THESIS PROPOSAL: DEGREE-DRIVEN GEOMETRIC ALGORITHM DESIGN

DAVID L. MILLMAN

1. INTRODUCTION

The correct implementation of geometric algorithms is surprisingly difficult. In part, this stems from the fact that geometric algorithms are often designed for Real-RAM, a computational model that provides arbitrary precision arithmetic operations at unit cost. Commodity hardware provides only finite precision and may result in arithmetic errors. While the errors may seem small, if ignored, they may cause branching errors, which may cause an implementation to reach an undefined state, produce erroneous output, or crash. Liotta, Preparata and Tamassia [29] proposed that in addition to considering the resources of time and space, an algorithm designer could also consider the precision necessary to guarantee a correct implementation, they called this design technique *degree-driven algorithm design*. By considering the time, space, and precision for a problem we arrive at new solutions, gain further insight, and find simpler representations.

I propose to investigate and implement degree-driven algorithms for Voronoi diagrams and organize the implementations into a kernel for a degree-driven algorithm library. In Section 2 I define terminology, we recall the cause of numerical error and various approaches to its management in geometric algorithms, I describe the model of computation used in this thesis, and we recall the main results using this model of computation. In Section 3 I outline the thesis, each subsection corresponds to a chapter, which focuses on a problem. In each subsection, I present the problem; provide a brief summary of the related work; and describe the chapter goal, what is complete, and what is left. In Section 4 I provide the time line for the defense in Spring 2012.

2. BACKGROUND

Precision analysis provides us with tools for determining the maximal precision required by an algorithm, and helps us classify the conditions under which an algorithm's implementation will be "correct", which we see in Section 2.1 has different meanings in different contexts and communities. In this section, I define some terminology, we recall the cause of numerical error and various approaches to its management in geometric algorithms, I describe the model of computation used in this thesis and we recall the main results using this model of computation.

Date: April 20, 2011.

Let us illustrate some important definitions and concepts on the example:

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

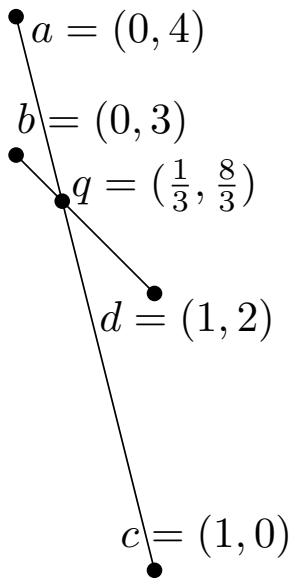


FIGURE 1. Two segments with an intersection coordinate that does not lend itself to a floating point representation.

The input to a geometric algorithm is a set of single-precision numeric coordinates and combinatorial relationships between the coordinates. In IsSegInter, 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 paring of points into segments.

A geometric *construction* produces a new geometric object from the coordinate values of the input (Sometimes, the output precision of a construction is higher than the input, in Section 2.2 I describe how one can analyze precision.) Consider, for example, producing the intersection point q of lines $\overleftrightarrow{ac}$ and $\overleftrightarrow{bd}$.

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

$$q_x = \frac{\begin{vmatrix} a_x c_y - c_x a_y & a_x - c_x \\ b_x d_y - d_x b_y & b_x - d_x \end{vmatrix}}{\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - d_x & b_y - d_y \end{vmatrix}}, \qquad q_y = \frac{\begin{vmatrix} a_x c_y - c_x a_y & a_y - c_y \\ b_x d_y - d_x b_y & b_y - d_y \end{vmatrix}}{\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - d_x & b_y - d_y \end{vmatrix}}. \tag{1}$$

In Algorithm 1, `IsSegInterByConstruction`, we solve IsSegInter by constructing the intersection point q of the lines through our segments and testing properties of q.

Recall that a computer uses finite precision *floating point* numbers to approximate a subset of the reals. Most often, floating point numbers are represented in base 2 with a biased exponent E. They are stored 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 value, the highest order bit of the mantissa is assumed to be 1 and so it does not need to be explicitly represented. The value of a normalized floating point number is $(-1)^s \times 1.m \times 2^{e-E}$.

