Fast Polynomial Root Finder, Part One

Fast Polynomial Root Finder, Part One.

By Henrik Vestermark (hve@hvks.com)

Abstract:

In general Newton's method for finding roots of polynomials is an effective and easy algorithm to both implement and use. However certain weakness is exposed when trying to find roots in a polynomial with multiple roots. This paper highlights the weakness and devises a modification to the general Newton algorithm that can effectively cope with the multiple roots issue and deal with the usual pitfalls in using the Newton method to find polynomial roots. This paper is part of a multi-series of papers on how to use the same framework to implement different root finder methods.

Introduction:

Newton's method for finding the roots of polynomials is one of the most popular and simple methods. Newton's methods use the following algorithm to progressively find values closer and closer to the root.

$$x_{n+1} = x_n - \frac{P(x_n)}{P\prime(x_n)}$$

By finding one root at a time. There exist other methods that progressively iterate towards all roots simultaneously. These are methods like Aberth-Ehrlich or Durand-Kerner. However, they have other issues that make them less desirable to implement. Of course, there are many other methods to consider. Among them are Halley, Householder 3rd order, Ostrowski, Laguerre, Graeffe, Jenkins-Traub(most likely the most famous), Eigenvalue method, and many others. All of these methods are available in a fast and stable version. Readers can look at [1] that go through 20+ different method and their implementation. For now, I will just go over the practical implementation for a robust and stable root finder using Newton's method. We will furthermore require that the Polynomial have complex coefficients. The algorithm is the same regardless of whether the Polynomial has real (part two) or complex coefficients (this paper).

Fast Polynomial Root Finder, Part One

Contents

The task at hand

Finding the Polynomial roots using Newton's method is usually straightforward to implement:

$$x_{n+1} = x_n - \frac{P(x_n)}{P\prime(x_n)}$$

Typically, you go through these steps.

1) Eliminate Simple roots where the roots are zero.
2) Setup the Newton iteration

   a. Find a suitable start guess
   b. Evaluate both $P(x_n)$ and $P'(x_n)$ using Horner method
   c. Find the step size $P(x_n)/P'(x_n)$
   d. Compute the next x_{n+1}
   e. Repeat b-d until the stopping criterion is met
   f. Divide the newly founded root up in P(x) to compute the new reduced $P_{new}(x)$
   g. Repeat a-f until we are left with a first or second-degree Polynomial
3) Solve the first or second-degree polynomial directly

You will go more or less through the same steps for many of the other root-finding methods.

The issue with Newton's method:

In itself, the Newton method is not necessarily stable but requires extra code to handle the classical pitfalls when implemented as a robust, fast, and stable solution. By just looking at the above formula (2) it is clear we will have an issue when P'(x) is zero or close to zero. But that is not the only issue you will encounter. Sometimes if we hit a local minimum, you can get step length P(x)/P'(x) that throws the search far away from any roots. It also matters where to start the search for the root since the above algorithm will only converge to a root when you are somehow close to the roots. The Newton method has a convergence order of two meaning that the number of correct direct doubles for each iteration. But when trying to find a solution that contains multiple roots e.g. $(x-2)^2(x-3)=0$ the convergence rate drops to a linear rate requiring many more iterations to find the root.
We would need to address the multiple root issues and ensure we maintain the convergence order of 2 in these situations as well. At some point, we need to figure out when to stop a search and be happy with the accuracy at the same time. Our goal is to take it to the limit of what the IEEE754 floating point standard can handle as implemented in the C++ *double* type. If we somehow relax our stopping criterion the inaccuracies will propagate to the other roots that will drift further and further away from the real roots. Lastly, when a root is found we need to divide the root up in the Polynomial and repeat the search for the new reduced polynomial. When making a synthetic division you have a choice between what is known as forward deflation, backward deflation, or composite deflation. The choice can have an impact on the accuracy of the roots.

The multiple roots issue

Consider the polynomial:

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

As you can see below the roots are well separated.

