2005

An Approximate Arrangement Algorithm for Semi-Algebraic Curves

Victor Milenkovic

Elisha Sacks
Purdue University, eps@cs.purdue.edu

Report Number:
05-012

AN APPROXIMATE ARRANGEMENT ALGORITHM FOR SEMI-ALGEBRAIC CURVES

Victor Milenkovic
Elisha Sacks

CSD TR #05-012
June 2005

An approximate arrangement algorithm for semi-algebraic curves

Victor Milenkovic
University of Miami

Elisha Sacks
Purdue University

Abstract

An arrangement algorithm is presented for plane curves. The inputs are 1) continuous, compact, x-monotone curves and 2) a module that computes approximate crossing points of these curves. The module output is ϵ accurate but can be inconsistent, meaning that three curves are in cyclic y order over an x interval. The algorithm sweeps the curves with a vertical line using the crossing module to compute and process sweep events. When the sweep detects an inconsistency, the algorithm breaks the cycle to obtain a linear order. The algorithm is correct for any input with no special treatment of degeneracies. The number of vertices in the output is $V = 2n + N + \min(3kn, n^2/2)$ and the running time is $O(V \log n)$ for n curves with N crossings and k inconsistencies. The output arrangement is realizable by curves that are $O(\epsilon + k^2\epsilon)$ close to the input curves, except in $k^2\epsilon$ neighborhoods of the curve tails. An implementation is described for semi-algebraic curves based on a numerical equation solver. This implementation is fast and accurate even on high degree inputs with many degeneracies.

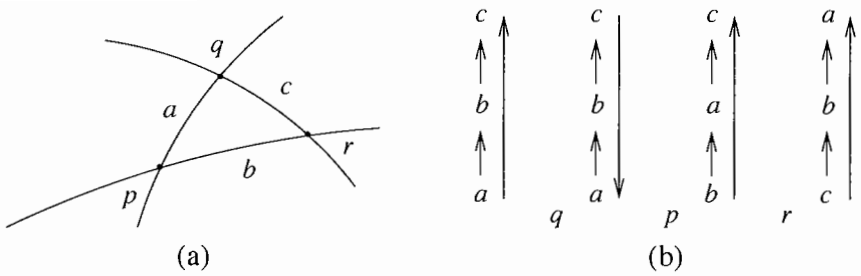

Figure 1: Inconsistency: (a) true geometry; (b) computed order depicted with vertical arrows.

1 Introduction

We present an arrangement algorithm for plane curves based on approximate computation of curve crossing points. The arrangement of n curves with N crossings can be computed in $O((n + N) \log n)$ time by sweeping. The analysis assigns unit cost to geometric operations, such as partitioning a curve into x-monotone segments, intersecting two curves, and sorting vertices along an axis. For semi-algebraic curves, the operations reduce to constructing algebraic numbers and computing the signs of polynomials at these numbers. The classical techniques for manipulating algebraic numbers incur a computational cost that grows rapidly with degree and bit complexity. The same problem arises in incremental insertion or in any other arrangement algorithm.

The mainstream approach to this problem is to accelerate the geometric computations via custom algorithms, constructive root bounds, and floating point filters. This approach has led to arrangement algorithms for lines, circles, conics, and cubics. We present an alternate research direction that constructs arrangements using approximate geometric computation via numerical equation solving. The motivation is that numerical solvers are highly accurate and are orders of magnitude faster than algebraic computation. The entire scientific computing community relies on numerical computation, so it should be applicable to computational geometry.

The first issue is that numerical solvers lack rigorous running time and error bounds, although their qualitative behavior is well understood. We take the computer science approach to this issue: define a computational model, verify it experimentally, and analyze algorithms in it. We encapsulate the numerics in a *crossing module* that computes the x values where a pair of curves cross and their y order between crossings. We specify that its running time is constant and that its output is ϵ accurate. This is a bound on the backward error: the distance between the input curves and *realization curves* for which the output is correct. The arrangement algorithm performs all geometric computations with the crossing module. We analyze the algorithm in terms of the module specifications, whereas we experimentally verify the module implementation.

The challenge is to reconcile the approximate nature of numerical computation with Euclidean geometry. Approximate geometric computations can violate the laws of geometry, just as floating point operations can violate the laws of algebra. In our computational model, this problem arises when the crossing module assigns three curves an inconsistent, cyclic vertical order. The canonical example (Figure 1) is curves a, b, c that form a triangle whose diameter is less than ϵ. The crossing module computes the vertices p, q, r with ϵ accuracy, incorrectly places q to the left of p, and correctly places r to the right of p. The curves are in cyclic order on (q_x, p_x): a is below b from the a/b output, b is below c from the b/c output, and c is below a from the a/c output.

We construct arrangements with a sweep algorithm that handles inconsistencies. The curves are swept with a vertical line using the crossing module to compute and process sweep events. When the sweep detects an inconsistency, it breaks the cycle to obtain a linear order. The algorithm is correct for any input with no special treatment of degeneracies. The number of vertices in the output is $V = 2n + N + \min(3kn, n^2/2)$ and the running time is $O(V \log n)$ for n curves with N crossings and k inconsistencies. The output arrangement is realizable by curves that are $O(\epsilon + k^2\epsilon)$ close to the input curves. As in any backward error analysis, the realization curves are proved to exist, but are not constructed.

The algorithm suffers from two weaknesses that reflect the gap between our computational model and the empirical reality of numerical computing. The running time and the error bound depend on k, which is $O(n^3)$. In practice, k is constant for generic input and is $O(N)$ for degenerate input. The error bound does not apply in a $k^2\epsilon$ neighborhood of the curve tails. We can fix this by extending each curve to the left by a short horizontal line segment, called a *telomere,* but the telomeres can cause $O(n^2)$ extra crossings. In practice, the error bound holds at tails without telomeres, and using telomeres speeds up the arrangement calculation.

We implement the arrangement algorithm for semi-algebraic curves. We use the eigenvector method [20] to compute the complex roots of systems of two polynomials. Root finding in the only numerical operation that we need to implement the crossing module. The mean/max ϵ values are $10^{-16}/10^{-12}$ for curves of degree 1 to 10. Finding the roots of two polynomials of degree d takes cd^4 time with $c = 18$ microseconds on a 3GHz Pentium. The running time and the error bound are independent of the number of inconsistencies k across a range of inputs.

We validate the software on the core operations of curve fitting and curve intersection. Any system that models with planar curves needs these operations. Exact methods are impractical because iteration generates points and curves of unbounded bit complexity: fit curves to points, intersect these curves to generate new points, fit curves to these points, and so on. The operations are challenging for approximate methods because degeneracy is common. One source of degeneracy is three curves that meet at a point by design, whereas three random curves meet with probability zero. Another source is nearly identical curves. It is easy to generate a copy of a curve by fitting a second curve to a set of its points. The two curves can differ slightly because of rounding error in the fitting algorithm. We show that our software handles both types of degeneracy.

The rest of the paper is organized as follows. Section 2 surveys prior work on arrangement algorithms. Section 3 specifies the input to our algorithm. Section 4 presents and analyzes the algorithm. Section 5 describes the implementation for semi-algebraic curves and its empirical validation. Section 6 discusses our results.

2 Prior work

We discuss prior work on arrangement algorithms that employ exact geometry, perturbation, and numerical approximation. Halperin [9] surveys arrangements with a focus on linear objects.

Exact Methods Exact Computational Geometry employs custom geometric algorithms, constructive root bounds, and floating point filters to compute correct combinatorial structures. Yap [23] surveys the approach. The main results are as follows.

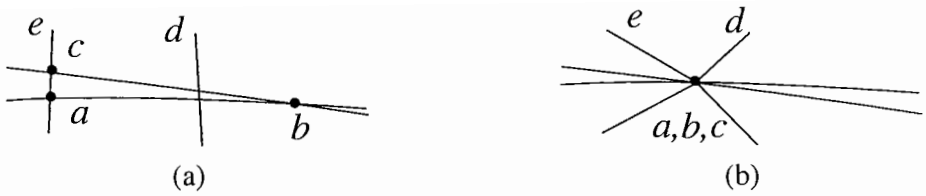

Figure 2: Collapsing triangle abc: (a) actual curves; (b) realization.

Keyser et al [12] compute arrangements of non-degenerate rational parametric curves with an $O(n^2)$ algorithm. Arranging 12 curves of degree at most 4 with 80 bit coefficients takes 1142 seconds on a 400MHz Pentium 2.