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

Algorithm 1 `IsSegInterByConstruction`(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 $q =$ `Intersect`(a, c, b, d) /* See Equation 1 */
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** $t_1, t_2 \in [0, 1]$ **then**
8: **return** INTERSECT
9: **else**
10: **return** NOINTERSECT
11: **end if**

This now familiar representation was not always so common. In an interview [41], 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 there difference was zero. Kahan, along with Coonen and Stone proposed the IEEE Standard 754-1985, which is the basis for the "float" and "double" types in may high level programming languages such as C, C++, C#, and Java. In these languages, "float" is implemented with the IEEE-754 single precision floating point format called *binary32* 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 $(-1)^s \times (1 + \Sigma_{i=1}^{23} b_i 2^{-i}) \times 2^{e-127}$.

Thus, the computer rounds even a simple rational such as $8/3$ to a number near $2.\overline{6}$. For a real value x, we notate its floating point representation as $\texttt{fl}(x)$. For a geometric object o, defined by coordinate values, $\texttt{fl}(o)$ is the geometric object induced from applying `fl` to the coordinate values of o. For example, the point $\texttt{fl}(q)$ has coordinate values $(\texttt{fl}(1/3), \texttt{fl}(8/3))$. For some calculations the effect of rounding to a floating point value is negligible, however, without care, errors in geometric calculations may occur. For example, in Figure 2, $\texttt{fl}(q)$ does not lie on either segment.

Let us now consider a different solution to IsSegInter that avoids a possibly erroneous construction of an intersection point. The algorithm is built from primitives called *predicates*, which are tests of the sign of multivariate polynomials with variables from the input coordinates. *Operations* supply a geometric meaning to the positive, negative and zero values returned by a predicate. Consider an example seen throughout the introduction:

`Orientation`(p, q, r)**:** We define a predicate on the single precision-coordinate values of of two-dimensional points p, q and r as,

$$\text{sign}(q_x r_y - q_x p_y - p_x r_y - r_x q_y + r_x p_y + p_x q_y) = \text{sign}(\begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}). \quad (2)$$

The positive, negative or zero result of the predicate is interpreted by the `Orientation`(p, q, r) operation as whether the path from p to q to r forms a left turn, right turn or follows a line, respectively. For the segments shown in Figure 2, `Orientation`(a, c, d) is a left hand turn, `Orientation`(a, c, b) is a right hand turn and `Orientation`(a, c, q), with q not rounded, follows a line.

In Algorithm 2, `IsSegInterByOrientation`, 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 `IsSegInterByOrientation`(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** `Orientation`$(a, c, b) \neq$ `Orientation`(a, c, d) and `Orientation`$(b, d, a) \neq$ `Orientation`(b, d, c) **then**
2: **return** INTERSECT
3: **else**
4: **return** NOINTERSECT
5: **end if**

The `IsSegInterByOrientation` algorithm avoids the construction of an intersection coordinate and arrives at a simpler algorithm. In Section 2.2, we see how to analyze the arithmetic precision of both segment intersection algorithms, and find that `IsSegInterByOrientation` uses less precision than `IsSegInterByConstruction`.

2.1. **Definitions to Correctness.** Theoreticians usually prove an algorithm's correctness in the Real-RAM model of computation [35]. In this model, arithmetic operations are exact and take unit time. The proof provides the algorithm's preconditions and an implementer uses the preconditions to check if an input is appropriate. However, if too few arithmetic bits are available to correctly evaluate the algorithm's predicates, operations and constructions, an implementation may fail for some inputs that satisfy Real-RAM preconditions [20, 23].

