Practical Implementation of Polynomial Root Finders
By Henrik Vestermark (hve@hvks.com)

Abstract:
There are many practical issues arising from designing and implementing a polynomial root finder. If you look into this, you will discover that there exist many different methods for how to find the roots of a polynomial. Some are in today’s standard consider to be obsolete other are still hanging around and some are consider the state of the art.
This paper go through practical issues arising for designing and implementing these different methods. The paper also highlight some of the many difference between the methods and there is a discussion how to overcome the typical loss of accuracy when dealing with multiplicity of roots higher than one.

As always, there is plenty of C++ source code available to show how you from a practical point of view are implementing these different methods.

Introduction:
In general, there exist two different types of root finder methods.

- Method that find one or two roots at a time. E.g. Newton, Halley, Jenkins-Traub, Ostrowski etc
- Method that simultaneous find all roots at once. E.g Durand-Kerner or Aberth-Ehrlich method, Rutishauser QD method.

Within these methods, that find one or two roots at a time. We have typical two variation.

Classic Method. E.g. Newton, Halley, Jenkin-Traub, Ostrowski, Householder 3\textsuperscript{rd} etc.
Matrix Method. E.g. Eigenvalues using QR algorithm.

Some methods requires the evaluation of the polynomial at a real or complex point together with the first and second derivative of the polynomial. Others typical matrix methods do not required this. For all the methods that find one or two roots at a time you will need to divide the root up in the original polynomial and the restart the search for the next root until all roots has been found. Therefore, it is also important that we cover how to do that. Now all the methods are iterative methods meaning that you first find a suitable starting point for your roots and then through a number of iterations, you get closer and closer to the roots until some stopping criteria has been satisfied. We will also discuss and show how to ensure you have bound the roots at the highest possible accuracy. Finally, we will go through some of the most well known methods and see how they fare against each other. Normally if the polynomial is well behaved it is relative easier to find all the roots. However, if a polynomial has
multiple roots (multiplicity >1) then most methods slow down and required a much higher number of iterations and the final accuracy of the roots is also reduced. We will also discuss ways to overcome this issue and maintain a convergence rate at the same rate as for well-behaved roots.

Since we provide algorithm in C++ it is understood that default headers is include e.g. when using the complex<> template class in C++ it is understood that the appropriate template header has been included. E.g. #include <complex>

In this paper we will address the following root finding methods:

- Newton
- Halley
- Householders 3rd
- Ostrowski multi-point
- Laguerre
- Eigenvalue
- Durand-Kerner
- Aberth-Ehrlich
- Rutishauser Quotient-Difference
- Tangent-Graeffe
- Jenkins-Traub
- Bairstows
- Ostrowski Square-root

Before we address any roots, finding methods above in details we first need to get the basic done. The first section show how to:

- Evaluate a Polynomial at a given point
- Deflate a polynomial with the root found
- Find a suitable start guess for our root finder
- Use suitable stopping criteria for our iterative process
- Perform Polynomial Taylor shifting if needed.
- Finding simple roots of a Polynomial including linear and quadratic solutions

The next section is all about the various root finding method. There exist many more method than what is presented here, but this will covered the most useful of the methods out there.

The appendix will covered various supporting function or larger root finding methods like the Jenkins-Traub where the c++ source code take up too many pages and I believe it was more appropriated to moved them to the Appendix.
Practical Implementation of Polynomial Root Finders

Contents
Abstract: .................................................................................................................... 1
Introduction: .............................................................................................................. 1
Evaluation of Polynomials: ........................................................................................ 6
   Algorithm Horner with real coefficients at a real point ....................................... 7
   Algorithm Horner with real coefficients at a complex point ............................... 7
   Algorithm Horner with complex coefficients at a complex point ...................... 8
Deflation of a Polynomial: ......................................................................................... 8
   Forward Deflation of Polynomials: ........................................................................ 9
      Algorithm Forward deflation with real coefficients with a real root ............... 10
      Algorithm Forward deflation with real coefficients with a complex root ........ 10
      Algorithm Forward deflation with complex coefficients with a complex root... 10
   Backward Deflation of Polynomials: ................................................................... 11
      Algorithm Backward deflation with real coefficients with a real root ............ 12
      Algorithm Backward deflation with real coefficients with a complex root ...... 12
      Algorithm Backward deflation with complex coefficients with a complex root 13
   Forward or Backward Deflation? ........................................................................ 13
   Composite Deflation of Polynomials: ................................................................. 14
      Algorithm composite deflation of real polynomial with real root ............... 14
      Algorithm for composite deflation of real Polynomial with a complex root .. 15
      Algorithm for composite deflation of complex Polynomial with a complex root 16
   A suitable start guess ............................................................................................ 16
   Priori for the root with the smallest magnitude .................................................. 17
      Algorithm for starting point for Polynomial with real coefficients .......... 18
      Algorithm for starting point for Polynomial with complex coefficients ...... 18
      Algorithm Upper bound for the smallest magnitude root with real coefficients 19
      Algorithm Upper bound for the smallest magnitude root with complex coefficients .................................................. 19
   Priori Bounds for all roots ................................................................................... 20
      Algorithm Kalantari formula for real coefficients ........................................ 20
      Algorithm for Kalantaris formula for Complex coefficients ....................... 21
      Algorithm for Deutsch formula for real coefficients .................................... 21
      Algorithm for Deutsch formula for complex coefficients ......................... 21
   A Suitable stopping criteria .................................................................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error in arithmetic’s operations:</td>
<td>22</td>
</tr>
<tr>
<td>A simple upper bound:</td>
<td>23</td>
</tr>
<tr>
<td>A better upper bound:</td>
<td>23</td>
</tr>
<tr>
<td>Case 1: Stopping criteria</td>
<td>23</td>
</tr>
<tr>
<td>Case 2: Adams Stopping Criteria</td>
<td>24</td>
</tr>
<tr>
<td>Case 3: Grant &amp; Hitchins stopping criteria</td>
<td>25</td>
</tr>
<tr>
<td>Igarashi’s Stopping criteria</td>
<td>27</td>
</tr>
<tr>
<td>Garwick’s &amp; Ward’s</td>
<td>29</td>
</tr>
<tr>
<td>An even better upper bound</td>
<td>30</td>
</tr>
<tr>
<td>JLN Sopping criterion 1</td>
<td>30</td>
</tr>
<tr>
<td>JLN Stopping criterion 2</td>
<td>31</td>
</tr>
<tr>
<td>JLN Stopping criterion 3</td>
<td>31</td>
</tr>
<tr>
<td>JNL Stopping criterion 4</td>
<td>31</td>
</tr>
<tr>
<td>Polynomial Taylor Shifting</td>
<td>32</td>
</tr>
<tr>
<td>Algorithm for Polynomial Taylor shift with real coefficients</td>
<td>33</td>
</tr>
<tr>
<td>Algorithm for Polynomial Taylor shift with complex coefficients</td>
<td>33</td>
</tr>
<tr>
<td>Finding the roots of the Polynomial</td>
<td>35</td>
</tr>
<tr>
<td>Simple roots</td>
<td>35</td>
</tr>
<tr>
<td>Algorithm for eliminating zero roots for Polynomial with real coefficients</td>
<td>35</td>
</tr>
<tr>
<td>Algorithm for eliminating zero roots for Polynomial with complex coefficients</td>
<td>35</td>
</tr>
<tr>
<td>Algorithm for Quadratic roots for Polynomial</td>
<td>36</td>
</tr>
<tr>
<td>Algorithm for Quadratic roots for Polynomial with complex coefficients</td>
<td>37</td>
</tr>
<tr>
<td>Determine the multiplicity of a root</td>
<td>37</td>
</tr>
<tr>
<td>Newton and higher order derivative based methods</td>
<td>39</td>
</tr>
<tr>
<td>Newton method</td>
<td>39</td>
</tr>
<tr>
<td>Algorithm for the Newton method for Complex coefficients Polynomial</td>
<td>42</td>
</tr>
<tr>
<td>Halley method</td>
<td>44</td>
</tr>
<tr>
<td>Algorithm for the Halley method for Complex coefficients Polynomial</td>
<td>46</td>
</tr>
<tr>
<td>Householder 3(^{rd}) order method</td>
<td>48</td>
</tr>
<tr>
<td>Algorithm for the Householder method for Complex coefficients Polynomial</td>
<td>49</td>
</tr>
<tr>
<td>Ostrowski’s multi-point method</td>
<td>52</td>
</tr>
<tr>
<td>Algorithm for the Ostrowski’s multi-point method for Complex coefficients Polynomial</td>
<td>53</td>
</tr>
<tr>
<td>How the method and higher orders method stack up against each other</td>
<td>55</td>
</tr>
<tr>
<td>Laguerre’s method</td>
<td>56</td>
</tr>
<tr>
<td>Algorithm for the Laguerre’s method for Complex coefficients Polynomial</td>
<td>56</td>
</tr>
<tr>
<td>Matrix methods</td>
<td>59</td>
</tr>
</tbody>
</table>
Practical Implementation of Polynomial Root Finders

Eigenvalue Method ................................................................. 59
Algorithm for the Eigenvalue method for Complex coefficients Polynomial .... 59
Simultaneous method ............................................................... 66
Durand-Kerner method ............................................................ 66
Algorithm for the Durand-Kerner method for Complex coefficients Polynomial.
................................................................. 66
Aberth-Ehrlich method ............................................................. 68
Algorithm for the Aberth-Ehrlich method for Complex coefficients Polynomial
................................................................................................. 68
Rutishauser QD method ............................................................ 70
Algorithm for the Rutishauser QD method for Real coefficients Polynomial.... 71
Other Polynomial roots method ................................................ 74
Ostrowski Square root method ................................................ 74
Graeffe’s Root-Squaring method ............................................ 74
Bairstow’s Method ................................................................. 74
Algorithm for the Bairstow’s method for Real coefficients Polynomial........ 75
Reference .................................................................................... 77
Appendix ..................................................................................... 79
Jenkins-Traub ................................................................. 79
Algorithm for the Jenkins-Traub method for Complex coefficients Polynomial 79
Aberth-Ehrlich supporting functions ........................................ 91
Bairstow supporting functions ................................................. 93
Evaluation of Polynomials:

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

To evaluate a polynomial $P(z)$ where

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

We generally use Horner [1] 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 $P(z)$.

This evaluation of $P(z)$ requires therefore $n$ multiplications and $n$ additions for a total of $2n$ operations. The above mention recurrence works well for polynomial with real coefficients evaluated at a real point $x$, as well as for polynomials with complex coefficients evaluated at a complex point $Z = x + iy$ in which case multiplication and addition is replaced with the complex multiplication and addition for complex arithmetic given by:

Complex multiplication:  $(a+ib)(c+id) = (ac - bd) + i(ad+bc)$

Complex addition:  $(a+ib) + (c+id) = a + c + i(b+d)$

Since a Complex multiplication requires 4 ‘real’ multiplications and 2 additions the total number of operations involving is $4n+2n$ or $6n$ ‘real’ operations for polynomials with complex coefficients evaluated at a complex point.

In the case of a polynomial $P$ with real coefficients evaluated at a complex point $Z$ we in general are using Horner recurrence but in a special version using only real arithmetic:
$Z = x + iy$

$p = -2x$

$q = x^2 + y^2$

$b_n = a_n$

$b_{n-1} = a_{n-1} - pb_n$

$b_k = a_k - pb_{k+1} - qb_k + q = n - 2, \ldots, 1$

$b_0 = a_0 + xb_1 - qb_2$

$P(Z) = b_0 + iyb_1$

It therefore requires $4n$ operations instead of $2n$ for the real case to evaluate a polynomial with real coefficients and a complex point $Z$.

<table>
<thead>
<tr>
<th>Polynomial</th>
<th>Real coefficient</th>
<th>Complex coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real point</td>
<td>$2n$</td>
<td>$4n$</td>
</tr>
<tr>
<td>Complex point</td>
<td>$4n$</td>
<td>$6n$</td>
</tr>
</tbody>
</table>

Algorithm Horner with real coefficients at a real point

```c
double horner(const int n, const double a[], const double r, double *fz) {
    double fval;

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

    return fval;
}
```

Algorithm Horner with real coefficients at a complex point

```c
double horner(const int n, const double a[], const complex<double> z) {
    int i;
    double p, q, r, s, t;

    p = -2.0 * z.real();
    q = norm(z);
    s = 0; r = a[0];
    for (i = 1; i < n; i++)
        {t = a[i] - p * r - q * s;
         s = r;
         r = t;
        }
```
Algorithm Horner with complex coefficients at a complex point

```cpp
// Evaluate a polynomial with complex coefficients a[] at a complex point z and return the result fz
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
double horner( const int n, const complex<double> a[], const complex<double> z )
{
    complex<double> fval;
    fval = a[ 0 ];
    for( int i = 1; i <= n; i++ )
        fval = fval * z + a[ i ];
    return fval;
}
```

Deflation of a Polynomial:

For many methods e.g. Newton, Halley, Householder 3rd etc you find one or two roots at a time and then divided the found root up in the polynomial to deflate it and then continue using the iterations methods to found new roots until all the roots has been found.

For polynomial with real coefficients, you find either a real roots or a complex root. There is a special property for Polynomial with Real coefficients that complex root always appear in pairs as the complex root and its complex conjugated root. For Polynomial with complex coefficients, you only find one root at a time. Therefore, we have three scenario to deal with.

1) How to divide a real root up into the real polynomial
2) How to divide the complex root and the complex conjugated root up in the real polynomial
3) How to divide a complex root up in a polynomial with complex coefficients

If you have a polynomial with either real or complex coefficients:

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

And a root R (either real or a complex number). We are trying to find the deflated polynomial that satisfied the equation:

\[ P(z) = Q(z)(z - R) \]

where

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

and

\[ Q(z) = b_{n-1} z^{n-1} + b_{n-2} z^{n-2} + \ldots + b_1 z + b_0 \]
Practical Implementation of Polynomial Root Finders

Now to obtain the b’s you can either start with finding the highest coefficients \(b_{n-1}\) and work your way down to \(b_0\) which is called forward deflation or the opposite find the coefficients starting with \(b_0\) and work your way up to \(b_{n-1}\) which is called backward deflation.

