Fast Polynomial Root Finder - Part Four.

By Henrik Vestermark (hve@hvks.com)

Abstract:

We elaborated in the part Four paper on the Laguerre method for finding Polynomial roots and devised a modified version dealing efficiently with Polynomials with real coefficients. This paper is part of a multi-series of papers on how to use the same framework to implement different root finder methods.

Introduction:

In the first paper (part one), we developed a highly efficient and robust polynomial root-finder based on the Newton method, specifically designed for complex polynomial coefficients. In part two we elaborated on the change to dealing with Polynomials with real coefficients. In part three we looked at using the same framework to implement higher-order methods (Halley) and discussed if we gain any advantages from using higher-order methods compared to the standard Newton method. Part Four we look at the Laguerre method and in Part Five we dwell into one of the simultaneous methods e.g., the Durand-Kerner a completely different method however we will see how we can still fairly easily map it into the existing framework established in parts One, Two, and Three.

Fast Polynomial Root Finder - Part Four

Contents

Fast Polynomial Root Finder - Part Four. 1
Abstract: 1
Introduction: 1
Why Laguerre's Method? 2
Laguerre's Method: An Efficient Technique for Polynomial Root Approximation 3
Comparing Laguerre and Halley? 5
What to Modify? 5
The Implementation of the Laguerre Method 6
The C++ code 7
Example 1. 14
Example 2. 16
Example 3. 17
Recommendation 19
Conclusion 20
Reference 20

Why Laguerre's Method?

Before writing this article, I carefully considered selecting a method that diverged from the previous parts of the series. I chose Laguerre's method for several compelling reasons, particularly its underappreciated efficiency when compared to other root-finding methods. Laguerre's method excels with higher-degree polynomials, making it an outstanding choice. Unlike many other methods that struggle with accuracy and convergence as polynomial degrees increase, Laguerre's method remains robust and effective, even for complex, high-degree polynomials.

Its superior convergence properties are noteworthy, especially when the initial guess is reasonably close to a true root. This rapid convergence owes itself in part to the method's utilization of both the first and second derivatives of the polynomial in its iterative formula. Additionally, Laguerre's method stands out for its ability to handle complex roots adeptly. This is a crucial feature since many higher-degree polynomials possess complex roots. It adeptly uncovers both real and complex roots.

Moreover, unlike other methods that might falter or converge to incorrect roots when the initial guess is less precise, Laguerre's method boasts strong global convergence characteristics. This means it's more likely to converge to a root from a wider range of starting points.

Fast Polynomial Root Finder - Part Four

In practical scenarios, especially within engineering and the physical sciences where polynomial equations are prevalent, Laguerre's method emerges as a reliable and efficient tool for tackling complex problems.

Laguerre's Method: An Efficient Technique for Polynomial Root Approximation

Laguerre's method is a powerful numerical approach employed to approximate the roots of polynomial equations. It exhibits remarkable effectiveness, especially when dealing with polynomials possessing complex roots, often outperforming other root-finding methods, including the well-known Newton's method. This method, originally introduced by Laguerre in 1898, is a significant contribution to numerical analysis, as documented in references like McNamee [8] and [11].

One distinguishing feature of Laguerre's method is its utilization of both the first and second derivatives of the polynomial function P(x). This incorporation of derivative information results in third-order convergence, which aligns with the convergence rate of the Halley method, as described in Part Three. The Polynomial efficient index, denoted as $3^{\frac{1}{3}} = 1.44$, is shared between Laguerre's method and the Halley method. The Laguerre method is:

$$x_{n+1} = x_n - \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}}$$

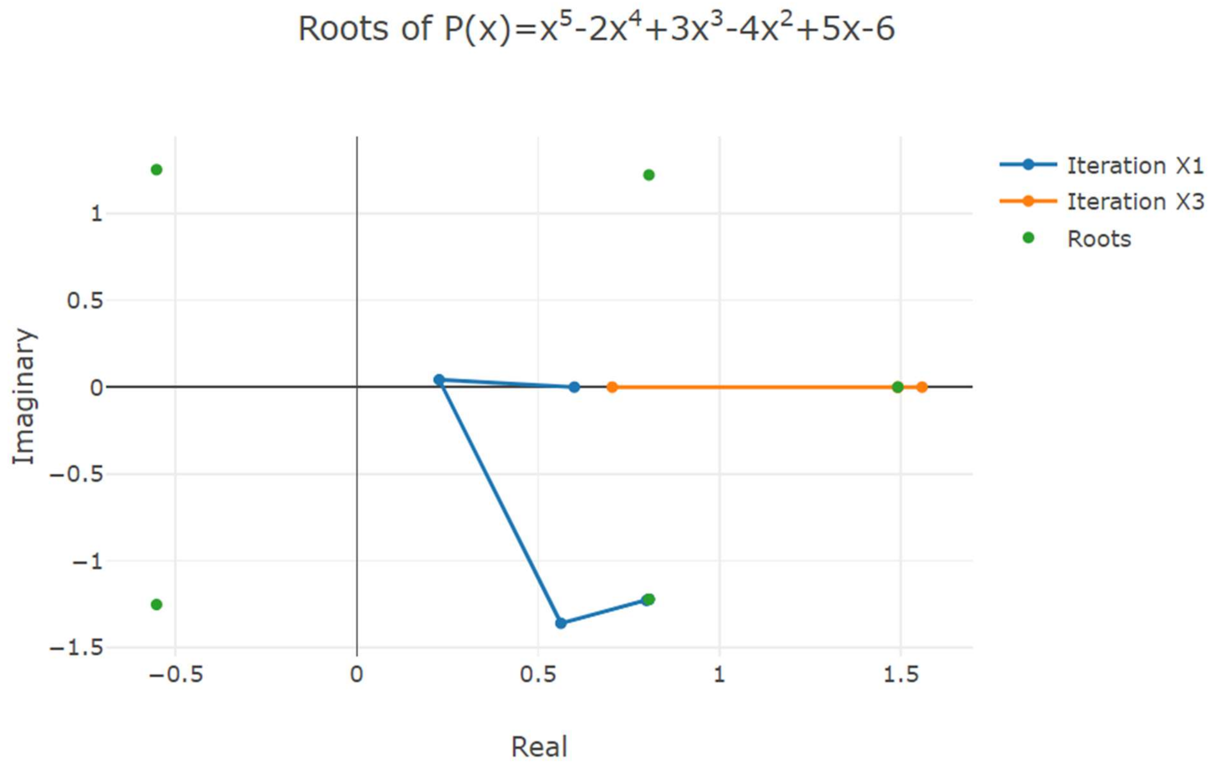
The sign ± is chosen to maximize the absolute value of the denominator and

