High-Performance Polynomial Root Finding for Graphics

CEM YUKSEL, University of Utah, USA

We present a computationally-efficient and numerically-robust algorithm for finding real roots of polynomials. It begins with determining the intervals where the given polynomial is monotonic. Then, it performs a robust variant of Newton iterations to find the real root within each interval, providing fast and guaranteed convergence and satisfying the given error bound, as permitted by the numerical precision used.

For cubic polynomials, the algorithm is more accurate and faster than both the analytical solution and directly applying Newton iterations. It trivially extends to polynomials with arbitrary degrees, but it is limited to finding the real roots only and has quadratic worst-case complexity in terms of the polynomial's degree.

We show that our method outperforms alternative polynomial solutions we tested up to degree 20. We also present an example rendering application with a known efficient numerical solution and show that our method provides faster, more accurate, and more robust solutions by solving polynomials of degree 10.

CCS Concepts: • **Mathematics of computing → Nonlinear equations**; • **Computing methodologies →** *Ray tracing*; Collision detection;

Additional Key Words and Phrases: Polynomial solver, cubic solver, quartic solver, Newton iterations.

ACM Reference Format:
Cem Yuksel. 2022. High-Performance Polynomial Root Finding for Graphics. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3, Article 7 (July 2022), 15 pages. https://doi.org/10.1145/3543865

1 INTRODUCTION

Polynomials are virtually everywhere in computer graphics. They are not only used for representing various curves and surfaces, but also needed for solving numerous problems that involve root finding, such as ray intersection tests [Aydinlilar and Zanni 2021; Dokter et al. 2019; Steinberger and Grabner 2010] and continuous collision detection [Tang et al. 2014], just to name a few. Unfortunately, only quadratic (second order) polynomials have an efficient analytical solution. For polynomials of higher degrees, either there is no analytical solution or the analytical solution is considered too expensive and inaccurate. The common numerical methods, on the other hand, are often unreliable. This severely limits our ability to efficiently handle various problems involving polynomials and often forces researchers to develop alternative approaches, resulting in substantial amount of additional effort, some of which may lead to sub-optimal solutions.

In this paper, we show that a robust and computationally-efficient numerical solution exists for polynomials of higher degrees than 2, by presenting a simple and effective method for finding the real roots of polynomial equations. Our solution begins with splitting the given polynomial into a finite number of monotonic pieces. For each piece, we perform a variant of Newton iterations to quickly and robustly find the root, if any, with the desired level of accuracy.

We show that our method with cubic (third order) polynomials outperforms both the analytical solution and directly applying Newton iterations in terms of speed, accuracy, and robustness. For higher-order polynomials, the robustness and efficiency of our method persists.

We provide an extensive evaluation with randomly-generated polynomials. To demonstrate the effectiveness of our method, we also present a challenging example rendering problem with an existing highly-efficient numerical solution. Our results show that we can outperform the existing solution by constructing and then solving polynomials of degree 10.

Author's address: Cem Yuksel, cem@cemyuksel.com, University of Utah, USA.

2020. 2577-6193/2022/7-ART7
https://doi.org/10.1145/3543865

1.1 Related Work

Notwithstanding the amount of work on numerical approaches for polynomial root finding in other fields [McNamee 2007; McNamee and Pan 2013], the consensus in computer graphics appears to be that solving polynomials is not practical (e.g. [Reshetov and Luebke 2018]). Indeed, while recent methods can handle extremely high-order (1000+) polynomials and even deliver complex roots with extreme precision (500+ bits) [Pan 2002; Sagraloff and Mehlhorn 2016], their performance is inadequate for relatively low-order polynomials (∼10 or less) that are of interest in graphics.

For cubic polynomials, there exists an analytical formula [Cardano 1570; Press et al. 1992], but it involves computationally-expensive cubic root and trigonometric functions. Also, implementing it in a way to minimize precision loss due to truncation can be a challenge. Strobach [2011] proposes numerical iterations on top of this analytical solution to improve the accuracy at the cost of additional computation. Blinn [2006a,b,c, 2007a,b] offers an alternative analytical solution with similar operations but using homogeneous form. This results in improved numerical precision but reduced performance.

Using Newton iterations might be considered an attractive alternative. Indeed, Newton iterations can deliver fast and accurate solutions for any nonlinear equation, but they are notoriously unreliable. Combining them with bisection can ensure convergence [Press et al. 1992], but only if one can determine the finite interval of each root. This is exactly the solution we describe in this paper, which can be considered a form of *real root isolation* [Collins and Loos 1976; Mourrain et al. 2005]. Our splitting process, however, is not just for isolating the roots, but it also forms monotonic pieces for efficient root finding that follows splitting.

An alternative approach would be picking a good starting point for Newton iterations. Neumark [1965] provides a table-based solution for a low-precision approximation of the roots that can then be refined via Newton iterations, but its stability is not guaranteed. Flocke [2015] presents an approach for evaluating starting positions for Newton iterations that would guarantee convergence with relatively few steps. However, it begins with *rescaling* the given polynomial, which is an expensive process itself (involving a cubic root operation for cubic polynomials).

Quartic (fourth order) polynomials have also received special attention [Christianson 1991; Shmakov 2011; Strobach 2010; Yacoub and Fraidenraich 2012]. In fact, Flocke [2015] also describes a solution for quartic polynomials, which begins with solving the roots of their derivatives. This is similar to our method, but aims at finding a good starting point for Newton iterations, as opposed to our approach of simply isolating the roots without rescaling the polynomial.

For higher-order polynomials, the *RPOLY* method of Jenkins and Traub [1970] can numerically compute all roots, including complex ones. Another popular alternative involves solving the eigenvalues of the polynomial's companion matrix using QR decomposition. These methods are relatively complicated to implement and do not provide a desirable performance for polynomials of relatively low degrees (i.e. ≤ 10).

1.2 Contributions

The main contribution of this work is to show that polynomials of relatively low degrees (i.e. ≤ 10) can be solved efficiently and that they can be safely used in high-performance graphics applications. Our solution of splitting the given polynomial into monotonic pieces is simple and straightforward, though we show that is it highly effective in practice.

Specifically for cubic polynomials, we show that regular Newton iterations are highly unreliable. Our method is not only significantly faster on average but also provides a robust solution. Just like regular Newton iterations, for high-performance graphics applications that can tolerate numerical errors, our method can be used with a fixed number of iterations. This is particularly important for high-performance GPU implementations. Also, our method uses iterations only when a valid root is detected and quickly returns without any numerical iteration when there is no root.

For higher-order polynomials, we show that a high-performance solution exists and that it can outperform prior methods used in graphics.