**Forward Deflation of Polynomials:**

To do forward deflation we try to solve the equations starting with the highest coefficients \(a_n\):

\[
a_n z^n + a_{n-1} z^{n-1} + ... + a_1 z + a_0 = (b_{n-1} z^{n-1} + b_{n-2} z^{n-2} + ... + b_1 z + b_0)(z - R)
\]

The recurrence is given by:

\[
a_n = b_{n-1} \\
a_k = b_{k-1} - R \cdot b_k \quad k = n-1, ..., 1 \\
a_0 = -R \cdot b_0
\]

Now solve it for b’s you get:

\[
b_{n-1} = a_n \\
b_k = a_{k+1} + R \cdot b_{k+1} \quad k = n-2, ..., 0
\]

This simple algorithm works well for polynomials with real coefficients and real roots and complex coefficients with complex roots basically using the same recurrence just using complex arithmetic instead. A special case is real coefficients with complex roots. A complex roots and its complex conjugated root will be the same as divided the polynomial \(P(Z)\) with 2\(^{nd}\) order polynomial of the two complex conjugated roots \((x+i y)\) and \((x-i y)\) or \((z^2-2xz+(x^2+y^2))\). Letting \(r=2x\) and \(u=x^2+y^2\)

\[
P(z) = Q(z)(z^2 + rz + u)
\]

where \(P(z) = a_n z^n + a_{n-1} z^{n-1} + ... + a_1 z + a_0\)

and \(Q(z) = b_{n-2} z^{n-2} + b_{n-3} z^{n-3} + ... + b_1 z + b_0\)

The recurrence is giving by:

\[
a_n = b_{n-2} \\
a_{n-1} = b_{n-3} + r b_{n-2} \\
a_{n-2} = b_{n-4} + r b_{n-3} + u b_{n-2} \\
a_2 = b_0 + r b_1 + u b_2 \\
a_1 = r b_0 + u b_1 \\
a_0 = u b_0
\]

Now solve it for b’s you get:
Practical Implementation of Polynomial Root Finders

\[
\begin{align*}
    b_{n-2} &= a_n \\
    b_{n-3} &= a_{n-1} - r \times b_{n-2} \\
    b_k &= a_{k+2} - rb_{k+1} - ub_{k+2} \quad k = n-4, \ldots, 0
\end{align*}
\]

Algorithm Forward deflation with real coefficients with a real root

```c
// Real coefficients and Real root with forward deflation.    // Return the new degree of the deflated polynomial and the result in a[0..n-1]  // Notice that a[0] is a_n, a[1] is a_{n-1} and a[n]=a_0    //
int forwarddeflation( const int n, double a[], const double x )
{
    int i;
    double r;

    for( r = 0, i = 0; i < n; i++ )
        a[ i ] = r = r * x + a[ i ];
    return n - 1;
}
```

Algorithm Forward deflation with real coefficients with a complex root

```c
// Complex root forward deflation for real coefficients  // Return the new degree of the deflated polynomial and the result in a[0..n-2]  // Notice that a[0] is a_n, a[1] is a_{n-1} and a[n]=a_0    //
int forwarddeflation( const int n, double a[], const double z )
{
    int i;
    double r, u;

    r = -2.0 * z.real();
    u = z.norm();
    a[ 1 ] -= r * a[ 0 ];
    for( i = 2; i < n - 1; i++ )
        a[ i ] = a[ i ] - r * a[ i - 1 ] - u * a[ i - 2 ];
    return n - 2;
}
```

Algorithm Forward deflation with complex coefficients with a complex root

```c
// Complex root forward deflation for complex coefficients.    // Return the new degree of the deflated polynomial and the result in a[0..n-1]  //
```
Backward Deflation of Polynomials:

To do backward deflation we try to solve the equations starting with the lowest coefficients \(a_0\) and work our way up to \(a_n\):

\[
a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0 = (b_{n-1} z^{n-1} + b_{n-2} z^{n-2} + \ldots + b_1 z + b_0)(z - R)
\]

The recurrence is given by:

\[
a_0 = -R b_0
\]

\[
a_k = b_{k-1} - R b_k \quad k = 1, \ldots, n - 1
\]

\[
a_n = b_{n-1}
\]

Now solve it for b’s you get:

\[
b_0 = -\frac{a_0}{R}
\]

\[
b_k = (b_{k-1} - a_k) / R \quad k = 1, \ldots, n - 2
\]

\[
b_{n-1} = a_n
\]

For complex conjugated roots we again divide the quadratic factor \((z^2 - 2xz + (x^2 + y^2))\) up in the polynomial \(P(z)\) this time starting from the back. Letting \(r = -2x\) and \(u = x^2 + y^2\)
Practical Implementation of Polynomial Root Finders

\[ P(z) = Q(z)(z^2 + rz + u) \]

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

and \( Q(z) = b_{n-2} z^{n-2} + b_{n-3} z^{n-3} + \ldots + b_1 z + b_0 \)

The recurrence is giving by:

\[
\begin{align*}
 a_0 &= ub_0 \\
 a_i &= ub_i + rb_0 \\
 a_k &= ub_k + rb_{k-1} + b_{k-2} & k = 2,\ldots,n-2 \\
 a_{n-1} &= rb_{n-2} + b_{n-3} \\
 a_n &= b_{n-2}
\end{align*}
\]

Now solve it for \( b \)’s and you get

\[
\begin{align*}
 b_0 &= a_0 / u \\
 b_i &= (a_i - r* b_0) / u \\
 b_k &= (a_k - b_{k-2} - rb_{k-1}) / u & k = 2,\ldots,n-2
\end{align*}
\]

Algorithm Backward deflation with real coefficients with a real root

```c
// Real root backward deflation for real Polynomial coefficients.
// Return the new degree of the deflated polynomial and the result in a[0..n-1]
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// int backwarddeflation(const int n, double a[], const double x)
//{
//  int i;
//  double r, s, t;
//  if (x != 0.0)
//    for (r = 0, t=a[n], i = n - 1; i >= 0; i--)
//      { s = t; t = a[i];
//        a[i] = r = (r - s) / x;
//      }
//    return n - 1;
//}
```

Algorithm Backward deflation with real coefficients with a complex root

```c
// Complex root forward deflation for real coefficients
// Return the new degree of the deflated polynomial and the result in a[0..n-2]
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// int backwarddeflation(const int n, double a[], const complex<double> z)
```
{ int i; double r, s, t, u, v;  
  r = -2.0 * z.real(); 
  u = norm(z); 
  s = a[n - 1];  t = a[n - 2];  v = a[n - 3];  
  a[n - 2] = a[n] / u; 
  a[n - 3] = (s - r * a[n - 2]) / u; 
  for (i = n - 4; i >= 0; i--)  
  { 
    s = t;  t = v;  v = a[i]; 
    a[i] = (s - r * a[i + 1] - a[i + 2]) / u; 
  }  
  return n - 2;  
}

Algorithm Backward deflation with complex coefficients with a complex root

// Complex root forward deflation for complex coefficients.  
// Return the new degree of the deflated polynomial and the result in a[0..n-1]  
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0  
// int backwarddeflation(const int n, complex<double> a[], const complex<double> z)  
{ int i;  complex <double> z0 = 0, s, t; 
  if (z != z0)  
    for (i = n - 1, t = a[n]; i >= 0; i--)  
    { 
      s = t;  t = a[i]; 
      a[i] = (z0 - s) / z; 
    }  
  return n - 1;  }

Forward or Backward Deflation?

Wilkinson [2] has shown that in order 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.

Although most root finding algorithm do find them in increasing order, it can’t be guaranteed and therefore in order to ensure the most stable deflation process you will use the composite deflation method, which is more complicated to handle than the forward or backward deflation technique.
**Composite Deflation of Polynomials:**

To carry out composite deflation you calculated the new coefficients by doing a forward deflations and saving the new coefficients in an array \( B[] \). The do a backward deflations and saying the new coefficients in an array \( C[] \). You then join the arrays \( B[] \) and \( C[] \) by finding the coefficients index with the lowest difference in the magnitude between the new calculated coefficients \( k \). You then take the forward deflation coefficients from the \( B[] \) from \( n..k+1 \) and the backward coefficients \( C[] \) from \( k-1..0 \) and the take the average for the coefficients \( k \) as \( b_k = \frac{1}{2} (B[k]+C[k]) \).

We then have the algorithm as follows to calculate the new coefficients \( b \)'s:

```
r=+Infinity
For( i=0..n-1)
    u=|B[i]|+|C[i]|
    If(u!=0) u=|B[i]-C[i]|/u
    If(u<r) u=r, k=i
For(i=k+1..n-1) b[i]=B[i];
bk= ½ (B[k]+C[k])
For(i=k-1..0) b[i]=C[i];
```

**Algorithm composite deflation of real polynomial with real root**

```c
// Real Polynomial and real root composite deflation.
// Return the new degree of the deflated polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
int compositedeflation(const int n, double a[], double z)
{
    int i, k;
    double r, u;
    double *b = new double[n], *c = new double[n];
    // Forward & Backward deflation
    for (r = 0, u = 0, i = 0; i < n; i++)
    {
        b[i] = r = r*z + a[i];
        c[n - i - 1] = u = (u - a[n - i]) / z;
    }
    // Join
    for (r = DBL_MAX, i = 0; i < n; i++)
    {
        u = fabs(b[i]) + fabs(c[i]);
        if (u != 0)
            { u = fabs(b[i] - c[i]) / u;
              if (u<r)
                  { r = u; k = i; }
            }
    }
    for (i = k - 1; i >= 0; i--)
        a[i] = b[i]; // Forward deflation coefficient
```
Algorithm for composite deflation of real Polynomial with a complex root

// Real Polynomial and complex root composite deflation.
// Return the new degree of the deflated polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0

int compositedeflation(const int n, double a[], complex<double> z)
{
    int i, k;
    double r, u;
    double *b = new double[n], *c = new double[n];

    // Forward & Backward deflation
    r = -2.0*z.real();
    u = norm(z);
    b[0] = a[0]; b[1] = a[1] - r*b[0];
    c[n - 2] = a[n] / u; c[n - 3] = (a[n - 1] - r*c[n - 2]) / u;
    for (i = 2; i < n - 1; i++)
    {
        b[i] = a[i] - r*b[i - 1] - u*b[i - 2];
        c[n - 2 - i] = (a[n - i] - c[n - i] - r*c[n - i - 1]) / u;
    }

    // Join
    for (r = DBL_MAX, i = 0; i < n - 1; i++)
    {
        u = fabs(b[i]) + fabs(c[i]);
        if (u != 0)
        {
            u = fabs(b[i] - c[i]) / u;
            if (u < r)
            {
                r = u; k = i;
            }
        }
    }

    for (i = k - 1; i >= 0; i--)
    {
        a[i] = b[i]; // Forward deflation coefficient
        a[k] = 0.5*(b[k] + c[k]);
    }
    for (i = k + 1; i < n - 1; i++)
    {
        a[i] = c[i]; // Backward deflation coefficient
    }
    delete[] b, c;
    return n - 2;
}
Algorithm for composite deflation of complex Polynomial with a complex root

// Complex Polynomial and complex root composite deflation.
// Return the new degree of the deflated polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a

int compositedeflation(const int n, complex<double> a[], complex<double> z)
{
    int i, k;
    double ua, ra;
    complex<double> r, u;
    complex<double> *b = new complex<double>[n], *c = new complex<double>[n];

    // Forward & Backward deflation
    for (r = 0, u = 0, i = 0; i < n; i++)
    {
        b[i] = r = r*z + a[i];
        c[n - i - 1] = u = (u - a[n - i]) / z;
    }

    // Join
    for (ra = DBL_MAX, i = 0; i < n; i++)
    {
        ua = abs(b[i]) + abs(c[i]);
        if (ua != 0)
        {
            ua = abs(b[i] - c[i]) / ua;
            if (ua < ra)
            {
                ra = ua; k = i;
            }
        }
    }

    for (i = k - 1; i >= 0; i--)
    {
        a[i] = b[i]; // Forward deflation coefficient
        a[k] = 0.5*(b[k] + c[k]);
        for (i = k + 1; i < n; i++)
            a[i] = c[i]; // Backward deflation coefficient
    }
    delete[] b, c;
    return n - 1;
}

A suitable start guess

To make the iterative methods faster to converge to Polynomial roots it is important that we somehow start and a suitable point that are in the neighborhood of a root. Luckily, many people has study this field and there is impressive 45 + methods for creating a priori bound of the roots as outline by J.McNamee, Numerical Methods for roots of Polynomials [7]. Most of the priori bounds is for finding the radius of a circle where all the roots are located within. 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 were the strategy is to find the roots with increasing order of magnitude.

**Priori for the root with the smallest magnitude.**

Most root finding implementation that I have seen do not pay too much attention to the starting point. E.g. [4] Grant-Hichins use 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 [5]. Were we find the starting point $z_0$ where the root with the smallest magnitude lies outside this circle:

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

The smallest root are located outside the circle with radius 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 yield a starting point $z_0=0.68$ which is close to the nearest root of $x=1$.

Now consider the Polynomial:

$$P(x) = (x-1)(x+2000)(x-3000)(x-4000) = x^4 + 2999x^3 - 10003E3x^2 - 2399E7x + 24E9$$

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

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

Sine we always choose an initial guess where the root with the smallest modulo is located outside this circle it could be handy if we could bound the upper radius where we are sure that the root with the smallest magnitude is located inside that circle.

We do have such a formula that can determine the upper radius for the root with the smallest magnitude:

$$\text{Radius} = \min \left\{ n \left| \frac{a_0}{a_1} \right| , \sqrt{ \frac{|a_0|}{a_n} } \right\}$$
Using the same Polynomial as above you get for \( P(x) = (x-1)(x+2)(x-3)(x-4) = x^4 + 2x^3 - 13x^2 - 14x + 24 \) and find the Radius to be 2.213. Which then bounds the first root to be between 0.68 < |z| < 2.2.

For \( P(x) = (x-1)(x+2000)(x-3000)(x-4000) = x^4 + 2999x^3 - 10003E3x^2 - 2399E7x + 24E9 \) you get a Radius of 4.002 which bound the root to be: 0.5 < |z| < 4.0

and finally with the Polynomial \( P(x) = (x+1000)(x-2000)(x+3000) = x^3 + 2E3x^3 - 5E6x^2 - 6E9 \) you get a Radius of 1,817 and the root is bound to be between: 600 < |z| < 1,817.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Startpoint</th>
<th>Priori Smallest</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^3 - 6x^2 + 11x - 6 = (x-1)(x-2)(x-3) )</td>
<td>0.27</td>
<td>1.64</td>
</tr>
<tr>
<td>( X^4 + 2x^3 - 13x^2 - 14x + 24 = (x-1)(x-2)(x-3)(x-4) )</td>
<td>0.68</td>
<td>2.2</td>
</tr>
<tr>
<td>( X^5 + 2000x^2 - 5E6x - 6E9 = (x+1000)(x-2000)(x+3000) )</td>
<td>600</td>
<td>1817</td>
</tr>
<tr>
<td>( X^6 + 2999x^3 - 10003E3x^2 - 2399E7x + 24E9 = (x-1)(x+2000)(x-3000)(x-4000) )</td>
<td>0.5</td>
<td>4.0</td>
</tr>
<tr>
<td>( X^5 - 1 )</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Algorithm for starting point for Polynomial with real coefficients**

```cpp
// Calculate a start point for the iteration that is suitable for
// finding zeros with increasing magnitude
// Start point calculation for a polynomial with real coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is a0, a[1] is an-1 and a[n]=a0
double startpoint( const int n, const double a[] )
{
    int i;
    double r, u, min;

    // Determine starting point
    r = log( fabs( a[ n ] ) );
    min = exp( ( r - log( fabs( a[ 0 ] ) ) ) / n );
    for( i = 1; i < n; i++ )
    {
        if( a[ i ] != 0 )
        {
            u = exp( ( r - log( fabs( a[ i ] ) ) ) / ( n - i ) );
            if( u < min )
            {
                min = u;
            }
        }
    }
    return 0.5*min;
}
```

**Algorithm for starting point for Polynomial with complex coefficients**
// Calculate a start point for the iteration that is suitable for
// finding zeros with increasing magnitude
// Start point calculation for a polynomial with complex coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
double startpoint(const int n, const complex<double> a[]) {
    double r, min, u;
    r = log(abs(a[n]));
    min = exp((r - log(abs(a[0]))) / n);
    for (int i = 1; i < n; i++)
        if (a[i] != complex<double>(0, 0)) {
            u = exp((r - log(abs(a[i]))) / (n - i));
            if (u < min)
                min = u;
        }
    return 0.5 * min;
}

Algorithm Upper bound for the smallest magnitude root with real coefficients

// Find the circle where the smallest magnitude root is located within
// Polynomial with real coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// double prioriSmallest(const int n, const double a[]) {
//    double min, min2=DBL_MAX;
//    min = pow(fabs(a[n]) / fabs(a[0]), 1.0 / n);
//    if (a[n - 1] != 0.0)
//        min2 = n*(fabs(a[n]) / fabs(a[n - 1]));
//    if (min2 < min)
//        min = min2;
//    return min;
//}

Algorithm Upper bound for the smallest magnitude root with complex coefficients

//Find the circle where the smallest magnitude root is located within
// Polynomial with Complex coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// double prioriSmallest(const int n, const complex<double> a[]) {
//    double min, min2=DBL_MAX;
//    min = pow(abs(a[n]) / abs(a[0]), 1.0 / n);
//    if (a[n - 1] != 0.0)
//        min2 = n*(abs(a[n]) / abs(a[n - 1]));
//    if (min2 < min)
//        min = min2;
The start point algorithm has proven to be very useful since you will always start your search somehow closed to the nearest root reducing the amount of iteration you would need to perform.

**Priori Bounds for all roots**

J. McNamee [7] did and extensive study of more than 45 methods to obtain the bounds and found that two methods yields the most accurate result among a high number of random polynomials with vary degree.

