Linear-Time Reordering in a Sweep-line Algorithm for Algebraic Curves Intersecting in a Common Point

Eric Berberich   Lutz Kettner

MPI–I–2007–1–001   July 2007

Abstract

The Bentley Ottmann sweep line algorithm is a standard tool to compute the arrangement of algebraic curves in the plane. If degenerate positions are not excluded from the input, variants of this algorithm must, among other things, handle $k \geq 2$ curves intersecting simultaneously in a single intersection point. In that situation, the algorithm knows the order of the curves immediately left of the intersection point and needs to compute the order immediately right of the intersection point.

Segments and lines can be reordered efficiently in linear time by simply reversing their order, except for overlapping segments. Algebraic curves can be sorted with $O(k \log k)$ geometric comparisons in their order immediately right of the intersection point. A previous result shows that algebraic curves whose degree is at most d can be reordered in $O(d^2 k)$ time, which is for constant d better than sorting.

In this paper, we improve the complexity of the reordering of algebraic curves to $O(k)$ time, i.e., independent of the degree of the algebraic curves. The maybe surprising implication is that algebraic curves, even of unbounded algebraic degree, cannot realize all possible permutations of their vertical order while passing through a common intersection from left to right. We give a short example for an infeasible permutation.

Both linear time algorithms require the knowledge of the intersection multiplicities of curves that are neighbors immediately left of the intersection point, i.e., $k-1$ intersection multiplicities.

1 Introduction

The Bentley Ottmann sweep-line algorithm [BO79] is a standard tool to compute the arrangement of segments, lines, or curves in the plane. In their original paper, Bentley and Ottmann describe the algorithm for reporting and counting segment intersections, but they exclude degeneracies, such as vertical segments and points where more than two segments intersect. They explain later in their paper how to handle vertical segments and that the algorithm extends immediately to x-monotone input curves. Since then, the sweep-line algorithm became so commonplace over the decades that good descriptions, including how to handle degeneracies and how to apply it to related problems, such as planar map or segment overlay computation or boolean operations on polygons in the plane, appear in several text books [dBvKOS00, Chapter 2] [MN99, Sections 10.7 & 10.8] [O'R98, Section 7.7].

In this paper we are specifically interested in how the degeneracy of several segments intersecting at once in a single point can be handled. Two solutions are commonly used: Perturbation methods report each pair of intersecting segments. Exact methods report the intersection only once. It has been discussed by Burnikel et al. [BMS94] that the exact method has various advantages, such as an indeed simple implementation of the sweep-line algorithm and better runtime and output size complexity in case of such degeneracies.

We study in this paper the exact method in the context of non-linear input data [FHK$^+$06], with the challenging example of algebraic curves of arbitrary degree as input.

For the sweep-line algorithm, curves have to be split into so called *sweepable segments*, which are (maximal) x-monotone segments that have no critical points in their interior and that fulfill some additional criteria, which are of no particular interest for the method we are studying in this paper. The details needed here are presented in Section 2. Throughout this work, we always assume a segment to be sweepable.

We continue with a short outline of the sweep-line algorithm and formulate then the problem solved in this paper.

1.1 Sweep-Line Algorithm

The sweep-line algorithm conceptually sweeps a vertical line, the *sweep line*, from left to right over the set of segments. We maintain three data structures with the following invariants: (1) All intersections of segments left of the sweep line have been reported and, depending of the application, used to build the output data structure, such as an arrangement. (2) All segments intersecting the sweep line are stored in sorted order in the *y-structure*. (3) All future segment intersection points of segments that are adjacent in the y-structure are stored together with all future segment endpoints in the *x-structure* sorted lexicographically according to their x- and y-coordinates. Key observation for the sweep-line algorithm is now that these invariants change only at discrete places, namely segment endpoints and intersection points, which are treated in a unified representation of an *event*. Following Mehlhorn and Näher [MN99, Sections 10.7 & 10.8] this event distinguishes simultaneously several segments ending in, several segments passing through, and several segments beginning in the event. We note that in general a vertical segment requires special handling, but its discussion can be omitted for our purposes.

An event is processed in three steps: The sweep-line algorithm first removes all segments from the y-structure that end at the event, then reorders the passing segments, and finally inserts the newly starting segments. In this work we focus on the middle step, the *reordering step*.

1.2 Reordering Step in the Sweep-Line Algorithm

We begin with some notation to describe the reordering step more precisely.

Definition 1 *Let p be a planar point that is the current event of the sweep-line algorithm. The maximal subsequence of segments in the y-structure passing through p is given by segments $s_1 \ldots s_k$ that are numbered according to their y-order just left of p. The supporting algebraic curve of a segment s_i is called f_i.*

Our goal is to reorder segments $s_1 \ldots s_k$ to reflect their order just right of the event, i.e., after the sweep line passed through the event and the segments have intersected each other in the common intersection point p. For straight-line segments this is obviously just an order reversal (except for overlapping segments), which can be implemented to require linear time in the number of reordered segments for common search tree data structures that one would use for the y-structure.