2 QUADRATIC POLYNOMIALS

For completeness, we begin with quadratic polynomials. The roots x_1 and x_2 of a quadratic polynomial $f(x) = ax^2 + bx + c = 0$ can be efficiently computed using the commonly-known formula

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a} \qquad \text{where} \qquad \Delta = b^2 - 4ac\,. \tag{1}$$

However, $-b \pm \sqrt{\Delta}$ can suffer from excessive truncation error for large b that is close to $\sqrt{\Delta}$, so the following variant is recommended for improved numerical stability [Press et al. 1992]

$$x_1 = -\frac{2c}{b + (\operatorname{sgn} b)\sqrt{\Delta}} \qquad \text{and} \qquad x_2 = -\frac{b + (\operatorname{sgn} b)\sqrt{\Delta}}{2a}\,. \tag{2}$$

Note that this variant works even when $a = 0$, producing $x_1 = -c/b$ and $x_2 = \pm\infty$. Blinn [2005] suggests a slightly different version in homogeneous form that avoids division by zero.

3 CUBIC POLYNOMIALS

Our numerical solution for cubics outperforms the analytical solutions on average with similar accuracy. It also allows achieving up to the full precision of the floating point representation or using limited precision for faster computation. Notably, we can quickly determine the absence of a root within a given interval and entirely skip numerical root finding. In this section, we consider roots within a given finite target interval $x \in [x_{\text{start}}, x_{\text{end}}]$. We discuss how to handle infinite intervals later in Section 5.

3.1 Splitting Cubic Polynomials

Given a cubic polynomial $f(x)$, our first step is to split it into monotonic pieces within the given target interval $x \in [x_{\text{start}}, x_{\text{end}}]$. A cubic polynomial can have up to 2 critical points, where its derivative is zero. In between and beyond these critical points, it is monotonic. Thus, for splitting f into monotonic pieces, we simply need to find the zero-crossings of its derivative $f'(x)$. This forms a quadratic equation, which has an efficient analytical solution. Once we find the roots of f', x_1 and x_2, within $[x_{\text{start}}, x_{\text{end}}]$, if any, we can split f up to 3 pieces with intervals $[x_{\text{start}}, x_1)$, $[x_1, x_2)$ and $[x_2, x_{\text{end}}]$.

Because each piece of f is monotonic, there cannot be more than one root within its interval. The existence of a root is also easy to check. For example, consider the interval $[x_1, x_2)$. For this monotonic piece to include a root, one and only one of $f(x_1)$ and $f(x_2)$ must be negative. If both of them are positive or both of them are negative, f cannot pass through zero within $[x_1, x_2)$. Note that this definition does not include special cases like $f(x_1) = 0$ when $f(x_2) > 0$.

3.2 Fast Numerical Root Finding for Cubic Polynomials

Once we identify the existence of a root by testing its end points, the remaining task is to efficiently find the root. We build our numerical root finding method on Newton iterations. Starting with a guess $x_r^{(0)}$, at each Newton iteration j the next guess is computed using

$$x_n^{(j+1)} = x_r^{(j)} - \frac{f(x_r^{(j)})}{f'(x_r^{(j)})} \; . \tag{3}$$

However, we cannot simply use $x_n^{(j+1)}$ as the next guess $x_r^{(j+1)}$, since Newton iterations are unstable, particularly when $|f'(x_r^{(j)})|$ is small. For cubics, this happens near the critical points x_1 and x_2. Thus, the resulting guess $x_n^{(j+1)}$ might be arbitrarily far from the root, possibly even more so than the previous guess $x_r^{(j)}$.

Yet, since our root finding operates within a given interval $[x_{\min}, x_{\max}]$, we can simply contain the next guess within this interval, using

$$x_r^{(j+1)} = \max\left(x_{\min}, \min\left(x_{\max}, x_n^{(j+1)}\right)\right) \; . \tag{4}$$

This simple solution ensures convergence with cubics (even when $x_{\min} = -\infty$ or $x_{\max} = \infty$), except for a special case when

$$\frac{f(x_r^{(j+1)})}{f'(x_r^{(j+1)})} = -\frac{f(x_r^{(j)})}{f'(x_r^{(j)})} \; , \tag{5}$$

which produces a next guess $x_r^{(j+2)} = x_r^{(j)}$, resulting in an infinite loop.

Fortunately, this special case can be easily avoided. It can only happen for the interval within the two critical points $[x_1, x_2]$ and only when two consecutive guesses are on either side of the inflection point x_c, where the second derivative changes sign, which is exactly at the center of the two critical points $x_c = (x_1 + x_2)/2$. To avoid it, we simply split the interval between the two critical points into two intervals $[x_1, x_c]$ and $[x_c, x_2]$. Only one of these intervals can contain a root (determined by evaluating $f(x_c)$) and the other one can be safely discarded.

In practice, however, infinite loops are possible (though rare) in other cases as well because of inaccuracies due to numerical truncation. Instead of trying to detect and prevent all such cases, which would unavoidably increase the cost of each iteration, we opt for using a fixed number of iterations without any convergence guarantee. In the rare cases when this unsafe loop fails to converge, we move to the numerical root finding solution with guaranteed convergence described below in Section 3.3. For performance-critical applications, however, it might be preferable to ignore such rare special cases and simply terminate when the iteration limit is reached.

We continue Newton iterations until the step size $|x_n^{(j+1)} - x_n^{(j)}|$ is below the given error threshold ϵ. This is a safe termination condition for cubic polynomials within the intervals we iterate.

Note that the given error threshold ϵ bounds the maximum error. The expected error can be significantly smaller than ϵ. This is because Newton iterations typically have quadratic convergence, meaning each iteration can find about half of the remaining bits. In our experiments, we have observed average error values that are 2 to 3 orders of magnitude smaller than ϵ with single (32-bit) precision and 3 to 8 orders of magnitude smaller than ϵ with double (64-bit) precision.

3.3 Numerical Root Finding with Guaranteed Convergence

The numerical root finding approach we adopt for guaranteed convergence is a hybrid solution that combines Newton iterations and bisection [Press et al. 1992]. This is achieved using two strategies, as explained below.

First, we iteratively shorten the root finding interval. Starting with the initial interval $[x_{\min}^{(0)}, x_{\max}^{(0)}]$, at each iteration we update it by splitting the interval at the current guess $x_r^{(j)}$. By using the sign of $f(x_r^{(j)})$, we can tell which side of this guess contains the root. Thus, the updated interval can be written as

$$\begin{aligned} x_{\min}^{(j+1)} &= x_r^{(j)}, \quad \text{if } \operatorname{sgn} f(x_{\min}^{(j)}) = \operatorname{sgn} f(x_r^{(j)}) \\ x_{\max}^{(j+1)} &= x_r^{(j)}, \quad \text{if } \operatorname{sgn} f(x_{\max}^{(j)}) = \operatorname{sgn} f(x_r^{(j)}) \; . \end{aligned} \tag{6}$$

Since we already compute $f(x_r^{(j)})$ for each Newton iteration, shortening the interval requires minimal additional computation.

