# Revisiting the Problem of Zeros of Univariate Scalar Béziers

Jinesh Machchhar[a,*], Gershon Elber[a]

[a]*Faculty of Computer Science, Technion Israel Institute of Technology, Israel*

**Abstract**

This paper proposes a fast algorithm for computing the real roots of univariate polynomials given in the Bernstein basis. Traditionally, the polynomial is subdivided until a root can be isolated. In contrast, herein we aim to find a root only to subdivide the polynomial at the root. This subdivision based algorithm exploits the property that the Bézier curves interpolate the end-points of their control polygons. Upon subdivision at the root, both resulting curves contain the root at one of their end-points, and hence contain a vanishing coefficient that is factored out. The algorithm then recurses on the new sub-curves, now of lower degree, yielding a computational efficiency. In addition, the proposed algorithm has the ability to efficiently count the multiplicities of the roots. Comparison of running times against the state-of-the-art on thousands of polynomials shows an improvement of about an order-of-magnitude.

## 1. Introduction and previous work

The problem of numerically finding zeros of univariate polynomials is ubiquitous in computer aided design [5] and engineering. Many geometric problems can be cast into that of finding zeros of polynomials, for instance, computing intersections of curves and surfaces [27, 29], contact analysis of shapes [14], kinematic analysis [2], etc. There have been many approaches to the problem of computing zeros of univariate polynomials in the past [13, 20], such as, based on Newton's method [11], using Descartes' rule of signs [6, 16, 25], based on subdivision [10, 22], to name a few.

In this work, we use the Bernstein representation for polynomials. Bernstein polynomials have several useful properties such as the variation diminishing property, the convex hull property and numerical stability with respect to perturbation of coefficients [8], which makes such a representation especially amenable for numerical applications.

One of the earliest methods to exploit the variation diminishing property of Bézier curves in order to isolate the roots of polynomials was given by Lane and Riesenfeld [17] in 1981. In 1990, Sederberg and Nishita [28] proposed the technique of Bézier clipping for identifying regions of the domain which contain roots. This was done by intersecting the convex hull of the control polygon with the zero axis. In 2007, Bartoň and Jüttler [1] improved the technique of Bézier clipping using degree reduction to generate a strip bounded by two quadratic polynomials, which encloses the graph of the input polynomial. This strip, when intersected with the zero axis, gives the new interval potentially containing the roots. This approach, that is also known as *quadratic clipping*, was shown to have quadratic convergence by Schulz [26]. In 2009, Liu et al. [18] improved quadratic clipping by using cubic polynomials, yielding faster rates of convergence. In 2007, Mørken and Reimers [21] utilized the close relationship between the spline and its control polygon for computing zeros of polynomials. They use the zeros of the control polygon as an initial guess for tracing the zeros of the polynomial. The control polygon is iteratively refined until the roots are found. In 2013, Ko and Kim [15] used bounding polygons to reduce the intervals containing roots of Bézier polynomials. A hybrid of the convex hull, sharp bounds [23] and quasi-interpolating bounds [31] was used to refine the

*Corresponding author

*Email addresses:* jineshmac@cs.technion.ac.il (Jinesh Machchhar), gershon@cs.technion.ac.il (Gershon Elber)

*Preprint submitted to Elsevier* *February 14, 2016*

intervals. Recently, in 2015, Chen et al. [4] improved the convergence rates achieved by Liu et al. [18] by bounding the polynomial of interest by a pair of rational cubic polynomials.

Our approach, as explained in Section 1.1 uses the fact that polynomials represented in the Bernstein-Bézier form admit efficient algorithms for polynomial multiplication and division [3, 9]. These are employed to factor out the roots already computed, thus reducing the degree of the polynomial, as part of the solution process.

### *1.1. Overview of our approach*

We now give an overview of our method, which exploits several properties of the Bézier curves, for computing zeros of scalar polynomials. The Bernstein basis for an $n$-degree univariate polynomial is given by the set of functions $\theta_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$, $t \in [0,1]$, for $i = 0, \ldots, n$. A degree $n$ scalar polynomial in the Bernstein basis is expressed as $c(t) = \sum_{i=0}^{n} p_i \theta_{i,n}(t)$ where, $p_i \in \mathbb{R}$ are the real coefficients of the polynomial. While $c(t)$ may have roots outside the domain $[0,1]$, in this paper we focus on finding the real roots of $c(t)$ in $[0,1]$. Throughout this paper, we will consider $[0,1]$ to be the domain of definition of all Bézier curves, unless stated otherwise.

We will exploit the fact that the Bézier curves interpolate the end-points of their control polygon. Upon finding a root, $t_0$, our algorithm subdivides $c(t)$ at $t_0$. Let $c_l(t)$ and $c_r(t)$ denote the resulting polynomials corresponding to the sub-intervals $[0, t_0]$ and $[t_0, 1]$, respectively, with their domains again mapped to $[0,1]$. Clearly, $c_l(t)$ and $c_r(t)$ vanish at 1 and 0, respectively. Since the Bézier curves interpolate the end-points of their control polygon, it follows that the last coefficient of $c_l(t)$ and the first coefficient of $c_r(t)$ are zero. $c_l(t)$ may be expressed as $(1-t)c_L(t)$ for some Bernstein polynomial $c_L(t)$ having one degree less than $c_l(t)$. Similarly, $c_r(t)$ may be written as $tc_R(t)$ for some Bernstein polynomial $c_R(t)$ with one degree less than $c_r(t)$. Factoring out the terms $(1-t)$ and $t$ from $c_l(t)$ and $c_r(t)$, respectively [3], eliminates the root $t_0$ from these two polynomials, yielding polynomials with smaller degrees to recurse upon that preserves all roots but $t_0$, thus giving a computational boost to the algorithm. This is explained schematically in Figure 1. The graph of polynomial $c(t)$ is shown in red, in Figure 1(a), along with its control polygon that is shown in green. $c(t)$ has two real roots, at $t_1$ and $t_2$. Assume we found the root at $t_2$. Figure 1(b) shows the subdivided polynomials $c_l(t)$ and $c_r(t)$ respectively, both containing the root, $t_2$, at the respective end-point of their domains, and hence have a vanishing coefficient there. Figure 1(c) shows the lower degree polynomials obtained after factoring out $(1-t)$ and $t$ from $c_l(t)$ and $c_r(t)$, respectively.

The rest of this paper is organized as follows. Our basic algorithm is explained in Section 2. An interesting efficient feature of our approach to count the multiplicities of roots, along with two more extensions of the algorithm are given in Section 3. One extension lets the search for roots go outside the domain of interest, and another initializes the Newton-Raphson method with better calculated seeds compared to the basic algorithm. We compare the running times of our implementation of the algorithm with those of the state-of-the-art alternatives, on thousands of polynomials. The results of the comparison, as given in Section 4, show an improvement of about an order-of-magnitude. Finally, we conclude the paper, in Section 5, with remarks on future work.