When we extend the sweep-line algorithm to handle curved input, we have, besides the obvious need for new geometric computations, to reconsider this reordering step; in contrast to intersecting pairs of straight-line segments that cross each other in the intersection point, intersecting segments of algebraic curves might intersect tangentially and do not cross. The *intersection multiplicity* between two segments of algebraic curves tells us whether the segments cross or do not cross at an intersection point: If the intersection multiplicity is odd, the segments cross, if it is even, the segments do not cross. We define the intersection multiplicity in the following Section 2.

How complex can k algebraic curves behave in a common intersection point? Can they achieve all possible permutations of their order? If so, the best runtime one could expect for reordering, without using additional information, is sorting from scratch to the right of the event. Then, the known order of the input segments to the left of the event cannot be used to any advantage.

But, Berberich et al. [BEH$^+$02] showed that k segments of algebraic curves with degree at most d can be reordered in $O(d^2k)$ time. Their algorithm assumes that the intersection multiplicities m_i between adjacent segments s_i, s_{i+1} in the y-structure are known. It makes $M := max_i\{m_i\}$ passes over the sequence of the k segments and reverses subsequences so that finally all pairs of odd intersection multiplicity have changed order and those of even intersection multiplicity have not changed order. Following Bézout's theorem [Wal50, III-§3.1], it holds $M \leq d^2$. This implies for constant d that algebraic curves cannot realize all permutations while passing through a common intersection.

Our result now shows that also algebraic curves of unbounded degree cannot realize all permutations while passing through a common intersection. In particular, Section 4 gives an algorithm for the reordering that runs in $O(k)$ time, which is independent of the degree of the involved algebraic curves. Similar to the work of Berberich et al. [BEH$^+$02] this algorithm assumes that the intersection multiplicity of neighboring segments is known. The key idea of the algorithm is a specific tree representation of the input, called the *multiplicity tree*. Its definition and construction is described in Section 3.

1.3 Generic Sweep-Line Implementations

Generic implementations of the sweep-line algorithm are, for example, available in LEDA[1] and CGAL's `Arrangement_2` package [WFZH07].[2] The latter nicely decouples the sweep-line algorithm from the actual construction of the desired output using the visitor design pattern [GHJV95]. Usually, the desired output consists of the induced arrangement which is reported as a doubly-connected edge list (DCEL). Just reporting all intersection points may be another output. LEDA's output is the induced arrangement represented in LEDA's graph type. Besides this issue, the two implementations mainly differ in how to maintain segments involved in a sweep event.

LEDA's implementation follows the description from above that for each event we deal with three lists of segments; those ending in, those passing through, and those beginning in an event.

On the other hand, CGAL's design maintains only two, but sorted, lists, namely segments adjacent to the left and segments adjacent to the right of an event. This

[1] LEDA homepage: `http://www.algorithmic-solutions.com/leda.htm`

[2] CGAL homepage: `http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Arrangement_2/Chapter_main.html`

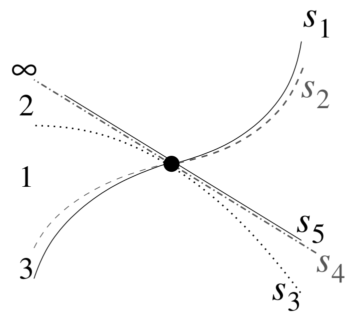

Figure 1: Segments changing their y-order in an intersection point. The numbers on the left are the intersection multiplicities of adjacent segments.

choice has implications for the sweep-line algorithm on how to process a single event. CGAL's implementation first removes all segments to the left of the event from the y-structure and inserts at the stored position all segments to the right of the event that have been kept sorted in the event. This sorting can be easy in some suitable situations, but in general, its worst case running time is $O((k+l)\log(k+l))$, where k is the number of passing segments and l is the number of starting segments at the event, which is in theory inferior to the linear time reordering presented here.

2 Intersection Multiplicity of Algebraic Curves

To define the multiplicity of intersection of two sweepable segments s_i, s_j we have to define the multiplicity of intersection for their supporting algebraic curves f_i, f_j first. This section mainly follows corresponding parts in [EKSW06].

Consider a square-free and y-regular algebraic curve f over the field $\mathbb{R}$. Its vanishing locus is a closed subset $f \subseteq \mathbb{R}^2$. Since the set $f \cap f_y$ of *critical points* is finite, $f \setminus f_y$ is an open subset of f. From the Implicit Function Theorem, it follows that every connected component of $f \setminus f_y$ is a parameterized curve

$$\begin{aligned} \gamma_i \colon]l_i, r_i[&\to \mathbb{R}^2 \\ x &\mapsto (x, \varphi_i(x)) \end{aligned} \tag{1}$$

with some analytic function φ_i (called the *implicit function*) and interval boundaries $l_i, r_i \in \mathbb{R} \cup \{\pm\infty\}$. In particular, every connected component of $f \setminus f_y$ is a 1-dimensional C^∞-submanifold of $\mathbb{R}^2$ which is homeomorphic to an open interval. The topological closure $A_i := \mathrm{cl}(\gamma_i(]l_i, r_i[))$ in $\mathbb{R}^2$ of each such component is called an *arc* of f. Since f is closed, $A_i \subseteq f$. The arcs of f form the set of maximal sweepable segments of f.

