# Fast and Exact Continuous Collision Detection with Bernstein Sign Classification

Min Tang[1]* Ruofeng Tong[1] Zhendong Wang[1] Dinesh Manocha[2]†
1. State Key Lab of CAD&CG, Zhejiang University 2. University of North Carolina at Chapel Hill
http://gamma.cs.unc.edu/BSC/

## Abstract

We present fast algorithms to perform accurate CCD queries between triangulated models. Our formulation uses properties of the Bernstein basis and Bézier curves and reduces the problem to evaluating signs of polynomials. We present a geometrically exact CCD algorithm based on the exact geometric computation paradigm to perform reliable Boolean collision queries. Our algorithm is more than an order of magnitude faster than prior exact algorithms. We evaluate its performance for cloth and FEM simulations on CPUs and GPUs, and highlight the benefits.

**CR Categories:** I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling

**Keywords:** Continuous collision detection, Bernstein sign classification, Exact geometric computation, Physically based simulation

**Links:** DL PDF WEB

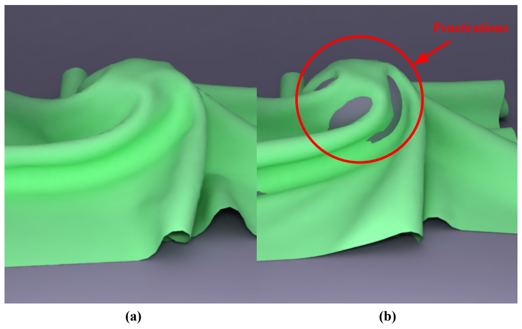


**Figure 1: Benefits of Reliable CCD Queries***: We highlight the benefits of our exact CCD algorithm on cloth simulation. Our algorithm can be used to generate a plausible simulation (a). If parameters are not properly tuned, floating-point-based CCD algorithms (b) can result in penetrations and artifacts.*

## 1 Introduction

The problem of fast and reliable collision detection arises in physically-based simulation, geometric computing, and robotics. Many applications require accurate algorithms that do not miss a single collision and maintain intersection-free meshes throughout the simulation. Some of the widely-used algorithms for contact computation are based on continuous collision detection (C-CD). Given two discrete instances or configurations of rigid or deformable models, CCD algorithms model the motion of each object or a mesh element using a continuous trajectory between the configurations and check for collisions along the trajectory. These algorithms are widely used for cloth simulation [Provot 1997; Bridson et al. 2002; Harmon et al. 2008; Brochu et al. 2012], rigid-body simulation [Redon et al. 2002], hair simulation [Selle et al. 2008], FEM simulation [Tang et al. 2011], robot motion planning [LaValle 2006; Tang et al. 2010a], dynamic solvers [Stam 2009], etc.

The simplest algorithms for triangular meshes linearly interpolate the trajectories of the vertices. In this case, contact computation reduces to performing a series of elementary tests between the vertices, edges, and faces using cubic polynomial root solvers [Provot 1997; Bridson et al. 2002]. Many high-level culling techniques have also been proposed to reduce the number of elementary tests performed between the meshes of complex models.

The elementary tests are typically implemented using finite-precision or floating-point arithmetic and use error tolerances. The numerical errors in arithmetic operations along with the tolerances can impact these elementary tests' accuracy (Fig. 1). There are two types of problems: *false negatives*, when the CCD algorithm may miss a collision; and *false positives*, when the CCD algorithm, acting conservatively, flags a non-colliding configuration as a collision. In order to overcome these problems, Brochu et al. [2012] proposed algorithms for exact CCD computation that can perform reliable collision queries. However, their approach can be relatively expensive due to use of large number of exact arithmetic operations. Moreover, its portability may be limited as efficient implementations of exact computation libraries are not easily available on all processors (e.g. GPUs).

**Main Results:** We present fast and accurate algorithms to perform reliable CCD queries. Our approach is based on using *coplanarity* and *inside* tests and reduces the computation to finding roots of algebraic equations and inequalities (i.e. a semi-algebraic set). We represent these functions using the Bernstein basis and exploit geometric properties of Bézier curves to design an efficient and reliable *Bernstein sign classification* (BSC) approach for CCD. The overall collision query is reduced to performing a series of sign evaluations of algebraic expressions and involves simple arithmetic operations. We also present a conservative elementary culling algorithm to improve the algorithm's performance. We use BSC to design two algorithms:

**1. BSC-exact:** This is an exact algorithm to perform CCD queries based on the *exact geometric computation* paradigm [Yap 2004] and is not susceptible to false positives or false negatives. We use extended precision arithmetic operations and accelerate the performance using floating-point filters. As compared to prior exact CCD algorithm [Brochu et al. 2012], we observe $10-25$X speedup on a single CPU core.

*e-mail:{tang_m,trf,westernseawolf}@zju.edu.cn
†e-mail:dm@cs.unc.edu

**2. BSC-float:** This is a finite-precision variant and is implemented using floating-point arithmetic operations. We have evaluated its performance on CPUs and GPUs and observe considerable speedups over prior floating-point CCD algorithms. Furthermore, we observe significant improvement in accuracy, i.e. significant reduction in the number of false positives and false negatives using our algorithm.

The overall algorithms are simple to implement, using only addition, subtraction, and multiplication operations. The use of the Bernstein basis and simple arithmetic operations results in reduced errors and improved efficiency. We highlight the benefits of algorithms using cloth and FEM simulation benchmarks.

## 2 Related Work

In this section, we give a brief overview of prior work on CCD algorithms, high-level collision culling, and the computation of the roots of polynomials.

Many techniques have been proposed for CCD between rigid models [Redon et al. 2002; Kim and Rossignac 2003], articulated models [Zhang et al. 2007], and deformable models [Volino and Thalmann 1994; Govindaraju et al. 2005; Hutter and Fuhrmann 2007; Tang et al. 2011]. At the lowest level, these algorithms perform elementary tests between triangle pairs. The elementary tests are typically performed by computing roots of cubic polynomials. Other CCD algorithms are based on conservative local advancement [Tang et al. 2009b]. All these methods are prone to floating-point errors and numerical tolerances. Therefore, they can result in false negatives and false positives. Wang [2014] has performed forward error analysis for elementary tests and used that analysis to derive tight error bounds for floating-point computation. This is used to reduce the number of false positive. In contrast, our BSC-exact algorithm and the approach described in [Brochu et al. 2012] are reliable. The tight error bounds in [Wang 2014] can be used to derive tighter error bounds for BSC-float.

**High-level Culling:** Many high-level techniques have been proposed to accelerate CCD computations by reducing the number of elementary tests between the triangle pairs, such as removing redundant elementary tests [Curtis et al. 2008; Tang et al. 2009a; Wong and Baciu 2006]. The simplest culling algorithms use BVHs (bounding volume hierarchies) based on k-DOPs or AABBs. Other methods use bounds on surface normals and curvature [Volino and Thalmann 1994; Provot 1997; Mezger et al. 2003] or perform self-collision culling [Schvartzman et al. 2010; Pabst et al. 2010; Zheng and James 2012]. Many of these algorithms are implemented using floating-point arithmetic operations and are prone to numerical errors.

**Polynomial Root Evaluation:** Many numerical iterative methods have been proposed to compute roots of polynomial equations. They tend to use tolerances and can result in false positives or false negatives for CCD computations. In computer graphics and geometric modeling, polynomials are represented using the spline basis, and their roots can be computed using the geometric subdivision methods, such as de Casteljau's algorithm [Farin 2002] or Bézier clipping [Sederberg and Nishita 1990]. These subdivision methods are implemented using finite-precision arithmetic and are also prone to roundoff errors. There is extensive literature in symbolic computation and computational geometry on reliably computing the roots of polynomials using exact arithmetic [Yap 2004; Mourrain et al. 2005].

## 3 CCD and Algebraic Formulation

In this section, we formulate CCD queries in terms of algebraic equations and inequalities. We assume that the vertices of the mesh move with a constant velocity during the time interval and that the CCD query reduces to performing two types of Boolean queries or elementary tests [Provot 1997; Bridson et al. 2002; Brochu et al. 2012]. These include the *VF query*, which checks whether a moving vertex intersects with a moving triangle, and the *EE query*, which checks whether a moving edge intersects with another moving edge. All these queries assume that the time interval is $t \in [0, 1]$ and that the initial configuration at $t = 0$ is intersection-free. If the Boolean query returns a positive answer, we can use techniques based on interval arithmetic to compute the intersection points or first time of contact to a desired precision. In many applications, only the parity of the number of collisions is needed for robust simulation [Brochu et al. 2012]. As a result, we focus on reliably computing a yes/no answer to the Boolean queries. The exact root and the first time of contact can be computed using root isolation and interval arithmetic techniques.

We first introduce the notations used in the rest of the paper. Next, we present some properties of Bernstein basis functions and Bézier curves that are used by our CCD algorithm.

### 3.1 Notations

We use following notations in the rest of the paper: Lower case letters in normal fonts (e.g. $a$, $b$, $a_i$,) represent scalar variables. Upper case letters (e.g., $L$, $J(t)$)) represent scalar functions. Lower case letters in bold face fonts (e.g. $\mathbf{a}$, $\mathbf{b}_t$) represent vector quantities. Upper case letters in bold face fonts (e.g., $\mathbf{L}$, $\mathbf{J}(\mathbf{t})$) represent vector-valued functions. $F'(t)$ and $F''(t)$ are the 1st and 2nd order of derivatives of a scalar function $F(t)$, respectively. The operators '$*$', '$\cdot$', and '$\times$' denote the usual scalar multiplication, dot product, and cross product, respectively. Operator $\mathbf{Sign}()$ returns the sign of a scalar variable. All the proofs of the lemmas, theorems and corollaries are in the supplementary material.

### 3.2 Bézier Curves and Bernstein Basis

We use the symbol $B_i^n(t)$ to represent the $i^{th}$ basis function of the Bernstein polynomials of degree $n$, i.e. $B_i^n(t) = \frac{n!}{i!(n-i)!}(1-t)^{n-i}t^i$, where $t \in [0, 1]$ and $0 \leq i \leq n$. The Bernstein polynomial basis is widely used in geometric modeling for curve and surface representation as well as in numerical analysis and computer algebra for root computations [Mourrain et al. 2005]. It is well-known that the polynomials expressed in the Bernstein basis have better numerical stability under perturbation of their coefficients than do those in the power basis [Farouki and Rajan 1987]. As a result, we represent the semi-algebraic set used for CCD queries in Bernstein basis.

