CAUCHY – Jurnal Matematika Murni dan Aplikasi
Volume 10 (2) (2025), Pages 1324-1348
p-ISSN: 2086-0382; e-ISSN: 2477-3344

Modified of Roots Finding Algorithm of High Degree Polynomials

Bandung Arry Sanjoyo*, Mahmud Yunus, and Nurul Hidayat

Department of Mathematics, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia

Abstract

Although the Durand-Kerner method is widely used across various fields of computer science, especially in numerical computing, it continues to encounter challenges in locating roots of high-degree polynomials, such as issues with accuracies of roots of the polynomial zeros. Our initial tests and observations on several methods for finding polynomial roots revealed that the roots' accuracy starts to degrade noticeably for polynomials where the degree exceeds 10. Based on considerations of algebraic concepts involving polynomial vector spaces, we introduce an improvement of the Durand-Kerner algorithm aimed at improving root precision. This approach includes targeted refinements in coefficient evaluation, identification of root types, and iterative polishing techniques. We also conducted a comparative evaluation to assess its effectiveness against the original Durand-Kerner method and MATLAB's roots() function. Overall, the enhanced algorithm delivers superior accuracy for complex roots—particularly in cases involving multiple zero or integer roots—outperforming both benchmarks, but its execution time increases substantially with polynomial degree.

Keywords: high-degree polynomial zeros, Durand-Kerner algorithm, polynomial root-finding, numerical stability.

1 Introduction

Despite being an old problem, the computation of the roots of large-sized high-degree polynomials has become one of the most important contemporary issues in science [1], [2], [3].One of the most popular methods for this purpose is the Durand-Kerner (DK) method, also known as Weierstrass iteration, which refines approximations to all roots at once. It enjoys particular popularity due to its conceptual simplicity and natural parallelism. Yet, the method is not without drawbacks: it might not converge at all or produce unusable results in particular for large degree polynomials [4], [5], [6], [7], [8].

As the demand for computational precision and efficiency continues to grow, the development of advanced root-finding algorithms has become increasingly critical for progress in scientific computing and engineering applications [1], [9], [10], [11], [12]. Over the past several decades, the pursuit of accurate and scalable methods for solving high-degree polynomial equations has remained a central focus in numerical analysis [3], [7], [13], [14], [15]. High-degree polynomials present substantial computational challenges due to the lack of general analytical solutions, necessitating the use of iterative numerical techniques with controllable error bounds.

*Corresponding author. E-mail: bandung@its.ac.id

Submitted: November 2, 2025 Reviewed: November 9, 2025 Accepted: November 25, 2025
DOI: https://doi.org/10.18860/cauchy.v10i2.37278

Several researchers have improved the DK algorithm by assigning initial values. These initial values significantly impact stability, convergence, and the number of iterations required to obtain the roots [4], [6], [16], [17], [18]. From a computational strategy perspective, to achieve stability in root finding, initial values can be combined with other initialization techniques. The lambda maximal bound, derived from the dominant eigenvalue of the companion matrix, consistently guarantees that all roots lie within the complex circle and supports fast and stable convergence [19]. On the other hand, iterative methods like Newton-Raphson and Laguerre are used to approximate improve accuracy of a single root. They are usually used in conjunction with polynomial deflation to successively find all roots of a given polynomial [20], [21]. However, with high-degree polynomials and polynomials which have complex roots, algorithms using the Newton–Raphson and Laguerre methods are also numerically unstable and inaccurate when multiple or closely spaced roots must be found. These difficulties underline the necessity for stronger and globally convergent algorithms to cover complexities of high-degree polynomials [22], [23].

The DK method for multiple roots suffers from convergence failures, including a decrease in the accuracy of real roots due to iteration cycles and floating-point instability at degree $n > 10$. Modifications for special cases are suggested, with evidence that the basic version is globally unstable [8]. The Weierstrass method on multiple roots, where the accuracy of real root approximation fails due to error magnification from simultaneous iterations, leads to deviations of up to 10^{-3} in floating-point double precision for high degrees. Convergence drops to linear (from quadratic), especially when the roots are close together ($< 10^{-6}$ distance), and floating-point errors worsen this at $n > 15$ [24], [25].

This study presents to improve numerical precision in resolving high-degree polynomial problems base on an adaptation of the Durand-Kerner technique. To achieve this objective, we classify polynomials based on the coefficients of the polynomial and their roots, examine foundational theorems relevant to root-finding techniques, and develop a computational algorithm inspired by the Durand-Kerner method. The rest of the paper is structured as follows: Section 2 presents the methodology used in this research. Section 3 presents the results and discussion. Finally, Section 4 concludes the study and outlines potential directions for future research.

2 Methods

To achieve the objectives of this research, the methodology consists of the following steps: examining the problem of polynomial root-finding, exploring fundamental theories related to root-finding techniques, designing the proposed algorithm, conducting evaluation and testing of the algorithm, and performing result analysis.

2.1 Polynomial zeros and the fundamental theories

Definition 1. *A function $p : F \to F$ is called a polynomial of degree n if there exist $a_1, a_2, ..., a_{n+1} \in F$ with $a_1 \neq 0$ such that*

$$p(x) = a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \cdots + a_nx + a_{n+1} \tag{1}$$

for all $x \in F$. Symbol F denote $\mathbb{R}$ or $\mathbb{C}$. In case $a_1 = 1$, the polynomial is called monic polynomial. The expression $p(x) = 0$ is called polynomial equation, and a number $z \in \mathbb{C}$ is called a zeros or root of a polynomial p if $p(z) = 0$ [2], [3], [26]. When coefficients $a_{k+2}, a_{k+3}, ..., a_{n+1}$ are zero for $k \geq 1$, the polynomial has k zero roots, i.e., $z_1 = z_2 = \cdots = z_k = 0$ and $n - k$ non zero roots $z_{k+1}, z_{k+2}, ..., z_n$, which are the roots of the deflated polynomial.

$$q(x) = x^{n-k} + a_2x^{n-k-1} + a_3x^{n-k-2} + \cdots + a_{n-k+1} \tag{2}$$

Theorem 1 (Fundamental theorem of algebra)**.** *If $p(x)$ is a monic polynomial of degree n, then the equation $p(x) = 0$ has at least one root, z, and the polynomial can be expressed as shown in Equation* (3)*.*

$$p(x) = (x - z)q(x) \tag{3}$$

where $q(x)$ is a polynomial of degree $n - 1$ [26].

Based on Equation (3), the polynomial $p(x)$ can also be expressed as in Equation (4).

$$p(x) = (x - z_1)(x - z_2) \cdots (x - z_n) \tag{4}$$

where $z_1, z_2, ..., z_n$ are the roots of the polynomial $p(x)$. Therefore, a polynomial $p(x)$ of degree n has n roots, which may be real and distinct, real and repeated, complex conjugate pairs, or a combination of these types.

Theorem 2 (Division algorithm for polynomials)**.** *Let $p(x)$ be a monic polynomial of degree n, and let $s(x)$ be a non-zero polynomial. Then, there exists unique polynomials $q(x)$ and $r(x)$ such that*

$$p(x) = s(x)q(x) + r(x) \tag{5}$$

where the degree of $r(x)$ is less than the degree of $s(x)$ [2], [26].

Let $p(x)$ be a monic polynomial of degree n with all coefficients of p in $\mathbb{R}$. Then $p(x)$ has unique factorization of the form given in Equation (6).

$$p(x) = (x - z_1) \cdots (x - z_m) \cdots (x^2 + b_1 x + c_1) \cdots (x^2 + b_k x + c_k) \tag{6}$$

where $z_1, ..., z_m, b_1, ..., b_k, c_1, ..., c_k \in \mathbb{R}$.

2.2 Durand-Kerner Algorithm

The general form of a monic polynomial's roots, as expressed in Equation (3), can be computed using the following iterative formula [1], [4], [9]:

$$z_i^{(k+1)} = z_i^k - \frac{p\left(z_i^k\right)}{\prod\limits_{\substack{j=1 \\ j \neq i}}^{n} \left(z_i^k - z_j^k\right)} \tag{7}$$

where z_i^k denotes the i-th root approximation at the k-th iteration. The Durand-Kerner algorithm is an iterative method based on Equation (7) that simultaneously computes all complex roots $z_i \in \mathbb{C}$, where $i = 1, 2, ..., n$.

The convergence of this method is highly sensitive to the choice of initial approximations z_i^0 [1], [9], [27]. Therefore, selecting initial guesses that are sufficiently close to the actual roots is critical for the algorithm's success. The iterative process continues until the approximations z_i^k converge to the actual roots. The overall steps of the root-finding procedure are outlined in Algorithm 1.

Given the initial value z_i^0, they should be sufficiently close enough to the roots z_i [28]. The initialization of z_i^0 is commonly performed using Cauchy's bound, as expressed in Equation (8) [16].

$$z_i^0 = re^{\theta i} \quad \text{where } r = 1 + \max_{2 \leq i \leq n+1} |a_i| \tag{8}$$

Algorithm 1 Durand-Kerner algorithm

Input: *coeffs* = [1, a_2, a_3, ..., a_{n+1}] which is the coefficient of $p(x)$.
Output: $z = [z_1, z_2, ..., z_n]$ which is the roots of $p(x) = 0$.
Algorithm:
1. Set initial value z_i^0 for $i = 1, 2, ..., n$.
2. Compute next $z_i^{(k+1)} = z_i^k - \dfrac{p(z_i^k)}{\prod_{\substack{j=1 \\ j \neq i}}^{n} (z_i^k - z_j^k)}$ for $i = 1, 2, ..., n$.
3. Repeat step 2. until $z_i^{(k+1)}$ closed to z_i^k or $p(z_i^{(k+1)})$ closed to zero.

The Algorithm 1 is considered to have converged when one of the following conditions is satisfied:
(i) the difference between successive approximations $|z_i^{k+1} - z_i^k|$ is less than a predefined tolerance ϵ_1;
(ii) the polynomial evaluation $|p(z_i^{k+1})|$ is less than ϵ_2, or
(iii) the number of iterations exceeds a specified maximum threshold.

The computational complexity of the Durand-Kerner method is $O(kn^2)$ where k is the number of iterations and n is the degree of the polynomial [29]. This complexity arises primarily from steps 2 and 3 of the algorithms.

2.3 Modification and analysis algorithm for polynomial zeros

The input and output design of the proposed method adhere to the same structure as the classical Durand-Kerner algorithm and MATLAB's roots() function. However, the key modifications are introduced in the initialization of root estimates values and the iterative root-finding process. The initial approximation z_i^0 are distributed along a circular boundary in the complex plane, where the radius is determined by the maximum modulus of the polynomial's roots. This boundary is referred to as the lambda maximal bound (λ_{max}) [19].

Here, λ_{max} denotes the spectral radius of the companion matrix C with associated the polynomial and is computed using power method defined as follows:

$$\lambda_{max} = \frac{x_k^T x_{k+1}}{x_k^T x_k} \tag{9}$$

where $x_0 \in \mathbb{R}^{n \times 1}$, and $x_k = C^k x_0$. The computation λ_{max} requires non-constant extra space and has a computational complexity of $O(n^2)$ flops [19], [30]. The radius of the circular boundary of the complex plane is set to $r = \lambda_{max} + \epsilon$ where $\epsilon > 0$ is a small margin ensuring that all roots lie within the circle. The initial points are then assigned as $z_i^0 = re^{i\theta_k}$, for $k = 1, 2, ..., n$ with θ_k uniformly spaced angle on $[0, 2\pi)$.

In comparison with Cauchy's bound, the lambda maximal bound yields the smallest radius that still enclosed all the roots [19].The choice of this boundary based on the lambda maximal value follows the recommendation by Kjellberg (1984), who stated that the initial values z_i^0 should lie within or on a circle in the complex plane and be sufficiently close to the actual roots z_i. This approach ensures better convergence behavior and numerical stability in the root-finding process [28]. Furthermore, the root-finding process incorporates classification mechanism that analyses the characteristics of the polynomial's coefficients and degree. These classifications include integer and non-integer coefficients, as well as the nature of the roots—whether they are integer, real, or complex values.