A generic change of coordinates makes $f_y(p) \neq 0$ for a regular point p, demonstrating that f is a manifold around a critical point p that is not a singular point of f. A singular point p, however, is singular and hence critical in any coordinate system.

The parameterization (1) involves a function φ_i which can be expressed as a convergent power series locally around any x in its domain. Such a power series is a special case of the more general notion of *Puiseux series* (a kind of series involving fractional powers of x) that allow parameterizations even at critical points, see [Wal50, IV.] and [BK86].

The intersection multiplicity of two curves f and g at a point p, defined as multiplicity of the corresponding root of the resultant of f and g in a generic coordinate system [BK86, p. 231], measures the similarity of these implicit power series.

Proposition 1 *Let $f, g \in \mathbb{C}[x,y]$ be two coprime y-regular algebraic curves. Let $p = (p_1, p_2) \in \mathbb{C}^2$ be an intersection point of f and g that is critical on neither of them. Then the intersection multiplicity of f and g at p is the smallest exponent d for which the coefficients of $(x - p_1)^d$ in the implicit power series of f and g around p disagree.*

Proof: (Sketch only.) Choose a generic coordinate system in which $p = (0,0)$. Consider f and g as univariate polynomials in y whose zeroes can be written as power series $\alpha_i(x)$ and $\beta_j(x)$, respectively. Choose indices such that the arcs intersecting at $(0,0)$ are α_1 and β_1. Recall that

$$\operatorname{res}(f,g,y)(x) = \ell(f)^{\deg(g)} \ell(g)^{\deg(f)} \prod_{i,j} (\alpha_i(x) - \beta_j(x)).$$

By choice of generic coordinates, $\alpha_i(0) - \beta_j(0) \neq 0$ for $(i,j) \neq (1,1)$. Hence the multiplicity d of $x = 0$ as a root of $\alpha_1(x) - \beta_1(x)$ is the multiplicity of $x = 0$ as a root of $\operatorname{res}(f,g,y)(x)$. □

A full proof is given by Walker [Wal50, Thm. IV-5.2]. The following corollary is immediate.

Corollary 1 *In the situation of Proposition 1 for $f, g \in \mathbb{R}[x,y]$, the two arcs of f and g intersecting at a point $p \in \mathbb{R}^2$ change sides iff their intersection multiplicity is odd.*

We now switch to two adjacent segments s_i, s_{i+1} in the y-structure supported by algebraic curves f_i and f_{i+1}. This step requires the precondition that sweepable segments do not contain critical points of their supporting algebraic curves in their interior. Why is this the case? First, the sweep-line algorithm originally allows to deal only with x-monotone parts. Second, the sweep-line algorithm will detect this special point anyhow, and it seems combinatorially better to split curves at singular points instead of creating segments that were found to intersect anyhow, beside the issue of a more sophisticated algebraic analysis to define the corresponding multiplicity of intersection. Note that splitting an algebraic curve into its sweepable segments at all critical points does not harm the sweep-line paradigm, and especially not the reordering step. Segments with critical points at their end, are either removed from the y-structure before the reordering step, or will be inserted right after it. Only segments' intersections at non-critical points are considered during the reordering steps.

This allows us to write a segment s_i, that is involved in the reordering, locally as an analytic implicit function

$$y = \varphi_i(x) = \sum_{d=1}^{\infty} a_d^{(i)} x^d \tag{2}$$

after translating the event to $(0,0)$. The coefficients of the implicit functions determine the y-order of segments just left and just right of the intersection.

Proposition 2 *With notation as above:*
Segment s_i lies below segment s_{i+1} right of the intersection iff

$$(a_1^{(i)}, a_2^{(i)}, \ldots, a_d^{(i)}, \ldots) <_{\text{lex}} (a_1^{(I+1)}, a_2^{(i+1)}, \ldots, a_d^{(i+1)}, \ldots).$$

Segment s_i lies below segment s_{i+1} left of the intersection iff

$$(-a_1^{(i)}, a_2^{(i)}, \ldots, (-1)^d a_d^{(i)}, \ldots) <_{\text{lex}} (-a_1^{(i+1)}, a_2^{(i+1)}, \ldots, (-1)^d a_d^{(i+1)}, \ldots).$$

(Here $<_{\text{lex}}$ is the lexicographic order relation on sequences of real numbers.)
Proof: It suffices to demonstrate the first part; the second part follows by substituting $-x$ for x. Iff the segments overlap, they coincide around the intersection and have equal coefficient sequences. Otherwise, a finite $m = \min\{d \mid a_d^{(i)} \neq a_d^{(i+1)}\}$ exists, and $\varphi_i(x) - \varphi_{i+1}(x) = (a_m^{(i)} - a_m^{(i+1)})x^m + \ldots$ is negative for small positive x iff $a_m^{(i)} < a_m^{(i+1)}$. □

By Proposition 1, the quantity m considered in the proof for non-overlapping segments is precisely the intersection multiplicity of the segments' supporting algebraic curves. Incorporating the case of overlap, we define the intersection multiplicity of segments s_i and s_{i+1} as $\min(\{d \mid a_d^{(i)} \neq a_d^{(i+1)}\} \cup \{\infty\})$. For reasons that come into sight later, we define ∞ to be an even number.