Given a cubic polynomial $Y(t)$, it can be expressed using the Bernstein basis, i.e.

$$Y(t) = k_0 * B_0^3(t) + k_1 * B_1^3(t) + k_2 * B_2^3(t) + k_3 * B_3^3(t). \quad (1)$$

It corresponds to a cubic Bézier curve $\mathbf{F}(t)$ in a plane, where:

$$\begin{aligned} \mathbf{F}(t) &= \begin{pmatrix} t \\ Y(t) \end{pmatrix} = \begin{pmatrix} 0 \\ k_0 \end{pmatrix} * B_0^3(t) + \begin{pmatrix} 1/3 \\ k_1 \end{pmatrix} * B_1^3(t) \\ &+ \begin{pmatrix} 2/3 \\ k_2 \end{pmatrix} * B_2^3(t) + \begin{pmatrix} 1 \\ k_3 \end{pmatrix} * B_3^3(t). \quad (2) \end{aligned}$$

We exploit some geometric properties of cubic Bézier curves in order to characterize inflection points and extreme points. An *inflection point* occurs where the curvature vanishes or changes its

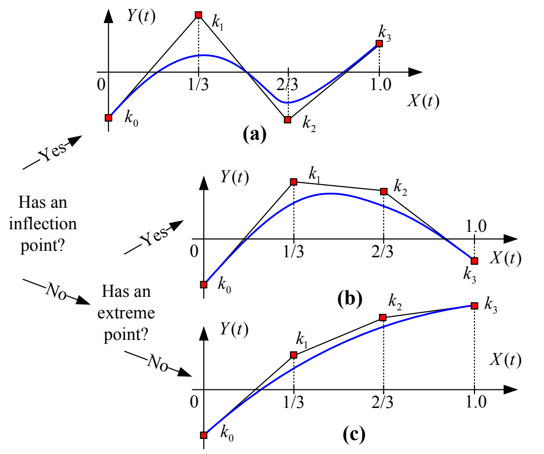


**Figure 2: Bézier Classifications:** *We classify the cubic Bézier curve into three categories (a)-(c), depending on whether it has an inflection point or an extreme point.*

bending direction. The *extreme points* correspond to local minima or maxima. Every cubic Bézier curve can be classified into three categories (as shown in Fig. 2), depending on whether it has any inflection point or extreme point over its domain ($t \in [0, 1]$) [Farin 2002]:

- **Case (a):** The curve has an inflection point.
- **Case (b):** The curve has no inflection point, but an extreme point.
- **Case (c):** The curve has neither an inflection point nor an extreme point.

The existence of an inflection point or an extreme point can be checked based on the lemmas in the supplementary material.

A cubic Bernstein polynomial can be decomposed into lower-degree polynomials based on the following theorem:

**Polynomial Decomposition Theorem:** *Let $G(t)$ and $H(t)$ be a cubic polynomial and a quadratic polynomial, respectively:*

$$\begin{aligned} G(t) &= i_0 * B_0^3(t) + i_1 * B_1^3(t) + i_2 * B_2^3(t) + i_3 * B_3^3(t), \\ H(t) &= j_0 * B_0^2(t) + j_1 * B_1^2(t) + j_2 * B_2^2(t). \end{aligned} \tag{3}$$

*$G(t)$ can be decomposed as:*

$$G(t) = L(t) * H(t) + K(t), \tag{4}$$

*where $L(t)$ and $K(t)$ are two linear polynomials:*

$$\begin{aligned} L(t) &= u_0 * B_0^1(t) + u_1 * B_1^1(t), \\ K(t) &= v_0 * B_0^1(t) + v_1 * B_1^1(t), \end{aligned} \tag{5}$$

*where $u_{[0,1]}$ and $v_{[0,1]}$ can be calculated from $i_{[0...3]}$ and $j_{[0...2]}$.*

## 3.3 CCD Queries

The CCD test between a triangle pair reduces to performing 6 VF queries and 9 EE queries. Each of these queries can be further decomposed into two parts [Provot 1997; Bridson et al. 2002]:

- **Coplanarity test:** The VF and EE queries involve the use of four deforming vertices. In order for a collision to occur, it is necessary that those four vertices be coplanar.
- **Inside test:** In addition to satisfy the coplanarity condition, we need to check whether the moving vertex is inside the triangle (VF), or the two edges intersect with each other at an interior point (EE).

The coplanarity test for a VF pair can be expressed as:

$$(\mathbf{p}_t - \mathbf{a}_t) \cdot \mathbf{n}_t = 0, \tag{6}$$

where $\mathbf{p}_t$ corresponds to the moving vertex, $\mathbf{a}_t, \mathbf{b}_t, \mathbf{c}_t$ are the vertices of the deforming triangle, and $\mathbf{n}_t$ is the normal vector of the triangle (i.e. $\mathbf{n}_t = (\mathbf{b}_t - \mathbf{a}_t) \times (\mathbf{c}_t - \mathbf{a}_t)$).

In order to perform an inside test for a VF pair, we need to perform three one-sided tests, i.e. $\mathbf{p}_t$ needs to be inside the triangle. This can be expressed based on the following inequalities:

$$((\mathbf{b}_t - \mathbf{p}_t) \times (\mathbf{c}_t - \mathbf{p}_t)) \cdot \mathbf{n}_t \geq 0, \tag{7}$$
$$((\mathbf{c}_t - \mathbf{p}_t) \times (\mathbf{a}_t - \mathbf{p}_t)) \cdot \mathbf{n}_t \geq 0, \tag{8}$$
$$((\mathbf{a}_t - \mathbf{p}_t) \times (\mathbf{b}_t - \mathbf{p}_t)) \cdot \mathbf{n}_t \geq 0. \tag{9}$$

The coplanarity and inside tests can be combined to find a common root of the following system of algebraic equation and inequalities (i.e. a semi-algebraic set). The VF query reduces to checking whether this semi-algebraic set has a real solution for $t \in [0, 1]$.

$$\begin{cases} (\mathbf{p}_t - \mathbf{a}_t) \cdot \mathbf{n}_t = 0, \\ ((\mathbf{b}_t - \mathbf{p}_t) \times (\mathbf{c}_t - \mathbf{p}_t)) \cdot \mathbf{n}_t \geq 0, \\ ((\mathbf{c}_t - \mathbf{p}_t) \times (\mathbf{a}_t - \mathbf{p}_t)) \cdot \mathbf{n}_t \geq 0, \\ ((\mathbf{a}_t - \mathbf{p}_t) \times (\mathbf{b}_t - \mathbf{p}_t)) \cdot \mathbf{n}_t \geq 0. \end{cases} \tag{10}$$

## 3.4 Coplanarity Tests using Bernstein Polynomials

In order to check the coplanarity of a vertex $\mathbf{p}_t$ and a triangle (defined by $\mathbf{a}_t$, $\mathbf{b_t}$, and $\mathbf{c_t}$), we need to calculate the projected distance between them along the direction of $\mathbf{n_t}$. If this distance becomes zero at any time in the interval, the four vertices are classified as coplanar based on following theorem.

**Coplanarity Test Theorem for a VF Pair:** *For a deforming triangle, whose initial and final positions are given as ($\mathbf{a_0}$, $\mathbf{b_0}$, $\mathbf{c_0}$) and ($\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{c_1}$) and a vertex with initial and final positions as $\mathbf{p_0}$ and $\mathbf{p_1}$, the coplanarity test can be formulated in terms of the following equation:*

$$\begin{aligned} &Y(t) = (\mathbf{p}_t - \mathbf{a}_t) \cdot \mathbf{n}_t = 0 \\ &= k_0 * B_0^3(t) + k_1 * B_1^3(t) + k_2 * B_2^3(t) + k_3 * B_3^3(t), \end{aligned} \tag{11}$$

*where $k_{[0..3]}$ are scalars can be calculated from ($\mathbf{a_0}$,$\mathbf{b_0}$, $\mathbf{c_0}$, $\mathbf{p_0}$) and ($\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{c_1}$, $\mathbf{p_1}$).*

The coplanarity test reduces to checking whether the 2D cubic Bézier curve $\mathbf{F}(t)$ (Equation (2)) defined in the $(X, Y)$ plane intersects with the X-axis.

## 3.5 Inside Tests using Bernstein Polynomials

We can also formulate the inside tests using Bernstein polynomials.

**Inside Test Theorem for a VF Pair:** *Given the triangle and the vertex defined by start and end positions over the interval $[0, 1]$, the inside test can be formulated in terms of the following inequality:*

$$\begin{aligned} ((\mathbf{b_t} - \mathbf{p_t}) \times (\mathbf{c_t} - \mathbf{p_t})) \cdot \mathbf{n_t} &= l_0 * B_0^4(t) + l_1 * B_1^4(t) \\ &+ l_2 * B_2^4(t) + l_3 * B_3^4(t) + l_4 * B_4^4(t) \geq 0, \end{aligned} \tag{12}$$

*where $l_{[0..4]}$ are scalars that can be calculated from ($\mathbf{a_0}$,$\mathbf{b_0}$, $\mathbf{c_0}$, $\mathbf{p_0}$) and ($\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{c_1}$, $\mathbf{p_1}$).*

**Simplified Inside Test Theorem for a VF pair:** *Based on combining Inequality (12) with Equation (11) and algebraic elimination, this inside test can be reduced to the following degree-two formulation:*

$$P(t) = p_0 * B_0^2(t) + p_1 * B_1^2(t) + p_2 * B_2^2(t) \geq 0, \tag{13}$$

*where $p_{[0\ldots2]}$ are scalars, which can be calculated based on $k_{[0\ldots3]}$ and $l_{[0\ldots4]}$, as shown in the supplementary material.*

## 3.6 CCD Tests using Bernstein Polynomials

The formulations for coplanarity and inside tests can be combined into the following system of equations and inequalities in terms of Bernstein polynomials:

$$\begin{cases} k_0 * B_0^3(t) + k_1 * B_1^3(t) + k_2 * B_2^3(t) + k_3 * B_3^3(t) = 0, \\ p_0 * B_0^2(t) + p_1 * B_1^2(t) + p_2 * B_2^2(t) \geq 0, \\ q_0 * B_0^2(t) + q_1 * B_1^2(t) + q_2 * B_2^2(t) \geq 0, \\ r_0 * B_0^2(t) + r_1 * B_1^2(t) + r_2 * B_2^2(t) \geq 0. \end{cases}$$

where $k_{[0\ldots3]}$ and $p_{[0\ldots2]}$ are scalars defined above, $q_{[0\ldots2]}$ and $r_{[0\ldots2]}$ are the coefficients corresponding to 2 other inside tests.

# 4 CCD Query Using Sign Evaluations

In this section, we use the formulation of CCD computation in terms of Bernstein polynomials and present accurate algorithms to perform CCD queries. Our formulation consists of two stages:

- **Geometric Coplanarity Test:** By deducing the signs of the polynomials at its extreme points and comparing with the signs of its end points in the interval $[0, 1]$, we can check for the existence of roots for coplanarity equations.
- **Geometric Inside Tests:** During this stage, we evaluate the signs of the inequalities at the roots that have passed coplanarity tests to check whether these roots also satisfy the inside tests.

## 4.1 Geometric Coplanarity Test

Our goal is to compute the roots of a cubic polynomial $Y(t)$ (defined by Equation (11) in domain $[0, 1]$). We use the characterization of Bézier curves into three different cases presented in Section 3.2. For the Case (a) in Section 3.2, we subdivide the curve at its inflection point, i.e. $t = \frac{k_2 - 2*k_1 + k_0}{k_0 - 3*k_1 + 3*k_2 - k_3}$, using de Casteljau's algorithm. The two subdivided curves either correspond to Case (b) or Case (c) in Section 3.2. We discuss both these cases:

- **Case (b):** If $k_0$ and $k_3$ have different signs, there is only one root in the domain. Otherwise, we use the following **Root-Finding Lemma** to determine whether there are zero roots or two roots in the domain.
- **Case (c):** If $k_0$ and $k_3$ have the same sign, there is no root; otherwise there is one root in its domain.

