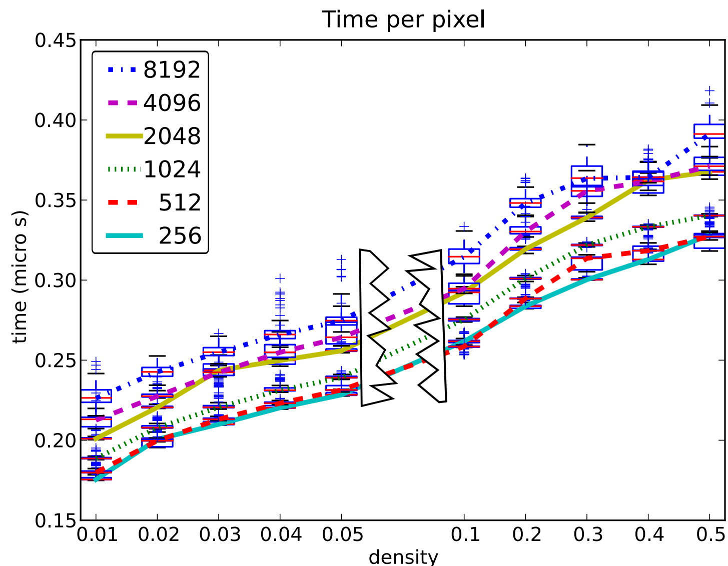


Figure 8.5: Time-per-pixel box plots of 100 runs of Usq on densities between .01 and .5 and grid sizes between 256 and 8192.

One might expect that since Usq has an expected $O(U^2)$ running time, the time-per-pixel should be constant across varying grid sizes and densities. In fact, one would expect higher per-pixel-times for smaller grids since overheads are amortized over fewer pixels.

First, to explain why the per-pixel times increase with the grid size, we fixed the density, varied the grid size, and used Valgrind [106] to capture the number of calls to each line. The per-pixel line counts were equal (or slightly higher for smaller grid sizes, as expected). Thus, we believe that the time increase on larger problems can be attributed to different memory usage patterns, e.g., at data sizes that no longer fit into L1 or L2 cache.

Second, to explain why per-pixel times increase with higher densities, recall that each discrete upper envelope calculation produces a run length encoding of the nearest neighbor transform along each horizontal line and that the sites are uniformly distributed. At density δ, we would expect that a row would intersect $\sqrt{\delta U^2}$ Voronoi cells, for a total output size of $\delta^{\frac{1}{2}} U^2$. The experimental times

are consistent with a large fraction of the running time being proportional to the output size.

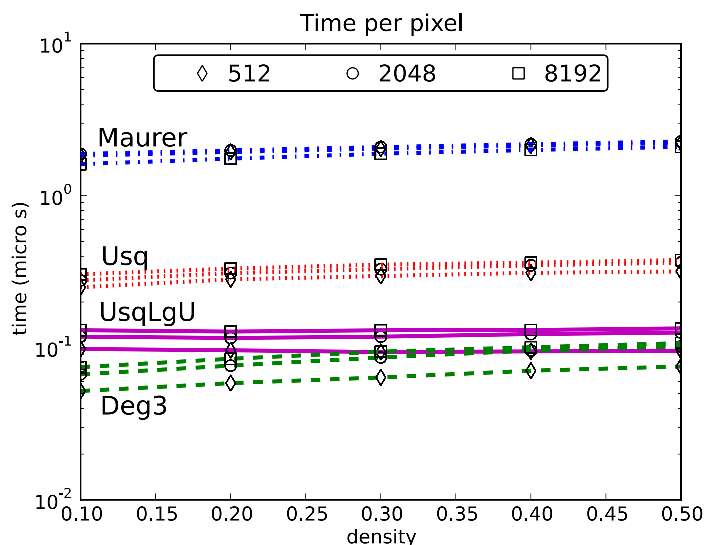


Figure 8.6: Time-per-pixel on a logarithmic scale for four implementations on grid sizes ranging from 512 to 8192 with densities ranging from .1 to .5.

Our second experiment compares the per-pixel time of the four implementations on the three grid sizes, $\{512, 2048, 8192\}$ and the five densities $\{0.1, \ldots, 0.5\}$. In Figure 8.6, we see a plot of density verses time-per-pixel. Each algorithm is shown as a different line style and the markers correspond to grid size. For every run, `Maurer` is the slowest followed by `Usq` (5–7x faster than `Maurer`), `UsqLgU`(15–23x faster than `Maurer`), and `Deg3` (20–33x faster than `Maurer`). First, notice that even though the computational complexity of `Usq` is less than `UsqLgU`, the clock time of `UsqLgU` is faster than `Usq` (by about 3x). The unexpected speed difference is because $\log U$ is not large, and `UsqLgU` maintains almost no data structure while `Usq` keeps track of a lot of data structures, and random number generation is slow. Next it is interesting to note that even though `Maurer` is degree 3 it is still slower than the degree 2 algorithms in `Usq` and `UsqLgU`. Finally, while the `Deg3` is the fastest implementation (at about 1.5x faster than `UsqLgU`) it uses degree 3.

From the second experiment, we can conclude that the degree of an algorithm may not correspond to the clock time. Both degree 2 algorithms are faster than a degree 3 algorithm and slower than a degree 3 algorithm. Thus, similar to asymptotic running time, degree can supply additional information when selecting an algorithm for solving a problem.

Our third experiment compares per-pixel time of the four implementations on 120 extracted boundaries of images from the MPEG7 CE Shape-1 Part B data set. We used the first 20 images from the fish, frog, lizard, lmfish, octopus, and turtle sets, ranging in size from 174^2 to 774^2 pixels. Since the extracted boundaries are closed curves or a small number closed curves, an image of U^2 pixels has $O(U)$ sites. In Figure 8.7, we see a scatter plot of image size verses time-per-pixel. Each algorithm is depicted with a different marker. As is expected, the per-pixel times of the all four algorithms are similar to the previous experiment, even though the grid sizes are smaller and site densities are substantially reduced.

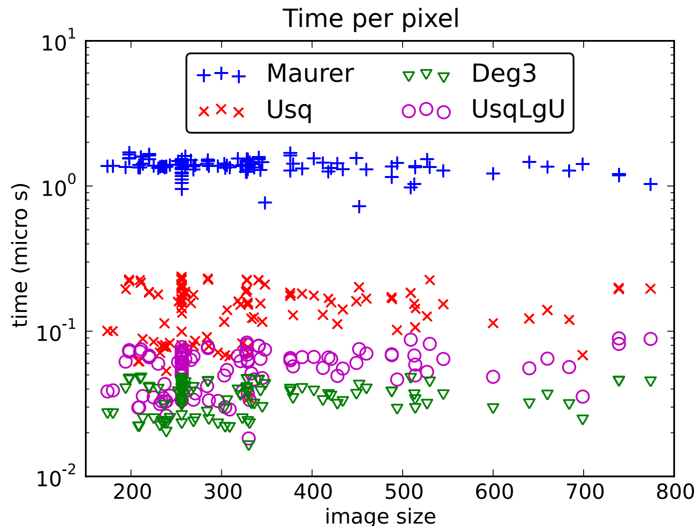


Figure 8.7: Time-per-pixel on a logarithmic scale for four implementations on boundaries extracted from 120 images from the MPEG7 data set.

8.4 Conclusions

We presented a $O(U^2)$ expected time degree 2 algorithm for computing the Nearest Neighbor transform of a $U \times U$ pixel image. The key step was in computing the discrete upper envelope of at most U lines in $O(U)$ expected time and degree 2. It seems possible that the discrete upper envelope could be computed deterministically by divide-and-conquer with more programming complexity, however, any non-randomized degree 2 algorithm that we attempted had an extra $\log U$ factor. Can the $\log U$ factor be removed, while still using degree 2, without randomization?

The algorithms presented generalize to higher dimensions, but what about other norms? L_1 and L_∞ are particularly interesting, as they each have a degree 1 distance comparison.

Chapter 9

Volume Calculation

Constructive solid geometry (CSG) can produce a high fidelity representation of a geometric model. However, the model is often discretized, converting it to a surface or volumetric mesh in order to enable a physical simulation where the model is the domain. Once discretized, properties such as volume or adjacency are much easier to compute. Discretizing, however, is not a trivial task, moreover, sometimes, it is not even necessary.

Monte Carlo (MC) methods for solving the neutron transport equation use CSG to represent the domain exactly without the need for spatial discretization. The high level of geometric detail is frequently desired when dealing with complex models for criticality safety and reactor analysis applications, and, as a result, the CSG representation has been used in several major MC transport solvers, such as MC21 [132], MCNP [139] and KENO-VI [12], which are popular in the MC community.

Previously, the MC particle transport community calculated volumes primarily to verify the quality of a model. Volumes did not directly influence the radiation transport calculation. However, recent efforts to include in-line feedback effects (e.g., depletion, thermal feedback, xenon feedback, etc.) in MC reactor calculations called for MC solvers to calculate volumes with a high degree of accuracy[52]. While some popular MC codes, such as MCNP [139], calculate the volumes of simple objects, no MC code available today is able to guarantee an accurate volume calculation for

all inputs when the modeling primitives have curved surfaces.

When codes cannot calculate component volumes analytically, users are typically required to input volumes, which are obtained from external calculations or a priori information. Many communities of computer science have investigated algorithmic volume calculations from various perspectives. For the time being, we can think of the volume calculation problem faced in MC community as finding the volume of a region described by the intersection of a finite number of algebraic inequalities, sometimes called a *semi-algebraic set*.

The computer science theory community studied volume calculation in arbitrary dimension by sampling points, and observed that computing volume, even for convex sets, is hard. Given $p \in \mathbb{Q}^d$ and $K \subset \mathbb{R}^d$ with K convex, a *well-guaranteed oracle* reports balls of radius $R > r > 0$ such that $B(p, R) \supset K \supset B(p, r)$. Füredi and Bàràny [47] showed that any deterministic, polynomial-time algorithm using a well-guaranteed oracle to give an upper bound V and lower bound v on the volume of a convex set in $\mathbb{R}^d$ must have error exponential in d. (Luckily, we are interested in a fixed d.) Dyer, Frieze and Kannan [38] showed that a probabilistic algorithm can do better using a *membership oracle*, with reports true if $p \in K$. Given K, a relative error $\epsilon > 0$, and a membership oracle, Dyer *et al.*'s algorithm finds with probability at least $3/4$ the volume of K with relative error less than ϵ in time polynomial in ϵ^{-1} and d.