$$G = \frac{p'(x_n)}{p(x_n)} \; and \; H = G^2 - \frac{p''(x_n)}{p(x_n)}$$

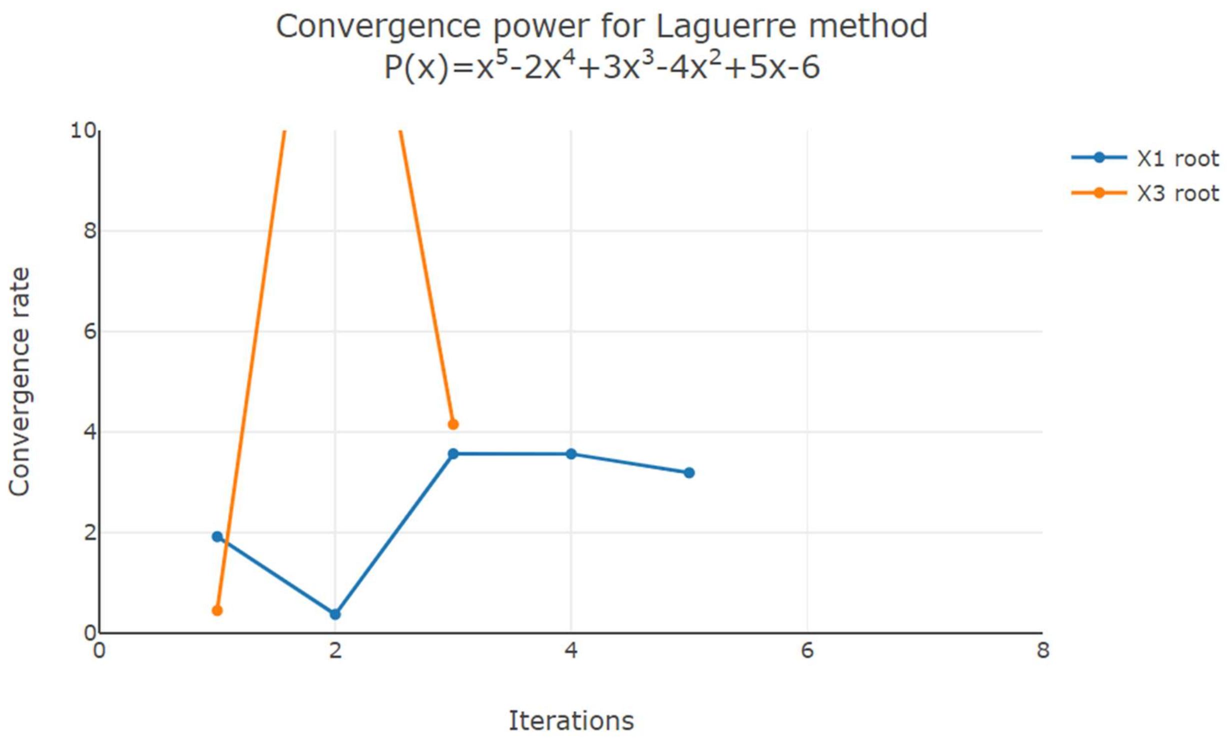
Where P(x) is the Polynomial, whose root is to be found, P'(x) is the first derivative, and P''(x) is the second derivative of the Polynomial. We notice that the $P(x_n)$ is in the denominator of G and H and is never zero otherwise we would have stopped the search based on this criterion. The method has the advantage of global convergence for most functions, but it's computationally more intensive than other methods due to the requirement of calculating both first and second derivatives.

Laguerre's method has a third-order rate of convergence for simple roots. This means that the number of accurate digits in the approximation roughly triples with each iteration, making it a very efficient method for finding the roots of a polynomial. However, the rate of convergence may be different for multiple roots. The third-order convergence makes Laguerre's method faster than many other root-finding techniques, such as Newton's method, which has a second-order rate of convergence for simple roots.

Here is an example of



The first root ends up as a complex conjugated root x=(0.80-i1.22). and the third root is just a simple real root at ~ 1.5. The remaining two roots are found directly from the deflated polynomial.



The above picture shows the convergence rate while approaching the roots and is in line with expectations.

As for all the root finder methods we have shown so far all are reduced to linear convergence for multiple roots (multiplicity > 1). Again, as for the Newton and Halley method, there exists a modified version of Laguerre's method that maintains the third-order convergence rate even for multiple roots.

The modified version is

$$x_{n+1} = x_n - \frac{n}{G \pm \sqrt{\left(\frac{n}{m} - 1\right)(nH - G^2)}}$$

Where m is the multiplicity of the root.
Most often you do not know *m* beforehand but you can use the same technique as presented in Part 1-3 (see the detailed description of the Newton method) where we continue using the below formula for m=2 up to n as long as for each m the $P(x_{n+1}^{m}) < P(x_{n+1}^{m-1})$

The modified Laguerre works very well for both Polynomials with real or complex coefficients and is a very stable method for finding Polynomial zeros.

Comparing Laguerre and Newton?

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 Laguerre's method, we need to evaluate P(x), P'(x), and P''(x) for each iteration, we get $3^{\frac{1}{3}} = 1.44$
Slightly larger than the Newton method but not enough that we should expect any measurable gain from using Laguerre's 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.

From Part Two, the Steps Include:

1. Finding an initial point

2. Executing the Laguerre iteration, including polynomial evaluation via the Horner method
3. Calculating the final upper bound
4. Polynomial deflation
5. Solving the quadratic equation

Ad 1,3,4,5) Will be identical to the Newton method and need no modification

Ad 2) We can use the Horned method unchanged to evaluate P(x), P'(x), and P''(x). However, we need to add another vector to hold the second derivative P''(x). The variable step size to handle multiple roots can be changed from m to the formula listed previously. Otherwise, we can again reuse the variable step size or reduce the step size and show it in both parts one and two.

