Exact and Efficient 2D-Arrangements of Arbitrary Algebraic Curves

Arno Eigenwillig* Michael Kerber*

This is the author-prepared version of the article. The definite version is published in the Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA08). ©SIAM 2008

Abstract

We show how to compute the planar arrangement induced by segments of arbitrary algebraic curves with the Bentley-Ottmann sweep-line algorithm. The necessary geometric primitives reduce to cylindrical algebraic decompositions of the plane for one or two curves. We compute them by a new and efficient method that combines adaptive-precision root finding (the Bitstream Descartes method of Eigenwillig et al., 2005) with a small number of symbolic computations, and that delivers the exact result in all cases. Thus we obtain an algorithm which produces the mathematically true arrangement, undistorted by rounding error, for any set of input segments. Our algorithm is implemented in the EXACUS library AlciX. We report on experiments; they indicate the efficiency of our approach.

1 Introduction

A fundamental task in computational geometry is this: Given a set of line or curve segments in the plane, compute their *arrangement* [2] [29]. Many algorithms for this task have been studied, a popular one is the sweep-line method going back to Bentley and Ottmann [5]. We show how to implement the basic geometric operations on points and segments needed by the sweep-line method to compute the arrangement of segments of algebraic curves of arbitrary degree (i.e., vanishing loci of polynomials $f(x,y) = \sum_{i,j} a_{ij}x^iy^j$) with integer coefficients. Our approach produces the *exact* result in *all* cases, including all degeneracies, but is also fast (see the experiments in Sec. 5) due to a judicious combination of symbolic and adaptive-precision numeric computations.

Reconciling exactness and efficiency in the computation of non-linear arrangements has attracted a lot of attention in recent years (see below), which is not surprising due to the importance and wide applicability of arrangements in computational geometry. We mention just two examples – boolean operations on polygons bounded by algebraic curves (an immediate consequence of our result) and arrangements of intersection curves on a surface or in its parameter space (important for CSG-to-BRep conversion in CAGD) – and refer to the surveys [2] [29] for more.

Related work This work follows the Exact Geometric Computation (EGC) paradigm coined by Yap [44]: use any numeric or symbolic technique for the basic geometric operations that works well, as long as it produces the mathematically true result. Existing EGC algorithms for non-linear arrangements have handled circular arcs [17] [41], conics [39] [7] [24], cubics [22], intersection curves of spatial quadrics [9], Bézier curves [32], and non-singular algebraic curves [33] [42] but not, to our knowledge, algebraic curves in full generality. [1]

There are non-EGC approaches to arrangements of curves: Milenkovic and Sacks [36] compute an approximate arrangement of algebraic curves and prove, under certain assumptions on the underlying numerical solver, that there exists a perturbation of the input which realizes the computed arrangement. Halperin and Shelton [31] show how to actually carry out a *controlled perturbation* of the input such that fixed-precision arithmetic suffices; see also [30] and [35]. The guarantee offered by these methods bounds *backward error*:

*Max-Planck-Institut für Informatik, Saarbrücken
email:{arno,mkerber}@mpi-inf.mpg.de

[1] Recently in [13], also quartic curves have been considered

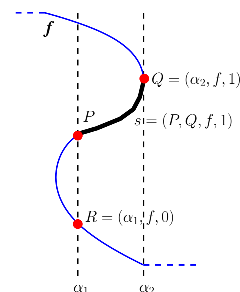
the result is correct for a slight (implicit or explicit) perturbation of the original input. Eigenwillig et al. [23] compute arrangements of Bézier curves with bounded *forward error*: the output is close to the true arrangement, but vertex coordinates of the true arrangement are not computed explicitly. These approaches have their merits but are simply inapplicable when an exact result is needed.

Besides computational geometry, symbolic computation [4] [14] [12] is a main source of important tools for our algorithm. The geometric analyses of curves and curve pairs in Sections 3 and 4 are essentially cylindrical algebraic decompositions (cads) of $\mathbb{R}^2$ for one and two polynomials, resp., augmented with adjacency information. Cads of $\mathbb{R}^d$ were introduced by Collins [15] in a decision method for the first-order theory of real closed fields; adjacency computation was added by Arnon et al. [3]. Their algorithms compute purely symbolically, which is slow. More recent work[2] [16] [11] [37] achieves significant speedups with numerical computations, esp. for root finding, but they have to fall back to symbolic computation in unfavorable situations. In contrast, our approach uses numerical adaptive-precision root finding based on the Bitstream Descartes method [21] uniformly – there is no fall-back to a symbolic method in hard cases.

Regarding the geometry of algebraic curves, we refer to the textbooks by Gibson [27] and Walker [38].

Contents We describe the sweep-line algorithm for curves in Section 2 and show how it reduces arrangement computation for segments to geometric analyses of curves that support segments and of pairs of such curves. Section 3 summarizes the analysis of one curve, a full treatment is given in [20]. Section 4, the main contribution of this work, extends these techniques to pairs of curves. We report on our implementation and on runtime measurements in Section 5, and present our conclusions in Section 6.

2 Sweep-line algorithm



Given a set of algebraic curve segments, we compute its arrangement using a sweep-line algorithm. The algorithm operates on x-coordinates, points on curves, and segments of curves. Every relevant x-coordinate α occurs as a pair (p, I) of a square-free polynomial $p(x) \in \mathbb{Z}[x]$ and a bounded interval $I \subseteq \mathbb{R}$ such that $x = \alpha$ is the unique root of $p(x) = 0$ in I. We call I an *isolating interval* for α. Every point $P = (\alpha, \beta)$ considered lies on some *supporting curve* f and is represented as a triple (α, f, i) such that $y = \beta$ is the ith real root of $f(\alpha, y) = 0$. We call i the *arc number* of P. As segments of a curve f, we allow only x-monotone segments whose x-range does not extend across an f-event (see below). We call such a segment s a *sweepable segment*. With the possible exception of the endpoints P and Q, all points on s have the same arc number i on the supporting curve f. We represent s as (P, Q, f, i). Arbitrary segments of a curve f can be split into sweepable segments using the curve analysis of f. Our algorithm also allows vertical line segments and segments with endpoints "at infinity", which occur naturally when an entire algebraic curve is broken into sweepable segments, and handles them akin to [8, §3.1]. Here, we restrict our presentation to bounded non-vertical segments for brevity.

