Efficient algorithms for line and curve segment intersection using restricted predicates

Jean-Daniel Boissonnat* Jack Snoeyink†

October 19, 1999

Abstract

We consider whether restricted sets of geometric predicates support efficient algorithms to solve line and curve segment intersection problems in the plane. Our restrictions are based on the notion of algebraic degree, proposed by Preparata and others as a way to guide the search for efficient algorithms that can be implemented in more realistic computational models than the Real RAM.

Suppose that n (pseudo-)segments have k intersections at which they cross. We show that intersection algorithms for monotone curves that use only comparisons and above/below tests for endpoints, and intersection tests, must take at least $\Omega(n\sqrt{k})$ time. There are optimal $O(n \log n + k)$ algorithms that use a higher-degree test comparing x coordinates of an endpoint and intersection point; for line segments we show that this test can be simulated using `CCW()` tests with a logarithmic loss of efficiency. We also give an optimal $O(n \log n + k)$ algorithms for red/blue line and curve segment intersection, in which the segments are colored red and blue so that there are no red/red or blue/blue crossings.

1 Introduction

All too often, a proof that a geometric algorithm is correct for the Real RAM computational model [22] does not imply that a correct implementation will run correctly on the limited precision arithmetic of a real computer. This fact has spurred three branches of research: First, researchers have studied how to correctly and efficiently evaluate predicates used by geometric algorithms. Much recent work has been devoted to combining floating point filters and exact evaluation of predicates; exact computation is performed when the floating point filter fails to provide a certified answer, which is usually rare. New methods have been designed for the exact evaluation of signs of determinants and arithmetic expressions [13, 2, 7], and various exact, adaptive arithmetics [9,

*INRIA, BP 93, 06902 Sophia Antipolis, France. `Jean-Daniel.Boissonnat@sophia.inria.fr` Research partially supported by ESPRIT IV LTR Project No 28155 (Galia).

†University of British Columbia, Department of Computer Science, and University of North Carolina, Department of Computer Science, CB 3175 Sitterson Hall, Chapel Hill, NC, USA, 27599–3175. `snoeyink@cs.unc.edu` Research partially supported by grants from NSERC, the Killam Foundation, and CIES.

23, 24, 26], and various floating point filters, both static and dynamic, have been experimentally tested [8, 15, 6]. Second, researchers have investigated algorithms that give approximate results with provable properties and guarantees on efficiency [16, 20, 25]. Third, researchers have considered the computational requirements of the problems themselves and developed algorithms that use "simpler" predicates. Our notion of "simpler" is found in Section 2.

It is this third branch that we follow in this paper. Following Boissonnat and Preparata [5], we study the classic problem of *segment intersection*: Given a set of n segments in the plane, report all pairs of intersecting segments. We consider this problem for sets of line segments and sets of segments of x-monotone curves in which any pair intersects in at most one point, at which they cross. That is, we restrict our attention to what may be called *pseudo-segments.* Some extensions to segments that intersect in two or more points are possible.

Forrest [14] has said, "Mathematically, the problem of reporting intersecting segments is trivial. Computationally, the problem is far from easy, and may be impossible to solve reliably and consistently." This comment may seem surprising, since there is a simple $\Theta(n^2)$ time algorithm that is optimal in the worst case: check all pairs for intersection. If only k pairs intersect, however, we might prefer an algorithm whose running time is sensitive to the *output size* k as well as its input size n. A running time of $O(n \log n + k)$ would be optimal, since $\Omega(k)$ time is required to write the output, and any algorithm that reports if any intersection occurs can solve *element uniqueness*, which takes $\Omega(n \log n)$ time [18].

An important question is whether a set of predicates allow an efficient algorithm. It is clear that the choice of predicates can affect the possibility of performing a computation: For example, if predicates are linear polynomials then it is impossible to determine whether two segments intersect—there is no algorithm for segment intersection at all. We show that efficiency is also affected. In Section 3 we establish an $\Omega(n\sqrt{k})$ lower bound for curve intersection algorithms that use predicates only to test order and orientation of curve endpoints, whether endpoints are above or below other curves, and whether two curves intersect (we do not allow predicates that order intersection points). Balaban's algorithm [3] can beat the lower bound and solve this problem optimally for curves using an additional predicate that compares orders of intersection points and endpoints. For line segments, we can adapt Balaban's algorithm to achieve $O(n \log^2 n + k \log n)$ time using only `CCW()` tests.

In Section 4 we consider the special case of the red/blue curve intersection problem, which is to find the intersecting pairs in a set of curves that have been colored red and blue so that there are no red/red or blue/blue crossings. In this case, we obtain an optimal $O(n \log n + k)$ algorithm for line and curve segments by adapting the trapezoid sweep algorithm of Chan [10].

2 Preliminaries and history of segment intersection

In this section, we discuss the predicates that our algorithms and lower bounds will be using.

2.1 Algebraic degree

We limit the computational predicates by *algebraic degree.* As formulated by Preparata and others [19, 5], an *elementary predicate* used by an algorithm is a test of the sign of a homogeneous multivariate polynomial whose arguments are a subset of the input variables; its *degree* is the maximum degree of its polynomial factors that are irreducible over the rationals and have non-constant sign. For example, the commonly-used *orientation test* for three points in the plane is an elementary predicate of degree 2:

$$\texttt{CCW}(p,q,r) = \operatorname{sign}\det\begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix} = \operatorname{sign}\Big((q_x - p_x)(r_y - p_y) - (r_x - p_x)(q_y - p_y)\Big).$$

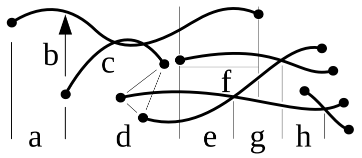

Figure 1: Predicates a–h

A *predicate*, which is a boolean combination of a constant number of elementary predicates, has *degree* equal to the maximum degree of its elementary predicates. The *degree of an algorithm* is the maximum degree of its predicates, and the *degree of a problem* is the minimum degree of any algorithm that solves the problem.