Fast Polynomial Root Finder, Part One

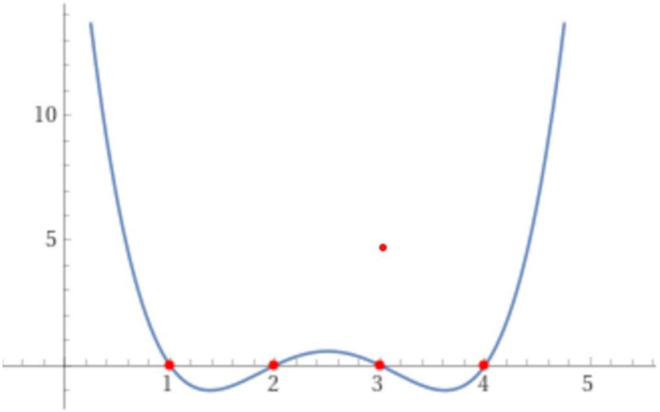

Figure 1. Well separated roots

Using a starting point of 0.5 the Newton iteration progresses as follows toward the first root:

	x	P(x)
Initial guess	0.5	
1	0.798295454545455	6.6E+00
2	0.950817599863883	1.7E+00
3	0.996063283034122	3.2E-01
4	0.999971872651986	2.4E-02
5	0.999999998549667	1.7E-04
6	0.999999999999999	8.7E-09
7	1.000000000000000	7.1E-15

As we can see we get the first root x=1 after only 7 iterations. We also notice that after the second iteration x_2=0.95, we roughly double the number of correct digits towards the first root for each iteration. An iteration method that doubles the number of correct digits for each iteration is said to have a convergence order of 2.

Now let's change the polynomial and introduce a double root at x=1:

$$P(x)=(x-1)^2\,(x-3)(x-4)=x^4-9x^3+27x^2-31x+12$$

With the same starting point x=0.5, we get a much slower convergence and after 27 iterations we get no more improvement towards the first root of x=1 and the results are only accurate to approximately the first 8 digits.

	x	P(x)
Initial guess	0.5	
1	0.713414634146341	2.2E+00
2	0.842942878437970	6.2E-01
3	0.916937117337937	1.7E-01

…		
10	0.999306565270595	1.2E-05
…		
20	0.999999322514237	1.1E-11
…	0.999999661405383	2.8E-12
27	0.999999996306426	0.0E+00

What exactly happens here?

if $P(x)=(x-1)^2\,(x-3)(x-4)$, then $P'(x)=(x-1)(4x^2-23x+31)$

The root x=1 is both a root for the original Polynomial P(x) but also of P'(x). see the image below.

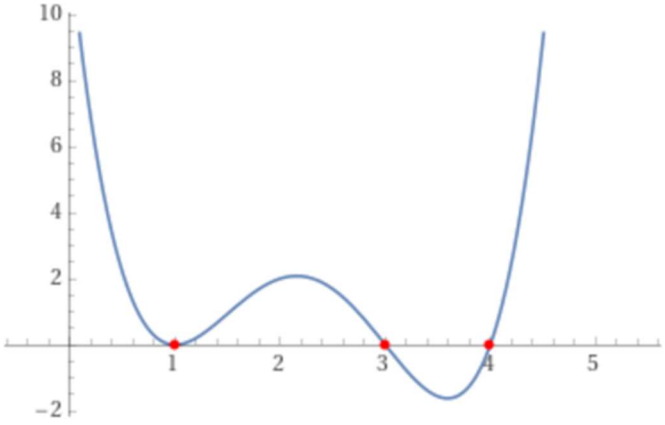

Figure 2. A double root at x=1

In a Newton iteration, when both P(x) and P'(x) go towards 0 introducing round-off errors in the accuracy of calculating the next x_{n+1} in a Newton iteration. For illustration, we repeat the iteration step but include the P'(x). Furthermore, we introduce the convergence rate q as well.

	x	P(x)	P'(x)	q
Initial guess	0.5			
1	0.713414634146341	2.2E+00	-1.0E+01	
2	0.842942878437970	6.2E-01	-4.8E+00	1.3
3	0.916937117337937	1.7E-01	-2.3E+00	1.2
…				
10	0.999306565270595	1.2E-05	-1.7E-02	1.1
…				
20	0.999999322514237	1.1E-11	-1.6E-05	1.0
…				
25	0.999999976999021	1.1E-14	-5.2E-07	1.0
26	0.999999996306426	5.3E-15	-2.8E-07	-
27	0.999999996306426	0.0E+00	-4.4E-08	-

Fast Polynomial Root Finder, Part One

We notice a couple of things; the convergence rate q is much slower than in our first example; ~2 versus ~1.1. Furthermore, we can see for each iteration that the root convergence with a linear factor of 2 instead of what we should expect from the quadratic factor 2 from our first example. For higher orders multiplicity of roots, gets even worse. E.g.

if $P(x)=(x-1)^3\,(x-4)$, then $P'(x)=(x-1)^2\,(4x-13)$

After 31 iterations we get x=0.999998662746209 which is only accurate to approximately the first 5 digits.

	x	P(x)	P'(x)	q
Initial guess	0.5			
1	0.659090909090909	4.4E-01	-2.8E+00	
2	0.768989234449761	1.3E-01	-1.2E+00	1.2
…				
29	0.999995827925540	4.4E-16	-2.9E-10	1.0
30	0.999998662746209	4.4E-16	-1.6E-10	-
31	0.999998662746209	0.0E+00	-1.6E-11	-

Around the tripe multiple roots at x=1 then P'(x) is very flat ~ 0 in a fairly wide radius around the multiple roots.

