High-Performance Polynomial Root Finding for Graphics

Supplemental Document

CEM YUKSEL, University of Utah

In this supplemental document we present additional experiments with randomly-generated Bernstein polynomials and specific cubic polynomial types. We also describe our ray intersection test with curves, using the closest point and thick curves. Finally, we include the pseudocode of our method.

ACM Reference Format:
Cem Yuksel. 2022. High-Performance Polynomial Root Finding for Graphics: Supplemental Document. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 1 (July 2022), 11 pages. https://doi.org/10.1145/3406185

1 CUBIC POLYNOMIALS

In addition to the single (32-bit) precision tests in the paper, here we present our test results with randomly-generated Bernstein polynomials using double (64-bit) precision floating-point numbers. Figure 1 shows the performance results for finding all roots. These results with double precision are slightly different, showing that the analytical solution is slightly faster than "Ours (low)" when there are two roots.

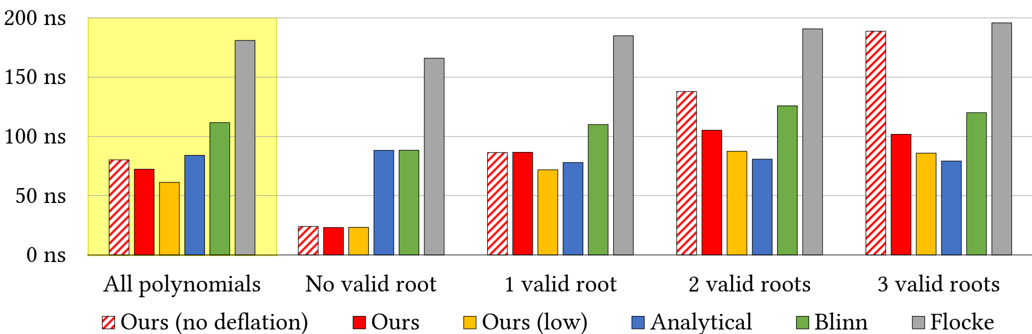

Fig. 1. ***Cubic polynomial double precision (64-bits) performance tests for finding all roots*** *within* $[0, 1]$ *for randomly-generated Bernstein polynomials with coefficients in* $[-1, 1]$. *The "all polynomials" group includes 27% with no root, 48% with a single root, 23% with two roots, and 2% with three roots. The other groups show the computation times for polynomials with different numbers of roots within* $[0, 1]$. *All numerical methods use* $\epsilon = 10^{-8}$, *except for "Ours (low)" that uses* $\epsilon = 5 \times 10^{-4}$, *producing an average error of* 5×10^{-17} *with "Ours" and* 7×10^{-8} *with "Ours (low)."*

Figure 2 shows the double (64-bit) precision tests for finding the first root within the target interval. These results are similar to the single (32-bit) precision ones in the paper, except that all methods take slightly longer time.

We have also experimented with different numerical root finding methods: *bisection*, *Ridder's method*, and *regula falsi* (the false position method). All of these methods provide guaranteed convergence and match the given error threshold ϵ. However, with our randomly-generated Bernstein polynomials, they all resulted in significantly slower convergence.

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

2020. 2577-6193/2022/7-ART
https://doi.org/10.1145/3406185

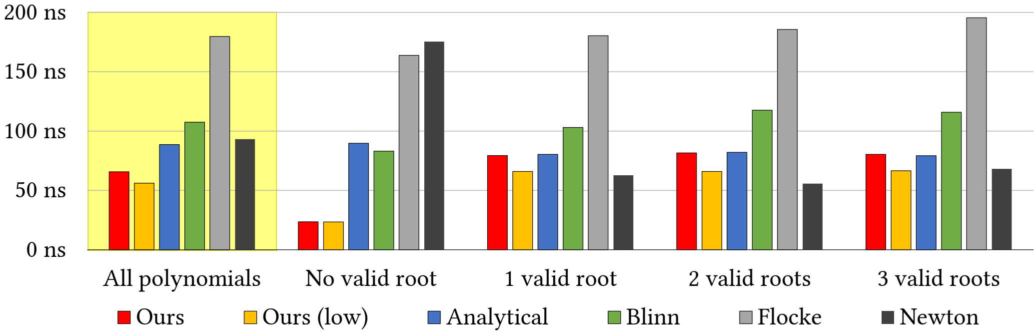

Fig. 2. *Cubic polynomial double precision (64-bits) performance for finding the first root* *using the same test setup in Figure 1. Note that regular "Newton" iterations fail to find an existing root for 18% of all polynomials (33% of polynomials with one root and 11% of them with two roots).*