A branch of computational geometry investigates approaches for avoiding the numerical errors that can be introduced in implementations. One could naively rely on machine precision (or some ϵ-tolerances). Kettner *et al.* [27] investigate the errors introduced with floating point arithmetic using simple operations, 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 complicated (and non-intuitive) structure. Kettner also

points out that ϵ-tolerances will not fix the problems raised by their investigation, it only rounds non-zero values to zero and enlarges the complicated error structure.

Sophisticated approaches add steps to ensure correct results. Yap [49] 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 five main approaches to numerical computational geometry.

Approach 1: Exact Geometric Computation (EGC). We can think of geometric computing as part combinatorial, for example traversing an embedded graph, and part numeric, for example determining if two segments intersect. Yap [48] observed that the interplay between the numerical and the combinatorial causes geometric algorithms to be difficult to implement. Thus, the 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.

Some predicates can be reduced to computing the sign of a determinant. Kaltofen and Villard [25] survey methods for computing the sign of the determinant of an integer matrix, and mention that computing the sign is at most as hard as computing the value. Clarkson [16] 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, and Avnaim *et al.* [1] describes how to compute the sign of 2×2 and 3×3 determinants with b-bit integer entries with b and $b + 1$ bits respectively.

Many geometric operations are determined by evaluating and comparing the signs of polynomials. The evaluation may demand more bits than are provided by floating point arithmetic. Thus, libraries such as CORE [26] and LEDA [9], which can represent and compare algebraic numbers, are a key ingredient for implementing EGC.

Approach 2: Arithmetic Filters. Usually, comparing low precision approximations to algebraic numbers is enough to get a correct result. Arithmetic filter approaches [8, 18, 19, 21] avoid comparing algebraic number when possible. They use floating point arithmetic to compute an interval that bounds the result of an expression, and compare the intervals. If the intervals overlap, expressions are reevaluated at a higher precision to tighten the interval to certify the answer.

For example, if we wanted to test the sign of the evaluation of a polynomial, it suffices to first use floating point arithmetic to compute an interval that bounds the evaluation, and then check if the lower bound is positive (or the upper bound is negative). However, if zero is in the interval, we reevaluate the polynomial with exact arithmetic to find the correct sign.

Approach 3: Adaptive Predicates. As mentioned for Arithmetic Filters, it is not always necessary to evaluate a predicate exactly to produce the correct result. Adaptive predicates, described in Priest's Ph.D. thesis [36] and implemented in Shewchuk's InCircle and Orientation Predicates [42], as well as Bernstein and Fussell's modeling system [4], evaluate only up to enough precision to guarantee a correct result.

Approach 4: Topological Consistency. For some applications an exact geometric construction is unnecessary and only some properties of the construction are required. Topologically consistent algorithms [43, 44, 45, 46, 47] strive to maintain the required properties; when multiple branch options are possible, the most likely branch is chosen. While the resulting construction may not be exact, it is in some sense consistent. For example, an exact Voronoi diagram is an embedding of a connect planar graph. A topologically consistent Voronoi diagram construction may only ensure that the output graph is connected (but maybe non-planar). If we were to pass the output graph to a search algorithm that assumes connectivity, the search will at least terminate.

Sometimes maintaining a limited set of properties is enough to achieve a goal. Sugihara and Iri [46] used a topologically consistent algorithm to build the first Voronoi diagram of over a million sites with single precision arithmetic.

Approach 5: Degree-Driven Algorithm Design. Most theoretical computer scientists analyze and optimize the running time and space used by an algorithm. *Degree-driven algorithm design*, proposed by Liotta, Preparata and Tamassia [29], adds the precision used by an algorithm into the mix. The design seeks to optimize (and balance) running time, space and precision simultaneously. This thesis explores degree-driven algorithm design for point location and triangulation. Next, I present the model of computation proposed by Liotta, Preparata and Tamassia and survey the results in degree-driven algorithm design.

