Fast Polynomial Root Finder-Part Five
Fast Polynomial Root Finder- Part Five.

By Henrik Vestermark (hve@hvks.com)

Abstract:

We elaborated in the part five paper on higher order method for finding Polynomial roots and devised a modified Durand-Kerner method dealing efficiently with Polynomials with real coefficients. This paper is part of a multi-series of papers on using the same framework to implement different root finder methods.

Introduction:

In this paper (part five), we looked at the simultaneous method for polynomial root finders. Two methods come to mind. The first one is the Weierstrass method developed in 1891 and later rediscovered by Durand and Kerner in the 1960ties. The other method is the Aberth-Erhlich method also from 1960ties. Sometimes you see the first method named just as Weierstrass or Durand-Kerner or sometimes combined and abbreviated as the WDK method. In this paper, we will refer to it as the Durand-Kerner method.
Both methods are considered to be very robust and stable. The Durand-Kerner method is a numerical technique used for simultaneously finding all the roots of a given polynomial. This method is particularly valuable in numerical analysis and computational mathematics for its efficiency and ability to handle polynomials of high degrees.

Fast Polynomial Root Finder-Part Five

Contents

Durand-Kerner Background

The method was independently developed by two mathematicians: Eugene Durand and Arthur Kerner. It improved upon the earlier ideas of Karl Weierstrass, hence its alternative name, the Weierstrass method. The Durand-Kerner method is a root-finding algorithm that falls into the category of simultaneous iterative methods, distinct from more traditional sequential methods like Newton's method.

The Polynomial Problem

Given a polynomial of degree with real or complex coefficients, the challenge is to find all its roots, which could also be complex. This problem is central in various fields of science and engineering, where determining the roots of polynomials is a frequent necessity.

Method Overview

The Durand-Kerner method starts with an initial guess for each root. These guesses should ideally be distinct. The method then iteratively refines these guesses using a specific formula. The beauty of this method lies in its simultaneous updating of all root approximations in each iteration, leveraging the polynomial's behavior at multiple points.

Advantages

1. Efficiency: The method can be quite efficient for polynomials of high degrees.
2. Generality: It applies to both real and complex polynomials.
3. Simultaneous Convergence: All roots are refined in parallel, leading to a potentially faster overall convergence.
4. Avoid the deflation process as typical e.g. Newton's method

Considerations

While the Durand-Kerner method is powerful, its performance can depend heavily on the initial guesses. Poor choices can lead to slow convergence or sometimes even convergence to incorrect values. Additionally, the method can be computationally intensive, especially for polynomials with a large number of roots.

Applications

The Durand-Kerner method finds applications in various domains, including control theory, signal processing, and computational physics, where solving polynomial equations is a fundamental task.

The Durand-Kerner method stands as an elegant and efficient solution to the age-old problem of finding polynomial roots, embodying a unique blend of mathematical creativity and computational practicality.

The Durand-Kerner Method

The Durand-Kerner method is a method that simultaneously converges to all the roots of the Polynomial using the following iteration:

$$x_{k+1} = x_k - \frac{P(x_k)}{\prod_{j=1,j\neq k}^{n}(x_k - x_j)}$$

where x_k is the kth approximation of the root, $P(x_k)$ is the value of the polynomial at x_k, and $\prod_{j=1,j\neq k}^{n}(x_k - x_j)$ is the product of the differences between x_k and the other approximations x_j.

The method is also called the Weierstrass method and the corrections factor:
$W_k = \frac{P(x_k)}{\prod_{j=1,j\neq k}^{n}(x_k - x_j)}$ is called the Weierstrass corrections.

In the above Weierstrass (Durand-Kerner) method, the “step size”, (or W_k) for each iteration, when updating the roots, does not have a direct geometric interpretation akin to the tangent line in Newton’s method. However, we can think about it in terms of moving towards the roots in the complex plane:

Each root approximation in the Weierstrass method is a point in the complex plane. During each iteration, this point moves closer to the actual root. The “step size” is essentially the distance in the complex plane that each root approximation moves during an iteration.
The step size for a particular root depends not only on the value of the polynomial at that point but also on the positions of all other current root approximations. This is unlike Newton’s method, where the step for each root is independent of the others.
There’s no direct equivalent of a tangent line or its intersection with the x-axis, as in Newton’s method. Instead, the Weierstrass method’s movement towards the root is based on a more complex interplay of all the roots.
Geometrically, one could visualize the root approximations as points in the complex plane that gradually spiral or move in a pattern toward the actual roots of the polynomial.

While the Weierstrass method lacks the straightforward geometric interpretation of Newton’s method, it can be visualized as a dynamic process in the complex plane where all root approximations simultaneously move towards their respective true roots, influenced by the positions of each other.

The Durand-Kerner method has a quadratic convergence rate (the same as Newton's method) and has some advantages compared to other root-finding methods. Compared to Newton's method we can see from the above formula that we don’t require the need of the Polynomial first derivative. Secondly, it is a simultaneous method so we don’t need to worry about dividing out the factors or using deflation techniques. and the associated accumulated errors arising from inaccuracy in the deflation process.

Since we normally have more than one root, we can write the above equation in a more concise form:

Fast Polynomial Root Finder-Part Five

$$x_i^{(k+1)} = x_i^{(k)} - \frac{P\left(x_i^{(k)}\right)}{\prod_{j=1,j\neq i}^{n}\left(x_i^{(k)} - x_j^{(k)}\right)} \quad i = 1, \dots, n \text{ and } k = 0,1, \dots$$