## 2. Root finding: the basic algorithm

In this section, we explain our algorithm for finding the real roots of univariate scalar polynomials, given in Bernstein form. The proposed algorithm is based on subdivisions at detected roots.

As explained in Section 1.1, the proposed algorithm achieves a reduction in the complexity of the problem by factoring out all the roots that are already found. A scalar Bernstein polynomial $c(t)$ of degree $n$ with $c(0) = 0$ may be expressed as $c(t) = tr(t)$, for some Bernstein polynomial $r(t)$ with degree $n-1$ [3]. The coefficients of $r(t)$ are obtained from those of $c(t)$ as $q_i = p_{i+1}\frac{n}{i+1}$, for $i = 0, \ldots, n-1$. The proof appears in Lemma 1, for completeness:

**Lemma 1.** *A degree $n$ scalar Bernstein polynomial $c(t) = \sum_{i=0}^{n} p_i \theta_{i,n}$ such that $c(0) = 0$ can be written as $c(t) = t \sum_{i=0}^{n-1} q_i \theta_{i,n-1} = tr(t)$, where, $q_i = p_{i+1}\frac{n}{i+1}$ for $i = 0, \ldots, n-1$.*

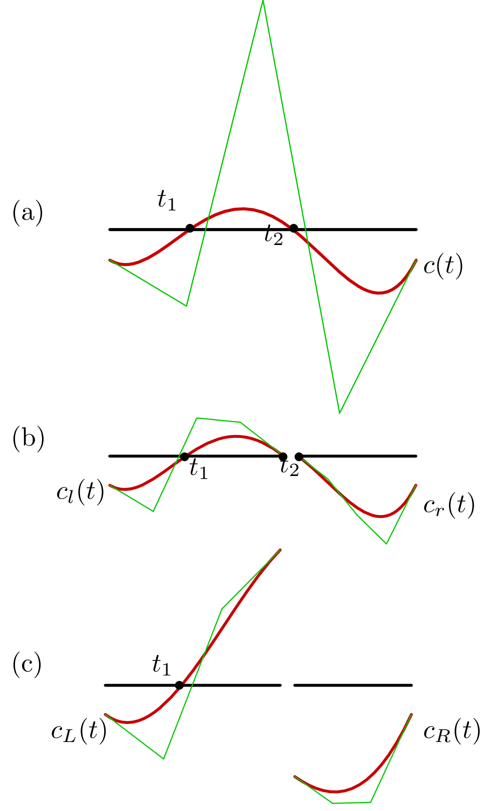


Figure 1: Overview of our approach. (a) Shows the graph of a degree 4 Bézier polynomial $c(t)$ in red with roots at $t_1$ and $t_2$, along with its control polygon in green. The $t$-axis is shown in black. (b) shows the polynomials $c_l(c)$ and $c_r(t)$ obtained after subdividing $c(t)$ at $t_2$. $c_l(t)$ and $c_r(t)$ are again of degree 4 and contain the root at $t_2$. (c) shows the degree 3 polynomials obtained after eliminating the root at $t_2$ from $c_l(t)$ and $c_r(t)$, factoring out $(1-t)$ and $t$, respectively.

**Proof.** Since $c(0) = 0$, the first coefficient, $p_0$, is zero, and we have,

$$\begin{aligned}
c(t) &= \sum_{i=0}^{n} p_i \binom{n}{i} t^i (1-t)^{n-i}, \\
&= \sum_{i=1}^{n} p_i \binom{n}{i} t^i (1-t)^{n-i}, \\
&= t \sum_{i=1}^{n} p_i \binom{n}{i} t^{i-1} (1-t)^{n-i}, \\
&= t \sum_{i=0}^{n-1} p_{i+1} \binom{n}{i+1} t^i (1-t)^{n-i-1}, \\
&= t \sum_{i=0}^{n-1} p_{i+1} \frac{n}{i+1} \binom{n-1}{i} t^i (1-t)^{n-i-1}, \\
&= t \sum_{i=0}^{n-1} q_i \theta_{i,n-1}(t), \\
&= tr(t),
\end{aligned}$$

where $q_i = p_{i+1} \frac{n}{i+1}$, for $i = 0, \ldots, n-1$, are the coefficients of the scalar Bernstein polynomial $r(t)$ having

degree $n-1$. □

Clearly, the term $t$ can be factored out from $c(t)$ in linear time with respect to the number of coefficients in $c(t)$. In a similar way, a degree $n$ scalar Bernstein polynomial $c(t)$ with $c(1) = 0$ may be expressed as $(1-t)s(t)$, for some scalar Bernstein polynomial $s(t)$ with degree $n-1$, whose coefficients are obtained from those of $c(t)$ as $p_i \frac{n}{n-i}$ for $i = 0, \ldots, n-1$. The proof is similar to that given in Lemma 1. More importantly, the following holds:

**Remark 2.** *Let the set of real roots of $c(t)$ be $\mathcal{R}_c$ and $c(0) = 0$. Then,*

$$\mathcal{R}_c = \mathcal{R}_r \cup \{0\},$$

*where, $c(t) = tr(t)$ and $\mathcal{R}_r$ is the set of real roots of $r(t)$.*

In other words, the set of real roots of $r(t)$ identifies with the roots of $c(t)$ up to the root at zero and hence, we can continue working with $r(t)$ instead of $c(t)$, without missing any root, but in a reduced complexity. A similar remark holds for a polynomial $c(t)$ with $c(1) = 0$.

As a first step, our algorithm employs a sufficient condition for discarding sub-domains which do not contain roots, by inspecting the signs of the coefficients of $c(t)$. If they are either all positive or all negative, the sub-domain does not contain roots and is purged. This follows from the convex-hull property of the Bézier curves.

If the domain is not discarded, an attempt is made to find a root in the current domain, numerically. We employ the Newton-Raphson method, which is known to have a quadratic rate of convergence [13], for this purpose and use an initial guess of 0.5 as an initial seed value. Upon successful finding of a root, $t_0$, the polynomial $c(t)$ is subdivided at $t_0$. As explained previously, both curves resulting from the subdivision of $c(t)$ contain the root, $t_0$, at the respective end-points of their domains. The root at $t_0$ is factored out from these two new curves and the resulting lower degree polynomials are recursed upon. Alternatively, if no root is found by the numeric step, $c(t)$ is subdivided in the middle of the domain and the resulting polynomials are recursed upon. The stopping criteria for the Newton-Raphson method is either a divergent step, or the search going outside the domain, or the number of iterations exceeding a limit, 100 in our case (a case that never occurred in all our tests).

Our method is summarized in Algorithm 1. The test for absence of roots (all coefficients positive or all negative) is performed by the routine PurgeProblem in Line 1 of Algorithm 1. The Newton-Raphson method is invoked in Line 4 by the routine NumericStep, with the mid-point (0.5) of the domain as the starting seed. The case when a root is found by NumericStep is handled in Lines 6 to 9 while the case when no root is found is executed in Lines 14 to 16. Note that we assume the curves are always within domain $[0, 1]$. Yet, we keep track of the real domain by propagating the $[t^{min}, t^{max}]$ values.