For polynomials of degree $n \leq 2$, the roots can be obtained with high precision and robustness. Specifically, for degree $n = 1$, the root is given by the formula $z = -a$, which avoids issues related

to numerical rounding or near-zero denominators. For degree $n = 2$, the roots are analytically derived using Equation (10):

$$z_1 = \frac{-a(2) + \sqrt{a(2)^2 - 4a(3)}}{2} \quad \text{dan} \quad z_2 = \frac{-a(2) - \sqrt{a(2)^2 - 4a(3)}}{2} \tag{10}$$

Equation (10) involves only basic arithmetic operations and a square root, which are well-conditioned for most input ranges. Moreover, the denominator is a constant (2), eliminating the risk of division by very small numbers that could amplify numerical errors. Consequently, the formula provides numerical stability and robustness in computing the roots of quadratic polynomials. For polynomials of degree $n > 2$, the algorithm is designed by analyzing the characteristics of the polynomial coefficients, rather than directly applying Equation (7) as done in the original Durand-Kerner algorithm.

The modified algorithm is analytically evaluated by measuring its computational workload and memory usage. The computational workload includes number of arithmetic operations and iteration count. This analysis provides insight into the efficiency and scalability of the proposed method compared to Durand-Kerner (DK) Algorithm.

2.4 Evaluation and testing of the algorithm

The algorithm was implemented in MATLAB and evaluated using a diverse set of polynomial equations with degrees starting from $n = 1$, under the constraint that the maximum coefficient value does not exceed the *maxflint* (maximum floating-point integer) threshold. This constraint ensures numerical stability during computation.

Performance metrics used in the evaluation include convergence rate, root estimates accuracy, and computational time. The experimental evaluation involved classifying the input polynomials into several categories:

(i) polynomials with integer coefficients,
(ii) polynomials with real (non-integer) coefficients,
(iii) polynomials with integer roots,
(iv) polynomials with clustered roots and ill-conditioned polynomials.

The accuracy and performance of the proposed root-finding method were compared against those of the Durand-Kerner algorithm and MATLAB's roots() function.

3 Results and Discussion

3.1 The Modified Algorithms

To improve the roots of the polynomial, the modified algorithms accommodate specific characteristics of polynomials:

i. If the coefficients $a_i = 0$, for $i = 2, 3, ..., n + 1$, the roots of the polynomial $z_i = 0$ for $i = 1, 2, ..., n$.
ii. If the coefficients are not all zero and $a_{n+1} = 0$, then one of the roots of the polynomial is necessarily zero and the polynomial effectively reduce to a polynomial of degree $n - 1$.
iii. If all coefficients a_i are integers, then some of the roots can be determined by examining the factors of the constant term a_{n+1}. Subsequently, the polynomial function is reduced using Equation (5).
iv. If all coefficients a_i are integers and the polynomial does not have any root that is a factor of the constant term a_{n+1}, then a quadratic factor of the form $x^2 + bx + c$ is sought, where b is an integer and c is factor of a_{n+1}. Subsequently, the polynomial function is reduced using Equation (6).

v. When the conditions outlined in points (i) to (iv) are not satisfied, the root-finding process is executed using the Durand-Kerner method with initial values placed on boundary of a circle whose radius corresponds to the lambda maximum bound [19].This initialization strategy is followed by a refinement step to improve designed to enhance the accuracy of the computed roots.

The Modified DK first eliminates the zero roots and then deflates the polynomial to a lower degree. For the test case, we use input polynomials of the form

$$p(x) = x^n + a_2x^{n-1} + a_3x^{n-2} + \cdots + a_{k+1}x^{n-k} \tag{11}$$

where $k \geq 1$. The polynomial has k zero roots, i.e., $z_1 = z_2 = \cdots = z_k = 0$ and $n - k$ nonzero roots $z_{k+1}, z_{k+2}, ..., z_n$, which are the roots of the deflated polynomial

$$q(x) = x^{n-k} + a_2x^{n-k-1} + a_3x^{n-k-2} + \cdots + a_{n-k+1}. \tag{12}$$

The overall steps of the newly proposed root-finding procedure, referred to as *the modified DK algorithm*, are summarized in Algorithm 2.

Algorithm 2 Proposed algorithm for polynomial zeros

Input: *coeffs* = [1, a_2, a_3, ..., a_{n+1}] which is the coefficient of $p(x)$.
Output: *roots* = [z_1, z_2, ..., z_n] which is the roots where $p(roots) = 0$.
Algorithm modifiedDK:
function roots = modifiedDK(coeffs)
1. [roots newCoeffs] = Calculate_zero_roots(coeffs);
   - Using Equation (2).
   - roots = [0 0 ... 0]; k = length(roots);
   - newCoeffs = coeffs(1:n-k); n = n-k;
2. if n = 1, then z = a_2.
3. if n = 2, then the roots calculated by Equation (9).
4. if (all coefficients a_i are integers and has linear or quadratic factors which integer coefficients)
   - Solve the polynomial using Equation (6).
5. SolvePolyInteger(newCoeffs);
   else
6. Solve polynomial newCoeffs using DK algorithm which employs initialization using lambda maximum bound.
end

The Algorithm 3 found an integer root z_i if $p(z_i)$ is less than ϵ_2. In Algorithm 3, the step *Find linear factor* of the coefficients a_{n+1} is performed first to detect possible integer or rational roots. When trial division is employed, the computational cost is approximately $O(\sqrt{|a_{n+1}|})$, which is generally negligible for moderate-sized coefficients, although it may become expensive for very large integers. The term $|\sqrt{a_{n+1}}|$ represents the upper bound on the number of candidates roots. Testing each candidate requires evaluating the polynomial, which incurs a cost of $O(n)$ per candidate. Since there are $l\sqrt{|a_{n+1}|}$ candidates to test, the total cost becomes $O(l\sqrt{|a_{n+1}|} \cdot n)$. The step deflation of the polynomial, deconv$(coeffs, poly(roots))$, using synthetic division after finding a root requires $O(n)$ operations per step, leading to a cumulative cost of $O(kn)$ if k roots are detected through factorization.

Algorithm 3 Algorithm for finding the zeros of polynomials with integer coefficients.

Input: *coeffs* = [1, a_2, a_3, ..., a_{n+1}] which is the integer coefficient of $p(x)$.
Output: *roots* = [z_1, z_2, ..., z_n] which is the roots where $p(roots) = 0$.
Algorithm SolvePolyInteger:
function roots = SolvepolyInteger(coeffs)
1. n = length(coeffs)-1; roots = [];
2. Find linear factor
   - factor = findFactors(abs(coeffs(n+1)));
   - idx = find(abs(p(factor) <= ϵ_2);
   - roots = [roots factor(idx)];
   - newCoeffs = deconv(coeffs, poly(roots));
3. Find quadratic factors
   - cekFaktorQuadrat = true;
   - while length(sisaPoli) > 1 && cekFaktorQuadrat
   - [Q, sisaPoli] = factorizedIntoQuadratFactor(newCoeffs);
   - if ∼isempty(Q)
   - r = solvequadrat(Q);
   - else
   - cekFaktorQuadrat = false;
   - r = Solve polynomial newCoeffs using DK algorithm which employs initialization using lambda maximum bound;
   - end
   - root = [root r];
   - end
   - end

end

The step *factorizedIntoQuadratFactor(coeffs)* is finding a quadratic factor of polynomial with coefficient $coeffs$ expressed in algorithm 4. Algorithm 4 describes the procedure for identifying a quadratic factor of a given polynomial. The resulting factor takes the form $x^2 + bx + c$ where c is a divisor of a_{n+1} (the constant term), and b is an integer within the range $[-|a_2|, |a_2|]$, where $a_2 = coeffs(2)$. The factorization of the constant term a_{n+1} requires approximately $O(\sqrt{|a_{n+1}|})$ operations. Candidate value c are then generated, with the number of candidates estimated as $l\sqrt{|a_{n+1}|}$. For each candidate b, the evaluation of possible values of b involves at most $2|a_2| + 1$ iterations. Dividing a polynomial of degree n by a quadratic factor incurs a computational cost on the order of $O(n)$ operations. Consequently, the search for a quadratic factor requires $O(l\sqrt{|a_{n+1}|}(2|a_2| + 1)n)$, or equivalently $O(n|a_2|\sqrt{|a_{n+1}|})$ in worst-case.

Therefore, the overall worst-case complexity of *SolvePolyInteger* is $O(n|a_2|\sqrt{|a_{n+1}|})$. When the input polynomial solely of integer coefficients and contains both linear and quadratic factors, the algorithm exhibited near-linear scaling with respect to the polynomial degree. This behaviour is consistent with the theoretical bound $O(n|a_2|\sqrt{|a_{n+1}|})$ for moderate coefficient magnitudes. Finally, the computational complexity of the modified DK in Algorithm 2 is $O(\max\{n|a_2|\sqrt{|a_{n+1}|}, n^2\})$.

Algorithm 4 Algorithm for finding a quadratic factor of a polynomial

Input: *coeffs* = [1, a_2, a_3, ..., a_{n+1}]
Output: $Q = [1, b, c]$ and *remainderPoly*
function [Q, remainderPoly] = factorizedIntoQuadratFactor(coeffs)
remainderPoly := coeffs
n := length(remainderPoly) − 1
faktorC := findFactors(remainderPoly(end))
for each c in faktorC **do**
  M := |remainderPoly(2)|
  for $b = -M$ **to** M **do**
    Q := [1, b, c]
    [D_temp, $sisa$] := deconv(remainderPoly, Q)
    if all($|sisa| < 10^{-8}$) **and** all($|D\_temp - \text{round}(D\_temp)| < 10^{-8}$) **then**
      remainderPoly := round(D_temp)
      return
    end if
  end for
end for
Q := [ ]
remainderPoly := coeffs

3.2 Experimental Setup and Performance Analysis

To assess the performance of the proposed root-finding algorithm, we conducted a series of numerical experiments on various families of univariate complex polynomials with degrees ranging from 2 to 30. The algorithm was implemented in MATLAB R2024b and executed on a desktop PC equipped with an Intel Core i7 processor, 16 GB RAM, 1.80 GHz clock speed, and 8 Logical Processor(s) running the Microsoft Windows 10 operating system.

For each polynomial type, we recorded both the accuracy of the computed roots and the total computational time. Accuracy was evaluated using polynomials with known exact roots or by measuring the residual norm, defined as $|p(x_k)|$, where x_k denotes the computed root.

The first experiment involves running the algorithm's program on input polynomials with real coefficients that have k zero roots. Figure 1 and Figure 2 present the performance of three root-finding — DK algorithm, Modified DK algorithm, and MATLAB's roots() function — in terms of mean error, maximum error, and runtime when applied to polynomials of degree 25 with k zero roots, where $k = 2, ..., 23$. The mean error (e_{mean}) refers to the average of the mean errors computed for all roots across each experiment for different values of k. The maximum error (e_{max}) refers to the largest maximum error observed across all experiments.

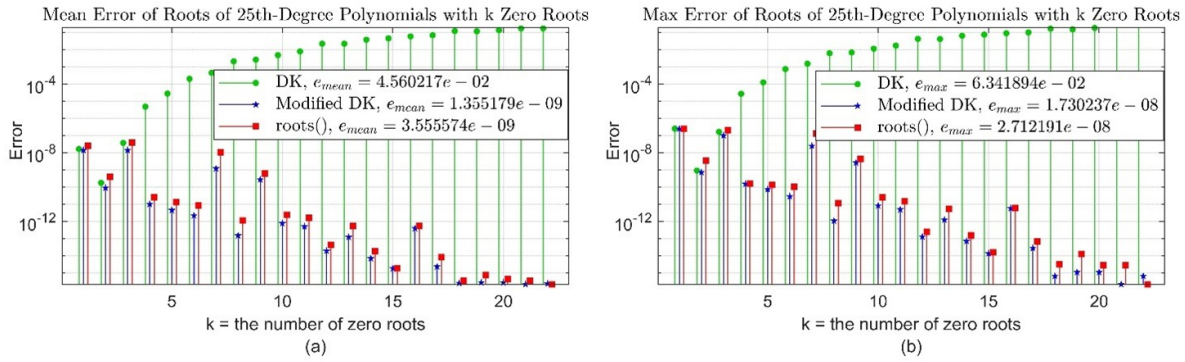

Figure 1: The comparisons of mean error of roots of polynomials of degree 25 with real coefficients and k zero roots

The graph in Figure 1a illustrates the mean error of the computed roots for each method. The DK algorithm yields the highest mean error, approximately 4.56×10^{-2}, indicating lower numerical accuracy. In contrast, the Modified DK algorithm and MATLAB's roots() function produce significantly lower mean errors, around 1.36×10^{-9} and 3.56×10^{-9}, respectively. These results suggest that the Modified DK algorithm, which simplifies the polynomial by first identifying zero roots and then reducing its degree, improves the accuracy of the computed non-zero roots. Figure 1b presents the maximum error observed among the computed roots. The Modified DK algorithm again demonstrates better precision than DK algorithm, with a maximum error of approximately 1.73×10^{-8}. In comparison, the MATLAB's roots() function shows a similar level of maximum errors, both in order of 10^{-8}. These findings reinforce the conclusion that the Modified DK algorithm is more robust in preserving root accuracy, particularly for polynomials with multiple zero roots.

