An Exact, Complete and Efficient Computation of Arrangements of Bézier Curves

Iddo Hanniel*
Department of Computer Science
Technion, Israel Institute of Technology

Ron Wein†
School of Computer Science
Tel-Aviv University, Israel

Abstract

Arrangements of planar curves are fundamental structures in computational geometry. The arrangement package of CGAL can construct and maintain arrangements of various families of curves, when provided with the representation of the curves and some basic geometric functionality on them. It employs the exact computation paradigm in order to handle all degenerate cases in a robust manner. We present the representations and algorithms that are needed for implementing arrangements of Bézier curves using exact arithmetic. The implementation is efficient and complete, handling all degenerate cases. In order to avoid the prohibitive running times incurred by an indiscriminate usage of exact arithmetic, we make extensive use of the geometric properties of Bézier curves for filtering. As a result, most operations are carried out using fast approximate methods, and only in degenerate (or near-degenerate) cases do we resort to the exact, and more computationally demanding, procedures. To the best of our knowledge this is the first complete implementation that can construct arrangements of Bézier curves of any degree, and handle all degenerate cases in a robust manner.

CR Categories: F.2.1 [Numerical Algorithms and Problems]: Computations on polynomials; J.2 [Computer Applications]: Physical Sciences and Engineering—Engineering

Keywords: Bézier curves, arrangements, exact computation, robustness, CGAL.

1 Introduction

Given a set $\mathscr{C}$ of planar curves, the *arrangement* $\mathscr{A}(\mathscr{C})$ is the subdivision of the plane induced by the curves in $\mathscr{C}$ into maximally connected cells of dimensions 0 (*vertices*), 1 (*edges*), or 2 (*faces*). The arrangement can be embedded as a planar graph, such that each arrangement vertex corresponds to a planar point, representing a curve endpoint or an intersection between two curves (or more), and each edge corresponds to an x-monotone subcurve of one of the curves in $\mathscr{C}$, whose interior is disjoint from all other subcurves. Arrangements are ubiquitous in computational geometry, and have numerous applications [Agarwal and Sharir 2000; Halperin 2004]. For instance, it is possible to compute Boolean operations on general polygons[1] by overlaying the arrangements formed by their boundary curves.

Robustness in geometric computation is a fundamental, difficult and well-known problem arising from the special nature of geometric algorithms [Hoffmann 1989; Hoffmann 2001; Yap 2004]. In the classic computational geometry literature two assumptions are usually made to simplify the design and analysis of geometric algorithms. First, inputs are in "general position" — that is, degenerate input (e.g., three curves intersecting at a common point) is precluded. Secondly, operations on real numbers yield accurate results (the "real RAM" model [Preparata and Shamos 1985], which also assumes that each basic operation takes constant time). Unfortunately, these assumptions do not hold in practice. Thus, an algorithm implemented from a textbook using machine-precision arithmetic may yield incorrect results, get into an infinite loop, or just crash, while running on a degenerate, or nearly-degenerate, input; see [Hoffmann 1989; Hoffmann 2001; Kettner et al. 2004; Schirra 2000] for examples.

CGAL, the Computational Geometry Algorithms Library,[2] is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. The arrangement package of CGAL [Wein et al. 2006a] adopts, as does CGAL in general, the *exact computation* paradigm [Yap and Dubé 1995]. Namely, it requires that all geometric operations are carried out in a precise manner, so it can correctly detect degenerate situations and handle them properly. It thus makes no general-position assumptions on its input.

The main component in the arrangement package is a class-template named `Arrangement_2<Traits>`.[3] The arrangement class is parameterized with a geometric *traits* class [Myers 1997], which defines a family of planar curves and supplies basic geometric operations on curves of this family. Given the traits-class primitives, the `Arrangement_2` template can perform operations like constructing an arrangement using the sweep-line algorithm, inserting new curves into the arrangement, answering queries on the arrangement, etc. Several traits-classes are provided with the arrangement package, among them a class that handles line segments, another for polylines, for circular arcs, and a traits class for conic arcs [Wein et al. 2006b]. While the primitive operations on linear curves, namely line segments and polylines, can be performed using exact rational arithmetic,[4] handling non-linear curves is more complicated and computationally demanding, as it requires exact computation with algebraic numbers. CGAL also includes several peripheral package that rely on the arrangement infrastructure. An important package whose implementation is based on the arrangement infrastructure is the 2D Boolean set-operation package [Fogel et al. 2006b], which supports operations on general polygons, bounded by any family of curves representable by some arrangement-traits class (i.e., line segments, conic arcs, etc.).

Bézier curves are a well known family of parametric curves, whose

*e-mail: `iddoh@cs.technion.ac.il`
†e-mail: `wein@tau.ac.il`

[1] A *general polygon* is a continuous domain bounded by a chain of arbitrary (not necessarily linear) curves.

[2] See the CGAL project homepage, at `http://www.cgal.org/`.

[3] CGAL prescribes the suffix _2 for all data structures of planar objects as a convention.

[4] See for example GMP — Gnu's multi-precision library, at `http://www.swox.com/gmp/`.

representation is based on the Bernstein polynomials. They have numerous applications in computer graphics and solid modeling. Bézier curves are used, for example, in 2D free-form sketches. Another common use of Bézier curves is for defining the outline of fonts. Furthermore, B-spline curves can be reduced to a sequence of Bézier curves using *knot insertion* [Cohen et al. 2001].

In this paper we describe the implementation of a geometric traits class that handles planar Bézier curves of an arbitrary degree. We discuss the exact implementation of the traits-class methods using certified algebraic computation. This implementation is complete, but unfortunately it incurs a large running-time penalty, which may be unacceptable for many practical applications. We therefore incorporate geometric filtering and lazy evaluation techniques in our traits class, making use of the geometric properties of Bézier curves. The resulting traits class is able to perform most computations using only operations on polygons that bound the curves and resort to exact algebraic methods only in degenerate or near-degenerate cases, where operations on the bounding polygons are not sufficient to yield unambiguous results. Our Bézier-traits class thus yields efficient running times, and at the same time is also complete and can handle all sorts of degeneracies in a precise manner.

Arrangements of free-form curves are an essential part of industrial CAD systems. They are used, for example, for sketches in the design process. To the best of our knowledge, all current CAD systems implementations use floating-point arithmetic with different numerical tolerances in their computations. Such implementations are, as noted above, not robust.

Previous work on robust arrangements of Bézier curves include [Neagu and Lacolle 1998] and [Hanniel and Halperin 2000]. Neagu and Lacolle describe a subdivision-based algorithm for computing the topological structure of an arrangement of convex Bézier curves by performing operations on the control polygons of the curves. Hanniel and Halperin constructed CGAL arrangements of polygonal curves that tightly approximate the subdivision induced by a set of Bézier curves. The construction is adaptive and the implementation supports point-location queries. However, both [Neagu and Lacolle 1998] and [Hanniel and Halperin 2000] assume that the input curves are convex and are in general position, thus they are far from providing a complete implementation.

Several software implementations arrangements of *implicit* algebraic curves[5] have been developed in the past. Computer algebra systems[6] can be used to compute exact arrangements of algebraic curves. Although such systems can handle all degeneracies, they are impractical due to their runtime overhead. MAPC [Keyser et al. 2000], a C++ library for manipulating algebraic points and curves, uses exact arithmetic to perform operations on algebraic curves and has been applied for computing arrangements of algebraic curves in the plane. While the MAPC implementation is efficient and manages to avoid many robustness issues by the use of exact arithmetic, it does not handle all degenerate situations. For example, it cannot identify a common intersection point of three (or more) curves. A geometric traits class that handles algebraic curves of degree three (cubic curves) [Eigenwillig et al. 2004], and another traits class for special types of curves of degree four [Berberich et al. 2005b], have been implemented under the the EXACUS project [Berberich et al. 2005a]. These implementations are complete and robust but they are limited in the degree of the curves they handle. Furthermore, any Bézier curve can be converted to an implicit algebraic form (see, e.g., [Sederberg and Parry 1986]) and thus, in theory, these classes can be used for arrangements of low-degree Bézier curves.