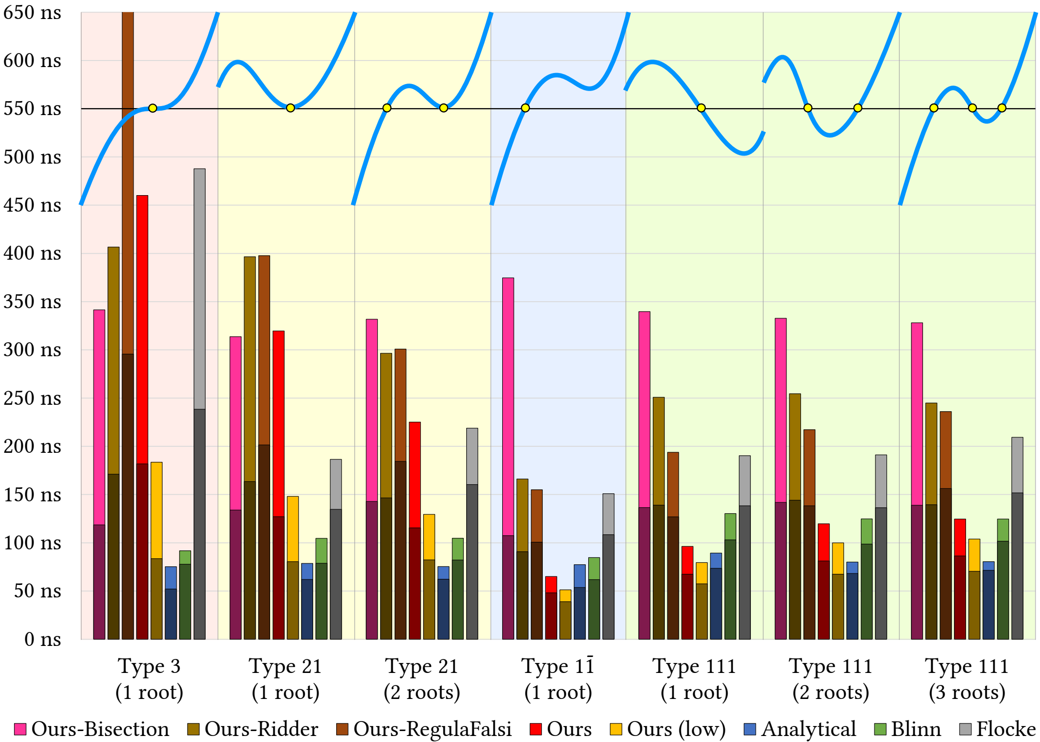

Fig. 3. *Experiments with different cubic polynomial types* *with different numbers of root within the target interval. The darker portions show the performance values with single (32-bit) precision experiments and the full bars correspond to double (64-bit) precision experiments.*

Nonetheless, there are special cases when these alternative numerical root finding methods can outperform the hybrid method we use combining Newton iterations with bisection. For exploring such cases, we have performed tests using different types of cubic polynomials, as classified by Blinn [2006]. The results are shown in Figure 3 along with a representative example for each type.

Type 3 cubic polynomials (the first group) include three roots at the same position with polynomials of the form $ax^3 + d = 0$. In this case, the convergence rate of Newton iterations drop from quadratic to linear. In fact, they perform clearly worse than bisection, which provides the best performance for this case among all methods we have tested. Indeed, every bisection iteration reduces the error by half, while Newton iterations can only reduce by 1/3 in this case. Notice that Flocke's approach of picking a good starting guess for Newton iterations (and shifting the polynomial to improve the accuracy of this special case) cannot fully save it from the reduced convergence rate. Yet, this starting guess certainly helps, as the absolute difference between "Ours" and Flocke's method is reduced in this case, though both methods perform poorly.