Figure 2 illustrates the computational time required by each method. The MATLAB's roots function achieves the fastest execution time, with a maximum runtime of approximately 1.56×10^{-4} seconds. This is significantly faster than both the DK and Modified DK algorithm, which have mean runtimes of 3.54×10^{-2} and 3.68×10^{-2} seconds, respectively. Although Modified DK algorithm is more accurate in computing roots (as shown in Figures 1a and 1b), it requires more computational time.

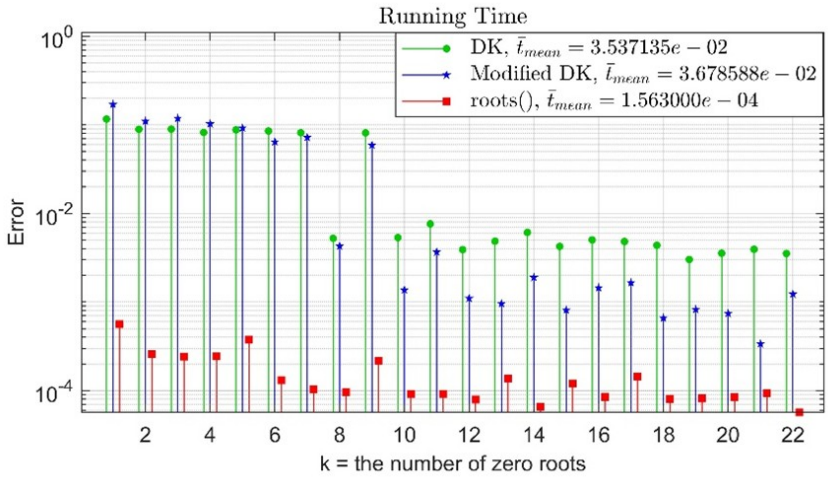

Figure 2: The comparisons of runtimes for polynomials of degree 25 with real coefficients and k zero roots

Testing on first-degree polynomials was not conducted because the roots of such polynomials can be obtained directly without involving floating-point operations that may lead to numerical errors. Testing on second-degree polynomials was performed on 500 polynomials with integer, real, and complex coefficients, for which the exact (true) roots are known. Additional experiments were also carried out on polynomials with randomly generated real coefficients, where the exact roots are unknown.

Figure 3 presents a comparison of the mean error for each root of second-degree polynomials with integer coefficients and integer roots, evaluated using the DK algorithm, a Modified DK algorithm, and MATLAB's roots() function over 300 trials. Both the DK algorithm and MATLAB's roots() function yielded average mean and maximum errors on the order of 10^{-9}, indicating significant numerical inaccuracies. In contrast, the Modified DK algorithm consistently produced a mean and maximum error of zero, demonstrating superior numerical stability and accuracy. Figure 4 illustrates the comparative computational time for root-finding in quadratic polynomials with integer coefficients and integer roots. The Modified DK algorithm achieved a performance improvement of approximately 7.5 times faster than DK algorithm and MATLAB's roots() function.

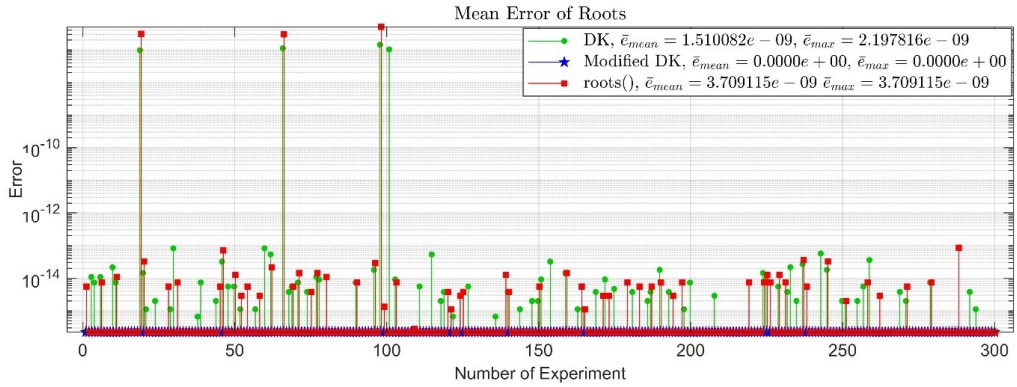

Figure 3: The comparisons of mean error of each root of second-degree polynomials with integer coefficients and integer roots.

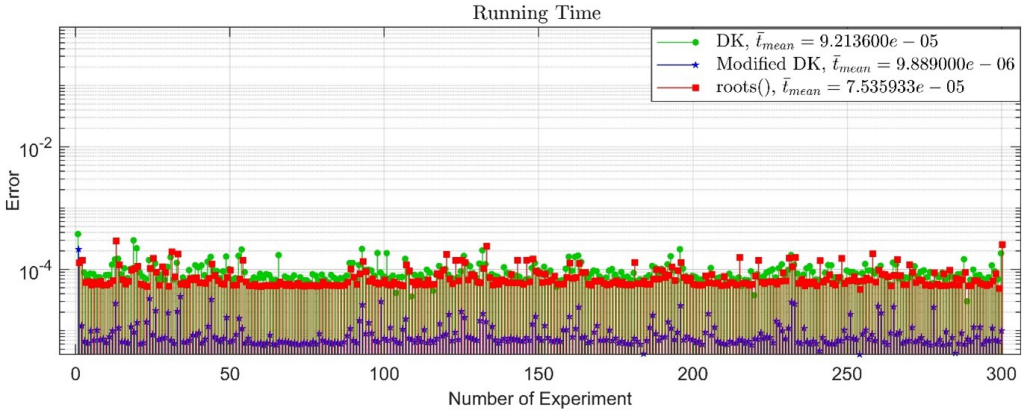

Figure 4: Comparison of running times of second-degree polynomials with integer coefficients and integer roots.

Figure 5 presents a comparison of the mean error for each root of second-degree polynomials with real coefficients and real roots, evaluated using the DK algorithm, a Modified DK algorithm, and MATLAB's roots() function over 200 trials. All three methods produced average mean and maximum errors on the order of 10^{-16}, indicating significant numerical inaccuracies. Figure 6 illustrates the comparative computational time for root-finding in quadratic polynomials with real coefficients and real roots. The Modified DK algorithm achieved a performance improvement faster than both the classical DK algorithm and MATLAB's built-in roots function.

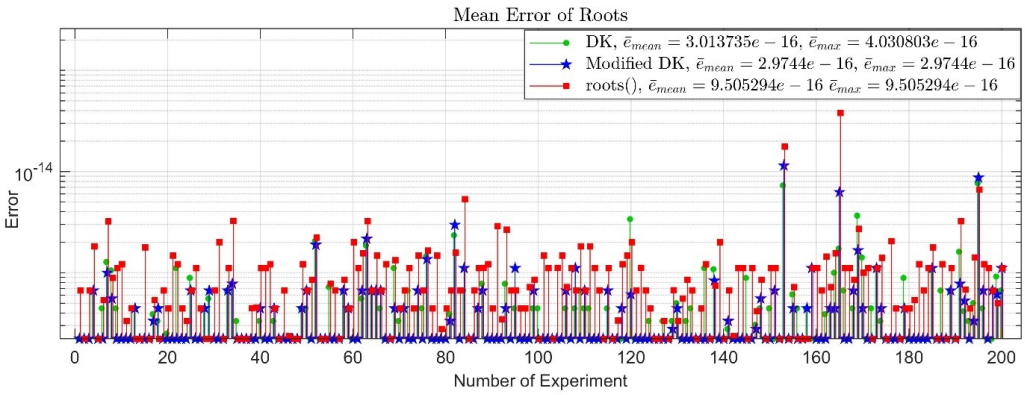

Figure 5: Comparisons of mean error of each root of second-degree polynomials with real coefficients and real roots.

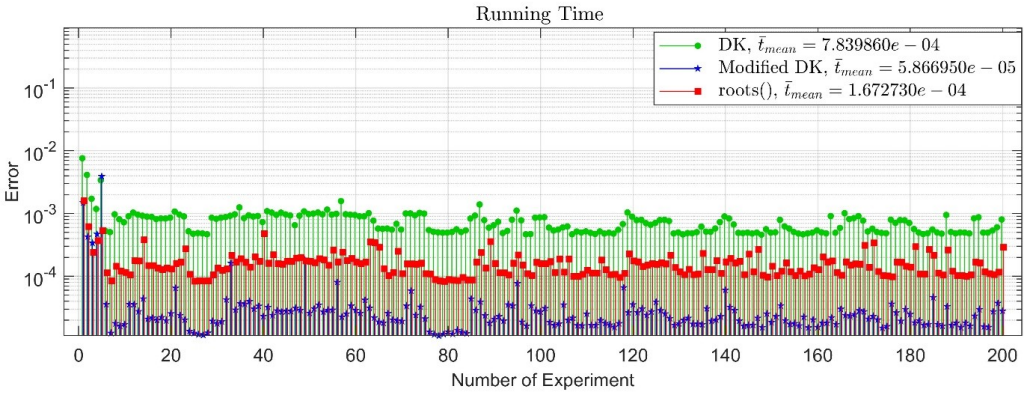

Figure 6: Comparison of running times of second-degree polynomials with real coefficients and real roots.

3.3 Comparative Analysis of Root-Finding Accuracy for High-Degree

The evaluation of high-degree polynomials was conducted on cases with integer, real, and complex roots, under the constraint that the magnitude of the coefficients does not exceed the maximum representable value within computer memory precision. Figures 7 through 9 present a comparative analysis of the mean error for three methods applied to polynomials with linear factors, integer coefficients, and integer roots, specifically for degrees 7, 17, and 29 with integer coefficients and integer roots. The primary metric for evaluation is the mean and maximum error of the computed roots across 30 independent trials.

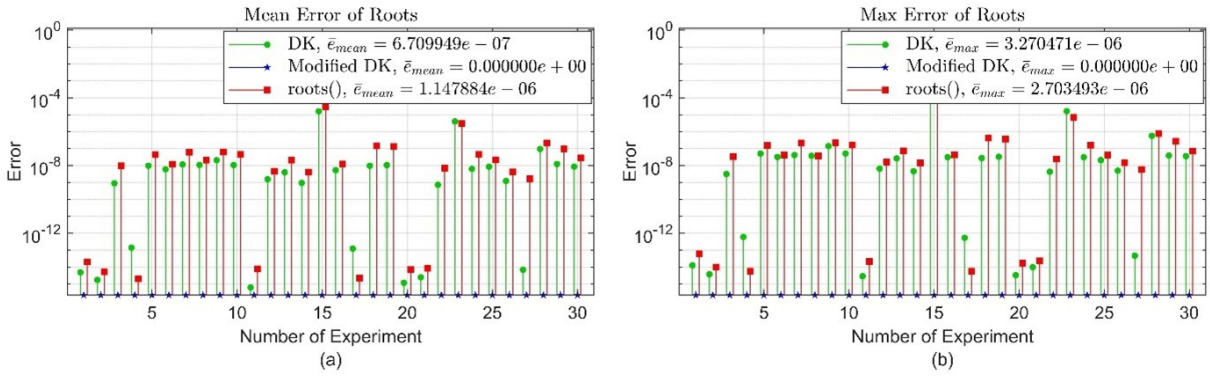

Figure 7: Comparisons of mean error of each root of polynomials degree 7 with integer coefficients and integer roots.