More applied research in graphics, computational geometry and computer aided design investigates volume calculation in 3D, where the problem is tractable. For the 3D setting, Lee and Requicha [85] survey volume calculation techniques in which they give an overview of different domain representations and for each representation provide a technique for computing its volume.

Perhaps the most natural approach is to compute the volume by constructing the boundary of the semi-algebraic set. To do so, one would have to compute the arrangement of the polynomial boundaries. Collin [29] was the earliest to build an arrangement for semi-algebraic sets in arbitrary dimensions, his algorithm is know as Collins decomposition. Collins decomposition

is implemented in computer algebra systems such as Mathematica [138], which use exact arithmetic. Collins algorithm is doubly exponential in the dimension d; this, combined with the difficulty of computing with algebraic numbers makes computing a decomposition time consuming. Schömer and Wolpert [118] used a Collins like algorithm to compute a cell in an arrangement of quadrics. Mourrain *et al.* [104] describe a sweep algorithm to produce a Collins decomposition. Many other papers extend Collins' result, both in theory (Chazelle *et al.* [26] reduced the combinatorial complexity of Collins' algorithm to singly exponential in d, although the algebraic complexity remains doubly exponential) and practice (Saunders *et al.* [116] implemented a parallel Collins decomposition).

A key step in computing a boundary representation of quadrics is intersecting quadrics. Hoffman [62] describes a numerical algorithm for tracing the intersection of two quadrics that intersect non-tangentially. Levin [87] gave an efficient algorithm to derive a parametric form of the intersection of quadrics. Other researches considered quadric intersections by extending Levin's work [134], finding near-optimal parameterizations [37] and implementing exact algorithms [84]. Some researchers identify the class of *natural quadrics*—spheres, cones, and circular cylinders, which occur frequently in computer-aided design—and develop faster geometric techniques to intersect them [95, 122]. In a series of papers and his Ph.D. thesis, Keyser[76, 77, 78, 79, 81, 80] used exact computation to convert CSG models with "low-degree" curved solids (e.g., cones, ellipses, tori, etc.) to boundary representation.

Alternatively, one could avoid computing intersections as much as possible by using subdivision techniques. Lee and Requicha [86] describes two techniques for decomposing a CSG representation into cubes or boxes to compute the volume (or other integral properties) of an object in CSG representation. Similar to Lee and Requicha's, we use a decomposition, however, we use different tests and in some cases (i.e., when it can be done quickly) we compute a boundary representation restricted to a box.

We describe a framework for computing a model's volume that decomposes the model into regions, simplifying the model's representation in each region until a volume calculation routine, called an *integrator*, can take over. In addition, we present specific examples of using representative integrators designed to handle the size, and potential complexity, seen in MC radiation transport models. Through a series of controlled numerical experiments we demonstrate that the new algorithm can produce accurate volume estimates within specified accuracy and confidence bounds. Furthermore, the results from these experiments indicate that the new method can produce these volume estimates up to several hundred times faster than the traditional sampling methods.

Section 9.1 provides a theoretical foundation for the CSG representations used in Monte Carlo transport codes. In particular, we give precise mathematical definitions for the primitives. The primitives are used to build components, which are nested to form the geometric model. Section 9.2 describes a robust domain decomposition algorithm that can be used to subdivide a given component into a set of simpler objects. The section also defines and analyzes the precision of a set of efficient tests for determining when an intersection can occur between a component and an axis-aligned box. These tests are used to classify regions and drive the partition process. Section 9.3 describes a set of volume integration routines that were optimized for the common base cases, which were observed using a separate tool built using the framework. As we anticipate a larger corpus of test models presenting other common base cases, adding base cases has been made quite easy.

The novelty of this framework is that we aim for a minimal set of tests on surfaces, which not only ensures robustness by limiting numerical computation to within surfaces, but also facilitates speed and accuracy improvements by specially coding common cases, as we demonstrate through experiments reported in Section 9.4. In addition, this work is a first step towards applying techniques from computational geometry to MC radiation transport solvers. It is our hope that this introduction will promote additional collaboration and flow of information between the two

technical communities. Finally, we note that here, we analyze the degree of the tests used in our framework, originally presented in [97]. The insights gained from the analysis led us to different predicates than we originally suggested. The new predicates require less arithmetic precision and are simpler to implement.

9.1 Model Representation

Many Monte Carlo codes that solve the neutron transport equation, such as MC21 [132] and MCNP [139], use a constructive solid geometry (CSG) (which is sometimes also referred to as *combinatorial geometry* representation) to define regions of space. A combinatorial geometry model combines basic spatial primitives, bounded by surfaces (planes, cylinders, spheres, ellipsoids, etc.), using Boolean operations (union, intersection, complement, difference). A model consists of one or more interior disjoint, closed regions, called *components*, which may be organized in a parent/child hierarchy to reduce memory overhead and to accelerate particle tracking (which is one of the time consuming steps when solving the transport equation with a Monte Carlo solver). We give concrete details for the surfaces, Boolean formulae, and components, as well as the interface that these objects should provide.

9.1.1 Primitives: Signed Quadric Surfaces

The modeling primitives are signed surfaces representing points in one of the regions that it bounds. A signed surface S consists of a quadratic polynomial $s(x, y, z)$ (sometimes called a quadric) and a sign bit. The zero set of this polynomial is the points on the surface S. The set $\{(x, y, z) \mid s(x, y, z) < 0\} \subseteq \mathbb{R}^3$ is the *negative half-space* of S, and $\{(x, y, z) \mid s(x, y, z) > 0\}$ is the *positive half-space*. A point $p \in \mathbb{R}^3$ has a *negative sense* with respect to S if p is in the negative half-space of S, and a *positive sense* if p is in the positive half-space. The sign bit for S indicates which half-space is the inside of the signed surface.

A signed surface S is traditionally stored as the ten coefficients of the polynomial

$$s(x, y, z) = A_1x^2 + A_2y^2 + A_3z^2 + A_4xy + A_5xz + A_6yz + A_7x + A_8y + A_9z + A_{10}. \quad (9.1)$$

Positive scalar multiples give the same surface; negative multiples reverse the positive and negative senses. Many "natural quadrics" have more compact representations, which can be used to accelerate and simplify computations. For the sake of brevity, for the remainder we will refer to a signed surface simply as a surface. In addition, when analyzing precision, we assuming that the coefficients of $s(x, y, z)$ are degree 1.

As an abstract data type, a surface must provide two tests described below. First, a surface must provide the `Inside`(S, q) tests, which returns true when a query point q is inside surface S. That is, when p has the same sense as the sign of S. In addition the surface must also provide the `classify`(S, b) test, which reports whether the points of an axis-aligned box b have the same sense, opposite sense, or both senses, with respect to the surface. In addition, the surface can optionally provide an *integrator* that calculates the volume of the primitive intersected with a box.

Planes and axis-aligned cylinders are common modeling primitives and have a greatly simplified `Inside` and `Classify` tests. As such, we found it worthwhile to implement them special cases in addition to general quadrics. In Section 9.2.1 we discuss an implementation and analyze the precision of both tests for general quadrics only, the special cases are more straightforward.

9.1.2 Component Hierarchy: Boolean Formulae

In many modeling tools, primitives are combined by Boolean operations[1] (union, intersection, difference, and complement) to describe more complicated regions. A multi-component model is described as a tree hierarchy in which each node represents a component defined by Boolean

[1]Often by regularized operations [111], which follow the operation by taking the boundary of the interior. Regularization does not change volume, but does remove lower dimensional features from an object.

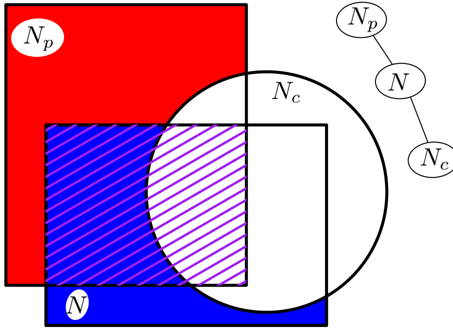


Figure 9.1: Hierarchy N_p-N-N_c, with comp. $C(N)$ striped purple and blue.

operations that is contained in its parent, disjoint from its siblings, and containing its children.

To be precise, there are three types of components described by an object represented as a node N in the hierarchy tree. A simple example is illustrated in Figure 9.1.

The *basic component* $B(N)$ is a region defined as a DNF (Disjunctive Normal Form, i.e., a union of intersections) formula of primitives that are bounded by quadric surfaces. $B(N)$ is a blue rectangle in the figure.

The *restricted component* $R(N)$, drawn as a purple-striped rectangle, is the intersection of $B(N)$ with the restricted component of its parent. Thus, restricted components nest, with the one at the root, N_0, always containing the entire model. The interiors of restricted components of siblings of N are assumed to be disjoint from that of N; it is the modeler's responsibility to enforce this. If we explicitly include the intersection of ancestors, a restricted component has a *3-level formula* (i.e., an intersection of unions of intersections).

The *hierarchical component* $C(N)$, striped purple and blue, is the restricted component $R(N)$ minus the restricted components of the children of N. We seek to compute the volumes of all hierarchical components in the tree.

As an abstract data type, the Boolean formulae and hierarchy tree F must provide the following interface. The `PointLocation`(F, q) construction returns the hierarchical component containing a query point q. The `Restriction`(F, R, b) construction takes a boolean formula

F, a box b, and the results R of the box classification test for all primitives and returns simplified formulae and tree by replacing primitives that are entirely outside or containing the box with false or true, respectively, and applying logical rules to obtain a equivalent formula without these logical constants. Note that these operations are entirely logic and data structure manipulation. As all the numeric evaluations are performed in the surfaces, these operations cannot introduce any numerical errors.

9.1.3 Problem Statement

The tree also drives the process of volume computation for the components, and we ask it to provide the *volume of each restricted component*. The volumes of the hierarchical components are then computed by having each parent subtract the volumes of its children.

To obtain these volumes, we extend the notion of an integrator from a surface to the entire hierarchy of components: an *integrator* is supposed to provide the volumes of a given box intersected with each restricted component in a hierarchy tree. Since it will do so by creating a spatial subdivision, which is a third hierarchical structure, we find the following sections clearer if we consider the problem of a single restricted component, so we have only a 3-level Boolean formula and an octree to distinguish between. Thus, in the following sections we focus on Problem 48. In fact, we evaluate all restricted components in the hierarchy tree together, which is important to prevent the subtractions from compounding errors.

Problem 48. Given an axis-aligned bounding box BB and a restricted component R, defined as a 3-level formula on signed surfaces $\{S_1, \ldots, S_n\}$, compute the volume of their intersection, $BB \cap R$, to specified precision.