2.2. **Model of Computation.** Liotta, Preparata, and Tamassia [29] proposed analyzing the precision of a geometric algorithm in terms of the *arithmetic degree* of the polynomials used in its predicates. Suppose that the input can be scaled to b-bit integers, then, a monomial of degree k can be evaluated in bk bits, and a polynomial of a monomials can be summed in $bk + \log_2 a$ bits. Therefore, 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 in this analysis. In the second chapter of the thesis, I build a simple summation primitive that allows us to recover the bits lost in the summation (should that be an issue). This summation primitive is simpler than the Kahan sum [24], which we suggested earlier [34], and is based on the observation that evaluating the sign of a polynomial is at most as hard as evaluating its value.

Following Liotta, Preparata, and Tamassia's definitions, define: the *degree of an operation* as the maximal degree of the predicates used in the operation; the *degree of a construction* as the degree of the polynomials used to represent the output of the construction; the *degree of a data structure* as the highest degree of the operations and constructions represented by the data structure; the *degree of an algorithm* as the maximum degree of its predicates, operations, and constructions; and the *degree of a problem* as the minimal degree of all algorithms that solve the problem. We pause for a moment to see an example of precision analysis carried out on the two algorithms, described earlier, for solving IsSegInter. (The analysis below is brief, the second chapter of this thesis will present a detailed analysis).

First, consider Algorithm 1, `IsSegInterByConstruction`. Testing if $\overleftrightarrow{ac}$ is parallel to $\overleftrightarrow{bd}$, in line 1, is degree 2. The `Intersect` construction, in line 4, computes the Cartesian coordinates of q, which are rational polynomials of degree 3 over degree 2. Solving for t_1 and t_2, in lines 5 and 6 respectively, computes rational polynomials of degree 3 over degree 3. Testing if t_1 and t_2 are in $[0, 1]$, in line 7, is degree 3. Thus, `IsSegInterByConstruction` is degree 3.

Next, consider Algorithm 2, `IsSegInterByOrientation`. The predicate in Equation 2, is degree 2. The four `Orientation` operations, in line 1, which evaluate only Equation 2, are degree 2. Thus, `IsSegInterByOrientation` is degree 2.

This discussion shows the degree of the IsSegInter problem is at most degree 2. One could ask, can we do better and solve IsSegInter degree 1? To provide an answer, the definition of the degree of a problem needs to be more clearly spelled out. I defer lower bounds on degree until the second chapter of the thesis.

We pause for a moment, again, so that I may explain why we carry out our analysis under the assumption that our input is scaled to an integer grid. As Chan and Pătraşcu [15] point out, the floating point plane is the union of integer grids with different scalings around the origin. In this thesis, we are interested in translation independent problems. Thus, the ability to have more precision for input near the origin is not particularly meaningful.

2.3. **Previous Results in Degree-Driven Algorithm Design.** The most complete study of degree-driven analysis was carried out for segment intersection problems [5, 6, 11, 30]. Boissonnat and Preparata [5] describe three problems for a set of n line segments, defined by their endpoints:

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 where 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; as such, a trivial algorithm checks all pairs using the degree two 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 Bentley-Ottmann sweep line algorithm [3] 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 [11], and Boissonnat and Snoeyink [6], 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 slap 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 [2] 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 (this seems to be the first example of a time-precision trade off). Finally, they show that even with restricted predicates, the red and 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 [30], 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 arrangement (but not the coordinates of the intersections) can be computed using a sweep line algorithm that runs in optimal $O(n \log n + k)$ using $O(n)$ space and degree 2.

Researchers also consider point location queries. By assumption, the coordinates of the input sites and query points are b-bit integers. Let $U = 2^b$, the set of representable points from which sites and queries are drawn is a $U \times U$ grid, sometimes referred to as a universe of size U [15].

Liotta, Preparata and Tamassia [29] 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 have to construct the trapezoidation. Millman and Snoeyink [33] consider how to construct Liotta's structure with a degree 3 randomized algorithm in a size U universe in time $O(n \log Un)$. Using a different approach, Millman and Snoeyink [34] describe a degree 2 construction of a degree 1 data structure that supports degree 2 logarithmic time point location queries. We further explore this construction in Section 3.2.