[5]Implicit algebraic curves are given by the equation $C(x,y) = 0$, where C is a bivariate polynomial with rational coefficients.

[6]See, e.g., Maple's homepage, `http://www.maplesoft.com/`.

However, as we will demonstrate in Section 4, implicitization results in a prohibitive runtime cost.

To the best of our knowledge, ours is the first complete implementation that can construct arrangements of Bézier curves of any degree, and handle all degenerate cases in a robust and efficient manner.

The rest of this paper is organized as follows. Section 2 describes the functionality our traits class must provide and discusses exact algebraic computation in the context of Bézier curves. In Section 3 we give the implementation details of our Bézier-traits class, which effectively combines geometric filtering techniques with exact algebraic computation. We conclude with some experimental results in Section 4, and give some future-work directions in Section 5.

2 Preliminaries

2.1 The Arrangement-Traits Concept

Most CGAL packages employ *generic programming* techniques to achieve maximal code flexibility, and make extensive use of class-templates and template functions. A class passed as a parameter to some template should be a *model* of some *concept*, namely it should fulfil some basic set of requirements imposed by the template. See, e.g., [Austern 1998] for a review of the main generic-programming techniques.

In our case, any traits class used to instantiate the `Arrangement_2` template, should define three nested types named `Curve_2`, `X_monotone_curve_2` and `Point_2`. The distinction between a *curve* and an *x-monotone curve* is due to the fact that the former type represents a general curve that may be self-intersecting and can comprise several disconnected branches, while the latter type refers to a continuous x-monotone curve.[7] As a general curve can have a complex topology and may be too difficult to handle, most arrangement-related algorithms operate on x-monotone curves; see, e.g., [Snoeyink and Hershberger 1989]. Objects of type `Point_2` are used to represent endpoints of x-monotone curves and intersections between curves.

Any traits-class that models the arrangement-traits concept should also provide several predicates and geometric constructions involving objects of the curve and point types it defines. The only operation applied on a `Curve_2` object is to subdivide it into x-monotone subcurves. We refer to this operation as `Make_x_monotone_2`. All other traits-class functionality involves only points and x-monotone curves. The main operations are listed below.

- Compare two points by their x-coordinates only (`Compare_x_2`), or lexicographically, by their x and then — if those are equal — by their y-coordinates (`Compare_xy_2`).
- Return the xy-lexicographically smaller endpoint of a given x-monotone curve, or its larger endpoint (these operations are named `Construct_min_vertex_2` and `Construct_max_vertex_2`, respectively).
- `Compare_y_at_x_2`: Given an x-monotone curve C and a point $p = (x_0, y_0)$ such that x_0 is in the x-range of C (namely x_0 lies between the x-coordinates of C's endpoints), determine whether p is above, below, or lies on C.
- `Compare_y_at_x_right_2`: Given two x-monotone curves C_1 and C_2 that share a common left endpoint p, determine the

[7]A planar curve C is *x-monotone* if every vertical line intersects it at most once. Vertical segments are defined to be *weakly* x-monotone and can also be handled by the arrangement class.

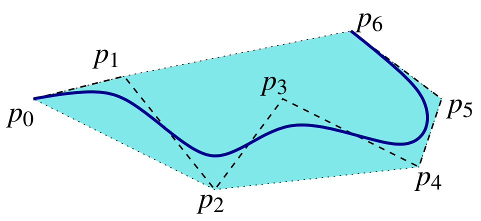


Figure 1: A Bézier curve defined by 7 control points; the control polygon is drawn with a dashed line. Note that the curve (thick solid line) lies in the convex hull of its control points $p_0,\dots,p_6$ (lightly shaded).

relative position of the two curves immediately to the right of p.

- `Intersect_2`: Compute the intersections between two given x-monotone curves C_1 and C_2, sorted in increasing lexicographical order. Each intersection object is either an intersection point, or an x-monotone curve representing an overlapping portion of C_1 and C_2.

2.2 Bézier Curves and Their Properties

A planar *Bézier curve B* is a parametric curve that is defined by a sequence of *control points* $p_0,\dots,p_n$ as follows:

$$B(t) = (X(t),Y(t)) = \sum_{k=0}^{n} p_k \cdot \frac{n!}{k!(n-k)!} \cdot t^k(1-t)^{n-k}\,, \quad (1)$$

where $t \in [0,1]$. The degree of the curve is therefore n — namely, $X(t)$ and $Y(t)$ are polynomials of degree n. In this paper, we assume that the coordinates of all control points are rational numbers,[8] so both $X(t)$ and $Y(t)$ are polynomials with rational coefficients.

The control-point sequence $p_0,\dots,p_n$ is often referred to as the *control polygon* of B. One important characteristic of Bézier curves is that the curve B is entirely contained in the convex hull of its control points; see Figure 1 for an illustration.

Another important property of Bézier curves is the *variation diminishing* property, which states that the number of intersections of any line ℓ with a Bézier curve B is not greater than the number of intersections of ℓ with B's control polygon. Note that this implies in particular that if the control polygon of B is x-monotone, then B is also x-monotone as every vertical line can intersect it at most once.

Derivatives of a Bézier curve can be determined geometrically from its control points. The first derivative of B can be expressed as a parametric Bézier curve of degree $n-1$ whose control points are given by $d_k = n(p_{k+1}-p_k)$. Note that the first derivative of a Bézier curve evaluated at $t=0$ is therefore $n(p_1-p_0)$, and similarly the derivative vector at $t=1$ is $n(p_n-p_{n-1})$.

The most fundamental algorithm for dealing with Bézier curves, which we use extensively in our application, is the de Casteljau subdivision algorithm. Given the Bézier curve B defined over $[0,1]$, we can subdivide B into two subcurves, one over the domain $0 \le t \le \tau$, and the other over the domain $\tau \le t \le 1$. Let $p_k^{(0)} = p_k$ for $0 \le k \le n$, and recursively compute, for $i = 1,2,\dots,n$ and $k = 0,1,\dots,n-i$:

$$p_k^{(i)} = (1-\tau)p_k^{(i-1)} + \tau p_{k+1}^{(i-1)}\,. \quad (2)$$

[8]When the point coordinates are given in decimal format it is easy to interpret them as rational numbers. It is also easy to convert any number in floating-point representation to a rational number, as it has a finite mantissa.

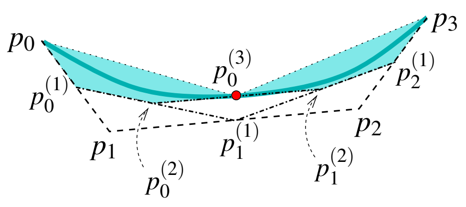


Figure 2: Applying de Casteljau's algorithm for bisecting a Bézier curve of degree 4. Note that $p_0^3 = B(\frac{1}{2})$, and the interior of the original curve $B(t)$ for $0 < t < \frac{1}{2}$ is contained in the convex hull of $p_0^{(0)},\dots,p_0^{(3)}$, while the interior of $B(t)$ for $\frac{1}{2} < t < 1$ is contained in the convex hull of $p_0^{(3)},\dots,p_3^{(0)}$.

The point $p_0^{(n)}$ obtained in the final stage equals $B(\tau)$. Moreover, the subcurve of B over the domain $0 \le t \le \tau$ is a Bézier curve defined using the control points $p_0^{(0)}, p_0^{(1)}, p_0^{(2)}, \dots, p_0^{(n)}$, and the subcurve over $\tau \le t \le 1$ is defined by $p_0^{(n)}, p_1^{(n-1)}, p_2^{(n-2)}, \dots, p_n^{(0)}$. The de Casteljau algorithm provides a way to evaluate a point $B(\tau)$ and its derivative along the curve, using basic arithmetic operations only. Especially note that if τ is rational then $B(\tau)$ is a point with rational coordinates. Furthermore, it can be shown that if the curve is repeatedly subdivided, the resulting control polygons converge to the curve; see Figure 2 for an illustration. For a more thorough introduction to Bézier curves and their properties see, e.g., [Cohen et al. 2001]. In the rest of the paper, subdivisions are performed at $\tau = \frac{1}{2}$, unless stated otherwise.