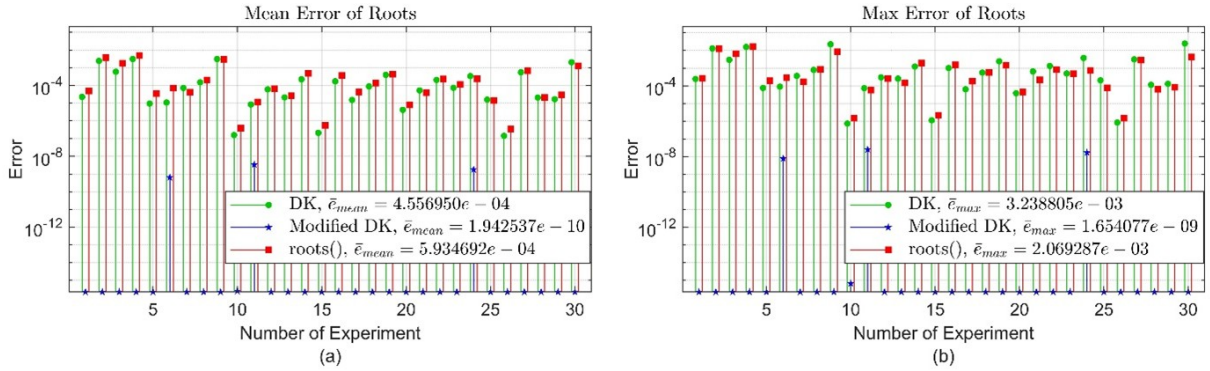

Figure 8: Comparisons of mean error of each root of polynomials degree 17 with integer coefficients and integer roots.

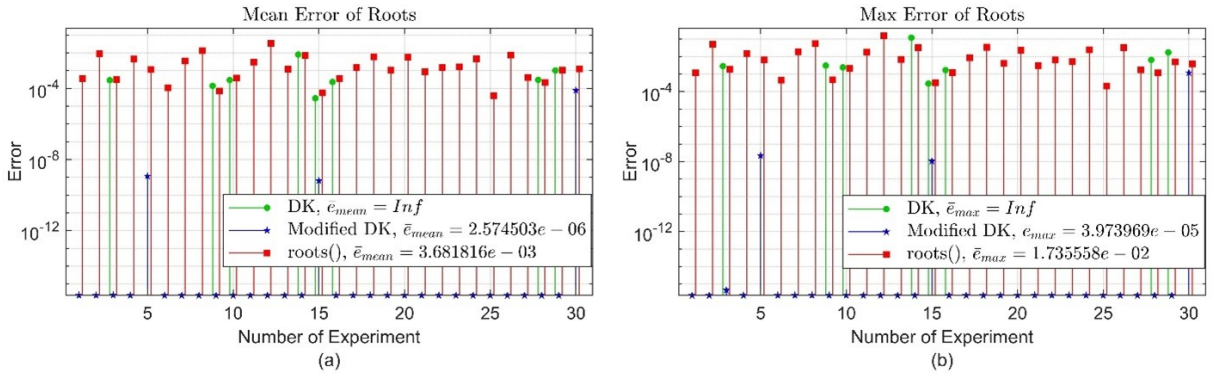

Figure 9: Comparisons of mean error of each root of polynomials degree 29 with integer coefficients and integer roots.

Figure 7 presents the error analysis for polynomials of degree 7. The Modified DK method achieves a mean error below computational threshold, indicating perfect recovery of all roots across all trials. The same result (mean error below computational threshold) also holds for polynomials of degree less than 7; however, those results are not shown here. In contrast, the DK method and MATLAB's roots() function yield mean errors of approximately 6.71×10^{-7} and 1.15×10^{-6}, respectively. The maximum error follows a similar trend, with Modified DK maintaining zero error, while DK and MATLAB's roots() reach up to 3.27×10^{-6} and 2.70×10^{-6}, respectively.

Figure 8 extends the analysis to polynomials of degree 17. The Modified DK method continues to outperform the other two, with a mean error of 1.94×10^{-10} and a maximum error of 1.65×10^{-9}. In comparison, the DK algorithm exhibits a mean error of 4.56×10^{-4} and a maximum error of 3.24×10^{-3}, while MATLAB's roots() shows a mean error of $5.93 \times 10^{-4}$and a maximum error of 2.07×10^{-3}. Figure 9, although not fully detailed here, is expected to follow the established trend. As polynomial degree increases to 29, the Modified DK algorithm is anticipated to maintain superior numerical stability and accuracy, while both DK algorithm and MATLAB's roots() are likely to suffer from increased error due to the accumulation of numerical instability inherent in high-degree polynomial root-finding.

The Modified DK algorithm demonstrates significantly improved accuracy and robustness in computing the roots of high-degree polynomials with integer coefficients and integer roots. Its performance remains consistent across varying degrees, outperforming both the classical DK algorithm and MATLAB's roots() function. These results suggest that the modifications to the DK algorithm effectively enhance its numerical stability, making it a reliable method for solving high-degree polynomials with integer-precision polynomial root finding.

To assess the computational performance especially on running time, experiments were conducted on the above polynomials of degrees 7, 17, and 29 with integer coefficients and integer roots. The average running time over 30 trials was recorded for each method and shown at Figure 10 through 12. For polynomials of degree 7, the MATLAB's roots() function demonstrated the fastest execution time with a mean of 1.79×10^{-4} seconds, followed by Modified DK algorithm at 2.47×10^{-3} seconds, and DK algorithm at 7.86×10^{-3} seconds. This indicates that Modified DK algorithm is approximately three times faster than DK algorithm, while MATLAB's roots() is significantly more efficient than both. At degree 17, the trend remains consistent. The MATLAB's roots() function maintained its superior speed with a mean running time of 1.53×10^{-4} seconds. Modified DK algorithm achieved a mean of 1.55×10^{-2} seconds, outperforming DK algorithm, which recorded 6.18×10^{-2} seconds. The Modified DK algorithm was roughly four times faster than DK algorithm, confirming its computational advantage at moderate polynomial degrees. However, for polynomials of degree 29, a notable deviation was observed. While MATLAB's roots() continued to exhibit the lowest running time (2.88×10^{-4} seconds), the Modified DK algorithm experienced a substantial increase in computational cost, with a mean running time

of 2.76 seconds—significantly higher than DK's 2.84×10^{-2} seconds. This suggests that the enhancements introduced in Modified DK algorithm, although beneficial for accuracy, may incur considerable overhead at higher degrees, potentially due to increased iteration complexity or convergence criteria.

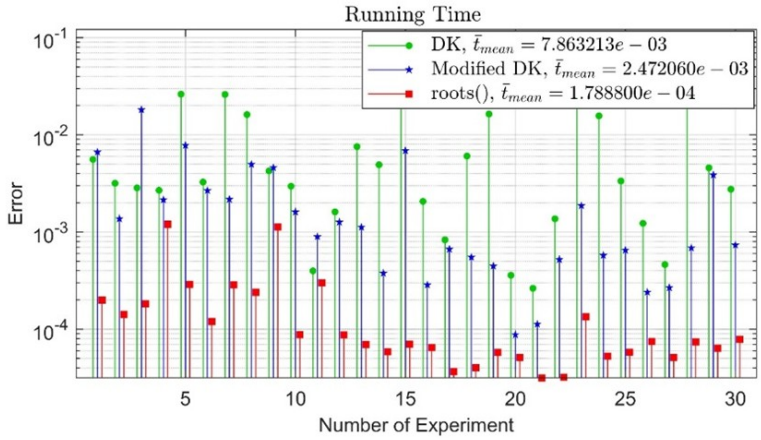

Figure 10: Comparison of running times of 7-th degree polynomials with integer coefficients and integer roots.

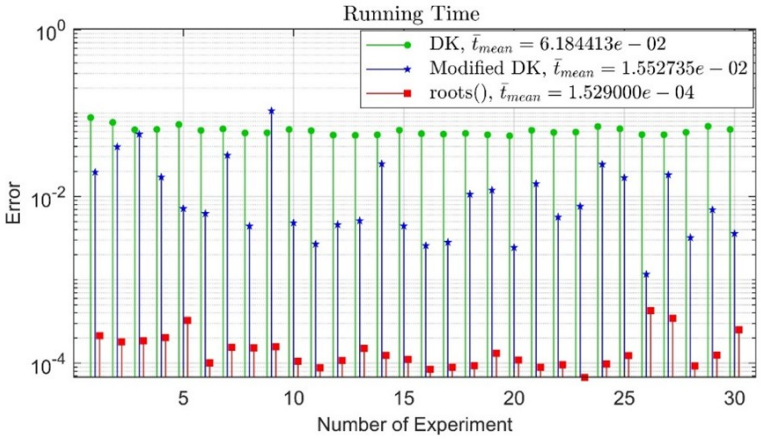

Figure 11: Comparison of running times of 17-th degree polynomials with integer coefficients and integer roots.

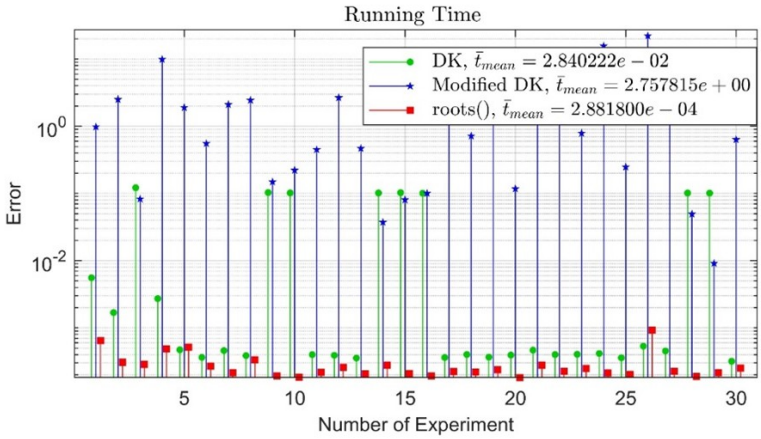

Figure 12: Comparison of running times of 29-th degree polynomials with integer coefficients and integer roots.

The running time evaluation reveals a clear trade-off between accuracy and computational efficiency. The Modified Durand-Kerner algorithm consistently outperforms the classical DK algorithm in both speed and precision for lower and moderate degrees. However, its scalability in terms of execution time becomes a limiting factor for high-degree polynomials. In contrast, MATLAB's roots() function remains the most efficient across all tested degrees, though it does not match the precision of Modified DK algorithm. These findings highlight the importance of selecting root-finding algorithms based on the specific requirements of accuracy and computational resources.

To assess the robustness and stability of root-finding methods, we include the Wilkinson polynomial as a critical benchmark due to its well-documented sensitivity to coefficient perturbations [31], [32].Figures 13(a) and 13(b) present the mean and maximum root errors, respectively, across varying polynomial degrees. The classical DK algorithm exhibits a rapid increase in both mean and maximum errors as the degree increases, indicating poor stability under the Wilkinson polynomial's challenging conditions. MATLAB's roots() function also shows increasing error, though it performs slightly better than DK algorithm in higher degrees. In contrast, the Modified DK algorithm demonstrates significantly improved accuracy. Both the mean and maximum errors remain consistently low across all tested degrees, suggesting that the modifications introduced in the algorithm effectively mitigate the instability typically associated with Wilkinson polynomials.