Algorithm 1 uses two tolerances, the numeric tolerance, $\epsilon$, and the subdivision tolerance, $\delta$. The roots are searched up to $\epsilon$, i.e., for each root, $t_0$, returned, $-\epsilon < c(t_0) < \epsilon$, while the minimal width of the domain of any subdivided curve, to be considered by our algorithm, is set by $\delta$. The termination of Algorithm 1 stems from looking at its two sub cases. If a root is found by the numeric step, the algorithm recurses upon two sub polynomials of one degree less. If no root is found by the numeric step, the algorithm recurses on two polynomials, each with domain width half that of the original polynomial. Hence, the algorithm terminates when either one of the following conditions hold: (i) the width of the problem domain $[t^{min}, t^{max}]$ falls below the subdivision tolerance, or (ii) the control polygon of $c(t)$ does not cross the $t$-axis, or (iii) the degree of $c(t)$ is one.

## 3. Extensions of the algorithm

We now consider several extensions of the basic algorithm from Section 2. In Section 3.1, we present an extension for counting the multiplicities of roots. In Section 3.2, we consider an alternative method of initializing the Newton-Raphson method and in Section 3.3, we portray an extended framework which lets the search for roots go outside the domain of interest.

**Algorithm 1** BézierZeroFactored($c, t^{min}, t^{max}, \epsilon, \delta$)

1: **if** PurgeProblem($c, \epsilon$) **then**
2:     **return** $\emptyset$;
3: **end if**
4: $t_0 \leftarrow$ NumericStep($c, \epsilon, 0.5$);
5: **if** $t_0 != \emptyset$ **then**
6:     $(c_l, c_r) \leftarrow$ Subdivide($c, t_0$);
7:     $c_L \leftarrow$ Factor1MinusT($c_l$);
8:     $c_R \leftarrow$ FactorT($c_r$);
9:     **return** BézierZeroFactored($c_L, t^{min}, t_0, \epsilon, \delta$) $\cup \{t_0\} \cup$ BézierZeroFactored($c_R, t_0, t^{max}, \epsilon, \delta$);
10: **else**
11:     **if** $t^{max} - t^{min} < \delta$ **then**
12:         **return** $\emptyset$;
13:     **end if**
14:     $t_0 \leftarrow \frac{t^{max}+t^{min}}{2}$;
15:     $(c_l, c_r) \leftarrow$ Subdivide($c, 0.5$);
16:     **return** BézierZeroFactored($c_l, t^{min}, t_0, \epsilon, \delta$) $\cup$ BézierZeroFactored($c_r, t_0, t^{max}, \epsilon, \delta$);
17: **end if**

### *3.1. Counting multiplicities of roots*

The extended ability to count the multiplicities of roots is naturally supported by our computational framework. Each time a root is factored out, the terminal coefficient of the resulting polynomial is inspected again. A vanishing coefficient, again after an elimination of $t$ or $(1-t)$, implies that the root is repeated. Hence, counting multiplicities is reduced to the examination of a single (terminal) scalar coefficient of $c_r(t)$ to be zero, and hence, is highly efficient. This is demonstrated schematically in Figure 2 for a double root. The pseudo-code for counting multiplicities of roots is given in Algorithm 2 which replaces Lines 7 and 8 in Algorithm 1. Herein, $m_0$ is the desired root multiplicity. For clarity, both input and output curves are designated as $c_l$ and $c_r$ in Algorithm 2 (against $c_L$ and $c_R$ in Algorithm 1) due to the recursive computation. This extended ability is examined in Section 4. Note that, the Newton-Raphson method converges linearly to a root with multiplicity greater than one [13].

**Algorithm 2** CountRootMultiplicities (substituted for Lines 7-8 in Algorithm 1)

1: $m_0 \leftarrow 0$;
2: **do**
3:     $c_l \leftarrow$ Factor1MinusT($c_l$);
4:     $c_r \leftarrow$ FactorT($c_r$);
5:     $m_0 \leftarrow m_0 + 1$;
6: **while** $|c_r(0)| < \epsilon$

### *3.2. Initialization of Newton-Raphson method*

It is known that the control polygon of a Bézier curve is an approximation of the curve itself. One can compute the intersection of the control polygon of $c(t)$ with the $t$-axis, in order to supply a better initial value for the Newton-Raphson method, following [21]. Due to the variation diminishing property, the number of intersections of the control polygon with the $t$-axis is greater than or equal to the number of real roots of $c(t)$. In this extension option, we examined both the first intersection point as well as the intersection point closest to the mid-point of the domain, between the control polygon and the $t$-axis. The potential computational benefits of both alternative initializations of the Newton-Raphson method are examined in Section 4.

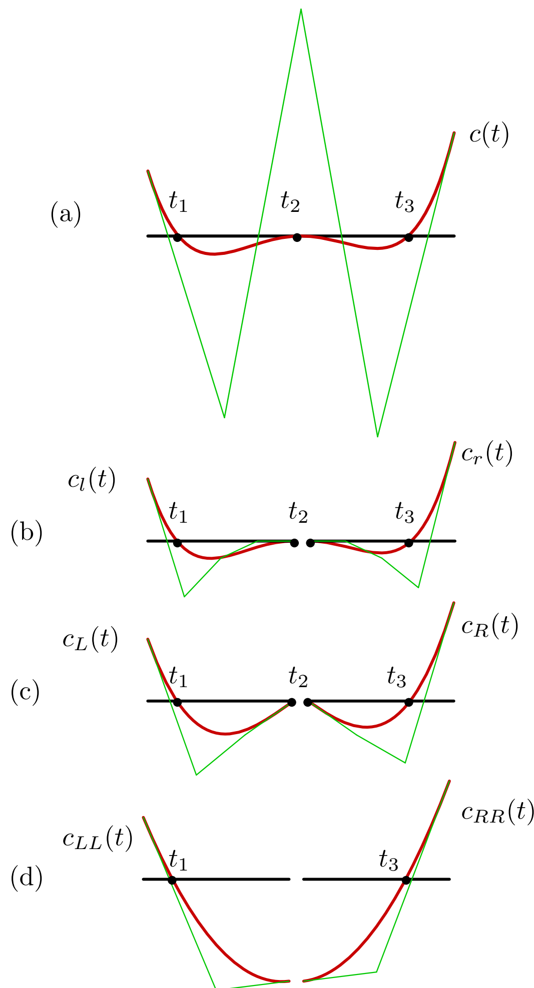


Figure 2: Counting multiplicities of roots: (a) shows the graph of a degree 4 Bézier polynomial $c(t)$ in red with roots at $t_1, t_2$ and $t_3$, with multiplicities 1, 2 and 1, respectively. The control polygon is shown in green and the $t$-axis in black. (b) shows the polynomials $c_l(c)$ and $c_r(t)$ obtained after subdividing $c(t)$ at $t_2$. $c_l(t)$ and $c_r(t)$ are also of degree 4 and contain the root at $t_2$. (c) shows the degree 3 polynomials $c_L(t)$ and $c_R(t)$ obtained after eliminating $(1-t)$ and $t$ from $c_l(t)$ and $c_r(t)$ respectively. Finally, (d) shows the degree 2 polynomials $c_{LL}(t)$ and $c_{RR}(t)$ obtained after eliminating the second root at $t_2$.