2.3 Algebraic Number Types

As we mentioned in the introduction, in order to handle non-linear curves in an exact manner, one needs to employ certified computation with algebraic numbers. A real number α is called *algebraic* if there exists a polynomial $P(x)$ with integer (or, equivalently, rational) coefficients such that $P(\alpha) = 0$. We say that α is an algebraic number of *degree d* if $d = \deg(P)$, and α is not a root of any other integer polynomial with a smaller degree.

The CORE library[9] [Karamcheti et al. 1999] and the numerical facilities of LEDA[10] [Mehlhorn and Näher 2000, Chapter 4] provide number-types for operating on algebraic numbers. Namely, it is possible to compute the kth largest root of a polynomial with integer coefficients and to apply basic arithmetic operations (namely, $+$, $-$, $\times$ and $\div$) on such roots. The computations are done in a *certified* manner, namely when we compare two algebraic numbers the result is guaranteed to be correct. Certified computation with algebraic numbers relies on the theory of separation bounds [Mignotte 1982]. A major endeavor is to use as tight a separation bound as possible in order to expedite the computation; see, e.g., [Burnikel et al. 2001; Li and Yap 2001]. Still, in many cases the estimated separation bounds are too small, incurring prohibitive running times. Thus, cascaded computations with the root operation — i.e., computing the root of a polynomial with *algebraic* coefficients — is either not supported (in CORE), or is currently too slow for most practical purposes (in LEDA).

The conic-traits class included in the arrangement package relies on CORE to handle arcs of rational *conic curves*, namely algebraic curves of degree up to 2 with rational coefficients. As the intersection-point coordinates are in this case algebraic numbers of degree 4 at most, computations in this traits class are performed

[9]`http://www.cs.nyu.edu/exact/core/`.
[10]`http://www.algorithmic-solutions.com/enleda.htm`.

rather efficiently.

Let us consider the case of Bézier curves. As mentioned earlier, we assume that all control points have rational coordinates, so the endpoints of the input curves are rational points. However, these curves should be subdivided into x-monotone subcurves. Given a curve $B(t) = (X(t), Y(t))$, we compute the roots $t_1 < t_2 < \ldots < t_k$ of $X'(t)$ that lie in $[0,1]$, such that the x-monotone subcurves are defined over the parameter intervals $[0,t_1], [t_1,t_2], \ldots, [t_k,1]$. The endpoints of these intervals are given by $B(t_i)$, where t_i $(1 \le i \le k)$ is an algebraic number of degree $\deg(B) - 1$.

Obtaining the intersection points between two Bézier curves $B_1(s) = (X_1(s), Y_1(s))$ and $B_2(t) = (X_2(t), Y_2(t))$ is more involved. In this case we have to compute the roots of the following polynomial system:

$$\left\{ \begin{array}{lcl} X_1(s) & = & X_2(t) \\ Y_1(s) & = & Y_2(t) \end{array} \right. . \tag{3}$$

This system of equations can be solved in the parameter space by calculating the roots of the resultant[11] polynomials $\mathrm{res}_t\left(X_1(t) - X_2(s), Y_1(t) - Y_2(s)\right)$, yielding the s-values, and $\mathrm{res}_s\left(X_1(t) - X_2(s), Y_1(t) - Y_2(s)\right)$, which gives us the t-values. Only solution pairs of the form $\langle s_0, t_0 \rangle$ (i.e., $B_1(s_0) = B_2(t_0)$), such that $s_0 \in [0,1]$ and $t_0 \in [0,1]$, are considered to be valid.

A Bézier curve may also be self-intersecting. A self-intersection point occurs if there exists $s_0, t_0 \in [0,1]$ such that $(X(s_0), Y(s_0)) = (X(t_0), Y(t_0))$. If we denote $X(t) = \sum_{k=0}^{n} \xi_k t^k$ and $Y(t) = \sum_{k=0}^{n} \eta_k t^k$, we have to solve the following system of polynomial equations:

$$\left\{ \begin{array}{l} \sum_{k=1}^{n} \xi_k (t^k - s^k) = 0 \\ \sum_{k=1}^{n} \eta_k (t^k - s^k) = 0 \end{array} \right. .$$

Note that all terms in the equation above are divisible by $(t - s)$. Dividing by $(t - s)$, we eliminate the trivial solution $s = t$ and obtain the solutions for the following system:

$$\left\{ \begin{array}{l} \sum_{k=1}^{n} \xi_k \left(\sum_{i=0}^{k} s^i t^{k-i} \right) = 0 \\ \sum_{k=1}^{n} \eta_k \left(\sum_{i=0}^{k} s^i t^{k-i} \right) = 0 \end{array} \right. . \tag{4}$$

The main problem is that the parameter-space solutions of Equation (3) are algebraic numbers of degree $\deg(B_1) \cdot \deg(B_2)$ (and of degree $(\deg(B_1) - 1)^2$ in case of a self-intersection). The complexity of the intersection-point coordinates, obtained by substituting these solutions into the curve equations, is even larger. As the efficiency of exact number-types quickly deteriorates as the degree of the number it handles increases, we try to avoid exact computations and only use them when absolutely necessary.

2.4 High-level Filtering

Several arrangement traits-classes for algebraic curves have been developed as part of the EXACUS project [Berberich et al. 2005a], among them we mention a traits class for cubic curves [Eigenwillig et al. 2004], and another that handles special types of curves of degree four [Berberich et al. 2005b]. The traits classes consider only *implicit* algebraic curves, defined as the zero-set of some bivariate rational polynomial with bounded degree, namely curves given by the equation $C(x,y) = 0$, where C is a bivariate polynomial with rational coefficients. They apply algebraic filtering techniques to efficiently manipulate such curves. Instead of explicitly computing the intersection points, each point is isolated in some small bounding box with rational coordinates, where the boundaries of the box are computed by isolating the roots of a univariate rational polynomial whose roots are the x-coordinates of the intersection point (namely, the resultant polynomial $\mathrm{res}_y(C_1, C_2)$). It is easy to compare the coordinates of two points in this representation if their isolating boxes are disjoint. Otherwise, it is possible to refine the isolating boxes by considering the Sturm sequences of the polynomials that induce their coordinates. Degenerate cases are determined by considering the greatest common divisor (GCD) of these polynomials, as two polynomial roots are equal only if the polynomials have a nontrivial GCD.

While algebraic filtering methods are very effective, they still make use of algebraic methods, such as resultant computation and univariate root-isolation involving polynomials with exact rational coefficients, which are applied to every pair of intersecting (or potentially intersecting) curves. In the case of the Bézier traits class we use similar ideas, namely we isolate intersection points inside bounding boxes, but also take advantage of the geometric properties of Bézier curves to construct these isolating boxes without applying any exact algebraic procedure whatsoever. Instead, we use bisection techniques that are all implemented using machine-precision or rational arithmetic, working on linear geometric objects and applying only rational operations. Only in cases where the rational procedures fail to give decisive results do we perform resultant computation and obtain the intersections in an exact algebraic manner.

3 The Bézier-Traits Class

The implementation of our Bézier-traits class consists of a layer that operates on bounding polygons, which serves as a filter for the exact algebraic layer. In this section we show how the representation of points and curves enables us to utilize the geometric properties of the Bézier curves (e.g., the subdivision and convex-hull properties) such that the computationally demanding algebraic methods are mainly invoked for handling degenerate cases, where the geometric properties fail to give conclusive results.

3.1 Representing Points and Curves

In Section 2.1 we explained that any traits class must define the nested types `Curve_2`, `X_monotone_curve_2` and `Point_2`. We next describe the structure of the nested types defined by our Bézier-traits class.

`Curve_2`: A Bézier curve B is uniquely characterized by its control polygon. The curve class therefore stores the control points $p_0, p_1, \ldots, p_n$ of B, that implicitly refer to the parameter domain $t \in [0,1]$. The polynomials $X(t)$ and $Y(t)$ can be deduced from the control polygon at any time, according to Equation (1), and are therefore computed only when needed by applying "lazy evaluation".

