PolySolve: A Numerically Stable Heuristic for Polynomial Root-Finding via Accelerated Genetic Algorithms

Jonathan Rampersad

November 24, 2025

Abstract

Finding the roots of high-degree polynomials is a fundamental problem in computational mathematics, with applications ranging from control theory to signal processing. While deterministic methods such as the companion matrix eigenvalue algorithm (used by NumPy) are computationally efficient, they often suffer from significant numerical instability and accuracy loss when applied to high-degree polynomials. This paper presents **PolySolve**, a novel Python library that implements a stochastic Genetic Algorithm (GA) to approximate real roots. To address the computational cost inherent in heuristic search, we implement a highly parallelized fitness function using **Numba** for CPU acceleration and **CUDA** for GPU acceleration. Our benchmarks demonstrate that while the GA is orders of magnitude slower than deterministic methods, it maintains exceptional numerical stability (Mean Absolute Error $\approx 10^{-15}$) for polynomials up to degree 100, a regime where standard deterministic solvers fail to maintain precision. Furthermore, our CUDA implementation achieves a $\approx 20\times$ speedup over the CPU baseline, demonstrating the feasibility of GPU-accelerated heuristic solvers for high-precision tasks.

1 Introduction

Polynomial root-finding is a classical problem in mathematics. For a polynomial $P(x)$ of degree n, the Fundamental Theorem of Algebra states there are exactly n complex roots. For $n < 5$, analytical solutions exist (e.g., the quadratic formula). However, for $n \geq 5$, solutions must be found numerically.

The current industry standard for this task, exemplified by the `numpy.roots` function, relies on constructing the **companion matrix** of the polynomial and computing its eigenvalues [1]. While this approach is deterministic and highly optimized for speed ($O(n^3)$), it is known to be numerically unstable for high-degree polynomials due to the ill-conditioning of the companion matrix [2]. In these high-degree regimes, floating-point errors accumulate, leading to results that diverge significantly from the true roots.

In this work, we propose a heuristic alternative: a **Genetic Algorithm (GA)** tailored for continuous optimization. Unlike deterministic solvers, a GA is less susceptible to the specific conditioning pitfalls of matrix operations. While historically considered too computationally expensive for root-finding, modern hardware acceleration allows us to reconsider their viability.

We introduce `PolySolve`, a library that leverages **Numba** (Just-In-Time compilation) and **CUDA** (Compute Unified Device Architecture) to accelerate the evaluation of candidate solutions.

Our contributions are as follows:

1. We design a continuous-domain Genetic Algorithm specifically for finding real roots of polynomials.
2. We implement a massive-scale parallel fitness evaluation kernel using CUDA, achieving significant performance gains on consumer GPUs.
3. We perform a comparative analysis against NumPy, demonstrating that `PolySolve` retains near-perfect accuracy (MAE $\approx 10^{-16}$) at degree 100, whereas the standard companion matrix approach degrades to an error magnitude of 10^{-5}.

2 Background and Related Work

2.1 Deterministic Root-Finding

The standard numerical approach for finding polynomial roots is the **eigenvalue method**. Given a monic polynomial $P(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_0$, one can construct a companion matrix C such that the characteristic polynomial of C is $P(x)$. The eigenvalues of C are the roots of $P(x)$ [1].

This method is implemented in widespread libraries such as NumPy and MATLAB. However, as noted by Wilkinson [2], the problem of finding roots can be ill-conditioned; tiny perturbations in coefficients (or floating-point rounding errors) can result in massive changes in the computed roots. This instability becomes pronounced as the degree n increases.

2.2 Genetic Algorithms in Continuous Domains

Genetic Algorithms are a class of evolutionary algorithms inspired by natural selection. A GA maintains a population of candidate solutions and iteratively refines them using selection, crossover, and mutation operations [3].

While GAs are typically slower than gradient-based or analytical methods, they excel in non-differentiable or complex search spaces. In the context of root-finding, a GA treats the root as a global optimization problem: minimizing the error $|P(x)|$. Previous work has explored this [4], but often without the hardware acceleration necessary to make the approach practical for high-precision requirements.

3 Methodology

We formulate the root-finding problem as a minimization task. We seek a value $x \in \mathbb{R}$ such that $|P(x)| \to 0$.

3.1 Genetic Algorithm Design

`PolySolve` implements a specialized Genetic Algorithm optimized for this continuous 1D search space.