9.2 Divide and Conquer Using Predicates

Here is a brief top-down description of the *divide and conquer integrator* for computing the volume of a restricted component (actually, the entire component hierarchy, as suggested in Section 9.1) using predicates mentioned in the previous section, and described in detail below.

The input is the Boolean formula on a set of surfaces defining primitives, and an initial axis-aligned bounding box. The procedure depends on a set of integrators for base cases, such as boxes that intersect a couple of surfaces; these are described in Section 9.3. The formula is first simplified by restriction to the box, which involves applying box classification for all surfaces; these predicates are detailed in the remainder of this section. If it is then simple enough, or the box is small enough, to apply a base case integrator, then this completes the computation. Otherwise the box is subdivided, and copies of the formula are given to each box; we use a subdivision into eight boxes, forming an *octree* [115] for our implementation and experiments. Evaluation on smaller boxes can be performed in parallel, as the only dependency is the summation of return values.

9.2.1 Surface/Axis-Aligned Box Classification

Before going into the details of the box classification test, we describe a predicate for determining if a conical curve is an ellipse and a construction for sampling a point inside an ellipse or ellipsoid.

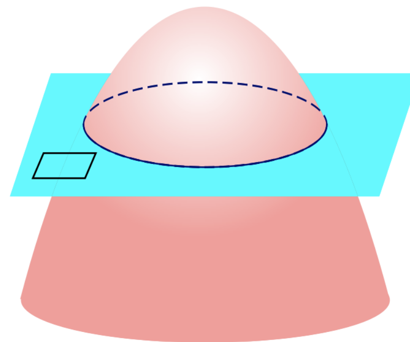


Figure 9.2: Intersection of a quadric with an axis-aligned plane.

Classify Conic

Recall that the intersection of a quadric and a plane is represented by the zero set of a *conic*. When the plane is orthogonal to a basis vector of the standard basis (as in Figure 9.2), computing

the resulting conic is easy. for example, the intersection of the plane $z = Z$ with a quadric is:

$$\begin{aligned} s(x,y,Z) &= A_1x^2 + A_2y^2 + A_3Z^2 + A_4xy + A_5Zx + A_6Zy + A_7x + A_8y + A_9Z + A_{10} \\ &= A_1x^2 + A_2y^2 + A_4xy + (A_5Z + A_7)x + (A_6Z + A_8)y + (A_3Z^2 + A_9Z + A_{10}) \\ &= \textcircled{1}\, x^2 + \textcircled{1}\, y^2 + \textcircled{1}\, xy + \textcircled{2}\, x + \textcircled{2}\, y + \textcircled{3}\, . \end{aligned}$$

This gives us a conic on the plane $z = Z$, which we can rewrite as a polynomial of degree 2 over two variables:

$$c(x,y) = B_1x^2 + B_2x^2 + B_3xy + B_4x + B_5y + B_6 = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} B_1 & \frac{B_3}{2} & \frac{B_4}{2} \\ \frac{B_3}{2} & B_2 & \frac{B_5}{2} \\ \frac{B_4}{2} & \frac{B_5}{2} & B_6 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = 0.$$

The coefficients B_1, B_2, and B_3 are degree 1, B_4 and B_5 are degree 2, and B_6 is degree 3. The polynomial can be written more conveniently as

$$c(x,y) = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} \textcircled{1} & \textcircled{1} & \textcircled{2} \\ \textcircled{1} & \textcircled{1} & \textcircled{2} \\ \textcircled{2} & \textcircled{2} & \textcircled{3} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = 0. \tag{9.2}$$

Conics represent ellipses, hyperbolas, parabolas, lines, etc. Levin [87] describes how to classify a conic by properties of its *discriminant*, the 3×3 symmetric matrix in the Equation 9.2. However, the classification requires high precision when computing eigenvalues of the discriminant.

For our purposes, we need only to determine if a conic is an ellipse. This observation greatly simplifies calculations. We define the `Ellipse`(c) test in the following lemma:

Lemma 49. Given a conic of form $c(x,y) = \textcircled{1}\, x^2 + \textcircled{1}\, y^2 + \textcircled{1}\, xy + \textcircled{2}\, x + \textcircled{2}\, y + \textcircled{3}$, `Ellipse`(c) returns true when c is a polynomial representing an ellipse using degree 5.

Proof. Let Q be the discriminant of c and Q_u be the upper left 2×2 matrix. Using Lawrence's conic classification [83], one can determine if a conic is an ellipse by verifying three criterion:

(a1) $|Q| \neq 0$

(a2) $|Q_u| > 0$

(a3) $B_2 * |Q_u| < 0$.

As $|Q| =$ $\textcircled{5}$ and $|Q_u| =$ $\textcircled{2}$, (a1) is degree 5 and (a2) is degree 2. Step (a3) tests the sign of $\textcircled{1}$ * $\textcircled{2}$, by first computing the signs of the two polynomials and comparing the signs we get that (a3) is degree 2. □

It follows from the above discussion that

Corollary 50. Given a surface S defined by degree 1 coefficients and a plane P orthogonal to a standard basis vector defined by a degree 1 coordinate, `Ellipse`(S, P) returns true when $S \cap P$ is a real ellipse using degree 5.

In our previous work [97] we did not consider or analyze the precision of our predicates and used Levin's classification directly.

Sample Point in an Ellipse And Ellipsoid

Next we describe how one can sample a point on the interior of an ellipse and an ellipsoid.

Lemma 51. Given an ellipse E formed by the intersection of a quadric S and an axis-aligned plane P, both with degree 1 coefficients, the `Sample`(S, P) construction returns a point in the interior of E with homogeneous coordinates of the form ($\textcircled{2}$, $\textcircled{3}$, $\textcircled{3}$) on the plane P

Proof. Recall that E is, a conic of the form

$$\begin{aligned} c(x,y) &= B_1x^2 + B_2y^2 + B_3xy + B_4x + B_5y + B_6 \\ &= \textcircled{1}\, x^2 + \textcircled{1}\, x^2 + \textcircled{1}\, xy + \textcircled{2}\, x + \textcircled{2}\, y + \textcircled{3}\, . \end{aligned}$$

The line $\ell_y := \frac{d}{dy}c(x,y) = 0$ passes thought the left and right most points of E and line $\ell_x := \frac{d}{dx}c(x,y) = 0$ passes though the top and bottom most points of E. Thus, the intersection point q of ℓ_x and ℓ_y is inside E. We compute q by solving the following linear system

$$\begin{pmatrix} 2B_1 & B_3 \\ B_3 & 2B_2 \end{pmatrix} \begin{pmatrix} q_x \\ q_y \end{pmatrix} = \begin{pmatrix} -B_4 \\ -B_5 \end{pmatrix},$$

which gives us that

$$q_x = \frac{B_3B_5 - 2B_2B_4}{4B_1B_2 - B_3^2} = \frac{\textcircled{1}\,\textcircled{2} - \textcircled{1}\,\textcircled{2}}{\textcircled{1}\,\textcircled{1} - \textcircled{2}} = \frac{\textcircled{3}}{\textcircled{2}} \qquad q_y = \frac{B_3B_4 - 2B_1B_5}{4B_1B_2 - B_3^2} = \frac{\textcircled{3}}{\textcircled{2}}.$$

Thus, q has homogeneous coordinates of the form ($\textcircled{2}$, $\textcircled{3}$, $\textcircled{3}$).

□

We use a similar technique as above to sample a point on the interior of an ellipsoid.

Lemma 52. Given an ellipsoid E defined by degree 1 coefficients, `Sample`(E) samples a point in the interior of E with homogeneous coordinates of the form ($\textcircled{3}$, $\textcircled{3}$, $\textcircled{3}$, $\textcircled{3}$).

Proof. The proof is similar as above. The lines $\frac{d}{dx}c(x,y) = 0$, $\frac{d}{dy}c(x,y) = 0$, and $\frac{d}{dz}c(x,y) = 0$

intersect at q inside the ellipse. As before, we compute q by solving the following linear system.

$$\begin{pmatrix} 2A_1 & A_4 & A_5 \\ A_4 & 2A_2 & A_6 \\ A_5 & A_6 & 2A_3 \end{pmatrix} \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} = \begin{pmatrix} -A_7 \\ -A_8 \\ -A_9 \end{pmatrix}$$

All entries of the matrix are degree 1, thus the determinant, which is the homogeneous coordinate, is degree 3. As each entry in the cofactor matrix is degree 2, we get

$$\begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} = \frac{1}{\textcircled{3}} \begin{pmatrix} \textcircled{2} & \textcircled{2} & \textcircled{2} \\ \textcircled{2} & \textcircled{2} & \textcircled{2} \\ \textcircled{2} & \textcircled{2} & \textcircled{2} \end{pmatrix} \begin{pmatrix} \textcircled{1} \\ \textcircled{1} \\ \textcircled{1} \end{pmatrix} = \frac{1}{\textcircled{3}} \begin{pmatrix} \textcircled{3} \\ \textcircled{3} \\ \textcircled{3} \end{pmatrix}.$$

Thus, q has homogeneous coordinates of the form $(\textcircled{3}, \textcircled{3}, \textcircled{3}, \textcircled{3})$ □

In our previous work [97], we suggested computing a sample point on the inside of an ellipse by taking the midpoint of the segment connecting the left and right most points. Also, we suggested computing a sample point in an ellipsoid by computing its silhouette curve in the z direction and projecting the curve down to the xy-plane. As the silhouette curve of an ellipsoid is an ellipse, we used the ellipse sampling procedure above to sampled a point p. We then computed the two intersection points of the line through p orthogonal to the xy-plane and took the sample point to be the midpoint of segment $\overline{p_1p_2}$.

The above discussion illustrates that sometimes examining predicates and considering precision as a resource results in simpler and more accurate primitives.

The Box Classification Test

Below we summarize the `Classify` test for a general quadric. Before proceeding, we will need one additional lemma about the `Inside` test.

Lemma 53. Given a surface S defined by degree 1 coefficients and a point q defined by degree 1 coordinates, `Inside`(S, q) returns if q is inside of S in constant time and degree 3.

Proof. Surface S is defined by a quadric (as in Equation 9.1). We can determine if q is inside S by taking the sign of $s(q_x, q_y, q_z)$, which is degree 3 as a polynomial in q_x, q_y, q_z and A_i for $i = 1, \ldots, 10$. □

Given an axis-aligned box b and a surface S, `Classify`(S, b) labels box b as ***Inside*** if all its points have the same sense as S, ***Outside*** if all its points have the opposite sense as S, or ***Intersecting*** if it contains points of both senses.