`Point_2`: As explained in Section 2.3, there are three types of points in our context: (i) input points with rational coordinates, (ii) split points of a curve into x-monotone subcurves, and (iii) intersection points. Points of the latter two types are characterized by some algebraic parameter value and their coordinates are algebraic numbers of a relatively high degree, thus computing them exactly is too costly for most practical applications.

The point is therefore represented by a list of references to curves that are incident to it (the *originating curves*, or *originators* for short), coupled with the corresponding parameter value on each curve. The exact value of this parameter may not be known, as we explain next. Note that this representation can handle degenerate situations such as multiple curves intersecting at a point, or a

[11]The *resultant* of two bivariate polynomials $P_1(x,y)$ and $P_2(x,y)$ with respect to y, denoted $\mathrm{res}_y(P_1, P_2)$, is a univariate polynomial in x.

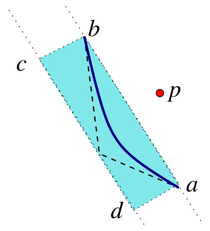


Figure 3: A Bézier curve (solid line) given by three control points. The control polygon is drawn with a dashed line. The skewed bounding box of the curve is lightly shaded. Note that the point p is to the right of the box since it lies to the right of both lines supporting $\overline{ab}$ and $\overline{dc}$.

coincidence of an endpoint and an intersection point.

For every reference to an originating curve we also store a small control polygon, resulting from repeated subdivision of that curve around the point. The control polygon bounds the point on the curve in some parameter sub-domain $[t', t'']$ (where $t'' - t' = 2^{-m}$, m being the number of subdivisions performed on the original curve). Because of the convex-hull property, the control polygon enables us to isolate the point by some small bounding box, namely the bounding box of its control points. The true coordinates of the point p are thus isolated in the intervals $[x_{\min}(p), x_{\max}(p)]$ and $[y_{\min}(p), y_{\max}(p)]$, respectively. The control polygon can be further refined upon request, yielding a tighter isolating box for the point. The exact algebraic coordinates of the point are evaluated only in a lazy manner to resolve ambiguities caused in degenerate or near-degenerate situations.

`X_monotone_curve_2`: An x-monotone portion of a Bézier curve is represented by references to its supporting curve (of type `Curve_2`) and to its endpoints. The usage of references make it trivial, for example, to check whether two x-monotone subcurves originate from the same curve.

3.2 Traits-Class Operations

We next explain how the traits class implements the operations listed in Section 2.1 on objects of the types defined above. We begin by describing the construction operations, namely the subdivision of curves into x-monotone subcurves and the computation of intersection points, and then give details of the various predicates that involve the constructed objects, paying special attention to the fact that these objects are lazily evaluated.

3.2.1 Construction Operations

`Make_x_monotone_2`: Given a curve $B(t) = (X(t), Y(t))$ defined for $t \in [0, 1]$, we can subdivide it into x-monotone subcurves by identifying all points $B(t_0)$ having a vertical tangent (namely $X'(t_0) = 0$). As we wish to avoid the computation of the polynomial roots in a precise manner, we use the variation diminishing property of Bézier curves, stating that an x-monotone control polygon guarantees that the curve is x-monotone. Thus, we recursively subdivide the curve B using de Casteljau's algorithm, purging away subcurves with x-monotone control polygons. Furthermore, a convex y-monotone control polygon guarantees that there is a single vertical-tangency point on the curve, so we can terminate the subdivision when the subcurve has a y-monotone convex control polygon and isolate the vertical-tangency point within the bounding box of this polygon.

The subdivision procedure described above encounters a degeneracy problem in the presence of y-coordinate inflection points. The control polygon in this case will always be non-convex around the point, so it is impossible to distinguish between two vertical-tangency points lying close together and between an inflection point. When we reach the precision limit without being able to isolate the vertical-tangency points, we therefore terminate the subdivision process and resort to the exact algebraic procedure of evaluating the roots of $X'(t)$. We mention that the subdivision algorithm will also fail in the degenerate case where the curve contains a cusp, yet the exact procedure will correctly identify this cusp, as at this point we have $X'(t_0) = Y'(t_0) = 0$.

In the remaining part of this section we consider only x-monotone Bézier curves, characterized by a curve $B(t)$ that is defined on some continuous range of t-values. We denote such subcurves with the same notation used for their supporting curves.

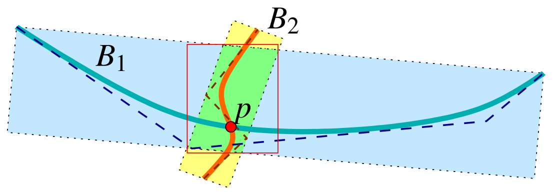
`Intersect_2`: To perform the parameter-space intersection of two curves B_1 and B_2 in a precise algebraic manner we have to compute the roots of the resultant polynomials $\operatorname{res}(X_1(s) - X_2(t), Y_1(s) - Y_2(t))$, but this process is computationally expensive and we wish to avoid it whenever possible. We therefore use the subdivision and convex-hull properties of Bézier curves to isolate intersection points. The idea is to isolate intersection points by recursively subdividing B_1 and B_2, purging away subcurves that do not intersect by the convex-hull property. The subdivision termination criteria is based on the "bounding cones" of the curves [Sederberg and Meyers 1988].

Given a curve $B(t) = (X(t), Y(t))$, its *bounding cone* is the cone spanned by the angles of its derivatives $B'(t) = (X'(t), Y'(t))$. The bounding cone is easy to compute for Bézier and spline curves, since the tangent curve $B'(t)$ can be computed directly from the control polygon by taking the differences of the control polygon of B $(p_1 - p_0, p_2 - p_1, \ldots, p_n - p_{n-1})$ as the control polygon of B' (as we are only interested in the direction of the derivatives, we do not need to scale these differences by n, as explained in Section 2.2). We can therefore easily compute the bounding cones of the two curves B_1 and B_2 and check whether they intersect only at the origin. If so, then the curves can intersect at most once (see [Sederberg and Meyers 1988] for the details). Otherwise, we have to bisect both curves using de Casteljau's algorithm and continue recursively with the four pairs of subcurves we generate.

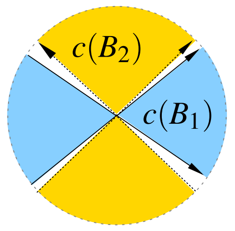
In our implementation, we calculate the bounding cones of the relevant subcurves of B_1 and B_2 in each subdivision step. Once we obtain disjoint bounding cones we know that there can be at most one intersection point, and we just have to check whether such an intersection really exists. This is done by considering the skewed bounding boxes of the curves (similar to the "fat lines" described in [Sederberg and Nishita 1990]). A *skewed bounding box* of a curve is a bounding box oriented according to the line connecting the two curve endpoints. We say that a point is on the left (resp. right) of a skewed bounding box if it is to the left (resp. right) of both segments that are parallel to the line between the curve's start and end points. See Figure 3 for an illustration.

Proposition 1 *Given two curves B_1 and B_2 and their skewed bounding boxes Σ_1 and Σ_2, if the start and end points of B_1 are on opposite sides of Σ_2 (i.e., one is to the left and the other to the right of Σ_2), and further, the start and end points of B_2 are on opposite sides of Σ_1, then there must be at least one intersection of the curves within the area of intersection of the boxes.*

Figure 4 displays the idea of Proposition 1 on two Bézier curves and their corresponding skewed bounding boxes. We check for the condition defined in Proposition 1 and if it is satisfied, we have isolated a single intersection point. If the condition is not satisfied, we continue with the subdivision. Given a skewed bounding box, checking whether the condition is satisfied can be done exactly us-



(a)



(b)

Figure 4: (a) Transversely intersecting Bézier curves, shown with their control polygons and skewed bounding boxes. The axes-aligned isolating box of the intersection point p is drawn in a thin solid line. Note that the endpoints of B_1 lie on opposite sides of the skewed bounding box of B_2 and similarly the endpoints of B_2 lie on opposite sides of the skewed bounding box of B_1 (see Proposition 1). (b) The bounding cones of the curves.

ing the orientation predicate. Computing the skewed bounding box of a subcurve is done by projecting its control points on the line that connects the first point and the last point, then adding the largest projection vectors to the endpoints to get the corners of the skewed bounding box. The projection of a point on a line is a rational operation and can therefore be computed exactly using rational numbers.