Representation: Each individual in the population is represented by a single 64-bit floating-point number (double precision), corresponding to a candidate root.

Initialization: The initial population of size N is generated by sampling from a uniform distribution $U(min, max)$. While these bounds can be user-defined, `PolySolve` defaults to using **Cauchy's bound** [5] to automatically determine the search interval. This ensures that the initialization space theoretically contains all real roots of the polynomial.

Fitness Function: The core of a GA is the fitness function. We define the fitness $F(x)$ of an individual x as:

$$F(x) = \frac{1}{|P(x)| + \epsilon} \tag{1}$$

where ϵ is a small constant to prevent division by zero. This function creates a fitness landscape with sharp peaks at the roots. As $P(x)$ approaches 0, $F(x)$ approaches infinity, creating strong selection pressure towards the roots.

Selection (Elitism): We utilize a rank-based selection mechanism. In each generation, the population is sorted by fitness. The top k individuals are preserved explicitly (Elitism) to ensure the best solution found so far is never lost. Simultaneously, a larger percentile of high-fitness individuals is designated as the "parent pool" to generate offspring for the next generation.

Reproduction and Mutation: Unlike traditional binary GAs, we employ a hybrid reproduction strategy tailored for continuous optimization. The next generation is constructed via three distinct operations:

1. **Blend Crossover (BLX-α):** Two parents p_1, p_2 are drawn from the parent pool. A child is sampled from the interval $[\min(p_1, p_2) - I \cdot \alpha, \max(p_1, p_2) + I \cdot \alpha]$, where $I = |p_1 - p_2|$. This allows the algorithm to explore the continuum between parents while expanding slightly into the immediate neighborhood.

2. **Multiplicative Mutation:** Selected individuals undergo multiplicative mutation to maintain relative precision across scales:
$$x_{new} = x_{old} \cdot (1 + \delta) \tag{2}$$
where $\delta \sim U(-m, m)$ and m is the mutation strength. Unlike additive noise, this scales with the magnitude of the root.

3. **Random Immigration:** To prevent premature convergence to local minima, a fixed fraction of the population is replaced in every generation with new random individuals sampled from the global search space.

Convergence: The algorithm terminates when the maximum number of generations is reached.

4 Implementation

The primary bottleneck in the methodology described above is the evaluation of $P(x)$ for thousands of individuals over thousands of generations. To address this, we implemented two hardware-accelerated backends.

4.1 CPU Acceleration (Numba)

We utilize the **Numba** library [6] to Just-In-Time (JIT) compile the critical fitness loop. The python function is decorated with `@numba.jit(nopython=True, fastmath=True, parallel=True)`. This allows the fitness evaluation of the population to be distributed across available CPU cores, bypassing the Python Global Interpreter Lock (GIL).

4.2 GPU Acceleration (CUDA)

For massive-scale parallelism, we implemented a custom CUDA kernel using **CuPy** [7]. The fitness kernel is written in C++ and compiled at runtime.

The kernel assigns one GPU thread to each individual in the population. The polynomial coefficients are stored in constant memory to minimize latency. The kernel computes the polynomial value using Horner's method (or direct expansion) and writes the fitness rank back to global memory.

```
// Simplified CUDA Kernel Logic
extern "C" __global__ void fitness_kernel(..., double* x_vals, double* ranks, ...) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < size) {
        // Evaluate polynomial P(x)
        double ans = ...;
        // Compute Fitness
        ranks[idx] = (ans == 0) ? MAX_DOUBLE : fabs(1.0 / ans);
    }
}
```

Listing 1: Simplified CUDA Kernel Logic

This approach allows us to scale the population size to tens of thousands with negligible impact on iteration time, as the GPU can evaluate the entire population in parallel.

5 Experimental Setup and Results

To evaluate the performance and accuracy of `PolySolve`, we conducted a series of benchmarks comparing it against `numpy.roots`.

Hardware Environment: Benchmarks were conducted on a desktop workstation with the following specifications:

- **CPU:** AMD Ryzen 5 7600x (6 Cores, 12 Threads)
- **GPU:** Nvidia RTX 4060 Ti (8GB VRAM)
- **RAM:** 32GB DDR5 5600MHz

Benchmark Methodology: We generated random polynomials of increasing degrees $d \in \{3, 5, 10, \ldots, 100\}$. For each degree, we measured:

1. **Execution Time:** Wall-clock time to find roots.
2. **Mean Absolute Error (MAE):** The average absolute value of the polynomial evaluated at the computed roots ($\frac{1}{k} \sum |P(x_{found})|$).