The two methods was Kalantari’s formula and the Deutsch’s ‘Simple’ formula. Where it was found that Kalantari’s methods has 30% closer bounds than Deutsch’s formula.

Kalantari’s formula:

Radius for all roots, $\rho$ (assuming Polynomial is in monic form):

$$\rho = \max_{i=1,...,n} |\rho_i|$$

$$|\rho| \leq \frac{1}{0.682328} \max_{k=4,...,n+3} \left\{ \frac{a_{n-1}a_{n-k+3} - a_{n-1}a_{n-k+2}}{a_{n-2}a_{n-k+3} + a_{n-k+1}|1|^{1/k-1}} \right\}$$

Where $a_{-1} = a_{-2} = 0$

Deutsch’s ‘simple’ formula (assuming Polynomial is in monic form):

$$|\rho| \leq |a_{n-1}| + \max_{i=0,...,n-2} \left\{ \frac{a_i}{c_{i+1}} \right\}$$

Algorithm Kalantari formula for real coefficients

```c
// Kalantaris formula for priori upper bound for largest root
// Polynomial with Real coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
double prioriKalantaris(const int n, double a[])
{
    double r, s, t, u, max = 0, a0 = a[ 0 ], a1, a2;
    r = 0; s = 0; t = a[n] / a0;
    a1 = a[1] / a0; a2 = a[2] / a0;
    for (int k = n; k >= 1; k--)
    {
        u=fabs( (a1 * t - s) *a1 - a2 * t + r);
        u = pow(u, 1.0 / (k + 2));
        if (u > max )
            max = u;
        r = s; s = t; t = a[k - 1] / a0;
    }
    return min;
}
```
Algorithm for Kalantaris formula for Complex coefficients

```cpp
// Kalantaris formula for priori upper bound for largest root
// Polynomial with Complex coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is a₀, a[1] is an-1 and a[n]=a₀
double prioriKalantaris(const int n, complex<double> a[])
{
    double u, max = 0;
    complex<double> r, s, t, a0 = a[0], a1, a2;
    r = 0; s = 0; t = a[n] / a0;
    a1 = a[1] / a0; a2 = a[2] / a0;
    for (int k = n; k >= 1; k--)
    {
        u = abs((a1 * t - s) * a1 - a2 * t + r);
        u = pow(u, 1.0 / (k + 2));
        if (u > max)
            max = u;
        r = s; s = t; t = a[k - 1] / a0;
    }
    max /= 0.682328;
    return max;
}
```

Algorithm for Deutsch formula for real coefficients

```cpp
// Deutsch's simple formula for priori upper bound for largest root
// Polynomial with Real coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is a₀, a[1] is an-1 and a[n]=a₀
double prioriDeutsch(const int n, const double a[])
{
    int k;
    double max=0, r;
    for (k = n; k > 1; k--)
    {
        r = fabs(a[k] / a[k - 1]);
        if (r > max)
            max = r;
    }
    max += fabs(a[1] / a[0]);
    return max;
}
```

Algorithm for Deutsch formula for complex coefficients

```cpp
// Deutsch's simple formula for priori upper bound for largest root
// Polynomial with Complex coefficients a[]
// n is the degree of the Polynomial
// Notice that a[0] is a₀, a[1] is an-1 and a[n]=a₀
```
```cpp
double prioriDeutsch(const int n, const complex<double> a[]) {
    int k;
    double max = 0, r;
    for (k = n; k > 1; k--)
    {
        r = abs(a[k] / a[k - 1]);
        if (r > max)
            max = r;
    }
    max += abs(a[1] / a[0]);
    return max;
}
```

The below table shows a few polynomials and how the Kalataris and Deutsch formula stack up against each other.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Kalantaris</th>
<th>Deutsch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomials</td>
<td>Radius</td>
<td>Radius</td>
</tr>
<tr>
<td>$x^3 - 6x^2 + 11x - 6 = (x-1)(x-2)(x-3)$</td>
<td>6.57</td>
<td>7.83</td>
</tr>
<tr>
<td>$x^3 + 2000x^2 - 5E6x - 6E9 = (x+1000)(x-2000)(x+3000)$</td>
<td>4,106</td>
<td>4,500</td>
</tr>
<tr>
<td>$x^5 + 2999x^4 - 10003E3x^3 + 2399E7x + 24E9 = (x-1)(x+2000)(x-3000)(x+4000)$</td>
<td>5,831</td>
<td>6,334</td>
</tr>
<tr>
<td>$x^5 - 1$</td>
<td>1.46</td>
<td>Infinity</td>
</tr>
</tbody>
</table>

The priori bounds can be used if you strategy is to find roots with decreasing magnitude since you would know what your max startpoint should be (Kalantaris formula) to start the root search.

For simultaneous methods e.g. Durand-Kerner or Aberth-Ehrlich we need to use a different strategy since we do not find one (or two roots) at a time but iterate simultaneously to all roots. Therefore, we also need to find a suitable starting point for each roots. With the Priori bounds for all roots, we at least know in what range (max radius) we need to start the search within.

**A Suitable stopping criteria**

When doing 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 finder 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. It is not all method that use the value of $P(z)$ as the stopping criteria. Typically Matrix method that do not rely on evaluation of $P(z)$ use a different approach discuss later on.

**Error in arithmetic's operations:**

J.H.Wilkinson in “Rounding errors in algebraic processes” [6] has showed that the errors in performing algebraic operations are bound by:
\( \varepsilon < \frac{1}{2} \beta^{1-t} \quad \beta \text{ is the base, and } t \text{ is the precision} \) (Assuming round to nearest)

For the Intel microprocessor series and the IEEE754 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 above information for a polynomial with degree \( n \).

<table>
<thead>
<tr>
<th>Number of operations:</th>
<th>Polynomials Real coefficient</th>
<th>Complex coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real point</td>
<td>(</td>
<td>a_0</td>
</tr>
<tr>
<td>Complex point</td>
<td>(</td>
<td>a_0</td>
</tr>
</tbody>
</table>

**A better upper bound.**

In this category, we have among others Adams [1] and Grant & Hitchins [2] stopping criteria for polynomials.

Polynomials root finders usually can handle polynomials with both real and complex coefficients evaluated at a real or complex number. In principle, we have 3 different scenarios (real coefficients at a real point, real coefficients at a complex point and complex coefficients at a complex point) that we must deal with to calculate a root to the limitations of the machine precision. Since the bound of the round off errors is different for these 3 scenarios we need to evaluate them individually.

**Case 1: Stopping criteria**

Polynomial with real coefficients \( a_n \) evaluated at a real point \( x \), using Horner’s method:

\[
\begin{align*}
  b_n &= a_n \\
  b_k &= b_{k+1}x + a_k & k = n-1,\ldots,0
\end{align*}
\]

And error bound can be computed using similar recurrence as follows, see Kahan[7]:

---

3 May 2020

Page 23
Practical Implementation of Polynomial Root Finders

\[ e_n = |b_n| \frac{1}{2} \]

\[ e_k = e_{k-1} |x| + |b_k| \quad k = n-1, \ldots, 0 \]

\[ e = (4e_0 - 2|b_0|) e \quad \text{where} \quad e = \frac{1}{2} \beta^{-r} \]

Algorithm Kahan Stopping Criteria

```cpp
// Calculate the upper bound for the rounding errors performed in a polynomial with real coefficient a[] at a real point z. Kahan
// double upperbound(const int n, const double a[], const double r)
{
    int i;
    double t, e;
    t = a[0]; e = abs(t)*(0.5);
    for (i = 1; i<n; i++)
    {
        t = t*r + a[i];
        e = abs(r)*e + abs(t);
    }
    e = (2 * e - abs(t))*pow((double)DBL_RADIX, -DBL_MANT_DIG + 1);
    return e;
}
```

Case 2: Adams Stopping Criteria

Polynomial with real coefficients \(a_n\) evaluated at a complex point \(z\), using Horner’s method.

\[ Z = x + iy \]
\[ p = -2x \]
\[ q = x^2 + y^2 \]
\[ b_n = a_n \]
\[ b_{n-1} = a_{n-1} - pb_n \]
\[ b_k = a_k - pb_{k+1} - qb_{k+2} \quad k = n-2, \ldots, 1 \]
\[ b_0 = a_0 + xb_1 - qb_2 \]
\[ P(Z) = b_0 + iyb_1 \]

Adams [1] has shown that an error bound can be computed using the following recurrence:
Algorithm Adams Stopping Criteria

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

Case 3: Grant & Hitchins stopping criteria

Polynomial with complex coefficients $z_n$ evaluated at a complex point $z$, using Horner’s method. This gets a little bit more complicated. Grant and Hitchins [2] derive an upper error bound for the errors in evaluating the polynomial as follows

$$P(Z) = (a_n + ib_n)Z^n + (a_{n-1} + ib_{n-1})Z^{n-1} + ... + (a_1 + ib_1)z + (a_0 + b_0)$$

Using the Horner’s method and keeping track on the real component $c_k$ and the imaginary component $d_k$ of the coefficient separately we get:
Using these values an error bound can now be calculated using the recurrence:

\[
g_n = 1, \quad h_n = 1
\]

\[
g_k = |x(g_{k+1} + |c_{k+1}|)| + |y(h_{k+1} + |d_{k+1}|)| + |a_k| + 2|c_k| \quad k = n-1, \ldots, 0
\]

\[
h_k = |y(g_{k+1} + |c_{k+1}|)| + |x(h_{k+1} + |d_{k+1}|)| + |b_k| + 2|d_k|
\]

A now the error is \((g_0 + ih_0)e\), where \(e = \frac{1}{2} \beta^{1-t}\). Now since the recurrence in itself introduce error \([2]\) 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 in a complex point:

\[
e = (g_0 + ih_0)e(1 + e)^5n \quad \text{where} \quad e = \frac{1}{2} \beta^{1-t}
\]

Other methods in this category is Igarshi’s, Garwick’s and Ward’s. The nice parts with these stopping criteria are that they do not discriminate whether the polynomial is with real or complex coefficients at a real or complex point as long as the calculation is done with proper respect for the type of the coefficient and the type of evaluation point.

**Algorithm Grant & Hitchins Stopping Criteria**

```cpp
// Calculate a upper bound for the rounding errors performed in a
// polynomial with complex coefficient a[] at a complex point z. ( Grant & Hitchins test )

double upperbound(const int n, const complex<double> a[], complex<double> z)
{
    int i;
    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 (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(nc); 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;
    }
}
```
Igarashi’s Stopping criteria

Igarashi’s suggested back in 1984 a new stopping criterion for finding the roots of the polynomial $P(z)$.

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

Igarashi’s suggested a stopping criterion after the $i$'th iteration when:

$$|P(z_i) - B(z_i)| \geq \min(|P(z_i)|, |B(z_i)|)$$

Where $B(z) = zP'(z) - C(z)$ and $C(z) = zP'(z) - P(z)$. Of course, they have to be evaluated prior to the subtraction and you get the following two evaluation that can be calculated using the Horner methods.

$$zP'(z) = n a_n z^n + (n-1) a_{n-1} z^{n-1} + \ldots + a_1 z$$

$$C(z) = (n - 1) a_n z + (n - 2) a_{n-1} z^{n-1} + \ldots + a_2 z^2 - a_0$$

Initially when you are far from the root the $|P(z_i) - B(z_i)|$ will be smaller than $\min(|P(z_i)|, |B(z_i)|)$, however as you approach the root both $P(z_i)$ and $B(z_i)$ will go towards zero but then $|P(z_i) - B(z_i)|$ will be dominated by the round-off errors and become larger than $\min(|P(z_i)|, |B(z_i)|)$ providing a suitable stopping criteria for the root search.

Igarashi suggest that the search will terminate if one of the three conditions arise:

a) If $P(z_i)$ or $B(z_i) = 0.0$

b) If $P(z_i)B(z_i) < 0$

c) If $P(z_i)B(z_i) > 0$ and ($2|P(z_i)| \leq |B(z_i)|$ or $2|B(z_i)| \leq |P(z_i)|$)

Algorithm Igarashi with real coefficients at a real point

```cpp
// Igarashi stopping criteria for Polynomial with real coefficients
// at a real point r
// n is the degree of the polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// bool Igarashi(const int n, const double a[], const double r)
// {
// double *zP = new double[n+1];
// double *C = new double[n+1];
// double px, zpx, cx, bx;
// for (int i = 0; i <= n; i++)
// {
// zP[i] = (n - i) * a[i];
// C[i] = (n - i - 1) * a[i];
// }
// e = abs(complex<double>(ng,nh) ) * pow(1 + tol, 5 * n) * tol;
// return e;
// }
```
Algorithm Igarashi with real coefficients at a complex point

```cpp
// Igarashi stopping criteria for Polynomial with real coefficients
// at a complex point z
// n is the degree of the polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// bool Igarashi(const int n, const double a[], const complex<double> z)
// {
//     double *zP = new double [n + 1];
//     double *C = new double [n + 1];
//     complex<double> px, zpx, cx, bx;
//     for (int i = 0; i <= n; i++)
//         {zP[i] = (double)(n - i) * a[i];
//         C[i] = (double)(n - i - 1) * a[i];
//         } horner(n, a, z, &px);
//     horner(n, zP, z, &zpx);
//     horner(n, C, z, &cx);
//     bx = zpx - cx;
//     delete[] zP, C;
//     if (px == 0.0 || bx == 0.0) return true;
//     if (px*bx < 0) return true;
//     if (2 * fabs(px) <= fabs(bx) || 2 * fabs(bx) <= fabs(px)) return true;
//     return false;
// }
```

Algorithm Igarashi with complex coefficients at a complex point

```cpp
// Igarashi stopping criteria for Polynomial with complex coefficients
// at a complex point z
// n is the degree of the polynomial
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
// bool Igarashi(const int n, const complex<double> a[], const complex<double> z)
// {
//     complex<double> *zP = new complex<double> [n + 1];
//     complex<double> *C = new complex<double> [n + 1];
//     complex<double> px, zpx, cx, bx;
//     for (int i = 0; i <= n; i++)
//         {zP[i] = (double)(n - i) * a[i];
//         C[i] = (double)(n - i - 1) * a[i];
//         } horner(n, a, z, &px);
//     horner(n, zP, z, &zpx);
//     horner(n, C, z, &cx);
//     bx = zpx - cx;
//     delete[] zP, C;
//     if (px == 0.0 || bx == 0.0) return true;
//     if (px.real()*bx.real() < 0 || px.imag() * bx.imag() < 0) return true;
//     if (2 * abs(px) <= abs(bx) || 2 * abs(bx) <= abs(px)) return true;
//     return false;
// }
```
Garwick’s & Ward’s
Garwick (see JLN[5]) introduce this very simple stopping criterion that states that when the increment from two iterative steps $e_i > e_{i-1}$, where $e_i = |z_i - z_{i-1}|$ then the root $z_{i-1}$ is found. When convergence has first started then the rate of convergence does not decrease until a root is found. Ward (see JLN[8]) improve on the initial problem with Garwick precondition issue and states the following stopping criterion:

$$z_{i-1} \text{ is a root if } e_i > e_{i-1}, \text{where } e_i = |z_i - z_{i-1}|$$

JLN [8] replace it to:

$$z_{i-1} \text{ is a root if } e_i \geq e_{i-1}, \text{where } e_i = |z_i - z_{i-1}|$$

After numerical results, shows Ward original failed to stop under certain conditions.