2.2 Predicates for segment intersection

Algorithms for computing the intersections of line segments or monotone curve segments typically use a subset of the following predicates, which are all to be found in Figure 1.

a. *x-order of endpoints:* Compare the x-coordinates of endpoints of two curves.

b. *endpoint above/below curve:* Given a monotone curve segment s and an endpoint p whose x coordinate lies between the x-coordinates of s, determine if p is above or below s.

c. *curve intersection test:* Determine whether two monotone curve segments intersect at all. If it is known that the curves cross an odd number of times, then intersection can be tested using the x-order and above/below predicates.

d. *orientation* `CCW()`*:* Given three points, p, q, and r, determine if $\triangle pqr$ has a counter-clockwise orientation. This predicate is typically used to implement above/below or intersection tests for line segments.

e. *x-order of endpoint & intersection point:* Compare the x coordinates of an endpoint and an intersection point of two curves. This endpoint/intersection order test seems to be important for efficiency of segment intersection algorithms.

f. *intersect in slab:* Determine if the intersection of two curves occurs in the vertical slab defined by endpoints of two curves. Directly reducible to the previous endpoint/intersection order test.

g. *order of intersections on curve:* Given a monotone curve s, determine the order of particular intersection points with curves t and u. This predicate is required in order to build arrangements of curves.

h. *x-order of intersections:* Compare the x-coordinates of a pair of intersection points. This predicate is required in order to build trapezoidations of curves.

2.3 Degree of the segment intersection problem

Boissonnat and Preparata [5] have done an extensive study of the degrees of predicates and algorithms for segment intersection in the case where line segments are specified by the coordinates of their endpoints. We summarize their findings in this subsection, then consider their extension to circular arcs as an example.

Algorithm	Degree	$O(\cdot)$ time	Space	Solves
check pairs	2	n^2	n	seg intersection
Bentley-Ottmann [4]	5	$(n+k)\log n$	n	trapezoidation
Chazelle-Edelsb. [12]	4	$n\log n + k$	$n+k$	arrangement
Boiss.-Preparata [5]	3	$(n+k)\log n$	n	seg intersection
Balaban [3]	3	$n\log n + k$	n	seg intersection
this paper	2	$n\log^2 n + k\log n$	n	line seg inter.
Boiss.-Preparata [5]	2	$(n+k)\log n$	n	red/blue seg int.
this paper	2	$n\log n + k$	n	red/blue seg int.

Table 1: Degrees of selected line segment intersection algorithms

Since the intersection test can be reduced to the evaluation of degree 2 polynomials (e.g., by four orientation tests), the algorithm that checks all pairs demonstrates that the line segment intersection problem has degree 2.

Bentley and Ottmann's sweep algorithm [4] uses the predicates for *endpoint x-order* (degree 1), *endpoint/intersection order* (degree 3), and *intersection x-order* (degree 5). Chazelle and Edelsbrunner's algorithm [12] avoids the last, but does use the predicate for *intersection order along each segment* (degree 4). These two algorithms actually solve harder problems: Bentley-Ottmann can produce a trapezoidation of the line segments, which is a degree 5 problem, and Chazelle-Edelsbrunner can produce the arrangement of segments, which is a degree 4 problem [5].

Boissonnat and Preparata [5] describe a degree 3 "lazy sweep" algorithm that uses at most the *endpoint/intersection order* predicate; they observe that Balaban's optimal algorithm, which we describe in detail in Section 3.2, uses the *intersect in slab* predicate of the same degree. Table 1 lists degrees, running times, and working space requirements (excluding output size) for these algorithms.

The general algorithms of Bentley and Ottmann, of Balaban, and of Boissonnat and Preparata can compute the intersections of monotone curves that are pseudo-segments; it suffices to adapt the same predicates to curves. When the curves may intersect in more than one point, we can still adapt the algorithms of Bentley and Ottmann (or of Balaban) to perform a linear number of additional intersection tests during the sweep to make sure that no even parity intersections are missed.

As would be expected, predicates for curves have higher degree. To be more precise, we must specify the curves and their input representation. Table 2 lists bounds on the degrees of various predicates for line segments, as well as for three simple examples of representations of monotone circular arcs that are illustrated in Figure 2.

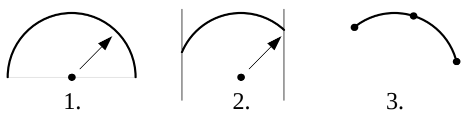

Figure 2: Representations for arcs

1. semi-circles defined by center, radius, and one bit to denote upper or lower half,
2. arcs of circles defined by center, radius, the x coordinates of endpoints, and one bit to denote upper or lower arc, and
3. arcs of circles defined by two endpoints and a middle point.

The proofs can be found in the appendix.