We do not explain how to compute the intersection multiplicity for a particular pair of segments s_i, s_{i+1}. If a generic coordinate system is given, locating intersection points of s_i, s_{i+1} by resultant computations produces the required intersection multiplicity as a by-product for free; for details see [Wal50, Thm. IV-5.2], or the proof of Proposition 1. We do not discuss the effort required in a non-generic coordinate system. The computation of such intersection multiplicities may heavily depend on the algebraic degree of the involved curves and is discussed elsewhere.

We next show that the multiplicity of intersection for an arbitrary pair of segments s_i, s_j in the given sequence can be obtained from the sequence of intersection multiplicities for adjacent pairs s_i, s_{i+1} only.

Proposition 3 *With notation as above, the intersection multiplicity of two segments s_i and s_j, $1 \leq i < j \leq k$, is* $\min\{m_i, \ldots, m_{j-1}\}$.

Proof: By induction on j. The base case $j = i+1$ is clear. For the inductive step from j to $j+1$, let $m = \min\{m_i, \ldots, m_{j-1}\}$ be the intersection multiplicity of s_i and s_j. For $m_j = \infty$, the claim is clear. Otherwise, distinguish three cases:

The first case is $m > m_j$. It holds that $a_d^{(j+1)} = a_d^{(j)} = a_d^{(i)}$ for $d < m_j$ and $a_d^{(j+1)} \neq a_d^{(j)} = a_d^{(i)}$ for $d = m_j$, so that the intersection multiplicity of s_i and s_{j+1} is $m_j = \min\{m, m_j\}$.

For $m < m_j$, we have equality for $d < m$ and inequality $a_d^{(j+1)} = a_d^{(j)} \neq a_d^{(i)}$ for $d = m$, demonstrating the intersection multiplicity $m = \min\{m, m_j\}$.

However, if $m = m_j$, then only a double inequality $a_d^{(j+1)} \neq a_d^{(j)} \neq a_d^{(i)}$ holds for $d = m$, but we need $a_m^{(j+1)} \neq a_m^{(i)}$. Proposition 2 helps: Since s_{j+1} lies above s_j and intersects with multiplicity $m_j = m$, we know $(-1)^m a_m^{(j+1)} > (-1)^m a_m^{(j)}$. By the analogous argument for s_j and s_i, we know $(-1)^m a_m^{(j)} > (-1)^m a_m^{(i)}$. Hence $(-1)^m a_m^{(j+1)} > (-1)^m a_m^{(i)}$, as required. □

3 Multiplicity Tree

We next define an ordered tree T, the *multiplicity tree*. Its leaves represent the segments s_i in their order left of the intersection point. Its nodes represent intersection multiplicities m_i. The construction of the tree is defined recursively:

- Pick the smallest multiplicity and create a root node with its value. Partition the sequence into subsequences that end wherever two adjacent segments are separated by this smallest multiplicity.
- Recursively create a subtree under the root node for each subsequence.
- The recursion stops with the trivial subsequence that represents a single segment for which we create a leaf.

Some observations on the resulting tree T. It has linear size in the number of segments. The multiplicity value in a node is strictly smaller than all multiplicity values in nodes of its subtrees. The leaf for segment s_i is linked to a node that has the multiplicity $\max(m_{i-1}, m_i)$. Each subtree S of T defines a maximal subsequence $s_q, s_{q+1}, \ldots, s_r$ with the property $m = \min\{m_q, \ldots, m_{r-1}\}$, where m is the multiplicity of the root node of S. We call s the *defining multiplicity* of the subsequence.

We now give an algorithm that builds the multiplicity tree in linear time. The algorithm maintains a stack of subtrees while it reads the sequence $m_1, \ldots, m_{k-1}$ from "bottom to top". After we have processed $m_1, \ldots, m_{i-1}$ and come to read m_i, the following invariant holds:

(I1) The root nodes of the stack elements have strictly increasing multiplicities.
(I2) Each subtree in the stack represents an unfinished maximal subsequence. The multiplicity of its root node is the defining multiplicity for the subsequence.
(I3) If a maximal subsequence is completed, then its representing subtree is a descendant of a stack element.
(I4) $s_1, \ldots, s_{i-1}$ are leaves of the respective subtrees in the stack.

At the bottom of the stack we place a dummy node with multiplicity $m_0 = 0$ as a sentinel. We also add a sentinel $m_k = 0$ to the sequence of multiplicities. The pseudo-code MAKETREE explains in detail how to create the tree.