Lemma 54. Given a surface S defined by degree 1 coefficients and an axis aligned box with degree 1 corners, `Classify`(S, b) returns the classification of the sense of the box with respect to S in degree 5.

Proof. For a general quadric, degree 4 edge tests assign values from box corners to pairs of variables in a quadric, and evaluate the discriminant of the third variable. This algebra computes the vertex tests along the way. If these tests identify the box as Intersecting, we return that label, otherwise we must continue to test faces and containment since, for example, a cylinder or sphere could intersect a box boundary only at faces.

To test if a face f intersects surface S, First we use `Ellipse`(S, P) to check if the intersection of S and the plane P containing f (as in Figure 9.2) is an ellipse.

If $S \cap P$ is not an ellipse, it is unbounded. Since we already know that vertices and edges around f do not intersect S, the face also does not intersect S. If $S \cap P$ is an ellipse then we use `Sample`(S, P) to sample a point q on P and in the interior of the ellipse to detect intersection with face f. Detecting the intersection involves degree 4 comparisons of p and the corners of b. Again, if we detect Intersecting, we return that label.

Finally, if S is an ellipsoid, using degree 4, test if `Sample`(S) is inside b to ensure it has the same sense as all corners, and report the final classification label for b.

The highest degree predicate above is the degree 5 `Ellipse` test. Thus, `Classify`(S, b) is degree 5. □

9.2.2 Component/Box Classification and Restriction

Extending the classification of a box b from surfaces to components is a purely logical operation, and the restriction of the 3-level formula to b is just data structuring. We summarize this observation in the theorem

Theorem 55. Given a box b with degree 1 corners, we can (conservatively) determine if a box b is inside, outside or straddling a component with degree 5.

Proof. Recall that `Restriction`(F, R, b) takes a 3-level formula, a box b, the results R of the box classification test for all primitives. We can apply the formula to conservatively combine the labels of b by the following rules, where we have abbreviated X: Intersecting, I: Inside, and O: Outside, and α: any.

complement $\overline{X} = X,\ \overline{I} = O$, and $\overline{I} = O$.

union $X \cup X = X,\ O \cup \alpha = \alpha$, and $I \cup \alpha = I$.

intersection $X \cap X = X,\ O \cap \alpha = O$, and $I \cap \alpha = \alpha$.

This is a conservative extension because the union of two objects that intersect box b may actually completely cover b, or their intersection within b may be empty. By classifying all such cases as Intersecting, however, the octree will simply refine the box and retest. Thus, the result can be guaranteed correct to the level of refinement. □

To avoid classifying every child box for every surface, we need to simplify formula F by restriction to box b. We describe how this can be done recursively, while exploring the octree in depth-first order. At any step we have a box b and the 3-level formula for $b \cap F$ that uses only surfaces that were deemed Intersecting the parent of b.

Initially we begin with a bounding box and the entire formula, represented as a tree of height three. (Recall that F is intersections of unions of intersections of surface primitives; each operator is represented by a list of pointers to the formulae or variables that it operates on.) We evaluate b with respect to each surface remaining in the formula. If b is Inside or Outside any surface, apply the above union/intersection operations (with commutativity and associativity) to simplify the formula. If the formula simplifies to Inside or Outside, or if we decide for some other reason to stop traversing at box b, then we are done. Otherwise, we refine box b, pass the simplified formula to each child, and recursively evaluate each child.

In this recursive algorithm it is imperative that any surface that intersects a box b must be detected as intersecting the parent of b. For example, if faces were not tested in the box classification test and a cylinder was missed because it pierced only a face of b, then it would be simplified away, and not be evaluated in any children of b.

9.3 Volume Estimators

An *integrator* determines if the geometry inside a box is sufficiently simple that it can evaluate the volumes in a component hierarchy; integrators serve as the base cases for the recursive divide & conquer algorithm.

Each integrator receives as input a box, an error interval and confidence value expressed either in absolute terms or relative to the volume of the box, and a component hierarchy. It can look at the size of the box, the number and kind of surfaces in the hierarchy, and its task is to evaluate the volumes of the relative components in the box to within the error interval with given confidence. We first describe several possible integrators, then return to the analysis of error intervals. As almost all integrators produce an approximate to an exact volume we do not analyze integrators in terms of their degree.

The Box Integrator The simplest integrator assigns a user-specified fraction of box volume

to each component that can intersect a box in the spatial hierarchy. This is exact when no surface passes through a box, since then the box is entirely inside one component. Otherwise the confidence is directly proportional to the volume of the box, so this can always be a fallback integrator when boxes become small.

The General Integrator: Monte Carlo A Monte Carlo integrator simply performs point-in-component tests. We obtain confidence bounds using the Wilson test [137], which, at the 95% confidence level suggests that the volume v of a component that receives x out of n samples is in the confidence interval

$$\frac{x+2}{n+4} \pm \frac{2\sqrt{x(1-x)/n+1}}{n+4}$$

We estimate the volume as $v = x/n$ so that the total volume within the box is conserved.

Single Plane Integrator When a single surface defines the portion of a component in an axis aligned box, by some case analysis (often a nontrivial amount) we can derive the domains of integration for the component's volume, which we can then evaluate numerically or analytically. Single planes are the easiest and most frequent case, and their implementation shows benefits immediately.

Pairs of Planes Integrator Extending the single quadric integrator to a pair is also possible. Again, the biggest payoff for the effort comes from handling pairs of planes – by implementing a special integrator for this case, any box that contains two planes becomes a base case and needs not be further subdivided.

Capped Cylinder Integrator As elliptic cylinders are infinite in one direction, they are often truncated by some other quadric (usually a plane). We found implementing a special integrator for an axis-aligned cylinder truncated by a plane orthogonal to its axis showed benefits. Untruncated axis aligned cylinders are also handled by this integrator.

Bundle of Cylinders Integrator Because bundles of axis-aligned pins and cylindrical containment vessels are so common in models, we found it worthwhile to implement a special integrator for pairwise-disjoint collections of axis-aligned cylinders (capped and uncapped) with circular cross sections. Intersecting cylinders are reduced to this case by subdivision.

9.3.1 Error bounds

One concern that arises with the spatial subdivision of the problem is how to combine error bounds from individual boxes. In the worst case, bounds on the errors from each box would have to sum: an error of $\pm\delta$ from K boxes would become an error of $\pm\delta K$. This worst case occurs if all errors are not independent and occur in the same direction, perhaps by a rounding problem in an iterator. Careful coding is needed to avoid accumulation of such error.

We get an average case if the errors are independent; the error bound from a combination would grow more like $\pm\delta\sqrt{K}$. The easiest illustration of this is to imagine a Monte Carlo volume estimate that chooses random points in each box and decides if they are inside or outside the volume, giving independent Bernoulli trials. It could instead choose points from the union of all boxes with the variance going from $np(1-p)$ for each of K boxes to $Knp(1-p)$ overall, so the standard deviation increases by $\sqrt{K}$.

The best case is also relevant – dependent errors can cancel if they are made in opposite directions, and we actually benefit from that by evaluating the entire component hierarchy at once. Consider Monte Carlo estimates again: if we determine which component contains a trial point, then the estimates to volume will at least add up to the volume of the box (up to machine precision) so that volume lost by one component will be gained by another and the total volume will be conserved.

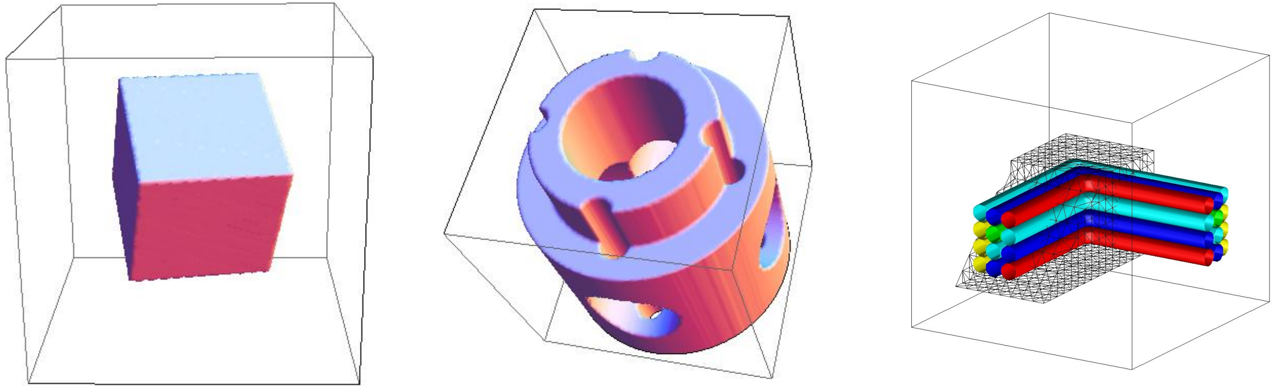


Figure 9.3: *Left:* Cube is ≈ 0.125 volume cube with a random rotation. *Center:* DrillCyl is a union of two intersections of a dozen cylinders and planes. *Right:* cPiped12 shown is a wedge-shaped block minus a 3×4 bundle of L-shaped pipes; we also test a 10×10 bundle of L-pipes in the model cPiped100 and a 100×100 bundle of L-pipes in the model cPiped10000.

9.4 Experiments

In this section we demonstrate how adding integrators in our framework can reduce the running time and/or increase the accuracy of volume computation on a set of models shown in Figure 9.3. Timings are from a 3.2GHz Intel Xeon processor with 12GB RAM running Ubuntu 10.04.

We ran several experiments on each of five models, three of which are shown in Figure 9.3 (images of the fourth and fifth models are omitted as they are visually similar to the right-most image). Each model is inside a cube of unit volume and tolerances refer to this volume. Cube is a randomly rotated $0.5 \times 0.5 \times 0.5$ cube, with volume approximately 0.125 (randomly rotating the cube induces a small error, reducing the volume by approximately 1e-07). We use this example to assess the accuracy of combinations of plane and Monte Carlo integrators. DrillCyl is a union of intersections of capped cylinders with drilled holes. We use this example to assess the performance of combinations of integrators on models with curved primitives. cPiped12 is a wedge with twelve L-shaped pipes of radius $.035$ removed, defined by 55 surfaces and 31 unions of intersections; we also do a larger models, cPiped100, with 100 L-shaped pipes of radius $.01$, defined by over 475 surfaces and 200 unions of intersections, and cPiped10000, with 10,000 L-shaped pipes of radius

Table 9.1: Numbers of boxes evaluated by different integrators for different models and tolerances, with volume & timing. Cube has no cylinders so we omit +Cyl; the other models have few planes, so we omit the +1 Plane line.