LEDA [13] and CGAL [5] compute arrangements of line segments via generalizations of Bentley's sweep algorithm that employ filtered rational arithmetic. Wein [21] extends the CGAL arrangement algorithm to conics. Arranging 20 random conics takes 2 seconds on a 450MHz Pentium 2. Berberich et al [3] extend the LEDA arrangement algorithm to conics. Arranging 60 random conics with 50 bit coefficients takes 49 seconds on a 846MHz Pentium 3. Eigenwillig et al [4] extend the LEDA arrangement algorithm to cubics. Arranging 60/90/120/250 random cubics with 100 bit coefficients takes 20/60/110/180 seconds on a 1.2GHz Pentium 3. Geismann et al [8] compute arrangements of special quartics (used to compute arrangements of 3D quadratics) with a sweep algorithm. Arranging 3 quartics with 30 bit coefficients takes 186 seconds on a Pentium 700. Wolpert [22] computes arrangements of nonsingular algebraic curves by an unimplemented sweep algorithm.

Mourrain et al [18] compute arrangements of 3D quadratics by an unimplemented plane sweep algorithm. Geismann et al [8, 19] compute arrangements of 3D quadratics. Keyser et al [11] compute arrangements of low-degree sculpted solids without degeneracies.

Perturbation Methods Halperin and Leiserowitz [10] compute arrangements of circles by a perturbation method that need not compute the correct combinatorial structure. They perturb the input so that each floating point computation is guaranteed to be correct with respect to the perturbed circles. Their method is useful when the perturbation is much less than the manufacturing accuracy, although the output may be incorrect for the input circles.

Numerical Methods Fortune [7] surveys prior work on numerical methods for robustness in computational geometry that attempts to make direct use of floating point in the spirit of numerical analysis. He notes two major approaches. One approach formalizes geometric rounding. Fortune points out that "the generalization to complex geometric objects is not straightforward." Triangles whose diameter is less than the floating point threshold (Figure 1) can round to points. The realization error can be large when the triangles are long and skinny (poor aspect ratio). For example, collapsing triangle abc in Figure 2 forces the realizations of lines d and e to intersect even though the true lines are far apart.

We follow the other approach that Fortune outlines: "A second floating-point approach is modeled on the error analysis of numerical methods, particularly linear algebra. The goal is to show that a suitably implemented algorithm provides an answer that is in some precise sense near the mathematically correct answer. Error analysis of geometric algorithms requires consideration of

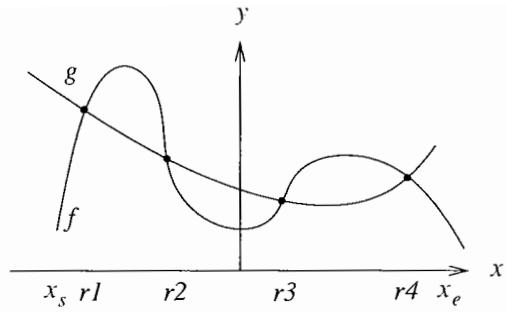

Figure 3: Sample crossing list.

both combinatorial and numeric structure. Often it is easy to argue that an algorithm produces combinatorially valid output... at least with suitably relaxed requirements. It has turned out to be much more difficult to argue that the numerical error associated with combinatorial structure is small. Full error analysis has been carried [out] only for a few simple algorithms." [7].

Our research distinguishes itself from prior work by expressing the running time and error in terms of the number of inconsistencies. We consider the approximate computation as a separate module and treat the number k of inconsistencies that it generates as an input property. We express the running time and the error in terms of k. We demonstrate experimentally that k is small, hence that the algorithm achieves "floating point speed with floating point accuracy." Moreover, we handle semi-algebraic curves, whereas prior work is restricted to linear objects.

Lines vs. Pseudo-Lines In an earlier survey paper [6], Fortune explains the meaning of "suitably relaxed requirements" for arrangements of line segments that are realized by pseudo-line segments. These are curves for which every pair intersects at most once. A pseudo-line arrangement is not necessarily a line arrangement. Recognizing a true line arrangement structure is equivalent to the existential theory of the reals, which is at least NP-hard and is in PSPACE. What is hard for lines is no easier for curves.

3 Input specification

The input to the arrangement algorithm is a set S of curves and a crossing module. A curve is a sub-manifold of the x, y plane that is the graph of a continuous function $y = f(x) : \mathbf{I}(f) \rightarrow \Re$ with $\mathbf{I}(f)$ a compact interval. Let $\min_x(f) = \min(\mathbf{I}(f))$ and $\max_x(f) = \max(\mathbf{I}(f))$. The curve endpoints are $\text{tail}(f) = (\min_x(f), f(\min_x(f)))$ and $\text{head}(f) = (\max_x(f), f(\max_x(f)))$.

The crossing module takes curves f, g and returns a *crossing list* $\langle f, g, r_1, r_2, \ldots, r_m \rangle$ where the r_i approximate the roots of $f(x) = g(x)$ at which their vertical order changes (Figure 3). The r_i are in the interior of $[x_s, x_e] = \mathbf{I}(f) \cap \mathbf{I}(g)$. The order f, g indicates that $f(x) < g(x)$ for $x_s \leq x < r_1$; otherwise the module returns $\langle g, f, r_1, r_2, \ldots, r_m \rangle$. The function $\text{next}(f, g, x)$ denotes the next crossing after x, the minimum element of $\{r_i \,|\, r_i > x\}$, and is undefined for $x \geq r_m$. The predicate $f <_x g$ denotes that f is below g at x according to the crossing list. It is true for $x \in [x_s, r_1) \cup [r_2, r_3) \cup [r_4, r_5) \cup \cdots$ and is false elsewhere in $[x_s, x_e]$. This definition implies $f <_x g \equiv \neg g <_x f$.

Curve endpoints and crossings are floating point numbers. The only operations that the sweep algorithm performs on them are $x < y$ and $x = y$, which are exact in floating point. We assume that $f <_x g$ and $\text{next}(f, g, x)$ are evaluated in constant time.

We assume that the crossing module is ϵ accurate according to the following definition. For the crossing list $\langle f, g, r_1, r_2, \ldots, r_m \rangle$, define $w_0 = [x_s, r_1 - \epsilon]$, $w_i = [r_i + \epsilon, r_{i+1} - \epsilon]$ for $i = 1, \ldots, m-1$, and $w_m = [r_m + \epsilon, x_e]$. If w_i is empty (for instance, $r_{i+1} - r_i < 2\epsilon$), redefine it as $w_i = [m_i, m_i]$ with $m_i = 0.5(r_i + r_{i+1})$, using $r_0 = x_s$ and $r_{m+1} = x_e$. If x is in some w_i and $f <_x g$, then 1) $f(x) \leq g(x)$ or 2) $\text{dist}((x, f(x)), g) \leq \epsilon$ and $\text{dist}((x, g(x)), f) \leq \epsilon$ where $\text{dist}((u, v), h)$ is the distance from point (u, v) to curve h. Redefining an empty w_i as a zero-length interval prevents an accurate crossing module from inserting in its output spurious pairs r_i, r_{i+1} of crossings with $r_{i+1} - r_i < 2\epsilon$. Section 5.2 shows that ϵ accuracy reflects what is reasonable to expect from a numerical solver.

The crossing module is inconsistent for curves f, g, h at x when they are in cyclic vertical order: $f <_x g$, $g <_x h$, and $h <_x f$. The inconsistent x lies in the interval $[r, s)$ that is bounded by the closest crossings/endpoints $r \leq x$ and $s > x$ in the three lists. The curves f, g, h and the interval $[r, s)$ comprise an inconsistency. Let k denote the number of inconsistencies among all triples of input curves for all x values. One could compute k by invoking the crossing module on all pairs of curves and examining the results for each triple. Thus, $k = \text{O}(cn^3)$ for curves with c crossings, where $c = \text{O}(d^2)$ for curves of algebraic degree d.

The concept of ϵ accuracy also covers degenerate intersections where two curves touch without crossing or are identical over an x interval. The crossing module omits the touching points or assigns them an arbitrary sign. It assigns either order to the identical intervals. Degeneracies and near degeneracies are the main cause of inconsistent crossing lists. The sweep algorithm detects and corrects them efficiently, accurately, and without symbolic perturbation.

4 Arrangement algorithm

The arrangement algorithm is a vertical line sweep that employs two data structures.