**Root-Finding Lemma:** *For a cubic polynomial $Y(t)$ (defined by Equation (11)) with an extreme point in its domain, its 1st derivative $Y'(t)$ is:*

$$\begin{aligned} Y'(t) &= 3 * (k_1 - k_0) * B_0^2(t) + 3 * (k_2 - k_1) * B_1^2(t) \\ &+ 3 * (k_3 - k_2) * B_2^2(t). \end{aligned}$$

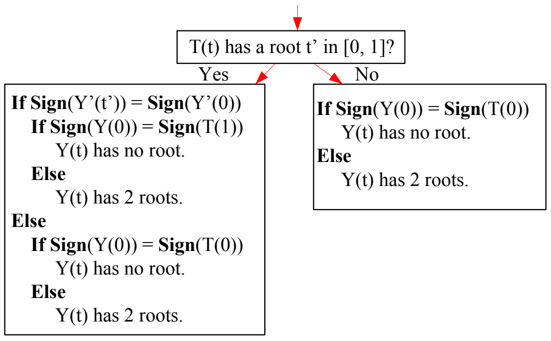

T(t) has a root t' in [0, 1]?
Yes
No
If Sign(Y'(t')) = Sign(Y'(0))
If Sign(Y(0)) = Sign(T(1))
Y(t) has no root.
Else
Y(t) has 2 roots.
Else
If Sign(Y(0)) = Sign(T(0))
Y(t) has no root.
Else
Y(t) has 2 roots.
If Sign(Y(0)) = Sign(T(0))
Y(t) has no root.
Else
Y(t) has 2 roots.

**Figure 3: Computing the Number of Roots of $Y(t)$:** *We can compute them based on sign evaluations.*

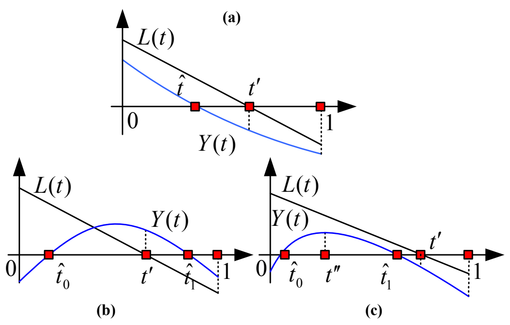


**Figure 4: Evaluate the Sign of $L(\hat{t})$:** *Based on* **Sign Determination Theorem I** *and* **Sign Determination Theorem II**, *we can evaluate the sign of $L(\hat{t})$.*

*We decompose $Y(t) = Y'(t) * S(t) + T(t)$, where $S(t)$ and $T(t)$ are two linear polynomials and can be calculated with the* **Polynomial Decomposition Theorem** *in Section 3.2. We use the classification in Fig. 3 to compute the number of roots of $Y(t)$.*

Based on this formulation, we can compute the number of roots for Case (b) and Case (c), and consequently for Case (a).

## 4.2 Geometric Inside Tests

In order to perform a specific inside test, along with the coplanarity test, we need to test the following system:

$$\begin{cases} Y(t) = 0, \\ P(t) \geq 0. \end{cases} \tag{14}$$

Here $Y(t)$ and $P(t)$ are defined by Equation (11) and Equation (13), respectively. We compute a similar system for the other two inside tests.

Based on the **Polynomial Decomposition Theorem** in Section 3.2, we can express:

$$Y(t) = L(t) * P(t) + K(t), \tag{15}$$

where $L(t)$ and $K(t)$ are linear polynomials.

Let $\hat{t}$ be a root of $Y(t)$ in the domain $[0, 1]$, i.e. $Y(\hat{t}) = 0, \hat{t} \in [0, 1]$. From Equation (15), we obtain $P(\hat{t}) = -K(\hat{t})/L(\hat{t})$. Therefore, the problem of computing the sign of $P(\hat{t})$ reduces to computing the signs of $K(\hat{t})$ and $L(\hat{t})$.

We use following theorems to compute the signs of $K(\hat{t})$ and $L(\hat{t})$:

```
If Sign(Y(t')) = Sign(Y(0))
    Sign(L($\hat{t}$)) ← Sign(L(1))
Else
    Sign(L($\hat{t}$)) ← Sign(L(0))
```

(a)

```
If Sign(Y(t')) ≠ Sign(Y(0))
    Sign(L($\hat{t}_0$)) ← Sign(L(0))
    Sign(L($\hat{t}_1$)) ← Sign(L(1))
Else
    If Sign(Y'(t')) = Sign(Y'(0))
        Sign(L($\hat{t}_0$)) ← Sign(L(1))
        Sign(L($\hat{t}_1$)) ← Sign(L(1))
    Else
        Sign(L($\hat{t}_0$)) ← Sign(L(0))
        Sign(L($\hat{t}_1$)) ← Sign(L(0))
```

(b)

**Figure 5: Rules for Evaluating the Sign of $L(\hat{t})$, $L(\hat{t}_0)$, and $L(\hat{t}_1)$:** *We use the rules in (a) and (b) for* **Sign Determination Theorem I** *and* **Sign Determination Theorem II**, *respectively.*

**Sign Determination Theorem I:** *Let $L(t)$ be a linear polynomial and $Y(t)$ be a cubic polynomial which corresponds to the Bézier curve of Case (b) in the domain $[0, 1]$ (Fig. 4(a)). Let:*

- *$L(t') = 0$, and $t' \in [0, 1]$,*
- *$Y(\hat{t}) = 0$, and $\hat{t} \in [0, 1]$.*

*We can use the rules in Fig. 5(a) to evaluate the sign of $L(\hat{t})$).*

**Sign Determination Theorem II:** *Let $L(t)$ be a linear polynomial and $Y(t)$ be a cubic polynomial that corresponds to the Bézier curve of Case (c) in the domain $[0, 1]$ (Fig. 4(b) and Fig. 4(c)). Let:*

- *$L(t') = 0$, and $t' \in [0, 1]$,*
- *$Y(\hat{t}_0) = 0$ and $Y(\hat{t}_1) = 0$, and $\hat{t}_0 \in [0, 1], \hat{t}_1 \in [0, 1], \hat{t}_0 < \hat{t}_1$,*
- *$Y'(t'') = 0$, and $t'' \in [0, 1]$. $Y'(t)$ is the 1st order of derivative of $Y(t)$.*

*We can use the rules in Fig. 5(b) to determine the sign of $L(\hat{t}_0)$) and $L(\hat{t}_1)$).*

Based on **Sign Determination Theorem I** and **Sign Determination Theorem II**, we can determine the sign of $L(\hat{t})$.

**Sign of $K(\hat{t})$:** The algorithm used to compute the sign of $L(\hat{t})$ can be directly used to compute the sign of $K(\hat{t})$.

Based on the signs of $L(\hat{t})$ and $K(\hat{t})$, we can compute the sign of $P(\hat{t})$ and consequently check whether the equality and inequality in Equation (14) are satisfied or not. This is repeated for the other two inequalities as well. If all of them are satisfied, then the answer to the CCD query is positive.

## 4.3 Conservative Culling Test

Many times there is no collision, and we use a simple culling scheme to accelerate the algorithm. This is similar to using the non-penetration filter [Tang et al. 2010b] or plane-culling [Brochu et al. 2012]. Our goal is to eliminate many VF pairs that do not satisfy the coplanarity condition (see Equation (11)). One sufficient condition is when all the coefficients $k_{[0...3]}$ are either greater than zero or less than zero. Instead of computing $k_{[0...3]}$ exactly, we use floating-point filters [Burnikel et al. 2001] to perform conservative culling. In other words, we compute $k_{[0...3]}$ using floating-point arithmetic. Instead of comparing them with zeros, we check whether they are all greater than $\hat{\epsilon}$, or all less than $-\hat{\epsilon}$, where $\hat{\epsilon}$ is a conservative error bound. The detailed method for computing $\hat{\epsilon}$ is in the supplementary material.

**Algorithm 1** VF-Test: CCD test for a VF pair.
**Input:** Positions at $t = 0$ and $t = 1$ for a deforming triangle $(\mathbf{a_0}, \mathbf{a_1}, \mathbf{b_0}, \mathbf{b_1}, \mathbf{c_0}, \mathbf{c_1})$ and a moving vertex $(\mathbf{p_0}, \mathbf{p_1})$.
**Output:** True or False for has a collision or no collision in $[0, 1]$.

```
1:  GetCoefficients()  // Get coefficients of $Y(t)$.
2:  // Perform conservative culling test.
3:  if ConservativeFilter() then
4:      Return False.
5:  end if
6:  ctype ← BezierType()  // Get type of the Bézier curve.
7:  // For case (a), subdivide and check on interval $[0, t']$ and $[t', 1]$.
8:  // Here $t'$ is corresponding to the inflection point.
9:  if ctype = Case A then
10:     Subdivide into two intervals $[0, t']$ and $[t', 1]$.
11:     Return VF-Test($[0, t']$) OR VF-Test($[t', 1]$).
12: end if
13: // For case (b) and case (c), continue checking.
14: // Perform Coplanarity Test (Section 4.1).
15: if !CoplanarityTest() then
16:     Return False.
17: end if
18: // Perform Inside Test (Section 4.2).
19: if !InsideTest() then
20:     Return False.
21: end if
22: Return True.  // A valid collision has been detected.
```

## 4.4 Overall VF Query Algorithm

Our overall algorithm for VF query is described in Algorithm 1. We first compute the coefficients of $Y(t)$, i.e. $k_{[0...3]}$ (Line 1), and perform the conservative culling test (Line 3–5). If the culling test fails, we classify the type of Bézier curves (Line 6). For case (a), we subdivide the interval $[0, 1]$ into two sub-intervals $[0, t']$ and $[t', 1]$, and recursively perform CCD tests on these sub-intervals (Line 9–12). For case (b) and (c), we perform the coplanarity test (Line 15–17) and inside tests (Line 19–21). If all these tests are positive, the response to VF collision query is positive (Line 22).

We use a similar algorithm for EE tests. The details of its derivation are given in the supplementary material. The main difference with respect to the VF test is in terms of the inequalities used for the inside tests.

**BSC-exact: Exact VF Computation:** In order to perform reliable collision queries, we use the well-known paradigm of *Exact Geometric Computation* [Yap 2004], which is widely used for geometric computations and has also been used to perform exact Boolean answers for CCD [Brochu et al. 2012]. The underlying philosophy is that we compute the correct answer to these Boolean queries assuming that we use exact arithmetic and there are no errors due to use of fixed precision or floating-point arithmetic or user specified tolerances. Our exact algorithm, BSC-exact, uses a combination of extended precision arithmetic operations and floating point filters. Our conservative-culling test only uses floating point filters and does not perform exact arithmetic operations. The rest of the computations include many expressions and evaluating signs of polynomials. All these computations can be accelerated using floating point filters.

**BSC-float: Floating-point Algorithm:** In some cases, optimized libraries for extended precision-arithmetic operations are not available on certain processors (e.g. GPUs). In this case, all the steps of Algorithm 1 are implemented using floating-point arithmetic and are prone to numerical errors. Our resulting algorithm, BSC-float, is based on the IEEE floating-point standard.

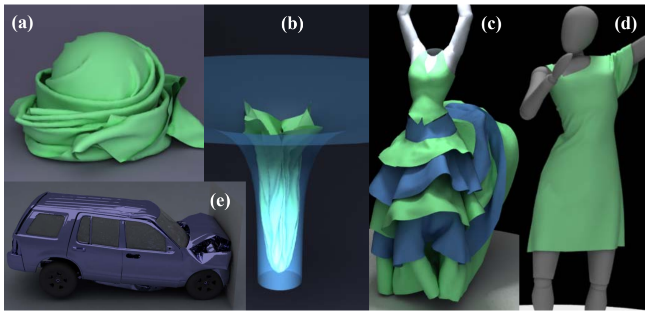


**Figure 6: Benchmarks:** *We use five different benchmarks arising from cloth and FEM simulations.*

## 5 Implementation and Performance

In this section, we describe our implementation and highlight the performance of our algorithm on several benchmarks.