We now underline its key steps. Pushing a new node onto the stack in line (11) corresponds to the opening of a new subsequence with defining multiplicity m_i. The extensions of v_{top} by v in lines (13) and (20) is an order-preserving concatenation of their stored sequences. Observe that these sequences can be made of linked subtrees. Reaching line (15) identifies that m_i closes those subsequences whose defining multiplicity is larger than the current m_i. It follows that all subtrees on the stack for these multiplicities have been completed. We therefore merge them into one subtree and attach it to the current node v.

Let us now consider the special case of two overlapping segments s_i and s_j. They belong to a sequence of pairwise overlapping segments $s_q, s_{q+1}, \ldots, q_j$, so that $m_i = m_{q+1} = \ldots = m_{r-1} = \infty$. As overlapping segments do not change their order when passing a common point, we only have to define ∞ to be a (very large) *even* number, to see that the multiplicity tree still stores the correct order to the left of p. After the sentinel $m_k = 0$ has been processed, we end up with the final tree T rooted by an extra node with "multiplicity" 0 as the only node on the stack.

Algorithm: MAKETREE

```
INPUT:  segments of algebraic curves s_1 to s_k
        intersection multiplicities m_1 to m_{k-1} between segments s_1 to s_k
OUTPUT: multiplicity tree T

m_0 ← 0
m_k ← 0
for i = 0...k do
    Create new node v representing m_i
    if m_{i-1} < m_i
        Attach s_i as leaf to v
    endif
    v_top ← top element of stack
    m_top ← stored multiplicity of v_top
    if m_i > m_top do
        Push v onto stack
    else if m_i = m_top do
        Extend v_top by v
    else if m_i < m_top do
        Pop all stack elements with multiplicities > m_i and
        join them into one subtree S by making each element,
        except for the lowest, a child of the element below.
        Attach S to v.
        v_top ← new top element of stack
        m_top ← stored multiplicity of v_top
        if m_i = m_top do
            Extend v_top by v
        else
            Push v onto the stack
        endif
    endif
    if (m_i ≥ m_{i+1}) do
        Attach s_{i+1} as a leaf to v_top
    endif
end
return top element of stack
```

4 Linear-time Reordering

The following observation allows us to use T for the reordering of segments just to the right of p. The first common ancestor of two segments s_i and s_j in the multiplicity tree represents the intersection multiplicity of s_i and s_j. Thus we obtain the order just right of p, if we reverse the order in all nodes with odd multiplicity.

Lemma 1 *The* MAKETREE *algorithm computes the multiplicity tree of the segments s_1 to s_k in time $O(k)$.*

Proof: Correctness: We show by induction over i that the invariants (I1)-(I4) hold. Let $i = 0$. The invariants hold trivially. Now assume that the invariants are met after m_{i-1} is inserted. We now show that they still hold after we inserted m_i. Invariant (I1): Only pushing a new node on the stack can destroy this invariant. New nodes are pushed in lines 11 and 22. In the former case, it is ensured that m_i is greater than the current top element of the stack. In the latter case, all stack elements with value $> m_i$ are removed first in line 15. Therefore, in both cases we have $m_{\text{top}} < m_i$ which cannot destroy invariant (I1). Invariant (I2): Remember that a subsequence can finish only if $m_i < m_{i-1}$. In this case, we can conclude that $m_{\text{top}} > m_i$, as by induction $m_{i-1} = m_{\text{top}}$. As subsequences remain open until reaching line 15, it is shown that all subtrees defined by stack elements represent unfinished sequences. The same argument shows invariant (I3). For invariant (I4) we have to show that s_i is a leaf of an (unfinished) subtree. To do so one needs to consider lines 5 and 6 in the current iteration together with lines 25 and 26 in the previous iteration. Let us start with the latter where we had for the current m_i the condition $m_i \leq m_{i-1}$, i.e., handling m_i will extend or close a subsequence in the current iteration. So s_i must be a leaf of the tree of that subsequence and therefore it already has been added in line 26 of the previous iteration. Otherwise, m_i just started a new subsequence (line 11) in this iteration and we added s_i as a leaf to the new subtree defined by v in line 06.

After we have processed m_k, the tree stored in the only remaining stack element has the additional root node with multiplicity 0. We claim that its only subtree represents the multiplicity tree as recursively defined above. Following invariant (I4) all s_i are contained in the subtree. Due to invariant (I3) all its subsequences, represented by its subtrees, are finished that are rooted by nodes with the defining multiplicity of the subsequence (invariant I2). Finally, invariant (I1) together with the subtree construction in line 15 of the algorithm show that the node multiplicities increase on each path from the root to a leaf.

Runtime: The tree size and the tree construction examines linearly many new nodes. Each node is pushed once on the stack and is removed once from the stack. At most a linear number of constant time merges between nodes can happen. □

The REORDERSEGMENTS algorithm consists of three steps: (i) It calls the MAKETREE algorithm to build a multiplicity tree T. (ii) It traverses T in a suitable depth-first-search leaf-traversal, where inner nodes with odd stored multiplicity are traversed in reversed order. (iii) It updates the y-structure corresponding to the new order read off T in the traversal. The algorithm can also return a sequence $s_{i_0}, \ldots, s_{i_k}$ such that $i_0, \ldots, i_k$ is the desired permutation of $1, \ldots, k$.