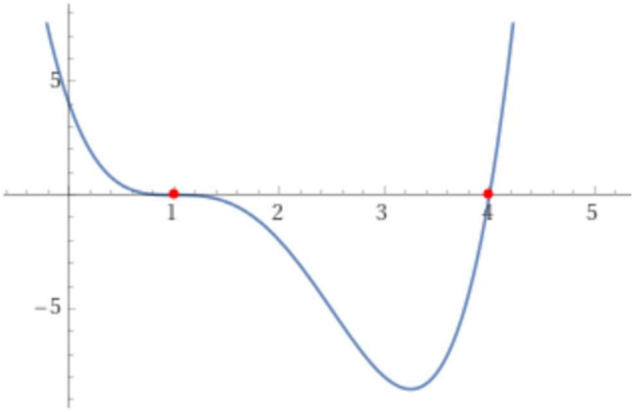

Figure 3. A triple root at x=1

What to do about multiple roots with the Newton iteration?

To overcome this reduction of the Newton step size we need to multiply it with a factor m so we instead used the modified Newton iteration.

$$x_{n+1} = x_n - m\frac{P(x_n)}{P\prime(x_n)}$$

Where *m* is the multiplicity of the roots. This is all well-known stuff. The challenge is how to find *m* in real life.

A suitable starting point for root finders

To make the iterative methods faster to converge to Polynomial roots it is important that we somehow start at a suitable point that is in the neighborhood of a root. Luckily, many people have studied this field and there are an impressive 45+ methods for creating a priori bound of the roots as outlined by J.McNamee, Numerical Methods for Roots of Polynomials [8]. Most a priori bounds are for finding the radius of a circle where all the roots are located. A few also deal with the radius of the circle where the root with the smallest magnitude is located. This is very useful for methods that find one root at a time and where the strategy is to find the roots with increasing order of magnitude.

Priori for the root with the smallest magnitude.

Most root-finding implementations that I have seen do not pay too much attention to the starting point. E.g. [6] Grant-Hitchins uses a fixed starting point of (0.001+i0.1). Instead of a fixed starting point, I would advocate for the starting point as implemented by Madsen [2]. Were we find the starting point z_0 where the root with the smallest magnitude lies outside this circle:

$$z_0 = \frac{1}{2} min_{0<k} \sqrt[k]{\left|\frac{a_0}{a_k}\right| e^{i\theta}},\ \theta = \arg\left(-\frac{P(0)}{P\prime(0)}\right)$$

The smallest root is located outside the circle with a radius $|z_0|$ in the complex plane.

Consider the Polynomial:

$$P(x)=(x-1)(x+2)(x-3)(x-4)= x^4+2x^3-13x^2-14x+24$$

The above formula yields a starting point z_0=0.68 which is close to the nearest root of x=1.

Now consider the Polynomial:

$$P(x)= (x-1)(x+1000)(x-2000)(x+3000)= x^4+1999x^3-5002E3x^2-5995E6x+6E9$$

The above formula yields a z_0= 0.5 (nearest root x=1)

After the first root x=1 is found the deflated polynomial is then P(x)= (x+1000)(x-2000)(x+3000)=x^3+2E3x^2-5E6x-6E9 and the above formula yield a new Starting point for a new search for the deflated Polynomial is z_0=600 (nearest root x=1,000)

This algorithm computes a reasonable and suitable starting point for our root search.

```
    // 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<complex<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] != complexzero)
        {
            double tmp = exp((a0 - log(abs(a[i]))) / static_cast<double>(n - i));
            if (tmp < min)
                min = tmp;
        }

    return min*0.5;
    };
```

Evaluation of the Polynomial at a complex point.

Most of the root-finding methods require us to evaluate a Polynomial at some point.