Second, we fall back on bisection when we detect that Newton iteration fails to produce a next guess inside the interval:

$$x_r^{(j+1)} = \begin{cases} x_n^{(j+1)}, & \text{if } x_{\min}^{(j+1)} < x_n^{(j+1)} < x_{\max}^{(j+1)} \\ \left(x_{\min}^{(j+1)} + x_{\max}^{(j+1)}\right)/2, & \text{otherwise} \; . \end{cases} \tag{7}$$

This occasional bisection step reduces the interval size by half. Thus, we guarantee shortening the interval at each iteration.

Note that Newton iterations are more likely to fail (i.e. produce a next guess outside the current interval) when the derivative f' approaches zero and thereby the Newton step size approaches infinity. In our case, we know that f' is zero at the two ends of our initial interval, since they are critical points. Therefore, to reduce the probability of a failed Newton step (and thereby improve convergence), we begin with an initial guess $x_r^{(0)}$ at the center of the initial interval, which places it away from the two known critical points.

This process can be used with $\epsilon = 0$ to find the root up to the numerical precision of the floating-point representation by simply checking if the Newton or bisection step moves the guess at all (i.e. when the difference between guesses falls below numerical precision).

3.4 Optimizing Cubic Root Finding via Deflation

The procedure explained above can be used for finding all (up to 3) roots of a cubic polynomial. Yet, when there is more than one root within the target interval, deflation is a more efficient strategy. After the first root x_R is found and the existence of at least one other root within the target interval is verified (by comparing the values of the polynomial at the end points of the intervals), we can deflate the cubic polynomial using the root x_R that we have already found, such that

$$f(x) = ax^3 + bx^2 + cx + d = (x - x_R)(a'x^2 + b'x + c') \; , \tag{8}$$

where the deflated quadratic polynomial has coefficients

$$a' = a \qquad\qquad b' = b + a'x_R \qquad\qquad c' = c + b'x_R = d \; . \tag{9}$$

After deflation, we can simply solve the resulting quadratic polynomial. This is considerably faster than using numerical root finding for the remaining roots. On the other hand, it may not satisfy the given error bound ϵ (with small ϵ) due to any numerical truncation error during quadratic root finding. This can be fixed by *polishing* the resulting roots of the deflated quadratic polynomial via additional Newton iterations. Alternatively, this deflation optimization can be skipped when the application requires a tight error bound. We use the latter strategy in our tests.

4 HIGHER-ORDER POLYNOMIALS

We solve higher-order polynomials recursively. Remember that our solution for a cubic polynomial begins with finding its critical points by solving its quadratic derivative. Similarly, for solving a quartic (fourth order) polynomial, we begin with finding its critical points by solving its cubic derivative. Then, we similarly split the given interval into 4 pieces.

Thus, our solution for a polynomial of degree d begins with solving a polynomial of degree $d-1$ to split the given interval up to d monotonic pieces. Then, within each piece that contains a root, we perform numerical root finding. This results in an algorithm with quadratic worst-case complexity in the polynomial degree d.

Our results show that this is an effective approach for polynomials with relatively small degrees that are of interest in graphics. With increasing d, however, one must not only consider the additional computation cost but also the fact that simply evaluating high-order polynomials is prone to numerical truncation errors. Nonetheless, as long as the floating-point representation provides the desired accuracy, our numerical root finding for high-order polynomials can find the roots with the given error bound ϵ.

The typical termination condition we use for cubics, which compares the Newton step size to a given tolerance ϵ, is not guaranteed to achieve the desired accuracy with higher-order polynomials. This is because the error is bounded by the size of the interval $x_{\max}^{(j)} - x_{\min}^{(j)}$, which can be arbitrarily larger than ϵ, even when the last Newton step size is smaller than or equal to ϵ.

One problem with Newton iterations for bounding the error is that the guesses $x_n^{(j)}$ often remain on one side of the root. Therefore, our iterations can bring one end of the interval close to the root, while keeping the other end unchanged. The bisection step can help, but it is used only when the Newton step fails to provide a valid guess within the interval. Often times, the bisection step is used only after Newton iterations converge to the exact root with the numerical precision of the floating-point representation, which may require too many steps.

Our solution to quickly bound the error by ϵ is to produce a guess that is likely to appear on the other side of the root. We accomplish this by replacing Equation 7 with

$$x_r^{(j+1)} = x_n^{(j+1)} + \begin{cases} \epsilon, & \text{if } \operatorname{sgn} f(x_{\min}^{(j)}) = \operatorname{sgn} f(x_r^{(j)}) \\ -\epsilon, & \text{otherwise} \end{cases} \tag{10}$$

only when $|x_n^{(j+1)} - x_n^{(j)}| \le \epsilon$. If the resulting guess $x_r^{(j+1)}$ is on the other side of the root, we can safely return $x_n^{(j+1)}$ as our final guess, knowing that it is at most ϵ away from the exact root. Otherwise, we continue using $x_r^{(j+1)}$ as the next guess, which is even closer to the root than $x_n^{(j+1)}$. When $\epsilon = 0$ (or when ϵ is smaller than numerical precision limit), we simply set $x_r^{(j+1)}$ as the next representable floating point number.

In our experiments, however, we have not observed a case when this additional test is required to bound the error. Therefore, in our tests we use it only when $\epsilon = 0$. Otherwise, we simply terminate the iterations when the Newton step size is smaller than or equal to ϵ.

Though using the deflation strategy accelerates root finding for cubic polynomials, it leads to an algorithm with exponential complexity (in terms of d) with higher-order polynomials. In theory, deflation can still reduce the number of numerical root finding operations for quartic polynomials when they have more than three roots. In practice, however, the overhead of checking for such cases may not always pay off. Therefore, in our tests with quartic and higher-order polynomials, we do not perform deflation (except when solving their cubic derivatives).

5 INFINITE INTERVALS

Finite intervals are far more common in graphics. Even for problems like ray intersections, where the ray parameters can go to infinity, the resulting polynomial equation can still be bounded, using a different parameter. Nonetheless, our solution can be extended to infinite intervals as well.

One simple solution for handling infinite intervals is *rescaling* the polynomial, which ensures that all coefficients are in $[-1, 1]$ and all roots are in $[-2, 2]$ (see Flocke [2015] for a detailed explanation).

This entirely circumvents the problem of working with infinite intervals. On the other hand, the computation cost of rescaling is substantial (particularly for higher-order polynomials) and often leads to an inefficient solution. Therefore, we favor directly working with infinite intervals.

With an infinite target interval, our splitting process forms (up to) two infinite pieces: one before the first critical point $(-\infty, x_1)$ and one after the last critical point $[x_{d-1}, \infty)$. We can easily tell if there is a root within these intervals by checking the sign of the highest-order coefficient a_d:

$$\operatorname{sgn} f(\pm\infty) = \begin{cases} \operatorname{sgn} a_d, & \text{if } d \text{ is even} \\ \pm\operatorname{sgn} a_d, & \text{if } d \text{ is odd}\ . \end{cases} \tag{11}$$

Note that care must be taken when $a_d = 0$, in which case we can use the remaining terms and treat it as a polynomial of degree $d - 1$.

