Project 1: Root Finding via Lagrange Polynomial Interpolation

Max Ortman Teammate 1 Teammate 2

February 15, 2026

The Algorithm

Each step we create a Lagrange Polynomial using four sampled points. We then evaluate the roots of that polynomial and add whatever point is closest to zero as a new sampled point. We then remove the point with the largest x value while making sure there is at least one point on each side of zero. This process is repeated until we have reached the desired number of iterations or the error is sufficiently small. Our method assumes that a range that has points on either side of zero is given, and the function only crosses zero once in that range.

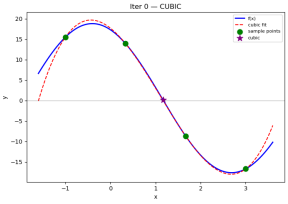
The code is too long to fit in the doc so please check out the attached main.py file for the implementation as we handle edge cases. But the sudo code is as follows:

```
def rootfind(f_:Callable, min:float, max:float) -> float:
  # get four points
  points = [min, min+(max-min)/3, min+2*(max-min)/3, max]

  cubic_count = 0
  bisect_count = 0

  for i in range(max_iter_):
      a, b, c, d = make_lagrange_cubic(points)
      roots = cubic_roots(a, b, c, d)

      # Filter the roots to meet the following:
      #  Root is within the range of points
      #  Root of polynomial is within a tolerance of being a root
      #  Root is not too close to the last best root (to prevent loops)
      canidate = get_candidate(roots, points)
      if canidate is not None:
        best = canidate
      else:
        # If no candidate roots are found we bisect the closest
        #  positive and negative points to find a new candidate root
        best = (min_pos[0] + min_neg[0]) / 2

      # Sample new point and add to list
      best_y = f_(best)

      # Return if within tolerance
      if abs(best_y) < tol_:
        return best

      # Add point
      points += (best, best_y)

      # Remove the nearest point on both sides of the root

      # Get xs and ys without the closest pos and neg to keep bounds
      filtered_points = [points not in (min_pos, min_neg)]
      furthest = furthest_point(filtered_points, best)
      points.remove(furthest)
```

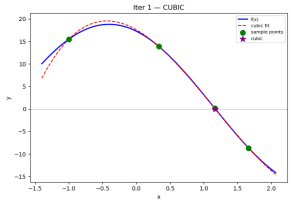
Explanation

You can use an interactive version in desmos with 3 points: `https://www.desmos.com/calculator/atqa25yh0u` to see how the method works. Blue is first step and red is the second.

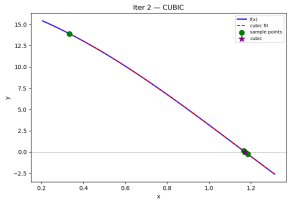
Here is also our method taking on a particularly challenging function. You can see it samples really close then slowly moves the leftmost point inwards. This is a larger issue that leads to some cases not performing as well as we would like. But when it doesn't do this it can one shot some examples



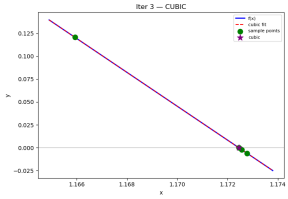
(a) Iteration 0



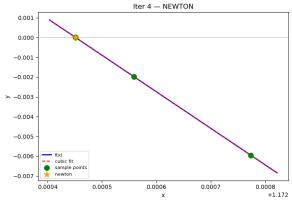
(b) Iteration 1



(c) Iteration 2



(d) Iteration 3



(e) Iteration 4

Figure 1: Visualization of the method on a challenging function. It samples closely (blue) and moves the leftmost point inwards (red). Note the performance in later iterations.

To estimate the zero for this function, we use sampled points to construct a Lagrange Polynomial. The basis polynomials are defined as:

$$L_i(x) = \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)}$$

Using these basis functions, the unique polynomial of degree n that passes through all $n + 1$ points is:

$$P(x) = \sum_i y_i L_i(x)$$

To start our method we sample the beginning, end, and middle 2 points. This gives us a set of 4 points to construct our Lagrange Polynomial. We then evaluate $P(x) = 0$ to get our initial estimate of the root. We then add that as a middle point and remove the point with the largest x value while making sure there is at least one point on each side of zero. This process is repeated until we have reached the desired number of iterations or the error is sufficiently small.

Analysis & Results

We benchmarked our method against Newton's method across 8 test functions. To avoid lucky initial conditions, we ran 10 trials per function with a random ± 0.1 jitter on the interval ends. Both methods use a tolerance of 10^{-12}. We also track whether each iteration used a cubic interpolation step or fell back to bisection, and whether the method failed to converge (hit the iteration cap or lost its bracket).

Table 1: Rootfinding comparison averaged over 10 trials with randomized bounds

	Iterations		Function Calls		Step Type		
Function	Cubic	Newton	Cubic	Newton	Cubic	Bisect	Fails
$sin(x) + 0.2$	3.0	3.0	7.0	10.0	3.0	0.0	0/10
$x^3 - x - 2$	1.0	3.7	5.0	12.1	1.0	0.0	0/10
$cos(x) - x$	3.7	4.0	7.7	13.0	3.5	0.2	0/10
$e^x - 3$	4.0	3.8	8.0	12.4	4.0	0.0	0/10
$x^5 - x - 1$	33.9	5.8	36.7	18.4	3.5	30.4	0/10
$ln(x) - 1$	5.0	4.0	9.0	13.0	5.0	0.0	0/10
$tan(x) - 2x$	100.0	8.0	74.0	25.2	35.0	35.0	10/10
$x^2 - 2$	1.0	3.6	5.0	11.8	1.0	0.0	0/10
Overall Avg	18.9	4.5	19.1	14.5	7.0	8.2	10/80
Mainly Cubic Avg	3.0	3.7	7.0	12.1	2.9	0.0	0/60
Mainly Bisect Avg	67.0	6.9	55.4	21.8	19.3	32.7	10/20

When the cubic step works (cubic-dominant functions), our method averages 3.0 iterations and 7.0 function calls, beating Newton's 3.7 iterations and 12.1 calls. Functions like $x^3 - x - 2$ and $x^2 - 2$ converge in a single

iteration since the Lagrange cubic fits them almost exactly. But it still performs just as well on functions like $sin(x) + 0.2$ and $e^x - 3$ where the cubic is a rougher approximation.

When the cubic step fails and we fall back to bisection (bisect-dominant functions), performance degrades significantly. For $x^5 - x - 1$, only 3.5 of 33.9 iterations are cubic — the rest are bisection. For $\tan(x) - 2x$, the method fails on all 10 trials: the near-singularity causes all four sample points to end up on the same side of zero, breaking the bracket. The cubic + bisect counts don't sum to 100 here because the method breaks early when it loses the bracket.

Our function also performs much better when it comes to function sampling. With the mainly cubic functions, it averages 7.0 calls vs Newton's 12.1.

This method can be very efficient when the cubic step works, but it can also be very slow when it doesn't. I would say replacing the bisect with newton's method would be a good next step to improve performance on the harder functions, but it would also add complexity and potential failure modes. But in a dataset of mainly polynomial-like functions or functions that have a high sample cost, this method could be a strong choice.