To evaluate a polynomial P(z) were:

$$P(z)=a_n z^n+a_{n-1}z^{n-1},\ldots,a_1 z+a_0$$

We use the Horner [4] method given by the recurrence:

$$b_n=a_n$$
$$b_k=b_{k-1}z+a_k, \quad k=n-1,\ldots,0$$
$$P(z)=b_0$$

The last term of this recurrence b_0 is then the value of $P(z)$. Horner method has long been recognized as the most efficient way to evaluate a Polynomial at a given point. The algorithm works for coefficients to be either real or complex numbers.

```
    // Evaluate a polynomial with complex coefficients a[] at a complex point z and
    // return the result
    // This is the Horner's methods
    auto horner = [](const vector<complex<double>>& a, const complex<double> z)
    {
        const size_t n = a.size() - 1;
        complex<double> fval=a.front();
        eval e;

        for (size_t i = 1; i <= n; i++)
            fval = fval * z + a[i];

        e = { z, fval,abs(fval) };
        return e;
    };
```

A suitable stopping criterion for a root.

In [8] they go over many different techniques to compute a suitable stopping criterion. See also [1]. Many roots finders can use the method used by Adams [5] or Hitching [6] to find a suitable stopping criterion for polynomials with either real or complex coefficients.

Fast Polynomial Root Finder, Part One

When doing the iterative method, you will at some point need to consider what stopping criteria you want to apply for your root finders. Since most iterative root finders use the evaluation of the polynomial to progress it is only natural to continue our search until the evaluation of P(z) is close enough to 0 to accept the root at that point.

Error in arithmetic operations

J.H.Wilkinson in [7] has shown that the errors in performing algebraic operations are bound by:

$\varepsilon < \frac{1}{2}\beta^{1-t}$, β is the base, and t is the precision (assuming round to nearest)

Notice $\frac{1}{2}\beta^{1-t} = \beta^{-t}$

For the Intel microprocessor series and the IEE754 standard for floating point operations $\beta = 2$ and $t=53$ for 64bit floating point arithmetic or 2^{-53}

A simple upper bound

A simple upper bound can then be found using the above information for a polynomial with degree n.

	Polynomials	
Number of operations:	**Real coefficient**	**Complex coefficients**
Real point	$\lvert a_o \rvert \cdot 2n \cdot 2^{-53}$	$\lvert a_o \rvert \cdot 4n \cdot 2^{-53}$
Complex point	$\lvert a_o \rvert \cdot 4n \cdot 2^{-53}$	$\lvert a_o \rvert \cdot 6n \cdot 2^{-53}$

A better upper bound

In this category, we have among others Adams [5] and Grant & Hitchins [6] stopping criteria for polynomials.
Polynomial root finders usually can handle polynomials with both real and complex coefficients evaluated at a real or complex number. Since Adams' stopping criterion is for Polynomials with real coefficients, we will use Grant & Hitchins bound which is similar but for Polynomials with complex coefficients.

Grant & Hitchins stopping criteria for polynomials with complex coefficients

Polynomial with complex coefficients z_n evaluated at a complex point z, using Horner's method. Grant and Hitchins [6] derive an upper error bound for the errors in evaluating the polynomial as follows using the recurrence where $z=x+iy$ and the complex coefficients are represented as a_n+ib_n:

$c_n=a_n,\ d_n=b_n,\ g_n=1,\ h_n=1;$
$c_k=xc_{k+1}-yd_{k+1}+a_k, \quad k=n-1,\ldots,0$
$d_k=yc_{k+1}+xd_{k+1}+b_k$
$g_k=|x|(g_{k+1}+|c_{k+1}|)+|y|(h_{k+1}+|d_{k+1}|)+|a_k|+2|c_k|$
$h_k=|y|(g_{k+1}+|c_{k+1}|)+|x|(h_{k+1}+|d_{k+1}|)+|b_k|+2|d_k|$

Now the error is $(g_0+ih_0)\varepsilon$, where $\varepsilon = \frac{1}{2}\beta^{1-t}$. Now since the recurrence in itself introduces error [6] safeguard the calculation by adding the upper bound for the rounding errors in the recurrence, so we have the bound for evaluating a complex polynomial at a complex point:

$$e=(g_0+ih_0)\varepsilon(1+\varepsilon)^{5n}, \quad \text{where } \varepsilon=\tfrac{1}{2}\beta^{1-t}$$

There exist other methods that are also useful to consider, see [1]

```
// Calculate an upper bound for the rounding errors performed in a
// polynomial with complex coefficient a[] at a complex point z.
// (Grant & Hitchins test)
auto upperbound = [](const vector<complex<double>>& a, complex<double> z)
{
    const size_t n = a.size() - 1;
    double nc, oc, nd, od, ng, og, nh, oh, t, u, v, w, e;
    double tol = 0.5 * pow((double)_DBL_RADIX, -DBL_MANT_DIG + 1);

    oc = a[0].real();
    od = a[0].imag();
     og = oh = 1.0;
    t = fabs(z.real());
    u = fabs(z.imag());
    for (size_t i = 1; i <= n; i++)
    {
        nc = z.real() * oc - z.imag() * od + a[i].real();
        nd = z.imag() * oc + z.real() * od + a[i].imag();
        v = og + fabs(oc);
        w = oh + fabs(od);
        ng = t * v + u * w + fabs(a[i].real()) + 2.0 * fabs(nc);
        nh = u * v + t * w + fabs(a[i].imag()) + 2.0 * fabs(nd);
        og = ng;
        oh = nh;
        oc = nc;
        od = nd;
    }
    e = abs(complex<double>(ng, nh)) * pow(1 + tol, 5 * n) * tol;
    return e;
};
```