### *3.3. Searching for roots outside the domain*

Consider letting the Newton-Raphson search for roots go outside the domain, $[0, 1]$, of interest. If such a root is found, it can still be factored out, thus, reducing the degree of the problem. If a root, $t_0 > 1$, is found, the input polynomial $c(t)$ is subdivided at $t_0$ to obtain a polynomial, $c_l$, corresponding to the domain $[0, t_0]$ and a polynomial, $c_d$ corresponding to the domain $[1, t_0]$. The polynomial $c_d$ is immediately discarded, being outside the domain $[0, 1]$. Clearly, $c_l(1) = c(t_0) = 0$, and as before, the factor $(1-t)$ is removed from $c_l$. However, herein $c_l$ is again subdivided at $\frac{1}{t_0}$ to bring $c_l$ back to the original $[0, 1]$ domain. A similar sequence of steps are followed when the root is found below zero, $t_0 < 0$. The potential benefits of this possible extension are also discussed in Section 4.

## 4. Results

The algorithm described in the previous sections was implemented in the IRIT [7] solid modeling environment. This section first compares the running times of the presented approach, on thousands of polynomials,

with the running times of three other algorithms, also implemented in IRIT, all on a PC with a 3.4 GHz CPU (single thread) and 4 GB (32-bit executable) memory. Further, the presented algorithm is compared with Quadratic clipping [1] and Cubic clipping [18] on the polynomials used in [1, 18]. We also compared the running times of the presented algorithm with those of the roots finding algorithm of Matlab [19]. Finally, we compared our algorithm with that proposed in [15].

A numeric tolerance of $10^{-10}$ ($\epsilon$ in Algorithm 1) and a subdivision tolerance of $10^{-3}$ ($\delta$ in Algorithm 1) were used for all the runs, wherever applicable, unless stated otherwise. The subdivision tolerance is used to terminate the algorithm when the width of the sub-domain becomes small, i.e., when $t^{max} - t^{min} < \delta$. In essence, the separation of adjacent nearby roots is limited by this subdivision tolerance. Hence, roots which are apart by less than $\delta$ are deemed identical. The roots returned from all the different methods were compared against each other to ensure the completeness and correctness of the solutions. Running times for the basic approach presented in Section 2 appear under the column **BZF** (Bézier-Zero-Factored) in Tables 1-5. The running times for the extension given in Section 3.2 for alternative initialization of the Newton-Raphson method, using the first intersection and the one close to the middle of the domain, found between the control polygon of $c(t)$ and the $t$-axis, appear under the columns **BZF-NRI1** and **BZF-NRI2** respectively, in Tables 1-5.

The first method that we compare our approach with, computes the roots of $c(t)$ by computing the points of intersection between two curves in $\mathbb{R}^2$ [29], viz., the graph of $c(t)$ and the $t$-axis. This method inspects the intersection of the double wedge of $c(t)$ with the $t$-axis to discard sub-intervals not containing roots [27]. The running times for this method appear under the column **CCI** (Curve-Curve-Intersection) in Tables 1-4.

The second method that we use for comparison, is a subdivision-based approach which uses cones bounding the tangent field of polynomial $c(t)$ in order to identify the intervals of domain which are guaranteed to contain at most one root [27]. Once such an interval is identified, a root is searched using the Newton-Raphson method. Further, this method inspects if the coefficients of $c(t)$ are either all positive, or all negative, in order to check for absence of roots, in which case, the sub-domain is discarded. This method, in our implementation, is geared towards solutions of multi-variate polynomials and hence has some computational overhead. The running times for this method of numerically computing zeros appear under the column **NCZ** in Tables 1-4.

Finally, we compare our method against an analytic algorithm for finding the roots [24]. This can be used only on polynomials with degree less than 6, though our implementation only supported polynomials with degree less than 5 and uses Euler's approach which first eliminates the coefficient of the cubic term in the input polynomial. The running times for the same appear under the column **Analytic** in Tables 1-4.

In order to ensure the correctness of the time measurements, for each measurement, an aggregate time for $10^4$ runs on each polynomial was noted and divided by $10^4$. We used four sets of polynomials for comparison, as described below:

1. Scalar univariate Bernstein polynomials with degrees between 3 and 100 were created by randomly sampling the coefficients of the polynomials. In this case, the number of real roots was typically quite small compared to the degree of the polynomial. For each degree, 100 different polynomials were created, some of which are shown in Figure 3. The running times and the number of roots reported in Table 1 are the average of these.

2. Scalar univariate Bernstein polynomials with degrees ranging from 3 to 14, with the number of real roots equal to the degree, were created. This was done by generating monomials with randomly generated roots and multiplying these together. The resulting polynomial in the power basis was converted into the Bernstein basis. A few of these polynomials are shown in Figure 4. The comparison of running times for these polynomials for all the six methods appears in Table 2.

3. Polynomials having roots with multiplicity of two, with degrees between 8 and 21, were created in a manner similar to that for degree $n$ polynomials with $n$ roots described above, albeit, by repeating one of the monomials. The running times for the same are tabulated in Table 3. In this case, our algorithm with the extension given in Section 3.1 correctly counted the multiplicities. A few of these polynomials are plotted in Figure 5.

| #CtrlPt | Avg #Root | Analytic | CCI | NCZ | BZF | BZF-NRI1 | BZF-NRI2 |
|---|---|---|---|---|---|---|---|
| 4 | 0.90 | 0.31 | 6.11 | 12.32 | 1.17 | 1.09 | 1.08 |
| 5 | 1.15 | 0.48 | 8.83 | 16.81 | 1.53 | 1.35 | 1.39 |
| 6 | 1.38 | - | 10.94 | 20.45 | 1.90 | 1.71 | 1.78 |
| 7 | 1.50 | - | 10.66 | 23.46 | 2.16 | 1.88 | 1.95 |
| 8 | 1.64 | - | 14.23 | 25.55 | 2.53 | 2.25 | 2.33 |
| 10 | 2.04 | - | 19.77 | 33.58 | 3.49 | 2.85 | 3.03 |
| 15 | 2.16 | - | 26.84 | 39.88 | 4.80 | 4.20 | 4.24 |
| 20 | 2.82 | - | 42.95 | 58.17 | 8.03 | 6.97 | 7.01 |
| 50 | 4.80 | - | 167.54 | 185.63 | 37.21 | 31.24 | 31.31 |
| 100 | 6.74 | - | 1923.35 | 1508.87 | 342.67 | 225.62 | 245.25 |