5.1 Computational Performance (Time)

As expected, the deterministic `numpy.roots` is significantly faster than the iterative GA. However, the CUDA implementation provides a substantial speedup over the CPU-based GA.

Table 1: Comparison of Execution Time

Degree	NumPy (CPU) [s]	PolySolve (CPU) [s]	PolySolve (GPU) [s]
3	0.0001	10.37	0.58
40	0.0002	11.67	0.63
100	0.0026	12.27	0.71

The GPU implementation achieves a speedup of approximately **20x** over the Numba-accelerated CPU version. Notably, the GPU time remains nearly constant regardless of degree, as the massive parallelism hides the slight increase in arithmetic intensity.

5.2 Numerical Accuracy

The accuracy results reveal the primary contribution of this work.

Table 2: Comparison of Accuracy (Mean Absolute Error)

Degree	NumPy MAE	PolySolve (CPU) MAE	PolySolve (GPU) MAE
3	1.24×10^{-15}	2.89×10^{-16}	2.89×10^{-16}
40	$2.96 \times 10^{+35}$	2.40×10^{-16}	2.40×10^{-16}
80	$2.49 \times 10^{+44}$	1.72×10^{-16}	1.72×10^{-16}
100	$3.12 \times 10^{+117}$	1.36×10^{-16}	1.36×10^{-16}

At degree 3, both methods are accurate. However, as degree increases, NumPy's accuracy degrades exponentially. At degree 100, NumPy has an error of 10^{+117}, while `PolySolve` maintains double-precision accuracy (10^{-16}).

6 Discussion

The results illuminate a clear trade-off between computational speed and numerical stability.

The Stability of Heuristics: The accuracy degradation in `numpy.roots` is a well-documented consequence of the companion matrix method. As the degree increases, the matrix eigenvalues become highly sensitive to numerical noise. In contrast, `PolySolve` re-evaluates the polynomial $P(x)$ explicitly at every generation. The fitness function $1/|P(x)|$ acts as a "ground truth" that does not degrade with degree. As long as the GA converges, it converges to a highly precise solution.

The Cost of Precision: This stability comes at a cost. The GA is orders of magnitude slower than the eigenvalue method. However, the introduction of GPU acceleration mitigates this significantly. By reducing the runtime from ≈ 12 seconds (CPU) to ≈ 0.6 seconds (GPU), the method moves from "prohibitively slow" to "acceptable for high-precision applications."

Limitations: Currently, `PolySolve` is configured to search for real roots within a bounded interval. Unlike `numpy.roots`, it does not automatically find complex roots, nor does it guarantee finding *all* roots in a single pass without careful parameter tuning (population size, mutation rate).

7 Conclusion

This paper presented `PolySolve`, a Python library that applies GPU-accelerated Genetic Algorithms to the problem of polynomial root-finding. We demonstrated that while traditional deterministic solvers are faster, they lack the numerical stability required for high-degree polynomials (e.g., $n = 100$). Our GA approach, while computationally heavier, maintains near-perfect double-precision accuracy across all tested degrees.

Future work will focus on extending the search space to the complex plane and implementing polynomial deflation techniques to efficiently locate all roots of a polynomial sequentially.

References

[1] R. A. Horn and C. R. Johnson, *Matrix Analysis*. Cambridge University Press, 1985.

[2] J. H. Wilkinson, ”The evaluation of the zeros of ill-conditioned polynomials,” *Numerische Mathematik*, vol. 1, no. 1, pp. 150-166, 1959.

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[4] H. Bhasin and S. Bhatia, ”Use of Genetic Algorithms for Finding Roots of Algebraic Equations,” *International Journal of Computer Science and Information Technologies*, vol. 2, no. 4, pp. 1693-1696, 2011.

[5] A. L. Cauchy, ”Sur la résolution des équations numériques et sur la théorie de l'élimination,” *Exercises de Mathématiques*, vol. 4, pp. 65-128, 1829.

[6] S. K. Lam, A. Pitrou, and S. Seibert, ”Numba: A LLVM-based Python JIT compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1-6, 2015.

[7] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, ”CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations,” in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.