1. a list L of curves, called the *sweep list,* that represents the order of the curves from lowest to highest along a vertical sweep line. L is implemented as a red-black binary tree whose in-order traversal is the list order. The successor and predecessor of f in L are denoted $\text{succ}(f)$ and $\text{pred}(f)$.

2. a priority queue P of events: $\text{insert}(f, x)$, $\text{remove}(f, x)$, $\text{swap}(f, g, x)$, and $\text{check}(f, g, x)$. Events are dequeued in increasing x order. Ties are broken arbitrarily, except that removes come before other events and inserts come after other events. P is implemented as a heap.

For each $f \in S$, the algorithm initializes P with $\text{insert}(f, \min_x(f))$ and $\text{remove}(f, \max_x(f))$. It then repeatedly dequeues an event from P and processes it as follows.

- $\text{insert}(f, x)$: Insert f into L: if $f <_x g$ at node g, go left else go right. If f is not first in L, enqueue $\text{check}(\text{pred}(f), f, x)$ into P. If f is not last in L, enqueue $\text{check}(f, \text{succ}(f), x)$.

- $\text{remove}(f, x)$: If f is neither first nor last in L, enqueue $\text{check}(\text{pred}(f), \text{succ}(f))$. Remove f from L.

- swap(f, g, x): If $g \neq \text{succ}(f)$, discard the event. Otherwise, swap f and g in L. Enqueue check(g, f, x). If f is not last, enqueue check$(f, \text{succ}(f), x)$. If g is not first, enqueue check$(\text{pred}(g), g)$.

- check(f, g, x): If $g \neq \text{succ}(f)$ or $\max_x(f) \leq x$ or $\max_x(g) \leq x$, discard the event. If $g <_x f$, enqueue swap(f, g, x). Otherwise, if the next crossing $r = \text{next}(f, g, x)$ is defined, enqueue swap(f, g, r).

Event processing presupposes that curves can be located in L. Location is performed by assigning every curve a pointer to its tree node, and every node a pointer to its parent. The evolving sweep list is converted to an arrangement structure using standard techniques. When P becomes empty, the sweep ends and the arrangement is complete. Vertical line segments can be added to the output in linear time; we omit the details.

4.1 Running Time

The sweep defines an *output crossing list* for each pair of curves. Let $L(r)$ denote the state of L immediately after the algorithm finishes processing every event in P with $x \leq r$. Let $f <'_r g$ denote that f precedes g in $L(r)$. The f, g output crossing list is $\langle f, g, r'_1, r'_2, \ldots, r'_{m'} \rangle$ where the r'_i are the x values where f and g swap in L. The r'_i are identical to the r_i in the absence of inconsistency, but differ when swaps are discarded or are added at non-crossings by check events. Due to inconsistency, it is even possible that $g <_{x_s} f$ yet $f <'_{x_s} g$ at $x_s = \max(\min_x(f), \min_x(g))$: the input crossing list is $\langle g, f, r_1, r_2, \ldots, r_m \rangle$ and the output crossing list is $\langle f, g, r'_1, r'_2, \ldots, r'_{m'} \rangle$. We have $f <'_x g$ for $x \in [x_s, r'_1) \cup [r'_2, r'_3) \cup [r'_4, r'_5) \cup \cdots$ and $g <'_x f$ elsewhere in $[x_s, x_e]$.

We show that the output crossing lists are consistent and have $C = N + \min(3kn, n^2/2)$ crossings. This implies that the arrangement has $V = 2n + C$ vertices and that the running time is $O(V \log n)$.

Lemma 4.1 *The output crossing lists are consistent.*

Proof. If $f <'_x g$ and $g <'_x h$, f precedes g precedes h in $L(x)$, so f precedes h and $f <'_x h$. □

Lemma 4.2 *Immediately after* insert(f, x) *is processed,* $\text{pred}(f) <_x f$ *and* $f <_x \text{succ}(f)$.

Proof. After insertion and before balancing, f is a leaf. Its successor is the nearest ancestor g whose left subtree contains f. The insertion went left at g, so $f <_x g$. Balancing the tree does not change the successor. The predecessor proof is analogous. □

Lemma 4.3 *If* $g = \text{succ}(f)$ *in* $L(x)$, *then* $f <_x g$.

Proof. Let $[a, b)$ be a maximal interval on which $g = \text{succ}(f)$ in $L(x)$. Some event at a establishes $g = \text{succ}(f)$ and enqueues check(f, g, a). There can be many establishing and disestablishing events at a; only the end result matters. The fact that $g = \text{succ}(f)$ in $L(a)$ implies that $f <_a g$. Otherwise, the check would enqueue swap(f, g, a) and $g = \text{succ}(f)$ in $L(a)$ would be false. The value of $f <_x g$ next changes at $r = \text{next}(f, g, a)$. If $r < b$ were true, swap(f, g, r) would be executed, which would contradict the maximality of $[a, b)$. Hence, $r \geq b$ and $f <_x g$ on $[a, b)$. □

This lemma shows that adjacent curves in $L(x)$ are in crossing list order. We generalize this *local consistency* property to sublists of $L(x)$. The list $H = h_1, \ldots, h_p$ is locally consistent when $h_i <_x h_{i+1}$ for $i < p$. It is *minimal* when removing any of $h_2, \ldots h_{p-1}$ yields an inconsistent sublist, which implies that $h_{i+2} <_x h_i$ for $i < p - 1$. Although non-adjacent curves in $L(x)$ need not be in crossing list order, they can be linked by a minimal locally consistent list of length $k_x + 2$ with $k_x \leq k$ the number of inconsistencies at x.

Lemma 4.4 *If $f <'_x g$, there exists a minimal locally consistent list from f to g of length at most $k_x + 2$.*

Proof. The list $h_1 = f, h_2 = \mathrm{succ}(h_1), \ldots, h_p = g$ is locally consistent by Lemma 4.3. If $h_i <_x h_{i+2}$ for some $i < p-1$, delete h_{i+1} to obtain a locally consistent list of length $p-1$. Repeat this process as long as possible to obtain a minimal list of length l. Each of the $l - 2$ triples of consecutive list elements is inconsistent at x, so $l - 2 \leq k_x$ and $l \leq k_x + 2$. □

Lemma 4.5 *If $f <'_x g$ and $g <_x f$, then f, h_2, h_3 are inconsistent at x for some $h_2, h_3 \in S$.*

Proof. Form the minimal locally consistent list from $h_1 = f$ to $h_l = g$. We have $l > 2$ because $g <_x f$, so $h_1 = f, h_2, h_3$ are inconsistent at x. □

Lemma 4.6 *The algorithm executes at most $C = N + \min(3kn, n^2/2)$ swap events.*

Proof. Let the crossing list for $f, g \in S$ be $\langle f, g, r_1, \ldots, r_m \rangle$, so $f <_x g$ is constant on the $m + 1$ intervals $(-\infty, r_1), [r_1, r_2), [r_2, r_3), \ldots, [r_{m-1}, r_m), [r_m, \infty)$. The only time $\mathrm{swap}(f, g, x)$ is executed is when $g = \mathrm{succ}(f)$ and $g <_x f$. This makes $g = \mathrm{succ}(f)$ false, so no further swaps are enqueued in the current interval. Therefore, at most one swap is executed in each of the $m + 1$ intervals.

If a swap is executed at $a < r_1$, it is $\mathrm{swap}(g, f, a)$, since $f <_a g$, and g precedes f in L before the swap. Suppose f is inserted later than g and let $b = \min_x(f)$. If $b = a$, Lemma 4.2 implies that f cannot be inserted as $\mathrm{succ}(g)$. Inserts are processed after deletes and swaps, so the intervening curves persist in $L(a)$ and f, g cannot swap at a. We conclude that $b < a$, $g <'_b f$, and f belongs to an inconsistency according to Lemma 4.5. Charge the extra crossing to this inconsistency. Each curve can have $n - 1$ crossing lists, hence $n - 1$ extra crossings, and each inconsistency involves 3 curves. The k inconsistencies are charged at most $3k(n - 1)$ times. Also, there can be at most one extra crossing for each of the $n(n - 1)/2$ pairs of curves. □

Theorem 4.7 *The arrangement has at most $V = 2n + C$ vertices and the sweep has running time $\mathrm{O}(V \log n)$.*

Proof. Swaps generate at most C vertices by Lemma 4.6, insertions and deletions generate $2n$ vertices, and checks generate no vertices. This proves the vertex bound of V. Each insert, remove, and executed swap enqueues up to three checks. Each check enqueues at most one swap. Therefore, the total number of events is a constant times the number of insert, remove, and executed swaps, which is bounded by V. An event is processed by updating L and P in $\mathrm{O}(\log n)$ time. □

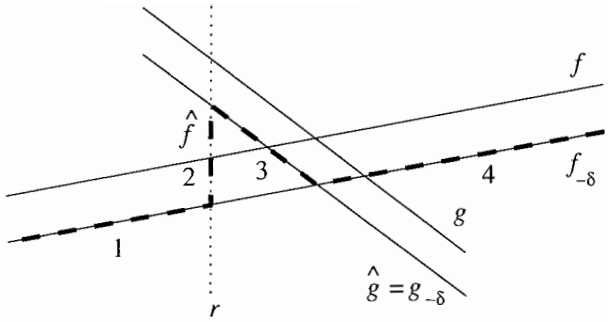

Figure 4: Realization of curves f and g near crossing r.