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. Chan and co-authors [14] show how to compute all nearest neighbor queries using degree 2 predicates with an expected time $O(U^2)$ algorithm, which we encounter in Section 3.3

The Gabriel graph of a point set S is a embedded graph whose vertices are the points of S and whose 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. Matula and Sokal [31] show 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 [28] 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.

3. OUTLINE

In this section, I outline the thesis, each subsection corresponds to a chapter, which focuses on a problem. In each subsection, I present the problem; provide a brief summary of the related work; and describe the chapter goal, what is complete, and what is left.

3.1. **Chapter 2: Primitives.**

In this chapter, I present predicates, operations, constructions, and results of the ongoing degree lower bounds and irreducibility discussions. This chapter is the furthest along, as such, I only present the goal.

Goal: I propose to provide descriptions, precision analysis and book quality code for all predicates, operations and constructions, discussed in the thesis and useful for degree-driven algorithm design. This chapter will conclude with results on lower bounds and irreducibility.

3.2. **Chapter 3: Post-office Queries for Some Points in the Plane.**

A common geometric problem is to preprocess a set of n point sites in the plane to efficiently answer queries of the form: what is the closest site to a point p. A popular solution, shown in Figure 3.2, is to compute a trapezoidation of the Voronoi diagram in $O(n \log n)$ time, the interior of each trapezoid is closest to only one site, therefore, a $O(\log n)$ time point location on the trapezoidation provides the desired result.

However, this solution uses at least degree 4 to compute the Voronoi diagram, degree 5 to to build the trapezoidation and degree 6 for queries (degree 3 queries are possible if we are a little clever). For cases where the query points are of the same precision as the input sites, Liotta, Preparata, and Tamassia [29] describe a degree 6 algorithm that produces a degree 1 data structure capable of answering queries with degree 2 predicates. We can think of the sites and query points of the same precision as lying on a $U \times U$ grid.

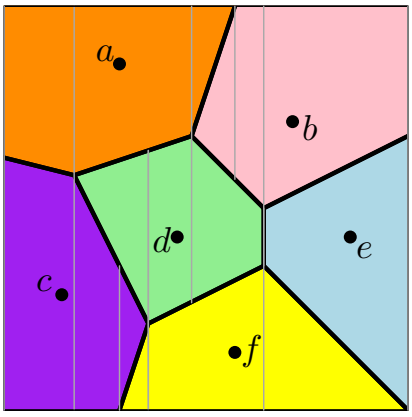


FIGURE 2. A trapezoidation of the Voronoi diagram of 6 sites. The interior each trapezoid contains points closest to only one site.

Millman and Snoeyink [33] showed how to build Liotta's structure with a degree 3 randomized algorithm in time $O(n \log Un)$. Recently, Millman and Snoeyink [34] provided a degree 2 constriction for a degree 1 data structure that answers post office queries in degree 2 and $O(\log n)$ time. The analysis for the running time of our construction is incomplete. We used the generalized random incremental analysis framework [17, Chapter 9], but one of the conditions for the framework is more difficult to prove than originally expected. (The degree of a configuration must be constant, however it is not clear this is true in our case).

Goal: I propose to complete the analysis of [34], describe the algorithm, provide an implementation, book quality code and experimental results. Should I be unable to complete the analysis, I will explore whether a divide-and-conquer algorithm can give us a sub-quadratic time degree 2 construction. Should that be unsuccessful, I would like to implement our RIC degree 2 algorithm and observe the experimental running time, implement the degree 3 solution, and provide book quality code and experimental results for both.