Having isolated an intersection point using two subcurves, we can construct a representation of the intersection point, bounding it in the skewed bounding box of the two intersecting subcurves, without the need to construct the resultant polynomials explicitly. While this geometric subdivision-based isolation procedure will terminate for transversal intersection, it will fail for degenerate tangential intersections. For these cases, the construction of the resultant and the computation of its roots is required. We therefore terminate the subdivision method and move to the algebraic method when the subdivision passes some bound. The actual subdivision termination criteria depends on the implementation. For rational number types we terminate the subdivision at a user-defined bound. In the future, we plan to have an implementation of the traits class that uses interval arithmetic (see Section 5). For such interval-arithmetic number types the criterion for subdivision termination will be when the bounding intervals do not enable a verified comparison.

Note that the procedure of bounding the intersection points may fail in the following cases: (i) there are two intersection points lying very close together, (ii) there exists an intersection point whose multiplicity is greater than 1, or (iii) the curves overlap. The exact computation can readily tell us whether B_1 and B_2 overlap, as the resultants $\mathrm{res}\,(X_1(s) - X_2(t), Y_1(s) - Y_2(t))$ are identically zero in this case. However, as we wish to avoid the resultant computation, we use Bézout's theorem,[12] which states that non-overlapping curves can meet in at most $M = \deg(B_1) \cdot \deg(B_2)$ points in $\mathbb{R}^2$. We therefore sample $M+1$ points with rational coordinates on B_1 by computing $p_k = B(\frac{k}{M})$ for $0 \le k \le M$ and check whether they all lie on B_2. This check is rather simple for point with rational coordinates as it involves computing the zeros of a univariate polynomial with rational coefficients. Note that if the curves are not identical, it is usually sufficient to compare just a single point. In the degenerate case of an overlap we need as many as $M+1$ checks.

3.2.2 Traits-Class Predicates

As described in Section 3.1, every point whose coordinates are not precisely known stores small control polygons, whose bounding box isolates the point coordinates. The `Compare_x_2` predicate is therefore implemented as follows. Given two points p_1 and p_2, check if $x_{\max}(p_1) < x_{\min}(p_2)$ or whether $x_{\min}(p_1) > x_{\max}(p_2)$, in which case we are done. Otherwise, we apply some refinement steps on the bounding boxes (see the previous subsection) until their x-ranges do not overlap any more. If we still fail, we either have a degeneracy, or a near-degeneracy that cannot be determined without using algebraic arithmetic. In this case we use the originators of each point to compute its exact representation, and perform an exact comparison to obtain a guaranteed result. Similar techniques also apply for the `Compare_xy_2` predicate.

As we see, the exact computation of a pair of curves is done only when their intersection point is involved in a degenerate or near-degenerate situation. However, two Bézier curves usually have many intersection points, some of which may be involved in near-degeneracies. The traits class therefore caches the intersections it has computed in an exact manner, namely it maps each pair of curves to a list of their *exact* intersection points. Similarly, it also caches the split points of a single curve into x-monotone subcurves, in case they have been exactly evaluated. When we have to obtain the exact representation of a point, we first look for it in the cache, and perform the computation only if we failed to find it there.

The predicate `Compare_y_at_x_2` is perhaps the most difficult to implement. Assume that we know that the point p is not on the x-monotone Bézier curve B. We can then subdivide the curve using de Casteljau's algorithm until the bounding box of the control polygon (and hence the convex hull of the control polygon) is totally above or totally below the point p (or above/below the isolating box of this point). The convergence of the subdivision procedure guarantees that the procedure terminates.

The difficulty lies in the fact that deciding whether a point $p = (x_0, y_0)$ is on a parametric B curve is not an easy task. For a rational point this amounts to extracting the root t_0 of the polynomial $X(t) - x_0 = 0$ that lies in the parameter range of B (recall that the curve is x-monotone, so there exists a unique solution with this property), and then checking whether $Y(t_0) = y_0$. However, in case the point has algebraic coordinates, the free coefficient of $X(t) - x_0 = 0$ is algebraic, and we cannot obtain an exact representation of the t-value (see the discussion in Section 2.3).

A possible solution is to implicitize the curve [Sederberg and Parry 1986] and check whether the point (x_0, y_0) satisfies the implicit representation. This process, however, requires to maintain the curve's implicit representation after computing the resultant $\mathrm{res}_t\,(X(t), Y(t))$. We implemented a different approach, which checks for the intersection of the curve B with one of the originating curves of p. Let B_p be the originating curve of p and s_p be the

[12] See, e.g, E. W. Weisstein, "Bézout's Theorem", from MathWorld — a Wolfram web resource:
`http://mathworld.wolfram.com/BezoutsTheorem.html`.

parameter value such that $p = B_p(s_p)$ (recall that this information is stored with the point). If an intersection exists and one of the intersection parameters s of B_p is identical to s_p, then p is on B. To simplify the computation we choose B_p as the minimal-degree originator of p.

The drawback of this procedure is the need to intersect B and B_p even if the construction of the arrangement does not require to intersect them (e.g., B and B_p never become neighbors on the status line when applying the sweep-line algorithm). We therefore use the following strategy. First, we compare the bounding boxes of B and p and perform some refinement steps until we manage to separate their y-range, or until reaching our precision bound. If we manage to separate the bounding boxes, we can carry out the comparison quite cheaply. Otherwise we have to resort to exact methods and test whether p lies on B, as explained above. If it does, then we have a decisive result and we are done. Otherwise, we continue to refine the bounding boxes, this time using exact rational arithmetic with no precision bound. As p does not lie on B, this process will eventually yield the correct comparison result.

The implementation of the `Compare_y_at_x_right_2` predicate is also intricate when handling a degenerate case. We are given a point p, which is an intersection of B_1 and B_2, such that $B_1(s_0) = B_2(t_0)$. The parameter values s_0 and t_0, or at least their approximate representation, are stored in p's originator list. If p is a simple transversal intersection point of B_1 and B_2, their order to the right of p can be easily determined by comparing the directions of the tangent vectors $\left(X_1'(s_0), Y_1'(s_0)\right)$ and $\left(X_2'(t_0), Y_2'(t_0)\right)$. The bounding-cones algorithm we use to isolate the intersection points also gives us a guarantee that these directions are well separated, even if s_0 and t_0 are only given as approximate intervals.

In degenerate cases, we may have intersection points whose multiplicity is greater than 1. One can try comparing the curvatures of the curves, but this becomes cumbersome and for curves with the same curvature at p, a higher degree comparison is hard to formulate. We therefore use a different approach that makes use of the fact that sampling dense rational points on a parametric curve is relatively easy. We first check whether B_1 and B_2 intersect to the right of p or not. In the former case, we can compare the curves at a point between p and the nearest intersection point to the right of p. In the latter case, we can compare the curves at some point between p and the right endpoint that is closer to p. The comparison itself is done by choosing a rational point $(\hat{x}, \hat{y})$ on B_1 and checking whether it is above or below B_2. This can be done by solving $X_2(t) = \hat{x}$ and evaluating $Y_2(t_0)$ at the resulting parameter t_0.

4 Experimental Results

The running times reported in this section were obtained on a 3 GHz Pentium IV machine with 2 GB of RAM, running a Linux operating system. The software was compiled using the Gnu C++ compiler (g++ Version 3.3.5).

We have implemented two versions of a traits class that can handle Bézier curves. The first employs the exact algebraic procedures described in Section 2.3 in all constructions and predicates, without using any geometric filtering. We refer to it as the "exact Bézier-traits class". The second class uses a filtering layer of bounding polygons to avoid exact algebraic computation, as described in Section 3. In the filtering layer we employ exact rational computations, with a precision bound of 2^{-53}. We refer to this class as the "filtered Bézier-traits class".