Let $[x_{\min}, \infty)$ be an infinite interval that contains a root. To find the root, we perform numerical root finding starting with an initial guess of $x_r^{(0)} = x_{\min} + \delta$, where δ is an arbitrary offset. Here, δ is used for moving the initial guess sufficiently far from the critical point $x_{\min}$, where the derivative f' is zero and would cause numerical problems with Newton iterations.

We cannot perform bisection within an infinite interval, so we replace the bisection case in Equation 7 with $x_r^{(j+1)} = x_{\min}^{(j+1)} + \delta$, which is used only when $x_r^{(j+1)} = x_{\min}^{(j+1)}$ and $x_{\max}^{(j+1)} = \infty$.

Thus, when using infinite intervals, the only change to the entire algorithm is replacing bisection operations with a shift by an arbitrary amount δ, as long as one end of the current interval remains ∞ or $-\infty$. When the next guess falls on the other side of the root, the interval becomes finite and we can continue with bisection when needed, as explained above.

If f has no critical points, we end up with a single infinite interval $(-\infty, \infty)$. In that case, we can start with an arbitrary initial guess $x_r^{(0)}$. After the first iteration, one side of the interval becomes finite (using Equation 6), and we can continue as explained above. Note that f must always have a critical point when the polynomial's degree d is even, so single infinite intervals $(-\infty, \infty)$ can only happen when d is odd, in which case there is always a real root.

6 EVALUATION

We evaluate our polynomial root finding solution using both randomly-generated polynomials and an example rendering application using polynomials up to degree 10. All performance results are measured on an Intel Xeon CPU E5-2643 v3 at 3.4 GHz using single-threaded computation. All polynomials are pre-generated and all compared methods process exactly the same data. The source code of our polynomial root finding method is available online [Yuksel 2022].

6.1 Cubic Polynomials

Cubic polynomials are commonplace in computer graphics. For example, roots of cubic equations are needed for continuous collision detection [Tang et al. 2012] and some curve formulations [Yan et al. 2017; Yuksel 2020]. That is why we begin our evaluation with cubic polynomials.

We compare the performance of our method to the standard *analytical* solution [Press et al. 1992], *Blinn*'s analytical solution [Blinn 2007b], *Flocke*'s numerical solution [Flocke 2015], and regular Newton iterations up to 40 steps. We use an error threshold ϵ that produces a similar average error with our method as compared to Blinn's analytical solution. Note that ϵ bounds the maximum error and the expected error is typically orders of magnitude smaller. The standard analytical solution, however, produces higher error, so we also include results with a lower-precision version of our method, "*Ours (low)*," with larger ϵ to match the average error of the standard analytical solution. In addition, we include a variant of our method, "*Ours (no deflation)*," which does not use deflation for computing the second and third roots, demonstrating the significance of the deflation optimization.

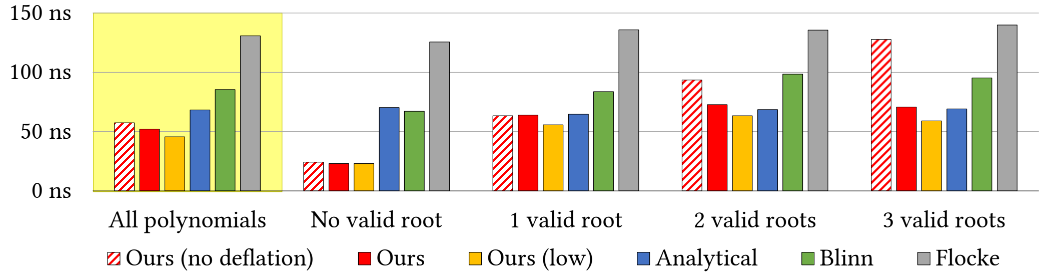


Fig. 1. ***Cubic polynomial single precision (32-bits) performance tests for finding all roots*** *within* $[0,1]$ *for randomly-generated Bernstein polynomials with coefficients in* $[-1,1]$*. The "all polynomials" group includes 27% with no root, 48% with a single root, 23% with two roots, and 2% with three roots. The other groups show the computation times for polynomials with different numbers of roots within* $[0,1]$*. All numerical methods use* $\epsilon = 3.5\times10^{-4}$*, except for "Ours (low)" that uses* $\epsilon = 10^{-2}$*, producing an average error of* 5×10^{-8} *with "Ours" and* 5×10^{-5} *with "Ours (low)."*

For ours tests, we use Bernstein polynomials with randomly-generated coefficients within $[-1,1]$. The resulting expanded polynomials (needed for efficient computation), however, can have coefficients with many orders of magnitude difference, particularly with higher-order polynomials, providing a wide range of numerical values.

Our test results for finding all roots within the target interval $[0,1]$ using single-precision (32-bit) floating-point numbers are summarized in Figure 1. Comparing all randomly-generated polynomials (the first group), we can see that our method with either threshold produces the best performance. This is mainly because we can quickly identify the cases with no valid root (the second group), which is 27% of all polynomials in these tests. Still, our method maintains a slight advantage over the analytical alternatives even when there is a root. Notice that "Ours (low)" has slightly lower computation times than "Analytical" and "Ours" has a more prominent saving as compared to the analytical solution of "Blinn."

Notice that deflation can provide a sizable performance boost when there are 3 valid roots, since, after computing the fist one, the other two can be found by solving a quadratic instead.

The numerical solution of "Flocke" can converge with fewer Newton steps than ours, but this does not make up for the overhead of rescaling the polynomial and the cost of operating on the deflated polynomial, resulting in clearly lower performance in all cases. Note that Flocke's method also uses deflation.

Regular Newton iterations, on the other hand, cannot be used for finding all roots. Nonetheless, for many applications, simply finding the first root can be sufficient. Therefore, we provide test results for finding the first root in Figure 2.

One important difficulty with regular Newton iterations is determining the absence of a root. Our implementation begins iterations at the start of the target interval and clamps the guesses to the target interval for stability. The absence of a root is detected only when Newton iterations do not converge or they are stuck at one end of the interval due to clamping. As a result, regular Newton iterations have the worst performance when there is no root (second group in Figure 2).