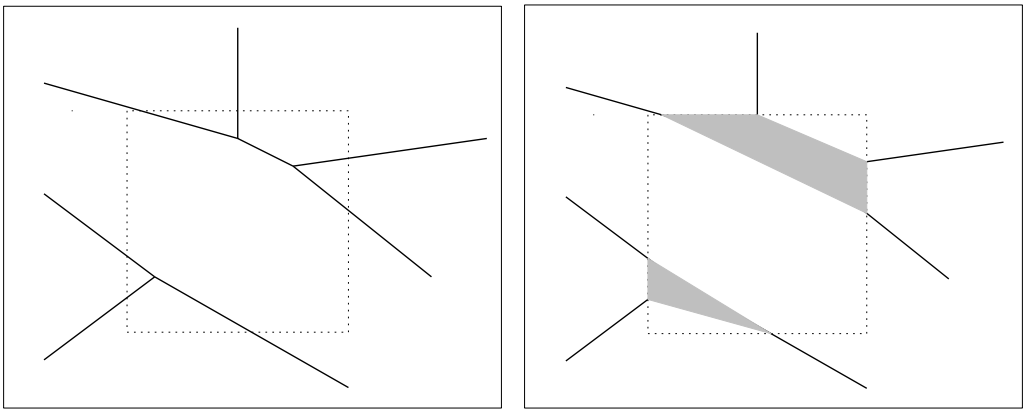
3.3. **Chapter 4: Nearest Neighbor Transform.** The nearest neighbor transform of a $U \times U$ black and white image assigns to every pixel the coordinates of the closest black pixel under the Euclidean metric. It is equivalent to computing the Voronoi diagram on a grid; optimal algorithms to compute the transform perform this computation starting from the list of coordinates of the black pixels [7, 13, 32], and take $O(U^2)$ time.

Many researchers explored GPU and parallel and serial CPU algorithms for computing the nearest neighbor transform; however, none of the previous work can guarantee a correct output without exact arithmetic. Breu *et al.* [7] proposed the first linear time algorithm, but their algorithm requires five-fold precision. Both Chan and Maurer *et al.*'s dimensional reduction algorithms [12, 13, 32] require three-fold precision. Hoff and others [22] 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; the precision is determined by the resolution of the Z-buffer. Recently, approximate GPU algorithms were proposed [37, 38, 40], but they cannot guarantee an exact result. Cao *et al.* [10] showed how to adapt 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 did not address the precision of the predicates. In a private correspondence, we determined that their published implementation uses degree 5 predicates, and can be reduced to degree 3.

Chan and co-authors [14] showed how to adapt Chan's dimensionality reduction algorithm [12, 13] to two-fold precision, focusing on the algebraic degree of the geometric predicates. Our degree 2 algorithm takes expected $O(U^2)$ time and $O(U^2)$ space. We also propose a representation that can be stored and implemented using $O(n \log U)$ space.

Goal: Our solution, written up in [14], only contains a sketch of the construction, without analysis. In this chapter I propose to provide the details of the construction, analysis, book quality code, and experimental results for our implementation.

3.4. **Chapter 5: Triangulations.** A triangulation of a finite set of points S in the plane, is a planar subdivision whose vertices are the points of S and no edges can be added without causing the subdivision to become non-planar. Since every polygon in the plane can be triangulated, the bounded faces of a triangulation are triangles. The boundary of a triangulation is the convex hull of S, which can be computed with degree 2.



(A) Voronoi diagram (B) rp-Voronoi diagram

FIGURE 3. The Voronoi vertices in a grid cell on the left contract to two rp-vertices, depicted as gray convex polygons on the right. Voronoi edges that intersect a grid line are present in the rp-Voronoi diagram.

A triangle τ of a triangulation of S has an empty circle if the interior of the circumcircle of τ contains no points of S. Every triangle of the Delaunay triangulation has this empty circle property. It is well know that the Delaunay triangulation is the dual of the Voronoi diagram. Less well known is that the `InCircle` predicate, and computing the Delaunay triangulation, requires degree 4.

In this chapter, I will explore computing triangulations with less than degree 4 predicates. Below are a few directions that I would like to explore, then I state my goals for this chapter.

Millman and Snoeyink [33] proposed the degree 3 *rp-Voronoi diagram* for a point location. The diagram captures any Voronoi edge that intersects a grid line, and thus its dual captures corresponding Delaunay edges. However, as seen in Figure 3, our construction collapses trees of Voronoi vertices that occur in the same grid cell to a single rp-vertex. The result creates (possible non-convex) polygonal holes in the induced subdivision. If all of holes can be triangulated in $O(n \log n)$ time this would produce a triangulation with degree 3.