The first set of experiments we describe involves the construction of the arrangement of a set of Bézier curves of degree 2, given by three control points. A degree 2 Bézier curve is always a parabolic arc, and can thus be also represented by the traits class for conic arcs included in the arrangement package. Table 1 compares the construction times of arrangements of random parabolic arcs (the data sets contain $10, 20, 30$ and 40 curves, respectively), each given by three control points within the unit square. The complexity of each arrangement (number of vertices, edges and faces) is also given in the table. Figure 5 shows two of the arrangements.

We compare the conic-traits class with the two versions of the Bézier-traits class, the exact and the filtered one. The inferior performance of the conic-traits class is due to the fact that it has to compute the supporting parabola by considering five points on the curve, resulting in a rather complex representation. Also note that the gain achieved by filtering is not large in this case, as we only have to compute algebraic numbers of a relatively small degree. In order to compare our exact methods with traditional floating-point methods, we have also implemented a traits class that serves as a wrapper for the procedures that handle Bézier curves in the IRIT modeling environment.[13] IRIT performs all geometric operations using floating-point arithmetic, hence it is very fast compared to the exact methods we use. However, it may be instable in near-degenerate scenarios. For example, on the *parabolas_40* random input (see Table 1), the floating-point traits class produced incorrect results in some of the predicates, which caused the software to crash. Indeed, in this case we have a near-degenerate scenario, where our traits class needs to resort to exact computation for three different intersection points.

The main feature of our scheme for handling Bézier curves is that it produces topologically correct results in case of degeneracies. Consider for instance the example illustrated in Figure 7, where we have four curves (three parabolas and one line segment) all intersecting at a common point. Moreover, two of the parabolas are tangent to one another at this point. We are able to identify this degenerate situation and correctly construct the arrangement, which consists of 10 vertices, 12 edges and 4 faces in this case. The arrangement construction takes about 0.1 seconds. We are not aware of any other software that is capable of handling such a degenerate configuration of Bézier curves in a robust manner.

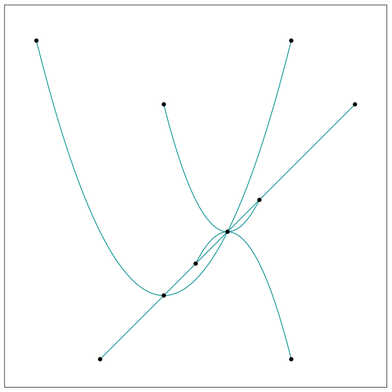
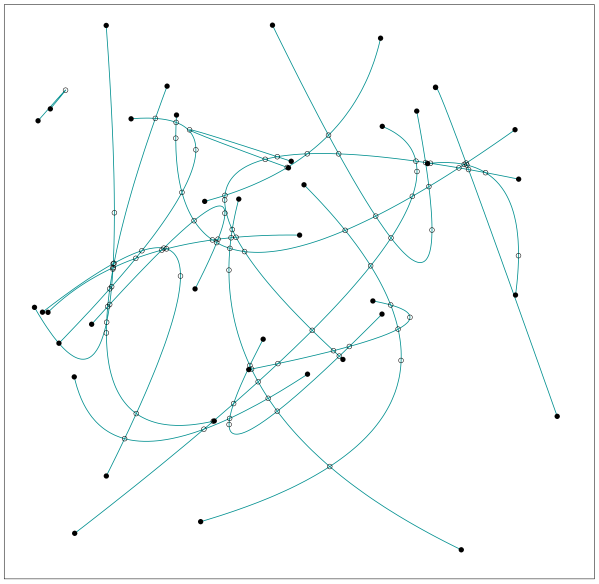


Figure 7: An arrangement induced by four Bézier curves in degenerate position.

Figure 8 shows another degenerate scenario that our traits classes can successfully handle. A Bézier curve of degree 2 passes through the self-intersection point of a degree-3 curve. This forces us to compute the self-intersection point in an exact manner, as explained in Section 2.3.

[13]See, G. Elber, "The IRIT modeling environment": `http://www.cs.technion.ac.il/~irit/`.


(a)

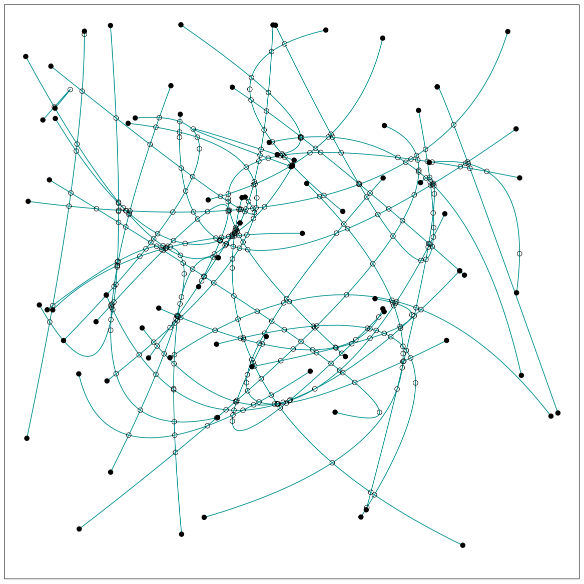

(b)

Figure 5: Arrangements induced by random parabolic arcs, given as Bézier curves of degree 2: (a) the *parabolas_20* data set, (b) the *parabolas_40* data set. Points that are not computed in an exact manner and given by their bounding box are drawn as small circles. Other points (usually endpoints given as part of the input) are drawn as small discs.

Table 1: Constructing the arrangements of random parabolas using different traits classes. Times are given in seconds.

Input File	Arrangement size			IRIT floating-point traits	Conic-arc traits	Exact Bézier traits	Filtered Bézier traits
	$\|V\|$	$\|E\|$	$\|F\|$				
parabolas_10	57	75	20	0.007	20.149	0.332	0.240
parabolas_20	125	173	51	0.012	57.407	1.060	0.632
parabolas_30	238	362	127	0.023	126.024	2.380	1.576
parabolas_40	422	697	277	**N/A**	232.315	4.385	3.752

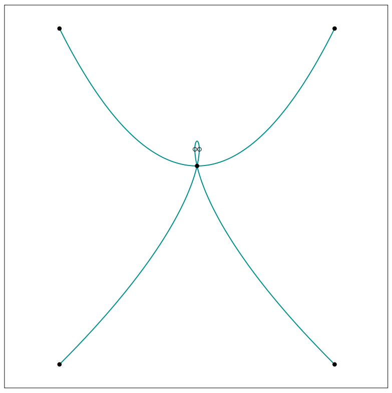


Figure 8: An arrangement containing self-intersections.

The effectiveness of filtering is emphasized when working with Bézier curves of larger degrees. Table 2 shows the construction times of arrangements of Bézier curves of degree 3 up to 5.[14] In this case we used hand-drawn inputs, where the control points were all specified on the integer grid $[0,10000] \times [0,10000]$. Each input file contains 10 Bézier curves of the same degree. The induced arrangements are shown in Figure 6. Observe that the filtered Bézier-traits class now runs 10–250 times faster than the exact Bézier-traits class. Also note that the runtime overhead of using our filtered traits class, when compared to using the IRIT floating-point traits class, was reduced in Table 2 in comparison to Table 1 (10–25 times slower in Table 2 compared to 30–70 times slower in Table 1). The reason is that the input in Table 1 was given in floating-point numbers, which were converted to long rational numbers, whereas the input in Table 2 was given in integers. For the floating-point computations this did not make a difference, but for rational number types integer input is easier to handle.

Table 2: Constructing the arrangements of Bézier curves of higher degrees. Times are given in seconds.