Predicate test	Line Segments	Semi-Circles	Circle/ x-range	3-point Arc
a. x-order of endpoints	1	1	1	1
b. endpoint above/below curve	2	2	2	4
c. curve intersection test	2	2	4	12
e. x-order of end & intersection	3	4	4	12
g. order of intersections on curve	4	6	6	16
h. x-order of intersections	5	12	12	44

Table 2: Degrees of predicates for circular arcs under three different representations

3 General segment intersection

Boissonnat and Preparata [5] asked whether there is an $O(n \log n + k)$ algorithm of degree 2 for line segment intersection. We show that there is none for curve segments by giving a lower bound of $\Omega(n\sqrt{k})$ in Section 3.1. Then, after describing Balaban's optimal, degree 3 algorithm in Section 3.2, we show in Section 3.3 that for line segments, Balaban's algorithm can be made degree 2 with a logarithmic loss in efficiency.

Under restricted predicates, it can be helpful to consider how the input curves can be deformed without changing the results of any predicates. If an algorithm uses only the predicates for endpoint orientation and whether endpoints are above or below a curve, then the dashed curve segment in Figure 3 is equivalent to the line segment r—an algorithm cannot distinguish between them.

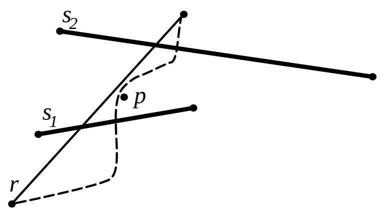

Figure 3: Witness p for $r \cap s_1$

Since the intersection point does not have a definite location when the segment r is deformed, we define the *witness* for the intersection of r and s to be the leftmost endpoint p that certifies that the order has changed from the initial order. In Figure 3, endpoint r is initially below segments s_1 and s_2, which can be checked by applying the above/below test to curves and left endpoints. Point p is the witness for the intersection of r with s_1, since r is above and s_1 is below p. The right endpoint of r is the witness for intersection with s_2.

Under different terminology, witnesses also play an important role in Boissonnat and Preparata's lazy sweep [5]. When a point p is the next endpoint to be processed, then they call *prime* the pairs of consecutive segments having p as witness.

3.1 A lower bound for curves

In this subsection, we give a lower bound for algorithms that must compute the intersection of curves from arbitrary computational tests on curve segment endpoints, tests whether an endpoint of one curve is above or below another curve, and tests that determine whether, but not where, two curves intersect. Using the deformability of the curves, it is not hard to show that any such algorithm requires $\Omega(n\sqrt{k})$ tests even to count the number of intersections. (Mention of intersection tests in the following theorem is actually redundant for pseudo-segments, since we can implement the intersection test with a contant number of above/below predicates.)

Theorem 1 *Any algorithm that counts all k intersecting pairs among a set of n segments and uses only order and orientation tests on endpoints, above/below tests for endpoints and curves, and intersection tests for curves, requires $\Omega(n\sqrt{k})$ time.*

Proof: Suppose that the algorithm asks for test results from an adversary. We describe the behavior of an adversary that holds $n + m$ curves and answers in such a way that the algorithm must ask nm queries to determine whether there are $\binom{m}{2}$ or $\binom{m}{2} + 1$ intersections.

The adversary fixes all the curve endpoints so that m long curve segments that all cross each other pass from left to right above n shorter curve segments, as in Figure 4. The algorithm can perform whatever computation it wishes on endpoints. For above/below or intersection tests involving two long segments, the adversary reports the order or intersection. For queries involving one or two short segments, the adversary reports "no intersection."

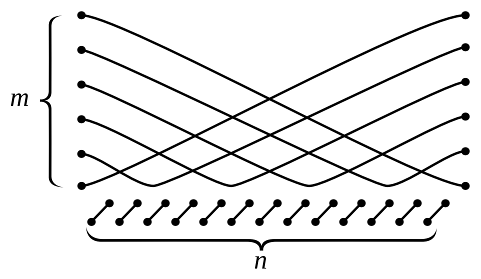

Figure 4: The $n + m$ curves of the adversary

Because a reported intersection says nothing about the coordinates of the intersection, if the algorithm fails to ask one of the nm queries about the intersection of a short segment s with a long curve γ, then the adversary can deform the arrangement of long curves so that γ has its minimum y coordinate immediately above the upper endpoint of s. Without changing the intersection patterns or witnesses for any other curves, the adversary has the freedom to make γ intersect s or not, so the algorithm cannot have the correct number of intersections. ∎

Chan, in private communication, [11] has given a randomized algorithm that will run in expected $O(n\sqrt{k \log n})$ time.

3.2 Balaban's algorithm

In 1995, Balaban gave a clever algorithm for the line segment intersection problem [3]. He first described an algorithm that runs in $O((n + k) \log n)$ time and uses $O(n)$ space, then used ideas from fractional cascading to remove the $\log n$ factor from the k. We give a high-level description of his first algorithm, and a more detailed count of the types of predicates that it uses.

Balaban applies his algorithm to a *slab* consisting of all points with x coordinates in the half-open interval $(a, b]$, where a and b are x coordinates of endpoints of segments. Given a set S of segments that intersect the slab and whose vertical order along the line $x = a$ is known (for those segments that end to left of the slab), he computes all intersections in the slab and the order of the segments along the line $x = b$.

Segments that intersect both lines $x = a$ and $x = b$ are said to *span* the slab. If all the segments span the slab, then we have a portion of an arrangement of lines, and the algorithm can find the intersections and order along $x = b$ by a sorting procedure that will be given in detail in Lemma 3. Thus, we first focus on the more interesting case in which some segments end inside the slab.

A subset $A \subset S$ of the segments spanning a slab is called a *staircase* if no two of the segments of A intersect, and A is maximal—any other segment of S spanning the slab intersects a segment of A. To find a staircase, Balaban uses a simple greedy procedure that he calls `Split()`. Figure 5 illustrates segments spanning a slab, and the staircase found by `Split()`.