The Implementation of the Laguerre Method

This is the same source as in parts two and three except for the change needed to evaluate Laguerre's iteration step instead of the Newton or Halley step.

The implementation of this root finder follows the method as layout in Part One.

1) First, we eliminate simple roots (roots equal to zero)
2) Then we find a suitable starting point to start our Laguerre Iteration, this also includes termination criteria based on an acceptable value of P(x) where we will stop the current iteration.
3) Start the Laguerre iteration
    a. The first step is to find the $dx_n=\frac{n}{G\pm\sqrt{(n-1)(nH-G^2)}}$ and of course, decide what should happen if the denominator is zero. The sign in the denominator is chosen to maximize the absolute value of the denominator. When this condition arises, it is most often due to a local minimum and the best course of action is to alter the direction with a factor $dx_n=dx_n(0.6+i0.8)k$. This is equivalent to rotating the direction with an odd degree of 53 degrees and multiplying the direction with the factor k. A suitable value for k=5 is reasonable when this happens.
    b. Furthermore, it is easy to realize that if $P'(x_n)\sim 0$. You could get some unreasonable step size of dx_n and therefore introduce a limiting factor that reduces the current step size if $abs(dx_n)>5\cdot abs(dx_{n-1})$ than the previous iteration's step size. Again, you alter the direction with $dx_n=dx_n(0.6+i0.8)5(abs(dx_{n-1})/abs(dx_n))$ if that happens.
    c. These two modifications (a and b) make his method very resilient and make it always converge to a root.
    d. The next issue is to handle the issue with multiplicity > 1 which will slow the third-order convergence rate down to a linear convergence rate. After a suitable dx_n is found and a new $x_{n+1}=x_n-dx_n$ we then look to see if $P(x_{n+1})>P(x_n)$: If so we look at a revised $x_{n+1}=x_n-0.5dx_n$ and if $P(x_{n+1})\geq P(x_n)$ then he used the original x_{n+1} as the new starting point for the next iteration. If not then we accept x_{n+1} as a better choice and continue looking at a newly revised $x_{n+1}=x_n-0.25dx_n$. If on the other hand the new $P(x_{n+1})\geq P(x_n)$ we used the previous x_{n+1} as a new starting point for the next iterations. If not then we assume we are nearing a new saddle point and the direction is altered

with $dx_n=dx_n(0.6+i0.8)$ and we use $x_{n+1}=x_n-dx_n$ as the new starting point for the next iteration.
if on the other hand $P(x_{n+1})\leq P(x_n)$: Then we are looking in the right direction and we then continue stepping in that direction using $x_{n+1}=x_n-\frac{n}{G\pm\sqrt{\left(\frac{n}{m}-1\right)(nH-G^2)}}$ m=2,..,n as long as $P(x_{n+1})\leq P(x_n)$ and use the best m for the next iterations. The benefit of this process is that if there is a root with the multiplicity of m then m will also be the best choice for the stepping size and this will maintain the third-order convergence rate even for multiple roots.

4) Processes a-d continue until the stopping criteria are reached where after the root x_n is accepted and deflated up in the Polynomial. A new search for a root using the deflated Polynomial is initiated.

We divide the iterations into two stages. Stage 1 & Stage 2. In stage 1 we are trying to get into the convergence circle where we are sure that Laguerre's method will converge towards a root. Since this is a different method, we can't use the same computation as the Newton method. However, we will do it anyway since in practice it works pretty well. When we get into that circle, we automatically switch to stage 2. In stage 2 we skip step d) and just use an unmodified Laguerre step: $x_{n+1}=x_n-\frac{n}{G\pm\sqrt{(n-1)(nH-G^2)}}$ until the stopping criteria have been satisfied. In case we get outside the convergence circle, we switch back to stage 1 and continue the iteration. We use the same criteria to switch to stage 2 as we did for both the Newton and Halley methods.

Now we have everything we need to determine when to switch to stage 2.

The C++ code

The C++ code below finds the Polynomial roots with Polynomial real coefficients using Laguerre's method. There are only very few changes made from the Newton version to implement Laguerre's 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     :   Laguerre.cpp
 * Module ID Nbr   :
 * Description     :   Solve n degree polynomial using Laguerre's method
 * --------------------------------------------------------------------------
 * Change Record   :
 *
 * Version    Author/Date         Description of changes
 * -------  -------------   ----------------------
 * 01.01      HVE/24Sep2023 Initial release
 *
 * End of Change Record
 * --------------------------------------------------------------------------
*/

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

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

//#include "../intervalprecision.h"

using namespace std;