Type 21 cubic polynomials (the second and third groups) have one double and one single root, representing another challenging case for Newton iterations. These polynomials barely touch the $y = 0$ line at the double root. In these experiments, the double root is always inside the target interval. Finding this double root is the most challenging case for our method. In fact, both with 32-bit and 64-bit floating-point numbers, our method failed to find the root for 7.4% of these polynomials, regardless of the chosen error threshold (including $\epsilon = 0$). The other methods in Figure 3 had even higher failure rates: the analytical solution failed 33% of them, Blinn's method failed 12% with 32-bits and 21% with 64-bits, and Flocke's method failed 40% of them. RPOLY, however, failed for only 5% of these polynomials.

The computation cost of these double roots is another source of concern. Again, Newton iterations result in a poor convergence rate and bisection provides a similar performance in our tests.

Type $1\bar{1}$ cubic polynomials include one single real root and one complex conjugate pair. Finding the real root poses no difficulty for our method in this case. The same is true for Type 111 cubic polynomials with three distinct real roots. Type $1\bar{1}$ and Type 111 cubic polynomials are the most common ones and Newton iterations provide a good convergence rate in these cases, resulting in much better performance than the alternative numerical root finding techniques.

We have also experimented with the Accelerated Newton's Method of McDougall et al. [2019], which uses an estimate of the second derivative to reduce the iteration count. In our tests, the reductions in the iteration count was not sufficient to cover the additional cost per iteration. The one exception is the case of Type 21 cubic polynomials, in which the Accelerated Newton's Method resulted in about 10% faster computation than ours using the same error threshold.

2 ERROR THRESHOLD

The error threshold parameter ϵ determines the resulting accuracy and the computation cost. We provide experiments with randomly-generated Bernstein polynomials for finding all roots in $[0, 1]$ with different ϵ parameters. We also provide results with $\epsilon = 0$ and 0^* that includes additional computations to bound the error and skips the deflation optimization for best accuracy, which is taken as the reference for computing the error of the solutions with other ϵ values.

Figure 4 show our results with single (32-bit) precision, grouped by different ϵ parameter values. The bars show ϵ, the maximum error, and the average error. We also show the average value of the polynomials computed at the roots found and the computation times as curves. Notice that single (32-bit) precision is not sufficient to bound the maximum error when ϵ is relatively small. Through reducing ϵ can reduce the average error, numerical truncation can lead to a larger error than ϵ in some cases, particularly when $f(x) = 0$ for a range of x values near the actual root. Notice that the average error can be more than 3 orders of magnitude smaller than ϵ in these tests (see $\epsilon = 10^{-3}$).

Our results with double (64-bit) precision are shown in Figure 5 for degrees 3, 4, and 5, and Figure 6 for degrees 6, 8, and 10. Double (64-bit) precision is significantly better for bounding the error, though numerical truncation still limits the maximum error. Notice that the average error can be more than 6 orders of magnitude smaller than ϵ, particularly with $\epsilon = 10^{-6}$ in these tests.

Fig. 4. *The error and computation times for different error threshold ϵ parameters for polynomials of degrees 3, 4, and 5 with single (32-bit) precision.*