In addition, the following preconditions holds:

1. \[ e_i \leq 10^{-7} \text{ if } |z_{i-1}| < 10^{-4} \]
2. \[ \frac{e_i}{|z_{i-1}|} \leq 10^{-3} \text{ if } |z_{i-1}| \geq 10^{-4} \]

Algorithm Garwick & Ward with real roots

```cpp
// Garwick stopping criteria.
// r, r1 & r2 is the 3 latest root estimations.
// Convergence rate only decrease due to rounding errors then
// we continue until the new r has a larger step size than the previous
// r1 (due to round off errors)
// return true if stopping criteria has been reach otherwise false
//
bool Garwick(const double r, const double r1, const double r2 )
{
    double e1, e2;

    e1 = fabs(r - r1); // Newest stepsize
    e2 = fabs(r1 - r2); // Previous stepsize
    if( fabs(r1) < 1E-4 && e1 <= 1E-7 ||
        fabs( r1 ) >= 1E-4 && e1/fabs(r1)<=1E-3)
        if (e1 >= e2) return true;
    return false;
}
```
Algorithm Garwick & Ward with complex root

```cpp
// Garwick stopping criteria.
// z, z1 & z2 is the 3 latest root estimations.
// Convergence rate only decrease due to rounding errors then
// we continue until the new z has a larger step size than the previous
// z1 (due to round off errors)
// return true if stopping criteria has been reach otherwise false

bool Garwick(const complex<double> z, const complex<double> z1, const complex<double> z2)
{
    double e1, e2;
    e1 = abs(z - z1); // Newest stepsize
    e2 = abs(z1 - z2); // Previous stepsize
    if (abs(z1) < 1E-4 && e1 <= 1E-7 ||
        abs(z1) >= 1E-4 && e1 / abs(z1) <= 1E-3 )
        if (e1 >= e2) return true;
    return false;
}
```

An even better upper bound

JL Nikolajsen [8] wrote an excellent paper suggesting a new stopping criterion for iterative root finding. His suggesting eliminate unnecessary function evaluations and immediately stop the iterations when no further improvement to the roots is possible. JLN outline four possible stopping criteria capable of also handling ill-conditioned root. The method works equally well for both real and complex roots. Instead of repeating JLN finding I will just summarized the 4 different stopping criteria

**JLN Sopping criterion 1**

\[ z_i \text{ is a root if } \frac{s_i^2}{s_{i-1}} \geq s_m \]

Precondition: \( s_{i-1} \geq \frac{s_m}{q_m^2} \)

\( S_i \) is the number of matching leading bits (MLBs) of the two successive iterates \( z_{i-1} \) and \( z_i \), \( s_m \) is the length of the IEEE 754 floating point double precision e.g. \( s_m=53 \) bits and \( q_m \) is the convergence order of the iterative method used. E.g. Newton is 2, Halley is 3 and Laguerre is also 3 etc.
**JLN Stopping criterion 2**

\[ z_{i+1} \text{ is a root if } \frac{s_i^2}{s_{i-1}} > s_{i+1} \]

*Preconditions:* \( s_{i-1} \geq \frac{s_m}{q_m^2} \) and \( s_i - s_{i-1} \geq \frac{s_m}{q_m^2} \)

This stopping criteria is used when criterion 1 convergence rate is not quite fast enough to trigger the stopping criterion 1.

**JLN Stopping criterion 3**

Stopping criterion 3 is used to catch stop after a single iteration if needed and comes in two sub-criteria

\[ z_i \text{ is a root if} \]

1: \( z_0 \neq 0 \) and \( s_1 \geq \frac{s_m}{2} \)
2: \( z_0 = 0 \) and \( s_1 \geq s_m \)

\[ z_i \text{ is a root if} \]

1: \( s_i - s_{i-1} \geq \frac{s_m}{2} \) or
2: \( s_1 - s_{i-1} \geq \frac{s_m}{4} \) and \( s_{i+1} - s_i < s_i - s_{i-1} \) when \( i \geq 2 \)

**JNL Stopping criterion 4**

The last stopping criteria is.

\[ z_{i+1} \text{ is a root if } s_{i+2} \leq s_{i+1} \text{ with the precondition: } \]
\[ s_{i-1} \geq b, s_i \geq b \text{ and } b = 8 \]

As already, mention I encourage readers to study JLN method [8] in details and JLN more elaborated explanation and details of the method.
Polynomial Taylor Shifting

Sometimes it can be practical not to solve a given Polynomial directly but instead solve a Polynomial where all the roots are shifted a certain distance from the original polynomial. A classic example is Rutishauser QD method for finding Polynomial roots. One of the drawbacks with the Rutishauser QD method is that it requires all coefficients to be $a_i \neq 0$ for $i = 0, \ldots, n$.

e.g. $x^5 - 1$ can’t be solved with that method. However if we Taylor shift the roots to the left with 2 we get a new Polynomial $x^5 + 10x^4 + 40x^3 + 80x^2 + 80x + 31$

Now all the coefficients $a_i \neq 0$ for $i = 0, \ldots, n$ and we can now find the roots of the new Polynomial to be:

$X_1 = -0.9999999999999998$
$X_2 = (-2.8090169943749466 + i0.5877852522924708)$
$X_3 = (-2.8090169943749466 - i0.5877852522924708)$
$X_4 = (-1.6909830056250537 - i0.951056516295154)$
$X_5 = (-1.6909830056250534 + i0.9510565162951539)$

Adding the shifting value back (+2) you get:

$X_1 = +0.9999999999999998$
$X_2 = (-0.8090169943749466 + i0.5877852522924708)$
$X_3 = (-0.8090169943749466 - i0.5877852522924708)$
$X_4 = (0.30901699437494745 + i0.9510565162951536)$
$X_5 = (0.30901699437494745 - i0.9510565162951535)$

Which is the roots to the Polynomial $x^5 - 1$.

J Gathen [21] is a good reference for fast Taylor shifts algorithms.

Here is the algorithm where $z_0$ is the shift value

Given $P(z) = a_nz^n + a_{n-1}z^{n-1}, \ldots, a_1x + a_0$ We try to find Polynomial $Q(z) = q_nz^n + q_{n-1}z^{n-1}, \ldots, q_1x + q_0$ That represent the $z_0$ shifted Polynomial.

Arrange $P(z)$ in a matrix form, where $z_0$ is the shift value:

$$M = \begin{bmatrix}
    a_{n-1}z_0^{n-1} & a_nz_0^n \\
    a_{n-2}z_0^{n-2} & a_nz_0^n \\
    \vdots & \ddots \\
    a_2z_0^2 & a_nz_0^n \\
    a_0z_0^0 & a_nz_0^n
\end{bmatrix}$$

Compute: $M[i,j+1] = M[i-1,j] + M[i-1,j+1]$ for $j = 0, 1, \ldots, n-1$; $i = j+1, \ldots n$

Then $q_i = \frac{M[n,i+1]}{z_0^i}$ for $i = 0, 1, \ldots, n-1$; and $q_n = a_n$
Algorithm for Polynomial Taylor shift with real coefficients

/*
* Given the n - degree polynomial : p(x) = anxn + an - 1xn - 1 + ... + a1x + a0
* We must obtain new polynomial coefficients qi, by Taylor shift q(x) = p(x + x0).
* We’ll use the matrix t of dimensions m x m, m=n+1 to store data.
* Compute ti, 0 = an - i - 1x0^(n - i - 1) for i = 0..n - 1
* Store ti, i + 1 = anx0^n for i = 0..n - 1
* Compute ti, j + 1 = ti - 1, j + ti - 1, j + 1 for j = 0..n - 1, i = j + 1..n
* Compute the coefficients : qi = tn, i + 1 / x0^i for i = 0..n - 1
* The highest degree coefficient is the same : qn = an
*/
void taylorShift(const int n, double a[], double shift)
{
    int i, j, m = n + 1;
    double **t;
    if (shift == 0) return; // No shift, no change
    t = new double *[m];
    for (i = 0; i < m; ++i)
        t[i] = new double[m];
    for (i = 0; i < n; ++i)
    {
        t[i][0] = a[i + 1] * pow(shift, n - i - 1);
        t[i][i + 1] = a[0] * pow(shift, n);
    }
    for (j = 0; j < n; ++j)
        for (i = j+1; i <= n; ++i)
            t[i][j + 1] = t[i-1][j] + t[i-1][j+1];
    for (i = 0; i < n; ++i)
        a[n-i] = t[n][i + 1] / pow(shift, i);
    for (i = 0; i < m; ++i)
        delete t[i];
    delete [] t;
}

Algorithm for Polynomial Taylor shift with complex coefficients

/*
* Given the n - degree polynomial : p(x) = anxn + an - 1xn - 1 + ... + a1x + a0
* We must obtain new polynomial coefficients qi, by Taylor shift q(x) = p(x + x0).
* We’ll use the matrix t of dimensions m x m, m=n+1 to store data.
* Compute ti, 0 = an - i - 1x0^(n - i - 1) for i = 0..n - 1
* Store ti, i + 1 = anx0^n for i = 0..n - 1
* Compute ti, j + 1 = ti - 1, j + ti - 1, j + 1 for j = 0..n - 1, i = j + 1..n
* Compute the coefficients : qi = tn, i + 1 / x0^i for i = 0..n - 1
* The highest degree coefficient is the same : qn = an
*/
void taylorShift(const int n, complex<double> a[], double shift)
{
    int i, j, m = n + 1;
    complex<double> **t;
    if (shift == 0) return; // No shift, no change
    t = new complex<double> *[m];
    for (i = 0; i < m; ++i)
        t[i] = new complex<double> [m];
    for (i = 0; i < n; ++i)
    {
        t[i][0] = a[i + 1] * pow(shift, n - i - 1);
        t[i][i + 1] = a[0] * pow(shift, n);
    }
for (j = 0; j < n; ++j)
    for (i = j + 1; i <= n; ++i)
        t[i][j + 1] = t[i - 1][j] + t[i - 1][j + 1];

for (i = 0; i < n; ++i)
    a[n - i] = t[n][i + 1] / pow(shift, i);

for (i = 0; i < m; ++i)
    delete t[i];
delete[] t;
Finding the roots of the Polynomial

By now, we have nearly all we need to present a full solution to our root finder algorithm. However, we still need a few helpful piece of code.

**Simple roots**

Eliminate the simple zeros, which is zeros equal to x=0

It is well now that prior to using any iteration methods you can eliminate simple zeros or zeros of polynomial for two or less degree. Simple zeros is where roots is x=0. Moreover, it is always when the last coefficients of the polynomial is zero. E.g.

\[ 3x^3 + 2x^2 + x = 0 \]

Since the last coefficients (the constant term is zero) we have immediately found a root x=0, dividing it up in the original example yields a quadratic Polynomial \[ 3x^2 + 2x + 1 = 0 \] where we can applied the quadratic formula and find the remaining two roots directly.

There exist direct solutions for Cubic and Quadratics Polynomials however, I have never seen them implemented in a general Polynomial root finder.

**Algorithm for eliminating zero roots for Polynomial with real coefficients**

```c
// For Polynomial with complex coefficients a[],
// Eliminate all zero roots from the polynomial
// N is the degree of the Polynomial
// The complex solutions is stored in res[i]
// The new degree is return and the remaining coefficients is in a[]
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
int zeroroots(const int n, const double a[], complex<double> res[]) {
 int i;
 for (i = n; a[i] == 0.0; --i) {
 res[i] = complex<double>(0.0);
 }
 return i;
}
```

**Algorithm for eliminating zero roots for Polynomial with complex coefficients**

```c
// For Polynomial with complex coefficients a[],
// Eliminate all zero roots from the polynomial
// N is the degree of the Polynomial
// The complex solutions is stored in res[i]
```
The 1st order and quadratic solutions is pretty straightforward and is presented without any explanation other

**Algorithm for Quadratic roots for Polynomial with real coefficients**

```cpp
// Solve linear or quadratic equation
// For Polynomial with real coefficients a[],
// The real or complex solutions is stored in res[1] and res[2]
// Notice that a[0] is a2, a[1] is a1 and a[2]=a0
//
// void quadratic(const int n, const double a[], complex<double> res[] )
// {
//     double r;
//     if (n == 2)
//     {
//         if (a[1] == 0)
//         {
//             r = -a[2] / a[0];
//             if (r < 0)
//             {
//                 res[1] = complex<double>(0, sqrt(-r));
//                 res[2] = complex<double>(0, -res[1].imag());
//             }
//             else
//             {
//                 res[1] = complex<double>(sqrt(r), 0);
//                 res[2] = complex<double>(-res[1].imag(), 0);
//             }
//         }
//         else
//         {
//             r = 1 - 4 * a[0] * a[2] / (a[1] * a[1]);
//             if (r < 0)
//             {
//                 res[1] = complex<double>(-a[1] / (2 * a[0]), a[1] * sqrt(-r) / (2 * a[0]));
//             }
//             else
//             {
//                 res[1] = complex<double>((-1 - sqrt(r)) * a[1] / (2 * a[0]), 0);
//             }
//         }
//     }
```
Algorithm for Quadratic roots for Polynomial with complex coefficients

```
// For Polynomial with complex coefficients a[],
// The real or complex solutions is stored in res[1] and res[2]
// Notice that a[0] is a2, a[1] is a1 and a[2]=a0
// void quadratic(const int n, const complex<double> a[], complex<double> res[])
{
    complex<double> v;
    if (n == 1)
    {
        res[1] = -a[1] / a[0];
    }
    else
    {
        if (a[1] == complex<double>(0))
        {
            res[2] = -res[1];
        }
        else
        {
            v = sqrt(complex<double>(1) - complex<double>(4) * a[0] * a[2] / (a[1] * a[1]));
            if (v.real() < 0)
            {
                res[1] = (complex<double>(-1) - v) * a[1] / (complex<double>(2) * a[0]);
            }
            else
            {
                res[1] = (complex<double>(-1) + v) * a[1] / (complex<double>(2) * a[0]);
            }
        }
    }
}```

**Determine the multiplicity of a root**

Lastly, we need a way to determine the multiplicity of a root. This is not needed for all the methods however, it can be quite useful to know in advance what multiplicity for a root we are dealing with. The drawback is that we need to be somehow close to the root in order to estimate the multiplicity with some accuracy.

There exist several methods to determine the multiplicity. I will just mention a few all investigated by J. MCNamee [7].
Lagouanelle (1966) gives a method of estimating the multiplicity, $m$ of a root $\partial_j$, namely

$$m_j = \lim_{z \to \partial_j} \left\{ \frac{p'(z_n)^2}{p'(z_n)^2 - p(z_n)p''(z_n)} \right\}$$

Drawback is of course that you also need access to the second derivative of $P$.

Traub (1964) uses:

$$m = \frac{\ln(P(z_n))}{\ln(P'(z_n))}$$

Rounded to the nearest integer.

Madsen (1973) (as implemented in root finder for Newton) forms $z_i + pdz_i$, for $p=1,2,\ldots,n$ where $dz_i = -\frac{p(z_i)}{p'(z_i)}$ and choose the $p$ where $|p(z_i + pdz_i)|$ is the minimum therefore we don’t need to explicit evaluate $m$ prior.
Newton and higher order derivative based methods

Newton method

The newton method is most likely the most used root finder algorithm out there. It is really simple to implement but in its naked form it does not always convergence, particular if you start a long way away for the root or do not use special heuristics to make it converge. The Newton iterations algorithm looks like this.

\[
z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n)}
\]

Graphically the next iteration step can be visualized by the interception of the tangent and the x-axis as pictured below.

In order to compare this method with other we use an efficiency index to see how it stack up against other derivative based methods.

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

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>2</td>
<td>( \frac{1}{2^2} = 1.42 )</td>
</tr>
</tbody>
</table>

However above formula suffer in convergence speed when dealing with multiplicity of root >1 there we use the modified version that maintain convergence order even for multiple roots. See [11]
\[ z_{n+1} = z_n - m \frac{P(z_n)}{P'(z_n)} \]

In general a typical template code layout for a Newton method is.