Or using the Weierstrass correction

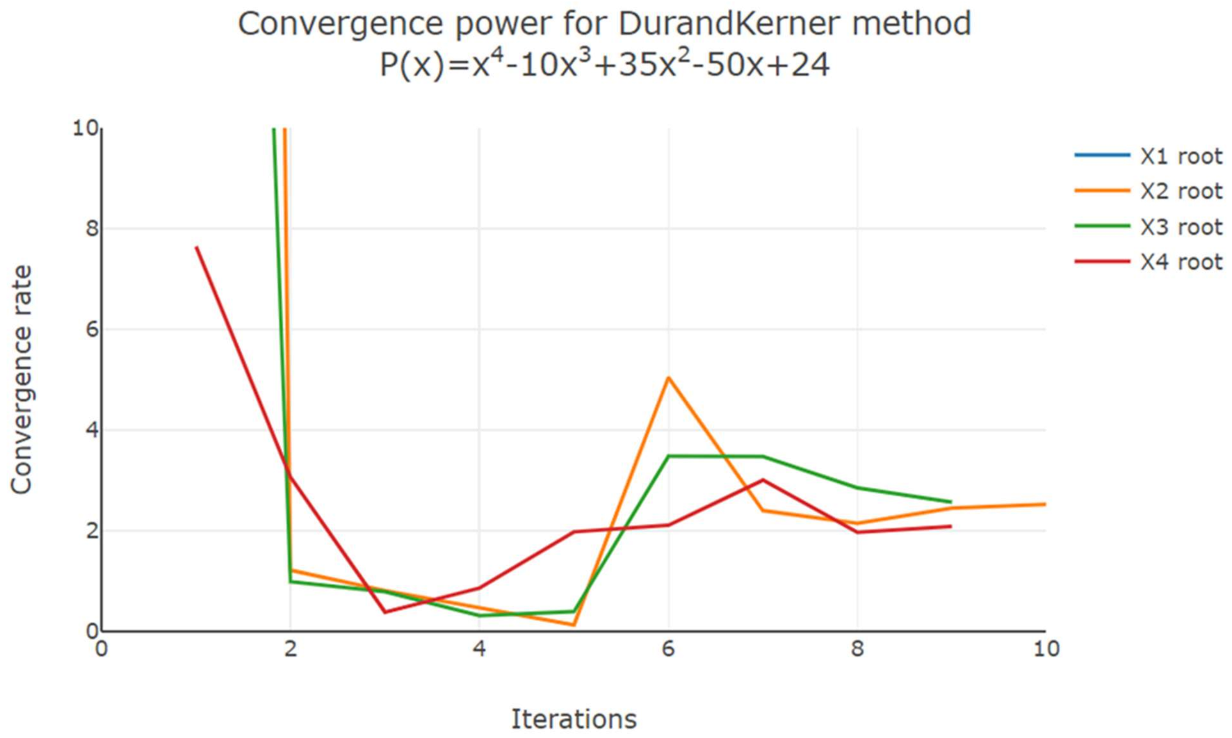
$$x_i^{(k+1)} = x_i^{(k)} - W_i^{(k)} \;\; i = 1, \dots, n \text{ and } k = 0,1, \dots$$

As usual, the method has only linear convergence when multiplicity > 1. The starting point used in the code example below is primitive as follows:

$$x_i^{(0)} = (0.4 + i0.9)^{i-1} \quad i = 1, \dots, n$$

Take the polynomial:

$P(x)=(x-1)(x-2)(x-3)(x-4)=1x^4-10x^3+35x^2-50x+24$

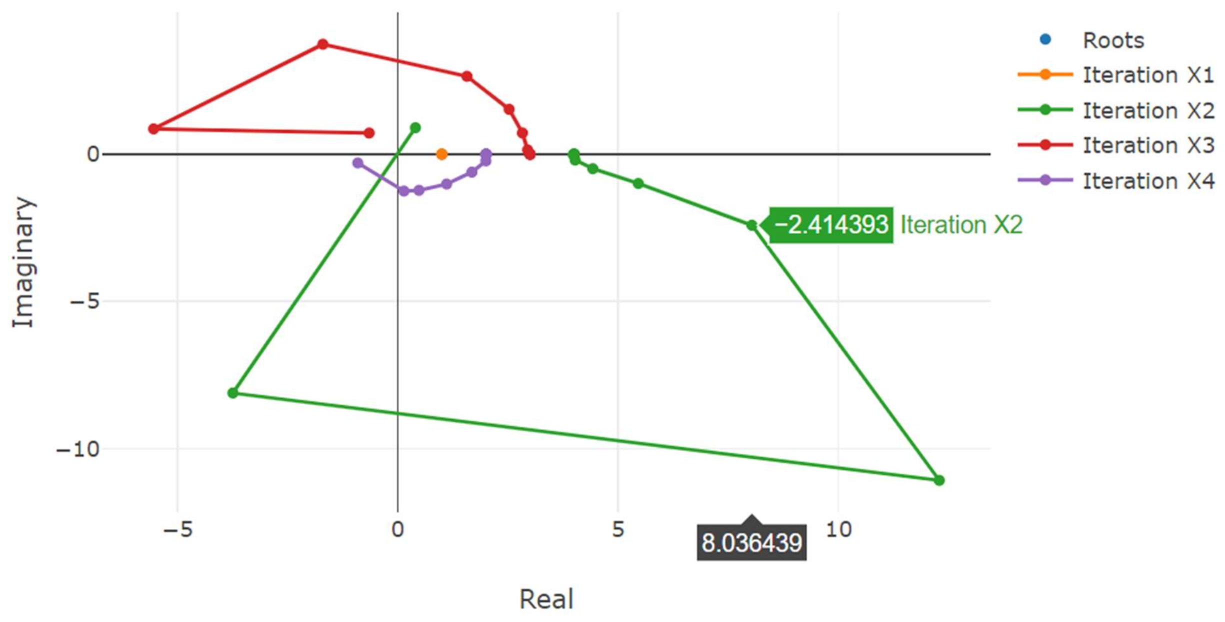


We see the convergence rate approaching a little over 2 in line with the expectation for the Durand-Kerner method. Also, notice that it takes around 6 iterations before we begin to see some traction, and 9 to 10 iterations are required to reach the desired accuracy.

And the iterations trail towards the roots.

Fast Polynomial Root Finder-Part Five

Roots of $P(x)=x^4-10x^3+35x^2-50x+24$



We notice that the second root (green path) starts at the complex point (0.4+i0.9) and then goes on a longer field trip into the complex plane before gravitating towards the root x=4. The same is partly true for the third root (red path) that starts in (-0.7+i0.7) and then goes out in the wrong direction, turns around, and navigates towards the root at x=3. This is very typical for the Durand-Kerner method in that initially it can go out into a seemingly wild direction but then get their act together and move towards the roots.