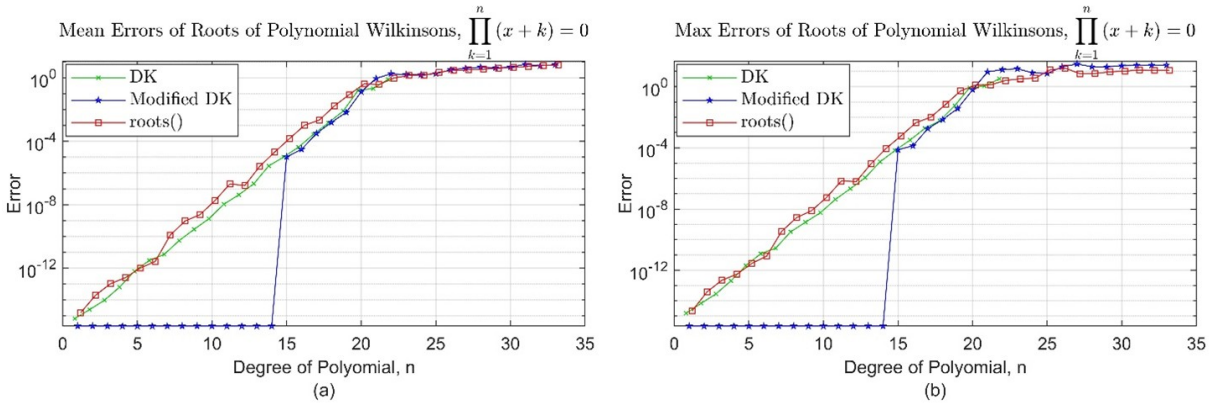

Figure 13: Comparisons of mean error of each root of Polynomial Wilkinsons for $n = 3 - 35$

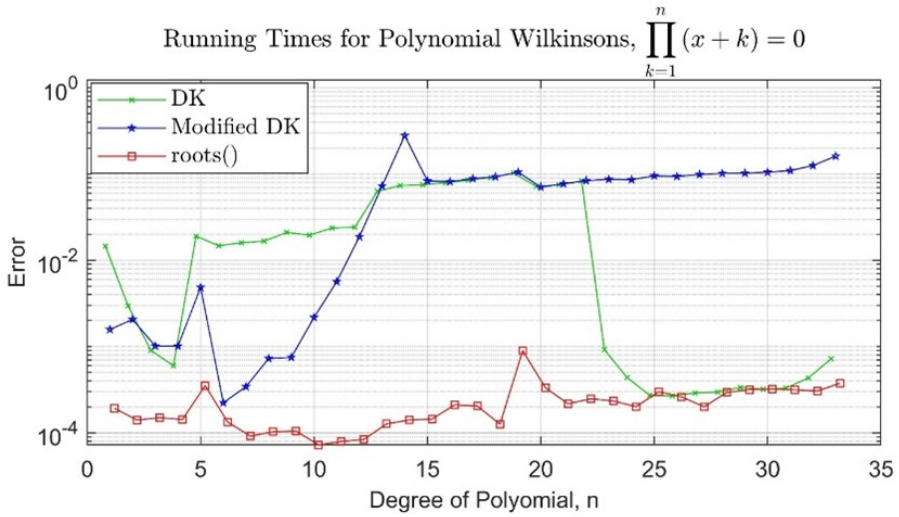

Figure 14: Running Time Comparisons Polynomial Wilkinsons for $n = 9$

Figure 14 reveals that MATLAB's roots() function is consistently the fastest across all degrees, benefiting from optimized internal routines. The classical DK algorithm shows moderate

computational cost, increasing gradually with polynomial degree. However, the Modified DK algorithm, while superior in accuracy, incurs a substantially higher computational cost, especially for higher-degree polynomials. This trade-off suggests that the algorithm's enhanced stability comes at the expense of increased iteration complexity or more stringent convergence criteria. Such behaviour is expected in algorithms designed for precision over speed, particularly when applied to numerically unstable problems.

To further investigate the numerical stability of root-finding algorithms, a perturbation was introduced to the Wilkinson polynomial by subtracting $2^{-23}x^{19}$ from the original form:

$$\prod_{k=1}^{20}(x+k) - 2^{-23}x^{19} = 0 \tag{13}$$

This modification targets the 18th coefficient, simulating a subtle yet impactful change that challenges the precision of numerical solvers. Given the Wilkinson polynomial's sensitivity to coefficient perturbations, this test serves as a stringent benchmark for evaluating algorithmic robustness [31].

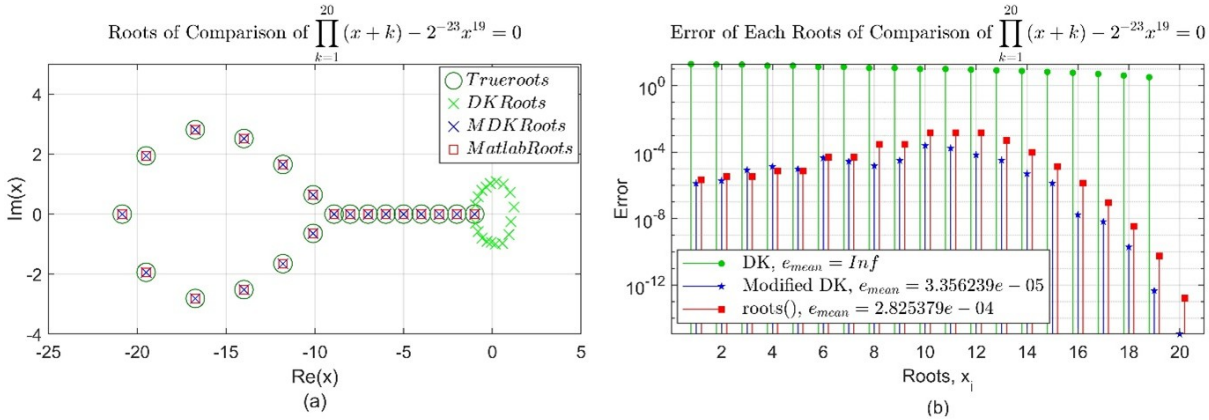

Figure 15: Error comparisons of each roots polynomial Wilkinsons with perturbations for $n = 20$

Figure 15a presents a visual comparison of the computed roots in the complex plane. The true roots (green circles) lie precisely on the real axis at integer positions from -1 to -20. The Modified DK algorithm (blue crosses) closely approximates these positions, with minimal deviation. MATLAB's roots() function (red squares) shows slightly larger deviations, particularly for roots near the center of the distribution. The classical DK algorithm (green crosses), however, fails to converge accurately, with several roots deviating significantly from their true positions. This visual evidence underscores the superior stability of the Modified DK algorithm in handling perturbed polynomials, while highlighting the limitations of the classical DK algorithm under such conditions.

Figure 15b provides a quantitative comparison of the error associated with each computed root. The classical DK algorithm exhibits catastrophic failure, with a mean error reported as infinite ($e_{\text{mean}} = \infty$), indicating divergence or non-convergence in the root-finding process. MATLAB's roots() function achieves a mean error of 2.83×10^{-4}, while the Modified DK algorithm significantly outperforms it with a mean error of only 3.36×10^{-5}. This result confirms that the Modified DK algorithm not only maintains converged but also delivers high precision in the presence of subtle coefficient perturbations. Its error profile remains consistently low across all roots, demonstrating its robustness and reliability.

The Wilkinson polynomial test confirms that the Modified DK algorithm offers superior accuracy and robustness in root-finding tasks involving ill-conditioned polynomials. Although it is computationally more expensive than the classical DK algorithm and MATLAB's roots() function, its performance in terms of error control makes it a valuable tool for applications where

precision is critical. These findings underscore the importance of including Wilkinson polynomial benchmarks in algorithmic validation to ensure reliability under extreme numerical conditions.

To assess the numerical accuracy of root-finding algorithms, we also conducted experiments on polynomials of degrees 7, 9, and 18 with integer coefficients and complex roots. The evaluation focused on three methods: the classical DK algorithm, a Modified DK algorithm, and MATLAB's roots() function. The mean and maximum errors of the computed roots were measured across 30 independent trials.

For polynomials of degree 7 in Figure 16, all methods demonstrated high precision, with mean errors on the order of 10^{-15}. The Modified DK method yielded the lowest mean error (4.93×10^{-15}), followed closely by DK (6.61×10^{-15}), while MATLAB's roots() exhibited slightly higher error (1.14×10^{-14}). The maximum error results followed a similar trend, confirming the reliability of all methods at low degrees.

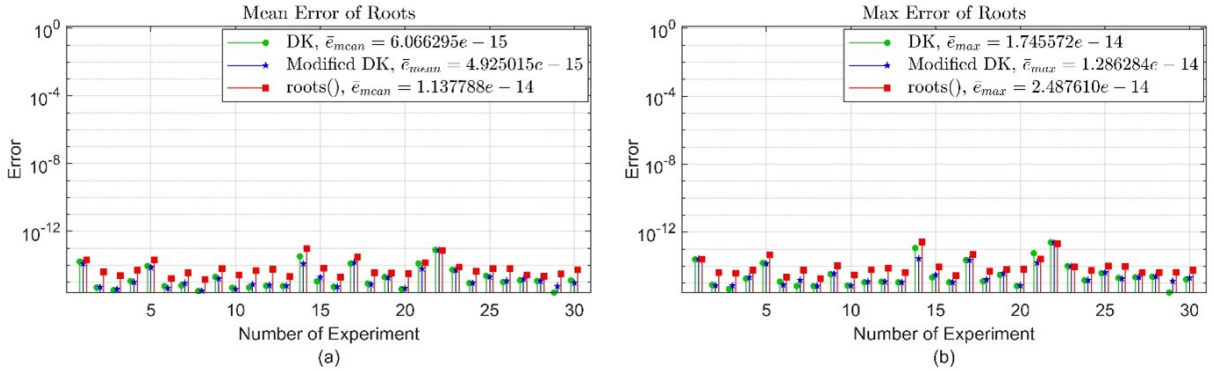

Figure 16: Comparisons of mean error of each root of 7-th degree polynomials with integer coefficients and complex roots with real part and imaginer part of the roots are integer.

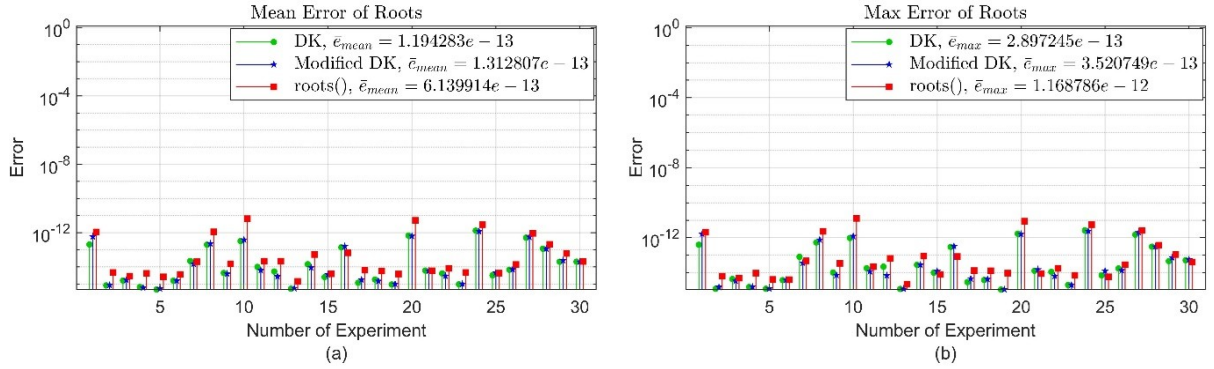

Figure 17: Comparisons of mean errors of each root of 9-th degree polynomials with integer coefficients and complex roots with real part and imaginer part of the roots are integer.

As the degree increased to 9 in Figure 17, the error magnitudes rose accordingly. The Modified DK and DK algorithm maintained comparable performance, with mean errors of 1.31×10^{-13} and 1.19×10^{-13}, respectively. In contrast, the MATLAB's roots() function showed a significantly higher mean error of 6.14×10^{-13}, indicating reduced numerical stability. The maximum error for MATLAB's roots() reached 1.17×10^{-12}, nearly four times greater than that of DK and Modified DK algorithms.

At degree 18 in Figure 18, the differences became more pronounced. The Modified DK algorithm achieved a mean error of 1.42×10^{-9}, substantially outperforming DK algorithm (1.78×10^{-8}) and MATLAB's roots() (3.60×10^{-8}). The maximum error for Modified DK was also the lowest (7.35×10^{-9}), compared to DK (2.03×10^{-7}) and MATLAB's roots() (2.14×10^{-7}).

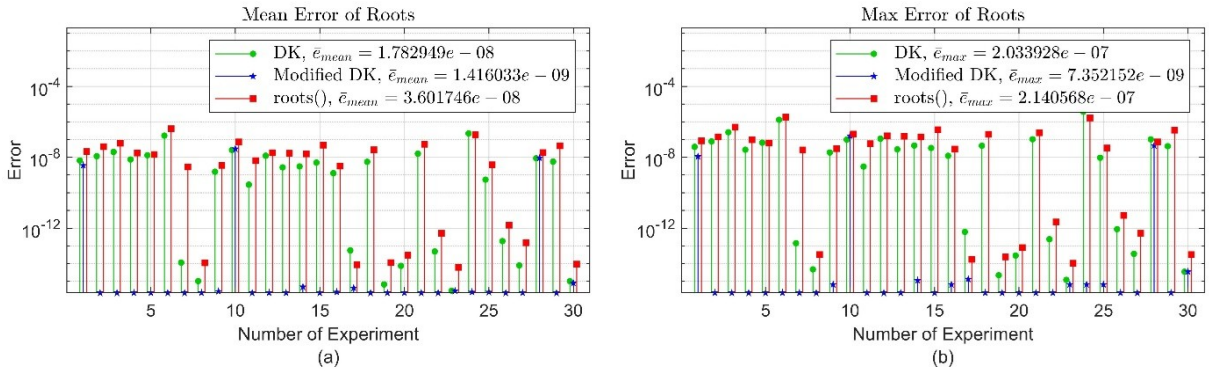

Figure 18: Comparisons of mean errors of each root of 18-th degree polynomials with integer coefficients and complex roots with real part and imaginer part of the roots are integer.