When there is a root, however, regular Newton iterations do not incur the overhead of our method and perform faster. Yet, the results are unreliable. In these tests, for about one third of the polynomials that contain one valid root, regular Newton iterations failed to find it. For polynomials with two valid roots, the failure rate of regular Newton iterations is 11%. Therefore, skipping our

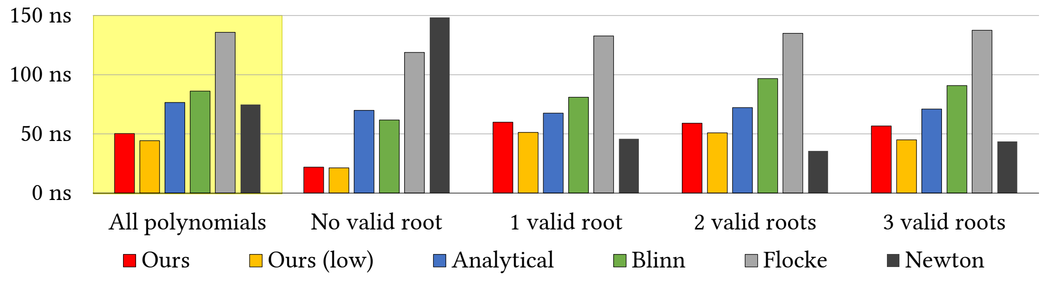


Fig. 2. ***Cubic polynomial single precision (32-bits) performance for finding the first root*** *using the same test setup in Figure 1. Note that regular "Newton" iterations fail to find an existing root for 18% of all polynomials (33% of polynomials with one root and 11% of them with two roots).*

overhead and directly using Newton iterations leads to highly unreliable results, even when it provides a performance advantage.

In addition, when regular Newton iterations converge, the resulting root is not always the one closest to the starting point and there is no simple way to check or find the other root, if it exists.

In these tests for finding the first root, our method provides a more prominent performance advantage over the analytical solutions, Blinn's variant, and Flocke's method. This is because our root finding is performed at most once and the other methods cannot skip their overheads.

In our supplemental document, we provide a more extensive evaluation of cubic polynomials, including special cases when Newton iterations converge linearly (as opposed to quadratically). We also include variants of our method that replace our numerical root finding with pure *bisection*, *Ridder's method*, and *regula falsi* (i.e. the false position method), showing that they uniformly provide inferior performance, except for one trivial special case of three repeated roots (i.e. polynomials of the form $ax^3 + d = 0$) when Newton iterations converge slower than bisection.

A notable failure case for all methods we tested, including ours, is polynomials with two repeated (i.e. duplicated) roots. This forms curves that are tangent to the $y = 0$ line. All numerical and analytical methods we tested can fail to detect such roots (see the supplemental document for more details). In practice, duplicated roots appear close to cases with no solution, for example when a ray is tangent to the surface or a vertex barely grazes the surface of a triangle.

6.2 Higher-Order Polynomials

For evaluating higher-order polynomials, we compare our method to RPOLY [Jenkins and Traub 1970], a popular polynomial root-finder that is optimized for polynomials with real coefficients and also appears in a relatively recent graphics article [Chlumský et al. 2018]. The implementation we use for RPOLY is based on the open-source Simbody multibody physics API [Sherman et al. 2011] with additional optimizations that eliminate memory allocation inside the solver.

Again, we use randomly-generated Bernstein polynomials with coefficients in $[-1, 1]$. The results are summarized in Table 1.

In these tests, we include three different error thresholds for our method: "Ours (low)" uses $\epsilon = 5 \times 10^{-4}$, "Ours (high)" uses $\epsilon = 10^{-8}$, and "Ours (full)" uses $\epsilon = 0$, which computes the roots to the full floating-point precision. To minimize the error, "Ours (full)" does not use deflation while computing the cubic derivatives, thereby incurs a considerable performance hit. For both "Ours (low)" and "Ours (high)," the average error values we get in these tests are lower than RPOLY, while "Ours (full)" converges to the root values up to double (64-bit) precision.

Table 1. *Bounded root finding times for double (64-bit) precision higher-order polynomials.*

d	no root	1 root	2+ roots	**Ours (low)**		**Ours (high)**		**Ours (full)**		RPOLY	
3	27%	48%	24%	0.08 μs	1.0×	0.09 μs	1.2×	0.17 μs	2.1×	0.57 μs	7.1×
4	22%	45%	33%	0.17 μs	1.0×	0.21 μs	1.2×	0.56 μs	3.3×	1.35 μs	7.9×
5	18%	42%	41%	0.29 μs	1.0×	0.36 μs	1.3×	1.03 μs	3.6×	2.42 μs	8.4×
6	15%	38%	47%	0.45 μs	1.0×	0.58 μs	1.3×	1.63 μs	3.6×	3.40 μs	7.5×
7	12%	35%	53%	0.64 μs	1.0×	0.83 μs	1.3×	2.33 μs	3.7×	4.28 μs	6.7×
8	10%	32%	58%	0.86 μs	1.0×	1.14 μs	1.3×	3.16 μs	3.7×	5.20 μs	6.0×
9	9%	29%	62%	1.13 μs	1.0×	1.52 μs	1.3×	4.19 μs	3.7×	6.22 μs	5.5×
10	8%	27%	66%	1.45 μs	1.0×	1.97 μs	1.4×	5.45 μs	3.8×	7.36 μs	5.1×
20	2%	12%	86%	9.04 μs	1.0×	12.19 μs	1.3×	43.13 μs	4.8×	22.36 μs	2.5×
30	1%	6%	93%	24.61 μs	1.0×	43.73 μs	1.8×	174.05 μs	7.1×	47.44 μs	1.9×

As can be seen in Table 1, "Ours (low)" achieves significantly faster computation than RPOLY, starting with 8.2× for cubic polynomials and down to 1.9× for degree 30 polynomials. "Ours (high)" also maintains a faster performance with orders of magnitude improved accuracy. "Ours (full)" converges to the exact root with faster computation than RPOLY up to degree 10.

The performance advantage of our method over RPOLY should not be simply attributed to the fact that our method can quickly skip roots that are outside of the target interval, though it certainly helps. As can be seen in Table 1, a majority of the polynomials have a valid root and more of them have 2 or more roots with increasing polynomial degree d in these tests.

RPOLY, however, computes all roots, including complex ones. Therefore, it is entirely reasonable that it requires more computation time. On the other hand, these complex roots and the roots outside of the target interval have no use in practical applications. In fact, this incurs the additional cost of identifying the valid real roots, some of which may be misclassified as complex numbers with small (but nonzero) imaginary components.

Also notice that the relative computation time of "Ours (full)" is significantly higher for degree 30 polynomials. This hints that 64-bit floating-point representation we use in these tests may become insufficient for our method as the degree of the polynomial increases. Indeed, even if our method could deliver a desirable performance for even higher degree polynomials, it would not be advisable without using a higher-precision floating-point representation than 64 bits.

6.3 Infinite Intervals

We evaluate the performance of our method with infinite intervals by comparing it to RPOLY and "Ours (high rescaled)" that applies polynomial rescaling, which ensures that all real roots are within $[-2, 2]$, prior to using our method for root finding within this finite interval. The results are presented in Table 2.

Comparing "Ours (high)" and "Ours (high rescaled)," we can see that our method for handling infinite interval provides superior performance to bounding the intervals by first rescaling the polynomial (which also scales ϵ). In fact, our approach is highly-efficient for handling infinite intervals. The performance drop of this unbounded case in Table 2 as compared to the bounded case in Table 1 is mainly due to the fact that more roots must be computed for the unbounded case.