Conceptually, the sweep-line algorithm sweeps the plane with a vertical line ℓ from left to right and records features of the arrangement as ℓ passes over *event points*, that is, points at which segments start, intersect, or end [5] [34, §10.7]. To do so, the algorithm processes event points in xy-lexicographic order while maintaining the following invariant: Left of ℓ, the arrangement has already been constructed. On ℓ, we see a sorted sequence of intersections with segments, which is stored in the *Y-structure*. Right of ℓ lies the unexplored part of the plane. The *X-structure* stores, in xy-lexicographic order, some of the future event points; at least those at which segments start, end, or segments adjacent on the sweep line intersect.

The sweep line is advanced across an event as follows.

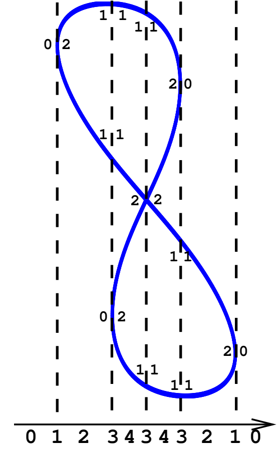
- Find the segments involved in the event by locating the event point in the Y-structure, exploiting its order. *(Requires: Comparison of y-coordinates of point and segment at common x-coordinate.)*
- Remove ending segments from Y-structure. *(Requires: Comparison of event point and endpoints.)*
- Reorder remaining intersecting segments according to the situation right of the event (see below). *(Requires: Intersection multiplicity of segments.)*

[2] We refer to [20] for a detailed discussion of previous work on computing a cad with adjacencies of $\mathbb{R}^2$ for just one polynomial.

- Add starting segments to the sorted Y-structure. *(Requires: Comparison of event point and endpoints. — Comparison of segment y-order right of common point.)*
- Add intersections of newly adjacent segments to the sorted X-structure. *(Requires: Computation of intersection points. — Lexicographic comparison of event points.)*

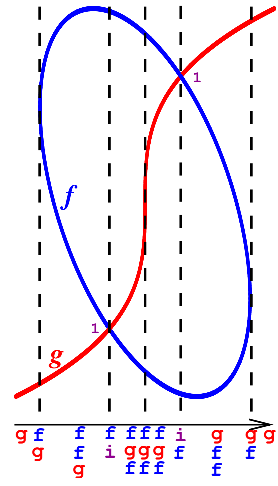
The sweepable segments we consider possess an analytic implicit function $y = \varphi(x)$ around each point in their interior. Our way of finding the intersection points of segments adjacent in the Y-structure produces their intersection multiplicity as a by-product, indicating up to which degree the power series expansions of their implicit functions agree. This is enough to reorder them (see [7] [22, §5]) in linear time (see [10]). As a consequence, the complexity analysis of the sweep-line method in terms of basic geometric operations remains unchanged from the straight-line case.

All comparisons above reduce to comparing x-coordinates and to comparing points on at most two supporting curves over a common x-coordinate. Hence these comparisons as well as computing intersections reduce to a geometric analysis of a single curve or a pair of curves in a cad-like fashion; cf. [22, §5] [6].



Single curves Let C be an algebraic curve with defining equation $f(x, y) = 0$. In what follows, we will identify C with the polynomial f, talking about the curve f. We assume for brevity that f is square free (i.e., $h^2 \mid f \Rightarrow \deg(h) = 0$) and that f contains no vertical line component (i.e., $\forall a \in \mathbb{C}\colon (x - a) \nmid f$).

We define the *f-events* to be the x-coordinates where f has a critical point (singularity or regular point with vertical tangent line) or a vertical asymptote. For each event α, an *f-stack* is created that reflects the geometry of f at α. This stack stores the number of intersections on the curve with the line $x = \alpha$ in increasing order. For each such intersection point, it stores how many arcs of the curve enter that point from left side, and how many are leaving to the right side. Also, information about vertical asymptotes approaching $x = \alpha$ is stored. For intervals between consecutive events, f-stacks are created as well. See Section 3 for more.



Curve pairs We now consider two curves f and g such that $h := \gcd(f, g)$ is constant. (This is no restriction; if h is non-constant, we can re-define segments on f in terms of f/h or h, same for g.)

The x-coordinates of intersections of f and g together with f-events and g-events are called fg-events. Note that the number and relative position of arcs of f and g on a vertical line can only change at fg-events. For each fg-event and each interval between fg-events, we store an *fg-stack*, which records the vertical order of f-arcs and g-arcs. In addition, we store the intersection multiplicities of intersection points (α, β) at fg-events α which are neither f- nor g-events.

Computing an fg-stack over an interval I is easy: Just pick a rational or binary number $a \in I$, then find and sort the real roots y of $f(a, y)$ and $g(a, y)$, which are all simple and distinct. However, fg-stack construction over an event α is hard, because $f(\alpha, y)$ and $g(\alpha, y)$ may have common roots, multiple roots, and not necessarily rational coefficients. Section 4 describes our new method for this.

3 Curve analysis

Given $f \in \mathbb{Z}[x, y]$, we have to compute the events (x-coordinates of critical points and asymptotes), and to build the stacks at each event (number of points and number of incident arcs on either side at each point). As it will be useful in Section 4, we also determine isolating intervals for the y-coordinate of the stack points (we call them *f-intervals*) and an upper bound on their multiplicities as roots of $f_\alpha := f(\alpha, y)$. To do so, we use the method presented in [20]. For completeness, we summarize it briefly. Let us begin with the generic case.

Definition 3.1 *We call a curve $f \in \mathbb{Z}[x, y]$* generic*, if*

*(G1) f is y-*regular*, i.e., its leading term as a polynomial in y does not involve x; and*

(G2) for each $\alpha \in \mathbb{R}$, the polynomial $f_\alpha \in \mathbb{R}[y]$ has at most one multiple root in $\mathbb{C}$.

We first give an algorithm `generic_analyze` that either performs the curve analysis of f or reports a failure. If f is generic, then `generic_analyze` does not report a failure. Later on, we give the full algorithm that uses `generic_analyze` to perform the curve analysis without genericity conditions.

The algorithm `generic_analyze` The algorithm consists of three steps: It finds the events of f; then computes the points on the stacks over each event α by isolating the real roots of f_α; finally, it computes the number of arcs on either side for each point. The most interesting part is root isolation of f_α for an event α. The polynomial f_α has algebraic coefficients and multiple roots; both complicates root finding.