constexpr int       MAX_ITER = 50;
// Find all polynomial zeros using a modified Laguerre method
// 1) Eliminate all simple roots (roots equal to zero)
// 2) Find a suitable starting point
// 3) Find a root using the Laguerre method
// 4) Divide the root up in the polynomial reducing its order with one
// 5) Repeat steps 2 to 4 until the polynomial is of the order of two whereafter the
remaining one or two roots are found by the direct formula
// Notice:
//      The coefficients for p(x) is stored in descending order. coefficients[0] is
a(n)x^n, coefficients[1] is a(n-1)x^(n-1),...,  coefficients[n-1] is a(1)x,
coefficients[n] is a(0)
//
static vector<complex<double>> PolynomialRootsLaguerre(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;       // Size of Polynomial p(x)
    eval pz;        // P(z)
    eval pzprev;    // P(zprev)
    eval p1z;       // P'(z)
    eval p1zprev;   // P'(zprev)
    complex<double> z;      // Use as temporary variable
    complex<double> dz;     // The current stepsize dz
    complex<double> newtondz;
    int itercnt;    // Hold the number of iterations per root
    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 = coeff.size() - 1; n > 0 && coeff.back() == 0.0; --n)
```

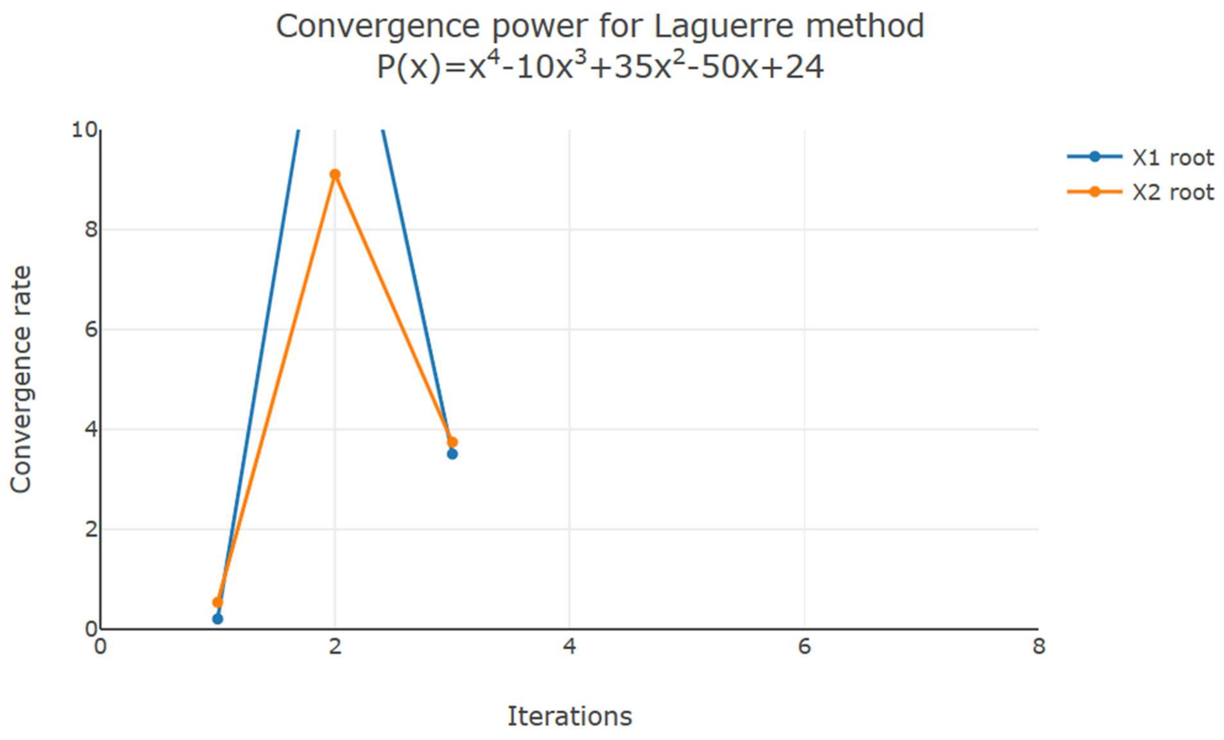
```
        roots.push_back(complexzero);  // Store zero as the root

    // Compute the next starting point based on the polynomial coefficients
    // A root will always be outside the circle from the origin and radius min
    auto startpoint = [&](const vector<double>& a)
    {
        const size_t n = a.size() - 1;
        double a0 = log(abs(a.back()));
        double min = exp((a0 - log(abs(a.front()))) / static_cast<double>(n));

        for (size_t i = 1; i < n; i++)
            if (a[i] != 0.0)
            {
                double tmp = exp((a0 - log(abs(a[i]))) / static_cast<double>(n - i));
                if (tmp < min)
                    min = tmp;
            }

        return min * 0.5;
    };

    // Evaluate a polynomial with real coefficients a[] at a complex point z and
    // return the result
    // This is Horner's method, 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 Laguerre iteration for polynomial order higher than 2
    for (; n > 2; --n)
    {
        const double Max_stepsize = 5.0; // Allow the next step size to be up to 5 times
larger than the previous step size
        const complex<double> rotation = complex<double>(0.6, 0.8);  // Rotation amount
        double r;               // Current radius
        double rprev;           // Previous radius
        double eps;             // The iteration termination value
        bool stage1 = true;     // By default it start the iteration in stage1
        int steps = 1;          // Multisteps if > 1
        eval p2z;               // P''(z)
        vector<double> coeffprime;   // vector holding the prime coefficients
        vector<double> coeffprime2;  // Laguerre vector holding both the prime and
double prime coefficients

        // Calculate coefficients of p'(x)
        for (int i = 0; i < n; i++)
            coeffprime.push_back(coeff[i] * double(n - i));
        // Calculate coefficients of p''(x)
        for (int i = 0; i < n - 1; i++)       // Laguerre
            coeffprime2.push_back(coeffprime[i] * double(n - i - 1));   // Laguerre

        // Step 2 find a suitable starting point z
        rprev = startpoint(coeff);      // Computed startpoint
        z = coeff[n - 1] == 0.0 ? complex<double>(1.0) : complex<double>(-coeff[n] /
coeff[n - 1]);
        z *= complex<double>(rprev) / abs(z);

        // Setup the iteration
        // Current P(z)
        pz = horner(coeff, z);

        // pzprev which is the previous z or P(0)
        pzprev.z = complex<double>(0);
        pzprev.pz = coeff[n];
        pzprev.apz = abs(pzprev.pz);

        // p1zprev P'(0) is the P'(0)
        p1zprev.z = pzprev.z;
        p1zprev.pz = coeff[n - 1];       // P'(0)
        p1zprev.apz = abs(p1zprev.pz);

        // Set previous dz and calculate the radius of operations.
        dz = pz.z;      // dz=z-zprev=z since zprev==0
        r = rprev *= Max_stepsize; // Make a reasonable radius of the maximum step size
allowed
        // Preliminary eps computed at point P(0) by a crude estimation
        eps = 2 * n * pzprev.apz * pow((double)_DBL_RADIX, -DBL_MANT_DIG);

        // Start iteration and stop if z doesnt change or apz <= eps
        // we do z+dz!=z instead of dz!=0. if dz does not change z then we accept z as a
root
        for (itercnt = 0; pz.z + dz != pz.z && pz.apz > eps && itercnt < MAX_ITER;
itercnt++)
        {
            complex<double> G, H, gp, gm, u;