4.2 Realizability

The sweep algorithm constructs a combinatorial arrangement with one vertex per insert, remove, and executed swap. It determines the vertex x coordinates, but not their y coordinates. We prove that this combinatorial structure is realized by curves that are close to the input. The proof consists of three steps: (1) define offset curves that realize the sweep output; (2) show that the realization is δ accurate when the output crossing lists are δ *realizable;* and (3) prove δ realizability. A separate paper shows how to construct a generic embedding that realizes the endpoint y coordinates.

Lemma 4.8 *For every $x \in \mathbf{I}(f)$, $\mathrm{dist}((x,y),f)$ is unimodal in y.*

Proof. The distance is zero at $y = f(x)$ and is positive elsewhere. Let p and q be points such that $p_x = q_x$ and $p_y > q_y > f(p_x)$. It suffices to show that $\mathrm{dist}(q,f) < \mathrm{dist}(p,f)$. Let p' be a point of f nearest to p. If $p'_y \le q_y$, then $\mathrm{dist}(q,f) \le |qp'| < |pp'| = \mathrm{dist}(p,f)$. If $p'_y > q_y$, we know $f(p_x) < q_y$ and $f(p'_x) = p'_y > q_y$. By the intermediate value theorem, there exists $x \in (p_x, p'_x)$ such that $f(x) = q_y$. Let $q' = (x, f(x))$. Hence, $\mathrm{dist}(q,f) \le |qq'| = |q_x - q'_x| < |q_x - p'_x| = |p_x - p'_x| \le |pp'| = \mathrm{dist}(p,f)$. □

Lemma 4.8 proves the existence of curves $y = f_{+\delta}(x)$ and $y = f_{-\delta}(x)$ at distance $\delta > 0$ above and below f for $x \in \mathbf{I}(f)$. Using them, we realize each curve $f \in S$ with

$$\hat{f}(x) = \max_{e <'_x f} e_{-\delta}(x).$$

The condition $e <'_x f$ is shorthand for $e \in S$, $x \in \mathbf{I}(e)$, and $e <'_x f$. We define $f <'_x f$ for $x \in \mathbf{I}(f)$, so $f_{-\delta}(x)$ is included in the maximum. The function $\hat{f}(x)$ is possibly discontinuous when $\{e \mid e <'_x f\}$ changes, which can only happen at f crossings. We define $\hat{f}$ to be the continuous curve that results from filling in the discontinuities with vertical line segments.

Figure 4 shows the realization of line segments f and g near crossing r. The segments $f_{-\delta}$ and $g_{-\delta}$ are parallel to f and g. The realization curve $\hat{f}$ (shown with bold dashes) consists of four line segments. In segment 1, $\hat{f}(x) = f_{-\delta}(x)$ because $f <'_x g$. In segments 3 and 4, $\hat{f}(x) = \max\{f_{-\delta}(x), g_{-\delta}(x)$ because $g <'_x f$. The crossing r causes the discontinuity in $\hat{f}$ that is bridged by segment 2. The realization curve $\hat{g}$ is $g_{-\delta}$.

Lemma 4.9 *$f <'_x g$ implies $\hat{f}(x) \le \hat{g}(x)$.*

Proof. By Lemma 4.1, $e <'_x f$ and $f <'_x g$ imply $e <'_x g$. Therefore $\hat{f}(x)$ is the maximum of a subset of the elements of $\hat{g}(x)$ and $\hat{f}(x) \leq \hat{g}(x)$. □

Lemma 4.9 shows that vertical order of $\hat{f}$ and $\hat{g}$ equals the output crossing list order. The domain of $\hat{f}$ equals that of f. Thus, the $\hat{f}$ realize the output crossing lists. The vertical segments are a consequence of the discontinuity in $f <'_x g$ at output crossings. They can be eliminated by a local perturbation to obtain realization *functions* that equal zero at crossings.

Definition 4.1 *The f, g output crossing list is δ* **realizable** *at x if $f <'_x g$ implies $f_{-\delta}(x) \leq g_{+\delta}(x)$.*

Lemma 4.10 *If all output crossing lists are δ realizable at x, then $f_{-\delta}(x) \leq \hat{f}(x) \leq f_{+\delta}(x)$.*

Proof. Since $f_{-\delta}(x)$ is one of the elements in the definition of $\hat{f}$, $f_{-\delta}(x) \leq \hat{f}(x)$. For each e satisfying $e <'_x f$, $e_{-\delta}(x) \leq f_{+\delta}(x)$ by δ realizability, so $\hat{f}(x) \leq f_{+\delta}(x)$. □

It remains to prove that each f, g output crossing list is δ realizable, except near x_s. Let x_0 be a value such that $f <'_{x_0} g$. Select a minimal locally consistent list (MLCL) from f to g at x_0 (as defined prior to Lemma 4.4). Let x_1 be the minimal value such that this list is an LCL for $x \in [x_1, x_0]$. If f and g do not swap or start at x_1, select an MLCL at $x = x_1^-$, the largest floating point value less than x_1. Let x_2 be the minimal value such that this list is an LCL for $x \in [x_2, x_1)$. Similarly, construct $x_3, x_4, \ldots$. Let $l_j + 1$ denote the length of the LCL in the interval $[x_{j+1}, x_j)$.

Lemma 4.11 $l_j \leq k + 1 - j$.

Proof. By construction, f and g do not start or swap at x_i, $i = 1, 2, \ldots, j$. Since x_i is the leftmost value at which the MLCL at x_{i-1}^- remains an LCL, some other segment in the LCL must start at x_i or some pair of consecutive segments in the LCL must have a crossing at x_i. Therefore, x_i is at or to the left of the beginning of one of the inconsistencies in the MLCL. (The left endpoint of this inconsistency might be to the right of x_i because the LCL is not necessarily minimal at x_i.) Since each x_i^- is to the left of a different inconsistency, the number of inconsistencies in the interval $[x_{j+1}, x_j)$ is at most $k - j$. The result follows from Lemma 4.4. □

We say that an interval $[x_{j+1}, x_j)$ is long if $x_j - x_{j+1} \geq 2l_j\epsilon$. Our strategy for analyzing the realizability of f and g at x_0 depends on whether there is a long interval. Lemma 4.13 (and its helper Lemma 4.12) applies when there are no long intervals and Lemma 4.14 applies when there is a long interval.