Lemma 2 *Given the segments S that intersect a slab $(a, b]$, and the order of those that intersect the line $x = a$, a staircase can be found by $O(|S|)$ endpoint/intersection order tests.*

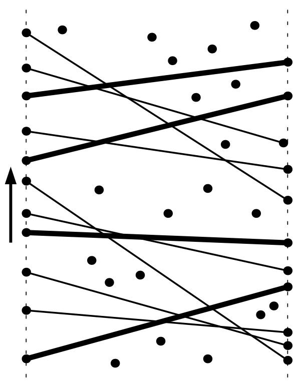

Figure 5: Staircase found by `Split()`

Proof: Consider the segments that span slab $(a, b]$ in order of increasing y coordinate along the vertical line $x = a$. We create a staircase that contains the first segment, then repeatedly test whether the next spanning segment s intersects the top segment in the staircase within the slab. If not, then add s to the staircase.

It should be clear that the segments added are disjoint—If s does not intersect the top segment in the slab, then it does not intersect any segment in the slab. They also form a maximal set—any segment not added intersects at least one segment in the staircase. Since we have assumed that endpoints define slab boundaries, we may use predicates that compare x-coordinates of endpoints to other endpoints and intersection points to determine which segments span the slab and check for intersection with the top segment in $O(1)$ time per segment. ∎

Balaban's algorithm recursively finds intersecting pairs among the segments not in the staircase. It cuts the slab into two at the median x coordinate of endpoints in the slab. Recursively finding the intersections in the left slab also produces the order along the cutting line. This allows the algorithm to recursively find the intersections in the right slab, which in turn gives the order of segments along the line $x = b$. The segments of the staircase can be merged into this order.

Lemma 3 *Let S be set of segments intersecting slab $(a, b]$ and S' be a staircase for $(a, b]$ such that there are k' intersecting pairs with a segment from each of S and S'. If the order of all segments intersecting $x = a$ is known, and, in each set, the order of segments that intersect $x = b$ is known, then the intersections between S and S' and the merged order along $x = b$ can be found using $O(|S| \log |S'| + |S| + |S'|)$ above/below tests and $O(|S| + |S'| + k')$ endpoint/intersection order tests.*

Proof: Endpoints of S can be located in the staircase by binary search using above/below tests. For any segment with both endpoints in the slab, we report intersections with every staircase segment between the two endpoints. We can do the same for segments that intersect $x = a$ but not $x = b$, since we know the order along $x = a$.

For segments that intersect $x = b$, we must find the merged order and intersections. This is quite easy to do in $O(|S| + |S'| + k')$ operations. First, merge S and S' using the ordering along $x = a$ or of endpoints in the slab. This will be the correct y-order along $x = b$ if there are no intersections ($k' = 0$), and it can be checked by asking whether adjacent segments intersect in the slab using endpoint/intersection x-order tests. Whenever an intersection is found, it is reported, and the intersecting pair are swapped in the y-order and new adjacencies are tested. This produces the correct order along $x = b$ by a sort algorithm whose running time is linear in $|S| + |S'|$ plus the number of inversions, which is the number of intersections, k'. ∎

Theorem 4 *Balaban's algorithm applies $O(n \log^2 n)$ above/below tests and $O(n \log n + k)$ endpoint/intersection order tests.*

Proof: Consider the recursion tree in which each node corresponds to a recursive call for a particular slab. The recursion tree splits the endpoints in a balanced fashion, and thus has depth $O(\log n)$.

We can account for above/below tests that arise in the merge (Lemma 3) by charging them to endpoints. Each of the $2n$ endpoints appears in the slab of at most one node per level, where it is charged for $O(\log n)$ tests, for a total of $O(n \log^2 n)$.

Endpoint/intersection order tests from splitting and merging (Lemmas 2 and 3) can be charged to intersections and segments. Each intersection point appears as a charge in the merge (Lemma 3) in at most one node: The nodes whose slabs contain intersection point q form a path from root to leaf; the charge for q is applied either at the leaf, or in the first node where one of the segments defines q joins the staircase—once a segment appears in a staircase at a node, it does not appear in a subtree of that node, so q cannot be charged twice.

Each segment s appears in $O(\log n)$ nodes where s ends in the corresponding slab, in $O(\log n)$ nodes where s spans the slab but does not span the parent, and in nodes where s spans the slab and the parent, because it intersects a segment in the staircase of the parent. Thus, the total number of endpoint/intersection tests charged against intersections and segments is $O(n \log n + k)$. ∎

We can see that Balaban's algorithm is degree 3 for line segments; it makes heavy use of the predicate for comparing x coordinates of endpoints and intersection points to determine whether intersection points occur in the slab of a node. It can apply to curves as well as line segments if we simply provide correct implementations of the predicates.

3.3 An output-sensitive, degree-2 algorithm for segments

The lower bound of Section 3.1 shows that Balaban's algorithm cannot be modified to find the intersections of pseudo-segments without the predicate for endpoint/intersection order. We were surprised to find that it can find the intersections of segments using only the degree-two `CCW`() test and with a logarithmic loss of efficiency. The key observation is that the lower-bound adversary can use the flexibility of the pseudo-segments to force the algorithm to explicitly obtain "no" answers to all intersection tests. With line segments, a group of "no" answers can be obtained by using `CCW`() tests to form convex hulls of endpoints and then testing tangents to the hulls. We describe this modification in this section.