Note that we will have to argue about concrete implementations for the y-structure to analyze the runtime of step (iii). A y-structure is, in an abstract data type sense, a sorted sequence, and can be realized for example with a balanced tree, a heap, or LEDA's skiplist. We will make a conservative assumption about their runtimes in the proof below.

Theorem 1 *The* REORDERSEGMENTS *algorithm computes the y-order of segments of algebraic curves s_1 to s_k passing through a common point p immediately to the right of p from the order immediately to the left of p in time $O(k)$.*

Proof: Correctness of reordering: Consider two arbitrary segments s_i and s_j with $i < j$. Their y-order right of p differs from their y-order left of p iff $s = \min\{m_i, \ldots, m_{j-1}\}$ is odd. The first common ancestor of s_i and s_j in the multiplicity tree represents this multiplicity s. The order is reversed in this node iff s is odd, and no other reordering of tree nodes affects the order of s_i with respect to s_j.

Runtime: The tree size is linear, it is built in linear time, and its modified traversal requires linear time as well. The necessary update operations on a sorted sequence data structure representing the y-structure fall into two categories; we expect at most $O(\log k)$ time for locating and $O(k)$ time for removing the old subsequence, and (k) time for the k insertions (at known position). [3] □

5 An Example of an Infeasible Reordering

A direct implication of Theorem 1 is that the order of algebraic curves cannot realize all permutations when passing through a common point p. We illustrate an impossible permutation with a minimal number of segments in Figure 2, which is due to A. Eigenwillig[4]. To the left we have four segments, s_1, s_2, s_3, and s_4, and we assume that we can assign multiplicities m_1, m_2, and m_3 such that the given permutation to the right of the event, namely s_2, s_4, s_1, and s_3, is attained.

[3]Observe the localized setting, i.e., in a particular implementation a single update operation might take more time, e.g., due to balancing issues. We do not this discuss this problem, as, e.g., operations on LEDA's skiplist satisfies the given bounds.

[4]Personal communication, July 2007.

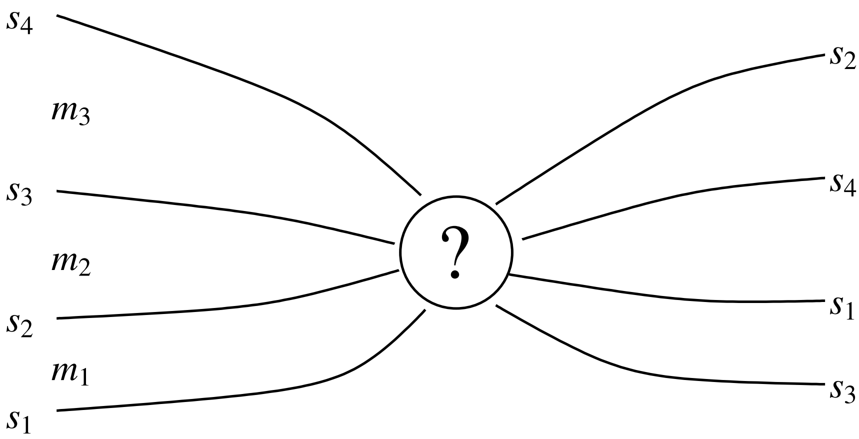

Figure 2: The order of segments $s_1, \dots, s_4$ shown on the right cannot be obtained by assigning odd or even multiplicities m_1, m_2, and m_3.

To do so, let us first have a closer look at the required parities of m_1, m_2, and m_3.

- s_1 and s_2 do not change order $\Rightarrow m_1$ must be even.
- s_2 and s_3 do change order $\Rightarrow m_2$ must be odd.
- s_3 and s_4 do not change order $\Rightarrow m_3$ must be even.
- s_1 and s_3 do change order $\Rightarrow \min\{m_1, m_2\}$ is odd $\Rightarrow m_2 < m_1$.
- s_2 and s_4 do change order $\Rightarrow \min\{m_2, m_3\}$ is odd $\Rightarrow m_2 < m_3$.

It follows that $\min\{m_1, m_2, m_3\} = m_2$ must be odd, which implies that s_1 and s_4 must change order. But s_1 and s_4 do not change their order, which is a contradiction. Our assumption that the given permutation can be realized is false, q.e.d.

6 Conclusions

We have shown how to reorder a set of segments $s_1, \dots, s_k$ passing through a common point p in a sweep-line setting in time $O(k)$. Our result removes the former dependency on the degree d of the algebraic curves and thus improves the combinatorial time complexity for the prevalent sweep-line paradigm in computing arrangements of algebraic curves.