Table 1: Comparison of running times of the different algorithms on scalar Bernstein polynomials with randomly generated control points. All times are in micro-seconds. See Figure 3 for a few examples of such polynomials.

| #CtrlPt | #Roots | Analytic | CCI | NCZ | BZF | BZF-NRI1 | BZF-NRI2 |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 0.48 | 22.59 | 36.99 | 3.83 | 2.83 | 2.59 |
| 5 | 4 | 0.59 | 43.62 | 78.57 | 4.38 | 4.15 | 3.74 |
| 6 | 5 | - | 43.56 | 92.04 | 6.12 | 4.96 | 4.72 |
| 7 | 6 | - | 53.10 | 114.70 | 6.18 | 6.40 | 6.15 |
| 8 | 7 | - | 72.16 | 136.48 | 9.10 | 7.65 | 7.28 |
| 9 | 8 | - | 93.52 | 157.09 | 8.97 | 8.82 | 8.80 |
| 10 | 9 | - | 88.21 | 201.03 | 12.18 | 11.06 | 10.96 |
| 11 | 10 | - | 113.20 | 195.19 | 13.80 | 11.89 | 12.56 |
| 12 | 11 | - | 107.83 | 281.53 | 13.74 | 14.42 | 16.35 |
| 13 | 12 | - | 202.44 | 382.29 | 15.98 | 16.59 | 16.58 |
| 14 | 13 | - | 158.58 | 306.96 | 22.65 | 18.52 | 19.77 |
| 15 | 14 | - | 166.22 | 361.96 | 24.74 | 22.00 | 24.31 |

Table 2: Comparison of running times of the different algorithms on scalar Bernstein polynomials having the number of real roots equal to the degree of the polynomial. All times are in micro-seconds. A few examples of such polynomials are shown in Figure 4.

4. A Wilkinson's polynomial with degree $n$ is defined as $c(t) = \prod_{i=0}^{n-1}(t - \frac{i}{n-1})$. Wilkinson's polynomials are known for their numerical instability, in particular, the sensitivity of the roots to perturbation of coefficients [30]. The comparison of running times on Wilkinson's polynomials with degrees 13 and 20 are given in Table 4. The polynomials are shown in Figure 6. For polynomials of degree 13 and 20, the mean deviation of the roots returned by our method from the actual roots is $1.8 \times 10^{-15}$ and $1.5 \times 10^{-9}$, and the maximum deviation is $5.5 \times 10^{-15}$ and $7.3 \times 10^{-9}$, respectively, using the double precision for representation of the coefficients.

As can be observed from Tables 1-4, Algorithm **BZF** is faster than the other methods by almost an order-of-magnitude. Further, **BZF-NRI1** and **BZF-NRI2** show computational benefit over simple **BZF**, as seen in Tables 1-5, with some exceptions as can be seen in Table 3 for polynomial with 21 control points. While initializing the Newton-Raphson method with a location near the middle of the domain is expected to more likely succeed compared to an initialization at the end of the domain, it requires the computation of all the intersections between the control polygon of $c(t)$ and the $t$-axis. Hence, in some cases **BZF-NRI1** performs better than **BZF-NRI2**, for instance, in Table 1, while in some cases it is vice versa, for instance, on the polynomials with high degree in Table 3.

It turns out that the out-of-domain root searching extension discussed in Section 3.3, allowing the domain to expand beyond $[0, 1]$, does not show a significant improvement in running times over the simple **BZF**. This is probably because even if a polynomial has a root outside $[0, 1]$, in most cases, that portion of polynomial gets discarded by the routine PurgeProblem (Line 1 of Algorithm 1), as the control polygon is for domain

| #CtrlPt | # Unique Roots | Analytic | CCI | NCZ | BZF | BZF-NRI1 | BZF-NRI2 |
|---|---|---|---|---|---|---|---|
| 9 | 7 | - | 102.90 | 169.35 | 8.16 | 9.87 | 9.02 |
| 10 | 8 | - | 107.82 | 214.43 | 10.89 | 9.84 | 9.66 |
| 11 | 9 | - | 125.18 | 211.18 | 11.34 | 12.27 | 12.54 |
| 12 | 10 | - | 774.45 | 240.14 | 12.62 | 13.03 | 14.02 |
| 13 | 11 | - | 351.35 | 285.84 | 14.61 | 15.06 | 15.27 |
| 14 | 12 | - | 1850.59 | 319.09 | 20.84 | 19.75 | 20.36 |
| 15 | 13 | - | 764.16 | 295.76 | 20.11 | 19.50 | 18.36 |
| 16 | 14 | - | 1394.23 | 408.83 | 22.11 | 23.50 | 22.71 |
| 17 | 15 | - | 1047.41 | 415.89 | 24.34 | 24.79 | 26.46 |
| 18 | 16 | - | 537.87 | 422.28 | 27.79 | 27.20 | 29.35 |
| 19 | 17 | - | 365.31 | 505.46 | 36.66 | 41.34 | 33.74 |
| 20 | 18 | - | 1597.59 | 553.07 | 38.18 | 41.47 | 33.76 |
| 21 | 19 | - | 622.01 | 554.67 | 36.33 | 46.53 | 37.95 |

Table 3: Comparison of running times of the different algorithms on scalar Bernstein polynomials of degree $n$ having $n-1$ distinct roots, with one root with multiplicity of two. All times are in micro-seconds. A few examples of such polynomials appear in Figure 5.

| #CtrlPt | #Roots | Analytic | CCI | NCZ | BZF | BZF-NRI1 | BZF-NRI2 |
|---|---|---|---|---|---|---|---|
| 14 | 13 | - | 187.66 | 233.68 | 14.46 | 15.79 | 15.32 |
| 21 | 20 | - | 347.11 | 487.15 | 40.17 | 32.89 | 37.00 |

Table 4: Comparison of running times of the different algorithms on Wilkinson's polynomials of degree 13 and 20. All times are in micro-seconds. Wilkinson's polynomials of degree 13 and 20 are shown in Figure 6.