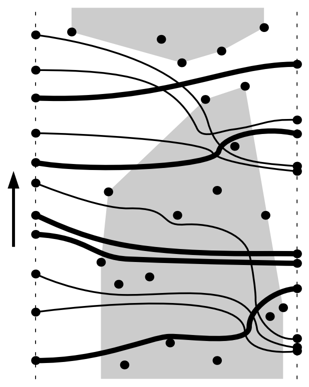

Figure 6: Convex hulls used by `Split`()

We can conceptually deform the curves to push intersections to the right without crossing endpoints. This deformation preserves the witness for each intersection. Figure 6 shows the deformation applied to the slab of Figure 5. Notice that this is equivalent to assume that two segments "intersect in a slab" if and only if

1. they intersect and
2. the witness belongs to the slab.

We assume that for a slab $(a, b]$ the vertical order received along $x = a$ and produced along $x = b$ are the orders of the deformed curves. We must modify the split and merge operations to respect this new order.

For `Split`(), we again consider the segments spanning the slab in increasing order and start by adding the first segment s to the staircase; s will always denote the highest segment in the staircase.

To keep track of potential witnesses in a slab, we maintain two convex hull structures: a deletion-only hull structure $\mathcal{A}$ [17] that contains all endpoints in the slab above s, and an insertion-only hull structure $\mathcal{B}$ [22], that contains all endpoints below s. (It is sufficient to store the lower hull for $\mathcal{A}$ and the upper hull for $\mathcal{B}$.)

A spanning segment t that enters the slab above s has a witness to intersection with s in the slab if and only if a point of $\mathcal{B}$ lies above t. It is sufficient to test one point—the vertex of the hull $\mathcal{B}$ whose tangent is parallel to t—and this test point can be found on the hull in $O(\log n)$ time.

If there is no witness, then t can be added to the staircase. We remove the points below t from $\mathcal{A}$—by repeatedly deleting the vertex of $\mathcal{A}$ with tangent parallel to t—and insert these points into $\mathcal{B}$. Then segment t becomes the new s. This completes the `Split()` operation; the final hulls are shaded in Figure 6.

Lemma 5 *Given the segments S that intersect a slab $(a, b]$, and the order of those that intersect the line $x = a$, we can compute a staircase using $O(|S| \log n)$* `CCW()` *tests.*

We employ the convex hulls $\mathcal{A}$ and $\mathcal{B}$ again to find the intersections of the staircase with the remaining segments and determine the ordering along the line $x = b$. Notice that the convex hulls have already solved problem of locating the endpoints in the slab in the staircase. Furthermore, for segments whose right endpoint is in the slab, knowing the endpoint location and the order along the line $x = a$ is sufficient to find all intersections. We therefore assume that we have a staircase in which we have located the left endpoints of segments and we want to find their order as they cross the line $x = b$.

Recall that we cannot determine the true order, but want the order consistent with moving intersections to the right while respecting witnesses and the monotonicity condition. Fortunately, this is easier than it sounds. We use two symmetric passes to find intersections; one for segments "going up the staircase" and the second for segments "going down." To go up, build the lower hull $\mathcal{A}$ of points above the lowest segment s of the staircase, and test each segment t whose left endpoint is below the stair s to see whether any point of $\mathcal{A}$ is also below t. The answer is yes if and only if s and t intersect in the slab; if they intersect then we can exchange their order along the line $x = b$. Thus, in $O(\log n)$ time for each segment and each intersection discovered, we obtain the intersections and the order.

Lemma 6 *Let S be a set of segments intersecting slab $(a, b]$ and S' be a staircase for $(a, b]$ such that there are k' intersecting pairs with a segment from each of S and S'. If the order of all segments intersecting $x = a$ is known, and, in each set, the order of segments that intersect $x = b$ is known, then the intersections between S and S' and the merged order along $x = b$ can be found using $O((|S| + |S'| + k') \log n)$* `CCW()` *tests.*

Theorem 7 *Balaban's algorithm can be modified to solve the line segment intersection problem using $O(n \log^2 n + k \log n)$* `CCW()` *tests.*

Proof: As in the proof of Theorem 4, we account for the tests from split and merges by charging them to endpoints and segments in the recursion tree. We describe only the modifications.

Each endpoint is now charged for convex hull insertion and deletion in each of the $O(\log n)$ slabs that contain it. The total cost of convex hull operations is bounded by $O(n \log^2 n)$.

Because we have replaced each of Balaban's constant-time "intersection-in-slab" test by a tangent computation, each segment and intersection is now charged $O(\log n)$ instead of $O(1)$. Thus, the total number of tests in the algorithm is $O(n \log^2 n + k \log n)$. ■

Balaban was able to obtain an optimal algorithm by shaving a factor of $\log n$ off the charge to endpoints by using the location of endpoints in slabs to help the location in their parents. Unfortunately, our extra logarithmic factor enters also on the charges to segments; it is not clear to us how to remove it.

4 Red/blue curve intersection

An important special case of the segment intersection problem is the *red/blue segment intersection*, in which the input is given as two sets of curves, one red and one blue, such that there are no red/red or blue/blue crossings. When the curves are pseudo-segments, having at most one crossing per pair, we can obtain an optimal $O(n \log n + k)$ running time by modifying Chan's trapezoid-sweep algorithm [10].

4.1 Chan's algorithm

Chan's algorithm for red/blue segment intersection [10] works as follows. First, compute a *trapezoidation* of the blue segments and the red endpoints. That is, compute the decomposition of the plane that results from extending vertical segment upwards and downwards from every red and blue endpoint to the first blue segment. This can be done in $O(n \log n)$ time by a standard *plane sweep* that keeps track of the ordering of blue segments crossing a vertical line as it sweeps across the plane.