Polynomial Deflation strategy

After we have found a root, we need to make a synthetic division of that root up in the current Polynomial to reduce the polynomial degree and prepare to find the next root. The question then arises do you use Forward or Backward Deflation?
Wilkinson [7] has shown that to have a stable deflation process you should choose *forward* deflation if you find the roots of the polynomial in increasing magnitude and always deflate the polynomial with the lowest magnitude root first and of course, the opposite *backward* deflation when finding the roots with decreasing magnitude.
To do forward deflation we try to solve the equations starting with the highest coefficients a_n:

$$a_nz^n+a_{n-1}z^{n-1}+\cdots+a_1z+a_0=(b_{n-1}z^{n-1}+b_{n-2}z^{n-2}+\cdots+b_1z+b_0)(z-R)$$

And R is the root.
Now solve it for b's you get the recurrence:

$b_{n-1}=a_n$
$b_k=a_{k+1}+R\cdot b_{k+1}, \qquad k=n-2,\ldots,0$

This simple algorithm works well for polynomials with real coefficients and real roots or complex coefficients with complex roots using the same recurrence just using complex arithmetic instead.

```
// Deflate polynomial and compute new coefficients in coeff
 z = 0;
 for (int j = 0; j < n; j++)
     z = coeff[j] = z * pz.z + coeff[j];
 coeff.resize(n);
```

The Implementation of K. Madsen Newton Algorithm

The implementation of this root finder follows the method as first described by K. Madsen in [2]. Which was an AlgolW implementation. This implementation below is a modified version translated into C++ and uses a more modern structure including the C++ STL library. The first step is to lay out the process.
Of course, the most interesting part is the section "Start the Newton iteration" Madsen [2] provides a very fast and efficient implementation that not only finds the roots in surprisingly few iterations but also handles the usual issues with the Newton method. I do not plan to repeat what is so excellent as described in [2] but just highlight some interesting areas of his Newton implementation.

1) First, we eliminate simple roots (roots equal to zero)
2) Then we find a suitable starting point to start our Newton Iteration, this also includes termination criteria based on an acceptable value of P(x) where we will stop the current iteration.
3) Start the Newton iteration
   a. The first step is to find the $dz_n=P(z_n)/P'(z_n)$ and of course, decide what should happen if $P'(z_n)$ is zero. 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 $dz_n=dz_n(0.6+i0.8)m$. This is equivalent to rotating the direction with an odd degree of 53 degrees and multiplying the direction with the factor m. A suitable value for m =5 is reasonable when this happens.
   b. Furthermore, it is easy to realize that if $P'(z_n)\sim 0$. You could get some unreasonable step size of dz_n and therefore introduced a limiting factor that reduced the current step size if $abs(dz_n)>5\cdot abs(dz_{n-1})$ than the previous iteration's step size. Again, you alter the direction with $dz_n=dz_n(0.6+i0.8)5(abs(dz_{n-1})/abs(dz_n))$.
   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 2nd order convergence rate down to a linear convergence rate. After a suitable dz_n is found and a new $z_{n+1}=z_n-\frac{P(z_n)}{P'(z_n)}$ we then look to see if $P(z_{n+1})>P(z_n)$: If so we look at a revised $z_{n+1}=z_n-0.5dz_n$ and if $P(z_{n+1})\geq P(z_n)$ then he used the original z_{n+1} as the new starting point for the next iteration. If not then we accept z_{n+1} as a better

choice and continue looking at a newly revised $z_{n+1}=z_n-0.25dz_n$. If on the other hand the new $P(z_{n+1})\geq P(z_n)$ we used the previous z_{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 $dz_n=dz_n(0.6+i0.8)$ and we use $z_{n+1}=z_n-dz_n$ as the new starting point for the next iteration.
if on the other hand $P(z_{n+1})\leq P(z_n)$: Then we are looking in the right direction and we then continue stepping in that direction using $zn+1=z_n-mdz_n$, m=2,..,n as long as $P(z_{n+1})\leq P(z_n)$ 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 2nd-order convergence rate even for multiple roots.