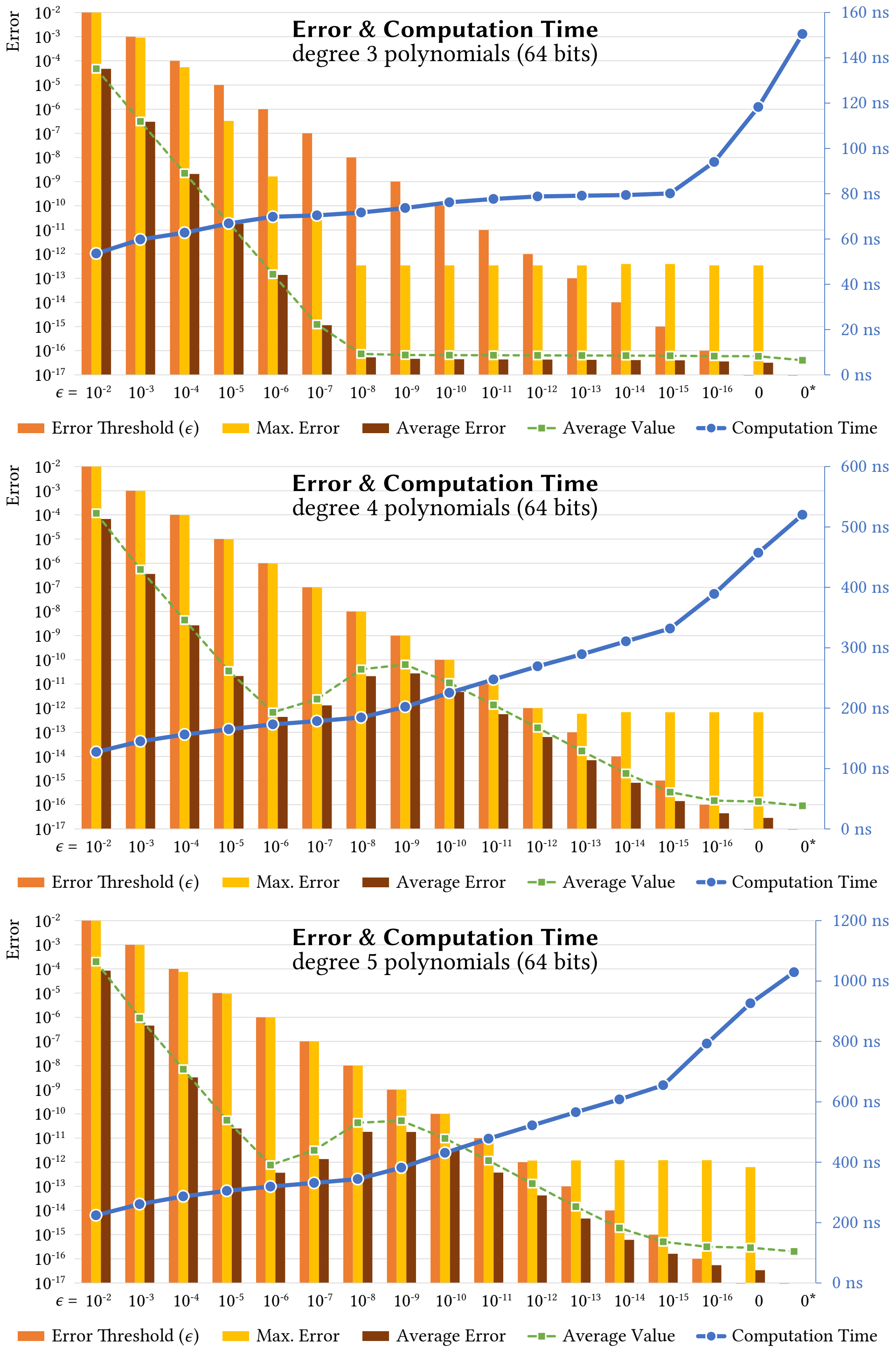

Fig. 5. *The error and computation times for different error threshold ϵ parameters for polynomials of degrees 3, 4, and 5 with double (64-bit) precision.*