            // Calculate current P'(z) and P''(z)
            p1z = horner(coeffprime, pz.z);
            p2z = horner(coeffprime2, pz.z);
            // Compute G and H
            G = p1z.pz / pz.pz;
            H = G * G - p2z.pz / pz.pz;
            H = (complex<double>(static_cast<int>(n)) * H - G * G); // Save H for
later=nH-G^2
            u = sqrt(complex<double>(static_cast<int>(n) - 1) * H);
            gp = G + u;
            gm = G - u;
            if (abs(gp) < abs(gm))
                gp = gm;
            // Calculate dz, change directions if zero
            if (abs(gp) == 0.0)                 // If Laguerre denominator is zero then
rotate previous dz direction
                dz *= rotation * complex<double>(Max_stepsize);
            else
                dz = complex<double>(static_cast<int>(n)) / gp;

            // Check the Magnitude of Laguerre's step
             r = abs(dz);
             if (r > rprev) // Large than 5 times the previous step size
             {   // then rotate and adjust step size to prevent wild step size near
P'(z) close to zero
                 dz *= rotation * complex<double>(rprev / r);
                 r = abs(dz);
             }
             rprev = r * Max_stepsize;  // Save 5 times the current step size for the
next iteration check of reasonable step size

             // Calculate if stage1 is true or false. Stage1 is false if the
Newton/Laguerre converge otherwise true
             z = (p1zprev.pz - p1z.pz) / (pzprev.z - pz.z);
             stage1 = (abs(z) / p1z.apz > p1z.apz / pz.apz / 4) || (steps != 1);

            // Step accepted. Save pz in pzprev
            pzprev = pz;

            z = pzprev.z - dz;      // Next z
            pz = horner(coeff, z);  //ff = pz.apz;
            steps = 1;
            if (stage1)
            {  // Try multiple steps or shorten steps depending if P(z) is an
improvement or not P(z)<P(zprev)
                bool div2;
                complex<double> zn, dzn=dz;
                eval npz;
                steps++;
                for (div2 = pz.apz > pzprev.apz; steps <= n; ++steps)
                {
                    if (div2 == true)
                    {  // Shorten steps
                        dzn *= complex<double>(0.5);
                        zn = pzprev.z - dz;
                    }
                    else
                    {   // Compute new dz
                        dzn = sqrt(complex<double>(double(n)/double(steps) - 1) * H);
                        gp = G + dzn;
                        gm = G - dzn;
                        if (abs(gp) < abs(gm))
                            gp = gm;
                        dzn = complex<double>(static_cast<int>(n)) / gp;
                        zn = pzprev.pz - dzn;
                    }
                    // Evaluate new try step
                    npz = horner(coeff, zn);
                    if (npz.apz >= pz.apz)
                    {
                        --steps; break; // Break if no improvement
                    }

                    // Improved => accept step and try another round of step
                    pz = npz;
                    dz = dzn;

                    if (div2 == true && steps == 3)
                    {   // To many shorten steps => try another direction and break
                        dz *= rotation;
                        z = pzprev.z - dz;
                        pz = horner(coeff, z);
                        break;
                    }
                }
            }
            else
            {   // calculate the upper bound of error using Grant & Hitchins's test for
complex coefficients
                // Now that we are within the convergence circle.
                eps = upperbound(coeff, pz.z);
            }
        }

        // Real root forward deflation.
        //
        auto realdeflation = [&](vector<double>& a, const double x)
        {
            const size_t n = a.size() - 1;
            double r = 0.0;

            for (size_t i = 0; i < n; i++)
                a[i] = r = r * x + a[i];

            a.resize(n);    // Remove the highest degree coefficients.
        };

        // Complex root forward deflation for real coefficients
        //
        auto complexdeflation = [&](vector<double>& a, const complex<double> z)
        {
            const size_t n = a.size() - 1;
            double r = -2.0 * z.real();
            double u = norm(z);

            a[1] -= r * a[0];
            for (int i = 2; i < n - 1; i++)
                a[i] = a[i] - r * a[i - 1] - u * a[i - 2];

            a.resize(n - 1); // Remove top 2 highest degree coefficients
        };

        // Check if there is a very small residue in the imaginary part by trying
        // to evaluate P(z.real). if that is less than P(z). We take that z.real() is a
better root than z.
        z = complex<double>(pz.z.real(), 0.0);
        pzprev = horner(coeff, z);
        if (pzprev.apz <= pz.apz)
        { // real root
            pz = pzprev;
            // Save the root
            roots.push_back(pz.z);
            realdeflation(coeff, pz.z.real());
        }
        else
        {   // Complex root
            // Save the root
            roots.push_back(pz.z);
            roots.push_back(conj(pz.z));
            complexdeflation(coeff, pz.z);
            --n;
        }

    }   // End Iteration

    // 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;
    };

    if (n > 0)
        quadratic(coeff);

    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 Laguerre Iteration for Polynomial=+1x^4-10x^3+35x^2-50x+24
            Stage 1=>Stop Condition. |f(z)|<2.13e-14
            Start   : z[1]=0.2 dz=2.40e-1 |f(z)|=1.4e+1
Iteration: 1
            Laguerre Step:  z[1]=1 dz=-7.46e-1 |f(z)|=8.9e-2
            Function value decrease=>try multiple steps in that direction
            Try Step:  z[1]=1 dz=-7.46e-1 |f(z)|=8.1e-1
                  : No improvement. Discard the last try step
Iteration: 2
            Laguerre Step:  z[1]=1 dz=-1.45e-2 |f(z)|=2.1e-6
            In Stage 2. New Stop Condition: |f(z)|<2.18e-14
Iteration: 3
            Laguerre Step:  z[1]=1 dz=-3.53e-7 |f(z)|=1.1e-14
            In Stage 2. New Stop Condition: |f(z)|<2.18e-14
Stop Criteria satisfied after 3 Iterations
Final Laguerre  z[1]=1 dz=-3.53e-7 |f(z)|=1.1e-14
Alteration=0% Stage 1=33% Stage 2=67%
```