We use the Bitstream Descartes method [21], which computes isolating intervals for the exact real roots of a square free polynomial, using only approximations of the coefficients. It recursively subdivides an initial interval containing all roots into intervals (c, d) for which the Descartes test indicates that they contain $r = 0$ or $r = 1$ real roots. This test is given by Descartes' Rule of Signs (see [19] for history).

Theorem 3.2 (Descartes' Rule, Bernstein form) *Consider a polynomial $g(t) = \sum b_i B_i^n[c, d](t)$ expressed in the Bernstein basis $B_i^n[c, d](t) = \binom{n}{i} \frac{(t-c)^i (d-t)^{n-i}}{(d-c)^n}$ for an interval $[c, d]$. Let v be the number of sign variations in the coefficient sequence $(b_0, \ldots, b_n)$. Let r be the number of roots of g in (c, d), counted with multiplicities. Then $v \geq r$ and $v \equiv r \pmod 2$. In particular, $v = r$ if $v \leq 1$.*

In the presence of an r-fold real root β with $r > 1$, the Bitstream Descartes method would run indefinitely on f_α, trying to separate the r roots at β. To overcome this, we precompute some exact information about the polynomial f_α, namely k, the degree of $\gcd(f_\alpha, f'_\alpha)$, and m, the number of *distinct* real roots of f_α, using the *Sturm-Habicht sequence* of f [28] (or equivalently the *signed subresultant sequence* [4, §4]).

The *m-k-Bitstream-Descartes method* proceeds as follows: *Run the Bitstream Descartes method on f_α, exploring subintervals in breadth-first order, and stop in two cases: First, if $m - 1$ simple roots are found and only one further interval counts more than zero in the Descartes test, then report this last interval as the mth isolating interval. Second, if the Descartes test counts k or less in all intervals, report a failure.*

If f is generic, (G2) implies that the m-k-Bitstream-Descartes method and thus `generic_analyze` do not fail. The second stopping criterion ensures termination even if there are no $m - 1$ simple roots, cf. [19].

On success, the f-intervals at α have been computed. Descartes' rule gives an upper bound for the multiplicity of the root inside each interval. The $m - 1$ simple roots of f_α are adjacent to exactly one arc on each side, all remaining arcs run into the mth isolating interval. The y-coordinates of points over α can be approximated to any precision by refining the f-intervals further. The Bitstream Descartes method provides a framework that controls the numerical precision necessary for the coefficients of f_α and thus for α when we refine f-intervals. Beyond this, there is no need for controlling numerical precision in our algorithm.

Curve analysis in all cases If f is not generic, `generic_analyze` may fail. We can escape from non-genericity by passing to the *sheared curve*

$$\mathrm{Sh}_s f := f(x + sy, y)$$

with all but finitely many choices of a *shear factor* $s \in \mathbb{Z}$, see [4, Prop. 11.23].

The full curve analysis works as follows: *Apply* `generic_analyze` *on f. On success, return. Otherwise, choose a shear factor s at random and compute the sheared curve $\mathrm{Sh}_s f$. Apply* `generic_analyze` *on $\mathrm{Sh}_s f$. If it fails, choose another shear factor s and retry. On success, apply* `transform_analyze`, *using the stacks of $\mathrm{Sh}_s f$.*

The procedure `transform_analyze` takes a curve analysis of $\mathrm{Sh}_s f$ as input and computes the stacks of f without any genericity condition. This is not a trivial operation, because the notion of a critical point depends on the coordinate system. However, we have to omit it for brevity and refer to [20, §6] instead.

4 Curve pair analysis

We consider two algebraic curves f and g for which $\gcd(f, g)$ is constant. The goal is to determine the sequence of fg-events and an fg-stack at each event. The fg-events are the x-coordinates of intersections

of f and g together with the f-events and g-events. An fg-stack is a sequence of elements `f`, `g`, and `i` that represents the vertical order of points of f, points of g, and intersection points of f and g on the vertical line at the event. Additionally, an fg-stack at an fg-event that is neither f- nor g-critical records the intersection multiplicity of each intersection point. Generally, these intersection multiplicities arise as by-products of our method, without extra effort. Only in special cases does their determination necessitate a change into generic coordinates. For brevity, we do not go into details here of how they are obtained.

Both the fg-events and their stacks are computed in a "lazy" fashion: When the sweep-line algorithm invokes a geometric operation involving the curve pair (f, g) for the first time, the fg-events are computed. Also, the computation of each stack is delayed until it is queried the for the first time by the sweep-line algorithm. This avoids unnecessary computations, since many curve pairs might not be considered at all, and only few stacks might be needed for other curve pairs.

4.1 Finding events

The f- and g-events are already known from the f- and g-stacks. The x-coordinates of intersections are roots of the resultant polynomial $R := \mathrm{res}_y(f, g) \in \mathbb{Z}[x]$. Since $\gcd(f, g)$ is constant, $R \neq 0$.

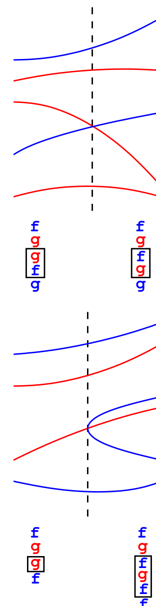
The algorithm proceeds as follows: *Compute* $R = \mathrm{res}_y(f, g)$. *Factor* $R = R_1^1 \cdots R_t^t$ *such that* $R_1, \ldots, R_t \in \mathbb{Z}[x]$ *are square free* [26, §8.1]. *(R_i has exactly the roots of R with multiplicity i.) Isolate the roots of every R_i with the Descartes method. Merge all of them and the f-events and g-events into one sorted sequence. For each x-coordinate α in the sequence, store its multiplicity as root of R (which is 0 if $R(\alpha) \neq 0$).*

4.2 Stack creation

From now on, we fix one fg-event α, and specify our algorithm `stack_create` to compute its fg-stack. We use three different methods to do so: The method `simple_create` handles resultant roots α of multiplicity 0 or 1; in these cases, the stack can be computed mostly from the position of arcs of f and g approaching $x = \alpha$ from the left and right hand side. In generic cases, this method handles all simple intersections. For roots α of higher multiplicity, the method `generic_create` builds the stack by precomputing some symbolic information and considering the f-stack and the g-stack at α; at least, it *tries* to do so but fails in the presence of covertical intersections. If failed, the curve pair is sheared (by shearing both curves), analyzed, and the information from the sheared coordinate system is used to build the stack in original coordinates with the procedure `shear_create`.