However, the algorithm depends on the knowledge of intersection multiplicities of adjacent curves. The cost for their computation would need to be considered in addition when comparing this approach in practice with other approaches, such as the naive sorting to the right. Remember the situation of a generic coordinate system as in Proposition 1 where the multiplicity of intersection is computed as a by-product when two curves were checked to intersect using a resultant approach. As we require the intersections anyhow, these values come for free and the proposed reordering step will surely outperform previously known methods. If the computation of intersection multiplicities itself requires to call additional geometric analyses, (e.g., applying a shear to obtain a generic coordinate system and obtain the intersection multiplicities from it) the total runtime in practice, measured in seconds, will depend on different factors, like the degree of the involved curves, their coefficient's bitlength, et cetera. Experiments to obtain a better understanding for these differences are planned.

Therefore, our next goal is to implement the proposed algorithm in the SWEEPX library of EXACUS [BEH+05] and in CGAL's `Arrangement_2` package. The implementation is currently hindered by the fact that CGAL's sweep-line algorithm does not store passing segments of an event explicitly.

Last but not least, one may ask if the algorithm can be extended to cope with some uncertainty, such as if some m_i are not know.

7 Acknowledgements

We want to thank Arno Eigenwillig for providing detailed material on intersection multiplicities of algebraic curves, and for the example given in Section 5. Our thanks also goes to Michael Hemmer and Kurt Mehlhorn for fruitful discussions.

This work is partially supported by the IST Programmes of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473, Effective Computational Geometry for Curves and Surfaces (ECG) and Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

References

[BEH+02] 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 *Proc. 10th European Symposium on Algorithms (ESA)*, LNCS 2461, pages 174–186. Springer, 2002.

[BEH+05] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, and Nicola Wolpert. EXACUS: Efficient and exact algorithms for curves and surfaces. In *Proc. 13th European Symposium on Algorithms (ESA)*, LNCS 3669, pages 155–166. Springer, 2005.

[BK86] Egbert Brieskorn and Horst Knörrer. *Plane algebraic curves*. Birkhäuser Verlag, Basel, 1986. Translated from German by John Stillwell.

[BMS94] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[BO79] Jon L. Bentley and Thomas A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[EKSW06] Arno Eigenwillig, Lutz Kettner, Elmar Schömer, and Nicola Wolpert. Exact, efficient, and complete arrangement computation for cubic curves. *Comput. Geom. Theory Appl.*, 35(1):36–73, 2006.

[FHK+06] Efi Fogel, Dan Halperin, Lutz Kettner, Monique Teillaud, Ron Wein, and Nicola Wolpert. Arrangements. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, chapter 1, pages 1–66. Spinger, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[O'R98] Joseph. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.

[Wal50] Robert Walker. *Algebraic Curves*. Princeton University Press, 1950.

[WFZH07] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Computational Geometry Theory and Applications*, 38(1–2):37–63, 2007.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2007-RG1-002	T. Hilldenbrand, C. Weidenbach	Superposition for Finite Domains
MPI-I-2007-5-001	G. Ifrim, G. Kasneci, M. Ramanath, F.M. Suchanek, G. Weikum	NAGA: Searching and Ranking Knowledge
MPI-I-2007-4-004	C. Stoll	A Volumetric Approach to Interactive Shape Editing
MPI-I-2007-4-003	R. Bargmann, V. Blanz, H. Seidel	A Nonlinear Viseme Model for Triphone-Based Speech Synthesis
MPI-I-2007-4-002	T. Langer, H. Seidel	Construction of Smooth Maps with Mean Value Coordinates
MPI-I-2007-4-001	J. Gall, B. Rosenhahn, H. Seidel	Clustered Stochastic Optimization for Object Recognition and Pose Estimation
MPI-I-2007-2-001	A. Podelski, S. Wagner	A Method and a Tool for Automatic Veriication of Region Stability for Hybrid Systems
MPI-I-2006-5-006	G. Kasnec, F.M. Suchanek, G. Weikum	Yago - A Core of Semantic Knowledge
MPI-I-2006-5-005	R. Angelova, S. Siersdorfer	A Neighborhood-Based Approach for Clustering of Linked Document Collections
MPI-I-2006-5-004	F. Suchanek, G. Ifrim, G. Weikum	Combining Linguistic and Statistical Analysis to Extract Relations from Web Documents
MPI-I-2006-5-003	V. Scholz, M. Magnor	Garment Texture Editing in Monocular Video Sequences based on Color-Coded Printing Patterns
MPI-I-2006-5-002	H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum	IO-Top-k: Index-access Optimized Top-k Query Processing
MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-Aware Global df Estimation in Distributed Information Retrieval Systems
MPI-I-2006-4-010	A. Belyaev, T. Langer, H. Seidel	Mean Value Coordinates for Arbitrary Spherical Polygons and Polyhedra in $\mathbb{R}^3$
MPI-I-2006-4-009	J. Gall, J. Potthoff, B. Rosenhahn, C. Schnoerr, H. Seidel	Interacting and Annealing Particle Filters: Mathematics and a Recipe for Applications
MPI-I-2006-4-008	I. Albrecht, M. Kipp, M. Neff, H. Seidel	Gesture Modeling and Animation by Imitation
MPI-I-2006-4-007	O. Schall, A. Belyaev, H. Seidel	Feature-preserving Non-local Denoising of Static and Time-varying Range Data
MPI-I-2006-4-006	C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel	Enhanced Dynamic Reflectometry for Relightable Free-Viewpoint Video
MPI-I-2006-4-005	A. Belyaev, H. Seidel, S. Yoshizawa	Skeleton-driven Laplacian Mesh Deformations
MPI-I-2006-4-004	V. Havran, R. Herzog, H. Seidel	On Fast Construction of Spatial Hierarchies for Ray Tracing
MPI-I-2006-4-003	E. de Aguiar, R. Zayer, C. Theobalt, M. Magnor, H. Seidel	A Framework for Natural Animation of Digitized Models
MPI-I-2006-4-002	G. Ziegler, A. Tevs, C. Theobalt, H. Seidel	GPU Point List Generation through Histogram Pyramids
MPI-I-2006-4-001	A. Efremov, R. Mantiuk, K. Myszkowski, H. Seidel	Design and Evaluation of Backward Compatible High Dynamic Range Video Compression
MPI-I-2006-2-001	T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard	On Verifying Complex Properties using Symbolic Shape Analysis
MPI-I-2006-1-007	H. Bast, I. Weber, C.W. Mortensen	Output-Sensitive Autocompletion Search