Input File	Arrangement size			IRIT traits	Exact traits	Filtered traits
	$\|V\|$	$\|E\|$	$\|F\|$			
deg_3	40	47	9	0.004	1.220	0.112
deg_4	52	64	14	0.016	7.916	0.192
deg_5	50	59	12	0.020	51.079	0.244

As we have already mentioned in the introduction, the arrangement package of CGAL serves as an infrastructure for several related packages, an important one being the Boolean set-operations package. That is, using our traits class we can compute regular-

[14]In the next set of experiments we also consider curves of degree 6. While the filtered Bézier-traits class can efficiently handle curves of higher degrees, the running times for the exact traits class usually explode, so we do not bring experimental results with curves of higher degrees.

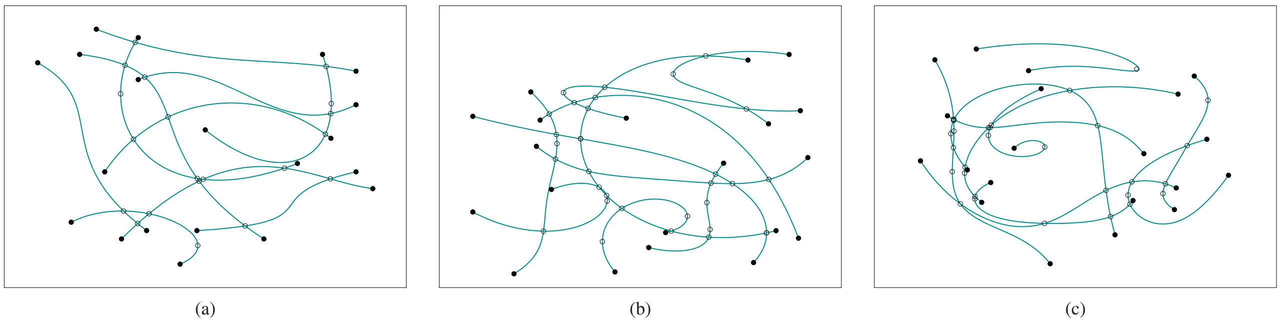


Figure 6: Arrangements induced by sets of 10 Bézier curves of different degrees: (a) degree 3, (b) degree 4 and (c) degree 5. Points that are not computed in an exact manner and given by their bounding box are drawn as small circles. Other points are drawn as small discs.

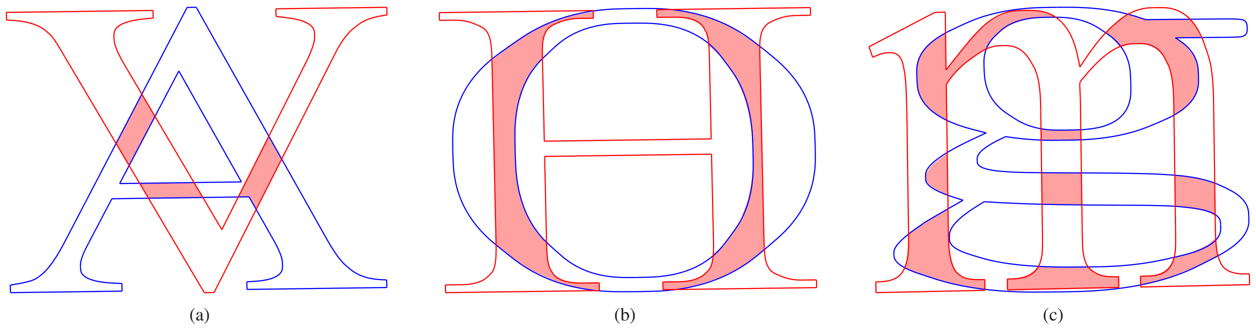


Figure 9: Computing the intersection of characters in the Times New Roman font: (a) *A* and *V*, (b) *H* and *O*, (c) *m* and *g*. The intersecting regions in each case are lightly shaded.

ized Boolean operations on general polygons whose boundaries are given as closed sequences of Bézier curves.

In our experiments we used characters taken from the *Times New Roman* font. These characters are given as sequences of Bézier curves of degrees 3 and 6 that specify the outer boundary of the characters and the boundary of the holes inside it (e.g., the character *B* has two holes, while *V* is simple and has no holes). In Table 3 we give the statistics of intersecting pairs of characters. For each pair ch_1, ch_2 we give the number of curves that describe their boundaries (n_1 and n_2, respectively), as well as the sizes of the general polygons that comprise the intersection $ch_1 \cap ch_2$. We use the exact Bézier-traits class and the filtered traits class and compare the running times each class achieves. Figure 9 shows some of the character pairs we used in our experiments.

Note that we separate between T_{init}, the time needed to construct the two input polygons, and T_{comp}, the time needed to compute the intersection by overlaying the two arrangements that correspond to ch_1 and ch_2. While the filtered Bézier-traits class performs the overlay 100–1000 times faster than the exact traits class, it also performs the initialization much faster. This is due to the fact that it does not construct the explicit polynomial representation of the Bézier curve and just stores the sequence of control points per curve.

5 Conclusions and Future Work

We have presented a first implementation of a complete and robust arrangement of Bézier curves of any degree. Our approach combines exact algebraic techniques with efficient methods based on the geometric properties of Bézier curves. Thus, we manage to handle all degenerate and near-degenerate situations robustly, while maintaining a relatively low runtime overhead. We have shown the value and efficiency of our approach by comparing it to exact approaches and to traditional floating-point implementations. The usefulness of our work was demonstrated with a Boolean operation application on generalized polygons bounded by Bézier curves. We plan to make our work publicly available within the CGAL library.

The work presented in this paper can be extended in several directions. The efficiency of the geometric filtering techniques can be improved by trying different representations of the curve's control polygon. Currently we use the Cartesian kernel of CGAL with the GMP rational number-type for manipulating the control polygon. Implementing the geometric layer with different kernels that use filtering may speed up the computations in this layer. The geometric layer can also work with other number types. We plan to implement the geometric layer using interval arithmetic and extended floating point numbers, and we expect this to speed-up the application considerably, though there might be cases for which rational arithmetic will prove superior.

The geometric algorithms used by the filtering layer may also be improved. Yap [2006] has recently presented an adaptive subdivision algorithm for complete Bézier curve intersection, which has similarities to the work descrived in this paper. The main difference is the use of of a geometric separation bound to terminate the subdivision without resorting to manipulation of algebraic numbers. We consider incorporating Yap's algorithm in future implementations of the intersection function within our traits class. While a curve-intersection algorithm is a first step in constructing arrangements, other functions are also needed, as we have shown. Further research in the direction of geometric separation bounds may provide tools to reduce further the use of algebraic number-types in our application.

Our implementation currently deals only with Bézier curves. A natural extension of our application is the support of spline curves.

Table 3: Computing the intersection of Times New Roman characters, given as Bézier polygons. Times are given in seconds.

					Exact traits		Filtered traits	
ch_1	n_1	ch_2	n_2	Result size	T_{init}	T_{comp}	T_{init}	T_{comp}
A	23	V	20	4, 4, 5	0.105	17.4	0.028	0.154
H	48	O	16	21, 21	0.195	174.9	0.032	0.325
S	35	Z	15	6, 19, 24	0.382	367.7	0.030	0.502
m	52	g	43	4,11,13,6,5,9,14,5,7	0.402	543.6	0.072	0.532
g	43	s	29	31, 29	0.360	737.8	0.069	0.734

Given a spline representation, we can construct an equivalent sequence of contiguous Bézier curves (a Bézier polycurve) through repeated *knot insertion* [Cohen et al. 2001, Chapter 17]. Since knot insertion does not involve algebraic computations, this can be done in a straightforward manner. Thus, by preprocessing the input spline curves, our application can support arrangements of splines. Alternatively, since the spline representation has similar properties to Bézier curves (e.g., the convex hull and the variation diminishing properties), we can implement the geometric filtering layer directly on the spline representation, and obtain the Bézier polycurve representation only by lazy evaluation, when we have to resort to exact computation.