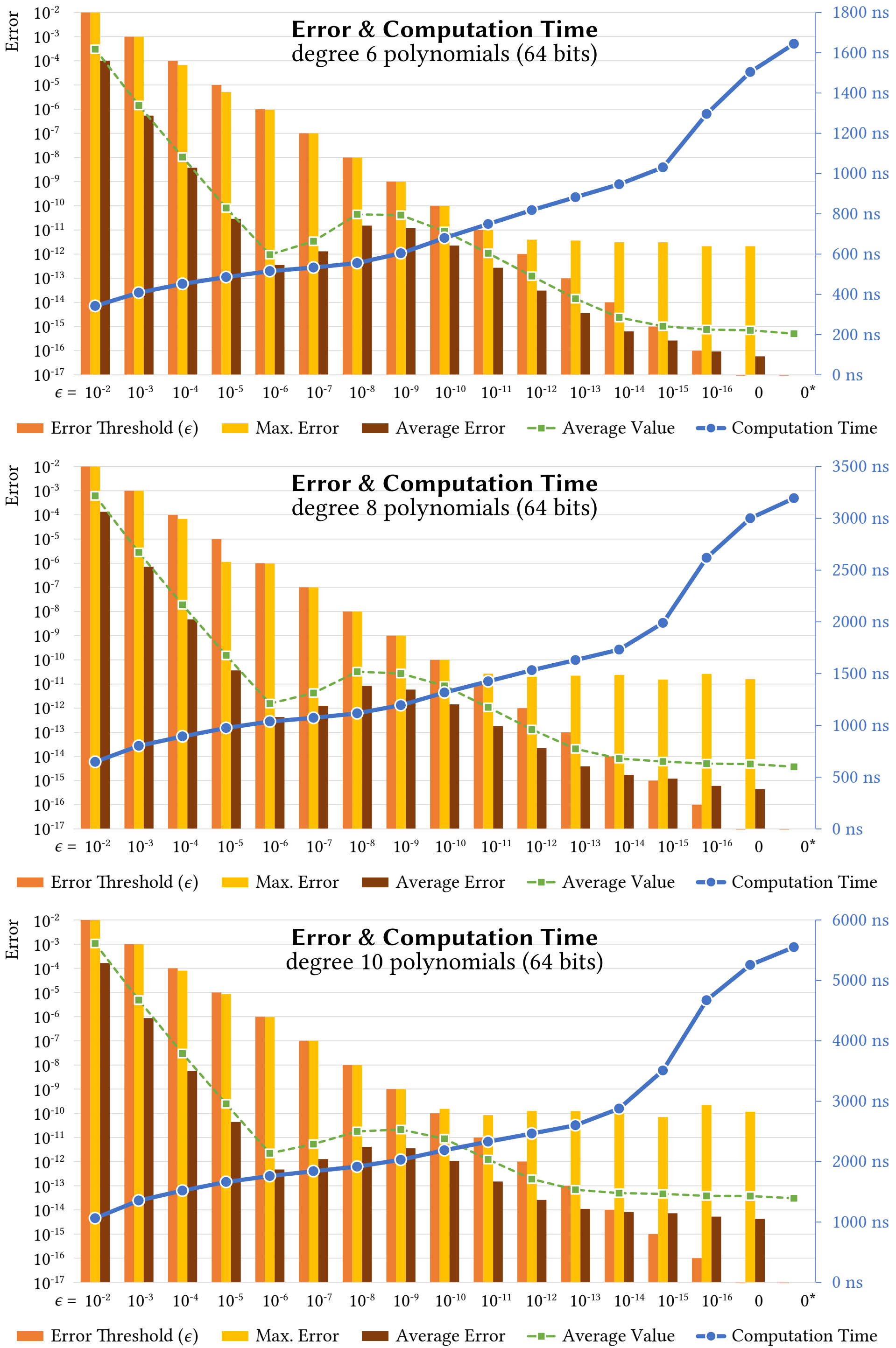

Fig. 6. *The error and computation times for different error threshold ϵ parameters for polynomials of degrees 6, 8, and 10 with double (64-bit) precision.*

3 COMPARISON TO QR DECOMPOSITION

An alternative method for solving higher-order polynomials is using QR decomposition of the polynomial's companion matrix. The Eigen library [Guennebaud et al. 2010] includes an unsupported branch with an implementation of this approach. Our experiments with this implementation resulted in 50× to 100× slower results as compared to our method. Yet, these results may not be representative of the QR method.

4 POLYNOMIAL RAY-CURVE INTERSECTIONS

In the paper, we present experiments with ray-curve intersections for hair rendering, using either the closest-point method or thick curves. Here, we provide the derivations of polynomial formulations for these intersection tests.