The Gabriel graph is a subgraph of the Delaunay triangulation that contains only the edges e such that a circle, centered at the midpoint of e with diameter length of e contains no other points of S. Matula and Sokal [31] showed 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 [28] showed that Matula and Sokal's algorithm is degree 6, but provided a degree 2 algorithm that accomplishes the same result. However, Liotta's algorithm must first use degree 4 to compute the Delaunay triangulation. We can test if an edge of the Gabriel graph has the empty circle property with degree 2; thus, by brute force we can compute the Gabriel graph with degree 2 with $O(n^3)$ time. But, can we compute the Gabriel graph directly in sub-cubic time with degree 2?

In chapter 4 I will describe how to compute, using degree 2, the nearest neighbor transform of a point set on a grid in expected $O(U^2)$ time. Rong *et al.* [39] investigates how to adapt the nearest neighbor transform to a Delaunay triangulation, but they use degree 4 to test the empty circle property. Can we do a similar adaptation without using `InCircle` and produce a valid triangulation? Can we get rid of the $O(U^2)$ preprocessing by applying some of the techniques described by Millman and Snoeyink [34]?

Goal: I propose to describe a triangulation that can be computed efficiently with two or three-fold precision, provide book quality code and experiments for an implementation and some of the properties that the proposed triangulation possesses. Some properties may include angle bounds of the triangulation, or how far it is in the flip graph from the Delaunay? Should these properties be too difficult to discover, experimental results may be supplied.

4. Time line for Spring 2012 graduation

- **June 1, 2011** Draft chapters of:
  - Chapter 2: Geometric Primitives
  - Chapter 4: Nearest Neighbor Transform
- **Sept 1, 2011** Implementation of:
  - degree 2 and/or degree 3 Voronoi construction.
  - a degree 2 or degree 3 triangulation.

In Sept 2010, I contacted Professor Jeffay about teaching Fall 2011 or Spring 2012. He said Fall 2011 will "likely work" and Spring 2012 is "absolutely doable". As many students report that little research is accomplished in their teaching semester, below, I split my time line dependent on the semester that I teach.

Teach in the Fall:

- **Jan 1, 2012** Drafts of:
  - Chapter 3: Post-office Queries for Some Points in the Plane.
  - (working draft of) Chapter 5: Triangulations.
- **March 1, 2012** Draft of:
  - Chapter 5: Triangulation.
- **~April 1, 2012** Thesis Defense.

Teach in the Spring:

- **Oct 15, 2012** Draft of:
  - Chapter 3: Post-office Queries for Some Points in the Plane.
- **Dec 15, 2012** Draft of:
  - Chapter 5: Triangulation.

- **~March 1, 2012** Thesis Defense

REFERENCES

[1] F. Avnaim, J. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17:111–132, 1997. 10.1007/BF02522822.

[2] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 211–219, New York, NY, USA, 1995. ACM.

[3] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *Computers, IEEE Transactions on*, C-28(9):643 –647, 1979.

[4] G. Bernstein and D. Fussell. Fast, exact, linear booleans. *Computer Graphics Forum*, 28(5):1269–1278, Jul 2009.

[5] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comput.*, 29(5):1401–1421, 2000.

[6] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. *Computational Geometry*, 16(1):35 – 52, 2000.

[7] 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.

[8] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Appl. Math.*, 109:25–47, April 2001.

[9] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in leda. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 418–419, New York, NY, USA, 1995. ACM.

[10] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *I3D '10: Proceedings of the 2010 symposium on Interactive 3D graphics and games*, New York, NY, USA, 2010. ACM.

[11] T. M. Chan. Reporting curve segment intersections using restricted predicates. *Computational Geometry*, 16(4):245 – 256, 2000.