Next, sweep over the trapezoids, where the *sweep front* is the boundary between those trapezoids that are entirely to the right of the vertical line $x = x_s$ and those that contain some point with x-coordinate at most x_s (shaded in Figure 7). During the sweep, maintain the invariant that all red/blue intersections have been reported for red segments to the left of the sweep front, up to the first (leftmost) intersection with the sweep front. In Figure 7, the intersections for dashed portions of the red segments have not been reported; this includes where the lowermost red segment recrosses the front.

The sweep front changes when x_s reaches the left side of a trapezoid. We re-establish the invariant by tracing any red segment that enters this trapezoid (whether from the top, bottom, or left side) through the trapezoidation until the red segment ends or reaches the sweep front. Segments that enter or leave through the top and bottom can be charged to intersections; segments that enter at left and leave at right can be considered as a group. The endpoint/intersection order test is sufficient to determine where a segment enters or leaves a trapezoid.

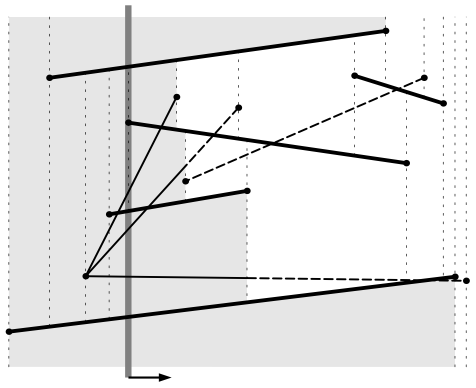
Figure 7: Sweeping trapezoids; "blue" segments are darker

As its data structures, Chan's algorithm needs only two ordered lists, one of trapezoids and one of red segments, ordered along sweep line $x = x_s$ or, equivalently, along the front. This makes his algorithm relatively easy to program, and causes it to perform considerably better than Bentley-Ottmann [4] or hereditary segment trees [21] on practical data [1]. It also works for curve segments where more than one intersection point is allowed, provided the endpoint/intersection order test has been implemented correctly.

4.2 Eliminating the test for endpoint/intersection order

With only above/below tests, it is impossible to determine where red segments enter and leave the trapezoids on the front. Figure 8 shows a deformation of the four red curves from Figure 7 that is consistent with the above/below tests on endpoints, even though the curves cross different sets of trapezoids. Once again, we must use endpoints as witnesses of intersection.

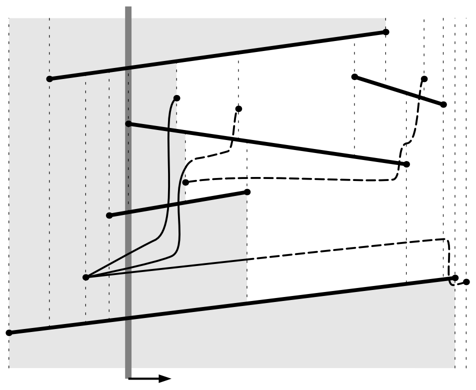
Figure 8: Push intersections to right

Conceptually deform the red curves, while respecting monotonicity and above/below tests, so that all intersections occur as far to the right as possible. This is how the deformation of Figure 8 was chosen. Then sweep with a modified invariant: that intersections with a red segment whose witnesses are in or behind the sweep front have been reported, up to the first chance for the segment to leave the sweep front.

To maintain the invariant, our algorithm maintains the ordered list of blue segments that intersect the sweep line $x = x_2$, and, for each trapezoid on the front, a *bundle* of the deformed red segments that have entered that trapezoid from the left. The blue list supports logarithmic-time insertion and deletion. Bundles support logarithmic-time insertion, deletion, and binary search for a point on the sweep line (with constant time if the point is above or below all segments of the bundle) and split and merge in time proportional to the size of the bundle split off or merged, with a maximum of logarithmic time. Both lists and bundles can be implemented as standard balanced search trees.

Our algorithm maintains pointers between adjacent bundles and blue segments, and assumes that each red segment can discover its bundle in logarithmic time. In Chan's original algorithm, bundles and their pointers were located when needed by searching lists of red and blue segments; we find it easier to establish correctness for sweeping deformed red curves if the algorithm maintains bundles explicitly.

Theorem 8 *The red/blue segment intersection problem for x-monotone pseudo-segments can be solved in optimal $O(n \log n + k)$ time using endpoint x-order and above/below tests.*

Proof: As noted in the previous section, work must be performed to re-establish the invariant when, and only when, the sweep line reaches a new trapezoid—that is, when it reaches an endpoint of a segment. We can assume, by using y coordinates to break ties in x coordinates, that if segments begin or end at the same x coordinate, then they begin or end at the same point. Thus, we distinguish cases by the color of this endpoint.

Red endpoint r: At a red endpoint one trapezoid, τ, ends and another, τ', begins. Also, some red segments may end at r and others may begin.

Let r_H and r_L denote the highest and lowest red segments ending at r. If the bundle for r_H is above τ, then r_H and all segments between r_H and τ must be traced through blue segments below until they enter τ. We can split the bundle of r_H, collect all bundles below r_H, and merge into the bundle for τ—this work can be charged to red/blue intersections that are discovered. On the other hand, if the bundle for r_L is below τ, then r_L and all segments between r_L and τ must be traced through blue segments above in a similar manner.

Finally, red segments ending at r can be deleted, and those beginning at r can be inserted. The total time for data structure manipulation is proportional to the number of intersections detected plus $O(\log n)$ times the number of segments beginning and ending.