Model Name	Alg	Total Boxes	Integrators (% of total boxes)						Total Volume	Time sec
			Box	MC	1pl	2pl	Cyl	Bun		
Cube	MC	1	-	100.0	-	-	-	-	0.1245446	0.09
tol: ±2.2e-03	+octree	20,280	57.4	42.6	-	-	-	-	0.1251001	0.02
vol: 0.1249998	+1 Plane	4,768	58.6	15.4	26.0	-	-	-	0.1249996	<.01
	+2 Plane	610	64.3	3.0	9.8	23.0	-	-	0.1250019	<.01
Cube	MC	1	-	100.0	-	-	-	-	0.1249980	131.12
tol: ±1.1e-04	+octree	2,060,7294	57.1	42.9	-	-	-	-	0.1250000	18.16
vol: 0.1249998	+1 Plane	168,428	57.3	14.3	28.4	-	-	-	0.1249998	0.37
	+2 Plane	1,380	66.5	1.2	10.0	22.3	-	-	0.1249998	<.01
DrillCyl	MC	1	-	100.0	-	-	-	-	0.3867179	0.20
tol: ±2.2e-03	+octree	73,200	55.3	44.7	-	-	-	-	0.3866701	0.08
vol: 0.3866281	+2 Plane	66,144	55.1	43.2	1.3	0.4	-	-	0.3868198	0.08
	+Cyl	10,872	48.2	15.5	1.5	2.2	32.7	-	0.3866121	0.02
DrillCyl	MC	1	-	100.0	-	-	-	-	0.3866408	286.00
tol: ±1.1e-04	+octree	78,737,968	57.1	42.9	-	-	-	-	0.3866279	73.62
vol: 0.3866281	+2 Plane	67,367,700	57.1	42.8	<0.1	<0.1	-	-	0.3866278	62.73
	+Cyl	378,512	54.2	14.1	1.3	2.7	27.6	-	0.3866278	0.65
cPiped12	MC	1	-	100.0	-	-	-	-	0.0660653	0.13
tol: ±2.2e-03	+octree	24,753	48.6	51.4	-	-	-	-	0.0655403	0.03
vol: 0.0657512	+2 Plane	13,784	43.1	53.4	1.9	1.5	-	-	0.0658770	0.03
	+Cyl	8,667	39.3	32.7	2.5	2.4	23.1	-	0.0657460	0.02
	+Bun	3,322	44.5	22.3	6.3	6.2	12.9	8.1	0.0657594	0.02
cPiped12	MC	1	-	100.0	-	-	-	-	0.0657498	192.75
tol: ±1.1e-04	+octree	34,070,947	56.6	43.4	-	-	-	-	0.0657512	32.73
vol: 0.0657512	+2 Plane	20,543,405	56.2	43.7	<0.1	<0.1	-	-	0.0657509	20.05
	+Cyl	1,296,975	39.4	23.1	0.9	1.0	35.7	-	0.0657511	2.36
	+Bun	198,090	52.0	15.1	5.9	6.6	19.4	1.0	0.0657512	0.56
cPiped100	MC	1	-	100.0	-	-	-	-	0.0731463	0.60
tol: ±2.2e-03	+octree	23,003	37.6	62.4	-	-	-	-	0.0731258	0.07
vol: 0.0731920	+2 Plane	11,936	21.0	75.6	2.6	0.8	-	-	0.0733605	0.06
	+Cyl	11,887	20.7	68.2	2.6	0.8	7.6	-	0.0732219	0.06
	+Bun	3,228	34.8	28.7	7.8	3.1	2.3	24.0	0.0732155	0.03
cPiped100	MC	1	-	100.0	-	-	-	-	0.0731951	790.28
tol: ±1.1e-04	+octree	62,392,744	54.8	45.2	-	-	-	-	0.0731921	63.96
vol: 0.0731920	+2 Plane	48,958,575	54.2	45.6	<0.1	<0.1	-	-	0.0731919	51.32
	+Cyl	8,527,009	38.4	25.2	0.3	0.4	35.7	-	0.0731919	14.58
	+Bun	482,756	49.6	16.8	4.8	7.0	18.5	3.3	0.0731919	1.41
cPiped10000	MC	1	-	100.0	-	-	-	-	0.0768654	183.04
tol: ±2.2e-03	+octree	23,003	43.6	56.4	-	-	-	-	0.0767527	2.40
vol: 0.0767715	+2 Plane	11,936	32.4	63.4	3.3	0.8	-	-	0.0768464	2.33
	+Cyl	11,887	32.2	63.6	3.3	0.8	<0.1	-	0.0768258	2.34
	+Bun	3,228	39.5	25.0	9.6	3.1	0.3	23.1	0.0767881	2.36
cPiped10000	MC	-	-	-	-	-	-	-	-	NA[a]
tol: ±1.1e-04	+octree	208,125,506	33.0	67.0	-	-	-	-	0.0767697	358.09
vol: 0.0767715	+2 Plane	195,211,080	31.4	68.6	<0.1	<0.1	-	-	0.0767696	348.25
	+Cyl	162,382,473	23.1	43.7	<0.1	<0.1	33.2	-	0.0767696	346.37
	+Bun	1,539,063	30.6	30.3	0.3	4.9	13.0	20.9	0.0767691	9.43

[a] Halted after 12 hours. Extrapolating from cPiped10000 with tolerance ±2.2e-03, time will be about 76 hours.

.001 defined by over 40,800 surfaces and 20,400 unions of intersections.

For each model we evaluated volumes with tolerances of $\pm$ 2.2e-03 and $\pm$ 1.1e-04. In our current implementation, tolerance is used primarily to determine the number of Monte Carlo samples needed over the unit volume, which in turn determines the maximum depth of our octree, since the number of samples in a box is proportional to its volume, and we require at least eight points in a box to subdivide it. Thus with tolerance 2.2e-03 we stop at depth 6, and with 1.1e-4 at depth 11. Descending further can change which integrators run. For example, in cPiped100 with tolerance $\pm$ 2.2e-03 Cyl integrates 7.6% of the boxes in volume calculation. However, when we explore the tree further with tolerance $\pm$1.1e-04, we find that Cyl integrates about 35.7% of the boxes. Because Cyl only integrates capped and uncapped single cylinders we need to travel further down the tree before we isolate a cylinder.

Each line of the table reports statistics of a run using all the integrators above that line, which explains the triangular pattern of numbers (e.g., the last line for each model uses all integrators). Table 9.1 reports the number of boxes on which integrators are run, and reports, for each type of integrator, the percentage of boxes that it evaluated. The number of boxes has a big effect on running time, but is not the whole story. Table 9.2 shows the drastic decrease in the number of samples required by the Monte Carlo integrator as some of its boxes are given to other integrators (the number of samples is directly proportional to the volume of the boxes in which MC integration is performed). Adding other integrators decreases the number of boxes for the MC integrator by a factor of 2, but the volume of boxes integrated by MC decrease by factors of more than 10. An integrator that is added but not used in many boxes can still reduce the number of boxes dramatically by serving as an early base case.

We observe that in all cases, the volume calculated with additional integrators is within the error tolerance, but between 1 and 5 orders of magnitude faster than MC. We attribute the speedup

mainly to the reduced number of MC samples, thus, even for course volume estimates with tolerance 2.2e-03, we see the benefit of additional integrators.

9.5 Conclusions

We described a divide and conquer framework for computing volumes of CSG models that supports adding special integrators for handling common or simple subproblems to decrease time and increase accuracy. The application to volume computation is just one of many possible; it is easy from our current structure to label boxes of the octree with components of the hierarchy that straddle them; such a data structure may more quickly locate and track particles while solving the transport equation.

Our exploration of error bounds and their propagation is rudimentary at this point, and much more can be said about decisions that affect the error from the various integrators. For example, if we start off with a bounding box whose minimal point's coordinates and side lengths are powers of two, each octree cell could have an exact representation. Thus, the volume of outside boxes would be exactly zero, and inside would be at most 3ulp away from the true box volumes. Tight error bounds for the other integrators is not so easy. Perhaps by keeping tight bounds on the numerical errors introduced by an integrator we may make a more informed traversal of the octree.

Table 9.2: Percentages of the unit volumes integrated by the different integrators; the decrease in Monte Carlo volume is directly related to the number of samples and running time.