Our method maintains a significant performance advantage as compared to RPOLY in the unbounded case as well. Beyond degree 10, however, the performance difference diminishes and RPOLY catches up to the performance of "Ours (low)" at degree 20.

Table 2. *Unbounded root finding times for double (64-bit) precision higher-order polynomials.*

d	**Ours (low)**		**Ours (high)**		**Ours (high rescaled)**		**Ours (full)**		RPOLY	
3	0.09 μs	1.0×	0.11 μs	1.2×	0.11 μs	1.2×	0.27 μs	2.9×	0.50 μs	5.4×
4	0.26 μs	1.0×	0.32 μs	1.3×	0.48 μs	1.8×	1.09 μs	4.2×	1.26 μs	4.9×
5	0.52 μs	1.0×	0.65 μs	1.3×	0.96 μs	1.9×	2.05 μs	3.9×	2.40 μs	4.6×
6	0.89 μs	1.0×	1.12 μs	1.3×	1.66 μs	1.9×	3.21 μs	3.6×	3.37 μs	3.8×
7	1.36 μs	1.0×	1.70 μs	1.2×	2.59 μs	1.9×	4.60 μs	3.4×	4.27 μs	3.1×
8	1.91 μs	1.0×	2.39 μs	1.2×	3.76 μs	2.0×	6.18 μs	3.2×	5.15 μs	2.7×
9	2.61 μs	1.0×	3.25 μs	1.2×	5.18 μs	2.0×	8.13 μs	3.1×	6.18 μs	2.4×
10	3.44 μs	1.0×	4.27 μs	1.2×	6.90 μs	2.0×	10.46 μs	3.0×	7.24 μs	2.1×
20	21.48 μs	1.0×	26.06 μs	1.2×	47.28 μs	2.2×	70.63 μs	3.3×	22.03 μs	1.0×
30	64.17 μs	1.0×	86.77 μs	1.4×	161.10 μs	2.5×	256.71 μs	4.0×	47.06 μs	0.7×

6.4 An Example Rendering Application

We also evaluate our method using an example graphics application: ray-hair intersections using polynomial hair curves with thickness. There are two reasons for picking this particular example. First, this is a challenging polynomial problem for our method, involving degree-10 polynomials and stress-testing our recursive approach. Second, there exists a relatively recent and highly-efficient customized solution for this problem: Reshetov and Luebke's [2018] *phantom ray-hair intersector*, providing a strong baseline to compare against.

A common approach for computing ray-hair intersections is comparing the distance of the closest point along the ray to the curve that passes through hair's center, the *center curve*. Ray hit is registered when this distance is smaller than the hair radius. This is a relatively easier problem for our method, but it does not provide the correct hit point with the hair's surface.

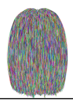
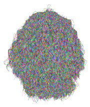
The phantom ray-hair intersector, however, computes ray intersections with *thick curves*, i.e. surfaces generated by sweeping a circle with varying radius along the center curve on its tangent plane. It finds the correct hit point with the hair's surface and catches intersections that the closest point approach can miss, so it is ideal for rendering close-up hair strands or cloth fibers. It is also efficient-enough to be used in distant views, where the hair thickness is imperceptible.

Directly solving for ray intersections with thick curves results in polynomials of degree 10 for cubic curves (see the supplemental document for the derivation), a previously impractical case for high-performance applications, motivating the phantom intersector method in the fist place. Yet, this is exactly what we do with our method. More specifically, we construct a degree 10 polynomial from the Bézier control points of the curve on-the-fly, right before each intersection test. This representation also includes curve's thickness variation defined by another cubic polynomial.

Our implementation uses Embree [Wald et al. 2014] ray tracing library for building and traversing the acceleration structure. Since our goal is to compare the cost of ray-hair intersection computation, we do not perform any shading and only use primary rays. This minimizes the computation differences due to possible variations in ray intersection decisions. Also, as with our other tests, we use a single CPU thread for rendering to avoid any parallelization overhead.

Resulting per-ray render times for primary rays are presented in Table 3. We use two hair models in these tests. The first one has more straight curves, providing a relatively easier case for phantom ray-hair intersector. Notice that for both models our method is 11% to 32% faster in these tests. Using 5× longer curves by down-sampling the original model reduces the efficiency of the ray tracing acceleration structure and inflates the overall render times with all methods, but it ensures

Table 3. *Hair rendering times with different methods using cubic hair curves.*

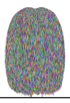
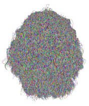
		d	Precision	Original Curves		5× Longer Curves	
	**Ours**, Closest Point	5	32 bits	0.33 μs/ray	0.53×	2.55 μs/ray	0.37×
	Ours, Thick Curves	10	64 bits	0.62 μs/ray	1.00×	6.81 μs/ray	1.00×
	Phantom Intersector	—	32 bits	0.69 μs/ray	1.11×	12.31 μs/ray	1.95×
	**Ours**, Closest Point	5	32 bits	0.53 μs/ray	0.47×	5.28 μs/ray	0.34×
	Ours, Thick Curves	10	64 bits	1.13 μs/ray	1.00×	15.74 μs/ray	1.00×
	Phantom Intersector	—	32 bits	1.50 μs/ray	1.32×	28.90 μs/ray	1.83×

that ray-hair intersection computation is the bottleneck. In this case, our method provides a more prominent performance advantage of 83% to 95% speed-up.

We also provide render times using the closest point approach, computed with our method by generating and then solving a degree 5 polynomial on-the-fly. As expected, it provides much shorter render times.

Additionally, in Table 4 we provide render times using the quadratic approximation of these cubic curves, generated using the method of Truong et al. [2020]. In this case, our method uses degree 6 polynomials with thick quadratic thick curves and cubic polynomials for the closest-point method. Therefore, this quadratic version significantly reduces the computation cost of our method, resulting in even more pronounced improvement over the phantom ray-hair intersector with 5× longer hair curves (more than 3× speed-up). Note that all methods are significantly faster with these quadratic curves, because they form curves with half the lengths of their cubic counterparts, improving the effectiveness of the ray tracing acceleration structure.