| | Polynomial | QuadClip (1.7 GHz CPU) | CubClip (1.7 GHz CPU) | BZF (3.4 GHz CPU) | BZF-NRI1 (3.4 GHz CPU) | BZF-NRI2 (3.4 GHz CPU) |
|---|---|---|---|---|---|---|
| Single root | $f_4(t)$ | 8.10 | 11.20 | 1.52 | 1.14 | 1.30 |
| | $f_8(t)$ | 13.00 | 15.40 | 1.36 | 1.34 | 1.33 |
| | $f_{16}(t)$ | 28.00 | 27.60 | 2.05 | 1.93 | 1.94 |
| Double root | $f_4(t)$ | 15.20 | 15.20 | 0.80 | 2.85 | 2.44 |
| | $f_8(t)$ | 26.60 | 27.10 | 0.82 | 1.18 | 1.17 |
| | $f_{16}(t)$ | 56.10 | 32.70 | 1.22 | 1.93 | 1.95 |
| Triple root | $f_4(t)$ | 200.00 | 45.40 | 3.44 | 3.45 | 3.23 |
| | $f_8(t)$ | 180.00 | 65.30 | 7.94 | 4.73 | 4.47 |
| | $f_{16}(t)$ | 174.00 | 86.10 | 35.14 | 16.24 | 35.46 |
| Near Double root | $f_4(t)$ | 15.10 | 24.10 | 3.03 | 2.44 | 2.27 |
| | $f_8(t)$ | 30.40 | 32.20 | 5.09 | 3.15 | 3.18 |
| | $f_{16}(t)$ | 63.20 | 48.20 | 6.39 | 4.39 | 4.43 |

Table 5: Comparison of running times for **BZF**, **BZF-NRI1**, **BZF-NRI2**, Quadratic clipping [1] and Cubic clipping [18], for an accuracy of $10^{-8}$ for all methods, on the polynomials used in [1, 18]. All times are in micro-seconds.

$[0,1]$ that contains no roots.

We further noted the running times of algorithms **BZF**, **BZF-NRI1** and **BZF-NRI2** on the 12 polynomials used in [1] and [18], with single, double, triple and near double roots, identifying all roots (and ignoring their multiplicities), as is done in [1, 18]. Each of these four classes has three polynomials in it, labeled as $f_4(t), f_8(t)$ and $f_{16}(t)$, with degrees 4, 8 and 16 respectively. We did not run tests on polynomials with near triple roots used in [18] for the lack of precision while converting these polynomials from the power basis to the Bernstein basis, when using the double precision representation. The running times for Quadratic clipping and Cubic clipping are noted from [18] for accuracy of $10^{-8}$, in which, a computer with Intel(R) 1.7 GHz processor and 512 MB RAM was reported. The comparison of the running times, given in

| Randomly generated coeffs. | | | Degree n with n roots | | | Repeated roots | | | Wilkinson's polynomials | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| #CtrlPt | Avg #Root | NR success rates (%) | #CtrlPt | #Roots | NR success rates (%) | #CtrlPt | # Unique Roots | NR success rates (%) | #CtrlPt | #Roots | NR success rates (%) |
| 4 | 0.90 | (90/154) 58% | 4 | 3 | (3/4) 75% | 9 | 7 | (7/10) 70% | 14 | 13 | (13/21) 62% |
| 5 | 1.15 | (116/201) 57% | 5 | 4 | (4/6) 66% | 10 | 8 | (8/12) 66% | 21 | 20 | (20/33) 61% |
| 6 | 1.38 | (139/244) 57% | 6 | 5 | (5/8) 63% | 11 | 9 | (9/11) 81% | | | |
| 7 | 1.50 | (150/270) 55% | 7 | 6 | (6/7) 86% | 12 | 10 | (10/13) 76% | | | |
| 8 | 1.64 | (166/301) 55% | 8 | 7 | (7/10) 70% | 13 | 11 | (11/15) 73% | | | |
| 10 | 2.04 | (204/333) 61% | 9 | 8 | (8/9) 89% | 14 | 12 | (12/15) 80% | | | |
| 15 | 2.16 | (217/360) 60% | 10 | 9 | (9/12) 75% | 15 | 13 | (13/19) 68% | | | |
| 20 | 2.82 | (282/482) 58% | 11 | 10 | (10/13) 77% | 16 | 14 | (14/20) 70% | | | |
| 50 | 4.80 | (480/789) 60% | 12 | 11 | (11/12) 91% | 17 | 15 | (15/19) 78% | | | |
| 100 | 6.74 | (674/1103) 61% | 13 | 12 | (12/12) 100% | 18 | 16 | (16/23) 69% | | | |
| (Averaged over 100 polynomials) | | | 14 | 13 | (13/18) 72% | 19 | 17 | (17/26) 65% | | | |
| | | | 15 | 14 | (14/18) 78% | 20 | 18 | (18/26) 69% | | | |
| | | | | | | 21 | 19 | (19/23) 82% | | | |

Table 6: Success rates of the numeric step of Algorithm 1 for each of the four classes of polynomials.

Table 5, shows that **BZF** is much faster than both Quadratic clipping as well as Cubic clipping, even after accounting for the difference in the processor speed used in [18] and that used in our tests. The CPU used in our tests was twice as fast as that used in [18], while the speed-up achieved by **BZF** is far more than twice, as can be seen from Table 5. Note that, the running times for **BZF** for double root polynomials are smaller than those for single root polynomials. This is explained by the fact that the double root polynomials in this case have the root at 0.5, which is also the seed used for initializing the Newton-Raphson method in **BZF**.

The function *roots* in Matlab [19] computes the complex roots of polynomials given in power basis. This is done by computing the eigen-values of the companion matrix [12] of the input polynomial. This approach does not require numeric tolerance as an input parameter. However, the gained precision of roots thus computed is less compared to our method. The roots returned by this Matlab function for the four classes of polynomials used for our tests, as described previously, had an average precision of $10^{-8}$. Further, since this method works with the power basis, it has issues with numerical stability. For instance, on the Wilkinsons's polynomial of degree 20, the achieved precision was $10^{-4}$, while our method gains a precision of $10^{-10}$. In order to compare our algorithm with the roots function of Matlab, we exported the C code of this function from Matlab and noted the running times of the same. It was observed that on polynomials with sparse real roots, e.g., polynomials obtained by randomly sampling the coefficients of the polynomials, as explained before, our algorithm was over 10 times faster than the Matlab function. On polynomials with dense real roots, e.g., degree-$n$ polynomials with $n$ real roots, our algorithm was about three times faster than the Matlab function.

Finally, we compared our method with that proposed in [15] on polynomials used in [15] and found our method to be two orders-of-magnitude faster. For instance, the time reported in [15] for computing roots of Wilkinson's polynomial of degree 20 is 20 milli-seconds for a precision of $10^{-7}$, on a PC with 2.2 GHz processor and 2 GB RAM, while **BZF** takes only 40.72 micro-seconds for the same, with a precision of $10^{-10}$, as shown in Table 4.