[12] T. M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 152–159, New York, NY, USA, 2004. ACM.

[13] T. M. Chan. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Comput. Geom. Theory Appl.*, 35(1):20–35, 2006.

[14] 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. electronic proceedings.

[15] 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.

[16] K. Clarkson. Safe and effective determinant evaluation. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:387–395, 1992.

[17] M. de Berg, O. Cheong, M. van Krefeld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3 edition, March 2008.

[18] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete and Computational Geometry*, 20(4):523–547, 1998-12-21.

[19] O. Devillers and F. P. Preparata. Further results on arithmetic filters for geometric predicates. *Comput. Geom. Theory Appl.*, 13:141–148, June 1999.

[20] S. Fortune. Robustness issues in geometric algorithms. In *FCRC '96/WACG '96: Selected papers from the Workshop on Applied Computational Geormetry, Towards Geometric Engineering*, pages 9–14, London, UK, 1996. Springer-Verlag.

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

[22] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[23] C. Hoffmann. The problems of accuracy and robustness in geometric computation. *Computer*, 22(3):31–39, 41, Mar 1989.

[24] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[25] 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. Proceedings of the International Conference on Linear Algebra and Arithmetic 2001.

[26] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 351–359, New York, NY, USA, 1999. ACM.

[27] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. *Comput. Geom. Theory Appl.*, 40(1):61–78, 2008.

[28] G. Liotta. Low degree algorithms for computing and checking gabriel graphs. Technical report, Providence, RI, USA, 1996.

[29] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM J. Comput.*, 28(3):864–889, 1999.

[30] A. Mantler and J. Snoeyink. Intersecting red and blue line segments in optimal time and precision. In *JCDCG '00: Revised Papers from the Japanese Conference on Discrete and Computational Geometry*, pages 244–251, 2001.

[31] 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.

[32] 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 Trans. Pattern Anal. Mach. Intell.*, 25(2):265–270, 2003.

[33] D. L. Millman and J. Snoeyink. Computing the implicit Voronoi diagram in triple precision. In F. K. H. A. Dehne, M. L. Gavrilova, J.-R. Sack, and C. D. Tóth, editors, *Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, August 21-23, 2009. Proceedings*, volume 5664 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2009.

[34] D. L. Millman and J. Snoeyink. Computing planar Voronoi diagrams in double precision: a further example of degree-driven algorithm design. In *SCG '10: Proceedings of the 2010 annual symposium on Computational geometry*, pages 386–392, New York, NY, USA, 2010. ACM.

[35] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[36] D. M. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, Berkeley, CA, USA, 1992.

[37] G. Rong and T.-S. Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, New York, NY, USA, 2006. ACM.

[38] G. Rong and T.-S. Tan. Variants of jump flooding algorithm for computing discrete Voronoi diagrams. In *ISVD '07. 4th International Symposium on Voronoi Diagrams in Science and Engineering, 2007.*, pages 176 –181, july 2007.

[39] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus. Computing two-dimensional delaunay triangulation using graphics hardware. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 89–97, New York, NY, USA, 2008. ACM.

[40] 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 (VISAPP)*, pages 435–442, 2009.

[41] C. Severance. Ieee 754: An interview with william kahan. *Computer*, 31(3):114–115, Mar 1998.

[42] J. R. Shewchuk. Robust Adaptive Floating-Point Geometric Predicates. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 141–150. Association for Computing Machinery, May 1996.

[43] K. Sugihara. Topology-oriented approach to robust geometric computation. *Lecture Notes in Computer Science*, 1741/1999:357–366, January 1999.

[44] K. Sugihara. Robust geometric computation based on topological consistency. *Lecture Notes in Computer Science*, 2073/2001:12–26, 2001.

[45] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *J. Inf. Process.*, 12:380–393, April 1990.

[46] 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, Sept. 1992.

[47] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation—an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000-05-21.

[48] C.-K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1-2):3–23, 1997.

[49] C. K. Yap. In praise of numerical computation. pages 380–407, 2009.