Here is how the algorithm `stack_create` proceeds: *Let μ be the multiplicity of α in R. If $\mu \in \{0, 1\}$, call* `simple_create` *and return. If an analysis in a sheared system has already been computed, call* `shear_create` *and return. Otherwise, call* `generic_create`. *On success, return. On failure, perform an analysis in a sheared system and run* `shear_create`.

We explain the three subalgorithms now.



Simple stack creation Here, α is either a simple root of R or no root, so the geometry of the curve pair around α is particularly easy: There is at most one intersection point p. If p exists, it is a regular point of f and g, and the intersection is transversal, i.e., f and g have different tangents at p. In the general case, neither of the two tangents is vertical, and we see one transposition of an f-arc and a g-arc when we compare the fg-stacks over the intervals incident to α. In the special case that one tangent at p, say the tangent to f, is vertical, we either see a transposition as in the previous case (if f changes sides with its vertical tangent, which happens for an inflection point), or p has two adjacent arcs on one side and none on the other (i.e., p is x-extreme), and the intersecting g-arc runs between the two f-arcs.

In any case, there is a specific pattern of change in the arc sequence, which cannot occur in the absence of an intersection. Looking for these patterns is an old trick. We give it a new twist by exploiting these patterns even if critical points exist at this event. It is the task of `simple_create` to create the fg-stack at α by spotting these patterns in the fg-stacks at the intervals incident to α on the left and on the right. We call them the L-stack and the R-stack. To do so, it goes simultaneously through the f-stack and g-stack at α from bottom

upwards and has to decide for the next f-point and the next g-point: Do they coincide in an intersection, or if not, does the f-point or the g-point come first? Points that have already been dealt with are marked.

In detail, `simple_create` proceeds as follows:

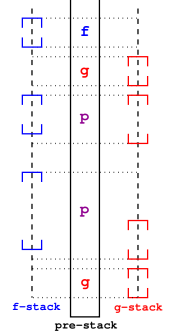
Initialization: *Fetch L-stack and R-stack. Unmark all arcs on the stacks. Create the f-stack and g-stack at α, if not existing yet. Unmark all points on the stacks. Create an empty fg-stack for α. Mark all arcs on L-stack and R-stack that are asymptotic at $x = \alpha$.*

Repeat until all points are marked: *Consider the lowest unmarked points at the f- and g-stack. Find the arcs at the L-stack and R-stack connected to them. If they show an intersection pattern described above, then add* `i` *(with multiplicity one) to the fg-stack, mark the two points and four arcs considered, and proceed with the next iteration. If the points both have an incident arc in the L-stack or both in the R-stack, use that to decide which one is next; otherwise refine the f-interval and g-interval to disjointness to decide which is next. If an f-point is next, add* `f`*, else add* `g` *to the fg-stack at α. Mark the next point and its incident arcs, and proceed with the next iteration.*

Once an intersection is detected, the test for the intersection pattern can be skipped, because a second intersection is impossible. Also, it is not always necessary to create the f- or g-stack at α. For instance, if α is not f-critical, one can start with the f-stack of the interval containing α (which contains the same combinatorial information and has been already computed in the curve analysis). Only if some f-interval is needed during the algorithm, one has to pass to the actual f-stack at α.

Generic stack creation We explain the subalgorithm `generic_create` next. It is incomplete in that it may succeed or fail. If it succeeds, it computes the stack of the curve pair at α. It is guaranteed to succeed at least for those cases in which $\deg_y(f_\alpha) = \deg_y(f)$ and $\deg_y(g_\alpha) = \deg_y(g)$, and the polynomials f_α and g_α have at most one common root in $\mathbb{C}$ (compare Lemma 4.3). In other cases, it may (but need not) fail.

As a first step, we compute a *pre-stack*, which contains a preliminary ordering of the points f and g at α. It is a sequence of entries `f`, `g` and `p`, where `p` stands for *pair*. Such a pair either corresponds to an `i`-entry of the actual stack (an intersection), or to a (`f`,`g`)-subsequence, or to a (`g`,`f`)-subsequence of the actual stack.



The pre-stack is computed as follows: *Create the f-stack and g-stack at α, if not existing yet. Refine each f-interval until it overlaps with at most one g-interval, and vice versa. As result, each f-interval and each g-interval is either disjoint from any other f- or g-interval – we call such intervals* singles *–, or it overlaps with exactly one interval from the other curve – we say that these two intervals form a* pair*. Sort the pairs and singles in increasing order, writing* `f` *for a single of f,* `g` *for a single of g, and* `p` *for a pair.*

We say that a pair *splits* if the corresponding f- and g-intervals become disjoint due to further refinement. If so, we replace the `p` entry by the correct subsequence in the pre-stack. The idea of `generic_create` is: Try to split all pairs in the pre-stack of α except one. Then verify an intersection at the last pair. If the presence of more than one intersection at α is sure, report a failure.

The key source of efficiency lies in the cutback of algebraic calculations. However, it is unavoidable to compute some exact algebraic information to ensure exactness. *Subresultants* [4, §4] [43, §3] [25] are our tool for this. We denote by $\mathrm{Sres}_i(f,g) \in \mathbb{Z}[x,y]$ the ith subresultant of f and g with respect to y, and by $\mathrm{sres}_{i,j}(f,g) \in \mathbb{Z}[x]$ the coefficient of y^j in $\mathrm{Sres}_i(f,g)$.

Lemma 4.1 *Let $f,g \in \mathbb{R}[x,y]$ and $\alpha \in \mathbb{R}$. If $\deg_y(f_\alpha) = \deg_y(f)$ and $\deg_y(g_\alpha) = \deg_y(g)$, then*

$$\deg(\gcd(f_\alpha, g_\alpha)) = \min\{k \geq 0 \mid \mathrm{sres}_{k,k}(f,g)(\alpha) \neq 0\}.$$

With Lemma 4.1, the integer $k = \deg(\gcd(f_\alpha, g_\alpha))$ is efficiently computable using the subresultant sequence. We use this number in the analysis of the stack:

Lemma 4.2 *With f, g, α as above, let $k = \deg(\gcd(f_\alpha, g_\alpha))$. If there exists a unique intersection point (α, β), $\beta \in \mathbb{C}$, of f and g at α, then $\beta \in \mathbb{R}$, and β is a root of multiplicity at least k of both f_α and g_α, and*

$$\beta = -\frac{\mathrm{sres}_{k,k-1}(f,g)(\alpha)}{\mathrm{sres}_{k,k}(f,g)(\alpha)}.$$

Recall that from the curve analysis, we know upper bounds for the multiplicity of each f- and g-interval, considered as roots of f_α and g_α. Moreover, [19] shows that this upper bound drops to the actual multiplicity when the isolating interval is sufficiently refined. For a pair, we define its multiplicity to be the minimum of the multiplicities of the contained f- and g-interval.