One main factor that contributed to the almost order-of-magnitude improvement in the performance over the state-of-the-art is the success rate in the numeric search step for roots (Line 4 in Algorithm 1). Newton-Raphson is very efficient in converging (quadratically for simple roots [13]) to the roots and our experiments (see Table 6) show that over 50% (and sometimes even over 90%) of the recursive invocations of **BZF** terminated with a successful finding of a new root in the domain in hand. This means that, amortized, Algorithm 1 was recursively invoked less than two calls per root! Further, it was observed that Algorithm 1 almost never terminated due to the condition that $t^{max} - t^{min} < \delta$ (Line 11 in Algorithm 1), except once,

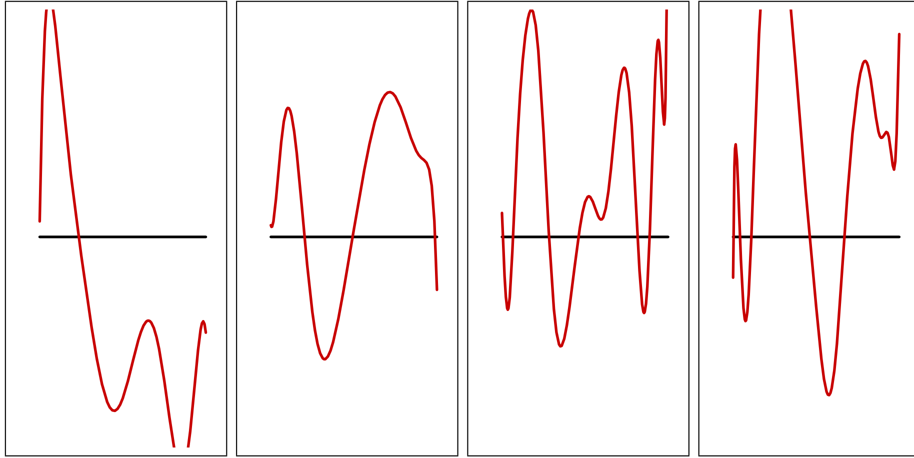

Figure 3: Some examples of scalar Bernstein polynomials with randomly generated coefficients. The domain $[0,1]$ is indicated by a line-segment shown in black and the polynomials are plotted in red. The curves are trimmed at the top and at the bottom for convenient display.

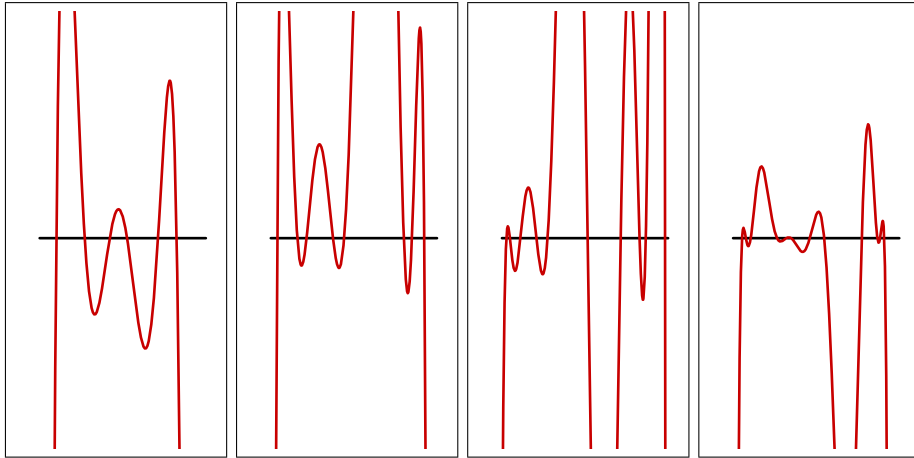

Figure 4: Some examples of scalar Bernstein polynomials with number of roots equal to the degree of the polynomial. The domain $[0,1]$ is indicated by a line-segment shown in black and the polynomials are plotted in red. The curves are trimmed at the top and at the bottom for convenient display.

when a root was present at a distance less than subdivision tolerance from the domain boundary, which is extremely rare. As noted at the beginning of this section, the subdivision tolerance $\delta$ governs the minimal distance between two different ajdacent roots and might affect the computation time only for almost-singular cases, viz., when roots are very close.

## 5. Conclusion

This paper presented a fast algorithm for finding zeros of scalar univariate polynomials given in the Bernstein form. A speed-up in running times compared to the previous state-of-the-art alternative approaches of almost an order-of-magnitude is reported. Unlike the traditional approach, wherein, the polynomial is subdivided until a root is isolated, the proposed algorithm derives its computational advantage from factoring out the roots already computed, hence, reducing the complexity of the sub-problems to recurse upon. Further, the Newton-Raphson method, which is known to be highly efficient, is very effectively exploited, wherever

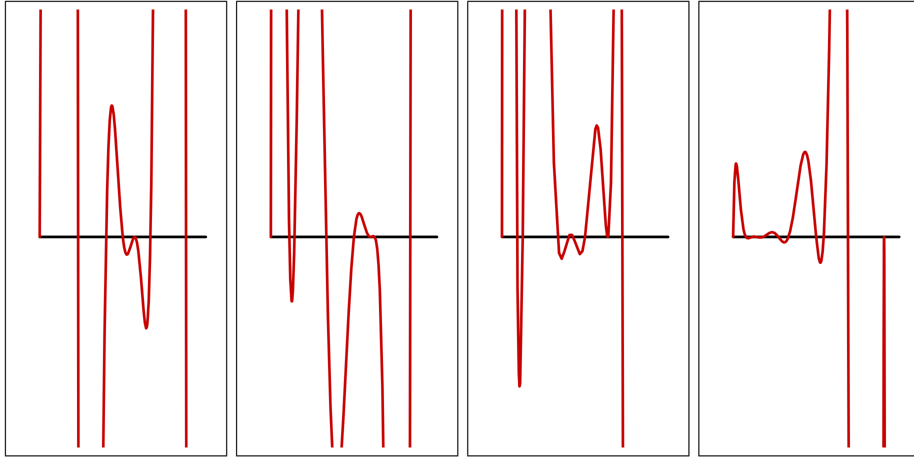

Figure 5: Some examples of scalar Bernstein polynomials having some roots with multiplicity greater than one. The domain $[0, 1]$ is indicated by a line-segment shown in black and the polynomials are plotted in red. The graph of the polynomial is tangent to the $t$-axis at the root with multiplicity greater than one. The curves are trimmed at the top and at the bottom for convenient display.

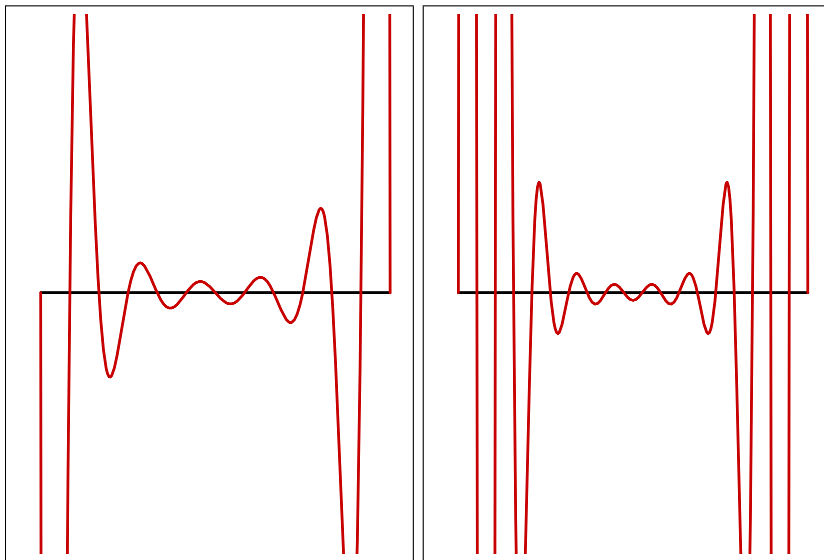

Figure 6: Wilkinson's polynomials with degrees 13 (left) and 20 (right). The domain $[0, 1]$ is indicated by a line-segment shown in black and the polynomials are plotted in red. The curves are trimmed at the top and at the bottom for convenient display.

possible. Finally, a salient feature of this method is the ability to efficiently compute the multiplicities of the roots, and is discussed in Section 3.1.