Blue endpoint b: At a blue endpoint where i blue segments end and j begin, we have $i + 1$ trapezoids end (all but two of which are triangles) and $j + 1$ trapezoids begin (again, all but two are triangles).

Let τ_H denote the ending trapezoid whose upper right vertex is not b, and let τ_L denote the ending trapezoid whose lower right vertex is not b. Note that $\tau_H = \tau_L$ if blue segments start, but do not end at b. By binary search on the bundles for all trapezoids, we can find which bundle to split by the point b, and then split it. If this bundle is below τ_H, then it and bundles above are merged into τ_H and red/blue intersections are reported. Similarly, if this bundle is below τ_L, then bundles are merged into τ_L and intersections are reported.

Next, blue segments ending at b are deleted and those starting at b are inserted into the blue list. If no blue segment starts at b, the bundles for τ_H and τ_L are merged.

Searching, and splitting and merging bundles for τ_H and τ_L take $O(\log n)$ time. All other splitting and merging can be charged to intersections reported.

By induction, we can show that the invariants are correctly maintained. The total time is $O(n \log n + k)$. ■

4.3 Difficulties with red/blue curves

Since the modification above treats red segments as deformable curves, it should be no surprise that it works for pseudo-segments. When a pair of red and blue curves can intersect in more than one point, there are additional complications.

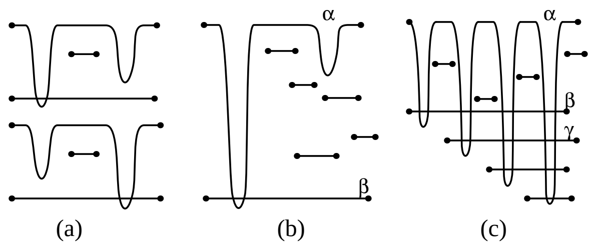

Figure 9: No witness

The primary difficulty is how to define a deformation of the red curves that moves intersection points to the right. Figure 9(a) illustrates that with two crossings the intersection can appear before or after another curve's endpoints without changing above/below relationships. Since many short curves may come between two longer, intersecting curves as in (b), there is no simple witness that limits how far an intersection can move.

5 Conclusions and open problems

We have shown that endpoint orientation tests, above/below tests, and intersection tests are not sufficient to give efficient, output-sensitive algorithms for the general problem of curve segment intersection, although they are sufficient to give an $O(n \log^2 n + k \log n)$ time algorithm for finding the k intersections of n line segment. In the red/blue case—where curves are colored red or blue and there are no red/red or blue/blue crossings—the k intersections for n line segments or n curve pseudo-segments can be found in optimal $O(n \log n + k)$ time.

Some open problems remain.

1. Is the logarithmic loss in efficiency necessary, or is there an optimal $O(n \log n + k)$ algorithm for line segment intersection using degree 2 predicates?

2. Is there an algorithm that achieves $\Theta(n\sqrt{k})$ for curve segment intersection using only endpoint orientation, above/below, and intersection tests? Chan [11] has recently communicated an algorithm that achieves $O(n\sqrt{k \log n})$.
3. Is there an efficient algorithm for red/blue curve segment intersections when pairs of curves may intersect in more than one point, using only endpoint orientation, above/below, and intersection tests?

A Establishing degree bounds for curves

In this appendix, we give the computations for the degree bounds listed in Table 2 for predicates for line segments and for the three example representations of monotone circular arcs from subsection 2.3 that are illustrated in Figure 2.

In all three representations of circular arcs, the x coordinates of endpoints are represented in the input, so comparing the x-order of endpoints is a degree 1 predicate.

When a circle is represented by center $(\frac{a}{2}, \frac{b}{2})$ and radius r, its standard equation is

$$x^2 + y^2 = ax + by + (r^2 - \frac{a^2}{4} - \frac{b^2}{4}) = ax + by - c. \tag{1}$$

When a circle is represented by three points, $p = (x_p, y_p)$, $q = (x_q, y_q)$ and $s = (x_s, y_s)$, the coefficients a, b and c are the solutions of the following system of three linear equations

$$\begin{pmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_s & y_s & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ -c \end{pmatrix} = \begin{pmatrix} x_p^2 + y_p^2 \\ x_q^2 + y_q^2 \\ x_s^2 + y_s^2 \end{pmatrix}$$

and we have

$$\begin{aligned} a &= \frac{1}{D} \begin{vmatrix} x_p^2 + y_p^2 & y_p & 1 \\ x_q^2 + y_q^2 & y_q & 1 \\ x_s^2 + y_s^2 & y_s & 1 \end{vmatrix}, \quad b = \frac{1}{D} \begin{vmatrix} x_p & x_p^2 + y_p^2 & 1 \\ x_q & x_q^2 + y_q^2 & 1 \\ x_s & x_s^2 + y_s^2 & 1 \end{vmatrix}, \\ c &= -\frac{1}{D} \begin{vmatrix} x_p & y_p & x_p^2 + y_p^2 \\ x_q & y_q & x_q^2 + y_q^2 \\ x_s & y_s & x_s^2 + y_s^2 \end{vmatrix}, \quad \text{with } D = \begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_s & y_s & 1 \end{vmatrix}. \end{aligned} \tag{2}$$

It follows from Equations (1) and (2) that the *incircle predicate* that decides whether a point lies inside a circle has degree 2 when the circle is defined by its center and its radius, and degree 4 if it is defined by three points.

To decide if a point A is above or below an arc γ, we first test if the point lies inside the circle ? supporting γ. If it does, then we can determine above or below, otherwise we compare the

y-coordinates of A and the center of ?. Thus, the degree of this predicate is again 2 when the circle ? is defined by its center and its radius, and 4 if it is defined by three points.