Model	Alg	Integrators (% of total volume)						Total	Total
Name		Box	MC	1pl	2pl	Cyl	Bun	Samples	sec
Cube	MC	-	100.0	-	-	-	-	999,995	0.09
tol: ±2.2e-03	+octree	96.7	3.3	-	-	-	-	34,528	0.02
vol: 0.1249998	+1 Plane	80.7	0.3	19.0	-	-	-	2,928	<.01
	+2 Plane	61.1	<0.1	9.3	29.6	-	-	72	<.01
Cube	MC	-	100.0	-	-	-	-	1,410,065,909	131.12
tol: ±1.1e-04	+octree	99.9	0.1	-	-	-	-	17,663,096	18.16
vol: 0.1249998	+1 Plane	80.9	<0.1	19.1	-	-	-	48,176	0.37
	+2 Plane	61.1	<0.1	9.3	29.6	-	-	32	<.01
DrillCyl	MC	-	100.0	-	-	-	-	999,995	0.20
tol: ±2.2e-03	+octree	87.5	12.5	-	-	-	-	130,784	0.08
vol: 0.3866281	+2 Plane	83.7	10.9	4.6	0.8	-	-	114,256	0.08
	+Cyl	36.1	0.6	2.5	0.8	60.0	-	6,720	0.02
DrillCyl	MC	-	100.0	-	-	-	-	1,410,065,909	286.00
tol: ±1.1e-04	+octree	99.6	0.4	-	-	-	-	67,494,688	73.62
vol: 0.3866281	+2 Plane	94.2	0.3	4.7	0.8	-	-	57,664,680	62.73
	+Cyl	36.5	<0.1	2.5	0.8	60.2	-	106,576	0.65
cPiped12	MC	-	100.0	-	-	-	-	999,995	0.13
tol: ±2.2e-03	+octree	95.1	4.9	-	-	-	-	50,932	0.03
vol: 0.0657512	+2 Plane	70.4	2.8	6.2	20.6	-	-	29,468	0.03
	+Cyl	69.2	1.1	6.2	20.6	2.9	-	11,332	0.02
	+Bun	68.3	0.3	6.2	20.6	0.9	3.8	2,964	0.02
cPiped12	MC	-	100.0	-	-	-	-	1,410,065,909	192.75
tol: ±1.1e-04	+octree	99.8	0.2	-	-	-	-	29,604,860	32.73
vol: 0.0657512	+2 Plane	73.0	0.1	6.2	20.6	-	-	17,953,588	20.05
	+Cyl	69.8	<0.1	6.2	20.6	3.4	-	598,522	2.36
	+Bun	68.5	<0.1	6.2	20.6	1.0	3.8	59,964	0.56
cPiped100	MC	-	100.0	-	-	-	-	999,995	0.60
tol: ±2.2e-03	+octree	94.5	5.5	-	-	-	-	57,392	0.07
vol: 0.0731920	+2 Plane	72.2	3.4	6.8	17.6	-	-	36,100	0.06
	+Cyl	72.2	3.1	6.8	17.6	0.4	-	32,440	0.06
	+Bun	70.4	0.4	6.7	17.6	<0.1	4.9	3,700	0.03
cPiped100	MC	-	100.0	-	-	-	-	1,410,065,909	790.28
tol: ±1.1e-04	+octree	99.7	0.3	-	-	-	-	56,352,288	63.96
vol: 0.0731920	+2 Plane	75.3	0.3	6.8	17.6	-	-	44,694,892	51.32
	+Cyl	73.7	<0.1	6.8	17.6	1.9	-	4,295,224	14.58
	+Bun	70.5	<0.1	6.7	17.6	0.1	5.0	162,002	1.41
cPiped10000	MC	-	100.0	-	-	-	-	999,995	183.04
tol: ±2.2e-03	+octree	95.0	5.0	-	-	-	-	51,920	2.40
vol: 0.0767715	+2 Plane	72.7	2.9	6.8	17.6	-	-	30,280	2.33
	+Cyl	72.7	2.9	6.8	17.6	<0.1	-	30,220	2.34
	+Bun	70.4	0.3	6.7	17.6	<0.1	4.9	3,232	2.36
cPiped10000	MC	-	-	-	-	-	-	-	NA [a]
tol: ±1.1e-04	+octree	98.4	1.6	-	-	-	-	279,088,846	358.09
vol: 0.0767715	+2 Plane	74.0	1.6	6.8	17.6	-	-	267,848,220	348.25
	+Cyl	73.7	0.8	6.8	17.6	1.1	-	141,769,844	346.37
	+Bun	70.5	<0.1	6.7	17.6	<0.1	5.1	931,534	9.43

[a] Halted after 12 hours. Extrapolating from cPiped10000 with tolerance ±2.2e-03, time will be about 76 hours.

Chapter 10

Using Degree-driven Algorithm Design to Achieve EGC

Recall from Chapter 2 that the Exact Geometric Computation (EGC) paradigm dictates that an algorithm's control flow should be independent of the machine on which the implementation is run; in particular, it should be the same as if the algorithm was implemented with real arithmetic. EGC's strength can be seen in its acceptance in software development. Well-known open-source software libraries CGAL [2] and LEDA [19] both have geometric kernels supporting EGC. Thus, it is worthwhile to conclude this thesis with a discussion of how degree-driven geometric algorithm design fits in with the most popular techniques for achieving the goals of EGC.

This thesis investigated the degrees of geometric primitives, predicates, constructions, and algorithms. By taking the degree of predicates and constructions into account at design time, we can reveal if we can implement an algorithm on native hardware, even when we cannot, we know what precision is sufficient. When we need more than native precision, the degrees of polynomials can help to select the appropriate techniques from the EGC literature. For the remainder of the chapter, I summarize the four most popular techniques for achieving the goals of EGC and discuss how degree analysis supports each technique to correctly evaluate the sign of a given polynomial corresponding to a predicate.

10.1 EGC Technique 1: Static Analysis

Given a polynomial corresponding to a predicate a programmer can carry out *static analysis* to determine the number of bits needed to correctly evaluate the sign. From the number of bits, the programmer can determine how large the coordinates of the input can be.

Recall from Chapter 3 how we can use degree to upper bound the required precision. We defined predicates as signs of multivariate polynomials with variables from the input. In this thesis, the coefficients of the monomials in the predicates are small constants with absolute value ≤ 8 (this is often the case for geometric predicates). Assume that the input coordinates are represented by b-bit integers. A degree k monomial with coefficient 1 can be represented with bk bits. Let δ be the number of bits needed to represent the coefficients. The monomials of a predicate can be represented with $bk + \delta$ bits. Let the number of variables in a predicate be s. The number of monomials is $O(s^k)$ and a polynomial of degree k can be evaluated with $bk + \delta + k \log_2 s$ bits. For predicates, we are only interested in the sign. In Appendix A we see how the sign can be computed exactly with only $(bk + \delta)$ bits.

With an upper bound on the number of arithmetic bits needed to evaluate a predicate we can determine the maximum number of bits b that can be used to represent the coordinates of the input. Assume that our machine supports w-bit arithmetic. We compute $b = \lfloor (w - \delta)/k \rfloor$. For example on a machine supporting 64-bit arithmetic, the degree 2 and degree 3 algorithms in Chapters 6-8 support reasonably sized input coordinates – approximately 30-bit and 20-bit coordinates, respectively.

10.2 EGC Technique 2: Software Implementations

Given a polynomial corresponding to a predicate a *software implementation* evaluates predicates using software libraries to simulate arbitrary precision arithmetic. The libraries vary in the number

types that they support. Some of the more popular libraries are listed below. GMP [50] supports arbitrary precision arithmetic for signed integers, rationals and floating point numbers. MPFR [46] is based on GMP and supports multiple-precision floating-point and has rounding semantics similar to that of IEEE-754. CORE [71] and LEDA [19] support arbitrary precision rational, floating point and algebraic numbers.

To the programmer, the benefits of software implementations are that they abstract the number types. The abstraction can reduce debugging and ease implementation. The programmer must be careful though. Karasick *et al.*[72] pointed out and Held and Mann [57] demonstrated that simply plugging in a library without care can cause slowdowns of over $10,000$x over hardware arithmetic. Yap [140] pointed out that with some tuning, often the predicate evaluation can be reduced to less than 10x slowdown. (Assuming that an algorithm spends 25% of the time evaluating predicates, a 10x slowdown causes the implementation to run three times longer.) Degree-driven algorithm design can help with tuning by bounding the number of bits computed in software.

In software implementations, precision is bounded only by memory. For very large memory bound problems, one must decide to trade memory for precision. As degree-driven algorithm design can be used to bound the number of bits for a predicate evaluation, it provides additional data for making the decision.

10.3 EGC Technique 3: Arithmetic Filters

Given a polynomial corresponding to a predicate an *arithmetic filter* computes an error bound on the polynomial's value and uses the error bound and value to determine if the sign is positive, negative, or failure. Failure is returned when the value is smaller than the error bound. Arithmetic filters are studied in theory [33, 34] and in practice[14, 18, 45].

Filters benefit programmers by allowing some control when an input is "close" to degenerate. For example, in a personal communication, Gary Miller reported using filters in programs for

physical simulation and meshing where the program selects query points. If the query caused a filter to fail, the program selects a different query point. The downside of filters is that unless error can be tightened to zero the filter cannot determine if a predicate is zero.

Brönnimann *et al.*[14] classified filters into three types: fully-static, semi-static, and dynamic. The differences are the values that the variables can take, the allowed arithmetic operations, and the approach to bounding error. Filter types are listed in order of increasing runtime overhead (decreasing speed), and trade speed for flexibility. For a *fully-static* filter [45], the variables are a bounded interval of integers, the operations are $\{+, -, \times\}$, and error bounds are precomputed at compile time. For a *semi-static* filter [18], the variables are an unspecified interval of floating point values, the operations are $\{+, -, \times, \div, \surd\}$, and error bounds are computed at runtime via a formula constructed at compile time. For a *dynamic* filter [14], the variables are an unspecified interval of floating point values, the operations are any operation that can produce an upper and lower bound, and error bounds are computed at run time using interval arithmetic.

To use static filters, code must have a clean separation between predicates and algorithms. Degree-driven algorithm design forces the designer to flesh out the separation at design time. For semi-static filters, the separation simplifies working out the error bounding formulas. For fully-static filters, error bounds follow directly from the degree analysis. Dynamic filters are the only type of filter supporting cascading constructions. While cascading constructions are sometimes necessary, the fact remains that they should be avoided when possible; degree-driven algorithm design reminds us of this fact.

10.4 EGC Technique 4: Adaptive Evaluation

Given a predicate, an *adaptive evaluation* attempts to reduce the number of bits used for computing the sign while guaranteeing that the sign is the same as in Real-RAM. Adaptive evaluations were considered for specific predicates [8, 123] (such as `InCircle`), predicates that can be expressed

as the signs of determinants [5, 27], and as signs of polynomials [15, 105].

Recall from earlier in the chapter that a degree k polynomial, of s variables of b bits, whose coefficients can be represented with δ bits can be evaluated with $bk+\delta+k\log_2 s$ bits. Clarkson [27] uses orthogonalization and approximate arithmetic to compute the sign of a $d \times d$ determinant with b-bit integer entries using $(2b + 1.5d)$ bits. Avnaim *et al.* [5] describe how to compute the sign of 2×2 and 3×3 determinants with b-bit integer entries with b and $b + 1$ bits respectively in $O(b)$ time. Brönnimann *et al.* [15] use modular arithmetic to compute the sign of multivariate polynomials using floating point computations and single precision arithmetic.

The most famous adaptive evaluations are described by Shewchuk [123] for the `InCircle` and `Orientation` predicates. The algorithms used for addition and multiplication in the evaluation borrows heavily from Priest [110]. Shewchuk extended and sped up Priest's work by considering a smaller set of architectures and rounding modes. More recently, Bernstein and Fussell [8] used ideas that paralleled Shewchuk's to implement the predicates in their modeling system. Nanevski, Blelloch, and Harper [105] pointed out that extending Shewchuk's work can be cumbersome and error prone. For example, the evaluator for `InSphere`, which is sign of a 4×4 matrix, is about 580 lines of C code. Moreover, the programmer needs to analyze the error for every polynomial. Nanevski, Blelloch, and Harper describe a system that automatically generates adaptive predicates for an arbitrary polynomial expressions consisting of addition, subtraction, multiplication, and squaring.

Like software implementations, adaptive evaluations can determine if a predicate is zero. In addition, adaptive evaluations can sometimes avoid higher precision calculations. Unlike filters, adaptive evaluations always returns the same result as Real-RAM. Perhaps the only negative to adaptive evaluations is that the set of arithmetic operations is more limited than dynamic filters and software implementations. For the predicates in this thesis, however, the limitation is not an issue.