```cpp
// Pseudo code for a newton iteration
// n = Polynomial degree
// a[] = real Polynomial coefficients
// Notice that a[0] is an, a[1] is an-1 and a[n] = a0
// res[] = found root
void Newton(int n, double a[], complex<double> res[])
{
    // Global initialization
    while(n>2)
    {
        // Per root initialization
        dz = startpoint(n, a);
        fz = horner(n, a, z); // fz = P(z)
        EPS = ... // Termination value of |P(z)|
        // Loop until z does not change or |fz| < EPS
        while(z + dz != z || abs(fz) < EPS)
        {
            // Do Newton, Halley, Ostrowski or Householder step
        }
        // Root found
        Res[n] = z; // Save found root
        n = deflation(n, a, z); // Deflate Polynomial with found root
    }
    Quadratic(n, a, res);
}
```

We will use this template layout for the other like based methods. E.g., Halley and Householders 3rd order.

Of course the most interesting part is the section “Do Newton steps” Madsen [5] provide a very fast and efficient implementation that not only find the roots in surprisingly few iterations but also handle the usual issues like. I do not plan to repeat what is so excellent describe in [5] but just highly some interesting area of his Newton implementation.

1) The first step is to find the \(dz_n = \frac{P(z_n)}{P'(z_n)}\) and of course decide what should happen if \(P'(z_n) = 0\). Madsen conclude that when this condition arise it is 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 with rotating the direction with an odd degree of 53 degree and multiply the direction with the factor \(m\). Madsen found that a suitable value for \(m = 5\) was reasonable when this happen

2) Furthermore, Madsen also realized that when if \(P'(z_n) \sim 0\) you could get some unreasonable step size of \(dz_n\) and therefore he introduce a scaling factor that
reduced the current step size if $abs(dz_n) > 5 \times abs(dz_{n-1})$ than the previous iterations step size. Again he alter the direction with $dz_n = dz_n(0.6 + i0.8) \times \frac{5 \times abs(dz_{n-1})}{abs(dz_n)}$

3) These two modification makes his method very resilience and make it always converge to a root.

4) The next improvement was to use 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 $z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ he then look to see if $P(z_{n+1}) > P(z_n)$:

a. Madsen 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 previous $z_{n+1}$ 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 new revised $z_{n+1} = z_n - 0.25dz_n$. If $P(z_{n+1}) \geq P(z_n)$ 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 alter 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.

b. Then we are looking in the right direction and we then continue stepping in that direction using $z_{n+1} = z_n - mdz_n$ $m = 2, \ldots, n$ 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 are a root with 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.

5) The process 1-5 continue until the stopping criteria is reach where after the root $z_0$ is accepted and deflated up in the Polynomial and a new search for a root using the deflated Polynomial is initiated.

Madsen also divide the iterations up in two stages. Stage 1 & Stage 2. In stage 1 we are trying to get into the circle where we are sure that the Newton method will converge towards a root. When we are getting into that circle, we automatically switch to stage 2. In stage 2 we skip step 4 & 5 and just use a pure Newton step $z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ until the stopping criteria has been satisfied. In case we get outside the convergence circle, we switch back to stage 1 and continue the iteration.

Madsen use the following criteria to switch to stage 2 based on the theorem 7.1 from Ostrowski [12] that states if K is a circle with center $w - \frac{P(w)}{P'(w)}$ and radius $\frac{|P(w)|}{|P'(w)|}$.

Then we have if the following two conditions is satisfied:

\[
p(w)p'(w) \neq 0 \quad \text{and} \quad 2\left|\frac{p(w)}{p'(w)}\right| \cdot \max_{z \in K}|p''(z)| \leq |p'(w)|
\]

That the Newton iterations with initial value w, will lead to a convergence of zero in the within the circle K. To simplify the calculation with make 2 substitutes, since
max|p''(z)| ≈ |p''(w)| and instead of p''(w) we replace it with a difference approximation

\[ p''(w) \approx \frac{p'(z_{k-1}) - p(w)}{z_{k-1} - w} \]

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

There is a few more tricks to this that the one describe above which has been removed from the code example below, but that is not important for the overall process.

Since both the Newton version for Polynomial with real coefficients and the version, using Complex coefficients is very similar in nature with the exception that the real coefficients version is using real arithmetic instead of complex arithmetic speeding up the iterative search I will only show the Complex coefficients version since it is easier to digest.

This algorithm below has been modified on a few places and ported from the original code in AlgoW to below C++ implementation. E.g. we use a better upperbound (xxx) for the Horner evaluation of the polynomial P(z) than was implemented in the original code.

**Algorithm for the Newton method for Complex coefficients Polynomial**

```cpp
// Find all root of a polynomial of n degree with complex coefficients using the modified Newton
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// The roots is stored in res[1..n] where res[n] is the first root found and res[1] the last root.
//
void Newton(int n, const complex<double> coeff[], complex<double> res[]) {
    int stage1, i;
    double r, r0, u, f, f0, eps, f1, ff;
    complex<double> z0, f0z, z, dz, f1z, fz;
    complex<double> *a1, *a;

    a = new complex<double>[(n + 1)]; // Copy the original coefficients
    for (i = 0; i <= n; i++) a[i] = coeff[i];
    // Eliminate zero roots
    n = zeroroots(n, a, res);
    // Create a1 to hold the derivative of the Polynomial a for each iterations
    a1 = new complex<double>[(n + 1)];
    while (n > 2) // Iterate for each root
        {
            // Calculate coefficients of f'(x)
            for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i, 0);

            u = startpoint(n, a); // Calculate a suitable start point
            z0 = 0; ff = f0 = abs(a[n]); f0z = a[n - 1];
            if (a[n - 1] == complex<double>(0))
                z = 1;
            else
                z = -a[n] / a[n - 1];
            dz = z = z / abs(z) * complex<double>(u);
            f1 = horner(n, a, z); f = abs(f1); r0 = 5 * u;
            // Initial use a simple upperbound for EPS until we get closer to the root
            r = f0;
```
eps = 6 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);

// Start the iteration
while (z + dz != z && f > eps)
{
    f1z = horner(n - 1, a1, z); f1 = abs(f1z);
    if (f1 == 0.0)
        dz *= complex<double>(0.6, 0.8) * 5.0;
    else
    {
        double wsq;
        complex<double> wz;

        dz = fz / f1z;
        wz = (f0z - f1z) / (z0 - z);
        wsq = abs(wz);
        stage1 = (wsq / f1 > f1 / f / 2) || (f != ff);
        r = abs(dz);
        if (r > r0)
            { // Try multiple steps or shorten steps depending
                int div2;
                double fn;
                complex<double> zn, fzn;

                zn = z;
                for (i = 1, div2 = f > f0; i <= n; i++)
                    { // Shorten steps
                        if (div2 != 0)
                            { // Shorten steps
                                dz *= 0.5; zn = z0 - dz;
                            }
                        else
                            zn -= dz; // try another step in the
                            // same direction
                    }
                fzn = horner(n, a, zn); fn = abs(fzn);
                if (fn >= f)
                    break; // Break if no improvement
                f = fn; fz = fzn; z = zn;
                if (div2 != 0 && i == 2)
                    { // To many shorten steps try another
                        dz *= complex<double>(0.6, 0.8);
                        z = z0 - dz;
                        fz = horner(n, a, z); f = abs(fz);
                        break;
                    }
            }
    }
}
Practical Implementation of Polynomial Root Finders

```cpp
// calculate the upper bound of errors using Grant & Hitchins's test
eps = upperbound(n, a, z);

//
z0 = complex<double>(z.real(), 0.0);
fz = horner(n, a, z0);
if (abs(fz) <= f)
    z = z0;
res[n] = z;
n = complexdeflation(n, a, z);
```

```cpp
quadratic(n, a, res);
delete[] a1, a;
```

**Halley method**

Let turn our attention to a higher order method. One of them is Halley, which is a cubic convergence method meaning that for each iteration step we triple the number of correct digits in our root.

The Halley's method uses the iteration:

\[
z_{n+1} = z_n - \frac{2P(z_n)P'(z_n)}{2P'(z_n)^2 - P(z_n)P''(z_n)}
\]

Or sometimes written as: ([13] Peter Acklam)

\[
z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \left[ 1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2} \right]^{-1}
\]

Where \( z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \) is the usual Newton iteration enhanced with the factor:

\[
\left[ 1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2} \right]^{-1}
\]

And are graphical shows below:
The efficiency index is slightly larger than the Newton method and in order to get a convergence order of 3 we need to also calculate the $P''(z_n)$.

As for the Newton method we don’t use this version since it will show the same weakness as the original Newton step when dealing with roots with a multiplicity higher than 1.

Instead, we used the modified version from Hansen & Patrick [14] for the Halley methods:

$$z_{n+1} = z_n - \frac{P(z_n)}{m + 1} \frac{P'(z_n)}{2m} - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}$$

Alternatively, written in another way:

$$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \left[ \frac{m + 1}{2m} - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2} \right]^{-1}$$

Where $z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ is the usual Newton iteration modified with a factor:

$$\left[ \frac{m + 1}{2m} - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2} \right]^{-1}$$

Unfortunately, it does not seem to work. Taking for example.

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

Clearly there is a double root at $x=2$ so setting a start guess of 0.5 and $m=2$ you are getting the following iterations that result in a root of $x=2.13$ which is not correct.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halley</td>
<td>3</td>
<td>$\frac{1}{3^3} = 1.44$</td>
</tr>
</tbody>
</table>
Instead, I use my own modified version for the Halley methods:

\[ z_{n+1} = z_n - \frac{m + 1}{2} \frac{P(z_n)P'(z_n)}{2P'(z_n)^2 - P(z_n)P''(z_n)} \]

Alternatively, written in another way:

\[ z_{n+1} = z_n - \frac{m + 1}{2} \frac{P(z_n)}{P'(z_n)} \left[ 1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2} \right]^{-1} \]

Where \( z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \) is the usual Newton iteration modified with a factor:

\[ \frac{m + 1}{2} \left[ 1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2} \right]^{-1} \]

Doing the same iteration with the same conditions as before you get

Initial guess 0.5

1 1.6632525083612000
2 1.9725835276790300
3 1.9998099710330100
4 1.99999990793100
5 1.99999990793100

Which is the correct root ~2.0.

We use the same “template” for the code for the Halley method as for the Newton method and get below code for an efficient implementation of the Halley method.

**Algorithm for the Halley method for Complex coefficients Polynomial**

```cpp
// Find all root of a polynomial of n degree with complex coefficients
// using the modified Halley
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// The roots is stored in res[1..n] where res[n] is the first root found
// and res[1] the last root.
//
// void Halley(int n, const complex<double> coeff[], complex<double> res[] )
{
    int i;
    bool stage1;
    double u, f, f1, f2, f0, ff, eps, fw;
    complex<double> z0, z, dz, fz2, fz1, fz0, fwz, wz, fz;
    complex<double> g, h;
    complex<double> *a2, *a1, *a;
    double r, r0;

    a = new complex<double>[n + 1]; // Copy the original coefficients
    for (i = 0; i <= n; i++) a[i] = coeff[i];
```
// Eliminate zero roots
n = zeroroots(n, a, res);
// Create a1 and a2 to hold the first and second derivative of the Polynomial a for each iterations
a1 = new complex<double>[n];
a2 = new complex<double>[n-1];
while( n > 2 )
{
    // Calculate coefficients of \( f'(x) \)
    for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i, 0);
    // Calculate coefficients of \( f''(x) \)
    for (i = 0; i < n - 1; i++) a2[i] = a1[i] * complex<double>(n - i - 1, 0);

    u = startpoint(n, a); // Calculate a suitable start point
    z0 = 0; ff = f0 = abs(a[n]); fz0 = a[n - 1];
    if (a[n - 1] == complex<double>(0))
        z = 1;
    else
        z = -a[n] / a[n - 1];
    dz = z = z / abs(z) * complex<double>(u);
    fz = horner(n, a, z); f = abs(fz); r0 = 5 * u;
    // Initial use a simple upperbound for EPS until we get closer to the root
    eps = 6 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);

    // Start iteration
    while( z + dz != z & & f > eps )
    {
        fz1 = horner(n - 1, a1, z); f1 = abs(fz1);
        if (f1 == 0.0) // True saddelpoint
        {
            dz = complex<double>(0.6, 0.8) * 5.0;
            z = z0 - dz; fz = horner(n, a, z); f = abs(fz);
            continue;
        }
        else
        {
            g = fz / fz1;
            fz2 = horner(n - 2, a2, z); f2 = abs(fz2);
            h = fz2 / fz1;
            h = g * h * complex<double>(0.5);
            dz = g / (complex<double>(1) - h);
            stage1 = (f2 / f1 > f1 / f / 2) || (f != ff);
            r = abs(dz);
            if (r > r0)
            {
                dz = complex<double>(0.6, 0.8) * (r0 / r); r = abs(dz);
            }
        }
    }
    z0 = z; f0 = f; fz0 = fz; z = z0 - dz;
    fz = horner(n, a, z); ff = f = abs(fz);
    if (stage1)
    { // In stage 1
        if (f > f0) // Check shorten stepsizes
        {
            for (i = 1; i <= n; i++)
            {
                dz *= complex<double>(0.5);
wz = z0 - dz;
fw = horner(n, a, wz, &fwz);
if (fw >= f)
    break;
f = fw; fz = fwz; z = wz;
if (i == 2)
    {
        dz *= complex<double>(0.6, 0.8);
        z = z0 - dz;
        fz = horner(n, a, z); f = abs(fz);
        break;
    }
}
else
    {
        // Try multiple steps in the same direction
        optimizing multiple roots iterations
        for (int m = 2; m <= n; m++)
            {
                wz = 0.5*(m+1) * dz;
                wz = z0 - wz;
                fwz = horner(n, a, wz); fw = abs(fwz);
                if (fw >= f)
                    break;  // No improvement.
                f = fw; fz = fwz; z = wz;
            }
    }
else
    {   // In Stage 2.
        // calculate the upper bound of erros using Grant & Hitchins's test
        eps = upperbound(n, a, z);
    }

// End Iteration
z0 = complex<double>(z.real(), 0.0);
fz = horner(n, a, z0);
if (abs(fz) <= f)
    z = z0;
res[n] = z;
n = complexdeflation(n, a, z);
}
quadratic(n, a, res);
delete [] a, a1, a2;
return;

Householder 3rd order method.

Household has generalized the higher order methods in which 1st order is the Newton and 2nd order is Halley’s method. Householders 3rd order has quantic convergence rate.

The Householder’s 3rd order method uses the following iteration:
$z_{n+1} = z_n - \frac{6P(z_n)P'(z_n)^2 - 3P(z_n)^2P''(z_n)}{6P'(z_n)^3 - 6P(z_n)P'(z_n)P''(z_n) + P(z_n)^2P'''(z_n)}$  

Substituting:

$t = \frac{P(z_n)}{P'(z_n)}, u = \frac{P''(z_n)}{P'(z_n)}, v = \frac{P'''(z_n)}{P'(z_n)}$

We can now write the householder’s 3$^{rd}$ order as following:

$z_{n+1} = z_n - \frac{t(1 - 0.5tu)}{1 - t(u - \frac{vt}{6})}$

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Householder 3$^{rd}$</td>
<td>4</td>
<td>$\frac{1}{4^4} = 1.41$</td>
</tr>
</tbody>
</table>

Equivalent with the Newton reduction the Householder 3$^{rd}$ order reduction is a factor of $\frac{3}{m^2}$ by multiplier the step size with the reverse factor we should ensure quartic convergence rate.

Our modified Householder 3$^{rd}$ order will be:

$z_{n+1} = z_n - \frac{m + 2}{3} \left[ \frac{6P(z_n)P'(z_n)^2 - 3P(z_n)^2P''(z_n)}{6P'(z_n)^3 - 6P(z_n)P'(z_n)P''(z_n) + P(z_n)^2P'''(z_n)} \right]$  

Or using the same substitution as before:

$z_{n+1} = z_n - \frac{m + 2}{3} t(1 - 0.5tu) \frac{1}{1 - t(u - \frac{vt}{6})}$

We use the same “template” for the code for the Householder 3$^{rd}$ order method as for the Newton method and get below code for an efficient implementation of the Householder 3$^{rd}$ order method.

**Algorithm for the Householder method for Complex coefficients**