Like any numeric algorithm, the robustness of the presented approach must be further examined. Specifically, roots with high orders of multiplicities are prone to numerical instabilities. Our test shows that using double precision, the detection of multiplicities of triple roots up to tolerances of $10^{-10}$ can be unstable for degrees larger than 4. This can be partially explained by the fact that each factoring operation scales the coefficients by $O(n)$, the degree of the polynomial. One possible way to alleviate this difficulty would be to inspect the $k$ terminal coefficients in order to detect a multiplicity of order $k$, and if found to be zero within some tolerance, factor them out once by dividing by $t^k$ or $(1 - t)^k$ (see also [3]), as the case may be. Further, one could consider replacing the all-positive or all-negative coefficients test used in our method for identifying sub-domains with no solution, by other, more sophisticated bounds such as quadratic clipping [1].

Finally, an extension of the proposed method to finding zeros of multi-variates would be very useful, for instance, in computing curve-curve or curve-surface intersections.

## 6. Acknowledgment

The authors would like to thank the anonymous reviewers for their invaluable comments. This work was supported in part by the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme FP7/2007-2013/ under REA grant agreement PIAP-GA-2011-286426, and was supported in part by the ISRAEL SCIENCE FOUNDATION (grant No.278/13).

## References

[1] M. Bartoň and B. Jüttler. Computing roots of polynomials by quadratic clipping. *Computer Aided Geometric Design*, 24(3):125–141, 2007.

[2] M. Bartoň, N. Shragai, and G. Elber. Kinematic simulation of planar and spatial mechanisms using a polynomial constraints solver. *Computer-Aided Design and Applications*, 6(1):115–123, 2009.

[3] L. Buse and R. Goldman. Division algorithms for Bernstein polynomials. *Computer Aided Geometric Design*, 25(9):850–865, 2007.

[4] X.-D. Chen, W. Ma, and Y. Ye. A rational cubic clipping method for computing real roots of a polynomial. *Computer Aided Geometric Design*, 38:40–50, 2015.

[5] E. Cohen, R. F. Riesenfeld, and G. Elber. *Geometric Modeling with Splines, An Introduction.* A K Peters, 2001.

[6] A. Eigenwillig, V. Sharma, and C. Yap. Almost tight recursion tree bounds for the Descartes method. *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC*, 2006:71–78, 2006.

[7] G. Elber. Irit modeling environment. `"http://www.cs.technion.ac.il/~irit/"`, January 2015.

[8] R. T. Farouki and V. T. Rajan. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4(3):191–216, 1987.

[9] R. Goldman. *Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling.* Morgan Kaufmann, 2002.

[10] S. Gopalsamy, D. Khandekar, and S. Mudur. A new method of evaluating compact geometric bounds for use in subdivision algorithms. *Computer Aided Geometric Design*, 8(5):337–356, 1991.

[11] T. A. Grandine. Computing zeroes of spline functions. *Computer Aided Geometric Design*, 6(2):129–136, 1989.

[12] K. M. Hoffman and R. Kunze. *Linear Algebra.* Pearson, 1971.

[13] E. Isaacson and H. Keller. *Analysis of Numerical Methods.* Dover Publications, 1966.

[14] Y.-J. Kim, G. Elber, and M.-S. Kim. Precise continuous contact motion for planar freeform geometric curves. *Graphical Models*, 76(5):580–592, 2014.

[15] K. Ko and K. Kim. Improved subdivision scheme for the root computation of univariate polynomial equations. *Applied Mathematics and Computation*, 219(14):7450–7464, 2013.

[16] W. Krandick and K. Mehlhorn. New bounds for the Descartes method. *Journal of Symbolic Computation*, 41(1):49–66, 2006.

[17] J. M. Lane and R. F. Riesenfeld. Bounds on a polynomial. *BIT Numerical Mathematics*, 21(1):112–117, 1981.

[18] L. Liu, L. Zhang, B. Lin, and G. Wang. Fast approach for computing roots of polynomials using cubic clipping. *Computer Aided Geometric Design*, 26(5):547–559, 2009.

[19] MathWorks. Matlab. `"http://www.mathworks.com/products/matlab/"`, January 2015.

[20] J. M. McNamee. A bibliography on roots of polynomials. *Journal of Computational and Applied Mathematics*, 47(3):391–394, 1990.

[21] K. M. Mørken and M. Reimers. An unconditionally convergent method for computing zeros of splines and polynomials. *Mathematics of Computation*, 76(258):845–865, 2007.

[22] B. Mourrain and J.-P. Pavone. Subdivision methods for solving polynomial equations. *Journal of Symbolic Computation*, 44(3):292–306, 2009.

[23] D. Nairn, J. Peters, and D. Lutterkort. Sharp, quantitative bounds on the distance between a polynomial piece and its Bézier control polygon. *Computer Aided Geometric Design*, 16(7):613–631, 2006.

[24] R. W. D. Nickalls. The quartic equation: Invariants and Euler's solution revealed. *The Mathematical Gazette*, 93:66–75, 2009.

[25] F. Rouillier and P. Zimmermann. Efficient isolation of polynomial's real roots. *Journal of Computational and Applied Mathematics*, 162(1):33–50, 2004.

[26] C. Schulz. Bézier clipping is quadratically convergent. *Computer Aided Geometric Design*, 26(1):61–74, 2009.

[27] T. W. Sederberg and R. J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5(2):161–171, 1988.

[28] T. W. Sederberg and T. Nishita. Curve intersection using Bézier clipping. *Computer-Aided Design*, 22(9):538–549, 1990.

[29] T. W. Sederberg and S. R. Parry. Comparison of three curve intersection algorithms. *Computer-Aided Design*, 18(1):58–63, 1986.

[30] J. H. Wilkinson. The evaluation of the zeros of ill-conditioned polynomials. Part I. *Numerische Mathematik*, 1(1):150–166, 1959.

[31] R.-J. Zhang and G.-J. Wang. Sharp bounds on the approximation of a Bézier polynomial by its quasi-control polygon. *Computer Aided Geometric Design*, 23(1):1–16, 2006.