```
            Deflate the real root z=1.0000000000000009
Start Laguerre Iteration for Polynomial=+1x^3-9x^2+25.999999999999993x-23.999999999999986
            Stage 1=>Stop Condition. |f(z)|<1.60e-14
            Start    : z[1]=0.5 dz=4.62e-1 |f(z)|=1.4e+1
Iteration: 1
            Laguerre Step:  z[1]=2 dz=-1.52e+0 |f(z)|=4.7e-2
            Function value decrease=>try multiple steps in that direction
            Try Step:  z[1]=2 dz=-1.52e+0 |f(z)|=3.4e-1
                  : No improvement. Discard the last try step
Iteration: 2
            Laguerre Step:  z[1]=2 dz=-2.27e-2 |f(z)|=1.4e-6
            In Stage 2. New Stop Condition: |f(z)|<1.42e-14
Iteration: 3
            Laguerre Step:  z[1]=2 dz=-6.91e-7 |f(z)|=0
            In Stage 2. New Stop Condition: |f(z)|<1.42e-14
Stop Criteria satisfied after 3 Iterations
Final Laguerre  z[1]=2 dz=-6.91e-7 |f(z)|=0
Alteration=0% Stage 1=33% Stage 2=67%
            Deflate the real root z=2
Solve Polynomial=+1x^2-7x+11.999999999999993 directly
Using the Laguerre Method, the Solutions are:
X1=1.0000000000000009
X2=2
X3=4.000000000000007
X4=2.999999999999993
Time used: 1 msec. Solvable level: Easy
```

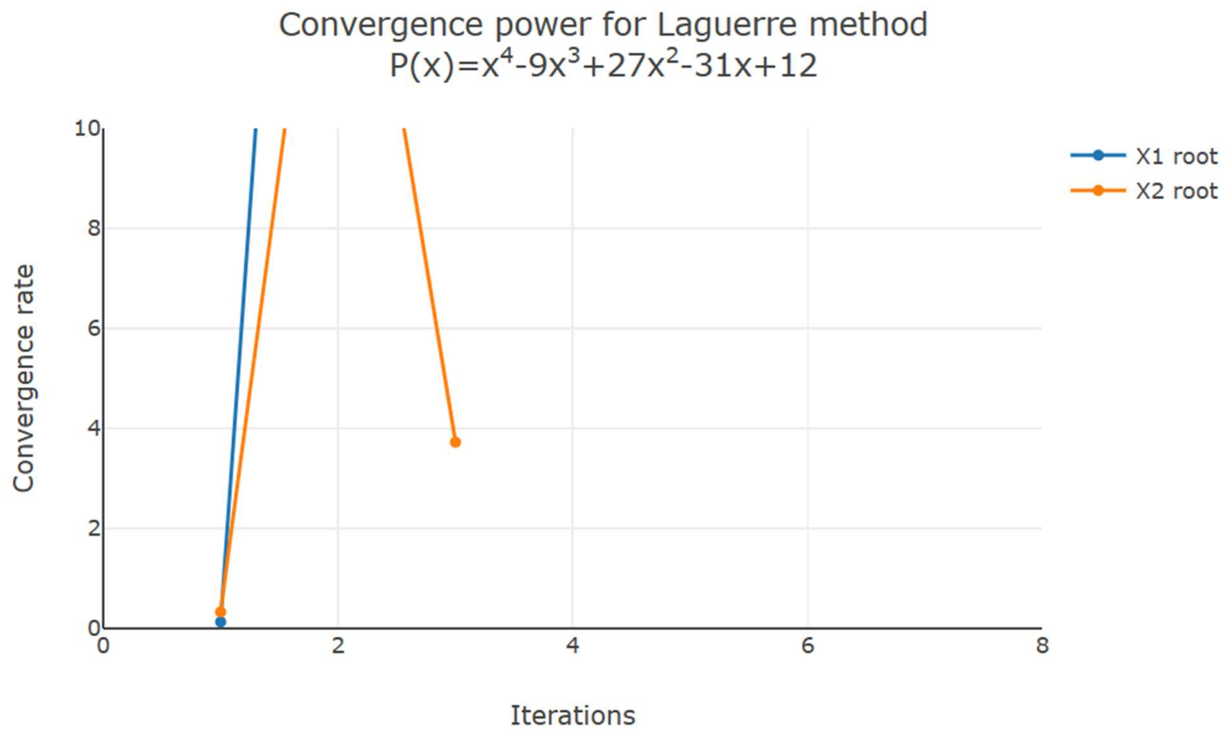


Example 2.

The same example just with a double root at x=1. We see that each step is a double step in line with a multiplicity of 2 for the first root.