4.1 Closest Point on along Ray to a Curve

Let $\mathbf{p}$ be a point along a ray with origin $\mathbf{q}$ and direction $\mathbf{d}$, such that $\mathbf{p} = \mathbf{q} + t\mathbf{d}$, where t is the parameter along the ray, $\mathbf{f}(s)$ represent a parametric curve formulation and $r(s)$ is the thickness (i.e. radius) of the curve at any parameter value s. We represent the derivative of the curve as $\mathbf{f}'(s) = d\mathbf{f}/ds$. At the closest point $\mathbf{p}$, the vector to the closest point on the curve $\mathbf{f}(s) - \mathbf{p}$ must be orthogonal to the ray, such that

$$\big(\mathbf{f}(s) - \mathbf{p}\big) \cdot \mathbf{d} = 0 \; . \tag{1}$$

Expanding this we can write

$$t = \frac{\big(\mathbf{f}(s) - \mathbf{q}\big) \cdot \mathbf{d}}{\mathbf{d}^2} \; , \tag{2}$$

where $\mathbf{d}^2 = \mathbf{d} \cdot \mathbf{d}$. At the closest point, $\mathbf{f}(s) - \mathbf{p}$ must also be orthogonal to the curve direction $\mathbf{f}'(s)$. Thus, we can write

$$\big(\mathbf{f}(s) - \mathbf{p}\big) \cdot \mathbf{f}'(s) = 0 \; , \tag{3}$$

which expands to

$$\big(\mathbf{f}(s) - \mathbf{q}\big) \cdot \mathbf{f}'(s) - t\mathbf{d} \cdot \mathbf{f}'(s) = 0 \; . \tag{4}$$

Substituting t and multiplying by $\mathbf{d}^2$, we get an equation that does not depend on t and only depends on s

$$\big(\mathbf{f}(s) - \mathbf{q}\big) \cdot \big(\mathbf{d}^2\mathbf{f}'(s) - (\mathbf{d} \cdot \mathbf{f}'(s))\; \mathbf{d}\big) = 0 \; . \tag{5}$$

Note that if $\mathbf{f}$ is a polynomial of degree n, $\mathbf{f}'$ is a polynomial of degree $n-1$, so the equation above is a polynomial of degree $2n-1$. Therefore, for a cubic polynomial curve it is degree 5 and for a quadratic curve it is degree 3.

To find the closest point, we must find the real roots s_i of Equation 5. Then, we can compute t and compare $\|\mathbf{f}(s_i) - \mathbf{p}\|$ to $r(s_i)$ for each root within the valid range.

Simplified Case Using Ray Coordinates. Consider the simplified case using ray coordinates where $\mathbf{q} = (0,0,0)$ and $\mathbf{d} = (0,0,1)$. Then, Equation 5 simplifies down to

$$f_x(s)\, f_x'(s) + f_y(s)\, f_y'(s) = 0 \; . \tag{6}$$

When $\mathbf{f}$ is a cubic polynomial, such that

$$f_x(s) = x_3\, s^3 + x_2\, s^2 + x_1\, s + x_0 \; , \text{ and} \tag{7}$$

$$f_x'(s) = 3x_3\, s^2 + 2x_2\, s + x_1 \; , \tag{8}$$

we get

$$f_x(s)f_x'(s) = 3x_3^2\,s^5 + 5x_3x_2\,s^4 + (2x_2^2 + 4x_3x_1)s^3 + (3x_2x_1 + 3x_3x_0)s^2 + (x_1^2 + 2x_2x_0)s + x_1x_0\ .$$

Once s is found, we can compute t using Equation 2, which simplifies down to $t = f_z(s)$.

4.2 Ray Intersection with a Thick Curve

A thick curve is defined by a center curve $\mathbf{f}(s)$ and a radius $r(s)$. All points $\mathbf{p}$ on the surface of a thick curve must satisfy two equations. The first one is the distance to the curve center

$$\|\mathbf{f}(s) - \mathbf{p}\| = r(s)\ , \tag{9}$$

which can be extended to

$$\big(\mathbf{f}(s) - \mathbf{q} - t\mathbf{d}\big)^2 - \big(r(s)\big)^2 = 0\ . \tag{10}$$

The second equation declares that a point on the curve must be on the plane defined by the corresponding center line position and direction. Thus,

$$\big(\mathbf{f}(s) - \mathbf{q} - t\mathbf{d}\big) \cdot \mathbf{f}'(s) = 0\ . \tag{11}$$

By rearranging the terms we get

$$t = \frac{\big(\mathbf{f}(s) - \mathbf{q}\big) \cdot \mathbf{f}'(s)}{\mathbf{d} \cdot \mathbf{f}'(s)}\ . \tag{12}$$