Lemma 4.12 *Let $p = (x, y_1)$ and $q = (x, y_2)$ with $y_1 \leq y_2$. If $\mathrm{dist}(p, f) \leq \delta$ and $\mathrm{dist}(q, g) \leq \delta$, then $f_{-\delta}(x) \leq g_{+\delta}(x)$.*

Proof. By Lemma 4.8, $f_{-\delta}(x) \leq p_y$ and $q_y \leq g_{+\delta}(x)$. □

Lemma 4.13 *If $f <_x g$, then f and g are $|x_0 - x| + 2\epsilon$ realizable at x_0.*

Proof. x either lies in or bounds an interval between crossings in which $f <_{x'} g$. By ϵ accuracy, even if x lies within ϵ of a crossing, there must exist x' with $|x' - x| \leq \epsilon$ in that interval such that $f(x') \leq g(x')$ or $\mathrm{dist}((x', f(x')), g) \leq \epsilon$. We apply Lemma 4.12 to both cases. In the first case, let $p = (x_0, f(x'))$ and $q = (x_0, g(x'))$. Hence, $p_y \leq q_y$ and $\mathrm{dist}(p, f) \leq |x_0 - x'| \leq |x_0 - x| + |x - x'| \leq |x_0 - x| + \epsilon$ and, similarly, $\mathrm{dist}(q, g) \leq |x_0 - x| + \epsilon$. In the

second case, let $(x'', g(x''))$ be the point of g within ϵ of $(x', f(x'))$. We have $|x' - x''| \leq \epsilon$ and $|f(x') - g(x'')| \leq \epsilon$. Set $p = (x_0, f(x''))$ and $q = (x_0, g(x''))$. Hence $p_y \leq q_y$ and $\text{dist}(p, f) \leq |p - (x', f(x'))| \leq |x_0 - x'| + |g(x'') - f(x')| \leq |x_0 - x| + |x - x'| + \epsilon \leq |x_0 - x| + 2\epsilon$ and $\text{dist}(q, g) \leq |q - (x'', g(x''))| = |x_0 - x''| \leq |x_0 - x| + |x - x'| + |x' - x''| \leq |x_0 - x| + 2\epsilon$. □

Lemma 4.14 *If $[x_{j+1}, x_j)$ is long, then f and g are $(x_0 - x_j) + 2l_j\epsilon$ realizable at x_0.*

Proof. Set $x = x_j - l_j\epsilon$. Since $x_j - x_{j+1} \geq 2l_j\epsilon$, $[x - l_j\epsilon, x + l_j\epsilon] \subseteq [x_{j+1}, x_j]$. Let $h_0, \ldots, h_{l_j}$ be the MLCL from f to g selected at x_j^-. Set $p^0 = (x, h_0(x)) = (x, f(x))$. Since $p_x^0 = x$ is at least $l_j\epsilon$ distant from an h_0, h_1 crossing, either $h_0(p_x^0) \leq h_1(p_x^0)$ or $\text{dist}(p^0, h_1) \leq \epsilon$ by definition of ϵ accuracy. If the former, set $p^1 = (p_x^0, h_1(p_x^0))$. If the latter, set p^1 to the point on h_1 within ϵ of p^0.

In either case, $|p_x^1 - p_x^0| \leq \epsilon$, and therefore p_x^1 lies at least $(l_j - 1)\epsilon$ distant from an h_1, h_2 crossing. Similarly, we define p^2 from p^1, p^3 from p^2, until we reach p^{l_j} on $h_{l_j} = g$. Each step in the sequence $p^0, p^1, \ldots, p^{l_j}$ is either straight up in y or no more than ϵ in distance. Therefore, either $p_y^0 \leq p_y^{l_j}$ or $|p^{l_j} - p^0| \leq l_j\epsilon$. In any case, $|p_x^{l_j} - p_x^0| \leq l_j\epsilon$.

If $p_y^0 \leq p_y^{l_j}$, set $p = (x_0, p_y^0)$ and $q = (x_0, p_y^{l_j})$. It follows that $p_y \leq q_y$ and $\text{dist}(p, f) \leq |p - p^0| = x_0 - x = (x_0 - x_j) + l_j\epsilon$ and $\text{dist}(q, g) = |q - p^{l_j}| = x_0 - p_x^{l_j} = (x_0 - x_j) + (x_j - x) + |x - p_x^{l_j}| = (x_0 - x_j) + l_j\epsilon + |p_x^0 - p_x^{l_j}| \leq (x_0 - x_j) + 2l_j\epsilon$.

If $|p^{l_y} - p^0| \leq l_j\epsilon$, the remaining analysis is the same as Lemma 4.13 with x_j taking the role of x, p_x^0 taking the role of x', $p_x^{l_j}$ taking the role of x'', and and $l_j\epsilon$ taking the role of ϵ. □

Theorem 4.15 *If $x_0 \geq \max(\min_x(f), \min_x(g)) + K\epsilon$, then f and g are $(K + 2)\epsilon$ realizable at x_0 for $K = \min(k(k + 3), 2\min(k + 1, n - 1)\min(k, n + N))$.*

Proof. Consider the sequence $x_0, x_1, \ldots, x_m$ such that $m \leq k$: the sequence might be longer, but we stop at $m = k$. If the sequence has no long intervals, then $x_0 - x_m < 2l_0\epsilon + 2l_1\epsilon + \cdots + 2l_{k-1}\epsilon \leq 2((k + 1) + k + \cdots + 2) = k(k + 3)\epsilon$. On the other hand, an MLCL can have at most n elements and hence $l_j \leq \min(k+1, n-1)$. Each x_j^- is to the left of the left end of an inconsistency which is either a segment left endpoint or crossing and there are $\min(k, n + N)$ of those. Since $x_j - x_{j+1} < 2l_j$, $x_0 - x_m < 2\min(k + 1, n - 1)\min(k, N)$. Hence, $x_0 - x_m < K\epsilon$, and since $x_0 \geq \max(\min_x(f), \min_x(g)) + K\epsilon$, neither segment starts at $x = x_m$ and $f <'_{x_m^-} g$ is defined. If $f <'_{x_m^-} g$ is true, $m = k$ (otherwise we would not have stopped at m) and $l_m \leq k - m + 1 = 1$. Therefore, $l_m = 1$, $f <_x g$ for $x \in [x_{m+1}, x_m)$, and $f <_{x_m} g$. If $g <'_{x_m^-} f$, the arrangement algorithm must have swapped f and g at $x = x_m$. But a post-condition of a swap is $f <_{x_m} g$. By Lemma 4.13, f and g are $|x_0 - x_m| + 2\epsilon \leq K\epsilon + 2\epsilon = (K + 2)\epsilon$ realizable at x_0.

If the sequence has a long interval, let $[x_{j+1}, x_j)$ be the long interval nearest to x_0 (smallest j). All intervals to the right of x_j must be short and therefore $|x_0 - x_j| \leq 2l_0\epsilon + 2l_1\epsilon + \cdots + 2l_{j-1}\epsilon$. By Lemma 4.14, f and g are $|x_0 - x_j| + 2l_j\epsilon \leq |x_0 - x_j| \leq 2l_0\epsilon + 2l_1\epsilon + \cdots + 2l_{j-1}\epsilon + 2l_j\epsilon$ realizable at x_0. Since $j < m$, $j \leq k - 1$ and the argument of the previous paragraph bounds this sum by $K\epsilon$. □

Trimming $K\epsilon$ off the left end of each segment f means restricting its domain to $[\min_x(f) + K\epsilon, \max_x(f)]$. Let f^t denote the trimmed segment. The following corollary underlies our practical solution to the lack of an error bound near curve tails. We add a short horizontal "telomere" line segment to the tail of each segment, calculate the arrangement, and then trim off the telomeres. In cell biology, a telomere at the end of a strand of DNA loses a few base pairs every time the cell

divides. The telomere does not encode any genes: it merely acts to protect the genes from loss of information. Analogously, our telomere segments protect the input segments from insertion error in the arrangement algorithm.

Corollary 4.16 *If each segment f is constant (horizontal) for $x \in [\min_x(f), \min_x(f) + K\epsilon]$ and if $K\epsilon$ is trimmed off each segment, then for all x such that $f^t <'_x g^t$, f^t and g^t are $(K+2)\epsilon$ realizable at x.*

Proof. Given a horizontal segment ab ($a_x < b_x$ and $a_y = b_y$) and a point p such that $p_x \geq b_x$, $\text{dist}(p, ab) = |p - b|$. Therefore, if f is constant for $x \in [\min_x(f), \min_x(f) + K\epsilon]$ and if $p_x \geq \min_x(f) + K\epsilon$, then the point $(x', f(x'))$ of f closest to p must have $x' \geq \min_x(f) + K\epsilon$. Therefore, $\text{dist}(p, f) = \text{dist}(p, f^t)$.

By definition of trimming, $f^t <'_x g^t$ implies $x \geq \max(\min_x(f), \min_x(g)) + K\epsilon$ and hence f and g are $(K + 2)\epsilon$ realizable at x: there exists p and q such that $p_x = q_x = x$, $p_y \leq q_y$, $\text{dist}(p, f) \leq (K + 2)\epsilon$, and $\text{dist}(q, g) \leq (K + 2)\epsilon$. By the previous paragraph, $\text{dist}(p, f) = \text{dist}(p, f^t)$ and $\text{dist}(q, g) = \text{dist}(q, g^t)$, and hence f^t and g^t are $(K + 2)\epsilon$ realizable at x. □

5 Implementation

This section describes our implementation of the arrangement algorithm for semi-algebraic curves and our validation on highly degenerate inputs.