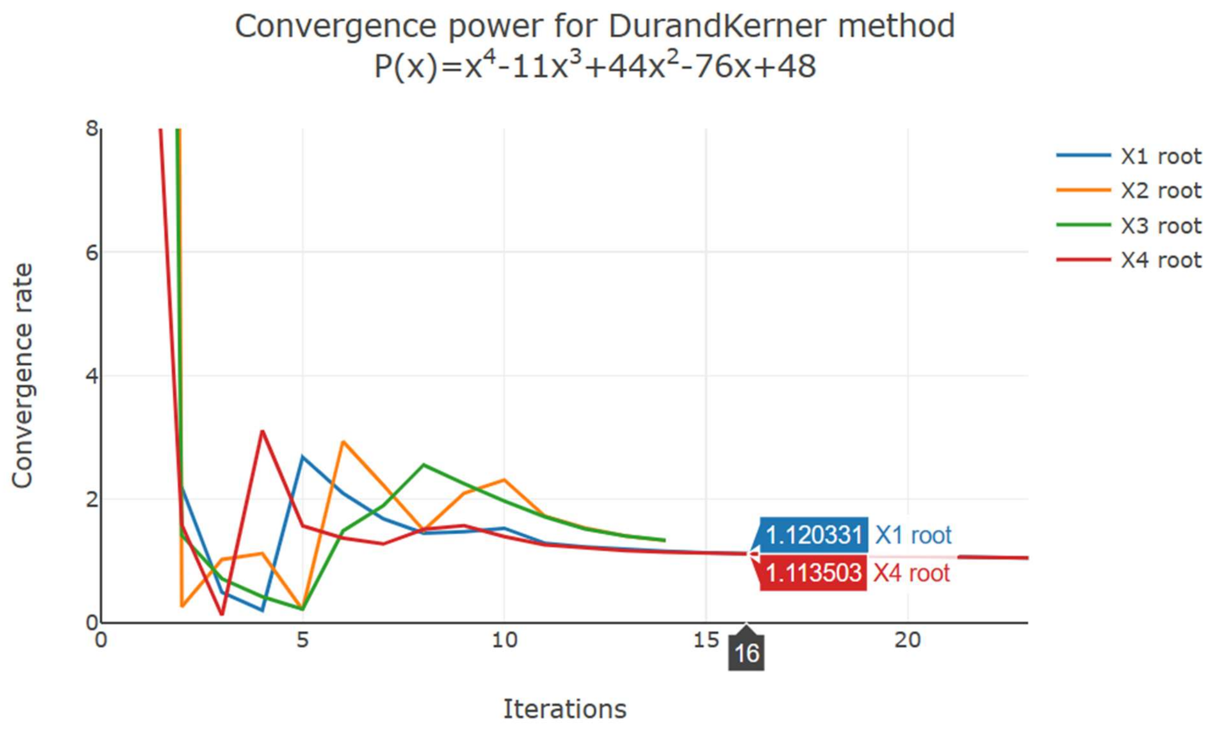
The issue with multiple roots for the Durand-Kerner Method

The Durand-Kerner also suffers from the issue with multiple roots where the convergence rate drops from quadratic (convergence power of 2) to a linear convergence power.

Take the polynomial:

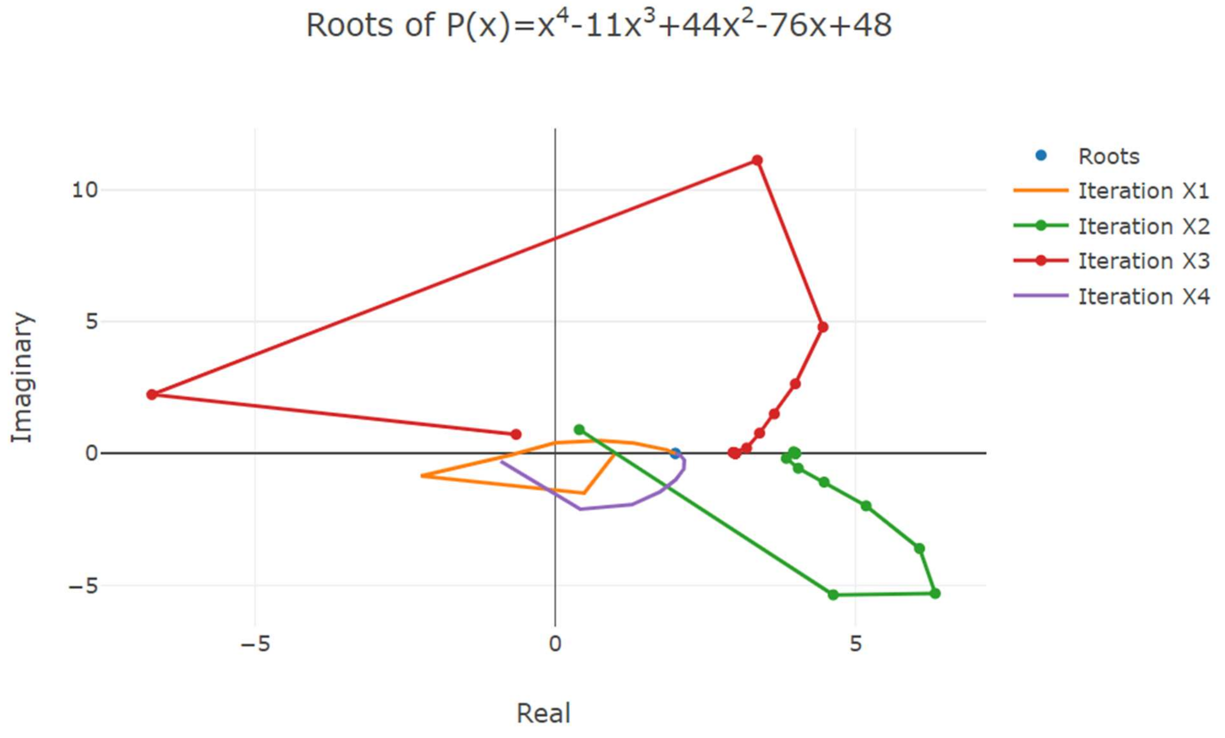
$$P(x)=(x-2)(x-2)(x-3)(x-4)=1x^4-11x^3+44x^2-76x+48.$$

That has a double root at x=2



We see that after some initial fluctuation, the convergence power stabilized between 1 and 2 with slow convergence as a result. We need 23 iterations before we can accept the result.

And the pathway to the roots.



We need to implement a modified version that can also cope with the multiple roots issue. Luckily the field has been studied and several authors have suggested modifications to improve the convergence rate [8]. However, to my knowledge, most of the suggestions only improve the convergence rate slightly and lack an efficient handling of the multiple root issues.

Safe Convergence Zone

The Durand-Kerner method is considered to be very stable and can nearly always converge from any start point. I say nearly because it has been documented that you can find a starting point where the method will not converge but compared with the Newton method it will be more robust and resilient to nearly always find a root regardless of the starting point. With that said it should not be compared directly with the Newton method but with the modified Newton version layout in part one and part two. If we compare these versions there is no difference in the stability and resiliency of both methods. For the Durand-Kerner method in [12], they establish a safe convergence zone when:

$$W^{(k)} < \frac{d^{(k)}}{2n+1}$$

Where $W^{(k)}$ is the kth iteration Weierstrass corrections, n is the polynomial degree and

$$d^{(k)} = \min_{j \neq i} \left| x_i^{(k)} - x_j^{(k)} \right|$$

The computation to determine if we are within a safe convergence turns out to be very useful in the implementation of the Durand-Kerner method.
A little side note here is that the above criteria for a safe convergence zone are never reached when dealing with multiple roots. Instead, we use a trick that if $|P(x_k)|<e^{(1/3)}$, where *e* is the stopping criteria for the polynomial evaluation then we will also consider it to be within the safe convergence zone.