Then, we can substitute t into the distance equation and multiply by $\big(\mathbf{d} \cdot \mathbf{f}'(s)\big)^2$ to get (dropping function notation for brevity)

$$\Big((\mathbf{d} \cdot \mathbf{f}')(\mathbf{f} - \mathbf{q}) - \big((\mathbf{f} - \mathbf{q}) \cdot \mathbf{f}'\big)\,\mathbf{d}\Big)^2 - (\mathbf{d} \cdot \mathbf{f}')^2 r^2 = 0\ . \tag{13}$$

If $\mathbf{f}(s)$ is a polynomial of degree n, then left term in this equation is a polynomial of degree $4n - 2$ and the right term is degree $2n + 2m - 2$, assuming r is a polynomial of degree m. For $m \le n$, it becomes degree $4n - 2$. Thus, if $\mathbf{f}$ is cubic, the equation above becomes degree 10, and if $\mathbf{f}$ is quadratic, it is degree 6.

Simplified Case Using Ray Coordinates. Once again, consider the simplified case using ray coordinates where $\mathbf{q} = (0, 0, 0)$ and $\mathbf{d} = (0, 0, 1)$. Then, Equation 13 simplifies down to

$$(f_z')^2\left(f_x^2 + f_y^2 - r^2\right) + (f_x'\,f_x + f_y'\,f_y)^2 = 0$$

Once we solve for s, we can compute t using Equation 12, which simplifies to

$$t = \frac{\mathbf{f}(s) \cdot \mathbf{f}(s)'}{f_z'(s)}\ . \tag{14}$$

5 PSEUDOCODE

We show the pseudocode of our method in Algorithm 1, showing the splitting step. It is slightly modified for cubic polynomials using the deflation optimization, as shown in Algorithm 2. A high-performance CPU implementation for cubic polynomials would expand the `for` loop on line 6 of Algorithm 2 to minimize conditional jumps. A high-performance GPU implementation should likely move the root finding (line 8) outside of the `for` loop to minimize thread divergence.

Both of these functions call the numerical root finding function when they detect the presence of a root. Our numerical root finding with optimizations for cubic polynomials is presented in Algorithm 3. Notice that this pseudocode checks the remaining size of the interval only after a

Algorithm 1: Pseudocode of our splitting step.

```
1  function PolynomialRoots(f, x_start, x_end)
2      X_C ← PolynomialRoots(f′, x_start, x_end)      // Find the critical points (the roots of f′).
3      X_E ← X_C ∪ {x_end}                            // Interval ends are the critical points plus x_end.
4      X_R ← {∅}                                      // Initialize the set of roots as empty.
5      x_1 ← x_start                                  // Set the start of the first interval.
6      y_1 ← f(x_1)                                   // Evaluate the polynomial at the start position.
7      foreach interval end x_2 ∈ X_E do              // For each interval
8          y_2 ← f(x_2)                               // Evaluate the polynomial at the end position.
9          if sgn y_1 ≠ sgn y_2 then                  // If there is a root within this interval
10             x_R ← FindRoot(f, f′, x_1, x_2, y_1, y_2)   // Find the root.
11             X_R ← X_R ∪ {x_R}                      // Add it to the list of roots.
12         end
13         x_1 ← x_2                                  // Set the start of the next interval.
14     end
15     return X_R                                     // Finally, return the set of roots.
```

Algorithm 2: Pseudocode of our splitting step with deflation (used for cubic polynomials).

```
1  function PolynomialRoots(f, x_start, x_end)
2      X_C ← PolynomialRoots(f′, x_start, x_end)      // Find the critical points (the roots of f′).
3      X_E ← X_C ∪ {x_end}                            // Interval ends are the critical points plus x_end.
4      x_1 ← x_start                                  // Set the start of the first interval.
5      y_1 ← f(x_1)                                   // Evaluate the polynomial at the start position.
6      foreach interval end x_2 ∈ X_E do              // For each interval
7          y_2 ← f(x_2)                               // Evaluate the polynomial at the end position.
8          if sgn y_1 ≠ sgn y_2 then                  // If there is a root within this interval
9              x_R ← FindRoot(f, f′, x_1, x_2, y_1, y_2)   // Find the first root.
10             X_R ← {x_R}                            // Create a list of roots.
11             if there is another root then          // If there is at least another interval with a root
12                 g ← deflated polynomial of f       // Set g as the deflated polynomial of f
13                 X_D ← PolynomialRoots(g, x_c, x_end)   // Find the roots of g
14                 X_R ← {x_R} ∪ X_D                  // Form the final list of roots.
15                 return X_R                         // Return the list of roots.
16             else return {x_R}                      // Return the one root we found.
17         end
18         x_1 ← x_2                                  // Set the start of the next interval.
19     end
20     return {∅}   // If we reach here, it means we have not found a root, so return empty set.
```