```
// Find all root of a polynomial of n degree with complex coefficient
// using the Halley 3rd order method
// Iterations algorithm:
// Define t=P(z)/P'(z)
// u=P''(z)/P'(z)
// v=P'''(z)/P'(z)
// xnext = xold - t * (1-0.5*t*u)/(1-t(u-1/6*v*t)
// the multiple root modifier is (m+2)/3;
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// The roots is stored in res[1..n] where res[n] is the first root found
```
// and res[1] the last root.
//
void Householder3(int n, const complex<double> coeff[], complex<double> res[])
{
    int i;
    bool stage1;
    complex<double> *a, *a1, *a2, *a3;
    double s, r, r0, eps;
    double f, f0, f1, f2, f3, fw, ff;
    complex<double> z, z0, dz, fz, fwz, wz, fz0, fz1, fz2, fz3;
    complex<double> t, u, v, g, h;

    a = new complex<double>[(n + 1)]; // Copy the original coefficients
    for (i = 0; i <= n; i++) a[i] = coeff[i];

    // Eliminate zero roots
    n = zeroroots(n, a, res);

    // Create a1 and a2 to hold the first and second derivative of the
    // Polynomial a for each iterations
    a1 = new complex<double>[n];
    a2 = new complex<double>[n - 1];
    a3 = new complex<double>[n - 2];

    while (n > 2)
    {
        // Calculate coefficients of f'(x)
        for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i, 0);
        // Calculate coefficients of f''(x)
        for (i = 0; i < n - 1; i++) a2[i] = a1[i] * complex<double>(n - i - 1, 0);
        // Calculate coefficients of f'''(x)
        for (i = 0; i < n - 2; i++) a3[i] = a2[i] * complex<double>(n - i - 2, 0);

        // Set z0
        z0 = complex<double>(0); f0 = abs(a[n]); fz0 = complex<double>(a[n - 1]);
        // Calculate z
        s = startpoint(n, a);
        if (a[n - 1] == complex<double>(0))
        {
            z = complex<double>(1);
        } else
        {
            z = -a[n] / a[n - 1];
            dz = z = z / abs(z) * complex<double>(s);
            fz = horner(n, a, z); ff = f = abs(fz);
            // Calculate safety zone as 5 times start guess s
            r0 = 5.0 * s;
            // calculate the preliminary upper bound of errors
            eps = 6 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);
            // Start iteration
            while (z + dz != z && f > eps)
            {
                /* Iterativ loop */
                fz1 = horner(n - 1, a1, z); f1 = abs(fz1);
                if (f1 == 0.0) /* True saddlepoint */
                {
                    dz *= complex<double>(0.6, 0.8) * 5.0;
                    z = z0 - dz; fz = horner(n, a, z); f = abs(fz);
                    continue;
                }
                else
                {
                    t = fz / fz1;
                    fz2 = horner(n - 2, a2, z); f2 = abs(fz2);
                    u = fz2 / fz1;
                    }
fz3 = horner(n - 3, a3, z); f3 = abs(fz3);

v = fz3 / fz1;

g = complex<double>(1.0) - complex<double>(0.5) * u

complex<double>(1.0 / 6.0));

dz = t * (g / h);

stage1 = (f2 / f1 > f1 / f / 2) || (f != ff);

r = abs(dz);

if (r > r0)
{
    dz *= complex<double>(0.6, 0.8) * (r0 / r);
    r = abs(dz);
}

r0 = r * 5.0;

z0 = z; f0 = f; fz0 = fz; z = z0 - dz;

fz = horner(n, a, z); ff = f = abs(fz);

if (stage1)
{
    // In stage 1
    if (f > f0) // Check shorten stepsizes
    {
        for (i = 1; i <= n; i++)
        {
            dz *= complex<double>(0.5);
            wz = z0 - dz;
            fwz = horner(n, a, wz); fw = abs(fwz);
            if (fw >= f)
            {
                f = fw; fz = fwz; z = wz;
                if (i == 2)
                {
                    dz *= complex<double>(0.6, 0.8);
                    z = z0 - dz;
                    f = horner(n, a, z, &fz);
                    break;
                }
            }
        }
    }
    else
    {
        // Try multiple steps in the same direction
        
        optimizing multiple roots iterations
        for (int m = 2; m <= n; m++)
        {
            wz = complex<double>((m + 2) / 3.0) * abs(fwz);
            fwz = horner(n, a, wz); fw = abs(fwz);
            if (fw >= f)
            {
                f = fw; fz = fwz; z = wz;
                break; // No improvement.
            }
        }
    }
}
else
{
    // In Stage 2.
    // calculate the upper bound of errors using
    Grant & Hitchins's test
    eps = upperbound(n, a, z);
}
Ostrowski’s multi-point method

The Ostrowski’s multi-point method for root finding is a two-step method (multi-point). First step is a regular Newton step and the second step is a correction that only requires one extra Horner evaluations. Thereby the method has a very high efficiency index of 1.59 and is a fourth order method. The Ostrowski method has generated a number of new Ostrowski like Methods that further extend the multi-step iteration idea to generate sixth, seventh and even eighth order convergence. Ostrowski has also given name to another method called Ostrowski square root method, which is not the same as the Ostrowski’s multi-point method.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ostrowski Multi-point</td>
<td>4</td>
<td>$\frac{1}{1.59} = 1.59$</td>
</tr>
</tbody>
</table>

$$y_n = z_n - \frac{p(z_n)}{p'(z_n)}$$

$$z_{n+1} = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n)p'(z_n)}$$

However above formula has only linear convergence if multiplicity $> 1$. You could add the modified Newton method to handle multiplicity $>1$; see below.

Stage 1

$$y_n = z_n - m \frac{p(z_n)}{p'(z_n)}, m \text{ is the multiplicity}$$

Stage 2

$$z_{n+1} = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n)p'(z_n)}$$

However, then the second refinement does not work well. My approach to this is therefore

a) When an iterative step $z_n$ is not near a root and we see improvement using the multi-step and or shortening of the step size (see the description of the Newton method) then stick with this modified Newton approach.
b) First when you do not see any improvement using the multi-step check and or shortening of step, then do the second refinement and obtain a 4th order convergence for the remaining iterations. Well what about multiplicity > 1. That is not a problem since it will keep the Newton method at stage 1 and convert quadratic to that root and in that, special case the Ostrowski multi-point method will not be a 4th order method, for simply root it will however be a 4th order method.

Algorithm for the Ostrowski’s multi-point method for Complex coefficients Polynomial

```cpp
// Find all root of a polynomial of n degree with complex coefficients
// using the modified Ostrowski
// Notice that a[0] is an, a[1] is an-1 and a[n]=0
// The roots is stored in res[1..n] where res[0] is the first root found and
// res[n] the last root.

void OstrowskiMP(int n, const complex<double> coeff[], complex<double> res[])
{
    int i; bool stage1;
    double r, r0, u, f, f0, eps, f1, ff;
    complex<double> z0, f0z, z, dz, f1z, fz, fz0;
    complex<double> *a1, *a;

    a = new complex<double>[n + 1]; // Copy the original coefficients
    while (n > 2) // Iterate for each root
    {
        // Calculate coefficients of f'(x)
        for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>((n - i, 0));

        u = startpoint(n, a); // Calculate a suitable start point
        z0 = 0; ff = f0 = abs(a[n]); f0z = a[n - 1];
        if (a[n - 1] == complex<double>(0))
            z = 1;
        else
        {
            double wsq;
            complex<double> wz;
            z = a[n] / a[n - 1];
            dz = z = z / abs(z) * complex<double>(u);
            f0z = horner(n, a, z0); f = abs(fz); r0 = 5 * u;
            // Initial use a simple upperbound for EPS until we get closer to the
            eps = 6 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);

            // Start the iteration
            while (z + dz != z && f > eps)
            {
                f1z = horner(n - 1, a1, z); f1 = abs(f1z);
                if (f1 == 0.0) // True Saddlepoint
                    dz *= complex<double>(0.6, 0.8) * 5.0;
                else
                {
                    double wsq;
                    complex<double> wz;
                    dz = fz / f1z;
                    wz = (f0z - f1z) / (z0 - z);
                    wsq = abs(wz);
                    stage1 = (wsq / f1 > f1 / f / 2) || (f != ff);
                    r = abs(dz);
                    if (r > r0)
                        
```
dz *= complex<double>(0.6, 0.8) * (r0 / r); r = abs(dz);
}

r0 = 5 * r;

z0 = z; f0 = f; f0z = f1z; fz0 = fz;
if (stage1)
{// Try multiple steps or shorten steps depending of f is an improvement or not
        int div2;
        double fn;
        complex<double> zn, fzn;
        zn = z;
        for (i = 1, div2 = f > f0; i <= n; i++)
        {
                if (div2 != 0)
                {
                        // Shorten steps
                        dz *= 0.5; zn = z0 - dz;
                }
                else
                        zn -= dz; // try another step in the same direction

                fzn = horner(n, a, zn); fn = abs(fzn);
                if (fn >= f)
                        break; // Break if no improvement

                f = fn; fz = fzn; z = zn;
                if (div2 != 0 && i == 2)
                        {// To many shorten steps try another direction
                        dz *= complex<double>(0.6, 0.8);
                        z = z0 - dz;
                        fz = horner(n, a, z); f = abs(fz);
                        break;
                        }
        }
        else
        {// calculate the upper bound of erros using Grant & Hitchins's test
                eps = upperbound(n, a, z);
        }

        if (f==ff) // No stage 1 improvement
        {// Do the Ostrowski step as second part of the multi-point iteration
                z = z - fz0 / (fz0 - complex<double>(2) * fz) * fz / f1z;
                fz = horner(n, a, z); ff = f = abs(fz);
        }

        z0 = complex<double>(z.real(), 0.0);
        fz = horner(n, a, z0);
        if (abs(fz) <= f)
                z = z0;
        res[n] = z;
        n = complexdeflation(n, a, z);
}

quadratic(n, a, res);
delete[] a1, a;
The Ostrowski multi-point iteration has given rise to a number of Ostrowski like iteration, capitalizing on the same idea, see [19] E.g. the 6th order convergence:

Stage 1  \( y_n = z_n - \frac{p(z_n)}{p'(z_n)} \)

Stage 2  \( v_n = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n) p'(z_n)} \)

Stage 3  \( z_{n+1} = v_n - \frac{p(z_n) p(v_n)}{p(z_n) - 2p(y_n) p'(z_n)} \)

Or

Stage 1  \( y_n = z_n - \frac{p(z_n)}{p'(z_n)} \)

Stage 2  \( v_n = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n) p'(z_n)} \)

Stage 3  \( z_{n+1} = v_n - \frac{p(z_n) p(v_n)}{p(z_n) + (\beta + 2)p(y_n) p'(z_n)} \)

With variation on \( \beta \) that gives and accelerated 6th order convergence. When \( \beta=-2 \) you have the previous 6th order convergence method.

**How the method and higher orders method stack up against each other**

To see how it works with the different methods lets each method against a simple Polynomial.

\[
P(x) = (x - 2)(x + 2)(x - 3)(x + 3) = x^4 - 13x^2 + 36
\]

The above mention Polynomial is an easy one for most methods. Moreover, as you can see the higher order method do requires fewer numbers of iterations. However, also more work to be done per iterations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Newton</th>
<th>Halley</th>
<th>Household 3rd</th>
<th>Ostrowski</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>Start guess</td>
<td>0.8320502943278436</td>
<td>0.8320502943278436</td>
<td>0.8320502943278436</td>
<td>0.8320502943278436</td>
</tr>
<tr>
<td>1</td>
<td>2.2536991416170737</td>
<td>1.6933271400922734</td>
<td>2.03343599287734</td>
<td>2.0863365344560694</td>
</tr>
<tr>
<td>2</td>
<td>1.9233571772166798</td>
<td>1.9899385955094577</td>
<td>1.999999577501767</td>
<td>1.999968127551831</td>
</tr>
<tr>
<td>3</td>
<td>1.997330690698116</td>
<td>1.9999993042509177</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1.999996107736492</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.9999999999916678</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Laguerre’s method

Another interesting method see McNamee [7] or [15]. Laguerre’s method requires accept to both the 1st and 2nd derivative of P(z), but has third order convergence. Laguerre’s method was as the name implied invented by Laguerre back in 1898.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laguerre</td>
<td>3</td>
<td>$\frac{1}{3^3} = 1.44$</td>
</tr>
</tbody>
</table>

$$z_{n+1} = z_n - a$$

where

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

sign $\pm$ is chosen to maximize the denominator

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

However above formula has only linear convergence if multiplicity > 1. You could add the modified Laguerre method to handle multiplicity >1; see below where $m$ is the multiplicity.

$$z_{n+1} = z_n - a$$

where

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

sign $\pm$ is chosen to maximize the denominator

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

Most often you do not know $m$ prior but you can use the technic by Madsen (see the detailed description of the Newton method) where we continue using below formula for $m=2$ up to $n$ as long as for each $m$ the $P(z_{n+1}^m) < P(z_{n+1}^{m-1})$

The modified Laguerre work extremely well and is a very stable algorithm for finding Polynomial zeros.

Algorithm for the Laguerre’s method for Complex coefficients

Polynomial

```c
// Find all root of a polynomial of n degree with complex coefficient using the modified Laguerre
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// The roots is stored in res[1..n] where res[n] is the first root found and res[1] the last root.
//
void Laguerre(int n, const complex<double> *coeff[], complex<double> *res[])
{
    int i;
    double u, f, fprev, f1, f2, f0, eps;
```
Practical Implementation of Polynomial Root Finders

```cpp
complex<double> z0, z, dz, f2z, f1z, fz;
complex<double> *a2, *a1, *a;
double r, r0;

a = new complex<double>[(n + 1)]; // Copy the original coefficients
for (i = 0; i <= n; i++)
a[i] = coeff[i];
// Eliminate zero roots
n = zeroroots(n, a, res);
// Create a1 and a2 to hold the first and second derivative of the
Polynomial a for each iterations
a1 = new complex<double>[(n - 1)];
a2 = new complex<double>[(n - 1)];
while (n > 2)
{
    // Calculate coefficients of f'(x)
    for (i = 0; i <= n - 1; i++) a1[i] = a[i] * complex<double>(n - i, 0);
    // Calculate coefficients of f''(x)
    for (i = 0; i <= n - 2; i++) a2[i] = a1[i] * complex<double>(n - i - 1, 0);
    u = startpoint(n, a); // Calculate a suitable start point
    z0 = 0; f0 = abs(a[n]);
    if (a[n - 1] == complex<double>(0))
        z = 1;
    else
        z = -a[n] / a[n - 1];
    dz = z = z / abs(z) * complex<double>(u);
    f2z = horner(n, a, z); f = abs(f2z); r0 = 5 * u;
    eps = 6 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);
    // Start iteration
    while (z + dz != z && f > eps)
    {
        complex<double> g, h, gp, gm, w;
        f1z = horner(n - 1, a1, z); f1 = abs(f1z);
        f2z = horner(n - 2, a2, z); f2 = abs(f2z);
        g = f1z / fz;
        h = g * g - f2z / fz;
        w = sqrt(complex<double>((n - 1) * (complex<double>(n) *
            h - g * g)));
        gp = g + w;
        gm = g - w;
        // Find the maximum value
        if (norm(gp) < norm(gm))
            gp = gm;
        // Calculate dz, change directions if zero
        if (abs(gp) == 0.0)
            dz *= complex<double>(0.6, 0.8) * 5.0;
        else
            dz = complex<double>(n) / gp;
        r = abs(dz); // Check for oversized steps
        if (r > r0)
        {
            dz = complex<double>(0.6, 0.8) * (r0 / r); r = abs(dz);
        }
        r0 = 5 * r;
        z0 = z; z = z0 - dz; fprev = f;
        fz = horner(n, a, z); f = abs(fz);
        if (f > fprev)
        