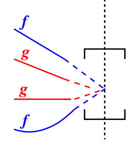
`generic_create` proceeds as follows:

Initialization: *Check that* $\deg_y f_\alpha = \deg_y f$ *and* $\deg_y g_\alpha = \deg_y g$*, report a failure otherwise. Compute the subresultant sequence of* f *and* g*, if not yet computed. Use Lemma 4.1 to compute* $k = \deg(\gcd(f_\alpha, g_\alpha))$.

Iterate over the pairs: *Refine the* f*- and* g*-interval for each pair until one of the following two conditions holds: First, if all remaining pairs show a multiplicity less than* k*; report a failure. Second, if only one pair remains, stop. (We call this last interval the* candidate.*)*

Similar to the analysis of the m-k-Descartes algorithm [20], one can show that this method terminates, and if it fails, this implies the presence of covertical (although not necessarily real) intersections at α.

The candidate is not necessarily an intersection, since f_α and g_α might intersect only at imaginary points. Therefore, the candidate must be checked further. We prefer to find cheap evidence for an intersection. First we check the parity of the already-computed k or μ. If one of these number is odd, the candidate must be an intersection (since each common imaginary root appears with its complex conjugate, so they push up both quantities by an even number). We remark that a unique intersection at α in a point that is not critical on both f and g makes $k = 1$, so it is handled relatively cheaply, even if the intersection is tangential, in contrast to the costly solutions of [22, §4.2.4] [42].



In other cases, we can spot an intersection by looking at the fg-stacks of the left and right neighboring intervals: If on one side, an arc of g is "sandwiched" between two arcs of f (or vice versa) that run into the candidate, then the curves must intersect.

`generic_create` treats the candidate as follows: *If* k *or* μ *is odd, replace the candidate with an intersection. Otherwise, examine the incident arcs of the left- and right-neighboring* fg*-stacks. If one arc of the one curve is between two arcs of the other curve, replace the candidate with an intersection. Otherwise, exploit the second part of Lemma 4.2: set* $\rho(\alpha) := -\frac{\mathrm{sres}_{k,k-1}(f,g)(\alpha)}{\mathrm{sres}_{k,k}(f,g)(\alpha)}$*. Check symbolically whether* $f(\alpha, \rho(\alpha)) = 0 = g(\alpha, \rho(\alpha))$*. If this is the case,* $(\alpha, \rho(\alpha)) \in \mathbb{R}^2$ *is an intersection, and so it must be the candidate pair; replace the candidate with an intersection. If this is not the case, report a failure.*

When does `generic_create` succeed? In analogy to the case of one curve, one can show:

Lemma 4.3 *If* $f \cdot g$ *is generic according to Definition 3.1,* `generic_create` *is successful for each event.*

Note that also for non-generic curve pairs, we might not observe a failure: `generic_create` might still work in presence of covertical (imaginary) intersections. Also, `generic_create` just might not be called for an x-coordinate that presents a degeneracy of the curve pair.

Stack creation with a shear In case the generic method in the original coordinate system failed for some α, `stack_create` makes a "detour" over the sheared curve pair to obtain enough information for building the fg-stack. The sheared analysis is performed as follows: *Choose some* $s \in \mathbb{Z}$ *at random. Compute* $\mathrm{Sh}_s f$ *and* $\mathrm{Sh}_s g$ *and analyze the single curves. Create all stacks of the sheared curve pair, applying* `simple_create` *if possible, and* `generic_create` *otherwise. If the latter fails, choose a new shear factor* s *and retry.*

This methods terminates since, eventually, the pair $(\mathrm{Sh}_s f, \mathrm{Sh}_s g)$ becomes generic, and `generic_create` succeeds by Lemma 4.3. Having the sheared curve pair analyzed, a so-called *intersect-info-object* is computed. It contains for each fg-event of the *original* curve pair the number of distinct intersection points at the event, and intervals indicating at which y-coordinates these intersections take place (approximately).

How is this information obtained from the sheared curve pair? Intersections of f and g are in bijective correspondence to intersections of $\mathrm{Sh} f$ and $\mathrm{Sh} g$. So the number of intersections of f and g at α can be obtained by explicitly shearing back the intersections of $\mathrm{Sh} f$ and $\mathrm{Sh} g$ (i.e., applying the inverse shear), and counting how many have x-coordinate α. To approximate the y-coordinate of an original intersection, note that the corresponding sheared intersection is approximated by a $\mathrm{Sh} f$-interval (and also by a $\mathrm{Sh} g$-interval). When shearing, the y-coordinate of a point does not change. Thus, one can simply store (a pointer to) the $\mathrm{Sh} f$-interval in the intersect-info-object to represent the y-coordinate of the original intersection as well.

To construct the intersect-info-object, the algorithm proceeds as follows: *Iterate over the intersection points of the sheared curve pair. Let* p^* *be the current intersection point. This point corresponds to a unique*

n		AlciX	cad2d	n		AlciX	cad2d
	a	0.106	0.393		a	3.059	4.656
6	b	0.135	0.404	12	b	3.117	4.322
	c	0.123	0.401		c	3.455	5.091
	a	0.747	1.073		a	8.569	12.814
9	b	0.793	1.417	15	b	7.559	12.250
	c	0.821	1.128		c	9.733	15.503

Table 1: Timings (in sec) for pairs of random curves with 50-bit coefficients and degree n.

intersection point p of the original curve f and g, and p's x-coordinate must be an fg-event. Compute an interval approximation of the x-coordinate of p, and refine it until it overlaps with exactly one fg-event γ. Obtain this approximation by considering both the x- and y-coordinates of p^* as interval approximations, yielding a rectangle containing p^*, and explicitly shear back the (rational) corners of this rectangle. The event so obtained is the x-coordinate γ of p; update γ's intersect-info-object by adding a new intersection with y-coordinate represented by the $\mathrm{Sh}f$*-interval of p^*.*

Knowing both a pre-stack and an intersect-info-object for α, creating the final stack is relatively straightforward: the pre-stack gives the ordering of the stack points, only that some non-intersecting points might be paired. With the intersect-info-object, these "wrong" pairs are easily detected.