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

In [2] they 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 the Newton method will converge towards a root. When we get into that circle, we automatically switch to stage 2. In stage 2 we skip step d) and just use an unmodified Newton step $z_{n+1}=z_n-\frac{P(z_n)}{P'(z_n)}$ 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.
In [2] they use the following criteria to switch to stage 2 based on the theorem 7.1 from Ostrowski [3] that states if K is a circle with center $w-\frac{P(w)}{P\prime(w)}$ And radius $|\frac{P(w)}{P\prime(w)}|$
Then we have guarantee convergence if the following two conditions are satisfied:

$$p(w)p'(w)\neq 0 \quad and$$
$$2|\frac{p(w)}{p'(w)}|\cdot\max_{z\varepsilon K}|p''(z)|\leq|p'(w)|$$

The Newton iterations with initial value w will lead to a convergence of zero within the circle K. To simplify the calculation we make 2 substitutes, since $\max_{z\varepsilon K}|p''(z)|\approx|p''(w)|$ and instead of p"(w) we replace it with a difference approximation $p''(w)\approx\frac{p'(z_{k-1})-p\prime(w)}{z_{k-1}-w}$

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 with complex coefficients. The same algorithm can be used if the Polynomial coefficients are real. See [1] for details.

```
/*
 ***************************************************************************
 *
 *                       Copyright (c) 2023
 *                       Henrik Vestermark
```

Fast Polynomial Root Finder, Part One

```
 *                      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     :   Newton.cpp
 * Module ID Nbr   :
 * Description     :   Solve n degree polynomial using Newton's (Madsen) method
 * -------------------------------------------------------------------------
 * Change Record   :
 *
 * Version    Author/Date          Description of changes
 * -------  -------------  ----------------------
 * 01.01      HVE/24Sep2023 Initial release
 *
 * End of Change Record
 * -------------------------------------------------------------------------
*/

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

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

using namespace std;
constexpr int       MAX_ITER = 50;

// Find all polynomial zeros using a modified Newton method
// 1) Eliminate all simple roots (roots equal to zero)
// 2) Find a suitable starting point
// 3) Find a root using Newton
// 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>> PolynomialRoots(const vector<complex<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)
```

Fast Polynomial Root Finder, Part One

```
    eval p1zprev;   // P'(zprev)
    complex<double> z;      // Use as temporary variable
    complex<double> dz;     // The current stepsize dz
    int itercnt;    // Hold the number of iterations per root
    vector<complex<double>> roots;  // Holds the roots of the Polynomial
    vector<complex<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() == complexzero; --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<complex<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] != complexzero)
        {
            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 complex coefficients a[] at a complex point z and
    // return the result
    // This is the Horner's methods
    auto horner = [](const vector<complex<double>>& a, const complex<double> z)
    {
        const size_t n = a.size() - 1;
        complex<double> fval=a.front();
        eval e;

        for (size_t i = 1; i <= n; i++)
            fval = fval * z + a[i];

        e = { z, fval,abs(fval) };
        return e;
    };

    // Calculate an upper bound for the rounding errors performed in a
    // polynomial with complex coefficient a[] at a complex point z.
    // (Grant & Hitchins test)
    auto upperbound = [](const vector<complex<double>>& a, complex<double> z)
    {
        const size_t n = a.size() - 1;
        double nc, oc, nd, od, ng, og, nh, oh, t, u, v, w, e;
        double tol = 0.5 * pow((double)_DBL_RADIX, -DBL_MANT_DIG + 1);

        oc = a[0].real();
        od = a[0].imag();
        og = oh = 1.0;
        t = fabs(z.real());
```

Fast Polynomial Root Finder, Part One

```
        u = fabs(z.imag());
        for (size_t i = 1; i <= n; i++)
        {
            nc = z.real() * oc - z.imag() * od + a[i].real();
            nd = z.imag() * oc + z.real() * od + a[i].imag();
            v = og + fabs(oc);
            w = oh + fabs(od);
            ng = t * v + u * w + fabs(a[i].real()) + 2.0 * fabs(nc);
            nh = u * v + t * w + fabs(a[i].imag()) + 2.0 * fabs(nd);
            og = ng;
            oh = nh;
            oc = nc;
            od = nd;
        }
        e = abs(complex<double>(ng, nh)) * pow(1 + tol, 5 * n) * tol;
        return e;
    };

    // Do Newton 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
        vector<complex<double>> coeffprime;

        // Calculate coefficients of p'(x)
        for (int i = 0; i < n; i++)
            coeffprime.push_back(coeff[i] * complex<double>(double(n - i), 0.0));

        // Step 2 find a suitable starting point z
        rprev = startpoint(coeff);      // Computed startpoint
        z = coeff[n - 1] == complexzero ? complex<double>(1.0) : -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 = 6 * n * pzprev.apz * pow((double)_DBL_RADIX, -DBL_MANT_DIG);
```