As was the case with filters, to use adaptive evaluations the code must have a clean separation between predicates and algorithms. Degree-driven algorithm design forces the designer to flesh out the separation at design time, easing the development of adaptive evaluations.

10.5 Conclusion

This chapter described the four most popular techniques for implementing EGC. The techniques can be very efficient when used together. For example, the filtered kernel in CGAL [2] by default uses semi-static and dynamic filters. For the most critical predicates, static filters are also available. When all filters fail, EGC is achieved by resorting to software implementations.

The predicates in this thesis are signs of polynomials. Since we use only the operations plus, minus, times, and squaring and avoid cascading constructions, we do not need the extra flexibility provided by dynamic filters. For implementing the predicates in this thesis, I would suggest using static analysis. If not enough bits are available, a good second choice is automatically generated adaptive predicates or static/semi-static filters with software implementations. The libraries GMP, MPFR, CORE, and LEDA all provide the operations we require, however, since we do not need to represent algebraic numbers GMP and MPRF are sufficient.

Appendix A

Sign of a Sum

Given a set of n signed β-bit integers $A = \{a_1, \ldots, a_n\}$. We can compute $\text{sign}(\sum_i a_i)$ by first summing all a_i and taking the sign, however, that may require $\beta + \log_2 n$ bits. In this section, I describe a simple algorithm for computing the sign of the sum exactly with β bits.

Kaltofen and Villard [70] survey methods for computing the sign of the determinant of an integer matrix and observe that computing the sign is at most as hard as computing the value. The same observation holds for computing the sign of the sum of a set of integers. The observation leads to a simple algorithm that removes $\log_2 n$ bits used for summing. It is based on *numerical summation* algorithms, which avoid rounding and truncation errors for summing a set of numbers. Some languages provide numerical summation algorithms, such as `fsum` in Python. Unfortunately, the are not standard in all languages and they do more work than is necessary to compute signs. Algorithms such as compensated sum [69], storing partial sums [123], or recursive summation [59], could be considered. However, as we are interested only in the sign of the sum, we can use an even simpler summation algorithm that produces an exact sign of a set of n signed β-bit integers A.

Think of the non-zero elements of A as if they were in two lists: P, the positives, and N, the negatives. Initialize summation variable $r = 0$, the sum of all numbers of A, not in N or P. We ensure a correct sign evaluation by maintaining the invariant that $r \in [\min(N), \max(P)]$. When r is non-negative we set r to be r plus the next value from N, and when r is non-positive we set r to be r plus the next value from P. We stop the sum once the sign can be determined, that is, when r is non-positive and values of P are exhausted or r is non-negative and all values of N are exhausted. Note that all the values of A are summed only when $\text{sign}(A) = 0$, which is necessary

if we wish to find the sign using only β bits. I flesh out the details in the two algorithms below.

First, define an algorithm $\texttt{Scan}(A, i, sign)$, in Algorithm 5, that takes an array of signed integers A, an index i and a sign $s \in \{-1, 1\}$, and returns the index $j \geq i$ with sign of $A[j]$ matching s or return $j = A.\text{length}$ if no sign matches. (Zeros in A will never match s.)

Algorithm 5 $\texttt{Scan}(A, i, s)$

```
1: j = i
2: while j < A.length and s * A[j] < 1 do
3:     j = j + 1
4: end while
5: return j
```

Second, define an algorithm $\texttt{SignOfSum}(A)$, in Algorithm 6, that iterates over N and P (in place) selecting elements from N and P as appropriate and summing to update r.

Algorithm 6 $\texttt{SignOfSum}(A)$: return the sign of the sum of the elements of array A where each element is represented by β bits, using β bits.

```
1: r = A[0]
2: n = Scan(A, 1, −1)
3: p = Scan(A, 1, +1)
4: while (r ≤ 0 and p < A.length) or (r ≥ 0 and n < A.length) do
5:     if r < 0 then
6:         r = r + A[p]
7:         p = Scan(A, p + 1, +1)
8:     else
9:         r = r + A[n]
10:        n = Scan(A, n + 1, −1)
11:    end if
12: end while
13: return sign(r)
```

We summarize in the following lemma:

Lemma 56. Given a set of n signed β-bit integers, we can compute the sign of the sum of the set in $O(n)$ time, constant space, and β bits.

Bibliography

[1] Autodesk Maya. `http://usa.autodesk.com/maya`.

[2] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[3] P. Alliez, O. Devillers, and J. Snoeyink. Removing degeneracies by perturbing the problem or perturbing the world. *Reliable Computing*, 6(1):61–79, 2000.

[4] F. Aurenhammer. Voronoi diagrams a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[5] F. Avnaim, J. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[6] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 211–219, New York, NY, USA, 1995. ACM Press.

[7] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.

[8] G. Bernstein and D. Fussell. Fast, exact, linear Booleans. *Computer Graphics Forum*, 28(5):1269–1278, July 2009.

[9] M. Bôcher. *Introduction to higher algebra*. Macmillan, 1907.

[10] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM Journal on Computing*, 29(5):1401–1421, 2000.

[11] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. *Computational Geometry*, 16(1):35–52, 2000.

[12] S. M. Bowman. Overview of the SCALE code system. In *The American Nuclear Society and the European Nuclear Society 2007 International Conference on Making the Renaissance Real*, number 94, pages 589–591. American Nuclear Society, 2007.

[13] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman. Linear time Euclidean distance algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:529–533, 1995.

[14] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1-2):25–47, 2001.

[15] H. Brönnimann, I. Z. Emiris, V. Y. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 174–182. ACM Press, 1997.

[16] K. Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223–228, 1979.

[17] M. Brubeck. Fortune's algorithm in C++. `http://www.cs.hmc.edu/~mbrubeck/voronoi.html`, June 2012.

[18] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 175–183. ACM Press, 1998.

[19] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 418–419. ACM Press, 1995.

[20] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*.

[21] T. M. Chan. Reporting curve segment intersections using restricted predicates. *Computational Geometry*, 16(4):245–256, 2000.

[22] T. M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 152–159. ACM Press, 2004.

[23] T. M. Chan. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Computational Geometry Theory and Applications*, 35(1):20–35, 2006.

[24] T. M. Chan, D. L. Millman, and J. Snoeyink. Discrete Voronoi diagrams and post office query structures without the InCircle predicate. In *Proceedings of the Nineteenth Annual Fall Workshop on Computational Geometry*, pages 33–34, 2009.

[25] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: point location in sublogarithmic time. *SIAM Journal on Computing*, 39(2):703, 2009.

[26] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. A singly exponential stratification scheme for real semi-algebraic varieties and its applications. *Theoretical Computer Science*, 84(1):77–105, 1991.

[27] K. L. Clarkson. Safe and effective determinant evaluation. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 387–395. IEEE Computer Society, 1992.

[28] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(1):387–421, 1989.

[29] G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. *Lecture Notes in Computer Science*, 33:134–183, 1975.

[30] T. T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 3rd edition, 2009.

[31] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational geometry: algorithms and applications*. Springer-Verlag, 3rd edition, 2008.

[32] M. de Berg, D. Halperin, and M. Overmars. An intersection-sensitive algorithm for snap rounding. *Computational Geometry Theory and Applications*, 36(3):159–165, 2007.

[33] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete & Computational Geometry*, 20(4):523–547, 1998.

[34] O. Devillers and F. P. Preparata. Further results on arithmetic filters for geometric predicates. *Computational Geometry Theory and Applications*, 13(2):141–148, 1999.

[35] D. P. Dobkin and D. Silver. Recipes for geometry and numerical analysis - part I: an empirical study. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 93–105. ACM Press, 1988.

[36] D. S. Dummit and R. M. Foote. *Abstract algebra*. Wiley, 3rd edition, 2004.

[37] L. Dupont, D. Lazard, S. Lazard, and S. Petitjean. Near-optimal parameterization of the intersection of quadrics: I. The generic algorithm. *Journal of Symbolic Computation*, 43(3):168–191, 2008.

[38] M. Dyer, A. Frieze, and R. Kannan. A random polynomial-time algorithm for approximating the volume of convex bodies. *Journal of the ACM*, 38(1):1–17, 1991.

[39] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[40] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete & Computational Geometry*, 1:25–44, 1986.

[41] I. Z. Emiris and J. F. Canny. A general approach to removing degeneracies. *SIAM Journal on Computing*, 24(3):650–664, 1995.

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

[43] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[44] S. Fortune. Robustness issues in geometric algorithms. In *Applied Computational Geometry, Towards Geometric Engineering*, pages 9–14. Springer-Verlag, 1996.

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

[46] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: a multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007.

[47] Z. Furedi and I. Barany. Computing the volume is difficult. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 442–447. ACM Press, 1986.

[48] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18(3):259–278, 1969.

[49] M. T. Goodrich, L. J. Guibas, J. Hershberger, and P. J. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 284–293. ACM Press, 1997.

[50] T. Granlund. GNU multiple precision arithmetic library 5. `http://gmplib.org`, 2012.

[51] D. H. Greene. Integer line segment intersection. Unpublished manuscript.

[52] D. P. Griesheimer. In-line feedback effects. In *Advanced Monte Carlo for Reactor Physics Core Analysis Workshop at PHYSOR*, 2012.

[53] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[54] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1-6):381–413, 1992.

[55] L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. *International Journal of Computational Geometry and Applications*, 8(2):157–176, 1998.

[56] M. Held and S. Huber. Topology-oriented incremental computation of Voronoi diagrams of circular arcs and straight-line segments. *Computer-Aided Design*, 41(5):327–338, 2009.

[57] M. Held and W. Mann. An experimental analysis of floating-point versus exact arithmetic. In *Proceedings of the 23rd Canadian Conference on Computational Geometry*, pages 261–266, 2011.

[58] J. Hershberger. Improved output-sensitive snap rounding. volume 39, pages 298–318. Springer New York, 2009.

[59] N. J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993.

[60] J. Hobby. Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199–214, 1999.

[61] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 277–286. ACM Press/Addison-Wesley Publishing Co., 1999.

[62] C. M. Hoffmann. *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., 1989.

[63] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *Computer*, 22(3):31–40, 1989.

[64] T. Imai. A topology-oriented algorithm for the Voronoi diagram of polygons. In *Proceedings of the Eighth Canadian Conference on Computational Geometry*, pages 107–112, 1996.

[65] H. Inagaki and K. Sugihara. Numerically robust algorithm for constructing constrained Delaunay triangulation. In *Proceedings of the Fourth Canadian Conference on Computational Geometry*, pages 171–176, 1992.