Another extension is to implement arrangements of rational Bézier curves. A *rational Bézier curve* of degree n is defined by a sequence of $n+1$ weighted control points, namely $\langle p_0, w_0\rangle, \langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle$ as follows:

$$B(t) = (X(t), Y(t)) = \frac{\sum_{k=0}^{n} w_k p_k \cdot \frac{n!}{k!(n-k)!} \cdot t^k (1-t)^{n-k}}{\sum_{k=0}^{n} w_k \cdot \frac{n!}{k!(n-k)!} \cdot t^k (1-t)^{n-k}} \,. \quad (5)$$

Rational Bézier curves are useful in many applications. In particular, they are used to represent conic sections. Since many of the geometric Bézier properties that we use apply to rational Bézier curves as well, the extension is natural.

While extending the arrangement to rational Bézier curves with rational-number coordinates[15] is straightforward, in many practical cases the coordinates of the weighted control points are algebraic. For example, a common representation for circular arcs contains square roots in the weights of the control points — namely, the w_k coefficients are algebraic numbers of degree 2.[16] Our current framework, however, supports only inputs containing rational numbers. In order to support a more useful implementation of rational Bézier curves, we therefore need to extend our framework to handle square roots as well. Since these are algebraic numbers of a low degree, they can be handled more efficiently than general algebraic numbers. However, this is left for future work.

Acknowledgements

The first author of this paper has been supported in part by the Israel Science Foundation (grant No. 857/04) and by European FP6 NoE grant 506766 (AIM@SHAPE).

The second author has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes), by the Israel Science Foundation (grant no. 236/06), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

The authors wish to thank Gershon Elber and Dan Halperin for their useful comments and encouragement.

[15]We use the term rational-number coordinates to denote that the number-type of the coordinates is rational and not algebraic. This should not be confused with *weighted control points*, which are sometimes referred to as rational control points.

[16]See [Cohen et al. 2001, Chapter 3] for a formulation of conics as rational functions, which explains why they contain roots in their weights when representing a circular arc.

References

AGARWAL, P. K., AND SHARIR, M. 2000. Arrangements and their applications. In *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publishers B.V., 49–119.

AUSTERN, M. H. 1998. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley.

BERBERICH, E., EIGENWILLIG, A., HEMMER, M., HERT, S., KETTNER, L., MEHLHORN, K., REICHEL, J., SCHMITT, S., SCHÖMER, E., AND WOLPERT, N. 2005. EXACUS: Efficient and exact algorithms for curves and surfaces. In *Proc. 13th Europ. Sympos. Alg. (ESA)*, Springer, vol. **3669** of *LNCS*, 155–166.

BERBERICH, E., HEMMER, M., KETTNER, L., SCHÖMER, E., AND WOLPERT, N. 2005. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *Proc. 21st Annu. ACM Sympos. Comput. Geom. (SCG)*, 99–106.

BURNIKEL, C., FUNKE, S., MEHLHORN, K., SCHIRRA, S., AND SCHMITT, S. 2001. A separation bound for real algebraic expressions. In *Proc. 9th Europ. Sympos. Alg. (ESA)*, Springer, vol. **2161** of *LNCS*, 254–265.

COHEN, E., ELBER, G., AND RIESENFELD, R. F. 2001. *Geometric Modeling with Splines: An Introduction*. A. K. Peters.

EIGENWILLIG, A., KETTNER, L., SCHÖMER, E., AND WOLPERT, N. 2004. Complete, exact and efficient computations with cubic curves. In *Proc. 20th Annu. ACM Sympos. Comput. Geom. (SCG)*, 409–418.

FOGEL, E., HALPERIN, D., KETTNER, L., TEILLAUD, M., WEIN, R., AND WOLPERT, N. 2006. Arrangements. In *Effective Computational Geometry for Curves and Surfaces*, J.-D. Boissonnat and M. Teillaud, Eds. Springer, ch. **1**, 1–66.

FOGEL, E., WEIN, R., ZUKERMAN, B., AND HALPERIN, D. 2006. 2D regularized boolean set-operations. In CGAL-*3.2 User and Reference Manual*, CGAL Editorial Board, Ed. `http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Boolean_set_operations_2/Chapter_main.html`.

HALPERIN, D. 2004. Arrangements. In *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds., 2nd ed. Chapman & Hall/CRC, ch. **24**, 529–562.

HANNIEL, I., AND HALPERIN, D. 2000. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 14th Workshop Alg. Eng. (WAE)*, Springer, vol. **1982** of *LNCS*, 171–182.

HOFFMANN, C. M. 1989. The problems of accuracy and robustness in geometric computation. *IEEE Computer* **22**, 3 (March), 31–41.

HOFFMANN, C. M. 2001. Robustness in geometric computations. *J. Computing and Information Science in Engineering* **1**, 2, 143–155.

KARAMCHETI, V., LI, C., PECHTCHANSKI, I., AND YAP, C. 1999. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom. (SCG)*, 351–359.

KETTNER, L., MEHLHORN, K., PION, S., SCHIRRA, S., AND YAP, C. 2004. Classroom examples of robustness problems in geometric computations. In *Proc. 12th Europ. Sympos. Alg. (ESA)*, Springer, vol. **3221** of *LNCS*, 702–713.

KEYSER, J., CULVER, T., MANOCHA, D., AND KRISHNAN, S. 2000. Efficient and exact manipulation of algebraic points and curves. *Computer-Aided Design* **32**, 11, 649–662.

LI, C., AND YAP, C. 2001. New constructive root bound for algebraic expressions. In *Proc. 12th ACM-SIAM Symp. Disc. Alg. (SODA)*, 496–505.

MEHLHORN, K., AND NÄHER, S. 2000. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press.

MIGNOTTE, M. 1982. Identification of algebraic numbers. *J. Algorithms* **3**, 3, 197–204.

MYERS, N. 1997. A new and useful template technique: “Traits”. In *C++ Gems*, S. B. Lippman, Ed., vol. **5** of *SIGS Reference Library*. 451–458.

NEAGU, M., AND LACOLLE, B. 1998. Computing the combinatorial structure of arrangements of curves using polygonal approximations. In *Proc. 14th Europ. Workshop Comp. Geom. (EWCG)*.
`http://www.unilim.fr/pages_perso/manuela.neagu/publis.html`.

PREPARATA, F. P., AND SHAMOS, M. I. 1985. *Computational Geometry: An Introduction*. Springer.

SCHIRRA, S. 2000. Robustness and precision issues in geometric computation. In *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publishers B.V., 597–632.

SEDERBERG, T. W., AND MEYERS, R. J. 1988. Loop detection in surface patch intersections. *Computer Aided Geometric Design* **5**, 2 (July), 161–171.

SEDERBERG, T. W., AND NISHITA, T. 1990. Curve intersection using Bézier clipping. *Computer Aided Design* **22**, 9 (November), 538–549.

SEDERBERG, T. W., AND PARRY, S. R. 1986. Comparison of three curve intersection algorithms. *Computer Aided Design* **18**, 1, 58–63.

SNOEYINK, J., AND HERSHBERGER, J. 1989. Sweeping arrangements of curves. In *Proc. 5th Annu. ACM Sympos. Comput. Geom. (SCG)*, 354–363.

WEIN, R., FOGEL, E., ZUKERMAN, B., AND HALPERIN, D. 2006. 2D arrangements. In CGAL*-3.2 User and Reference Manual*, CGAL Editorial Board, Ed.
`http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Arrangement_2/Chapter_main.html`.

WEIN, R., FOGEL, E., ZUKERMAN, B., AND HALPERIN, D. 2006. Advanced programming techniques applied to CGAL’s arrangement package. *Computational Geometry: Theory and Applications*. To appear.

YAP, C. K., AND DUBÉ, T. 1995. The exact computation paradigm. In *Computing in Euclidean Geometry*, D. Z. Du and F. K. Hwang, Eds., 2nd ed., vol. **1** of *Lecture Notes Series on Computing*. World Scientific, Singapore, 452–492.

YAP, C. K. 2004. Robust geometric computation. In *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O’Rourke, Eds., 2nd ed. Chapman & Hall/CRC, ch. 41, 927–952.

YAP, C. K. 2006. Complete subdivision algorithms I: Intersection of Bézier curves. In *Proc. 22nd Annu. ACM Sympos. Comput. Geom. (SCG)*, 217–226.