Algorithm 3: Pseudocode of our root finding with optimization for cubic polynomials.

```
1  function FindRoot(f, f', x1, x2, y1, y2)
2    | xr ← (x1 + x2)/2            // The initial guess is the center of the given interval.
3    | if |x2 − x1| ≤ 2ϵ then return xr // if the interval is small enough, return the initial guess
4    | if f is a cubic polynomial then // If f is a cubic polynomial, we can use a simpler loop.
5    |   | for a fixed number of iterations do     // Begin with a fixed number of iterations.
6    |   |   | xn ← xr − f(xr)/f'(xr)                          // Compute the Newton step.
7    |   |   | xn ← max(x1, min(x2, xn))                 // Clamp xn to the target interval.
8    |   |   | if |xr − xn| ≤ ϵ then return xn                   // If converged, return xn.
9    |   |   | xr ← xn                     // If not converged, take the step and continue.
10   |   | end
11   |   | if xr ∉ [x1, x2] then   // If xr is not a valid number (i.e. rare division by zero cases)
12   |   |   | xr ← (x1 + x2)/2                         // Start over with the initial guess.
13   |   | end
14   | end
15   | yr ← f(xr)                                         // Compute the polynomial at xr.
16   | while true do                                           // We loop until we converge.
17   |   | if sgn yr = sgn y1 then                          // If yr has the same sign as y1
18   |   |   | x1 ← xr     // Shrink the interval by setting the new start of the search interval.
19   |   | else                               // Otherwise (if yr and y1 have different signs)
20   |   |   | x2 ← xr       // Shrink the interval by setting the new end of the search interval.
21   |   | end
22   |   | xn ← xr − yr/f'(xr)                                   // Compute the Newton step.
23   |   | if x1 < xn < x2 then                                  // If it is a valid Newton step
24   |   |   | if |xr − xn| > ϵ then           // If the step size is larger than the error bound
25   |   |   |   | xr ← xn                                                  // Take the step.
26   |   |   |   | yr ← f(xr)   // Compute the polynomial at the new position and continue.
27   |   |   | else                                        // If the step size is small enough
28   |   |   |   | if sgn yr = sgn y1 then xr ← xn + ϵ              // Try to move the guess to
29   |   |   |   | else xr ← xn − ϵ                                  // the other side of the root.
30   |   |   |   | y ← f(xr)                  // Compute the polynomial at this new position.
31   |   |   |   | if sgn y ≠ sgn yr then             // If it is indeed on the other side of the root
32   |   |   |   |   | return xn                           // We can return our latest Newton step.
33   |   |   |   | else                                      // Otherwise, xr is closer to the root.
34   |   |   |   |   | yr ← y                    // We have already computed the polynomial at xr.
35   |   |   |   | end
36   |   |   | end
37   |   | else                                                       // If the Newton step failed
38   |   |   | xr ← (x1 + x2)/2                                      // We take a bisection step.
39   |   |   | if xr = x1 or xr = x2 or x2 − x1 ≤ 2ϵ then            // If the interval is too small
40   |   |   |   | return xr                                        // Return the bisection step.
41   |   |   | else yr ← f(xr)   // Compute the polynomial at the new position and continue.
42   |   | end
43   | end
```

bisection step. This may cause taking an extra step, but slightly reduces the cost of each step, providing faster results in our tests.

The C++ source code of our polynomial root finding method is available online [Yuksel 2022]. Note that this implementation may differ from the simplified pseudocodes in this document to achieve additional performance optimizations.

REFERENCES

J.F. Blinn. 2006. 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

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

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

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