MPI-I-2006-1-006	M. Kerber	Division-Free Computation of Subresultants Using Bezout Matrices
MPI-I-2006-1-005	A. Eigenwillig, L. Kettner, N. Wolpert	Snap Rounding of Bézier Curves
MPI-I-2006-1-004	S. Funke, S. Laue, R. Naujoks, L. Zvi	Power Assignment Problems in Wireless Communication
MPI-I-2005-5-002	S. Siersdorfer, G. Weikum	Automated Retraining Methods for Document Classification and their Parameter Tuning
MPI-I-2005-4-006	C. Fuchs, M. Goesele, T. Chen, H. Seidel	An Emperical Model for Heterogeneous Translucent Objects
MPI-I-2005-4-005	G. Krawczyk, M. Goesele, H. Seidel	Photometric Calibration of High Dynamic Range Cameras
MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data
MPI-I-2005-4-001	M. Fuchs, V. Blanz, H. Lensch, H. Seidel	Reflectance from Images: A Model-Based Approach for Human Faces
MPI-I-2005-2-004	Y. Kazakov	A Framework of Refutational Theorem Proving for Saturation-Based Decision Procedures
MPI-I-2005-2-003	H.d. Nivelle	Using Resolution as a Decision Procedure
MPI-I-2005-2-002	P. Maier, W. Charatonik, L. Georgieva	Bounded Model Checking of Pointer Programs
MPI-I-2005-2-001	J. Hoffmann, C. Gomes, B. Selman	Bottleneck Behavior in CNF Formulas
MPI-I-2005-1-008	C. Gotsman, K. Kaligosi, K. Mehlhorn, D. Michail, E. Pyrga	Cycle Bases of Graphs and Sampled Manifolds
MPI-I-2005-1-007	I. Katriel, M. Kutz	A Faster Algorithm for Computing a Longest Common Increasing Subsequence
MPI-I-2005-1-003	S. Baswana, K. Telikepalli	Improved Algorithms for All-Pairs Approximate Shortest Paths in Weighted Graphs
MPI-I-2005-1-002	I. Katriel, M. Kutz, M. Skutella	Reachability Substitutes for Planar Digraphs
MPI-I-2005-1-001	D. Michail	Rank-Maximal through Maximum Weight Matchings
MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae
MPI-I-2004-NWG1-001	B. Blanchet	Automatic Proof of Strong Secrecy for Security Protocols
MPI-I-2004-5-001	S. Siersdorfer, S. Sizov, G. Weikum	Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning
MPI-I-2004-4-006	K. Dmitriev, V. Havran, H. Seidel	Faster Ray Tracing with SIMD Shaft Culling
MPI-I-2004-4-005	I.P. Ivrissimtzis, W.-. Jeong, S. Lee, Y.a. Lee, H.-. Seidel	Neural Meshes: Surface Reconstruction with a Learning Algorithm
MPI-I-2004-4-004	R. Zayer, C. Rössl, H. Seidel	r-Adaptive Parameterization of Surfaces
MPI-I-2004-4-003	Y. Ohtake, A. Belyaev, H. Seidel	3D Scattered Data Interpolation and Approximation with Multilevel Compactly Supported RBFs
MPI-I-2004-4-002	Y. Ohtake, A. Belyaev, H. Seidel	Quadric-Based Mesh Reconstruction from Scattered Data
MPI-I-2004-4-001	J. Haber, C. Schmitt, M. Koster, H. Seidel	Modeling Hair using a Wisp Hair Model
MPI-I-2004-2-007	S. Wagner	Summaries for While Programs with Recursion