To assess the running time of three methods on polynomial with integer coefficient dan complex roots, experiments were conducted on the above polynomials of degrees 7, 9, and 18 with integer coefficients and complex roots and each method tested over 30 independent trials. As presented in Figure 19-21, for polynomials of degree 7, the MATLAB's roots() function exhibited the shortest mean running time (7.06×10^{-5} seconds), followed by DK algorithm (3.81×10^{-4} seconds), while Modified DK algorithm required the longest time (1.54×10^{-3} seconds). This pattern persisted for degree 9, where MATLAB's roots() remained the fastest (8.75×10^{-5} seconds), DK algorithm recorded 6.44×10^{-4} seconds, and Modified DK algorithm increased to 2.00×10^{-3} seconds.

At degree 18, the computational cost of Modified DK algorithm rose substantially to 2.78 seconds, compared to DK algorithm's 2.90×10^{-2} seconds and MATLAB's roots() at 3.25×10^{-4} seconds. This sharp increase suggests that the modifications introduced in the algorithm, while improving numerical accuracy, result in significantly higher computational overhead, particularly for high-degree polynomials.

The results demonstrate that the Modified DK algorithm consistently provides superior accuracy in computing the roots of polynomials with complex solutions, particularly as the polynomial degree increases. While all methods perform adequately at lower degrees, the Modified DK algorithm exhibits enhanced numerical stability and precision under more challenging conditions. These findings support its suitability for high-degree polynomial root-finding tasks where complex roots are involved.

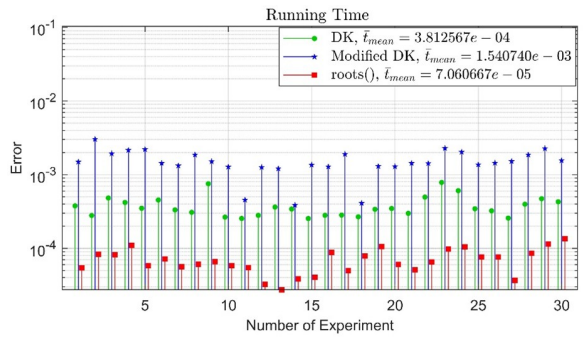

Figure 19: Comparison of running times of 7-th degree polynomials with integer coefficients and complex roots with real part and imaginer part of the roots are integer.

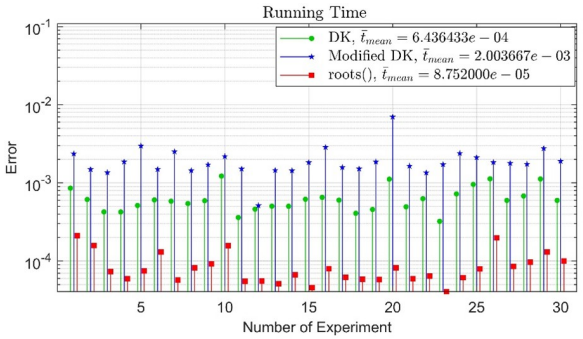

Figure 20: Comparison of running times of 9-th degree polynomials with integer coefficients and complex roots with real part and imaginer part of the roots are integer.

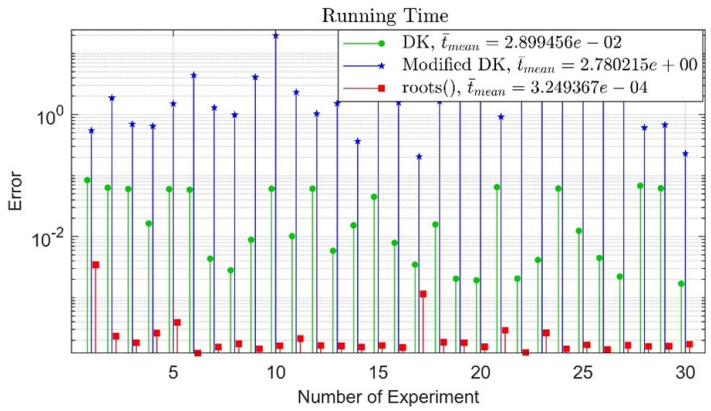

Figure 21: Comparison of running times of 18-th degree polynomials with integer coefficients and complex roots with real part and imaginer part of the roots are integer.

The running time analysis reveals a clear trade-off between computational efficiency and numerical precision.While the Modified DK algorithm consistently delivers superior accuracy, its execution time increases markedly with polynomial degree. In contrast, MATLAB's `roots()` function remains the most efficient across all tested degrees, though it does not match the precision of Modified DK algorithm in high-degree cases. These findings underscore the importance of selecting root-finding algorithms based on the specific requirements of accuracy and performance in practical applications.

To assess the numerical accuracy of root-finding algorithms, we conducted experiments on polynomials of degrees 7, 17, and 27 with real coefficients and real roots. The mean and maximum errors of the computed roots were measured across 30 independent trials.

All three methods demonstrated high precision at 7-th degree polynomials, as shown in Figure 22. The Modified DK algorithm yielded a mean error of 2.28×10^{-13}, closely matching DK algorithm (2.25×10^{-13}), while MATLAB's `roots()` exhibited a slightly higher error of 4.77×10^{-13}. The maximum errors followed a similar trend, with Modified DK and DK algorithm both below 7.5×10^{-13}, and MATLAB's `roots()` reaching 1.55×10^{-12}. As the degree increased shown in Figure 23, the error magnitudes rose accordingly. The Modified DK algorithm maintained superior accuracy with a mean error of 6.58×10^{-9}, outperforming DK algorithm (7.18×10^{-9}) and MATLAB's `roots()` (2.41×10^{-8}). The maximum error for Modified DK algorithm was 4.19×10^{-8}, significantly lower than DK algorithm (6.62×10^{-8}) and MATLAB's `roots()` (1.41×10^{-7}). At 27-th degree polynomial as in Figure 24, the classical DK algorithm failed to converge, resulting in infinite error values. The Modified DK algorithm remained stable, achieving a mean error of 1.42×10^{-4} and a maximum error of 1.40×10^{-3}. In contrast, the MATLAB's `roots()` function showed a mean error of 7.24×10^{-4} and a maximum error of 6.17×10^{-3}, indicating a substantial decline in accuracy.

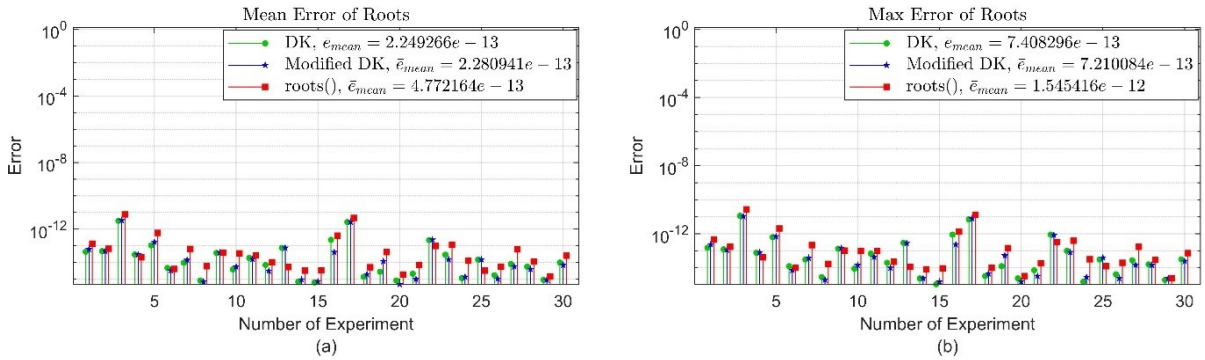

Figure 22: Comparisons of mean errors of each root of 7-th degree polynomials with real coefficients and real roots.

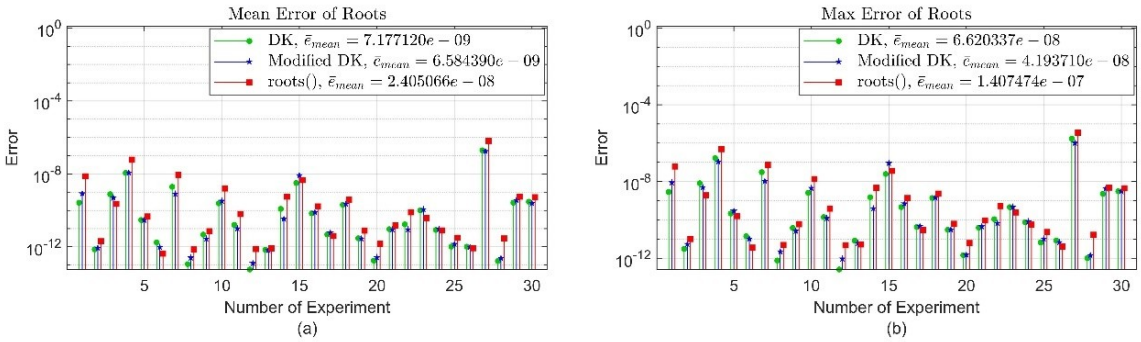

Figure 23: Comparisons of mean errors of each root of 17-th degree polynomials with real coefficients and real roots.

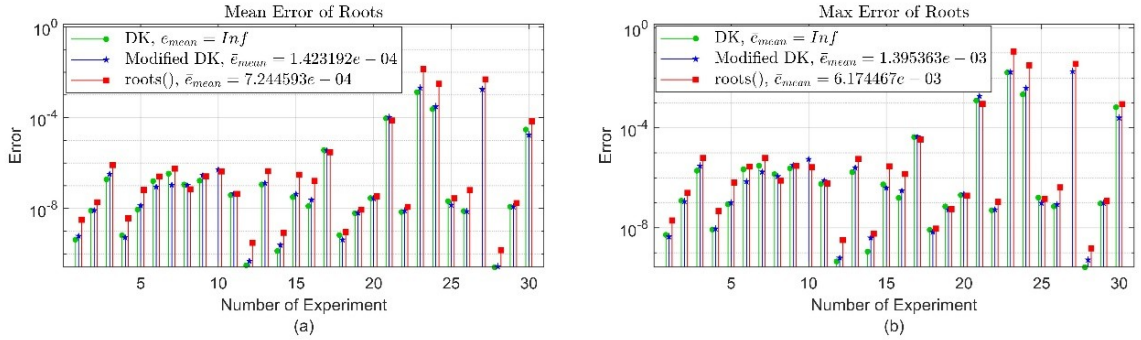

Figure 24: Comparisons of mean errors of each root of 27-th degree polynomials with real coefficients and real roots.

The results demonstrate that the Modified DK algorithm consistently provides superior accuracy and numerical stability when solving polynomials with real coefficients and real roots, particularly as the polynomial degree increases. While all methods perform adequately at lower degrees, the Modified DK algorithm maintains precision and robustness under more challenging conditions. The classical DK algorithm exhibits instability at higher degrees, and MATLAB's `roots()` function, although efficient, shows reduced accuracy in high-degree cases. These findings support the Modified DK algorithm as a reliable and accurate approach for root-finding in real-coefficient polynomial systems.

To assess the running time of three methods on polynomial with integer coefficient dan complex roots, experiments were conducted on the above polynomials of degrees 7, 17, and 27 with real coefficients and real roots and each method tested over 30 independent trials. The running time experiment are presented in Figure 25–27.