Fast Polynomial Root Finder, Part One

```
        // 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++)
        {
            // Calculate current P'(z)
            p1z = horner(coeffprime, pz.z);
            if (p1z.apz == 0.0)                 // P'(z)==0 then rotate and try again
                dz *= rotation * complex<double>(Max_stepsize);  // Multiply with 5 to
get away from saddlepoint
            else
            {
                dz = pz.pz / p1z.pz;  // next dz
                // Check the Magnitude of Newton'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
converge otherwise true
                z = (p1zprev.pz - p1z.pz) / (pzprev.z - pz.z);
                stage1 = (abs(z) / p1z.apz > p1z.apz / pz.apz / 4.0) || (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;
                eval npz;

                zn = pz.z;
                for (div2 = pz.apz > pzprev.apz; steps <= n; ++steps)
                {
                    if (div2 == true)
                    {  // Shorten steps
                        dz *= complex<double>(0.5);
                        zn = pzprev.z - dz;
                    }
                    else
                        zn -= dz;  // try another step in the same direction

                    // Evaluate new try step
                    npz = horner(coeff, zn);
                    if (npz.apz >= pz.apz)
                        break; // Break if no improvement

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

Fast Polynomial Root Finder, Part One

```
                    if (div2 == true && steps == 2)
                    {   // 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);
            }
        }

        // 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());
        pzprev = horner(coeff, z);
        if (pzprev.apz <= pz.apz)
            pz = pzprev;

        // Save the root
        roots.push_back(pz.z);

       // Deflate polynomial and compute new coefficients in coeff
       z = complex<double>(0);
       for (int j = 0; j < n; j++)
           z = coeff[j] = z * pz.z + coeff[j];
       coeff.resize(n);
      /*
        std::transform(coeff.begin(), coeff.end() - 1, coeff.begin() + 1, coeff.begin(),
            [pz](const complex<double>& coeff, const complex<double>& next_coeff) {
                return coeff * pz.z + next_coeff;
        });
        coeff.resize(n);
        */
    }   // End Iteration

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

        // Notice a[0] is !=0 since all roots=zero has been captured previously
        if (n == 1)
            roots.push_back(-a[1]/a[0]);
        else
        {
            if (a[1] == complexzero)
            {
                v = sqrt(-a[2] / a[0]);
                roots.push_back(v);
                roots.push_back(-v);
```

```
            }
            else
            {
                v = sqrt(complex<double>(1.0)-
complex<double>(4.0)*a[0]*a[2]/(a[1]*a[1]));
                if (v.real() < 0)
                    v = (complex<double>(-1.0) - v) * a[1] / (complex<double>(2.0) *
a[0]);
                else
                    v = (complex<double>(-1.0) + v) * a[1] / (complex<double>(2.0) *
a[0]);
                roots.push_back(v);
                roots.push_back(a[2] / (a[0] * 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 complex Polynomial:
+1x^3+(-13-i1)x^2+(44+i12)x+(-32-i32)
Start Newton Iteration for Polynomial=+1x^3+(-13-i1)x^2+(44+i12)x+(-32-i32)
Stage 1=>Stop Condition. |f(z)|<3.01e-14
Start : z[1]=(0.4+i0.2) dz=(4.31e-1+i2.46e-1) |f(z)|=2.6e+1
Iteration: 1
Newton Step: z[1]=(1+i0.7) dz=(-5.91e-1-i4.63e-1) |f(z)|=6.3e+0
Function value decrease=>try multiple steps in that direction
Try Step: z[1]=(2+i1) dz=(-5.91e-1-i4.63e-1) |f(z)|=1.1e+1
: No improvement=>Discard last try step
Iteration: 2
Newton Step: z[2]=(1.0+i1.0) dz=(-1.83e-2-i2.99e-1) |f(z)|=9.0e-1
In Stage 2=>New Stop Condition: |f(z)|<4.79e-14
Iteration: 3
Newton Step: z[2]=(1.0+i1.0) dz=(4.07e-2+i8.94e-3) |f(z)|=1.8e-2
In Stage 2=>New Stop Condition: |f(z)|<4.65e-14
Iteration: 4
Newton Step: z[4]=(1.000+i1.000) dz=(-6.04e-4-i5.04e-4) |f(z)|=6.3e-6
In Stage 2=>New Stop Condition: |f(z)|<4.65e-14
Iteration: 5
Newton Step: z[8]=(1.0000000+i1.0000000) dz=(2.39e-8-i2.81e-7) |f(z)|=8.2e-13
In Stage 2=>New Stop Condition: |f(z)|<4.65e-14
Iteration: 6
Newton Step: z[14]=(1.0000000000000+i1.0000000000000) dz=(3.32e-14+i1.53e-14)
|f(z)|=7.9e-15