5.1 Semi-algebraic curves

Semi-algebraic curves are defined in terms of algebraic curves. An algebraic curve is the zero set of a polynomial $F(x, y)$. A curve point is regular when ∇F is nonzero and is singular otherwise. The regular points partition into 1D manifolds, called branches, that are topological circles or lines. Two or more branches meet at a singular point. We specify an input curve as a compact, x-monotone segment of a branch. Every compact semi-algebraic curve can be expressed as a finite disjoint union of such curves.

Figure 5 shows two algebraic curves that cross at r and s. Curve 1 consists of three branches: a topological circle and two topological lines (the left/right loops of a horizontal figure 8) that meet at singular point c. Curve 2 consists of two unbounded branches. The dots mark the singular and critical points, which delimit the allowable input curves.

5.2 Crossing module validation

We partition each input algebraic curve into monotone segments by solving $F = F_y = 0$ to find singular and turning points then sweeping to compute the branch arrangement. For any x-monotone segment f of F and any value of $x \in [\min_x(f), \max_x(f)]$, we can determine $f(x)$ as follows. Fix x and solve $F(x, y) = 0$ for y. Using the combinatorial structure of the arrangement, choose the correct root in y. For each pair of x-monotone segments f, g of F, G, the f, g crossing list is constructed by solving $F = G = 0$, assigning the crossings to the monotone segments of each algebraic curve, and sampling midway between crossings to determine the vertical order. We store crossing lists to avoid recomputation.

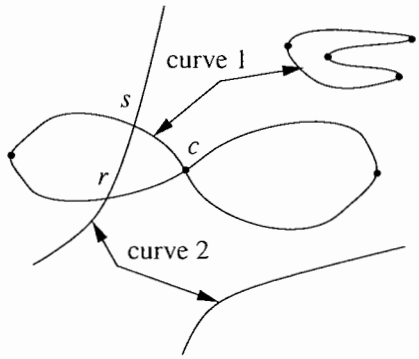

Figure 5: Algebraic curves.

We make no theoretical claims about these algorithms, hence omit the details. For both constructing the x-monotone segments and calculating their crossing lists, the sole numerical operation is computing the roots of a system of one or two polynomials by the eigenvector method [20].

Accuracy The crossing module is ϵ accurate based on extensive numerical experiments. The accuracy of numerical root finders is relative to the root magnitude, whereas our definition of ϵ accuracy is size independent. Hence, the crossing module is ϵ accurate for $\epsilon = D\mu$ where D is the magnitude of the largest root and μ is the floating point rounding unit (about 10^{-16} for ANSI double float). When we compute the portion of an arrangement inside a bounding circle of radius R, we discard exterior roots to obtain $\epsilon = R\mu$.

The ϵ-accuracy assumption is supported by an informal mathematical argument. Let F and G cross at p. Numerical analysis shows that the crossing module finds an approximate crossing, q, such that $|F(q)| < \epsilon$ and $|G(q)| < \epsilon$. The bound requires that the coefficients of F and G have bounded magnitude, which we enforce by scaling. The approximation error is $e = p - q$ and its magnitude is $C\epsilon$ with C the condition of the Jacobian matrix. Thus, the curves are in the correct vertical order outside a circle of radius $C\epsilon$ around p.

The condition is inversely proportional to the angle between the tangent lines and the magnitude of the gradients $\nabla F(p)$ and $\nabla Gn(p)$. Generically, the angle and the magnitudes are bounded away from zero, the condition is bounded, and the crossing module yields the correct vertical order outside a circle of order ϵ, hence is ϵ accurate by clause 1 of the definition in Section 3. When the gradients are near zero, the crossing point is nearly singular and the crossing module accuracy decreases. When the angle between the tangent lines is near zero, the circle radius is unbounded and clause 1 fails, but the perpendicular distance between the curves is $\mathrm{O}(\epsilon)$ inside the circle and the crossing module is ϵ accurate by clause 2.

We estimated ϵ on 10,000 pairs of algebraic curves of degree d with random coefficients in $[-1, 1]$. We sampled the crossing lists on the bounding box $-1 \leq x, y \leq 1$ with uniform spacing of 0.01 in x. For $p <_x q$, the error at x is bounded by $\min(0, q(x) - p(x))$. The mean/max errors are $10^{-16}/10^{-12}$ for degree 1–10. The arrangement algorithm validation provides similar ϵ estimates.

We expect the same accuracy for any input, except near singularities. Accuracy drops at isolated singularities. Intervals of singularity cannot be handled by any known numerical solver. Two such curves are identical or share a component. Floating point computation converts these into approximate properties. Nearly identical curves yield accurate crossing lists no matter what cross-

ings the solver computes. Shared components defeat our program. Nearly identical curves arise from the classical curve fitting operations, but shared components do not.

Speed We measured the running time for two curves of degree d. Theory ensures a polynomial bound. Experiments on random and degenerate inputs yields cd^4 time with $c = 18$ microseconds on a 3GHz Pentium 4 running RedHat 9 Linux. Half the time goes to matrix setup in C and the rest to eigenvalue computation with CLAPACK 3.0 [1]. A further factor of 2-4 speedup may be possible by using BLAS in matrix setup and by optimizing the CLAPACK implementation.

Inconsistencies The crossing module generates no inconsistencies on 100,000 triples of algebraic curves, of degree 1–10, with random coefficients in $[-1, 1]$. We replaced each curve $F(x, y)$ with $F(x, y) - F(0.1, 0.2)$, so that all the curves meet at $(0.1, 0.2)$ except for rounding error. Independently of degree, 6% of the curve triples are inconsistent on an interval of average length 10^{-15} near $(0.1, 0.2)$. We added a random constant in $[0, 10^{-14}]$ to each polynomial and obtained no inconsistencies. The arrangement algorithm validation yields about N inconsistencies for highly degenerate arrangements with N crossings. We conclude that inconsistencies cost nothing on generic inputs and cost at most a factor of two.

5.3 Arrangement algorithm validation

This arrangement algorithm validation has three goals. First, the algorithm handles inputs with many localized degeneracies: many vertices incident on many algebraic curves. Second, it handles "evil twins": multiple versions of the same algebraic curve with slightly different coefficients. Third, the speed and accuracy are within a factor of two of the limit imposed by the root solver. There is no log factor here: the algorithm is within a few percent of solving the fewest polynomial systems required to avoid missing arrangement vertices.

Section 5.3.1 describes how we generate "good" and "evil" arrangements to test the algorithm. Section 5.3.2 analyzes a typical output to support the goals of the validation. Section 5.3.3 reports statistical evidence that the typical example is representative.

5.3.1 Generating Arrangements

We validate our algorithm on inputs with many degeneracies and near degeneracies, which are the hardest cases for any algorithm. To create such an input, generate random sets of points in the unit square $|x|, |y| < 1$ and fit polynomials to them until there are 10 segments. Calculate the arrangement of these segments. Select random sets of vertices from this arrangement and fit polynomials to them until there are 90 more segments. Calculate the arrangement of the $10 + 90 = 100$ segments. Select random sets of vertices from these arrangements and fit polynomials to them until there are 900 more segments. Calculate the arrangement of the $10 + 90 + 900 = 1000$ segments. This is the "good" arrangement. To generate an "evil" arrangement, generate only 400 instead of 900 segments in this manner. Generate the remaining 500 segments as "evil twins" of the 100 segments in the second arrangement. To do so, select at random one of the polynomials of the first 100 segments. From the second arrangement, select a random set of vertices that lie on one of the segments of that polynomial. Fit an "evil twin" polynomial to those vertices. In the

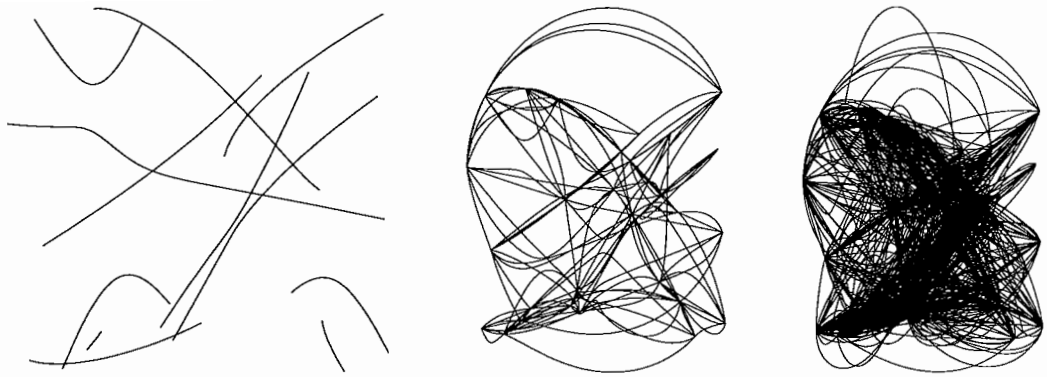

Figure 6: Input arrangements of degree 3.

absence of rounding error, the evil twin is identical to the original polynomial. However, there is rounding error in the calculation of the vertices and in the fitting to these vertices. Therefore, the evil twin is nearly identical to the original. Add segments from evil twins until there are 500 evil twin segments for a total of $10 + 90 + 400 + 500 = 1000$ segments.