```
For the real Polynomial:
+1x^4-9x^3+27x^2-31x+12
Start Laguerre Iteration for Polynomial=+1x^4-9x^3+27x^2-31x+12
            Stage 1=>Stop Condition. |f(z)|<1.07e-14
            Start    : z[1]=0.2 dz=1.94e-1 |f(z)|=6.9e+0
Iteration: 1
            Laguerre Step:  z[1]=0.8 dz=-6.32e-1 |f(z)|=2.1e-1
            Function value decrease=>try multiple steps in that direction
            Try Step:  z[1]=1 dz=-6.32e-1 |f(z)|=3.5e-6
                  : Improved. Continue stepping
            Try Step:  z[1]=1 dz=-6.32e-1 |f(z)|=1.2e-1
                  : No improvement. Discard the last try step
Iteration: 2
            Laguerre Step:  z[1]=1 dz=-5.59e-4 |f(z)|=2.5e-7
            Function value decrease=>try multiple steps in that direction
            Try Step:  z[1]=1 dz=-5.59e-4 |f(z)|=0
                  : Improved. Continue stepping
            Try Step:  z[1]=1 dz=-5.59e-4 |f(z)|=2.5e-7
                  : No improvement. Discard the last try step
Stop Criteria satisfied after 2 Iterations
Final Laguerre  z[1]=1 dz=-5.59e-4 |f(z)|=0
Alteration=0% Stage 1=100% Stage 2=0%
            Deflate the real root z=0.9999999999981045
Start Laguerre Iteration for Polynomial=+1x^3-8.000000000001895x^2+19.00000000001327x-
12.000000000022744
            Stage 1=>Stop Condition. |f(z)|<7.99e-15
            Start    : z[1]=0.3 dz=3.16e-1 |f(z)|=6.8e+0
Iteration: 1
            Laguerre Step:  z[1]=1 dz=-6.83e-1 |f(z)|=6.3e-3
            Function value decrease=>try multiple steps in that direction
            Try Step:  z[1]=1 dz=-6.83e-1 |f(z)|=1.2e+0
                  : No improvement. Discard the last try step
Iteration: 2
            Laguerre Step:  z[1]=1 dz=-1.05e-3 |f(z)|=4.8e-11
            In Stage 2. New Stop Condition: |f(z)|<6.66e-15
Iteration: 3
            Laguerre Step:  z[1]=1 dz=-7.96e-12 |f(z)|=8.9e-16
            In Stage 2. New Stop Condition: |f(z)|<6.66e-15
Stop Criteria satisfied after 3 Iterations
Final Laguerre  z[1]=1 dz=-7.96e-12 |f(z)|=8.9e-16
Alteration=0% Stage 1=33% Stage 2=67%
            Deflate the real root z=1.0000000000018952
Solve Polynomial=+1x^2-7x+12.000000000000004 directly
Using the Laguerre Method, the Solutions are:
X1=0.9999999999981045
X2=1.0000000000018952
X3=3.9999999999999964
X4=3.0000000000000036
```

Time used: 0 msec. Solvable level: Easy

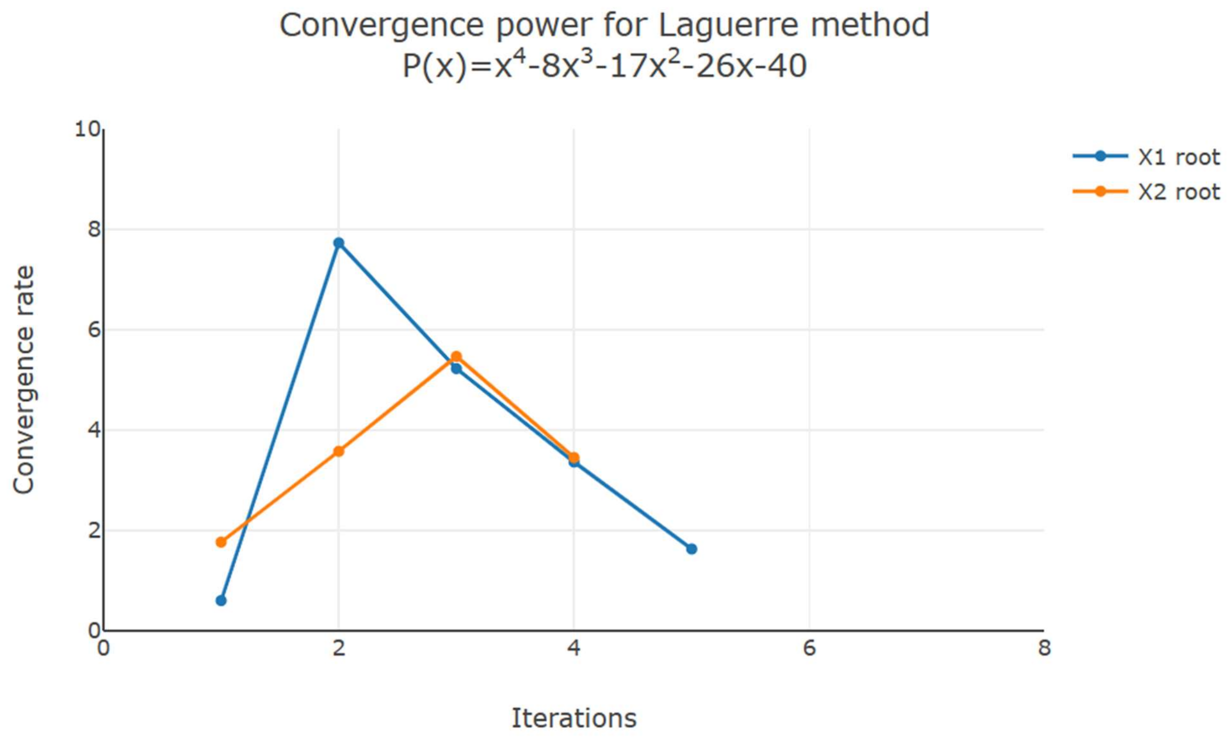


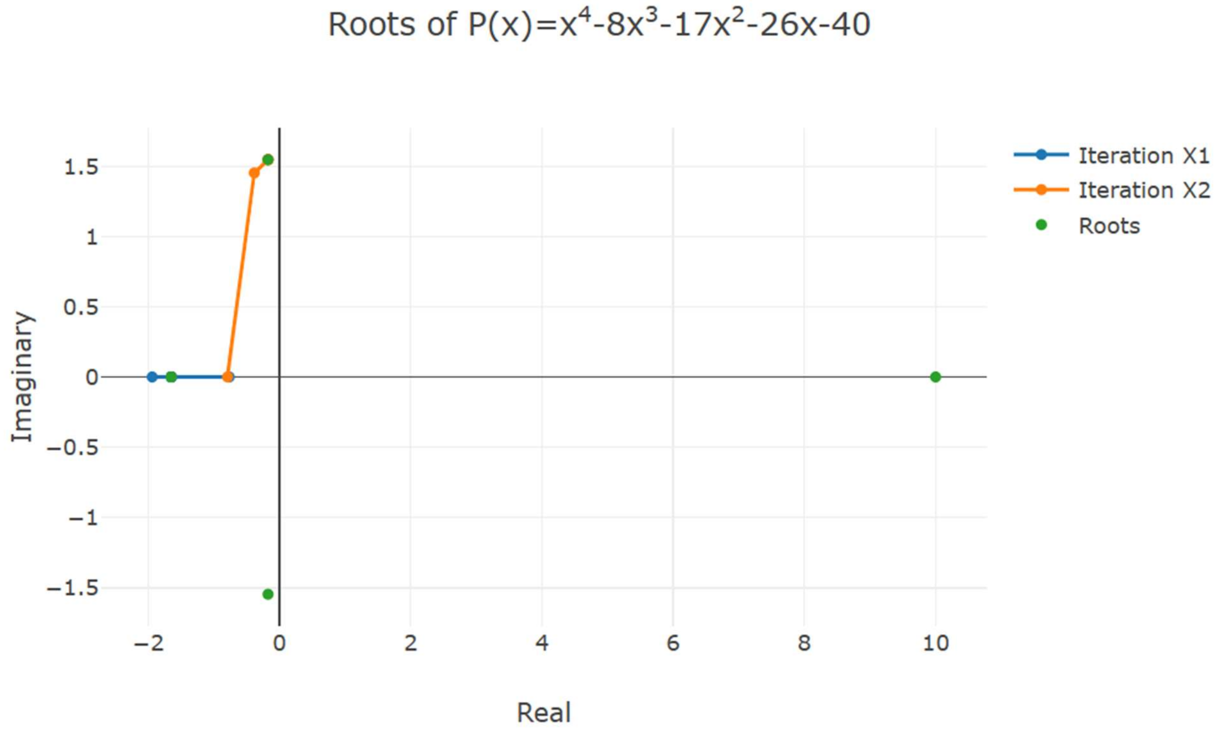
Example 3.

A test polynomial with both real and complex conjugated roots.