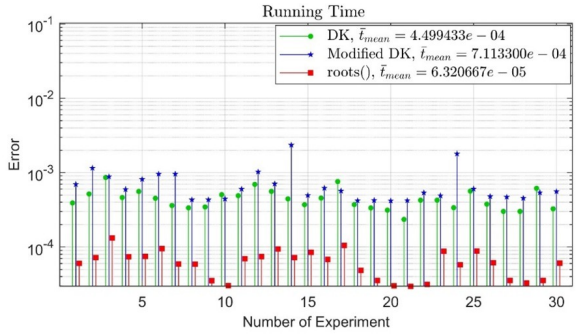

Figure 25: Comparison of running times of 7-th degree polynomials with real coefficients and real roots whose real and imaginary parts are integers.

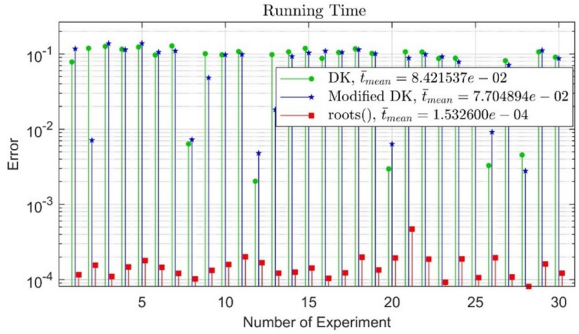

Figure 26: Comparison of running times of 17-th degree polynomials with real coefficients and real roots whose real and imaginary parts are integers.

In Figure 25, the MATLAB's `roots()` function exhibited the shortest mean running time at 6.32×10^{-5} seconds, followed by DK algorithm at 4.50×10^{-4} seconds, and Modified DK algorithm at 7.11×10^{-4} seconds. This indicates that MATLAB's `roots()` is approximately 7 times faster than DK algorithm and more than 11 times faster than Modified DK algorithm for low-degree polynomials.

For 17-th degree polynomials as in Figure 26, As the polynomial degree increased, the running times of DK and Modified DK algorithm rose significantly. DK algorithm recorded a mean time of 8.42×10^{-2} seconds, while Modified DK algorithm was slightly faster at 7.70×10^{-2} seconds. In contrast, MATLAB's `roots()` maintained a low execution time of 1.53×10^{-4} seconds, demonstrating its superior computational efficiency even for moderate-degree polynomials.

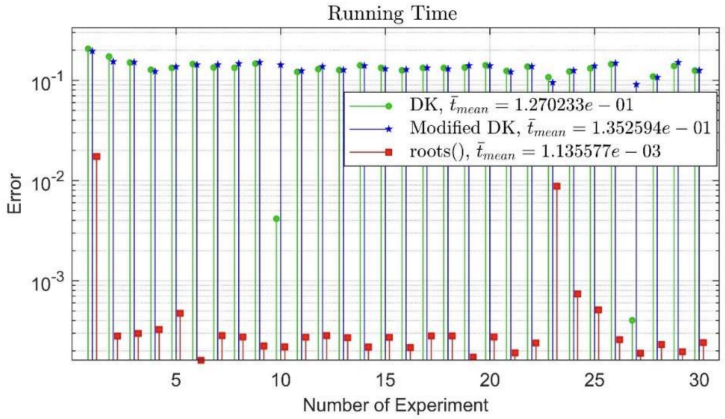

Figure 27: Comparison of running times of 27-th degree polynomials with real coefficients and real roots with real part and imaginer part of the roots are integer.

At the 27-th degree polynomials, both DK and Modified DK algorithm showed further increases in running time, with DK algorithm at 1.27×10^{-1} seconds and Modified DK algorithm at 1.35×10^{-1} seconds. Meanwhile, MATLAB's `roots()` remained significantly faster, with a mean running time of only 1.14×10^{-3} seconds. This suggests that MATLAB's `roots()` is over 100 times faster than the other two methods at this degree.

The running time analysis reveals that MATLAB's `roots()` function consistently outperforms both the classical and Modified Durand-Kerner algorithm in terms of computational efficiency across all tested polynomial degrees. While the Modified DK algorithm offers improved accuracy in root approximation, its execution time increases substantially with polynomial degree. These results highlight a clear trade-off between precision and performance, emphasizing the importance of algorithm selection based on application-specific requirements.

To assess the numerical accuracy of root-finding algorithms, we conducted experiments on polynomials of degrees 7, 17, and 27 with real coefficients and complex roots. The mean and maximum errors of the computed roots were measured across 30 independent trials. For 7-th degree polynomials as in Figure 28, all three methods demonstrated high precision at this low degree. The Modified DK algorithm yielded the lowest mean error (1.21×10^{-14}), closely followed by DK algorithm (1.21×10^{-14}), while MATLAB's `roots()` exhibited a slightly higher error (4.75×10^{-14}). The maximum error results followed a similar trend, with Modified DK and DK algorithm both below 3.6×10^{-14}, and MATLAB's `roots()` reaching 1.09×10^{-13}.

As the polynomial degree increased, the error magnitudes rose accordingly as in Figure 29. The Modified DK algorithm maintained competitive accuracy with a mean error of 3.92×10^{-12}, comparable to DK algorithm (3.68×10^{-12}), and significantly outperforming MATLAB's `roots()` (1.49×10^{-11}). The maximum error for Modified DK algorithm was 1.34×10^{-11}, slightly lower than DK algorithm (1.38×10^{-11}), and substantially better than `roots()` function (5.44×10^{-11}). At 27-th degree polynomial as in Figure 29, the classical DK algorithm failed to converge, resulting in infinite error values. The Modified DK algorithm remained stable, achieving a mean

error of 6.23×10^{-13} and a maximum error of 3.49×10^{-12}. In contrast, the MATLAB's `roots()` function showed a mean error of 2.62×10^{-12} and a maximum error of 1.80×10^{-11}, indicating a notable decline in accuracy. All three methods demonstrated high precision at 7-th degree polynomials, as shown in Figure 22.

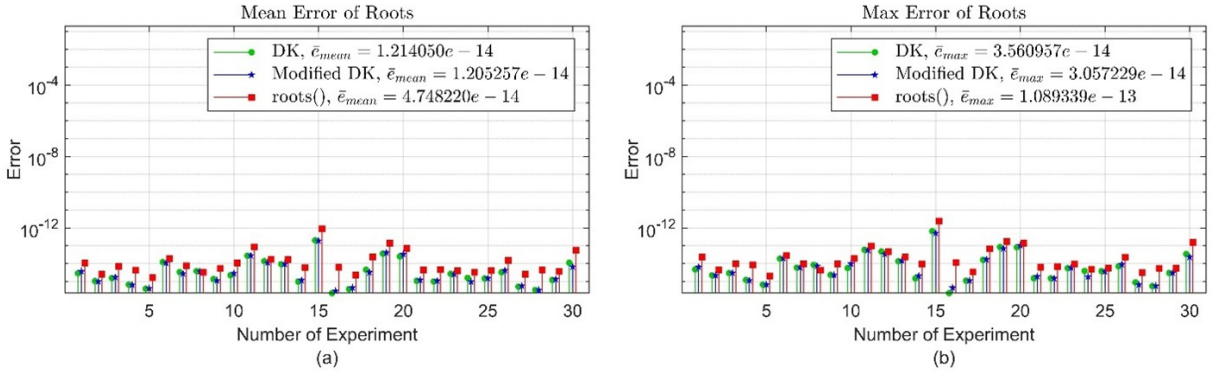

Figure 28: Comparisons of mean errors of each root of 7-th degree polynomials with real coefficients and complex roots.

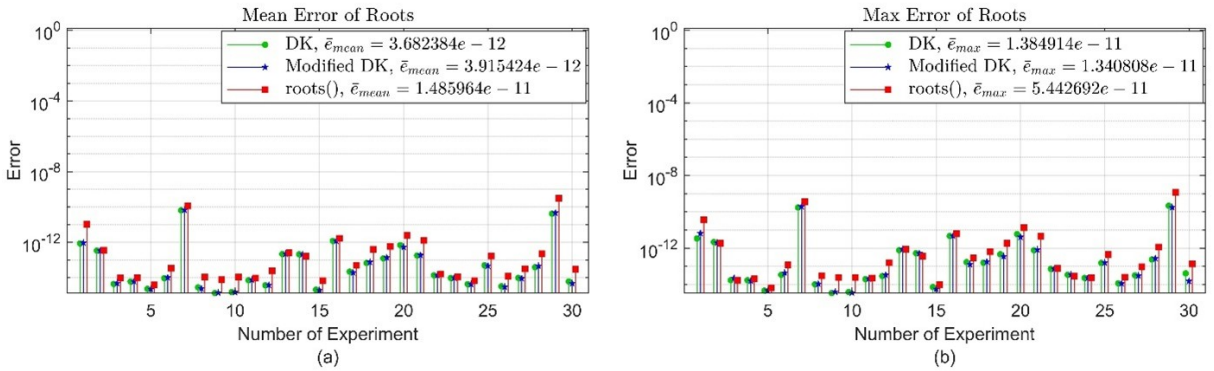

Figure 29: Comparisons of mean errors of each root of 17-th degree polynomials with real coefficients and complex roots.

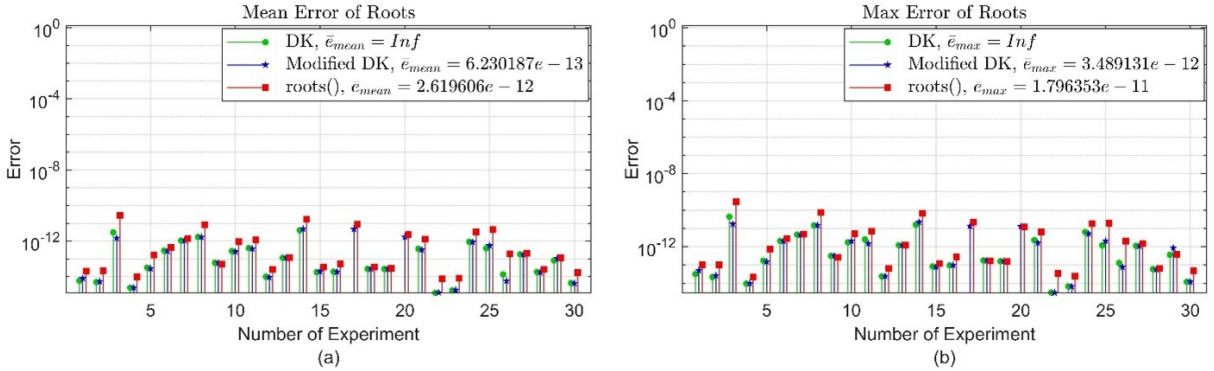

Figure 30: Comparisons of mean errors of each root of 23-th degree polynomials with real coefficients and complex roots.

The results demonstrate that the Modified DK algorithm consistently provides superior accuracy and numerical stability when solving polynomials with real coefficients and complex roots, particularly as the polynomial degree increases. While all methods perform adequately at lower degrees, the Modified DK algorithm maintains precision and robustness under more

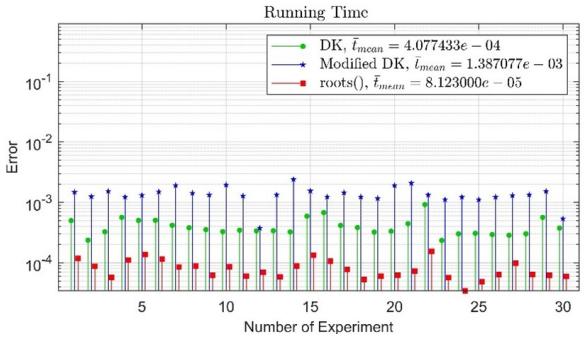

Figure 31: Comparison of running times of 7-th degree polynomials with real coefficients and complex roots.

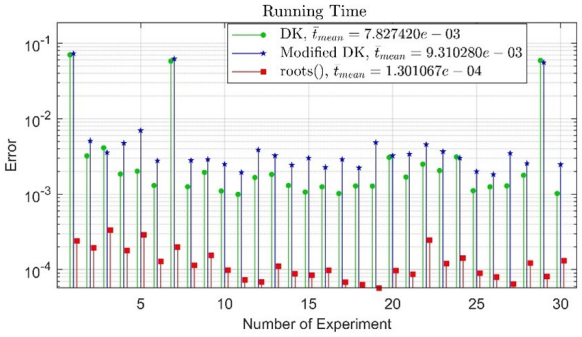

Figure 32: Comparison of running times of 17-th degree polynomials with real coefficients and complex roots.

challenging conditions. The classical DK algorithm exhibits instability at higher degrees, and MATLAB's `roots()` function, although efficient, shows reduced accuracy in high-degree cases. These findings support the Modified DK algorithm as a reliable and accurate approach for root-finding in real-coefficient polynomial systems with complex solutions.