Comparing Newton and the Durand-Kerner method

To compare different methods with others we can use a well-known efficiency index to see how it stacks up against other derivative-based methods.

The efficiency index is $q^{\frac{1}{p}}$, where q is the method convergence order and *p* is the number of polynomial evaluations for the method. For the Newton, method *p* is 2 since we need to evaluate both P(z) and P'(z) per iteration, and the Newton method has a convergence order of q=2 so we get Efficiency index= $2^{\frac{1}{2}} = 1.42$

For the Durand-Kerner method, we only evaluate P(x) for each iteration, we get $2^{\frac{1}{1}} = 2$
Which is larger than the Newton method.

The advantage of the Durand-Kerner method is the simultaneous iterations to all roots, it also avoids the deflation process that many other methods need to divide one of two roots up in the Polynomial. However, one of the disadvantages is that for polynomials with real coefficients, we can't take advantage of complex conjugated roots but need the simultaneous method to find them individually. Due to inaccuracy in the process, this can lead to complex roots not strictly being complex conjugated roots. Furthermore, the Durand-Kerner method usually requires more iterations before traction to a root happens compared to the Newton method.

What to Modify?

Compared to the Newton method (part two) we can luckily reuse most of the code already available with the Newton method. The Horner method for evaluating the Polynomial P(x) is the same, the same goes for the Adams test of when P(x) is sufficiently low to accept the root. We can skip the deflation process since we don't need it for the Durand-Kerner method. And technically we could also skip the direct solving of the quadratic equation since all roots will be found at once. We therefore have the modified steps for the Durand-Kerner method

The Durand-Kerner steps Include:

1. Finding an initial point
2. Executing the Durand-Kerner iteration, including polynomial evaluation via the Horner method
3. Detection of the multiple roots issue
4. Calculating the final upper bound. (Adam's Test)
5. Solving the quadratic equation. Only if the initial polynomial was of degree 2 or lower.

Finding the initial point.

In Parts One and Two we have established a suitable starting point for a root search so we know that a root always lies outside the circle in the complex plane of the starting point. However, we are iterating simultaneously towards all roots so we would need to set the initial start guess for all the roots. Unfortunately, there is no adequate algorithm to do that. Instead, we use a fixed starting point and then distribute the remaining starting point around an outward circle in the complex plane. Using the formula:

$$x_i=(0.4+i0.9)^i, \text{ for } i=0,\dots,n-1$$

A similar technique was also suggested in [11] with approximately the same result.

Other starting points e.g. Kalantaris. Kalantaris has a formula for computing an outer circle where all the roots are located. A suggested starting point could be an inward/outward circle with a starting radius of 2/3 of the Kalantaris circle. It kind of makes sense that you choose this technique but the test has shown that it does not reduce the number of iterations in any significant way.

Fast Polynomial Root Finder-Part Five

The modified Durand-Kerner improves the handling of multiple roots

As mentioned, before there is no direct modified version of the Durand-Kerner method that can deal efficiently with multiple roots. Instead, we can resort to a straightforward solution to that problem. When we are in the safe convergence zone we then check if we are dealing with a multiple root and if so, we simply apply the modified Newton step for the root under computation. This means that we discard the Weierstrass correction for that root and instead use the Newton step correction.

$$x_{k+1} = x_k - m\frac{P(x_k)}{P'(x_k)}, for\ m = 1,2,\dots,n$$

As long as P(x_{k+1}) is decreasing. The best Newton steps (best m) are then returned as the step size for the root.

However, how do we detect that we are dealing with multiple roots? A simple test is to check the convergence rate q given by:

$$q = \frac{\log(|x_k - x_{k-1}|)}{\log(|x_{k-1} - x_{k-2}|)}$$

For normal convergence towards simple roots the q ~2, however, when approaching multiple roots q usually falls in between 1.0 and 1.4 instead of 2, and if we further check that we are within the safe convergence zone then we have a pretty accurate detection of a multiple root situation.

Now selectively adding the Newton step correction also adds the need to evaluate P'(x) which was not needed when using Durand-Kerner unmodified. However, if we need to handle the multiple roots effectively, we can use the modified Newton correction in the case of multiple roots. In the multiple root example, the number of iterations drops from 23 to 12 iterations which is more than justified to add this Newton correction to the method.,

A suitable stopping criterion for the simultaneous root search

We already established a suitable stopping criterion for Polynomials with real coefficients and real or complex roots from Part Two.
Since we are simultaneously iterating toward all the roots at once and these roots can have different magnitudes, we can't just establish a single upper bound for the error in the evaluation of the polynomial. We have to carry individual stopping criteria for each root. We can however reuse Adam's test from Part Two applied individually for each root. Since we can't calculate the correct upper bound before we get closer to the roots, we are also using the simple upper bound from Part One and Two initially, and when we get closer to the root, we then call Adam's test for each root. The closeness is determined when an approximation to a root z is within the safe convergence zone previously established. This test ensures that we calculate the correct upper bound and ensure the correct termination of the root search.

Fast Polynomial Root Finder-Part Five

The Implementation of the Durand-Kerner Algorithm

We use the following algorithm for the modified Durand-Kerner Algorithm.

```
1) Eliminate all zero-roots
2) If the polynomial degree is less than 3 then solve it directly and exit
3) Set up the simultaneous roots search
    a. Compute the initial starting point for each $x_0, x_1, …, x_n$
    b. Compute the initial stopping criteria for each root $x_0, x_1, …, x_n$
    c. For each iteration k=1,… do
        i. For each root $x_0, x_1, …, x_n$
            1. If root $x_i$ has been flagged as found then continue to the next root
            2. Compute the Weierstrass correction $W(x_i)$
            3. Compute the next approximation $x_i^{(k+1)} = x_i^{(k)} - W\left(x_i^{(k)}\right)$
            4. Compute the convergence power q for $x_i$
            5. If within a safe convergence zone and $q<1.4$
                a. Do a modified Newton multiple root convergence step and update $W\left(x_i^{(k)}\right)$ with the modified Newton step
            6. If within the convergence zone then recalculate the upper bound for the root $x_i$
            7. if $x_i^{(k)} - W\left(x_i^{(k)}\right) = x_i^{(k)}$ or $P(x_i^{(k)})<eps$ then flag the root as finished for $x_i$
        ii. Compute if a safe convergence zone has been reached
    d. All root has now been found
```