Here is the procedure `shear_create`:

Initialization: *Create the pre-stack, if not existing yet. Create an intersect-info-object, if not existing yet.*

Iterate over the intersect-info-object for α: *Let I denote the current* $\mathrm{Sh}f$*-interval that approximates the y-coordinate of an intersection. Refine I until it overlaps with a unique pair; replace this pair with an intersection.*

Postprocessing: *Split each remaining pair on the pre-stack by repeatedly refining its intervals.*

By passing to a sheared system, the algorithm loses its locality: although only the stack at α is of interest, all intersection points of the curve pair are considered. Also, the analysis of the sheared system is usually more expensive since shearing increases the bitsize of a curve's defining equation, and it involves the complete curve analysis of $\mathrm{Sh}f$ and $\mathrm{Sh}g$. For these reasons, `stack_create` tries to avoid a shear and to analyze the curve pair directly using `generic_create` if possible. On the other hand, generic coordinates help a lot in reducing the amount of symbolic computations, and our approach nicely encapsulates coordinate changes inside the curve and curve pair analysis. It does not burden the user with the choice of one global generic coordinate system, as earlier approaches did [22, §4.4.3] [42].

5 Implementation

The algorithm presented has been implemented in the AlciX library as part of EXACUS[3] [6], a collection of C++ libraries for exact computations on curves and surfaces. AlciX mainly uses the EXACUS library NumeriX for computations with algebraic objects. The underlying number types can be chosen from LEDA [34] or CORE/GMP[4]. Also, for specific computations, boost[5] (interval arithmetic), NTL[6] (univariate modular gcd) and Maple (modular bivariate resultant) are in use.

Our curve and curve pair analyses are implemented as class templates that satisfy the EXACUS concepts AlgebraicCurve_2 and AlgebraicCurvePair_2. The SweepX library of EXACUS implements *Generic Algebraic Points and Segments* (GAPS) on top of them; these can be used with the LEDA-based sweep-line algorithm of SweepX and also with the sweep-line algorithm of the CGAL[7] Arrangement package [40]. In either case, the result is a doubly-connected edge list (DCEL) representing the arrangement. We have implemented

[3] `http://www.mpi-inf.mpg.de/projects/EXACUS/`
[4] `http://cs.nyu.edu/exact/core_pages/`
[5] `http://www.boost.org/`
[6] `http://www.shoup.net/ntl/`
[7] `http://www.cgal.org/`

c		AlciX	cad2d	c		AlciX	cad2d
	a	0.054	0.394		a	2.426	5.165
2^6	b	0.149	0.456	2^{10}	b	2.072	13.975
	c	0.113	0.457		c	2.432	5.051
	a	0.470	2.230		a	20.510	62.657
2^8	b	0.390	1.169	2^{12}	b	19.273	64.957
	c	0.378	1.734		c	19.098	64.602

Table 2: Timings (in sec) for pairs of random curves with c-bit coefficients and degree 6.

n, m		AlciX	cad2d	n, m		AlciX	cad2d
	a	0.216	0.638		a	7.903	191.710
$3, 2$	b	0.090	0.431	$4, 3$	b	8.268	215.337
	c	0.181	Error		c	8.162	199.597
	a	0.596	7.012		a	36.646	Prime*
$3, 3$	b	0.533	7.464	$4, 4$	b	39.589	Prime*
	c	0.630	5.780		c	21.437	552.641

Table 3: Timings (in sec) for projections from random surfaces with 8-bit coefficients and degrees n and m.
* Prime: cad2d halted because "Prime list exhausted".

both options; they showed a similar performance. The running times in the tables are for the creation of the LEDA-based arrangement.

Our implementation allows to exchange subalgorithms and data structures easily. Our experiments showed that there is no unique best configuration for *all* instances; sophisticated methods optimized for higher-degree instances seem to incur a considerable overhead that deteriorates the performance for low-degree arrangements. We ran experiments on a Linux PC with a Pentium-4 processor at 2.80 GHz and 1 GB of RAM.

We have compared AlciX with CubiX [22], another EXACUS library, providing curve and curve pair analyses for algebraic curves of degree up to three. CubiX' default configuration has been chosen for both libraries: LEDA number types are used, isolating intervals are refined by bisection, no modular gcd computation is used. This turned out to be optimal also for AlciX for cubic curves. We have tried benchmark instances from [22, §7] for comparison. AlciX is slower for the random (generic) arrangements by a factor 1.5, and equally fast (deviation $< 10\%$) for the degenerate instances (which still contain many simple intersections). The fact that CubiX' advantage shrinks when tangential intersections come into play indicates that they are handled better by AlciX. Beyond that, it is not too surprising that the numerous tricks of CubiX for the special case of degree 3 incur a moderate speedup over the general method of AlciX.

For algebraic curves in full generality, there is no exact implementation available from the Computational Geometry community. So we have compared AlciX' performance with software from the Symbolic Computation community, namely Brown's cad2d, an optimized version of `QEPCAD-B`[8] 1.46 for computing cylindrical algebraic decompositions in the plane. It uses floating point methods in the lifting step to speed up calculations in favorable situations [11].

For AlciX, we have observed the best performance with the following setting: CORE/GMP number types and the modular gcd algorithm from NTL (V. 5.4) are used, isolating intervals are refined with Abbott's Quadratic Interval Refinement [1] (the same setting was used in [20]). To compute resultants, we use the modular resultant algorithm from Maple 10 (via the OpenMaple interface) for curves of degree 11 and more. For smaller degree resultants and for subresultants, we implemented Ducos' algorithm [18]. It turned out that subresultant computation is the algorithm's main bottleneck for curves of high degree.

We have compared two kinds of input: For Tables 1 and 2, we created pairs of curves of degree n with random c-bit integer coefficients, three examples for each choice of n and c. For Table 1, we fixed $c = 50$

[8] `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`

c		AlciX	cad2d	c		AlciX	cad2d
	a	0.204	1.425		a	0.536	15.062
16	b	0.190	1.505	48	b	0.520	15.611
	c	0.127	0.597		c	0.536	15.107
	a	0.489	6.021		a	0.748	34.926
32	b	0.385	5.863	64	b	0.828	32.964
	c	0.131	1.255		c	0.678	33.059

Table 4: Timings (in sec) for projections from random surfaces with c-bit coefficients and degrees 3 and 2.