[66] H. Inagaki, K. Sugihara, and N. Sugie. Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams. In *Proceedings of the Fourth Canadian Conference on Computational Geometry*, pages 333–339, 1992.

[67] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056, 2006.

[68] S. Kahan and J. Snoeyink. On the bit complexity of minimum link paths: superquadratic algorithms for problem solvable in linear time. *Computational Geometry Theory and Applications*, 12(1-2):33–44, 1999.

[69] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40, 1965.

[70] E. Kaltofen and G. Villard. Computing the sign or the value of the determinant of an integer matrix, a complexity survey. *Journal of Computational and Applied Mathematics*, 162(1):133–146, 2004.

[71] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 351–359. ACM Press, 1999.

[72] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

[73] M. I. Karavelas and M. Yvinec. Dynamic additively weighted Voronoi diagrams in 2D. In *Tenth European Symposium on Algorithms*, pages 586–598, 2002.

[74] M. Keil. A simple algorithm for determining the envelope of a set of lines. *Information Processing Letters*, 39:121–124, 1991.

[75] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry Theory and Applications*, 40(1):61–78, 2008.

[76] J. Keyser. *Exact Boundary Evaluation for Curved Solids*. PhD thesis, 2000.

[77] J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha. ESOLID—a system for exact boundary evaluation. In *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, pages 23–34. ACM Press, 2002.

[78] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: a library for efficient and exact manipulation of algebraic points and curves. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 360–369. ACM Press, 1999.

[79] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate B-rep generation of low degree sculptured solids using exact arithmetic. In *Proceedings of the Fourth ACM Symposium on Solid Modeling and Applications*, pages 42–55. ACM Press, 1997.

[80] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate B-rep generation of low degree sculptured solids using exact arithmetic: I-representations. *Computer Aided Geometric Design*, 16(9):841–859, 1999.

[81] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate B-rep generation of low degree sculptured solids using exact arithmetic: II-computation. *Computer Aided Geometric Design*, 16(9):861–882, 1999.

[82] R. Klein. *Concrete and abstract Voronoi diagrams*, volume 400 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1989.

[83] D. J. Lawrence. *A catalog of special plane curves*. Dover Publishing, 1972.

[84] S. Lazard, L. M. Peñaranda, and S. Petitjean. Intersecting quadrics: an efficient and exact implementation. *Computational Geometry*, 35(1-2):74–99, 2006.

[85] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. I. known methods and open issues. *Communications of the ACM*, 25(9):635–641, 1982.

[86] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. II. a family of algorithms based on representation conversion and cellular approximation. *Communications of the ACM*, 25(9):642–650, 1982.

[87] J. Levin. A parametric algorithm for drawing pictures of solid objects composed of quadric surfaces. *Communications of the ACM*, 19(10):555–563, 1976.

[88] G. Liotta. Low degree algorithms for computing and checking Gabriel graphs. Technical report, 1996.

[89] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. *SIAM Journal on Computing*, 28(3):864–889, 1999.

[90] A. Mantler and J. Snoeyink. Intersecting red and blue line segments in optimal time and precision. In *Revised Papers from the Japanese Conference on Discrete and Computational Geometry*, pages 244–251. Springer-Verlag, 2001.

[91] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc. `http://www.mathworks.com`.

[92] D. W. Matula and R. R. Sokal. Properties of Gabriel graphs relevant to geographic variation research and the clustering of points in the plane. *Geographical Analysis*, 12(3):205–222, 1980.

[93] C. R. Maurer Jr., R. Qi, and V. Raghavan. A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(2):265–270, 2003.

[94] A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25:11–12, 1987.

[95] J. R. Miller and R. N. Goldman. Geometric algorithms for detecting and calculating all conic sections in the intersection of any two natural quadric surfaces. *Graphical Models and Image Processing*, 57(1):55–66, 1995.

[96] D. L. Millman. Degeneracy proof predicates for the additively weighted Voronoi diagram. Master's thesis, Courant Institute, New York University, 2007.

[97] D. L. Millman, D. P. Griesheimer, B. R. Nease, and J. Snoeyink. Robust volume calculations for constructive solid geometry (CSG) components in Monte Carlo transport calculations. In *PHYSOR 2012: Advances in Reactor Physics*. American Nuclear Society, 2012.

[98] D. L. Millman, S. Love, T. M. Chan, and J. Snoeyink. Computing the nearest neighbor transform exactly with only double precision. In *Ninth International Symposium on Voronoi Diagrams in Science and Engineering*, pages 66–74, 2012.

[99] D. L. Millman and J. Snoeyink. Computing the implicit Voronoi diagram in triple precision. In *Algorithms and Data Structures, 11th International Symposium, WADS*, volume 5664 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2009.

[100] D. L. Millman and J. Snoeyink. Computing planar Voronoi diagrams in double precision: a further example of degree-driven algorithm design. In *Proceedings of the 2010 Annual Symposium on Computational Geometry*, pages 386–392. ACM Press, 2010.

[101] D. L. Millman and V. Verma. A slow algorithm for computing the Gabriel graph with double precision. In *Proceedings of the 23nd Canadian Conference on Computational Geometry*, pages 485–487, 2011.

[102] T. Minakawa and K. Sugihara. Topology oriented vs. exact arithmetic - experience in implementing the three-dimensional convex hull algorithm. In *Proceedings of the Eighth International Symposium on Algorithms and Computation*, pages 273–282, 1997.

[103] T. Minakawa and K. Sugihara. Optimization methods and software. *Topology-Oriented Construction of Three-Dimensional Convex Hulls*, 10:357–371, 1998.

[104] B. Mourrain, J.-P. Técourt, and M. Teillaud. On the computation of an arrangement of quadrics in 3D. *Computation Geometry Theory and Applications*, 30(2):145–164, 2005.

[105] A. Nanevski, G. Blelloch, and R. Harper. Automatic generation of staged geometric predicates. *Higher-Order and Symbolic Computation (formerly LISP and Symbolic Computation)*, 16(4):379–400, 2003.

[106] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with Valgrind. In *IEEE International Symposium on Workload Characterization*, 2006.

[107] Y. Oishi and K. Sugihara. Topology-oriented divide-and-conquer algorithm for Voronoi diagrams. *Graphical Models and Image Processing*, 57(4):303–314, 1995.

[108] D. W. Paglieroni. Distance transforms: properties and machine vision applications. *CVGIP: Graphical Models and Image Processing*, 54(1):56–74, 1992.

[109] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer New York, 1985.

[110] D. M. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, University of California at Berkeley, 1992.

[111] A. G. Requicha. Representations for rigid solids: theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.

[112] G. Rong and T.-S. Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 109–116. ACM Press, 2006.

[113] G. Rong and T.-S. Tan. Variants of jump flooding algorithm for computing discrete Voronoi diagrams. In *Fourth International Symposium on Voronoi Diagrams in Science and Engineering*, pages 176–181, 2007.

[114] E. Sacks, V. Milenkovic, and M.-H. Kyung. Controlled linear perturbation. *Computer-Aided Design*, 43(10):1250–1257, 2011.

[115] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Survey*, 16(2):187–260, 1984.

[116] D. B. Saunders, H. R. Lee, and S. K. Abdali. A parallel implementation of the cylindrical algebraic decomposition algorithm. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, 1989.

[117] J. Schneider, M. Kraus, and R. Westermann. GPU-based real-time discrete Euclidean distance transforms with precise error bounds. In *International Conference on Computer Vision Theory and Applications*, pages 435–442, 2009.

[118] E. Schömer and N. Wolpert. An exact and efficient approach for computing a cell in an arrangement of quadrics. *Computation Geometry Theory and Applications*, 33(1-2):65–97, 2006.

[119] R. Seidel. The nature and meaning of perturbations in geometric computing. *Discrete & Computational Geometry*, 19(1):1–17, 1998.

[120] C. Severance. IEEE 754: an interview with William Kahan. *Computer*, 31(3):114–115, 1998.

[121] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 151–162. IEEE Computer Society, 1975.

[122] C.-K. Shene and J. K. Johnstone. On the lower degree intersections of two natural quadrics. *ACM Transactions on Graphics*, 13(4):400–424, 1994.

[123] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 141–150. ACM Press, 1996.

[124] J. Stolfi. *Primitives for computational geometry*. PhD thesis, Stanford University, 1989.

[125] J. Stolfi. *Oriented projective geometry: a framework for geometric computations*. Academic Press, 1991.

[126] K. Sugihara. Robust gift wrapping for the three-dimensional convex hull. *Journal of Computer and System Sciences*, 49(2):391–407, 1994.

[127] K. Sugihara. Topology-oriented approach to robust geometric computation. *Lecture Notes in Computer Science*, pages 357–366, 1999.

[128] K. Sugihara. Robust geometric computation based on topological consistency. *Lecture Notes in Computer Science*, pages 12–26, 2001.

[129] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *Journal of Information Processing*, 12(4):380–393, 1990.

[130] K. Sugihara and M. Iri. Construction of the Voronoi diagram for 'one million' generators in single-precision arithmetic. *Proceedings of the IEEE*, 80(9):1471–1484, 1992.

[131] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation—an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000.

[132] T. M. Sutton, T. J. Donovan, T. H. Trumbull, P. S. Dobreff, E. Caro, D. P. Griesheimer, L. J. Tyburski, D. C. Carpenter, and H. Joo. The MC21 Monte Carlo transport code. In *Joint International Topical Meeting M&C + SNA*. American Nuclear Society, 2007.

[133] E. Vinberg. *A course in algebra*. American Mathematical Society, 2003.

[134] W. Wang, R. Goldman, and C. Tu. Enhancing Levin's method for computing quadric-surface intersections. *Computer Aided Geometric Design*, 20(7):401–422, 2003.

[135] E. W. Weisstein. Cramer's rule – from Wolfram MathWorld. `http://mathworld.wolfram.com/CramersRule.html`.

[136] E. W. Weisstein. Vandermonde determinant – from Wolfram MathWorld. `http://mathworld.wolfram.com/VandermondeDeterminant.html`.

[137] E. B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

[138] Wolfram Research, Inc. Mathematica, 2008.

[139] X-5 Monte Carlo Team. MCNP – a general Monte Carlo n-particle transport code version 5. Technical Report LA-UR-03-1987, Los Alamos National Laboratory, 2003.

[140] C. K. Yap. Towards exact geometric computation. *Computational Geometric Theory and Applications*, 7(1-2):3–23, 1997.

[141] C. K. Yap. In praise of numerical computation. *Efficient Algorithms*, pages 380–407, 2009.