The C++ code

The C++ code below finds the Polynomial roots with Polynomial real coefficients using the modified Durand-Kerner method.

```
/*
 *******************************************************************************
 *
 *                        Copyright (c) 2023
 *                        Henrik Vestermark
 *                        Denmark, USA
 *
 *                        All Rights Reserved
 *
 *   This source file is subject to the terms and conditions of
 *   Henrik Vestermark Software License Agreement which restricts the manner
 *   in which it may be used.
 *
 *******************************************************************************
*/

/*
 *******************************************************************************
 *
 * Module name     :   DurandKerner.cpp
 * Module ID Nbr   :
 * Description     :   Solve n degree polynomial using the DUrand-Kerner method
 * -------------------------------------------------------------------------
 * Change Record   :
 *
 * Version    Author/Date         Description of changes
 * -------  -------------   ----------------------
 * 01.01      HVE/24Nov2023 Initial release
 *
 * End of Change Record
 * -------------------------------------------------------------------------
*/

// define version string
static char _VDURAND_[] = "@(#)testDurandKerner.cpp 01.01 -- Copyright (C) Henrik
Vestermark";

#include <algorithm>
#include <vector>
#include <complex>
#include <limits>
#include <iostream>
#include <functional>
#include <cmath>

using namespace std;

constexpr int       MAX_ITER = 200;

// This is the Durand-Kerner method
// It simultaneously finds all the roots
//
static vector<complex<double>> PolynomialRootsDurandKerner(const vector<double>&
coefficients)
{
    struct eval { complex<double> z{}; complex<double> pz{}; double apz{}; };
    const complex<double> complexzero(0.0); // Complex zero (0+i0)
    size_t n=coefficients.size()-1;         // Degree of Polynomial p(x)
    vector<complex<double>> roots;  // Holds the roots of the Polynomial
    vector<double> coeff(coefficients.size()); // Holds the current coefficients of P(z)

    copy(coefficients.begin(), coefficients.end(), coeff.begin());

    // Step 1 eliminate all simple roots
    for (; n > 0 && coeff.back() == 0.0; --n)
        roots.push_back(complexzero);  // Store zero as the root
    if (n == 0)  // if polynomial empty?
        return roots;

   // Solve any remaining linear or quadratic polynomial
   // For Polynomial with real coefficients a[],
   // The complex solutions are stored in the back of the roots
    auto quadratic = [&](const std::vector<double>& a)
    {
        const size_t n = a.size() - 1;
        complex<double> v;
        double r;

        // Notice that a[0] is !=0 since roots=zero has already been handle
        if (n == 1)
            roots.push_back(complex<double>(-a[1] / a[0], 0));
        else
        {
            if (a[1] == 0.0)
            {
                r = -a[2] / a[0];
                if (r < 0)
                {
                    r = sqrt(-r);
                    v = complex<double>(0, r);
                    roots.push_back(v);
                    roots.push_back(conj(v));
                }
                else
                {
                    r = sqrt(r);
                    roots.push_back(complex<double>(r));
                    roots.push_back(complex<double>(-r));
                }
            }
            else
            {
                r = 1.0 - 4.0 * a[0] * a[2] / (a[1] * a[1]);
                if (r < 0)
                {
                    v = complex<double>(-a[1] / (2.0 * a[0]), a[1] * sqrt(-r) / (2.0 * 
a[0]));
                    roots.push_back(v);
                    roots.push_back(conj(v));
                }
                else
                {
                    v = complex<double>((-1.0 - sqrt(r)) * a[1] / (2.0 * a[0]));
                    roots.push_back(v);
                    v = complex<double>(a[2] / (a[0] * v.real()));
                    roots.push_back(v);
                }
            }
        }
        return;
    };

    // Solve it directly?
    if (n <= 2)
    {
        quadratic(coeff);
        return roots;
    }

    // Evaluate a polynomial with real coefficients a[] at a complex point z and
    // return the result
    // This is Horner's methods avoiding complex arithmetic
    auto horner = [](const vector<double>& a, const complex<double> z)
    {
        const size_t n = a.size() - 1;
        double p = -2.0 * z.real();
        double q = norm(z);
        double s = 0.0;
        double r = a[0];
        eval e;

        for (size_t i = 1; i < n; i++)
        {
            double t = a[i] - p * r - q * s;
            s = r;
            r = t;
        }

        e.z = z;
        e.pz = complex<double>(a[n] + z.real() * r - q * s, z.imag() * r);
        e.apz = abs(e.pz);
        return e;
    };

    // Calculate an upper bound for the rounding errors performed in a
    // polynomial with real coefficient a[] at a complex point z.
    // (Adam's test)
    auto upperbound = [](const vector<double>& a, const complex<double> z)
    {
        const size_t n = a.size() - 1;
        double p = -2.0 * z.real();
        double q = norm(z);
        double u = sqrt(q);
        double s = 0.0;
        double r = a[0];
        double e = fabs(r) * (3.5 / 4.5);
        double t;

        for (size_t i = 1; i < n; i++)
        {
            t = a[i] - p * r - q * s;
            s = r;
            r = t;
            e = u * e + fabs(t);
        }
        t = a[n] + z.real() * r - q * s;
        e = u * e + fabs(t);
        e = (4.5 * e - 3.5 * (fabs(t) + fabs(r) * u) +
            fabs(z.real()) * fabs(r)) * 0.5 * pow((double)_DBL_RADIX, -DBL_MANT_DIG +
1);

        return e;
    };

    // Do Durand-Kerner iteration for polynomial order higher than 2,
    // The preliminary stopping value for P(x)
    const double eps = 4 * n * abs(coeff[n]) * pow((double)_DBL_RADIX, -DBL_MANT_DIG);
    const double epsLow = pow(eps, 1.0 / 3.0);
    size_t i;