N		AlciX	nodes	edges	N		AlciX	nodes	edges
	a	12.635	1300	2284		a	110.396	9684	18402
25	b	10.792	1058	1858	75	b	110.434	9978	18988
	c	13.488	1418	2502		c	124.191	11354	21758
	a	53.368	5128	9578		a	204.483	18296	35354
50	b	50.052	4804	8980	100	b	237.359	21694	41996
	c	47.506	4418	8250		c	223.088	20332	39338

Table 5: Timings (in sec) for N random curves of degree 6 and 50-bit coefficients.

and varied the degree. For Table 2, we fixed $n = 6$ and varied the coefficient bitsize. We expect a good performance of cad2d in these examples since it profits from its improved lifting in each instance.

For Tables 3 and 4, we created two surfaces F, G of degrees n and m with random c-bit coefficients, and have computed the arrangement of $\mathrm{res}_z(F, F_z)$, $\mathrm{res}_z(G, G_z)$, and $\mathrm{res}_z(F, G)$. Geometrically, these are the projections of the two silhouettes and the intersection curve of F and G. Generically, they have the degrees $n(n-1)$, $m(m-1)$, nm, resp., contain singularities and intersect tangentially. For Table 3, we fixed $c = 8$ and varied the surface degrees. For Table 4, we fixed the degrees $n = 3$, $m = 2$ and varied c.

AlciX performs slightly better than cad2d for random curve pairs (Tables 1 and 2). Note that both algorithms can cope with rather large coefficients. The more remarkable improvement, however, is visible for the degenerate arrangements of three curves (Tables 3 and 4): AlciX outperforms cad2d, since its optimized numerical stack construction applies in degenerate cases, too.

We have also investigated arrangements for more input segments: we fixed degree 6 and 50-bit coefficients, and created up to 100 algebraic curves. Table 5 shows the running times and the sizes of the resulting arrangements. Note that the running time is roughly proportional to the number of edges in the arrangement. This matches the linear-logarithmic output sensitive complexity bound for the sweep-line algorithm.

Finally, we have investigated the effect of shearing: We interpolate otherwise random curves of degree 7 through the covertical points $(0, 0)$ and $(0, 1)$ so that a shear must be applied for each curve pair. We observed a slowdown by a factor of roughly 4, compared to completely random curves of same size.

6 Conclusion

We have presented an algorithm for computing the exact planar arrangement induced by any set of segments of algebraic curves. For brevity, we had to skip the treatment of some special cases in this extended abstract, but they do not add substantial complications. The crucial new tool of our algorithm, compared to previous EGC approaches for arrangement computation, is exact yet efficient root isolation at the algebraic x-coordinate of an intersection by a careful combination of symbolic and adaptive-precision numeric techniques. For example, tangential intersections in regular points are not a hard case anymore, and all singular points can be treated in a systematic and uniform way. Sometimes, we have to change coordinates, but that is encapsulated in the geometric analyses and does not limit the user's choice of coordinates.

The experiments have indicated that our algorithm is efficient, both in handling many segments and in

handling degenerate geometric situations; at least as long as the degrees of individual curves do not hit the fundamental barrier inherent in (sub)resultant computation.

Acknowledgements We have enjoyed many useful discussions with Nicola Wolpert. Eric Berberich has been a great source of help during the implementation of AlciX and commented on a draft of this paper.

References

[1] J. Abbott: "Quadratic Interval Refinement for Real Roots". URL `http://www.dima.unige.it/~abbott/`. Poster presented at the 2006 Int. Symp. on Symb. and Alg. Comp. (ISSAC 2006).

[2] P. K. Agarwal, M. Sharir: "Arrangements and Their Applications". In: J.-R. Sack, J. Urrutia (eds.) *Handbook of Computational Geometry*, 49–119. Elsevier, 2000.

[3] D. S. Arnon, G. E. Collins, S. McCallum: "Cylindrical Algebraic Decomposition I+II". *SIAM J. on Computing* **13** (1984) 865–889. Reprinted in [14], pp. 136–165.

[4] S. Basu, R. Pollack, M.-F. Roy: *Algorithms in Real Algebraic Geometry*, *Algorithms and Computation in Mathematics*, vol. 10. Springer, 2nd edn., 2006.

[5] J. L. Bentley, T. A. Ottmann: "Algorithms for Reporting and Counting Geometric Intersections". *IEEE Transactions on Computers* **C-28** (1979) 643–647.

[6] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, N. Wolpert: "EXACUS: Efficient and exact algorithms for curves and surfaces". In: *Proc. of the 13th Annual European Symposium on Algorithms (ESA 2005)*, *LNCS*, vol. 3669. Springer, 2005 155–166.

[7] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, E. Schömer: "A Computational Basis for Conic Arcs and Boolean Operations on Conic Polygons". In: *Proc. of the 10th Annual European Symposium on Algorithms (ESA 2002)*, *LNCS*, vol. 2461. Springer, 2002 174–186.

[8] E. Berberich, E. Fogel, D. Halperin, K. Mehlhorn, R. Wein: "Sweeping and Maintaining Two-Dimensional Arrangements on Surfaces: A First Step". In: *Proc. of the 15th Annual European Symposium on Algorithms (ESA 2007)*, 2007 To appear in Springer LNCS.

[9] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, N. Wolpert: "An Exact, Complete and Efficient Implementation for Computing Planar Maps of Quadric Intersection Curves". In: *Proc. of the 21st Annual Symposium on Computational Geometry (SCG 2005)*, 2005 99–106.

[10] E. Berberich, L. Kettner: *Linear-Time Reordering in a Sweep-line Algorithm for Algebraic Curves Intersecting in a Common Point.* Research Report MPI-I-2007-1-001, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, 2007. URL `http://domino.mpi-inf.mpg.de/internet/reports.nsf/NumberView/2007-1-001`.

[11] C. W. Brown: "Constructing Cylindrical Algebraic Decompositions of the Plane Quickly", 2002. URL `http://www.cs.usna.edu/~wcbrown/`. Unpublished.

[12] B. Buchberger, G. E. Collins, R. Loos (eds.): *Computer Algebra: Symbolic and Algebraic Computation.* Springer, 2nd edn., 1983.