### 5.1 Implementation

We have implemented our algorithms on a standard PC (Intel i7-3770K CPU @3.5GHz, 4GB RAM, 64-bits Window 7 OS, NVIDIA Tesla K40c GPU). This includes a CPU-based C++ implementation of BSC-exact that uses a single core and uses an exact computation library based on interval arithmetic [Brochu et al. 2012]. We have also implemented BSC-float on a CPU (with C++) and a GPU (using CUDA 5.5) using hardware-supported floating-point operations.

We compare the performance of our algorithms with the following algorithms:

1. **El-Topo-exact:** This is the implementation of the exact algorithm of [Brochu et al. 2012], made available by the authors. It also uses plane-based culling to accelerate the computation, along with interval arithmetic-based filters and exact expansions for exact arithmetic operations. In order to compare the performance with BSC-exact, we use the same implementation of exact arithmetic operations.

2. **El-Topo-float:** This is a floating-point-based cubic root solver CCD implementation, available as part of El-Topo surface-tracking library [Brochu and Bridson 2009]. We measured its performance using a single thread on the CPU.

3. **BSC-float-GPU and El-Topo-float-GPU:** We also ported BSC-float and El-Topo-float algorithms to GPUs and tested their performance with multiple threads, referred to as BSC-float-GPU and El-Topo-float-GPU, respectively.

### 5.2 Benchmarks

In order to test the performance of our algorithms, we used five different benchmarks arising from different simulation scenarios that use CCD queries.

- **Dancer:** A dancer wearing a simple skirt with $5K - 10K$ triangles, the number of triangles change during the simulation due to adaptive computations. This benchmark has a high number of self-collisions (Figure 6(d)).

- **Twisting:** A cloth with $2K - 50K$ triangles twists severely as the underlying ball is rotating. This benchmark has a high number of self-collisions (Figure 6(a)).

- **Flamenco:** A fiery Flamenco dancer wearing a colorful skirt with ruffles. This benchmark ($49K$ triangles) has many inter- and intra-object collisions (Figure 6(c)).

- **Funnel:** A cloth with $2K - 42K$ triangles falls into a funnel and folds to fit into the funnel with many self-collisions (Figure 6(b)).

- **Crashing:** A Ford Explorer with $1.1M$ triangles crashes against a rigid wall and the deformation is simulated using finite-element meshing (Figure 6(e)).

The first three benchmarks (Dancer, Twisting, and Funnel) are generated by integrating our CCD algorithm into a cloth simulation system, ArcSim [Narain et al. 2012]. The input for the Flamenco and the Crashing benchmarks is given as discrete keyframes. We use linear interpolation between key-frames and check for inter-object and self-collisions. We also use BVH-based hierarchical culling (using AABBs) to reduce the number of elementary tests.

**Worst-Case Query Performance:** If there is no collision, our culling algorithm is able to discard many of those instances. The query time is higher when there is an actual contact. The worst-case query times for our algorithm vs. prior algorithms are:

- *BSC-exact:* The worst-case time for EE and VF queries are about 876 ns. In contrast, the worst-case query times for El-Topo-exact are 15 ms and $11\mu s$ for EE and VE queries, respectively.

- *BSC-float:* The worst-case time for EE and VF queries are about 105 ns. In contrast, the worst-case query times for El-Topo-float are about 953 ns for both queries on a CPU core. Moreover, we observe fewer incorrect query results using BSC-float.

### 5.3 Relative Performance on a CPU

Figure 7 highlights the performance of our algorithms, BSC-exact and BSC-float, and compares them with two prior CCD algorithms, El-Topo-exact and El-Topo-float, on a single CPU core. For all these benchmarks, the performance of BSC-exact is about $10-25$X faster than El-Topo-exact, and offers similar reliability. Furthermore, we observe up to an order of magnitude speedup in the floating point implementations. Our approach, BSC-float, involves fewer arithmetic operations, as compared to El-Topo-float. The combination of fewer operations and improved numerical stability properties of Bernstein polynomials also improves the accuracy of BSC-float, i.e. fewer incorrect results to the collision queries in terms of false-negatives or false-positives.

### 5.4 Relative Performance on a GPU

We have also evaluated the performance on the NVIDIA Tesla K40c GPU. We are not aware of any widely optimized extended precision libraries on GPUs, so we only evaluated the relative performance of BSC-float-GPU and El-Topo-float-GPU on various benchmarks. We compared the accuracy of query results with those computed by exact CPU-based implementations. In this case, BSC-float-GPU results in much fewer inaccurate collision queries as compared to El-Topo-float-GPU. The internal registers used in GPUs may have different precision from CPUs, so we may observe considerable differences in the accuracy results of BSC-float-GPU and El-Topo-float-GPU, as compared to their CPU counterparts. For example, many Intel processors use 80-bit internal registers for floating-point operations, and this may result in higher accuracy for CPU-based implementations. We have also integrated BSC-float and El-Topo-float into a GPU-based cloth simulation system [Tang et al. 2013] and

| Benchmarks | # of Tests | BSC-exact | El-Topo-exact | BSC-float | | El-Topo-float | | BSC-float-GPU | | El-Topo-float-GPU | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Avg. Query Time | Avg. Query Time | Avg. Query Time | # of Inaccurate Queries | Avg. Query Time | # of Inaccurate Queries | Avg. Query Time | # of Inaccurate Queries | Avg. Query Time | # of Inaccurate Queries |
| Dancer | 405M | 9ns | 274ns | 4.4ns | 1 | 45ns | 357 | 1.7ns | 5 | 2.1ns | 412 |
| Twisting | 70.3M | 12ns | 252ns | 5.6ns | 0 | 48ns | 98 | 1.8ns | 1 | 2ns | 121 |
| Funnel | 58.5M | 21ns | 293ns | 8.8ns | 0 | 44ns | 131 | 2.1ns | 1 | 3.7ns | 156 |
| Flamenco | 4.2M | 20ns | 261ns | 8.2ns | 0 | 43ns | 12 | 1.8ns | 2 | 2.5ns | 54 |
| Crashing | 31.6M | 16ns | 259ns | 7.5ns | 2 | 45ns | 45 | 2ns | 5 | 3.1ns | 60 |

**Figure 7: Performance and Comparison:** *We highlight the performance of various CPU and GPU-based algorithms on different benchmarks. We observe significant speedups using our algorithms based on BSC vs. prior algorithms implemented as part of El Topo [Provot 1997; Bridson et al. 2002; Brochu and Bridson 2009; Brochu et al. 2012]. Even though BSC-float is not guaranteed to be reliable, we observe very high accuracy in our benchmarks, i.e. very few incorrect answers to the queries.*

compared the runtime query performance of both CCD algorithms within that system. Figure 7 highlights the performance of BSC-float-GPU and El-Topo-float-GPU. Due to parallelism, the relative performance improvement of BSC-float-GPU over El-Topo-float-GPU is less than those on the CPUs.

## 5.5 Analysis

The computational costs of our exact CCD algorithm (BSC-exact) varies with respect to different cases described in Section 3.2:

- **Case (c)**: No operation cost for the coplanarity test; involves 3 polynomial decompositions and 3 polynomial evaluations (of degree 3) for inside tests.
- **Case (b)**: Its operation cost includes 1 polynomial decomposition and 1 polynomial evaluation (of degree 2) for the coplanarity test; 3 polynomial decompositions and 6 polynomial evaluations (three of degree 2 and three of degree 3) for the inside test.
- **Case (a)**: Its total operation cost is the sum of (c) and (b).

The overall operation count of our algorithm is much lower than Eltopo-exact and this results in considerable speedups, as shown in Fig. 7. Furthermore, we only perform simple arithmetic operations such as additions, subtractions, and multiplications (see details in the appendix). In terms of extended precision computations, the division operations are more expensive than these three operations and we avoid those expensive operations in our algorithm.

The first time of contact can be easily computed using root isolation We perform mid-point subdivision (using Bernstein formulation) recursively, after Algorithm 1 returns true. The subdivision terminates when the size of the interval containing the root is less than a user-threshold. The mid-point of the interval is used to compute the intersection points. This takes about $30 - 40$ ns/query.

We also compared the performance of our solver with the Jenkins-Traub solver [1]. It is more accurate than Newton-interval solver (e.g. used in El Topo-float), but about $3X$ slower. All such numeric solvers are prone to floating-point errors and can result in false-positives and false-negatives. In contrast, our BSC-exact algorithm is reliable and faster than most of these numeric solvers.

## 6 Limitations, Conclusions and Future Work

We have presented novel algorithms to perform accurate CCD queries between triangular meshes. We exploit properties of Bernstein functions and Bézier curves, reducing the CCD queries to evaluating signs of Bernstein polynomials and algebraic expressions. We present two versions of the algorithm based on exact geometric computation and IEEE floating-point implementations. We have implemented these algorithms on CPUs and GPUs. Our exact algorithm is more than an order of magnitude faster than prior exact algorithms. Furthermore, our floating-point variant is faster and more accurate than prior solvers for elementary tests.

Our approach has some limitations. Our current formulation assumes that the vertices move with a constant velocity. Our reliable algorithm assumes exact representation of vertices, edges, and faces and does not take into account any errors in the input. Our floating-point variant (BSC-float) is faster and more accurate than prior methods, but it does not guarantee a safe and reliable solution. We perform only Boolean collision queries; and additional computations based on root isolation would be needed to compute the first-time-of-contact.

There are many avenues for future work. Besides overcoming these limitations, it may be useful to derive a tight error bound on our floating-point variant and the exact number of bits needed for extended precision. This would help explain its high accuracy in our benchmarks. It would be useful to use our reliable CCD algorithm for other applications including hair simulation and dynamic solvers [Zhao et al. 2012]. Finally, we would like to develop reliable algorithms for high-level CCD culling and collision-response.

**Acknowledgements:** This research is supported in part by NSFC (61170140), the National Basic Research Program of China (2011CB302205), the National Key Technology R&D Program of China (2012BAD35B01), the Doctoral Fund of Ministry of Education of China (20130101110133). Ruofeng Tong is partly supported by NSFC (61170141), the National High-Tech Research and Development Program (No.2013AA013903) of China. We gratefully acknowl-edge the support of NVIDIA Corporation with the donation of the Tesla K40c GPU used for this research.

## References

BRIDSON, R., FEDKIW, R., AND ANDERSON, J. 2002. Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph. 21*, 3 (July), 594–603.

[1]http://www.codeproject.com/Articles/552678/Polynomial-Equation-Solver

BROCHU, T., AND BRIDSON, R. 2009. Robust topological operations for dynamic explicit surfaces. *SIAM J. Sci. Comput. 31*, 4 (June), 2472–2493.

BROCHU, T., EDWARDS, E., AND BRIDSON, R. 2012. Efficient geometrically exact continuous collision detection. *ACM Trans. Graph. 31*, 4 (July), 96:1–96:7.

BURNIKEL, C., FUNKE, S., AND SEEL, M. 2001. Exact geometric computation using cascading. *International J. Comp. Geometry and Applications 11*, 3, 245–266. Special Issue.

CURTIS, S., TAMSTORF, R., AND MANOCHA, D. 2008. Fast collision detection for deformable models using representative-triangles. In *SI3D '08: Proceedings of the 2008 Symposium on Interactive 3D graphics and games*, 61–69.

FARIN, G. 2002. *Curves and surfaces for CAGD: a practical guide*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

FAROUKI, R. T., AND RAJAN, V. T. 1987. On the numerical condition of polynomials in berstein form. *Comput. Aided Geom. Des. 4*, 3 (Nov.), 191–216.