In Stage 2=>New Stop Condition: |f(z)|<4.65e-14 Stop Criteria satisfied after 6 Iterations Final Newton z[14]=(1.0000000000000+i1.0000000000000) dz=(3.32e-14+i1.53e-14) |f(z)|=7.9e-15 Alteration=0% Stage 1=17% Stage 2=83%

Deflate the complex root z=(0.9999999999999999+i0.9999999999999998)
Solve Polynomial=+(1)x^2+(-12-i2.220446049250313e-16)x+(32+i3.552713678800501e-15) directly
Using the Newton Method, the Solutions are:
X1=(0.9999999999999999+i0.9999999999999998)
X2=(8.000000000000002-i4.440892098500625e-16)
X3=(3.999999999999999+i6.661338147750937e-16)

Example 2.

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

For the complex Polynomial:
+1x^3+(-10-i2)x^2+(16+i18)x+(-i16)
Start Newton Iteration for Polynomial=+1x^3+(-10-i2)x^2+(16+i18)x+(-i16)

Stage 1=>Stop Condition. |f(z)|<1.07e-14
Start : z[1]=(0.2+i0.2) dz=(2.48e-1+i2.21e-1) |f(z)|=9.1e+0
Iteration: 1
Newton Step: z[1]=(0.6+i0.6) dz=(-3.76e-1-i3.54e-1) |f(z)|=2.4e+0
Function value decrease=>try multiple steps in that direction
Try Step: z[1]=(1+i0.9) dz=(-3.76e-1-i3.54e-1) |f(z)|=3.7e-2
: Improved=>Continue stepping
Try Step: z[1]=(1+i1) dz=(-3.76e-1-i3.54e-1) |f(z)|=1.5e+0
: No improvement=>Discard last try step
Iteration: 2
Newton Step: z[2]=(1.0+i0.96) dz=(3.68e-4-i3.61e-2) |f(z)|=9.2e-3
Function value decrease=>try multiple steps in that direction
Try Step: z[2]=(1.0+i1.0) dz=(3.68e-4-i3.61e-2) |f(z)|=9.6e-7
: Improved=>Continue stepping
Try Step: z[2]=(1.0+i1.0) dz=(3.68e-4-i3.61e-2) |f(z)|=9.2e-3
: No improvement=>Discard last try step
Iteration: 3
Newton Step: z[5]=(1.0002+i1.0000) dz=(1.82e-4+i2.89e-5) |f(z)|=2.4e-7
Function value decrease=>try multiple steps in that direction
Try Step: z[5]=(1.0000+i1.0000) dz=(1.82e-4+i2.89e-5) |f(z)|=8.9e-16
: Improved=>Continue stepping
Try Step: z[5]=(0.99982+i0.99997) dz=(1.82e-4+i2.89e-5) |f(z)|=2.4e-7
: No improvement=>Discard last try step
Stop Criteria satisfied after 3 Iterations
Final Newton z[5]=(1.0000+i1.0000) dz=(1.82e-4+i2.89e-5) |f(z)|=8.9e-16
Alteration=0% Stage 1=100% Stage 2=0%

Deflate the complex root z=(0.9999999913789768+i0.9999999957681246)
Solve Polynomial=+(1)x^2+(-9.000000008621024-i1.0000000042318753)x+(8.000000068968186+i8.000000033855004) directly
Using the Newton Method, the Solutions are:

X1=(0.9999999913789768+i0.9999999957681246)
X2=(8.000000000000004-i2.6645352478243948e-15)
X3=(1.0000000086210223+i1.0000000042318753)

Conclusion

Presented is a modified Newton method originally based on [2] making the Newton method more efficient and stable for finding polynomial roots with complex coefficients. The same method can easily be applied to Polynomials with real coefficients.
This was part one, part two handled the case where we only found roots in a polynomial with real coefficients. However, the root can still be complex. Part three shows the adjustment needed to implement a higher-order method e.g. Halley. Part 4 how easy it is to fit another method like Laguerre's into the same framework.
A web-based Polynomial solver 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.