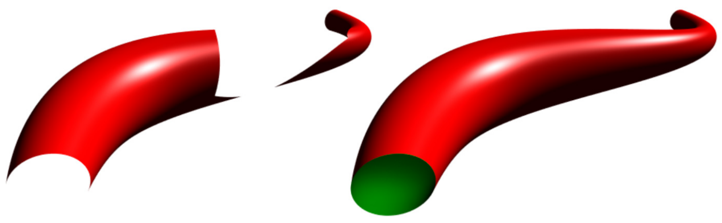
Table 4. *Hair rendering times with different methods using half-length quadratic hair curves.*

		d	Precision	Original Curves		5× Longer Curves	
	**Ours**, Closest Point	3	32 bits	0.22 μs/ray	0.66×	0.63 μs/ray	0.59×
	Ours, Thick Curves	6	64 bits	0.34 μs/ray	1.00×	1.08 μs/ray	1.00×
	Phantom Intersector	—	32 bits	0.36 μs/ray	1.07×	3.49 μs/ray	3.24×
	**Ours**, Closest Point	3	32 bits	0.29 μs/ray	0.68×	1.01 μs/ray	0.53×
	Ours, Thick Curves	6	64 bits	0.43 μs/ray	1.00×	1.90 μs/ray	1.00×
	Phantom Intersector	—	32 bits	0.59 μs/ray	1.38×	6.75 μs/ray	3.56×

These results show that our generic polynomial root finder, even with the overhead of generating the polynomials on-the-fly, can compete with and even perform faster than a customized high-performance solution for this problem. Nonetheless, the exact numbers in these results should not be overemphasized, as there are various factors determining the effective performance of a hair intersection routine.

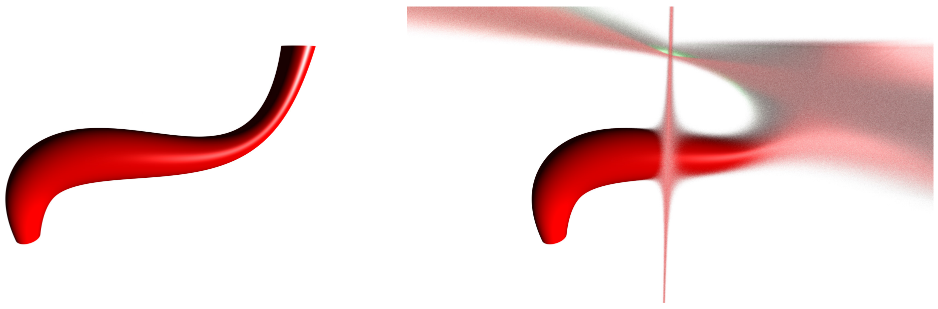
Furthermore, the shapes of the hair curves and the viewing angles impact the relative performance. Phantom ray-hair intersector exploits the relatively straight shape of typical hair strands and it can fail to converge with more complex shapes viewed from certain angles, as shown in Figure 3. By directly evaluating the polynomials, we avoid such limitations and also detect the intersections with the back side with minimal additional work.

Our solution, however, is not without potential errors either. In particular, numerical truncation can lead to substantial errors from certain view angles when using 32-bit precision, as shown in Figure 4. A more careful construction of the polynomials might mitigate such problems for this



(a) *Phantom Ray-Hair Intersector* **(b)** ***Ours*** *with 64-bit precision*

Fig. 3. *A thick cubic curve rendered using (a) phantom ray-hair intersector and (b) our method. Since our implementation finds all roots, we can also detect intersections with the back side.*



(a) ***Ours*** *with 64-bit precision* **(b)** ***Ours*** *with 32-bit precision*

Fig. 4. *An example showing potential consequences of numerical truncation with degree-10 polynomials using our method.*

example application. Nonetheless, it shows the importance of using higher-precision arithmetic while working with higher-order polynomials.

7 DISCUSSION

An important property of our approach is its robustness, i.e. it provides guaranteed convergence. Its accuracy, however, entirely depends on the chosen error threshold and the floating-point precision.

Two consecutive roots that are separated by less than ϵ or duplicated roots, forming polynomials that briefly cross zero, can be missed. This happens when the root is at or near a critical point. Detecting such roots is not guaranteed even with $\epsilon = 0$ due to numerical truncation that can prevent computing the exact position of the critical point using the polynomial's derivative. Such cases may require special handling. Fortunately, they are easy to isolate, as they happen when the polynomial is close to zero at a critical point.

Duplicated roots can also appear when solving a polynomial's derivative to find the critical points. Such occurrences, however, do not pose a problem for our method. This is because, even when a duplicated root of a polynomial's derivative is missed and the corresponding critical point is not found, the resulting interval that contains this missing critical point after our splitting step is monotonic. Therefore, any failure to detect such critical points is inconsequential.

Note that our discussions about duplicated roots also apply to any even number of coinciding roots. When an odd number of roots coincide, however, the polynomial changes sign, so such roots are easily detected. Yet, Newton iterations offer poor convergence with such roots.

As we show in the supplemental document, Newton iterations become inefficient near duplicated roots. Particularly with cubic polynomials in the form $ax^3 + d = 0$, where three roots coincide, Newton iterations converge linearly and even slower than pure bisection. More advanced methods that consider the second derivative (e.g. [McDougall et al. 2019]) can help and reduce the number of iterations, but they increase the iteration cost and negatively impact the overall performance in our experience. Yet, switching to a different numerical root finder for special cases, after computing the critical points, might be an interesting direction for future work.

Note that we do not use $|f(x)| \leq \epsilon$ as a termination condition for numerical root finding. Though using such a termination condition is sensible and would help with detecting duplicated roots, in graphics applications $f(x)$ and x may have very different units with very different scaling factors. Therefore, bounding the error in $f(x)$ by ϵ does not necessarily bound the error in x, which is why we refrain from using this termination condition.

8 CONCLUSION

We have presented an efficient and robust algorithm for computing the real roots of polynomials. Our solution can be used with polynomials of any degree, but it is more effective for relatively low-degree polynomials. We have shown that it outperforms both the analytical and numerical solution for cubic polynomials both in terms of computation speed and robustness. We have also shown that it can be used to replace existing solutions in graphics applications to deliver improved performance and accuracy.

ACKNOWLEDGMENTS

We are deeply grateful for the extremely helpful comments and suggestions by Alexander Reshetov. We thank Jim McCann, Ian Mallett, Kui Wu, Daqi Lin, and Peter Shirley for their feedback. We also thank Nathan Morrical for his help and Stefan Zellmann for sharing his implementation of the phantom intersector. Furthermore, we really appreciate the anonymous reviewers and the extremely valuable feedback they have provided to help us improve this work and our evaluation. This work was supported in part by NSF grant #1764071.