GOVINDARAJU, N., KNOTT, D., JAIN, N., KABUL, I., TAMSTORF, R., GAYLE, R., LIN, M., AND MANOCHA, D. 2005. Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH) 24*, 3, 991–999.

HARMON, D., VOUGA, E., TAMSTORF, R., AND GRINSPUN, E. 2008. Robust treatment of simultaneous collisions. *SIGGRAPH (ACM Transactions on Graphics) 27*, 3, 1–4.

HUTTER, M., AND FUHRMANN, A. 2007. Optimized continuous collision detection for deformable triangle meshes. In *Proc. WSCG '07*, 25–32.

KIM, B., AND ROSSIGNAC, J. 2003. Collision prediction for polyhedra under screw motions. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, SM '03, 4–10.

LAVALLE, S. M. 2006. *Planning Algorithms*. Cambridge University Press.

MEZGER, J., KIMMERLE, S., AND ETZMU$\beta$, O. 2003. Hierarchical techniques in cloth detection for cloth animation. *Journal of WSCG 11*, 1, 322–329.

MOURRAIN, B., ROUILLIER, F., AND ROY, M.-F. 2005. The Bernstein basis and real root isolation. In *Combinatorial and Computational Geometry*, MSRI Publications, 459–478.

NARAIN, R., SAMII, A., AND O'BRIEN, J. F. 2012. Adaptive anisotropic remeshing for cloth simulation. *ACM Trans. Graph. 31*, 6 (Nov.), 152:1–152:10.

PABST, S., KOCH, A., AND STRASSER, W. 2010. Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. *Computer Graphics Forum 29*, 5, 1605–1612.

PROVOT, X. 1997. Collision and self-collision handling in cloth model dedicated to design garments. In *Graphics Interface*, 177–189.

REDON, S., KHEDDAR, A., AND COQUILLART, S. 2002. Fast continuous collision detection between rigid bodies. *Proc. of Eurographics (Computer Graphics Forum) 21*, 3, 279–288.

SCHVARTZMAN, S. C., PÉREZ, A. G., AND OTADUY, M. A. 2010. Star-contours for efficient hierarchical self-collision detection. *ACM Trans. Graph. 29*, 4 (July), 80:1–80:8.

SEDERBERG, T. W., AND NISHITA, T. 1990. Curve intersection using Bézier clipping. *Comput. Aided Des. 22*, 9, 538–549.

SELLE, A., LENTINE, M., AND FEDKIW, R. 2008. A mass spring model for hair simulation. *ACM Trans. Graph. 27*, 3 (Aug.), 64:1–64:11.

STAM, J. 2009. Nucleus: Towards a unified dynamics solver for computer graphics. In *Proceedings of IEEE International Conference on Computer-Aided Design and Computer Graphics*, 1–11.

TANG, M., CURTIS, S., YOON, S.-E., AND MANOCHA, D. 2009. ICCD: interactive continuous collision detection between deformable models using connectivity-based culling. *IEEE Transactions on Visualization and Computer Graphics 15*, 544–557.

TANG, M., KIM, Y. J., AND MANOCHA, D. 2009. C2A: Controlled conservative advancement for continuous collision detection of polygonal models. *Proceedings of International Conference on Robotics and Automation*, 356–361.

TANG, M., KIM, Y. J., AND MANOCHA, D. 2010. CCQ: Efficient local planning using connection collision query. In *WAFR*, 229–247.

TANG, M., MANOCHA, D., AND TONG, R. 2010. Fast continuous collision detection using deforming non-penetration filters. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 7–13.

TANG, M., MANOCHA, D., YOON, S.-E., DU, P., HEO, J.-P., AND TONG, R. 2011. VolCCD: Fast continuous collision culling between deforming volume meshes. *ACM Trans. Graph. 30* (May), 111:1–111:15.

TANG, M., TONG, R., NARAIN, R., MENG, C., AND MANOCHA, D. 2013. A GPU-based streaming algorithm for high-resolution cloth simulation. *Computer Graphics Forum 32*, 7, 21–30.

VOLINO, P., AND THALMANN, N. M. 1994. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum 13*, 3, 155–166.

WANG, H. 2014. Defending continuous collision detection against errors. *ACM Trans. Graph. 33*, 4 (July), 122:1–122:10.

WONG, W. S.-K., AND BACIU, G. 2006. A randomized marking scheme for continuous collision detection in simulation of deformable surfaces. *Proc. of ACM VRCIA*, 181–188.

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

ZHANG, X., REDON, S., LEE, M., AND KIM, Y. J. 2007. Continuous collision detection for articulated models using Taylor models and temporal culling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*, 3, 15.

ZHAO, J., TANG, M., AND TONG, R. 2012. Connectivity-based segmentation for GPU-accelerated mesh decompression. *J. Comput. Sci. Technol. 27*, 6, 1110–1118.

ZHENG, C., AND JAMES, D. L. 2012. Energy-based self-collision culling for arbitrary mesh deformations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012) 31*, 4 (Aug.), 98:1–98:12.

# Fast and Exact Continuous Collision Detection with Bernstein Sign Classification

# Supplementary Material

Min Tang[1*] Ruofeng Tong[1] Zhendong Wang[1] Dinesh Manocha[2†]
1. State Key Lab of CAD&CG, Zhejiang University 2. University of North Carolina at Chapel Hill
*http://gamma.cs.unc.edu/BSC/*

## 1 Proofs

In this supplementary document, we provide proofs of various lemmas, theorems and corollaries used in the paper.

**Inflection Point Existence Lemma:** *For a cubic polynomial $Y(t) = k_0 * B^3_0(t) + k_1 * B^3_1(t) + k_2 * B^3_2(t) + k_3 * B^3_3(t)$, its 2nd order derivative $Y''(t)$ is:*

$$\begin{aligned} Y''(t) &= 6*(k_2 - 2*k_1 + k_0) * B^1_0(t) \\ &+ 6*(k_3 - 2*k_2 + k_1) * B^1_1(t). \end{aligned}$$

*If the two scalars $(k_2 - 2*k_1 + k_0)$ and $(k_3 - 2*k_2 + k_1)$ have different signs, there is an inflection point, otherwise there is no inflection point in $t \in [0, 1]$.*

*Proof.* The inflection point corresponds to the root of $Y''(t)$. If the two scalars $(k_2 - 2*k_1 + k_0)$ and $(k_3 - 2*k_2 + k_1)$ have different signs, there is one root for $Y''(t)$, i.e. an inflection point, otherwise there is no inflection point in $t \in [0, 1]$. □

**Extreme Point Existence Lemma:** *For a cubic polynomial $Y(t)$ (defined as above), its 1st derivative $Y'(t)$ is:*

$$\begin{aligned} Y'(t) &= 3*(k_1 - k_0) * B^2_0(t) + 3*(k_2 - k_1) * B^2_1(t) \\ &+ 3*(k_3 - k_2) * B^2_2(t). \end{aligned}$$

*If there is no inflection point in its domain and the two scalars ($k_1 - k_0$ and $k_3 - k_2$) have different signs, there is an extreme point, otherwise there is no extreme point in $t \in [0, 1]$.*

*Proof.* If there is no inflection point (i.e. no root for $Y''(t) = 0$) in $[0, 1]$, $Y'(t)$ is monotonic in the domain. It the two scalars ($k_1 - k_0$ and $k_3 - k_2$) have different signs, there is a root of $Y'(t) = 0$ in $[0, 1]$, which corresponds to an extreme point. Otherwise there is no extreme point in $[0, 1]$. □

**Polynomial Decomposition Theorem:** Let $G(t)$ and $H(t)$ be a cubic polynomial and a quadratic polynomial, respectively:

$$\begin{aligned} G(t) &= i_0 * B^3_0(t) + i_1 * B^3_1(t) + i_2 * B^3_2(t) + i_3 * B^3_3(t), \\ H(t) &= j_0 * B^2_0(t) + j_1 * B^2_1(t) + j_2 * B^2_2(t). \end{aligned} \tag{1}$$

$G(t)$ can be decomposed as:

$$G(t) = L(t) * H(t) + K(t), \tag{2}$$

where $L(t)$ and $K(t)$ are two linear polynomials:

$$\begin{aligned} L(t) &= u_0 * B^1_0(t) + u_1 * B^1_1(t), \\ K(t) &= v_0 * B^1_0(t) + v_1 * B^1_1(t), \end{aligned} \tag{3}$$

where $u_{[0,1]}$ and $v_{[0,1]}$ can be calculated from $i_{[0...3]}$ and $j_{[0...2]}$.

*Proof.* This can be proven by substituting the algebraic expressions.

$$\begin{aligned} L(t) * H(t) &= (j_0 * B^2_0(t) + j_1 * B^2_1(t) + j_2 * B^2_2(t)) \\ &* (u_0 * B^1_0(t) + u_1 * B^1_1(t)) \\ &= u_0 * j_0 * B^3_0(t) \\ &+ \frac{2*u_0*j_1 + u_1*j_0}{3} * B^3_1(t) \\ &+ \frac{u_0*j_2 + 2*u_1*j_1}{3} * B^3_2(t) \\ &+ u_1 * j_2 * B^3_3(t). \end{aligned} \tag{4}$$

Moreover,

$$\begin{aligned} K(t) &= K(t) * (1 - t + t)^2 \\ &= (v_0 * B^1_0(t) + v_1 * B^1_1(t)) \\ &* (B^2_0(t) + B^2_1(t) + B^2_2(t)) \\ &= v_0 * B^3_0(t) + \frac{2*v_0 + v_1}{3} * B^3_1(t) \\ &+ \frac{v_0 + 2*v_1}{3} * B^3_2(t) + v_1 * B^3_3(t). \end{aligned} \tag{5}$$

From Equation (4) and Equation (5), we obtain:

$$\begin{aligned} L(t) &* H(t) + K(t) \\ &= (u_0 * j_0 + v_0) * B^3_0(t) \\ &+ \frac{2*u_0*j_1 + u_1*j_0 + 2*v_0 + v_1}{3} * B^3_1(t) \\ &+ \frac{u_0*j_2 + 2*u_1*j_1 + v_0 + 2*v_1}{3} * B^3_2(t) \\ &+ (u_1 * j_2 + v_1) * B^3_3(t). \end{aligned} \tag{6}$$

Based on Equation (1) and Equation (2), we obtain:

$$i_0 = u_0 * j_0 + v_0 \tag{7}$$

$$i_1 = \frac{2*u_0*j_1 + u_1*j_0 + 2*v_0 + v_1}{3} \tag{8}$$

$$i_2 = \frac{u_0*j_2 + 2*u_1*j_1 + v_0 + 2*v_1}{3} \tag{9}$$

$$i_3 = u_1 * j_2 + v_1. \tag{10}$$

*e-mail:{tang_m,trf,westernseawolf}@zju.edu.cn
†e-mail:dm@cs.unc.edu

From Equation (7) and Equation (10), we obtain:

$$\begin{aligned} v_0 &= i_0 - u_0 * j_0 \\ v_1 &= i_3 - u_1 * j_2. \end{aligned}$$

We substitute these expressions into Equation (8) and Equation (9) and obtain:

$$\begin{array}{rcl} 3 * i_1 &=& 2 * u_0 * j_1 + u_1 * j_0 + 2 * v_0 + v_1 \\ &=& 2 * u_0 * j_1 + u_1 * j_0 \\ &+& 2 * (i_0 - u_0 * j_0) + i_3 - u_1 * j_2 \\ 3 * i_2 &=& u_0 * j_2 + 2 * u_1 * j_1 + v_0 + 2 * v_1 \\ &=& u_0 * j_2 + 2 * u_1 * j_1 \\ &+& i_0 - u_0 * j_0 + 2 * (i_3 - u_1 * j_2) \end{array}$$