A bivariate polynomial $F(x, y)$ of degree d has $D = (d + 1)(d + 2)/2$ coefficients. Since $F(x, y) = 0$ is independent of scale, there are only $D - 1$ degrees of freedom. Given $D - 1$ points, set one coefficient to 1 and plug the points into the equation to generate $D - 1$ linear equations in the other coefficients. To avoid stressing the linear solver, we reject sets of points with pairs nearer than 0.01. Subdivide the algebraic curve $F(x, y) = 0$ into monotone segments, and then select maximal subsegments that end at fit points or at turning points. If we detect a singularity, we reject F. Our algorithm handles singularities, but we validate on degeneracies only.

Figure 6 shows test arrangements with 10, 100, and 1000 segments of degree 3. The following is the histogram of incidences of algebraic curves on vertices from 1 to 50: 30 91044 935 455 262 140 95 58 55 45 50 34 24 25 24 20 17 15 11 12 10 7 9 9 5 5 7 1 3 3 0 6 1 3 1 0 1 0 5 0 1 0 1 2 0 0 1 0 0 1. For example, 30 vertices are incident on one algebraic curve. These include turning points, which are incident on two segments but only one curve. Most of the 91044 vertices incident on two curves are newly generated intersections. The evil twin arrangement (not shown) is even more degenerate: 80 41595 6029 2595 1257 668 425 290 192 150 108 110 68 44 44 29 28 28 24 17 25 22 11 17 20 13 11 16 11 11 8 7 4 4 6 5 4 2 0 4 4 2 0 2 0 1 0 1 1 0 0 0 0 1.

5.3.2 Typical Results

For the good arrangement, the estimated maximum realization error is the same as the maximum rootlist realization error ϵ, which in this example is $9.6194 \cdot 10^{-14}$. (We describe below how we estimate the realization error.) The total running time for the arrangement is 65 seconds, out of which 49 seconds were devoted to solving pairs of bivariate polynomial equations and classifying the roots. Although there were 1000 segments, there were only 359 algebraic curves (polynomial equations) with an average $1000/371 \approx 2.8$ segments each. Only 49061 out of a possible $359 \cdot 358/2 = 64261$ pairs of algebraic curves could "see each other": the pair was adjacent in the vertical sweep list at some point. The algorithm calculated roots for 49704 pairs, less than 2% more than the minimum necessary.

In this example, 50354 pairs of algebraic curves meet at vertices, which is about 1% more

than the number of pairs for which the algorithm calculated roots. For this reason, we consider it unlikely that a distribution-dependent technique, such as jacketing curves in polygons, would separate many non-intersecting curves and avoid root finding. No technique can separate curves that intersect, and the current algorithm already calculates roots for fewer pairs than intersect. However, if a distribution-dependent technique would help, it could be incorporated into our algorithm.

The number of pairs that see each other (49062) is about 75% of the theoretical maximum (64261). The reason that this percentage is so high is that algebraic curves have multiple segments and therefore more opportunities to see each other. We believe that pairs of segments meeting at turning points (vertical tangencies) are a possible source of numerical error, so we include them in the tests. To make sure that the algorithm is solving for the minimum number of roots even when fewer pairs see each other, we ran a test in which one segment per algebraic curve was selected for the final 900. This example has 109493 pairs that see each other, 113122 or about 3% more for which the algorithm calculates roots, and 124021 pairs or another 3% more that intersect. There are 933 algebraic curves, and so these number are about 25% of the maximum $933 \cdot 922/2 = 434778$. Again, the algorithm calculates very close to the theoretical minimum number of roots.

Each sweep event is an opportunity for incorrectly ordered segments to be properly correctly. Pairs of segments are never swapped unless their root list indicates. For this reason, we estimate the realization error by evaluating the error just to the left of each vertex. We evaluate the error for each segment incident on the vertex plus the segments directly above and below the vertex. For segments adjacent in the sweep, the maximum error is $9.6194 \cdot 10^{-14}$. Since adjacent segments are locally consistent, this error represents an estimate of the solver ϵ accuracy. Actually, this is an estimate of the solver's realizability, which is up to 2ϵ for accuracy ϵ. However, we will refer to this number as ϵ because it is easier to measure and is what matters in practice. We then looked at non-adjacent pairs of segments that were ordered contrary to their root list order. The maximum error for segments two apart in the sweep list is $1.25598 \cdot 10^{-14}$. The maximum error of segments three and four apart is $1.63053 \cdot 10^{-14}$ and $4.92924e \cdot 10^{-14}$. There was no error for segments farther apart in the sweep list. Hence, the estimated realization error is ϵ.

This experiment was run with a telomere length of 2^{12} rounding units. We chose this length to make it roughly comparable to the maximum ϵ we had seen. Telomeres do not decrease the error, but they reduce the running time by 28%. The root solving time is the same, so the improvement is even more dramatic. (Since the telomeres are horizontal, we calculate telomere/curve intersections by solving a univariate polynomial equation, and so telomeres do not directly increase the number of bivariate pairs to solve.) Paradoxically, adding telomeres makes it about twice as fast to compute the combinatorics of the arrangement. There is also a dramatic reduction in extra crossings: from 79 without telomeres (segments that are out of crossing list order when they first see each other) to 7 with telomeres. There are 2610 telomere/curve intersections, but these are removed when the telomeres are trimmed from the arrangement. The algorithm detects 4584 inconsistencies.

5.3.3 Statistics

This section describes statistics to support the typical results reported in the previous section. We ran 10 experiments on arrangements of algebraic degree 3 to 10. Tables 1 and 2 are for good and evil arrangements. For each pair of polynomial equations it solves, the algorithm checks if the number of roots equals the Bezout bound. If not, it samples to detect probable missing roots. It detected these for two good degree 10 arrangements and for one evil degree 10 arrangement. One

Table 1: Statistics for good arrangements.

degree	3	4	5	6	7	8	9	10
median N	108376	97431	91914	72095	62766	48257	37622	32951
maximum N	119237	113156	97867	82311	84872	54257	46680	34978
median k	9555	24148	34043	44025	18788	8188	6013	3381
maximum k	22537	30366	90591	70729	47346	13054	10561	6860
median $-\lg \epsilon$	44	41	41	40	40	40	40	39
minimum $-\lg \epsilon$	42	40	37	38	39	39	38	1
total time (sec.)	112	147	167	182	191	202	227	270
root time (%)	69	72	73	74	80	86	87	85
excess pairs (%)	3.3	2.2	0.5	0.1	0.0	0.0	0.0	0.0
extra crossings	14	16	31	74	21	9	12	18
error ($/\epsilon$)	1.0	1.0	1.1	1.0	1.0	1.0	1.0	1.1
speedup (%)	26	31	37	33	33	32	28	24
extra cross. red.	11	21	16	10	6	13	5	4
min. error ratio	1.0	1.0	1.0	0.5	1.0	1.0	0.9	1.0
max. error ratio	1.0	1.0	2.1	1.0	1.0	1.0	1.0	1.0

of these missing roots accounts for the very poor accuracy for degree 10 in Table 1.

The first column in Table 1 can be understood as follows. The first section describes the inputs: N is the number of segment intersections; k is the number of inconsistencies; and ϵ is the root list error. For each arrangement, the worst value is determined. The median refers to the median of the ten worst values. The second section describes the output. Calculating the roots used at least 68% of the running time. Roots were calculated for only 3.3% extra pairs of polynomials over the minimum necessary. There were at most 11 extra segment crossings resulting from inconsistencies: the $\min(3kn, n^2/2)$ term in the number of vertices. The error was 1.0 times ϵ: the $(1 + k^2)\epsilon$ term of the error. The third section describes the effect of telomeres. They improved the running time by 34%, reduced the number of extra crossing by at least a factor of 11, and changed the multiple of ϵ by factor between 1.0 and 1.0 (no change). The other columns are similar.