    int itercnt;            // Hold the number of iterations per root
    bool stage1 = true;     // Initially stage 1 is set to true when the convergence
test is true then stage1 is reset to false
    complex<double> z;      // Use as temporary variable
    vector<bool> finish(n, false);  // Vector flag indicates if an individual root
search is finish
    vector<double> upper(n, eps);   // The individual upper bound for each root that
needs to be satisfied
    vector<complex<double>> W(n);   // Weierstrass correction. (step size)
    vector<eval> pz;                // vector of P(z)
    vector<double> coeffprime;      // prime coefficients. Only needed when
multiplicity>1
    int found = 0;                  // No of roots founds
    vector<bool> NewtonTry(n, false);  // Vector flag indicates if last iteration
included a Newtonstep (multiplicity>1)

    // n>2 do the Durand-Kerner method
    // Algorithm only works on polynomials in monic form
    for (i = 0; i <= n; ++i)
        coeff[i] /= coeff[0];

    // Convergence test for the Durand-Kerner method
    // Max(W)< MIN(R[i]-R[j])/(2n+1), i=0..n-1, j=0..n-1 and i!=j
    auto TestConvergence = [](const vector<complex<double>>& R, const
vector<complex<double>>& W)
    {
        double wmax = 0.0;
        double dmin = numeric_limits<double>::infinity();
        const size_t n = W.size();  // W.size() and R.size() are identical

        // Compute dmin and wmax
        for (size_t i = 0; i < n; i++)
        {
            wmax = max(wmax, abs(W[i]));
            for (size_t j = i+1; j < n; ++j)
                    dmin = std::min(dmin, abs(R[i] - R[j]));
        }
        bool isConvergent = (wmax < dmin / (2 * n + 1));
        return isConvergent;
    };

    // Try multiple Newton Steps when dealing with multiple roots
    // If there is r multiple roots then the best next step is m=r
    auto MultipleNewtonSteps = [&](const vector<double>& a, const complex<double>& z,
const complex<double>& dz)
    {eval pzbest = horner(a, z - dz);
    for (int m = 1; m < n - 1; ++m)
    {
        eval pztry = horner(a, z - complex<double>(m + 1) * dz);
        if (pztry.apz >= pzbest.apz)
            break; // no improvement
        pzbest = pztry;
    }
    return pzbest;
    };

    // Set fixed initial start value for each root as an inward spiral from 1
    z = complex<double>(0.4, 0.9);
    for (i = 0; i < n; i++)
    {
        roots.push_back(z * pow(z, i));
        W[i]=roots.back();
        pz.push_back(horner(coeff, roots.back()));
    }

    if (stage1 == TestConvergence(roots, W))
        stage1 = !stage1;

    // Start iteration and stop when all roots have been found
    for (itercnt = 1; found<n && itercnt < MAX_ITER; itercnt++)
    {
        double q;                // Convergence power q
        complex<double> prevW;  // Used for q calculation
        for ( i = 0; i < n; ++i)  // Improve each root if not flagged as finish
        {
            if (finish[i] == true)
                continue;
            bool qcheck = false;
            // Calculate new root approximation
            complex<double> w(1);
            z= roots[i];        // Current root to improve

            // Calculate the Weierstrass modification
            for (int j = 0; j < n; ++j)
            {
                if (i != j)
                    w *= (z - roots[j]);
            }

            prevW = W[i];          // Save previous W[i]
            W[i] = pz[i].pz/(w);  // Compute new W[i]=P(z)/(w)

            // New root
            roots[i] -= W[i];
            pz[i] = horner(coeff, roots[i]);  // Calculate P(roots[i])
            // Check if we need to recalculate a precise upperbound e.g. Adams test
            if (pz[i].apz < epsLow || stage1 == false)
            {   // Calculate final upper bound for the root now that we are getting
close
                // or convergence test terminates stage1
                upper[i] = upperbound(coeff, z);
                qcheck = true; // set to trick the test of the Newton option
            }
            // Calculate convergence power q
            q = log(abs(W[i])) / log(abs(prevW));
            // When q<1.4 we need to check for multiplicity>1 using the Newton method
unless stopping criteria have already been fulfilled
            // Be aware after one Newton Step the q reflects the convergence power of
Newton not the Weierstrass step
            // This can lead to a false not needed Newton try step.
            if (z - W[i] != z && pz[i].apz >= upper[i] && qcheck && (abs(q) < 1.4 ||
NewtonTry[i]))
            {
                if (coeffprime.size() == 0)
                    // Calculate coefficients of p'(x), only once if needed
                    for (int j = 0; j < n; ++j)
                        coeffprime.push_back(coeff[j] * double(n - j));
                NewtonTry[i] = false;
                eval p1z = horner(coeffprime, z), p0z = horner(coeff, z);
                eval pzbest = MultipleNewtonSteps(coeff, z, p0z.pz / p1z.pz);
                if (pzbest.apz < pz[i].apz)
                {  // Newton gave an improvement for multiplicity>1
                    roots[i] = pzbest.z;
                    pz[i] = pzbest;
                    W[i] = pzbest.z - z;  // Save the step size
                    NewtonTry[i] = true;
                }
            }
            else
                NewtonTry[i] = false;

            if (z - W[i] == z || pz[i].apz < upper[i])
            {   // Root is found
                eval pz0;
                // flag current root as finish
                finish[i] = true;
                ++found;
                // Finalize root
                z = abs(z.real())>=abs(z.imag())? complex<double>(z.real(),0):
complex<double>(0,z.imag());
                pz0 = horner(coeff, z);
                if (pz0.apz <= pz[i].apz)
                {
                    pz[i] = pz0;
                    roots[i] = z;
                }
            }
        }

        if (stage1 == TestConvergence(roots, W))
            stage1 = !stage1;
    }

    return roots;
}
```