After rearranging the equations, we obtain:

$$\begin{aligned} 2 * (j_1 - j_0) * u_0 + (j_0 - j_2) * u_1 &= 3 * i_1 - 2 * i_0 - i_3 \\ (j_2 - j_0) * u_0 + 2 * (j_1 - j_2) * u_1 &= 3 * i_2 - 2 * i_3 - i_0 \end{aligned}$$

This can be expressed as:

$$\begin{array}{rcl} u_0 &=& \dfrac{\begin{vmatrix} 2 * (j_1 - j_2) & j_0 - j_2 \\ 3 * i_2 - 2i_3 - i_0 & 3 * i_1 - 2 * i_0 - i_3 \end{vmatrix}}{\begin{vmatrix} 2 * (j_1 - j_2) & j_0 - j_2 \\ j_2 - j_0 & 2 * (j_1 - j_0) \end{vmatrix}}, \\ u_1 &=& \dfrac{\begin{vmatrix} 2 * (j_1 - j_0) & j_2 - j_0 \\ 3 * i_1 - 2 * i_0 - i_3 & 3 * i_2 - 2i_3 - i_0 \end{vmatrix}}{\begin{vmatrix} 2 * (j_1 - j_2) & j_0 - j_2 \\ j_2 - j_0 & 2 * (j_1 - j_0) \end{vmatrix}}, \\ v_0 &=& i_0 - u_0 * j_0, \\ v_1 &=& i_3 - u_1 * j_2. \end{array}$$

□

**Coplanarity Test Theorem for a VF Pair:** *For a deforming triangle, whose initial and final positions are given as ($\mathbf{a_0}$,$\mathbf{b_0}$, $\mathbf{c_0}$) and ($\mathbf{a_1}$, $\mathbf{b_1}$, $\mathbf{c_1}$) and a vertex with initial and final positions as $\mathbf{p_0}$ and $\mathbf{p_1}$, the coplanarity test can be formulated as:*

$$\begin{aligned} &Y(t) = (\mathbf{p}_t - \mathbf{a}_t) \cdot \mathbf{n}_t \\ &= k_0 * B_0^3(t) + k_1 * B_1^3(t) + k_2 * B_2^3(t) + k_3 * B_3^3(t), \quad (11) \end{aligned}$$

*where $k_{[0..3]}$ are scalars:*

$$\begin{aligned} &k_0 = (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_0, \quad k_3 = (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_1, \\ &k_1 = (2 * (\mathbf{p}_0 - \mathbf{a}_0) \cdot \hat{\mathbf{n}} + (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_0)/3, \\ &k_2 = (2 * (\mathbf{p}_1 - \mathbf{a}_1) \cdot \hat{\mathbf{n}} + (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_1)/3. \end{aligned}$$

*and*

$$\begin{aligned} &\mathbf{n_0} = (\mathbf{b_0} - \mathbf{a_0}) \times (\mathbf{c_0} - \mathbf{a_0}),\ \mathbf{n_1} = (\mathbf{b_1} - \mathbf{a_1}) \times (\mathbf{c_1} - \mathbf{a_1}), \\ &\hat{\mathbf{n}} = (\mathbf{n_0} + \mathbf{n_1} - (\mathbf{v_b} - \mathbf{v_a}) \times (\mathbf{v_c} - \mathbf{v_a})) * 0.5, \\ &\mathbf{v_a} = \mathbf{a_1} - \mathbf{a_0},\ \mathbf{v_b} = \mathbf{b_1} - \mathbf{b_0},\ \mathbf{v_c} = \mathbf{c_1} - \mathbf{c_0}. \end{aligned}$$

*Proof.* The normal vector $\mathbf{n}_t$ of the deforming triangle at time $t$ can be represented as following:

$$\mathbf{n}_t = \mathbf{n_0} * B_0^2(t) + \hat{\mathbf{n}} * B_1^2(t) + \mathbf{n_1} * B_2^2(t), \quad (12)$$

where $B_i^2(t)$ is the $i^{th}$ basis function of the Bernstein polynomials of degree 2.

We define: $\alpha = B_0^2(t) = (1-t)^2$, $\beta = B_1^2(t) = 2 * t * (1-t)$, and $\gamma = B_2^2(t) = t^2$. As a result, Equation (12) becomes:

$$\mathbf{n}_t = \mathbf{n}_0 * \alpha + \hat{\mathbf{n}} * \beta + \mathbf{n}_1 * \gamma.$$

Given the moving vertex $\mathbf{p}_t = \mathbf{p}_0 * (1-t) + \mathbf{p}_1 * t$ and a vertex of the deforming triangle $\mathbf{a}_t = \mathbf{a}_0 * (1-t) + \mathbf{a}_1 * t$, their projected distance along $\mathbf{n}_t$ is:

$$\begin{array}{rcl} (\mathbf{p}_t - \mathbf{a}_t) \cdot \mathbf{n}_t &=& ((\mathbf{p}_0 - \mathbf{a}_0) * (1-t) + (\mathbf{p}_1 - \mathbf{a}_1) * t) \cdot \mathbf{n}_t \\ &=& ((\mathbf{p}_0 - \mathbf{a}_0) * (1-t) + (\mathbf{p}_1 - \mathbf{a}_1) * t) \\ && \cdot(\mathbf{n}_0 * \alpha + \hat{\mathbf{n}} * \beta + \mathbf{n}_1 * \gamma) \\ &=& (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_0 * (1-t) * \alpha \\ &+& (\mathbf{p}_0 - \mathbf{a}_0) \cdot \hat{\mathbf{n}} * (1-t) * \beta \\ &+& (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_1 * (1-t) * \gamma \\ &+& (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_1 * t * \gamma \\ &+& (\mathbf{p}_1 - \mathbf{a}_1) \cdot \hat{\mathbf{n}} * t * \beta \\ &+& (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_0 * t * \alpha. \end{array}$$

We substitute $\alpha$, $\beta$, and $\gamma$ and obtain:

$$\begin{array}{rcl} (\mathbf{p}_t - \mathbf{a}_t) \cdot \mathbf{n}_t &=& (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_0 * (1-t)^3 \\ &+& (\mathbf{p}_0 - \mathbf{a}_0) \cdot \hat{\mathbf{n}} * 2 * (1-t)^2 * t \\ &+& (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_1 * (1-t) * t^2 \\ &+& (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_1 * t^3 \\ &+& (\mathbf{p}_1 - \mathbf{a}_1) \cdot \hat{\mathbf{n}} * 2 * t^2 * (1-t) \\ &+& (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_0 * t * (1-t)^2. \quad (13) \end{array}$$

Base on Equation (11), we have the $k_0$, $k_1$, $k_2$, and $k_3$:

$$\begin{aligned} &k_0 = (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_0, \quad k_3 = (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_1, \\ &k_1 = (2 * (\mathbf{p}_0 - \mathbf{a}_0) \cdot \hat{\mathbf{n}} + (\mathbf{p}_1 - \mathbf{a}_1) \cdot \mathbf{n}_0)/3, \\ &k_2 = (2 * (\mathbf{p}_1 - \mathbf{a}_1) \cdot \hat{\mathbf{n}} + (\mathbf{p}_0 - \mathbf{a}_0) \cdot \mathbf{n}_1)/3. \end{aligned}$$

□

**Inside Test Theorem for a VF Pair:** *Given the triangle and the vertex defined by start and end positions over the interval* $[0, 1]$*, the inside test can be formulated as:*

$$\begin{aligned} ((\mathbf{b_t} - \mathbf{p_t}) \times (\mathbf{c_t} - \mathbf{p_t})) \cdot \mathbf{n_t} &= l_0 * B_0^4(t) + l_1 * B_1^4(t) \\ &+ l_2 * B_2^4(t) + *l_3 * B_3^4(t) + l_4 * B_4^4(t), (14) \end{aligned}$$

*where $l_{[0...4]}$ are scalars:*

$$\begin{aligned} &l_0 = \mathbf{m_0} \cdot \mathbf{n_0},\ l_1 = \frac{\mathbf{m_0} \cdot \hat{\mathbf{n}} + \hat{\mathbf{m}} \cdot \mathbf{n_0}}{2},\ l_3 = \frac{\hat{\mathbf{m}} \cdot \mathbf{n_1} + \mathbf{m_1} \cdot \hat{\mathbf{n}}}{2}, \\ &l_2 = \frac{\mathbf{m_0} \cdot \mathbf{n_1} + \mathbf{4} * \hat{\mathbf{m}} \cdot \hat{\mathbf{n}} + \mathbf{m_1} \cdot \mathbf{n_0}}{6},\ l_4 = \mathbf{m_1} \cdot \mathbf{n_1}, \end{aligned}$$

*and*

$$\begin{aligned} &\mathbf{n_0} = (\mathbf{b_0} - \mathbf{a_0}) \times (\mathbf{c_0} - \mathbf{a_0}),\ \mathbf{n_1} = (\mathbf{b_1} - \mathbf{a_1}) \times (\mathbf{c_1} - \mathbf{a_1}), \\ &\hat{\mathbf{n}} = (\mathbf{n_0} + \mathbf{n_1} - (\mathbf{v_b} - \mathbf{v_a}) \times (\mathbf{v_c} - \mathbf{v_a})) * 0.5, \\ &\mathbf{m_0} = (\mathbf{b_0} - \mathbf{p_0}) \times (\mathbf{c_0} - \mathbf{p_0}),\ \mathbf{m_1} = (\mathbf{b_1} - \mathbf{p_1}) \times (\mathbf{c_1} - \mathbf{p_1}), \\ &\hat{\mathbf{m}} = (\mathbf{m_0} + \mathbf{m_1} - (\mathbf{v_b} - \mathbf{v_p}) \times (\mathbf{v_c} - \mathbf{v_p})) * 0.5, \\ &\mathbf{v_a} = \mathbf{a_1} - \mathbf{a_0},\ \mathbf{v_b} = \mathbf{b_1} - \mathbf{b_0},\ \mathbf{v_c} = \mathbf{c_1} - \mathbf{c_0},\ \mathbf{v_p} = \mathbf{p_1} - \mathbf{p_0}. \end{aligned}$$

*Proof.* Its proof is similar to the proof of **Coplanarity Test Theorem for a VF Pair**. We can replace $(\mathbf{p}_t - \mathbf{a}_t)$ with $((\mathbf{b_t} - \mathbf{p_t}) \times (\mathbf{c_t} - \mathbf{p_t}))$. □

**Simplified Inside Test Theorem for a VF pair:** *Based on combining Inequality $G(t) \geq 0$ with Equation $Y(t) = 0$ and algebraic manipulation, this inside test can be reduced to the following lower degree constraint:*

$$P(t) = p_0 * B_0^2(t) + p_1 * B_1^2(t) + p_2 * B_2^2(t) \geq 0, \tag{15}$$

*where:*

$$Y(t) = k_0 * B_0^3(t) + k_1 * B_1^3(t) + k_2 * B_2^3(t) + k_3 * B_3^3(t).$$
$$G(t) = l_0 * B_0^4(t) + l_1 * B_1^4(t)$$
$$+l_2 * B_2^4(t) + *l_3 * B_3^4(t) + l_4 * B_4^4(t)$$

*and $p_{[0\ldots2]}$ are scalars, which can be calculated based on $k_{[0\ldots3]}$ and $l_{[0\ldots4]}$.*