To assess the running time of three methods on polynomial with integer coefficient dan complex roots, experiments were conducted on the above polynomials of degrees 7, 17, and 23 with real coefficients and complex roots and each method tested over 30 independent trials. The running time experiment are presented in Figure 31–33. In Figure 31, the roots function exhibited the shortest mean running time at 8.12×10^{-5} seconds, followed by DK algorithm at 4.08×10^{-4} seconds, and Modified DK algorithm at 1.39×10^{-3} seconds. This indicates that MATLAB's `roots()` is approximately 5 times faster than DK algorithm and more than 17 times faster than Modified DK algorithm for low-degree polynomials.

For polynomial 17-th degree as in Figure 32, as the polynomial degree increased, the running times of DK and Modified DK algorithm rose moderately. DK algorithm recorded a mean time of 7.83×10^{-3} seconds, while Modified DK algorithm was slightly slower at 9.31×10^{-3} seconds. In contrast, MATLAB's `roots()` maintained a low execution time of 1.30×10^{-4} seconds, demonstrating its superior computational efficiency even for moderate-degree polynomials.

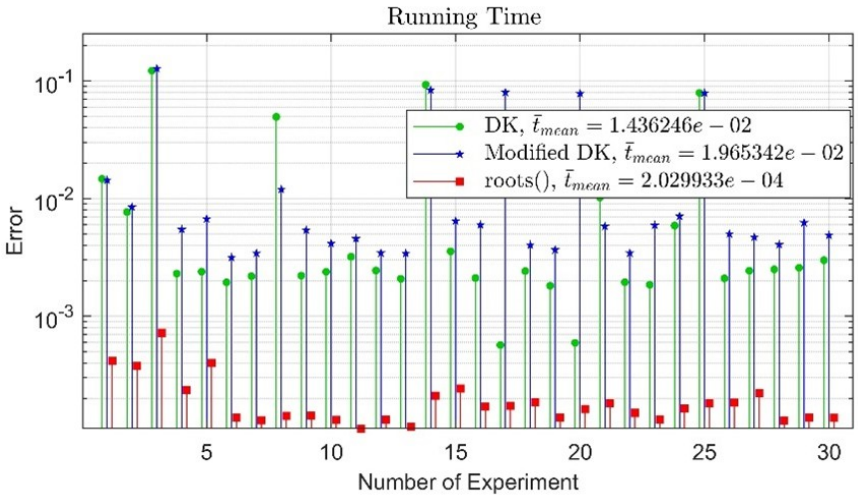

Figure 33: Comparison of running times of 23-th degree polynomials with real coefficients and complex roots.

At 23-rd degree polynomial as in Figure 33, both DK and Modified DK algorithm showed further increases in running time, with DK algorithm at 1.44×10^{-2} seconds and Modified DK algorithm at 1.97×10^{-2} seconds. Meanwhile, MATLAB's `roots()` remained significantly faster,

with a mean running time of approximately 1.50×10^{-4} seconds. This suggests that MATLAB's `roots()` is over 100 times faster than the other two methods at this degree.

The running time analysis reveals that MATLAB's `roots()` function consistently outperforms both the classical DK and Modified DK algorithm in terms of computational times efficiency across all tested polynomial degrees. While the Modified DK algorithm offers improved accuracy in root approximation, its execution time increases substantially with polynomial degree. These results highlight a clear trade-off between precision and performance, emphasizing the importance of algorithm selection based on application-specific requirements.

4 Conclusion

The proposed modification to the DK algorithm enhances both the accuracy and stability of root-finding for high-degree polynomials. Although it introduces additional computational times overhead at higher degrees, it offers a valuable alternative for applications requiring high numerical precision.

CRediT Authorship Contribution Statement

Bandung Arry Sanjoyo: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Visualization, Writing Original Draft, Supervision. **Mahmud Yunus:** Conceptualization, Methodology, Validation, Formal Analysis, Writing Review & Editing, Supervision. **Nurul Hidayat:** Conceptualization, Methodology, Resources, Data Curation, Visualization, Writing Review & Editing, Supervision, Project Administration.

Declaration of Generative AI and AI-assisted technologies

During the preparation of this work, the author used AI-assisted tools (Microsoft Copilot) to improve language clarity. After using these tools, the authorn reviewed and edited the content to ensure accuracy and originality. The author takes full responsibility for the final content of the manuscript.

Declaration of Competing Interest

The author declares no competing interests.

Funding and Acknowledgments

This research was funded by the Department of Mathematics, Institut Teknologi Sepuluh Nopember (ITS), Surabaya. The author gratefully acknowledges the support of Institut Teknologi Sepuluh Nopember for providing computational resources and a MATLAB software license.

Data and Code Availability

The MATLAB code and test data used in this study are available from the corresponding author upon reasonable request for research purposes.

References

[1] V. Y. Pan, "New progress in classic area: Polynomial root-squaring and root-finding," *arXiv preprint arXiv:2206.01727*, 2022.

[2] J. van Kan, G. Segal, and F. Vermolen, *Numerical Methods in Scientific Computing*. TU Delft Open Publishing, 2023.

[3] M. Shams, N. Kausar, S. Araci, and G. Oros, "Numerical scheme for estimating all roots of non-linear equations with applications," *AIMS Mathematics*, vol. 8, pp. 23 603–23 620, 2023. DOI: 10.3934/math.20231200.

[4] O. Aberth, "Iteration methods for finding all zeros of a polynomial simultaneously," *Mathematics of Computation*, vol. 27, no. 122, pp. 339–344, 1973.

[5] P. Batra, "Improvement of a convergence condition for the durand-kemer iteration," *Journal of Computational and Applied Mathematics*, vol. 96, pp. 117–125, 1998.

[6] H. Guggenheimer, "Initial approximations in durand-kerner's root finding method," *BIT Numerical Mathematics*, vol. 26, pp. 537–539, 1986. DOI: 10.1007/BF01935059.

[7] P. Marcheva and S. Ivanov, "On the semilocal convergence of a modified weierstrass method for the simultaneous computation of polynomial zeros," in *International Conference of Numerical Analysis and Applied Mathematics (ICNAAM)*, 2022, p. 420 012. DOI: 10.1063/5.0082007.

[8] B. Reinke, D. Schleicher, and M. Stoll, "The weierstrass–durand–kerner root finder is not generally convergent," *Mathematics of Computation*, vol. 92, pp. 839–866, 2023. DOI: 10.1090/mcom/3783.

[9] V. Y. Pan, "Solving a polynomial equation: Some history and recent progress," *SIAM Review*, vol. 39, pp. 187–220, 1997. DOI: 10.1137/S0036144595288554.

[10] M. Shams, N. Rafiq, N. Kausar, P. Agarwal, C. Park, and S. Momani, "Efficient iterative methods for finding simultaneously all the multiple roots of polynomial equation," *Advances in Difference Equations*, vol. 2021, p. 495, 2021. DOI: 10.1186/s13662-021-03649-6.

[11] B. Liu, Y. Yang, and M. Yu, "Enhancing numerical stability in multiport network synthesis with modified dk method," in *2024 IEEE International Microwave Filter Workshop (IMFW)*, IEEE, 2024, pp. 170–172. DOI: 10.1109/IMFW59690.2024.10477159.

[12] F. J. Hall, R. M. Marsli, and R. M. Marsli, "An application of gelfand's formula in approximating the roots of polynomials," *arXiv preprint arXiv:2505.03753*, 2025.

[13] D. Khomovsky, "On using symmetric polynomials for constructing root finding methods," *Mathematics of Computation*, vol. 89, pp. 2321–2331, 2020. DOI: 10.1090/mcom/3531.

[14] A. Tassaddiq, S. Qureshi, A. Soomro, E. Hincal, D. Baleanu, and A. Shaikh, "A new three-step root-finding numerical method and its fractal global behavior," *Fractal and Fractional*, vol. 5, no. 4, p. 204, 2021. DOI: 10.3390/fractalfract5040204.

[15] G. Milovanović, A. Mir, and A. Ahmad, "On the zeros of a quaternionic polynomial with restricted coefficients," *Linear Algebra and Its Applications*, vol. 653, pp. 231–245, 2022. DOI: 10.1016/j.laa.2022.08.010.

[16] V. Jain, "On cauchy's bound for zeros of a polynomial," *Approximation Theory and Its Applications*, vol. 6, pp. 18–24, 1990. DOI: 10.1007/BF02836305.

[17] W. Deren and Z. Fengguang, "On the determination of the safe initial approximation for the durand-kerner algorithm," *Journal of Computational and Applied Mathematics*, vol. 38, pp. 447–456, 1991. DOI: 10.1016/0377-0427(91)90188-P.

[18] A. Al-Swaftah, A. Burqan, and M. Khandaqji, "Estimations of the bounds for the zeros of polynomials using matrices," in *Mathematics and Computation*, D. Zeidan, J. Cortés, A. Burqan, A. Qazza, J. Merker, and G. Gharib, Eds., Springer Nature, 2023, pp. 25–37. DOI: 10.1007/978-981-99-0447-1_3.

[19] B. Sanjoyo, M. Yunus, and N. Hidayat, "A new initial approximation bound in the durand kerner algorithm for finding polynomial zeros," *arXiv preprint arXiv:2511.07728*, 2025.

[20] K. Madsen, "A root-finding algorithm based on newton's method," *BIT Numerical Mathematics*, vol. 13, pp. 71–75, 1973.

[21] H. Orchard, "The laguerre method for finding the zeros of polynomials," *IEEE Transactions on Circuits and Systems*, vol. 36, pp. 1377–1381, 1989. DOI: 10.1109/31.41294.

[22] T. R. Cameron, "An effective implementation of a modified laguerre method for the roots of a polynomial," *Numerical Algorithms*, vol. 82, pp. 1065–1084, 2019. DOI: 10.1007/s11075-018-0641-9.

[23] R. Imbach, V. Y. Pan, C. Yap, I. S. Kotsireas, and V. Zaderman, "Root-finding with implicit deflation," in *Computer Algebra in Scientific Computing*, M. England, W. Koepf, T. Sadykov, W. Seiler, and E. Vorozhtsov, Eds., Springer, 2019, pp. 236–245. DOI: 10.1007/978-3-030-26831-2_16.

[24] P. Marcheva and S. Ivanov, "Convergence analysis of a modified weierstrass method for the simultaneous determination of polynomial zeros," *Symmetry*, vol. 12, no. 9, p. 1408, 2020. DOI: 10.3390/sym12091408.

[25] P. Marcheva, "Fixed points and convergence of iteration methods for simultaneous approximation of polynomial zeros," Ph.D. dissertation, University of Plovdiv "Paisii Hilendarski", 2023.

[26] J. A. Gallian, *Contemporary Abstract Algebra*, 10th. Boca Raton: Chapman and Hall/CRC, 2020.

[27] D. Han, "The convergence of durand-kerner method for simultaneously finding all zeros of the polynomial," *Journal of Computational Mathematics*, vol. 18, pp. 567–570, 2000.

[28] G. Kjellberg, "Two observations on durand-kerner's root-finding method," *BIT Numerical Mathematics*, vol. 24, pp. 556–559, 1984. DOI: 10.1007/BF01934913.

[29] A. Terui and T. Sasaki, "Durand-kerner method for the real roots," *Japanese Journal of Industrial and Applied Mathematics*, vol. 19, pp. 19–38, 2002. DOI: 10.1007/BF03167446.

[30] X.-M. Niu and T. Sakurai, "A method for finding the zeros of polynomials using a companion matrix," *Japanese Journal of Industrial and Applied Mathematics*, vol. 20, pp. 239–256, 2003. DOI: 10.1007/BF03170428.

[31] J. Wilkinson, "The evaluation of the zeros of ill-conditioned polynomials. part ii," *Numerische Mathematik*, vol. 1, pp. 167–180, 1959. DOI: 10.1007/BF01386382.

[32] R. M. Corless and L. Sevyeri, "The runge example for interpolation and wilkinson's examples for rootfinding," *SIAM Review*, vol. 62, pp. 231–243, 2020. DOI: 10.1137/18M1181985.