REFERENCES

Melike Aydinlilar and Cedric Zanni. 2021. Fast Ray Tracing of Scale-Invariant Integral Surfaces. *Computer Graphics Forum* 40, 6 (2021), 117–134. https://doi.org/10.1111/cgf.14208

J.F. Blinn. 2006a. How to solve a cubic equation. Part 1. The shape of the discriminant. *IEEE Computer Graphics and Applications* 26, 3 (2006), 84–93. https://doi.org/10.1109/MCG.2006.60

J.F. Blinn. 2006b. How to Solve a Cubic Equation, Part 2: The 11 Case. *IEEE Computer Graphics and Applications* 26, 4 (2006), 90–100. https://doi.org/10.1109/MCG.2006.81

J. F. Blinn. 2005. How to solve a Quadratic Equation. *IEEE Computer Graphics and Applications* 25, 06 (nov 2005), 76–79. https://doi.org/10.1109/MCG.2005.134

James F. Blinn. 2006c. How to Solve a Cubic Equation, Part 3: General Depression and a New Covariant. *IEEE Computer Graphics and Applications* 26, 6 (2006), 92–102. https://doi.org/10.1109/MCG.2006.129

James F. Blinn. 2007a. How to Solve a Cubic Equation, Part 4: The 111 Case. *IEEE Computer Graphics and Applications* 27, 1 (2007), 100–103. https://doi.org/10.1109/MCG.2007.10

James F. Blinn. 2007b. How to Solve a Cubic Equation, Part 5: Back to Numerics. *IEEE Computer Graphics and Applications* 27, 3 (2007), 78–89. https://doi.org/10.1109/MCG.2007.60

Gerolamo Cardano. 1570. *Artis Magnae, Sive de Regulis Algebraicis Liber Unus.*

Viktor Chlumský, Jaroslav Sloup, and Ivan Simecek. 2018. Improved Corners with Multi-Channel Signed Distance Fields. *Comput. Graph. Forum* 37, 1 (2018), 273–287. https://doi.org/10.1111/cgf.13265

Bruce Christianson. 1991. Solving quartics using palindromes. *The Mathematical Gazette* 75, 473 (1991), 327–328.

George E. Collins and Rüdiger Loos. 1976. Polynomial Real Root Isolation by Differentiation. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation (SYMSAC '76).* Association for Computing Machinery, New York, NY, USA, 15–25. https://doi.org/10.1145/800205.806319

Mark Dokter, Jozef Hladky, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel, and Markus Steinberger. 2019. Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU. *Computer Graphics Forum* 38, 2 (2019), 93–103. https://doi.org/10.1111/cgf.13622

N. Flocke. 2015. Algorithm 954: An Accurate and Efficient Cubic and Quartic Equation Solver for Physical Applications. *ACM Trans. Math. Softw.* 41, 4, Article 30 (oct 2015), 24 pages. https://doi.org/10.1145/2699468

M. A. Jenkins and J. F. Traub. 1970. A Three-Stage Algorithm for Real Polynomials Using Quadratic Iteration. *SIAM J. Numer. Anal.* 7, 4 (1970), 545–566.

Trevor J. McDougall, Simon J. Wotherspoon, and Paul M. Barker. 2019. An accelerated version of Newton's method with convergence order 3+1. *Results in Applied Mathematics* 4 (2019), 100078. https://doi.org/10.1016/j.rinam.2019.100078

John M. McNamee. 2007. *Numerical Methods for Roots of Polynomials - Part I.* Elsevier.

John M. McNamee and Victor Pan. 2013. *Numerical Methods for Roots of Polynomials - Part II.* Elsevier.

Bernard Mourrain, Fabrice Rouillier, and Marie-Françoise Roy. 2005. Bernstein's basis and real root isolation. In *Combinatorial and Computational Geometry*. MSRI Publications, Vol. 52. Cambridge University Press, 459–478.

Stefan Neumark. 1965. *Solution of Cubic and Quartic Equations.* Pergamon. 5–11 pages. https://doi.org/10.1016/B978-0-08-011220-6.50005-6

Victor Y. Pan. 2002. Univariate Polynomials: Nearly Optimal Algorithms for Numerical Factorization and Root-finding. *Journal of Symbolic Computation* 33, 5 (2002), 701–733. https://doi.org/10.1006/jsco.2002.0531

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing.* Cambridge University Press, USA.

Alexander Reshetov and David Luebke. 2018. Phantom Ray-Hair Intersector. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 34 (aug 2018), 22 pages. https://doi.org/10.1145/3233307

Michael Sagraloff and Kurt Mehlhorn. 2016. Computing real roots of real polynomials. *Journal of Symbolic Computation* 73 (2016), 46–86. https://doi.org/10.1016/j.jsc.2015.03.004

Michael A. Sherman, Ajay Seth, and Scott L. Delp. 2011. Simbody: multibody dynamics for biomedical research. *Procedia IUTAM* 2 (2011), 241–261. https://doi.org/10.1016/j.piutam.2011.04.023 IUTAM Symposium on Human Body Dynamics.

Sergei L Shmakov. 2011. A universal method of solving quartic equations. *Int. J. Pure Appl. Math* 71, 2 (2011), 251–259.

Markus Steinberger and Markus Grabner. 2010. Wavelet-based Multiresolution Isosurface Rendering. In *IEEE/ EG Symposium on Volume Graphics*. The Eurographics Association. https://doi.org/10.2312/VG/VG10/013-020

Peter Strobach. 2010. The Fast Quartic Solver. *J. Comput. Appl. Math.* 234, 10 (sep 2010), 3007–3024. https://doi.org/10.1016/j.cam.2010.04.015

Peter Strobach. 2011. Solving cubics by polynomial fitting. *J. Comput. Appl. Math.* 235, 9 (2011), 3033–3052. https://doi.org/10.1016/j.cam.2010.12.025

Min Tang, Dinesh Manocha, Miguel A. Otaduy, and Ruofeng Tong. 2012. Continuous Penalty Forces. *ACM Trans. Graph.* 31, 4, Article 107 (jul 2012), 9 pages. https://doi.org/10.1145/2185520.2185603

Min Tang, Ruofeng Tong, Zhendong Wang, and Dinesh Manocha. 2014. Fast and Exact Continuous Collision Detection with Bernstein Sign Classification. *ACM Trans. Graph.* 33, 6, Article 186 (nov 2014), 8 pages. https://doi.org/10.1145/2661229.2661237

Nghia Truong, Cem Yuksel, and Larry Seiler. 2020. Quadratic Approximation of Cubic Curves. *Proc. ACM Comput. Graph. Interact. Tech. (Proceedings of HPG 2020)* 3, 2, Article 16 (2020), 17 pages. https://doi.org/10.1145/3406178

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 33, 4, Article 143 (jul 2014), 8 pages. https://doi.org/10.1145/2601097.2601199

Michel Daoud Yacoub and Gustavo Fraidenraich. 2012. A solution to the quartic equation. *The Mathematical Gazette* 96, 536 (2012), 271–275. https://doi.org/10.1017/S002555720000454X

Zhipei Yan, Stephen Schiller, Gregg Wilensky, Nathan Carr, and Scott Schaefer. 2017. K-curves: Interpolation at Local Maximum Curvature. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2017)* 36, 4, Article 129 (2017), 7 pages. https://doi.org/10.1145/3072959.3073692

Cem Yuksel. 2020. A Class of C2 Interpolating Splines. *ACM Transactions on Graphics* 39, 5, Article 160 (jul 2020), 14 pages. https://doi.org/10.1145/3400301

Cem Yuksel. 2022. Polynomial Roots. http://www.cemyuksel.com/?x=polynomials