*Proof.* We have:

$$Y(t) = k_0 * B_0^3(t) + k_1 * B_1^3(t) + k_2 * B_2^3(t) + k_3 * B_3^3(t) = 0 \tag{16}$$

and:

$$\begin{aligned} Y(t) &= k'_0 * B_0^4(t) + k'_1 * B_1^4(t) + k'_2 * B_2^4(t) \\ &+ k'_3 * B_3^4(t) + k'_4 * B_4^4(t) = 0. \end{aligned} \tag{17}$$

Here:

$$k'_0 = k_0,\ k'_1 = \frac{k_0 + 3 * k_1}{4},\ k'_2 = \frac{k_1 + k_2}{2}$$
$$k'_3 = \frac{3 * k_2 + k_3}{4},\ k'_4 = k_3.$$

We obtain

$$\begin{aligned} G(t) * k'_0 - Y(t) * l_0 &= \\ (l_1 * k'_0 - l_0 * k'_1) * B_1^4(t) &+ \\ (l_2 * k'_0 - l_0 * k'_2) * B_2^4(t) &+ \\ (l_3 * k'_0 - l_0 * k'_3) * B_3^4(t) &+ \\ (l_4 * k'_0 - l_0 * k'_4) * B_4^4(t). \end{aligned} \tag{18}$$

This Equation can be expressed as:

$$s_0 * B_0^3(t) + s_1 * B_1^3(t) + s_2 * B_2^3(t) + s_3 * B_3^3(t). \tag{19}$$

Here:

$$\begin{aligned} s_0 &= (l_1 * k'_0 - l_0 * k'_1) * 4 \\ s_1 &= (l_2 * k'_0 - l_0 * k'_2) * 2 \\ s_2 &= \frac{(l_3 * k'_0 - l_0 * k'_3) * 4}{3} \\ s_3 &= l_4 * k'_0 - l_0 * k'_4 \end{aligned} \tag{20}$$

And:

$$\begin{aligned} (G(t) * k'_0 - Y(t) * l_0) * k_0 - Y(t) * s_0 &= \\ (s_1 * k_0 - s_0 * k_1) * B_1^3(t) &+ \\ (s_2 * k_0 - s_0 * k_2) * B_2^3(t) &+ \\ (s_3 * k_0 - s_0 * k_3) * B_3^3(t). \end{aligned} \tag{21}$$

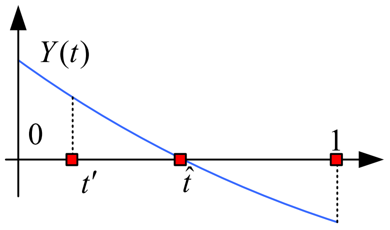
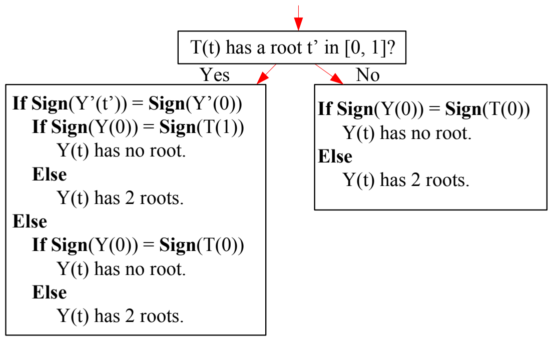


**Figure 1: Side Determination Theorem I:** *Given a $t' \in [0, 1]$, if $Y(t')$ has the same sign of $Y(0)$, then $t' < \hat{t}$, else $t' > \hat{t}$.*



```
If Sign(Y'(t')) = Sign(Y'(0))
    If Sign(Y(0)) = Sign(T(1))
        Y(t) has no root.
    Else
        Y(t) has 2 roots.
Else
    If Sign(Y(0)) = Sign(T(0))
        Y(t) has no root.
    Else
        Y(t) has 2 roots.
```

```
If Sign(Y(0)) = Sign(T(0))
    Y(t) has no root.
Else
    Y(t) has 2 roots.
```

**Figure 2: Computing the Number of Roots of $Y(t)$:** *We can compute them based on sign evaluations.*

This Equation can be expressed as:

$$p_0 * B_0^2(t) + p_1 * B_1^2(t) + p_2 * B_2^2(t). \tag{22}$$

Here:

$$\begin{aligned} p_0 &= (s_1 * k_0 - s_0 * k_1) * 3 \\ p_1 &= \frac{(s_2 * k_0 - s_0 * k_2) * 3}{2} \\ p_2 &= s_3 * k_0 - s_0 * k_3 \end{aligned} \tag{23}$$

Sine $Y(t) = 0$, we have:

$$\begin{aligned} P(t) &= (G(t) * k'_0 - Y(t) * l_0) * k_0 - Y(t) * s_0 \\ &= G(t) * k'_0 * k_0 = G(t) * k_0 * k_0. \end{aligned}$$

So $P(t)$ has the same sign of $G(t)$. □

**Side Determination Theorem I:** *Given a polynomial $Y(t)$, which has only one root $\hat{t} \in [0, 1]$ and $Y(\hat{t}) = 0$.* **Sign**$(Y(0)) \neq$ **Sign**$(Y(1))$. *Given a $t' \in [0, 1]$, if $Y(t')$ has the same sign as $Y(0)$, then $t' < \hat{t}$, otherwise $t' > \hat{t}$.*

*Proof.* We prove it using contradiction: If $Y(t')$ has the same sign of $Y(0)$ and $t' > \hat{t}$, then in the interval $[t', 1]$, **Sign**$(Y(t')) \neq$ **Sign**$(Y(1))$. Since Y(t) is a continuous function, it must have another root in the domain $[t', 1]$. This is contradictory to the fact that $\hat{(t)}$ is the only root in the domain $[0, 1]$. □

**Root Finding Lemma:** *For a cubic polynomial $Y(t)$ with an extreme point in its domain, its 1st derivative $Y'(t)$ is:*

$$\begin{aligned} Y'(t) &= 3 * (k_1 - k_0) * B_0^2(t) + 3 * (k_2 - k_1) * B_1^2(t) \\ &+ 3 * (k_3 - k_2) * B_2^2(t). \end{aligned}$$

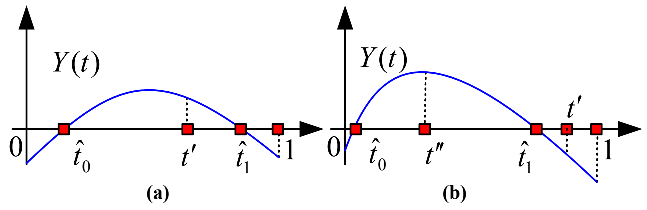


**Figure 3: Sign Determination Theorem II:** *For a $t' \in [0,1]$, if $Y(t')$ has the same sign of $Y(0)$, then $t' < \hat{t}$, else $t' > \hat{t}$.*

*We have $Y(t) = Y'(t) * L(t) + K(t)$, where $L(t)$ and $K(t)$ are two linear polynomials and can be calculated with the* **Polynomial Decomposition Theorem**. *We can use the rules in Fig. 2 to compute the number of roots of $Y(t)$.*

*Proof.* We define $t''$ as the only root of $Y'(t)$, i.e., $Y'(t'') = 0$. We need to determine the sign of $Y(t'')$. If $\mathbf{Sign}(Y(t'')) = \mathbf{Sign}(Y(0))$ then $Y(t)$ has no root, otherwise $Y(t)$ must have two roots in $[0,1]$. Since $Y(t'') = Y'(t'') * L(t'') + K(t'') = K(t'')$, $\mathbf{Sign}(Y(t'')) = \mathbf{Sign}(K(t''))$. So we need to determine the sign of $K(t'')$.

If $K(t)$ has no root in $[0,1]$, then $\mathbf{Sign}(K(t'')) = \mathbf{Sign}(K(0))$.

Otherwise, let $t'$ be the only root of $K(t)$ in $[0,1]$. Based on the **Side Determination Theorem I**, if $\mathbf{Sign}(Y'(t')) = \mathbf{Sign}(Y'(0))$, $t' < t''$, else $t' > t''$. If $t' < t''$, then $\mathbf{Sign}(K(t'')) = \mathbf{Sign}(K(1))$, else $\mathbf{Sign}(K(t'')) = \mathbf{Sign}(K(0))$. □

**Side Determination Theorem II:** *For a given polynomial $Y(t)$, in a domain $[0,1]$ which has two roots $\hat{t}_0$ and $\hat{t}_1$. $\mathbf{Sign}(Y(0)) = \mathbf{Sign}(Y(1))$. For $t' \in [0,1]$, if $Y(t')$ has a different sign than $Y(0)$ (Fig. 3(a)), then $\hat{t}_0 < t' < \hat{t}_1$, otherwise $\hat{t}_0, \hat{t}_1$ are on the same side of $t'$ (Fig. 3(b)).*

*Proof.* If $Y(t')$ has a different sign as compared to $Y(0)$, then $t' \in [\hat{t}_0, \hat{t}_1]$; otherwise:

- If $t' < \hat{t}_0$, $\mathbf{Sign}(Y(t')) \neq \mathbf{Sign}(Y(0))$, there is another root in the interval [0, t'], this contradicts the fact $Y(t)$ has two roots.
- If $t' > \hat{t}_1$, $\mathbf{Sign}(Y(t')) \neq \mathbf{Sign}(Y(1))$, there is another root in the interval [t', 1], this contradicts the fact that $Y(t)$ has two roots.

If $\mathbf{Sign}(Y(t')) = \mathbf{Sign}(Y(0)) = \mathbf{Sign}(Y(1))$, then $t' \ni [\hat{t}_0, \hat{t}_1]$, otherwise, $\mathbf{Sign}(Y(t')) = \mathbf{Sign}(Y(t''))$. This contradicts the fact that $\mathbf{Sign}(Y(0)) \neq \mathbf{Sign}(Y(t''))$, where $t''$ is the extreme point. □

**Sign Determination Theorem I:** *Let $L(t)$ be a linear polynomial and $Y(t)$ be a cubic polynomial which corresponds to the Bézier curve of Case (b) (Section 3.2) in the domain $[0,1]$. Let:*

- *$L(t') = 0$, and $t' \in [0,1]$.*
- *$Y(\hat{t}) = 0$, and $\hat{t} \in [0,1]$.*

*We have:*

```
If $\mathbf{Sign}(Y(t')) = \mathbf{Sign}(Y(0))$
    $\mathbf{Sign}(L(\hat{t})) = \mathbf{Sign}(L(1))$
Else
    $\mathbf{Sign}(L(\hat{t})) = \mathbf{Sign}(L(0))$
Endif
```

*Proof.* With **Side Determination Theorem I**, we have: if $\mathbf{Sign}(Y(t')) = \mathbf{Sign}(Y(0))$, then $t' < \hat{t} \Rightarrow \mathbf{Sign}(L(\hat{t}) = \mathbf{Sign}(L(1))$, else $t' > \hat{t} \Rightarrow \mathbf{Sign}(L(\hat{(t)}) = \mathbf{Sign}(L(0))$. □

**Sign Determination Theorem II:** *Let $L(t)$ be a linear polynomial and $Y(t)$ be a cubic polynomial which corresponds to the Bézier curve of Case (c) in the domain $[0,1]$. Let:*

- *$L(t') = 0$, and $t' \in [0,1]$.*
- *$Y(\hat{t}_0) = 0$ and $Y(\hat{t}_1) = 0$, and $\hat{t}_0 \in [0,1], \hat{t}_1 \in [0,1], \hat{t}_0 < \hat{t}_1$.*
- *$Y'(t'') = 0$, and $t'' \in [0,1]$. $Y'(t)$ is the 1st order derivative of $Y(t)$.*