Consider now checking if two circles C_1 and C_2 intersect. Subtracting the equations for two circles in standard form gives a line, called the *chordal line* of the two circles, whose equation is

$$(b_1 - b_2)\, y = -(a_1 - a_2)\, x + c_1 - c_2. \tag{3}$$

Using Equation (3), we can eliminate the y variable in one of the equations of the circles (1). We then obtain a univariate polynomial of degree 2 in x, namely $P(x) = Ax^2 + Bx + C$ where

$$\begin{aligned} A &= (a_1 - a_2)^2 + (b_1 - b_2)^2 \\ B &= (b_1 - b_2)\,(a_1 b_2 - a_2 b_1) - 2(a_1 - a_2)(c_1 - c_2) \\ C &= (c_1 - c_2)^2 + (b_1 - b_2)(b_1 c_2 - b_2 c_1) \end{aligned} \tag{4}$$

The two circles intersect if and only if the discriminant $\Delta = B^2 - 4AC$ of $P(x)$ is positive. Easy computations show that

$$\Delta = (b_1 - b_2)^2 \left((a_1 b_2 - a_2 b_1)^2 - 4(a_1 - a_2)(a_1 c_2 - a_2 c_1) - 4(b_1 - b_2)(b_1 c_2 - b_2 c_1) - 4(c_1 - c_2)^2 \right).$$

It follows that evaluating the sign of Δ is a degree 2 computation when the circles are defined by centers and radii, or a degree 12 computation when the circles are defined by three points.

Consider now checking if two (monotone) circle segments γ_1 and γ_2 intersect, which can only happen if the supporting circles ?$_1$ and ?$_2$ intersect. Let I and J be the intersection points of those circles. If the two endpoints A_1 and B_1 of γ_1 lie on opposite sides of the chordal line H of ?$_1$ and ?$_2$, γ_1 contains either I or J. Otherwise, if γ_1 is an upper arc, γ_1 contains both I and J or none of them depending whether the y-coordinate of A_1 is below or above L. The case of a lower arc is similar. Since the same discussion can be applied to γ_2, we conclude that the degree of the intersection predicate for circle segments is the same as the degree of the intersection predicate for circles.

We next evaluate the degrees of the predicates for endpoint/intersection order and intersection x-order. Consider two intersecting circles. The roots of $P(x) = 0$ are the x-coordinates of the intersection points of the circles. It then follows that sorting the x-coordinate x_E of an endpoint with respect to the x-coordinates of the intersection points reduces to evaluating the signs of $(Ax_E^2 + Bx_E + C)$ and $(2Ax_E + B)$. Sorting the x-coordinates of the intersection points of two pairs of circles reduces to evaluating the sign of

$$A(-B' + \sqrt{B'^2 - 4A'C'}) - A'(-B + \sqrt{B^2 - 4AC}).$$

By squaring twice, it can be seen that this is equivalent to evaluating the sign of

$$\left((AB' - A'B)^2 - A^2(B'^2 - 4A'C') - A'^2(B^2 - 4AC) \right)^2 - 4A^2A'^2(B^2 - 4AC)(B'^2 - 4A'C'),$$

which can be rewritten as

$$16A^2A'^2\ \left((AC'-A'C)^2-(AB'-A'B)(BC'-B'C)\right).$$

Equations (4) show that, when circles are represented by center and radius, $P(x)$ has integral coefficients A, B, C of degrees 2, 3, and 4, respectively. It follows that the degree of the endpoint/intersection predicate is 4 and that the degree of the intersection/intersection predicate is 12. When circles are represented by three points, the degrees of A, B, C become 10, 11 and 12 respectively, and the degree of the endpoint/intersection and intersection/intersection predicates are 12 and 44.

To order the intersections along one circle segment, we decide whether the two corresponding intersection lines cross inside the circle or outside. This takes degree 6 or degree 16.

References

[1] D. S. Andrews et al. Further comparison of algorithms for geometric intersection problems. In *Proc. 6th Internat. Sympos. Spatial Data Handling*, pages 709–724, 1994.

[2] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

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

[4] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.

[5] Jean-Daniel Boissonnat and Franco P. Preparata. Robust plane sweep for intersecting segments. Technical Report RR–3270, INRIA Sophia Antipolis, September 1997. `http://www.inria.fr:80/RRRT/publications-eng.html`.

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

[7] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Sign determination in Residue Number Systems. *Theoret. Comput. Sci.*, 210(1):173–197, 1999. Special Issue on Real Numbers and Computers.

[8] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections.* Ph.D thesis, Universität des Saarlandes, March 1996.

[9] Christoph Burnikel, Jochen Könnemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[10] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 263–268, 1994.

[11] Timothy M. Y. Chan. private communication, August 1998.

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

[13] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, October 1992.

[14] A. Robin Forrest. Invited talk on computational geometry and software engineering. 2nd Annu. ACM Sympos. Comput. Geom., 1986.

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

[16] Leonidas J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[17] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[18] D. T. Lee and F. P. Preparata. Computational geometry: a survey. *IEEE Trans. Comput.*, C-33:1072–1101, 1984.

[19] Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.

[20] Victor Joseph Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic.* PhD thesis, Carnegie–Mellon University, Pittsburg, Penn., 1988.

[21] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graph. Models Image Process.*, 56(4):304–311, 1994.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[23] D. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations.* Ph.D. thesis, Dept. of mathematics, Univ. of California at Berkeley, 1992.

[24] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[25] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4(2):179–228, 1994.

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