Example 1.

Here is an example of how the above source code is working.

```
For the real Polynomial:
+1x^4-10x^3+35x^2-50x+24
Start Durand-Kerner Iteration for Polynomial=+1x^4-10x^3+35x^2-50x+24
        Initial Stop Condition. |f(z)|<2.13e-14
        Start:
        z[1]=1 dz=1.00e+0 |f(z)|=0.0e+0
        z[2]=(0.4+i0.9) dz=(4.00e-1+i9.00e-1) |f(z)|=2.0e+1
        z[3]=(-0.7+i0.7) dz=(-6.50e-1+i7.20e-1) |f(z)|=8.7e+1
        z[4]=(-0.9-i0.3) dz=(-9.08e-1-i2.97e-1) |f(z)|=1.1e+2

Iteration: 1
        Stop Criteria satisfied after 1 Iteration
        Found a root z[1]=1 |f(x)|=0.00e+0
        Alteration=0%
        z[2]=(-4-i8) dz=(4.14e+0+i9.02e+0) |f(z)|=1.1e+4
        z[3]=(-6+i0.9) dz=(4.89e+0-i1.40e-1) |f(z)|=4.1e+3
        z[4]=(0.1-i1) dz=(-1.05e+0+i9.56e-1) |f(z)|=4.3e+1

Iteration: 2
        z[2]=(1e+1-i1e+1) dz=(-1.60e+1+i2.97e+0) |f(z)|=4.8e+4
        z[3]=(-2+i4) dz=(-3.84e+0-i2.88e+0) |f(z)|=9.9e+2
        z[4]=(0.5-i1) dz=(-3.41e-1-i2.60e-2) |f(z)|=2.7e+1

Iteration: 3
        z[2]=(8-i2) dz=(4.25e+0-i8.67e+0) |f(z)|=1.3e+3
        z[3]=(2+i3) dz=(-3.27e+0+i1.09e+0) |f(z)|=7.8e+1
        z[4]=(1-i1) dz=(-6.28e-1-i2.13e-1) |f(z)|=9.1e+0

Iteration: 4
```

```
          z[2]=(5.5-i1.0) dz=(2.58e+0-i1.42e+0) |f(z)|=7.7e+1
          z[3]=(3+i2) dz=(-9.58e-1+i1.12e+0) |f(z)|=1.2e+1
          z[4]=(2-i0.6) dz=(-5.74e-1-i4.04e-1) |f(z)|=2.2e+0

Iteration: 5
          z[2]=(4.4-i0.49) dz=(1.03e+0-i5.02e-1) |f(z)|=8.4e+0
          z[3]=(3+i0.7) dz=(-3.01e-1+i8.04e-1) |f(z)|=2.2e+0
          z[4]=(2-i0.2) dz=(-3.17e-1-i3.82e-1) |f(z)|=4.8e-1

Iteration: 6
          z[2]=(4.0-i0.20) dz=(4.00e-1-i2.95e-1) |f(z)|=1.3e+0
          z[3]=(3+i0.1) dz=(-1.18e-1+i5.77e-1) |f(z)|=3.1e-1
          z[4]=(2.0+i0.00025) dz=(-1.28e-2-i2.28e-1) |f(z)|=2.3e-2

Iteration: 7
          z[2]=(3.98-i0.0185) dz=(4.49e-2-i1.81e-1) |f(z)|=1.6e-1
          z[3]=(3.0-i0.0031) dz=(-5.83e-2+i1.48e-1) |f(z)|=1.1e-2
          z[4]=(2.00-i0.000149) dz=(1.18e-2+i3.95e-4) |f(z)|=3.2e-4

Iteration: 8
          z[2]=(4.000+i0.00001949) dz=(-1.97e-2-i1.85e-2) |f(z)|=8.7e-4
          z[3]=(3.00+i8.52e-8) dz=(4.26e-3-i3.06e-3) |f(z)|=2.8e-6
          z[4]=(2.0000-i1.1529e-8) dz=(-6.13e-5-i1.49e-4) |f(z)|=2.4e-8

Iteration: 9
          Stop Criteria satisfied after 9 Iterations
          Found a root z[4]=1.9999999999999971 |f(x)|=2.40e-18
          Alteration=0%
          z[2]=(4.00000-i3.84657e-11) dz=(1.44e-4+i1.95e-5) |f(z)|=1.2e-9
          z[3]=(3.000000+i1.619904e-14) dz=(1.39e-6+i8.52e-8) |f(z)|=3.3e-14

Iteration: 10
          Stop Criteria satisfied after 10 Iterations
          Found a root z[2]=3.9999999999999964 |f(x)|=2.13e-14
          Alteration=0%
          Stop Criteria satisfied after 10 Iterations
          Found a root z[3]=2.9999999999999947 |f(x)|=6.31e-30
          Alteration=0%

Using the Durand-Kerner Method, the Solutions are:
X1=1.9999999999999971
X2=2.9999999999999947
X3=3.9999999999999964
X4=1
```

Example 2.

```
For the real Polynomial:
+1x^4-8x^3-17x^2-26x-40
Start Durand-Kerner Iteration for Polynomial=+1x^4-8x^3-17x^2-26x-40
          Initial Stop Condition. |f(z)|<3.55e-14
          Start:
```

```
        z[1]=1 dz=1.00e+0 |f(z)|=9.0e+1
        z[2]=(0.4+i0.9) dz=(4.00e-1+i9.00e-1) |f(z)|=4.7e+1
        z[3]=(-0.7+i0.7) dz=(-6.50e-1+i7.20e-1) |f(z)|=2.9e+1
        z[4]=(-0.9-i0.3) dz=(-9.08e-1-i2.97e-1) |f(z)|=2.5e+1

Iteration: 1
        z[1]=(9+i2e+1) dz=(-7.78e+0-i2.26e+1) |f(z)|=3.3e+5
        z[2]=(0.1+i2) dz=(2.88e-1-i1.03e+0) |f(z)|=4.4e+1
        z[3]=(-1+i0.8) dz=(8.08e-1-i9.07e-2) |f(z)|=3.8e+1
        z[4]=(-0.9-i0.6) dz=(-5.56e-2+i3.28e-1) |f(z)|=2.9e+1

Iteration: 2
        z[1]=(1e+1-i2) dz=(-2.29e+0+i2.44e+1) |f(z)|=3.5e+3
        z[2]=(0.1+i1) dz=(1.69e-2+i7.20e-1) |f(z)|=2.5e+1
        z[3]=(-1-i0.3) dz=(-3.15e-1+i1.15e+0) |f(z)|=2.2e+1
        z[4]=(2-i2) dz=(-2.50e+0+i1.34e+0) |f(z)|=2.4e+2

Iteration: 3
        z[1]=(9.9+i0.61) dz=(1.18e+0-i2.36e+0) |f(z)|=7.4e+2
        z[2]=(0.2+i2) dz=(-8.43e-2-i3.61e-1) |f(z)|=2.7e+1
        z[3]=(-1.3-i0.13) dz=(1.72e-1-i2.07e-1) |f(z)|=1.5e+1
        z[4]=(-0.4-i2) dz=(2.08e+0-i4.07e-1) |f(z)|=1.7e+1

Iteration: 4
        z[1]=(10-i0.025) dz=(-1.12e-1+i6.33e-1) |f(z)|=3.1e+1
        z[2]=(-0.2+i2) dz=(3.73e-1+i1.23e-3) |f(z)|=1.9e+0
        z[3]=(-2-i0.03) dz=(3.76e-1-i9.92e-2) |f(z)|=2.8e+0
        z[4]=(-0.18-i1.6) dz=(-2.53e-1-i1.56e-3) |f(z)|=5.2e-1

Iteration: 5
        z[1]=(10.00-i0.0001396) dz=(2.37e-3-i2.50e-2) |f(z)|=1.8e-1
        z[2]=(-0.175+i1.55) dz=(-1.77e-2+i2.03e-2) |f(z)|=4.8e-2
        z[3]=(-1.7+i0.000043) dz=(-3.97e-2-i3.07e-2) |f(z)|=9.0e-3
        z[4]=(-0.175-i1.55) dz=(-1.54e-3-i7.45e-3) |f(z)|=1.6e-4

Iteration: 6
        z[1]=(10.0000-i1.10034e-8) dz=(5.34e-5-i1.40e-4) |f(z)|=1.5e-5
        z[2]=(-0.1747+i1.547) dz=(-3.62e-4+i5.99e-4) |f(z)|=3.8e-6
        z[3]=(-1.6506+i1.1970e-10) dz=(-1.63e-4+i4.35e-5) |f(z)|=9.8e-9
        z[4]=(-0.1746854-i1.546869) dz=(6.66e-7-i2.18e-6) |f(z)|=2.8e-12

Iteration: 7
        Stop Criteria satisfied after 7 Iterations
        Found a root z[2]=(-0.17468540428030596+i1.5468688872313963) |f(x)|=7.11e-15
        Alteration=0%
        Stop Criteria satisfied after 7 Iterations
        Found a root z[3]=-1.6506291914393882 |f(x)|=2.13e-14
        Alteration=0%
        Stop Criteria satisfied after 7 Iterations
        Found a root z[4]=(-0.17468540428030596-i1.5468688872313963) |f(x)|=7.11e-15
        Alteration=0%
        z[1]=(10.00000000+i4.467037524e-17) dz=(-4.71e-9-i1.10e-8) |f(z)|=5.5e-12
```

```
Iteration: 8
        Stop Criteria satisfied after 8 Iterations
        Found a root z[1]=10 |f(x)|=1.90e-28
        Alteration=0%

Using the Durand-Kerner Method, the Solutions are:
X1=(-0.17468540428030596-i1.5468688872313963)
X2=-1.6506291914393882
X3=(-0.17468540428030596+i1.5468688872313963)
X4=10
```