*We have:*

```
If $\mathbf{Sign}(Y(t')) \neq \mathbf{Sign}(Y(0))$
    $\mathbf{Sign}(L(\hat{t}_0)) = \mathbf{Sign}(L(0))$
    $\mathbf{Sign}(L(\hat{t}_1)) = \mathbf{Sign}(L(1))$
Else
    If $\mathbf{Sign}(Y'(t')) = \mathbf{Sign}(Y'(0))$
        $\mathbf{Sign}(L(\hat{t}_0)) = \mathbf{Sign}(L(1))$
        $\mathbf{Sign}(L(\hat{t}_1)) = \mathbf{Sign}(L(1))$
    Else
        $\mathbf{Sign}(L(\hat{t}_0)) = \mathbf{Sign}(L(0))$
        $\mathbf{Sign}(L(\hat{t}_1)) = \mathbf{Sign}(L(0))$
    Endif
Endif
```

*Proof.* With **Side Determination Theorem II**, we have: If $\mathbf{Sign}(Y(t')) \neq \mathbf{Sign}(Y(0))$, then $\hat{t}_0 < t' < \hat{t}_1 \Rightarrow \mathbf{Sign}(L(\hat{t}_0) = \mathbf{Sign}(L(0))$ and $\mathbf{Sign}(L(\hat{t}_1) = \mathbf{Sign}(L(1))$. Otherwise, $\hat{t}_0$, $\hat{t}_1$, and $t''$ are at the same side of $t'$, and with **Side Determination Theorem I**, we obtain:

if $\mathbf{Sign}(Y'(t')) = \mathbf{Sign}(Y'(0))$, then $t' < t''$, otherwise $t' > t''$.

$t' < t'' \Rightarrow t' < \hat{t}_0$ and $t' < \hat{t}_1 \Rightarrow \mathbf{Sign}(L(\hat{t}_0)) = \mathbf{Sign}(L(1))$ and $\mathbf{Sign}(L(\hat{t}_1)) = \mathbf{Sign}(L(1))$.

$t' > t'' \Rightarrow t' > \hat{t}_0$ and $t' > \hat{t}_1 \Rightarrow \mathbf{Sign}(L(\hat{t}_0)) = \mathbf{Sign}(L(0))$ and $\mathbf{Sign}(L(\hat{t}_1)) = \mathbf{Sign}(L(0))$. □

## 2 Error Bound for Conservative Culling

The conservative culling algorithm is described in Section 4.3. It uses a floating-point filter and we present an error-bound for that filter. Since our computation only uses addition, substraction, and multiple operations, it is relatively simple to derive such a bound.

According to the IEEE 754 standard, given an exact arithmetic operator $\times$ and its floating point counterpart $\bigotimes$, $a \times b = ROUND(a \bigotimes b)$ and $|a \bigotimes b - a \times b| \leq |a \times b| * \epsilon$. For the double precision format, $\epsilon = 2^{-52}$.

We use following rules to evaluate the error bounds for addition/subtraction and , repsetively:

**Rule I: Error bound for addition/substraction:** Give two numbers with rounding errors, i.e. $a + c_1 * \epsilon$ and $b + c_2 * \epsilon$, rounding error of the addition/substration operation on them will be bounded by:

$$\begin{aligned} \Delta &= (a + c_1 * \epsilon) \pm (b + c_2 * \epsilon) \\ &+ ||(a + c_1 * \epsilon) \pm (b + c_2 * \epsilon)|| * \epsilon \\ &= a \pm b + (c_1 \pm c_2) * \epsilon + ||a \pm b||) * \epsilon + O(\epsilon^2) \\ &< a \pm b + (c_1 \pm c_2) * \epsilon + (||a \pm b|| + 1) * \epsilon \end{aligned} \tag{24}$$

The accumulate rounding error is bounded by $(c1 \pm c2 + ||a \pm b|| + 1) * \epsilon$.

**Rule II: Error bound for multiply operation:** The multiple of $a + c_1 * \epsilon$ and $b + c_2 * \epsilon$ is $\Delta$ (with rounding error):

$$\begin{aligned} \Delta &= (a + c_1 * \epsilon) * (b + c_2 * \epsilon) \\ &+ ||(a + c_1 * \epsilon) * (b + c_2 * \epsilon)|| * \epsilon \\ &= a * b + (b * c_1 + a * c_2 + ||a * b||) * \epsilon + O(\epsilon^2) \\ &< a * b + (b * c_1 + a * c_2 + ||a * b|| + 1) * \epsilon \end{aligned} \tag{25}$$

The accumulative rounding error is bounded by $(b * c_1 + a * c_2 + ||a * b|| + 1) * \epsilon$.

In order to perform conservative culling, we need to test the signs of

$$\begin{aligned} k_0 &= (\mathbf{p_0} - \mathbf{a_0}) \cdot \mathbf{n_0}, \quad k_3 = (\mathbf{p_1} - \mathbf{a_1}) \cdot \mathbf{n_1}, \\ k_1 &= (2 * (\mathbf{p_0} - \mathbf{a_0}) \cdot \hat{\mathbf{n}} + (\mathbf{p_1} - \mathbf{a_1}) \cdot \mathbf{n_0})/3, \\ k_2 &= (2 * (\mathbf{p_1} - \mathbf{a_1}) \cdot \hat{\mathbf{n}} + (\mathbf{p_0} - \mathbf{a_0}) \cdot \mathbf{n_1})/3. \end{aligned}$$

and

$$\begin{aligned} \mathbf{n_0} &= (\mathbf{b_0} - \mathbf{a_0}) \times (\mathbf{c_0} - \mathbf{a_0}), \ \mathbf{n_1} = (\mathbf{b_1} - \mathbf{a_1}) \times (\mathbf{c_1} - \mathbf{a_1}), \\ \hat{\mathbf{n}} &= (\mathbf{n_0} + \mathbf{n_1} - (\mathbf{v_b} - \mathbf{v_a}) \times (\mathbf{v_c} - \mathbf{v_a})) * 0.5, \\ \mathbf{v_a} &= \mathbf{a_1} - \mathbf{a_0}, \ \mathbf{v_b} = \mathbf{b_1} - \mathbf{b_0}, \ \mathbf{v_c} = \mathbf{c_1} - \mathbf{c_0}. \end{aligned}$$

Let $\mathbf{n_v} = (\mathbf{v_b} - \mathbf{v_a}) \times (\mathbf{v_c} - \mathbf{v_a})$, for $k_1$ and $k_2$, it is equivalent to testing:

$$\begin{aligned} k_1' &= 2 * (\mathbf{p_0} - \mathbf{a_0}) \cdot \hat{\mathbf{n}} + (\mathbf{p_1} - \mathbf{a_1}) \cdot \mathbf{n_0} \\ &= (\mathbf{p_0} - \mathbf{a_0}) \cdot (\mathbf{n_0} + \mathbf{n_1} - (\mathbf{v_b} - \mathbf{v_a}) \times (\mathbf{v_c} - \mathbf{v_a})) \\ &+ (\mathbf{p_1} - \mathbf{a_1}) \cdot \mathbf{n_0}, \\ &= (\mathbf{p_0} - \mathbf{a_0}) \cdot (\mathbf{n_0} + \mathbf{n_1} - \mathbf{n_v}) + (\mathbf{p_1} - \mathbf{a_1}) \cdot \mathbf{n_0}, \end{aligned} \tag{26}$$

Similarly, we have:

$$k_2' = (\mathbf{p_1} - \mathbf{a_1}) \cdot (\mathbf{n_0} + \mathbf{n_1} - \mathbf{n_v}) + (\mathbf{p_0} - \mathbf{a_0}) \cdot \mathbf{n_1}. \tag{27}$$

For $k_0, k_1', k_2', k_3$, their rounding errors are sum of $\mathbf{i} \times \mathbf{j} \cdot \mathbf{k}$, where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are vectors, and

$$\begin{aligned} \mathbf{i} \times \mathbf{j} \cdot \mathbf{k} &= i_y * j_z * k_x - i_z * j_y * k_x \\ &+ i_z * j_x * k_y - i_x * j_z * k_y \\ &+ i_x * j_y * k_z - i_y * j_x * k_z \end{aligned}$$

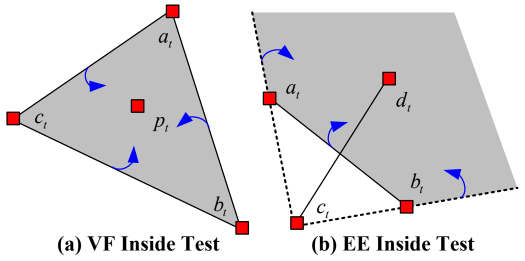


**Figure 4: Inside Tests:** *For VF and EE pairs, we need to perform inside tests to check if the vertex is inside the triangle, or the two edges are intersecting with each other, when their vertices are coplanar.*

We use the **Rule I** and **Rule II** to accumulate the rounding errors, and compute the error bounds for $k_0, k_1', k_2', k_3$ on-the-fly. These dynamically computed error bounds are used by the filtering algorithm.

## 3 EE Query Algorithm

As shown in Fig. 4(b), in order to perform an inside test for a EE pair, we need to perform three one-sided tests to make sure the two edges, $\mathbf{a}_t\mathbf{b}_t$ and $\mathbf{c}_t\mathbf{d}_t$, intersect with each other. This can be expressed based on the following inequalities:

$$((\mathbf{b}_t - \mathbf{d}_t) \times (\mathbf{c}_t - \mathbf{d}_t)) \cdot \mathbf{n}_t \geq 0, \tag{28}$$
$$((\mathbf{c}_t - \mathbf{d}_t) \times (\mathbf{a}_t - \mathbf{d}_t)) \cdot \mathbf{n}_t \geq 0, \tag{29}$$
$$((\mathbf{a}_t - \mathbf{d}_t) \times (\mathbf{b}_t - \mathbf{d}_t)) \cdot \mathbf{n}_t \leq 0. \tag{30}$$

The only different between EE query algorithm vs VF query algorithm is to use these inequalities for the inside tests. The rest of the formulation and algorithm structure is the same.

## 4 Avoiding Division Operations

A key aspect of the algorithm is that we don't perform any division operations. In practice, division operations are more expensive in the context of extended precision computation and it is harder to obtain tight error bounds for floating-point filters.

For a linear polynomial $L(t) = a * B_0^1(t) + b * B_1^1(t)$, its root is give $t' = \frac{a}{a-b}$. We do not need to perform the division by $(a - b)$, since we only need to:

- Check if $L(t)$ has a root in $[0, 1]$: We can check the signs of $a$, $b$. If they have the same sign, there is no root in $[0, 1]$, otherwise, there is 1 root in $[0, 1]$.
- Evaluate $Y(t')$, where $Y(t)$ is a cubic or quadratic polynomial in Bernstein form. In our algorithm, we only need to know the sign of Y(t). For a cubic polynomial: If $(a - b) > 0$, $\mathbf{Sign}(Y(\frac{a}{a-b}) = \mathbf{Sign}(Y(a))$, otherwise $\mathbf{Sign}(Y(\frac{a}{a-b}) = -\mathbf{Sign}(Y(a))$. For a quadratic polynomial, $\mathbf{Sign}(Y(\frac{a}{a-b}) = \mathbf{Sign}(Y(a))$.

In both these cases, we can compute the signs of the expression without explicitly performing a division operations.