[13] J. Caravantes, L. Gonzalez-Vega: "Computing the Topology of an Arrangement of Quartics". In: R. R. Martin, M. A. Sabin, J. R. Winkler (eds.) *IMA Conference on the Mathematics of Surfaces*, *LNCS*, vol. 4647. Springer, 2007 104–120.

[14] B. F. Caviness, J. R. Johnson (eds.): *Quantifier Elimination and Cylindrical Algebraic Decomposition, Texts and Monographs in Symbolic Computation.* Springer, 1998.

[15] G. E. Collins: "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition". In: *Second GI Conference on Automata Theory and Formal Languages*, *LNCS*, vol. 33, 1975 134–183. Reprinted in [14], pp. 85–121.

[16] G. E. Collins, J. R. Johnson, W. Krandick: "Interval Arithmetic in Cylindrical Algebraic Decomposition". *J. of Symbolic Computation* **34** (2002) 145–157.

[17] O. Devillers, A. Fronville, B. Mourrain, M. Teillaud: "Algebraic methods and arithmetic filtering for exact predicates on circle arcs". *Computational Geom.* **22** (2002) 119–142.

[18] L. Ducos: "Optimizations of the Subresultant Algorithm". *J. of Pure and Applied Algebra* **145** (2000) 149–163.

[19] A. Eigenwillig: "On Multiple Roots in Descartes' Rule and Their Distance to Roots of Higher Derivatives"'. *J. of Computational and Applied Mathematics* **200** (2007) 226–230.

[20] A. Eigenwillig, M. Kerber, N. Wolpert: "Fast and Exact Geometric Analysis of Real Algebraic Plane Curves". In: C. W. Brown (ed.) *Proocedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC 2007)*. ACM, 2007 151–158.

[21] A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, N. Wolpert: "A Descartes Algorithm for Polynomials with Bit-Stream Coefficients". In: *8th International Workshop on Computer Algebra in Scientific Computing (CASC 2005)*, *LNCS*, vol. 3718, 2005 138–149.

[22] A. Eigenwillig, L. Kettner, E. Schömer, N. Wolpert: "Exact, Efficient, and Complete Arrangement Computation for Cubic Curves". *Computational Geom.* **35** (2006) 36–73.

[23] A. Eigenwillig, L. Kettner, N. Wolpert: "Snap Rounding of Bézier Curves". In: *Proc. of the 23rd Annual Symposium on Computational Geometry (SCG 2007)*. ACM, 2007 158–167.

[24] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, E. P. Tsigaridas: "Towards an Open Curved Kernel". In: *Proc. of the 20th Annual Symposium on Computational Geometry (SCG 2004)*. ACM, 2004 438–446.

[25] J. von zur Gathen, T. Lücking: "Subresultants revisited". *Theoretical Computer Science* (2003) 199–239.

[26] K. Geddes, S. Czapor, G. Labahn: *Algorithms for Computer Algebra*. Kluwer, 1992.

[27] C. G. Gibson: *Elementary geometry of algebraic curves*. Cambridge University Press, 1998.

[28] L. Gonzalez-Vega, T. Recio, H. Lombardi, M.-F. Roy: "Sturm-Habicht Sequences, Determinants and Real Roots of Univariate Polynomials". In [14], pp. 300–316.

[29] D. Halperin: "Arrangements". In: J. Goodman, J. O'Rourke (eds.) *Handbook of Discrete and Computational Geometry*, chap. 24. CRC Press, 2nd edn., 2004.

[30] D. Halperin, E. Leiserowitz: "Controlled perturbation for arrangements of circles". *Internat. J. of Comput. Geom. & Appl.* **14** (2004) 277–310.

[31] D. Halperin, C. R. Shelton: "A perturbation scheme for spherical arrangements with application to molecular modeling". *Computational Geom.* **10** (1998) 273–287.

[32] I. Hanniel, R. Wein: "An exact, complete and efficient computation of arrangements of Bézier curves". In: *Proc. of the 2007 ACM Symposium on Solid and Physical Modeling (SPM 2007)*. ACM, 2007 253–263.

[33] J. Keyser, T. Culver, D. Manocha, S. Krishnan: "Efficient and exact manipulation of algebraic points and curves". *Computer-Aided Design* **32** (2000) 649–662.

[34] K. Mehlhorn, S. Näher: *LEDA: A platform for combinatorial and geometric computing*. Cambridge Univ. Press, 1999.

[35] K. Mehlhorn, R. Osbild, M. Sagraloff: "Reliable and Efficient Computational Geometry Via Controlled Perturbation". In: *Automata, Languages and Programming, 33rd Internat. Colloquium (ICALP'06), Part I*, *LNCS*, vol. 4051. Springer, 2006 299–310.

[36] V. Milenkovic, E. Sacks: "An Approximate Arrangement Algorithm for Semi-algebraic Curves". *Internat. J. of Comput. Geom. & Appl.* **17** (2007) 175–198.

[37] A. W. Strzebonski: "Cylindrical Algebraic Decomposition using validated numerics". *J. of Symbolic Computation* **41** (2006) 1021–1038.

[38] R. J. Walker: *Algebraic Curves*. Princeton University Press, 1950.

[39] R. Wein: "High-Level Filtering for Arrangements of Conic Arcs". In: *Proc. of the 10th European Symposium on Algorithms (ESA 2002)*, 2002 884–895.

[40] R. Wein, E. Fogel, B. Zukerman, D. Halperin: "2D Arrangements". In: *CGAL-3.3 User and Reference Manual*, 2007. URL `http://www.cgal.org/Manual/`.

[41] R. Wein, B. Zukerman: *Exact and Efficient Construction of Planar Arrangements of Circular Arcs and Line Segments with Applications*. Tech. Rep. ACS-TR-121200-01, Tel Aviv University, Israel, 2006.

[42] N. Wolpert: "Jacobi Curves: Computing the Exact Topology of Arrangements of Non-Singular Algebraic Curves". In: *Proc. of the 11th Annual European Symposium on Algorithms (ESA 2003)*, *LNCS*, vol. 2832. Springer, 2003 532–543.

[43] C. K. Yap: *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.

[44] C. K. Yap: "Robust Geometric Computation". In: J. E. Goodman, J. O'Rourke (eds.) *Handbook of Discrete and Computational Geometry*, chap. 41, 927–952. CRC Press, 2nd edn., 2004.