        // Start iteration
        while (z + dz != z && f > eps)
        {
            complex<double> g, h, gp, gm, w;
            f1z = horner(n - 1, a1, z); f1 = abs(f1z);
            f2z = horner(n - 2, a2, z); f2 = abs(f2z);
            g = f1z / fz;
            h = g * g - f2z / fz;
            w = sqrt(complex<double>((n - 1) * (complex<double>(n) *
                h - g * g)));
            gp = g + w;
            gm = g - w;
            // Find the maximum value
            if (norm(gp) < norm(gm))
                gp = gm;
            // Calculate dz, change directions if zero
            if (abs(gp) == 0.0)
                dz *= complex<double>(0.6, 0.8) * 5.0;
            else
                dz = complex<double>(n) / gp;
            r = abs(dz); // Check for oversized steps
            if (r > r0)
            {
                dz = complex<double>(0.6, 0.8) * (r0 / r); r = abs(dz);
            }
            r0 = 5 * r;
            z0 = z; z = z0 - dz; fprev = f;
            fz = horner(n, a, z); f = abs(fz);
            if (f > fprev)
        
```
{ for (i = 1; f > fprev && i <= n; i++)
    {
        // No improvement. try shorten the steps
        dz *= 0.5; z = z0 - dz;
        fz = horner(n, a, z); f = abs(fz);
        if (f > fprev && i == 3)
            {
                // If shorten does not help rotate
                and try another directions
                dz *= complex<double>(0.6, 0.8) * 5.0;
                z = z0 - dz;
                fz = horner(n, a, z); f = abs(fz);
                break;
            }
    }
}

else
    {
        // Try stepping in that directions (usually
        multiple roots)
        for (int m = 2; m <= n; m++)
            {
                complex<double> wdz, wz, fwz; double fw;
                w = sqrt(complex<double>((((double)n /
                (double)m - 1)) * (complex<double>(n) * h - g * g)));
                gp = g + w; gm = g - w;
                // Find the maximum value
                if (norm(gp) < norm(gm))
                    gp = gm;
                wdz = complex<double>(n) / gp;
                wz = z0 - wdz;
                fwz = horner(n, a, wz); fw = abs(fwz);
                if (fw >= f)
                    break;
                f = fw; z = wz; fz = fwz;
            }
    }

z0 = complex<double>(z.real(), 0.0);
fz = horner(n, a, z0);
if (abs(fz) <= f)
    z = z0;
res[n] = z;
n = complexdeflation(n, a, z);
}
quadratic(n, a, res );
delete[] a, a1, a2;
return;
Matrix methods

Eigenvalue Method

I guess the most famous of the matrix method is the Eigenvalue method, see McNamee [7] or [16]. Using the algorithm to find the eigenvalue can also be used for finding roots of a polynomial. For any polynomial, you can create the corresponding companion matrix and then find the eigenvalues for that matrix. The eigenvalues will then be the roots of the polynomial. One of the most efficient way of doing this is to form the companion matrix using an upper Hessenberg triangular matrix. An upper Hessenberg Matrix is a square matrix for which all the sub diagonal entries are zero and all the eigenvalues when this matrix is solved will be in the diagonal elements. However, in order to find the eigenvalue you will need to resort to some form of iterative algorithm. The QR algorithm is very well suited to finding the eigenvalues of an upper Hessenberg matrix. It requires $O(n^3)$ operations. The QR algorithm was developed in the late 1950’s. The basic idea is perform a QR decomposition, writing the matrix as a product of an orthogonal matrix and an upper triangular matrix, (Factor A=QR) multiply the factors in reverse order RQ and then iterate, see [17] & [18].

Given a polynomial of

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

A companion matrix can be written for the characteristic polynomial as :

$$\begin{bmatrix}
-a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\
 a_n & a_{n-1} & \cdots & a_2 & a_1 \\
 1 & 0 & 0 & \cdots & 0 \\
 0 & 1 & \cdots & 0 & 0 \\
 0 & 0 & 1 & \cdots & 0 \\
 0 & 0 & 0 & \cdots & 1
\end{bmatrix}$$

Or easier if the polynomial is already in ammonic form as:

$$\begin{bmatrix}
-a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\
 1 & 0 & 0 & \cdots & 0 \\
 0 & 1 & \cdots & 0 & 0 \\
 0 & 0 & 1 & \cdots & 0 \\
 0 & 0 & 0 & \cdots & 1
\end{bmatrix}$$

Algorithm for the Eigenvalue method for Complex coefficients

Polynomial

// Find all root of a polynomial of n degree with complex coefficient using the eigenvalue method
// The procedure complexeigenvalue computes the eigenvalues
// of arbitrary n by n complex matrix.
// This is a cpp version of an Java version that came from an Ada version of a NAG Fortran
// library subroutine TOMS 535.
// Some Fortran labels have been preserved for traceability
// A is the nxn Matrix, Lambda[] is the eigenvalues or the roots

void Complexeigenvalue(int n, complex<double> **A, complex<double> lambda[], bool test);

// Take the absolute value of the sum of the real and imag part of a complex number
double sumabs(complex<double> Z)
{
    return fabs(Z.real()) + fabs(Z.imag());
} // end sumabs

void Eigenvalue(int n, const complex<double> coeff[], complex<double> res[])
{
    int i, j;
    complex<double> **hess;

    // Create the upper Hessenberg form of the companion matrix
    hess = new complex<double> *[n];
    for (i = 0; i < n; i++)
        hess[i] = new complex<double>[n];
    for (i = 0; i < n; i++)
    {
        hess[0][i] = -coeff[i + 1] / coeff[0];
        for (j = 1; j < n; j++) hess[j][i] = 0;
        if (i != n - 1) hess[i + 1][i] = 1;
    }

    // Set predefined result. Change in the iteration
    for (int i = 0; i < n; i++)
        res[i + 1] = complex<double>(-999.0, -999.0);
    print_complex_matrix(n, hess);
    Complexeigenvalue(n, hess, &res[1], true);

    // Cleanup
    for (i = 0; i < n; i++)
        delete hess[i];
    delete[] hess;

    return;
}

// The procedure complexeigenvalue computes the eigenvalues
// of arbitrary n by n complex matrix.
// This is a cpp version of an Java version that came from an Ada version of a NAG Fortran
// library subroutine TOMS 535.
void Complexeigenvalue(int n, complex<double> **A, complex<double> lambda[], bool test) {
    int j, k, m, mm, its, itn, ien;
    double anorm = 0.0;
    double ahr, aahr, eps, xr, xi, yr, yi, zr;
    complex<double> accnorm;
    complex<double> x, y, z, yy, T, S;
    eps = pow(2.0, -53); // Double precision, otherwise -23 for float precision
    T = complex<double>(0.0, 0.0);
    itn = 30 * n; // Heuristic on maximum iterations
    ien = n - 1; // used as subscript, loop test <=ien
    // ien is decremented
    while (ien >= 0) {
        its = 0;
        // look for small single subdiagonal element
        while (true) {
            k = 0;
            // for kk in reverse low+1..ien loop
            for (int kk = ien; kk > 0; kk--) {
                ahr = sumabs(A[kk][kk - 1]);
                aahr = eps * (sumabs(A[kk - 1][kk - 1]) +
                sumabs(A[kk][kk]));
                if (ahr <= aahr) {
                    k = kk;
                    break;
                }
            }
            if (k == ien) { break; }//exit when k = ien;
            if (itn <= 0) {
                return;
            }
        }
        // Compute shift
        if (its == 10 || its == 20) {
            S = complex<double>((fabs(A[ien][ien - 1].real())) +
            fabs(A[ien - 1][ien - 2].real()),
        }
    }
```cpp
else {
    S = A[ien][ien];
    x = A[ien - 1][ien] * (A[ien][ien - 1]);
    if (sumabs(x) > 0.0)
        y = (A[ien - 1][ien - 1] - S) *
    complex<double>(0.5, 0.0);
    z = sqrt(y*y + x);
    if (y.real() * z.real() + y.imag() * z.imag() < 0.0)
        z = complex<double>(-z.real(), -
    z.imag()); // negate();
    yy = y + z;
    S = S - x / yy;
} // end if;
}

for (int i = 0; i <= ien; i++) // for i in low..ien loop
T = T + S;
its = its + 1;
itn = itn - 1;
j = k + 1;

// look for two consecutive small sub-diagonal elements
xr = sumabs(A[ien - 1][ien - 1]);
yr = sumabs(A[ien][ien - 1]);
zw = sumabs(A[ien][ien]);
m = k;
if (test) cout << "Looking for two consecutive small sub-
   diagonal elements" << endl;
for (mm = ien - 1; mm >= j; mm--) // for mm in reverse j..ien-1 loop // 460
    {
        //
        yi = yr;
        yr = sumabs(A[mm][mm - 1]);
        xi = zw;
        zw = xr;
        xr = sumabs(A[mm - 1][mm - 1]);
        if (yr <= (eps * zw / yi *(zw + xr + xi)))
            {
                m = mm;
                break;
            }
    } //end loop;

// triangular decomposition  A = L*R
for (int i = m + 1; i <= ien; i++) //for i in m+1..ien loop
```
\[
\begin{align*}
\{ \\
x &= A[i - 1][i - 1]; \\
y &= A[i][i - 1]; \\
\text{if (sumabs(x) } \geq \text{ sumabs(y))} & \\
\quad \{ \\
\quad \quad z &= y / x; \\
\quad \quad \lambda[i] &= \text{complex<double>}(1.0, 0.0); \\
\quad \} \\
\text{else} & \\
\quad \{ \\
\quad \quad \text{// interchange rows of A} \\
\quad \quad \text{for (int jj = i - 1; jj<n; jj++)} // \text{for j in i-1..n loop} \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad z &= A[i - 1][jj]; \\
\quad \quad \quad \quad A[i - 1][jj] &= A[i][jj]; \\
\quad \quad \quad \quad A[i][jj] &= z; \\
\quad \quad \quad \} // \text{end loop; } \\
\quad \quad z &= x / y; \\
\quad \quad \lambda[i] &= \text{complex<double>}(1.0, 0.0); \\
\quad \} // \text{end if; } \\
A[i][i - 1] &= z; \\
\text{for (int jj = i; jj<n; jj++)} // \text{for j in i .. N loop} \\
A[i][jj] &= A[i][jj] - z*A[i - 1][jj]; \\
\} // \text{end loop; } \\
\end{align*}
\]

// composition R*L = H 
for (int jj = m + 1; jj <= ien; jj++) // for j in m+1..ien loop 
\[
\begin{align*}
\{ \\
x &= A[jj][jj - 1]; \\
A[jj][jj - 1] &= \text{complex<double>}(0.0, 0.0); \\
\end{align*}
\]

// interchange columns of A if necessary 
if (lambda[jj].real() > 0.0) 
\[
\begin{align*}
\{ \\
\quad \text{for (int i = 0; i <= jj; i++)} // \text{for i in low .. j loop} \\
\quad \quad z &= A[i][jj - 1]; \\
\quad \quad A[i][jj - 1] &= A[i][jj]; \\
\quad \quad A[i][jj] &= z; \\
\quad \} // \text{end loop; } \\
\} // \text{end if} \\
\end{align*}
\]

// end interchange columns 

for (int i = 0; i <= jj; i++) // for i in low..j loop 
\[
\begin{align*}
\end{align*}
\]

// end accumulate transformations 

\} // end while loop
Example:

To see how it works I have run the Eigenvalue method against a Polynomial.

\[ P(x) = (x - 1)(x + 2)(x - 3)(x - 4) = x^4 + 2x^3 - 13x^2 - 14x + 24 \]

The initial start Matrix for the Companion matrix is:

Matrix \([4][4]=\]

\[
\begin{bmatrix}
-2, & 13, & 14, & -24, \\
1, & 0, & 0, & 0, \\
0, & 1, & 0, & 0, \\
0, & 0, & 1, & 0, \\
\end{bmatrix}
\]

A Stopping criteria is \(a_{n,n-1}<2^{-53}=1.1E-16\) and then the root is \(a_{nn}+T\), where \(T\) is the total shifts accumulated during the iterations. Initial \(T=0\).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Matrix</th>
</tr>
</thead>
</table>
| 0         | \[
\begin{bmatrix}
-2, & 13, & 14, & -24, \\
1, & 0, & 0, & 0, \\
0, & 1, & 0, & 0, \\
0, & 0, & 1, & 0, \\
\end{bmatrix}
\]  
\(T=0\), root \(z=a_{44}+T=0\) |
| 1         | -8.5, 15.15385, 36.28571, -24,  
-3.25, 7.576923, 18.14286, -12,  
0, -0.1656805, -2.791209, 1.846154,  
0, 0, -1.591837, 1.714286, |
\(T=0\), root \(z=a_{44}+T=1.714285714285714\) |
| 2         | -4.196546, 10.94186, 23.80479, -24,  
0.4922782, 0.7740278, 3.692172, -3.722449,  
0, 0.3553157, -2.325633, 1.414058,  
0, 0, 0.02904114, 0.05584421, |
\(T=0.9960419050229449\); root \(z=a_{44}+T=0.9789211335142289\) |
| 3         | -5.531117, 15.25141, 24.00036, -24,  
-0.2262973, 3.127983, 6.489766, -6.489669,  
0, -0.6452003, -3.585009, 2.588929,  
0, 0, -3.239539e-05, 0.003975545, |
\(T=0.9999940811490913\); root \(z=a_{44}+T=1.000017449846631\) |
| 4         | -4.911526, 9.058602, 24, -24,  
0.1022306, 1.079123, 5.508446, -5.508446,  
0, 0.5593045, -2.16758, 1.167585,  
0, 0, 8.845796e-11, 5.918781e-06, |
\(T=0.999999999775491\); root \(z=a_{44}+T=0.9999999999299005\) |
### Table: Practical Implementation of Polynomial Root Finders

<table>
<thead>
<tr>
<th>n</th>
<th>Polynomial Coefficients</th>
<th>Root Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-5.100082, 19.64759, 24, -24, -0.02638574, 3.91844, 6.007992, -6.007992, 0, -2.125897, -4.818358, 3.818358, 0, 0, -4.121658e-22, 2.245092e-11,</td>
<td>z = a_{54} + T = 0.9999999999999999 &lt; 1.1E-16</td>
</tr>
<tr>
<td></td>
<td>Stopping criteria satisfied:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = 0.9999999999999999, Root found at z = a_{54} + T = 0.9999999999999999, Deflate n = n - 1 and start new search</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-5.100082, 19.64759, 24, -0.02638574, 3.91844, 6.007992, -6.007992, 0, -2.125897, -4.818358, 3.818358, 0, 0, -4.121658e-22, 2.245092e-11,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = 0.9999999999975491; root z = a_{54} + T = -3.818358292455505</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1.895572, 11.96131, 24, 0.08192005, 4.808734, 5.711807, 0, -1.008662872, -0.02704938,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = -1.999297467539244; root z = a_{33} + T = -1.989086837319073</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-2.385603, 12.00001, 24, -0.2368707, 5.384199, 6.7698, 0, -1.314544e-06, -0.0007037192,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = -1.999999760449937; root z = a_{33} + T = -2.000001186778308</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-1.193048, 12, 24, 0.4164578, 4.193047, 4.386095, 0, 6.481681e-14, -2.3955e-07,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = -1.999999999999997; root z = a_{33} + T = -1.9999999999999993</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-5.381894, 12, 24, -2.925873, 8.381894, 12.76379, 0, -2.389547e-28, -3.090083e-14,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stopping criteria satisfied:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = -2.389547e-28; root z = a_{33} + T = -2.0000000000000001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deflate n = n - 1 and start new search</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-5.381894, 12, -2.925873, 8.381894,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = -2.381894370227082; root z = a_{33} + T = -2.381894370227082</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-7, 12, 0,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stopping criteria satisfied:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = 3.0000000000000001; root z = a_{33} + T = 3.0000000000000001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deflate n = n - 1 and start new search</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-7,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T = 3.0000000000000001; root z = a_{11} + T = -4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Finish searching for roots</td>
<td></td>
</tr>
</tbody>
</table>
Simultaneous method

As the name imply simultaneous methods find all roots simultaneous. The benefit is you do not have to deal with deflation of a polynomial and the associated accumulated errors arising from inaccuracy in the deflation process.

Durand-Kerner method

Invented by Wierstrass in 1903 and later rediscover by Durand, Kerner and others. Sometimes is goes by the name Durand-Kerner sometimes by Weierstrass.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Durand-Kerner</td>
<td>2</td>
<td>(\frac{1}{2^\sigma} = 2)</td>
</tr>
</tbody>
</table>

\[
Z_i^{(k+1)} = Z_i^{(k)} - \frac{P(Z_i^{(k)})}{\prod_{j=1,j\neq i}^{n} (Z_i^{(k)} - Z_j^{(k)})} \quad i = 1, ..., n \text{ and } k = 0,1, ...
\]

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

\[
Z_i^{(0)} = (0.4 + 0.9i)^{i-1} \quad i = 1, ..., n
\]

I recommend you use the starting points as outline by Aberth [2], see Aberth in the Appendix.

Algorithm for the Durand-Kerner method for Complex coefficients Polynomial.