```
For the real Polynomial:
+1x^4-8x^3-17x^2-26x-40
Start Laguerre Iteration for Polynomial=+1x^4-8x^3-17x^2-26x-40
            Stage 1=>Stop Condition. |f(z)|<3.55e-14
            Start    : z[1]=-0.8 dz=-7.67e-1 |f(z)|=2.6e+1
Iteration: 1
            Laguerre Step:  z[1]=-2 dz=1.17e+0 |f(z)|=1.9e+1
            Function value decrease=>try multiple steps in that direction
            Try Step:  z[1]=-3 dz=1.17e+0 |f(z)|=9.3e+1
                  : No improvement. Discard the last try step
Iteration: 2
            Laguerre Step:  z[1]=-2 dz=-2.91e-1 |f(z)|=8.4e-2
            In Stage 2. New Stop Condition: |f(z)|<1.91e-14
Iteration: 3
            Laguerre Step:  z[1]=-2 dz=1.58e-3 |f(z)|=2.0e-8
            In Stage 2. New Stop Condition: |f(z)|<1.91e-14
Iteration: 4
            Laguerre Step:  z[1]=-2 dz=-3.81e-10 |f(z)|=2.8e-14
            In Stage 2. New Stop Condition: |f(z)|<1.91e-14
Iteration: 5
            Laguerre Step:  z[1]=-2 dz=-5.34e-16 |f(z)|=0
            In Stage 2. New Stop Condition: |f(z)|<1.91e-14
Stop Criteria satisfied after 5 Iterations
Final Laguerre  z[1]=-2 dz=-5.34e-16 |f(z)|=0
```

```
Alteration=0% Stage 1=20% Stage 2=80%
        Deflate the real root z=-1.650629191439388
Start Laguerre Iteration for Polynomial=+1x^3-9.650629191439387x^2-1.0703897408530487x-
24.233183447530717
        Stage 1=>Stop Condition. |f(z)|<1.61e-14
        Start    : z[1]=-0.8 dz=-7.92e-1 |f(z)|=3.0e+1
Iteration: 1
        Laguerre Step:  z[1]=(-0.4+i1) dz=(-4.08e-1-i1.45e+0) |f(z)|=7.2e+0
        Function value decrease=>try multiple steps in that direction
        Try Step:  z[1]=(0.5+i2) dz=(-4.08e-1-i1.45e+0) |f(z)|=4.3e+1
            : No improvement. Discard the last try step
Iteration: 2
        Laguerre Step:  z[2]=(-0.17+i1.5) dz=(-2.09e-1-i9.45e-2) |f(z)|=1.0e-2
        In Stage 2. New Stop Condition: |f(z)|<1.36e-14
Iteration: 3
        Laguerre Step:  z[5]=(-0.17469+i1.5469) dz=(-5.79e-5+i3.16e-4) |f(z)|=2.7e-11
        In Stage 2. New Stop Condition: |f(z)|<1.36e-14
Iteration: 4
        Laguerre Step:  z[13]=(-0.1746854042803+i1.546868887231) dz=(-7.87e-13+i3.54e-13) |f(z)|=3.6e-15
        In Stage 2. New Stop Condition: |f(z)|<1.36e-14
Stop Criteria satisfied after 4 Iterations
Final Laguerre  z[13]=(-0.1746854042803+i1.546868887231) dz=(-7.87e-13+i3.54e-13) |f(z)|=3.6e-15
Alteration=0% Stage 1=25% Stage 2=75%
        Deflate the complex conjugated root z=(-0.17468540428030604+i1.5468688872313963)
Solve Polynomial=+1x-10 directly
Using the Laguerre Method, the Solutions are:
X1=-1.650629191439388
X2=(-0.17468540428030604+i1.5468688872313963)
X3=(-0.17468540428030604-i1.5468688872313963)
X4=10
Time used: 1 msec. Solvable level: Easy
```





The iterations trail towards the first two roots.

Recommendation

Since the efficiency index is comparable to Laguerre's methods there are no advantages or disadvantages to using Laguerre's over the Halley or Newton. I recommend sticking with the Newton method presented in Parts One and Two for simplicity. However, if you choose the Laguerre method you will not be disappointed.

Conclusion

We have presented a refined Laguerre's method, with a convergence rate of 3 (comparable to the Halley method) building upon the framework established in parts one, two, and three to efficiently and stably find roots of polynomials with real coefficients. I think it is a matter of taste and preference whether or not to use Laguerre's method over e.g., Newton's or Halley's method. 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. In Part Five we will examine one of the so-called simultaneous methods (Durand-Kerner. Aka. Weierstrass method)

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. Wikipedia Lageurre's method, https://en.wikipedia.org/wiki/Laguerre%27s_method