Root solving time always dominates. For the evil arrangements, the number of excess roots solved is larger, up to 25% for degree 3, but not excessively so, and so the running time is still within a small factor of the best it can be. The realization error was never more than 2ϵ. Adding telomeres sometimes made the realization error worse, but not by more than a factor of two. It always improved the running time and greatly reduced the number of excess crossings.

If $F(x, y) = 0$ and $G(x, y) = 0$ have nearly identical coeficients (evil twins), we solve for the roots of $F(x, y) = 0$ and $F(x, y) - G(x, y) = 0$ instead, with the second polynomial scaled so its largest magnitude coefficient is unity.

6 Conclusions

We have presented a robust arrangement algorithm for plane curves based on an ϵ accurate crossing module. Its performance is analyzed in terms of the number k of combinatorial inconsistencies that occur due to the approximation error. The running time and output size match those of the standard

Table 2: Statistics for evil arrangements.

degree	3	4	5	6	7	8	9	10
median N	78726	82809	76080	67779	54833	44966	37647	29829
maximum N	91118	93843	89879	77399	74077	52696	49203	34896
median k	48462	55682	67282	53411	32392	15447	13085	6909
maximum k	67639	69473	128594	92703	77796	22381	17617	12246
median $-\lg \epsilon$	43	41	43	41	40	40	38	31
minimum $-\lg \epsilon$	42	33	38	36	32	33	28	27
total time (sec.)	104	160	195	200	220	233	277	286
root time (%)	73	77	77	77	82	88	89	92
excess pairs (%)	22.4	13.3	4.5	1.7	0.9	0.2	0.2	0.0
extra crossings	12	24	37	76	23	9	23	43
error ($/\epsilon$)	1.2	1.6	1.0	1.0	1.6	1.0	1.3	1.4
speedup (%)	28	25	29	30	27	24	22	20
extra cross. red.	56	23	20	9	8	23	15	8
min. error ratio	1.0	1.0	0.6	1.0	0.8	0.7	1.0	1.0
max. error ratio	1.0	1.0	1.0	1.0	1.0	1.0	1.6	1.0

sweep algorithm with exact, unit-cost algebraic computation, plus a $kn \log n$ term with n the input size. The output accuracy is $\epsilon + k^2\epsilon$. We have presented extensive experimental evidence that inconsistencies at most double the output size even on highly degenerate inputs. Hence, the actual performance matches the standard sweep with floating point computation.

The only case where we found many inconsistencies is among triples of curves that almost meet at a point. The curves form a tiny triangle with 4 inconsistent vertex orders and 2 consistent orders. As the curve degree increases, the floating point resolution of the triangle decreases until the vertex order becomes essentially random. Small triangles occur in some applications. For example, consider the layout problem of cutting a maximum number of clothing parts from a strip of fabric. Every part will touch two other parts (or the strip boundary) in an optimal configuration, which implies that three contact curves cross in every three-part configuration space. In mechanical design, redundancy and symmetry can generate crossing triples of contact curves. Even so, the inconsistencies are confined to small regions. We conjecture that if the k inconsistencies are pairwise ϵ separated, where ϵ is the crossing module accuracy, then the running time is linear in k and the output accuracy is ϵ.

Inconsistency sensitive analysis is a new computational geometry paradigm that we plan to explore further. Our next goal is to construct and manipulate the configuration spaces of rigid planar parts, which are key to algorithmic part layout, mechanical design, and path planning. Another goal is solid modeling with explicit and implicit surfaces. In both cases, the computational geometry task is to arrange surface patches of high degree.

We also plan to develop iterative algorithms that cascade geometric computations, meaning that the output of each iteration is the input to the next iteration. Many non-geometric numerical algorithms use cascading, for example Newton's method. We believe that geometric algorithms would also use cascading extensively if there were an effective way to implement it. For example, Milenkovic uses cascaded numerical geometric operations in part layout [17, 14, 16]. However, one can construct any algebraic expression by cascading two simple geometric constructions: (1)

join two points to form a line and (2) intersect two lines [2, 15]. This suggests that exact geometric cascading is as hard as exact scientific computing, which is untenable. The shape modeling results suggest that our approach can make cascading practical by replacing this exponential factor with a small constant.

Acknowledgments

Research supported by NSF grants IIS-0082339 and CCR-0306214, and by the Purdue University Center for Graphics, Geometry, and Visualization.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] Behnke, Bachmann, Fladt, and Kunle. *Fundamentals of Mathematics, Volume II: Geometry*. MIT Press, Cambridge, MA, 1974.

[3] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Kurt Mehlhorn, and Elmar Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - Esa 2002, 10th Annual European Symposium*, volume 2461 of *Lecture Notes in Computer Science*, pages 174–186, Rome, Italy, 2002. Springer.

[4] Arno Eigenwillig, Lutz Kettner, Elmar Schömer, and Nicola Wolpert. Complete, exact, and efficient computations with cubic curves. In *Proceedings of the 20th ACM Symposium on Computational Geometry*, 2004.

[5] Eyal Flato, Dan Halperin, Iddo Hanniel, Oren Nechushtan, and Eti Ezra. The design and implementation of planar maps in CGAL. *The ACM Journal of Experimental Algorithmics*, 5(13), 2000.

[6] S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, Information Geometers, pages 81–128, 1993.

[7] S. Fortune. Robustness issues in geometric algorithms. In M. Lin and D. Manocha, editors, *Applied Computational Geometry*, volume 1148 of *Lecture Notes in Computer Science*, pages 9–14, New York, 1996. Springer.

[8] Nicola Geismann, Michael Hemmer, and Elmar Schömer. Computing a 3-dimensional cell in an arrangement of quadrics: exactly and actually! In *Proceedings of the 17th ACM Symposium on Computational Geometry*, pages 264–273, 2001.

[9] Dan Halperin. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of discrete and computational geometry*. CRC Press, Boca Raton, FL, second edition, 2004.

[10] Dan Halperin and Eran Leiserowitz. Controlled perturbation for arrangements of circles. In *Proceedings of the 19th ACM Symposium on Computational Geometry*, pages 264–273, San Diego, 2002.

[11] John Keyser, Tim Culver, Mark Foskey, Shankar Krishnan, and Dinesh Manocha. ESOLID—a system for exact boundary evaluation. *Computer-Aided Design*, 36(2):175–193, 2004.

[12] John Keyser, Tim Culver, Dinesh Manocha, and Shankar Krishnan. Efficient and exact manipulation of algebraic points and curves. *Computer Aided Design*, 32:649–662, 2000.

[13] K. Melhorn and S. Näher. *The LEDA platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[14] V. J. Milenkovic. Rotational polygon containment and minimum enclosure using only robust 2d constructions. *Computational Geometry: Theory and Applications*, 13:3–19, 1999.

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

[16] Victor J. Milenkovic. Densest translational lattice packing of non-convex polygons. *Computational Geometry: Theory and Applications*, 22:205–222, 2002.

[17] V.J. Milenkovic and K. Daniels. Translational polygon containment and minimal enclosure using mathematical programming. *International Transactions in Operational Research*, 6:525–554, 1999.

[18] Bernard Mourrain, Jean-Pierre Técourt, and Monique Teillaud. Sweeping an arrangement of quadrics in 3d. In *Proceeding of the 19th European Workshop on Computational Geometry*, pages 31–34, 2003.

[19] Elmar Schömer and Nicola Wolpert. An exact and efficient approach for computing a cell in an arrangement of quadrics. *Computational Geometry: Theory and Application*, 2003. To appear in Special Issue on Robust Geometric Algorithms and their Implementations.

[20] Hans Stetter and H. Michael Moeller. Multivariate polynomial equations with multiple zeros solved by matrix eigenproblems. *Numerische Mathematik*, 70:311–329, 1995.

[21] Ron Wein. High-level filtering for arrangements of conic arcs. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - Esa 2002, 10th Annual European Symposium*, volume 2461 of *Lecture Notes in Computer Science*, pages 884–895, Rome, Italy, 2002. Springer.

[22] Nicola Wolpert. Jacobi curves: computing the exact topology of arrangements of non-singular algebraic curves. In *Proceedings of the 11th ACM Symposium on Algorithms*, pages 532–543, 2003.

[23] Chee Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of discrete and computational geometry*. CRC Press, Boca Raton, FL, second edition, 2004.