Other Weierstrass-like simultaneous methods without the use of the derivatives

There are a few spin-off methods from the Durand-Kerner. There is Tanabe's [12] third-order method:

$$x_i^{(k+1)} = x_i^{(k)} - W_i^{(k)}\left(1 - \sum_{j=1, j\neq i}^{n} \frac{W_j^{(k)}}{x_i^{(k)} - x_j^{(k)}}\right) \quad i = 1, \dots, n \text{ and } k = 0,1, \dots$$

Which offers a third-order convergence rate.

And there is Nourein [12] a fourth-order method also based on the Wierstrass corrections:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{W_i^{(k)}}{1 + \sum_{j=1, j\neq i}^{n}\left(\frac{W_j^{(k)}}{x_i^{(k)} - W_i^{(k)} - x_i^{(k)}}\right)}$$

With the Durand-Kerner present here it is relatively simple to implement the two spins-off method.

Aberth-Erhlich simultaneous method

Another closely related method is the Aberth-Erhlich from 1967 [13]. This method also finds all the roots simultaneously however requires the use of the Polynomial first derivate P'(x)

It is a very robust method and has been implemented in the MPSolve software package. It is a third-order convergence method although it only approaches roots with multiplicity greater than one with linear convergence. Aberth-Ehrlich efficiency index is $3^{\frac{1}{2}} = 1.73$

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\frac{P\left(x_i^{(k)}\right)}{p'\left(x_i^{(k)}\right)}}{1 - \frac{P\left(x_i^{(k)}\right)}{p'\left(x_i^{(k)}\right)} \sum_{j=1, j\neq i}^{n} \frac{1}{\left(x_i^{(k)} - x_j^{(k)}\right)}} \quad i = 1, \dots, n\,; k = 0, 1, \dots$$

Aberth in his original paper [13] also describes suitable starting points for all roots.

Recommendation

Although the efficiency index is higher for the Durand-Kerner than for Newton's method it is generally not recommended to use the modified Durand-Kerner method over the Newton's method. The reason in my viewpoint is that the modified Newton method presented in part One and part two is usually much faster to get a tracking to a root and therefore use fewer iterations than Durand-Kerner. I recommend sticking with the Newton method presented in Parts One and Two. However, if you decide to use the Durand-Kerner it is still a solid choice of a good and efficient polynomial root finder method.

Conclusion

We have presented a refined Durand-Kerner method, building upon the framework established in part one and part two, despite the Durand-Kerner being quite different. Part Six will demonstrate the ease of integrating alternative multi-point methods. A web-based polynomial solver showcasing these various methods is available for further exploration and can be found on Polynomial roots that demonstrate many of these methods in action.

Reference

1. H. Vestermark. A practical implementation of Polynomial root finders. Practical implementation of Polynomial root finders vs 7.docx (www.hvks.com)
2. Madsen. A root-finding algorithm based on Newton Method, Bit 13 (1973) 71-75.
3. A. Ostrowski, Solution of equations and systems of equations, Academic Press, 1966.
4. Wikipedia Horner's Method: https://en.wikipedia.org/wiki/Horner%27s_method
5. Adams, D A stopping criterion for polynomial root finding.
   Communication of the ACM Volume 10/Number 10/ October 1967 Page 655-658
6. Grant, J. A. & Hitchins, G D. Two algorithms for the solution of polynomial equations to limiting machine precision. The Computer Journal Volume 18 Number 3, pages 258-264
7. Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood Cliffs, NJ 1963
8. McNamee, J.M., Numerical Methods for Roots of Polynomials, Part I & II, Elsevier, Kidlington, Oxford 2009

9. H. Vestermark, “A Modified Newton and higher orders Iteration for multiple roots.”, www.hvks.com/Numerical/papers.html
10. M.A. Jenkins & J.F. Traub, ”A three-stage Algorithm for Real Polynomials using Quadratic iteration”, SIAM J Numerical Analysis, Vol. 7, No.4, December 1970.
11. H.A. Yamani, A.D. Alhaidari, Iterative polynomial-root-finding procedure with enhanced accuracy, Saudi Center for Theoretical Physics, Saudi Arabia.
12. M. Petkovic, D. Herceg, S.Ilic, Safe convergence of simultaneous methods for polynomial zeros. Numeric Algorithm 17 (1998) 313-3312
13. O. Aberth, Iteration Methods for finding all zeros of a Polynomial Simultaneously, Mathematics Computation, Vol 27, Number 122, April 1973