```cpp
// Find all root of a polynomial of n degree with complex coefficient using the durand-kerner (Weierstrass method)
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// The roots is stored in res[1..n] where res[n] is the first root found and res[1] the last root.
//
// void DurandKerner( int n, const complex<double> coeff[], complex<double> res[] )
{
    bool dz_flag;
    int itercnt, i, j;
    double f, f0, eps, max_f;
    complex<double> z, zi, dz, fz, fz0, gz0;
    complex<double> *a, *w, *Z;
    bool *finish;
    a = new complex<double> [n + 1]; // Copy the original coefficients
    for (i = 0; i <= n; i++) a[i] = coeff[i];
    // Eliminate zero roots
    n = zeroroots(n, a, res);
    if (n > 2)
```
```cpp
{ 
    w = new complex<double>[(n + 1)];
    Z = new complex<double>[(n + 1)];
    finish = new bool[(n + 1)];
    f0 = abs(a[n]);
    eps = 6 * n * f0 * pow((double)DBL_RADIX, -DBL_MANT_DIG);
    // Calculate starting points
    z = complex<double>(0.4, 0.9);
    for (i = 1; i <= n; i++)
    {
        Z[i] = pow(z, i - 1);
        finish[i] = false;
    }

    max_f = 1; dz_flag = true;
    // Start iteration
    for (itercnt = 1; dz_flag && max_f > eps && itercnt < 2 * MAXITER; itercnt++)
    {
        max_f = 0; dz_flag = false;
        for (i = 1; i <= n; i++)
        {
            if (finish[i] == true) continue;
            zi = Z[i];
            fz0 = horner(n, a, zi); f0 = abs(fz0);
            for ( w[i] = fz0, j = 1; j <= n; j++)
            {
                if (i != j)
                {
                    dz = zi - Z[j];
                    w[i] /= dz;
                }
            }
            dz = w[i];
            z = zi - dz;
            fz = horner(n, a, z); f = abs(fz);
            Z[i] = z;
            dz_flag = dz_flag || (z + dz != z);
            if (f>max_f)
                max_f = f;
            if (f <= eps || (z + dz == z))
            {
                complex<double> z0;
                finish[i] = true;
                if (fabs(z.real()) >= fabs(z.imag()))
                    z0 = complex<double>(z.real());
                else
                    z0 = complex<double>(0, z.imag());
                fz = horner(n, a, z0);
                if (abs(fz) <= f)
                    Z[i] = z = z0;
            }
        }
        for (i = 1; i <= n; i++)
        { 
            res[i] = Z[i];
        }
        delete[] finish, Z, w;
    }
    else
        quadratic(n, a, res);
}
```

**Aberth-Ehrlich method**

Invented by Aberth and Ehrlich in 1967. See Aberth [2]

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 approach root with multiplicity > 1 with linear convergence.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aberth-Ehrlich</td>
<td>3</td>
<td>$\frac{3}{2} = 1.73$</td>
</tr>
</tbody>
</table>

$$z_i^{(k+1)} = z_i^{(k)} - \frac{p(z_i^{(k)})}{p'(z_i^{(k)})} \left( 1 - \frac{p(z_i^{(k)})}{p'(z_i^{(k)})} \sum_{j=1, j \neq i}^{n} \frac{1}{z_i^{(k)} - z_j^{(k)}} \right)$$

Aberth in his original paper [2] also describe suitable starting points for all roots. See Aberth supporting function in the Appendix.

**Algorithm for the Aberth-Ehrlich method for Complex coefficients**

**Polynomial**

Notice it also call a function called startpoint() that calculate a suitable start position for all roots. That can be found in the appendix, to avoid cluttering the implementation of the Aberth-Ehrlich method.

```cpp
void AberthEhrlich(int n, const complex<double> coeff[], complex<double> res[])
{
    bool dz_flag;
    int itercnt, i, j;
    double f, f0, f1, max_f, eps;
```
complex<double> z, zi, dz, fz, fz0, fz1;
complex<double> *a, *w, *Z;
bool *finish;
double *apolyr;

a = new complex<double> [n + 1]; // Copy the original coefficients
for (i = 0; i <= n; i++) a[i] = coeff[i];
// Eliminate zero roots
n = zeroroots(n, a, res);
if (n > 2)
{
    complex<double> *a1 = new complex<double> [n];
    /* Calculate coefficients of f'(x) */
    for (i = 0; i < n; i++)
        a1[i] = a[i] * complex<double>(n - i, 0);
    w = new complex<double> [n + 1];
apolyr = new double [n + 1];
    Z = new complex<double> [n + 1];
    finish = new bool [n + 1];
    // Simple upper bound for P(z) using horner with Complex coefficients
    f0 = abs(a[n]);
    eps = 6 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);
    for (i = 0; i <= n; i++)
        apolyr[i] = abs(a[i]);
    startpoints(n, apolyr, Z);
    for (i = 1; i <= n; i++)
        finish[i] = false;
    max_f = 1; dz_flag = true;
    // Start iteration
    for (itercnt = 1; dz_flag && max_f > eps && itercnt < 100; itercnt++)
    {
        max_f = 0; dz_flag = false;
        for (i = 1; i <= n; i++)
            { if (finish[i] == true) continue;
                z[i] = Z[i];
                fz0 = horner(n, a, z[i]); f0 = abs(fz0);
                fz1 = horner(n - 1, a1, z[i]); f1 = abs(fz1);
                for (w[i] = complex<double>(0, 0), j = 1; j <= n;
                    j++)
                    { if (i != j)
                        { dz = complex<double>(1, 0) / (zi - Z[j]);
                            w[i] += dz;
                        }
                        dz = fz1 / fz0 - w[i];
                        dz = complex<double>(1, 0) / dz;
                        w[i] = dz;
                        z = zi - dz;
                        fz = horner(n, a, z); f = abs(fz);
                        if (f > max_f)
                            max_f = f;
                        if (f <= eps || (z + dz == z))


```cpp
complex<double> z0;
finish[i]=true;
if (fabs(z.real()) >= fabs(z.imag()))
    z0 = complex<double>(z.real());
else
    z0 = complex<double>(0, z.imag());

fz = horner(n, a, z0);
if (abs(fz) <= f)
    Z[i] = z = z0;
```

for (i = 1; i <= n; i++)
    res[i] = Z[i];

delete[] finish, Z, w, a1, apolyr;

else
    quadratic(n, a, res);

delete[] a;
return;

---

**Rutishauser QD method**

Invented by Rutishauser in 1954. To my knowledge, nobody is using this nowadays. The Reasons is not as stable as the QR algorithm (Eigenvalue method) and the convergence order is only linear and requires many iterations to get some accurate roots. For a detail description, see P. Henrici [20].

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rutishauser QD</td>
<td>linear</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Conceptual the QD method is usually show by the table below given a Polynomial:

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

And where the first two rows is given by the Polynomial coefficients.

<table>
<thead>
<tr>
<th>Row 1</th>
<th>0</th>
<th>-a_{n-1}</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 2</td>
<td>0</td>
<td>\frac{a_{n-2}}{a_{n-1}}</td>
<td>\frac{a_{n-3}}{a_{n-2}}</td>
<td>\frac{a_{n-4}}{a_{n-3}}</td>
<td>\ldots</td>
<td>\frac{a_0}{a_1}</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
q_1^1 & \quad q_1^2 & \quad q_1^3 & \quad \ldots & \quad q_1^n \\
e_1^0 & \quad e_1^1 & \quad e_1^2 & \quad e_1^3 & \quad \ldots & \quad e_1^{n-1} & \quad e_1^n \\
q_2^1 & \quad q_2^2 & \quad q_2^3 & \quad \ldots & \quad q_2^n \\
e_2^0 & \quad e_2^1 & \quad e_2^2 & \quad e_2^3 & \quad \ldots & \quad e_2^{n-1} & \quad e_2^n \\
\vdots & \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots & \quad \vdots
\end{align*}
\]

For each iterations, you build a new q and e row.
The relationship between the previous q and e row is given by the recurrence, where \( i \) is the \( i \)'th iteration step.

\[
q_{i+1}^k = q_i^k + (e_i^k - e_{i-1}^k) \quad k = 1, 2, \ldots, n
\]

\[
e_{i+1}^k = e_i^k \frac{q_{i+1}^{k+1}}{q_k^k} \quad \text{for } k = 1, 2, n - 1
\]

\[
e_i^0 = e_i^n = 0
\]

The first row is the \( q_0^k \) the second row is the initial \( e_0^k \) row

Because new q’s only requires knowledge of the previous q row and the previous e row and the same for the e row, you can make a very efficient storage model where you only keep a row vector of the latest q and a row vector of the latest e. If all the roots are simple, you iterate until the \( e_{i-1}^{n-1} \) value is less than \( \text{EPS} \) and then the root is \( q_i^n \). \( \text{EPS} \) is typical chosen as \( 2^{-53} (~1.1\times 10^{-16}) \) using 8-bytes floating point numbers. (IEEE754). However if the root are not simple and have the same magnitude e.g. complex numbers that appears in pairs (complex and complex conjugated number) or we are dealing with a root with multiplicity > 1 then we would need to extract the quadratic factor and solve the 2\textsuperscript{nd} degree polynomial. The behavior indicating a quadratic root is when \( e_{i-2}^{n-2} \) gets less than the \( \text{EPS} \). The Quadratic Polynomial \( x^2 + Ax + B \) where:

\[
A_i^n = q_i^n + q_{i-1}^{n-1}
\]

\[
B_i^n = q_i^n \times q_{i-1}^{n-1}
\]

This polynomial can then be solve directly.

After the first simple root is found, you continue to use the iterations schema for the full “matrix” but now look at \( e_{i-2}^{n-2} \) for when to stop for the 2\textsuperscript{nd} root. In case, the first root was a double root then of course you look for when \( e_{i-3}^{n-3} \) get sufficient small etc. until all roots are found.

There is another drawback for Rutishauser QD method and that is that all coefficients \( a_n \) needs to be \( \neq 0 \). This is of course not always possible to guarantee and thereby the limit usage of that method. Although it could be overcome by using Polynomial Taylor shift to ensure that all \( a_n \neq 0 \) before applying the Rutishauser QD method. Polynomial Taylor shifting is describe elsewhere in this paper.

Since the QD method is not in use nowadays then the source code is only show for the case with a Polynomial with real coefficients and no check and use of Polynomial Taylor shifting.

**Algorithm for the Rutishauser QD method for Real coefficients**

```
// Find all root of a polynomial of n degree with real coefficient using the
// Progressive Rutishauser QD method
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// Requires that an..a0 is not zero
```

The roots is stored in res[1..n] where res[n] is the first root found and res[1] the last root.

```cpp
void RutishauserQD(int n, const double* coeff[], complex<double>* res[])
{
    const int maxiteration = 500;
    int i, j, offset=0;
    double *qV, *eV, *ap, qV2 = 0;
    double eps = pow((double)_DBL_RADIX, -DBL_MANT_DIG);

    ap = new double [n + 1]; // Copy the original coefficients and ensure the polynomial is in monic form
    for (i = 0; i <= n; i++)
        ap[i] = coeff[i]/coeff[0];
    // Eliminate zero roots
    n = zeroroots(n, ap, res);
    // Create the matrix qA and eA
    qV = new double[n];
    eV = new double[n];
    double a, b, aprev, bprev, da, db;
    if(n>0)
    {
        // Setup the first row of qA and eA for the Polynomial
        for (i = 0; i < n; i++)
        {
            if (i == 0)
                qV[i] = -ap[1];
            else
                qV[i] = 0.0;
            if (i == 0)
                eV[i] = 0.0;
            else
                eV[i] = ap[i + 1] / ap[i];
        }
        // Setup a,b for quadratic roots i.e. complex conjugated roots
        a = -(qV[n - 1] + qV[n - 2]);
        b = qV[n - 1] * qV[n - 2];
        // Do QD Iteration
        for (j = 1; j < maxiteration; ++j ) // Max 500 iterations
        {
            // Calculate new qV2 vector. Need if a quadratic root is detected otherwise not
            qV2 = qV[n - 2 - offset];
            // Calculate next qV vrow vector
            for (i = 0; i < n; ++i )
                qV[i] += (i == n - 1 ? -eV[i] : eV[i + 1] - eV[i]);
            // Calculate next eV row vector
            for (eV[0] = 0.0, i = 1; i < n; ++i )
                eV[i] *= (qV[i] / qV[i - 1]);
            // Determine if we need to stop the iteration
            if (fabs(eV[n - 1 - offset]) < eps) // Single root found
                { 
                    res[n - offset] = qV[n - 1 - offset];
                    if (++offset == n) {
                        break; // Finish stop the iteration
                    }
                }
            else
        }
    }
```

3 May 2020
```cpp
// Check if the quadratic factor is converging
aprev = a; bprev = b;
a = -(qV[n - 1 - offset] + qV[n - 2 - offset]);
b = qV[n - 1 - offset] * qV2;
da = a - aprev;
db = b - bprev;
// a+da==a && b+db==b is harder limit than fabs(eV[n - 2 - offset]) < eps
if (fabs(eV[n - 2 - offset]) < eps [a+da==a && b+db==b]) // Double root found pair
as either 2 real root or a pair of complex conjugated roots.
{
    // Find the quadratic roots
double r;
    if (a == 0)
    {
        r = -b;
        if (r < 0)
        {
            r = sqrt(-r);
            res[n - 1 - offset] = complex<double>(0, r);
            res[n - offset] = complex<double>(0, -r);
        }
        else
        {
            r = sqrt(r);
            res[n - 1 - offset] = complex<double>(r, 0);
            res[n - offset] = complex<double>(-r, 0);
        }
    }
    else
    {
        r = 1 - 4 * 1 * b / (a * a);
        if (r < 0)
        {
            res[n - 1 - offset] = complex<double>(-a / 2, a * sqrt(-r) / 2);
            res[n - offset] = complex<double>(-res[n - 1 - offset].real(), -res[n - 1 - offset].imag());
        }
        else
        {
            res[n - 1 - offset] = complex<double>((-1 - sqrt(r)) * a / 2, 0);
            res[n - offset] = complex<double>(b / res[n - 1 - offset].real(), 0);
        }
    }
    offset += 2;
    if (offset == n)
    {
        break; // Finish Stop the iteration
    }
}

// Cleanup
delete[] ap, qV, eV;
```
return;
}

Other Polynomial roots method
There exist many other methods; some is a variation over a previous method. We will just list a few.

**Ostrowski Square root method**

It is a third order convergence method derived by Ostrowski in 1973. The iteration step is outline below.

\[
    z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n)} \frac{1}{\sqrt{1 - \frac{p(z_n)p''(z_n)}{p'(z_n)^2}}}
\]

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ostrowski- Square roots</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

TBD.

**Graeffe’s Root-Squaring method**

Graeffe’s method was among the most popular methods for finding roots of in the 19th and 20th centuries. Graeffe, Dandelin, and Lobachevsky (Householder 1959, Malajovich and Zubelli 1999) invented it independently. Graeffe's method has a number of drawbacks, among which are that its usual formulation leads to exponents exceeding the maximum allowed by floating-point arithmetic and that it can map well-conditioned polynomials into ill-conditioned ones. However, Malajovich avoids these limitations in an efficient implementation and Zubelli (1999), which is the method implemented here. See the two reference [22] & [23]. For further reference, see [http://en.wikipedia.org/wiki/Graeffe's_method](http://en.wikipedia.org/wiki/Graeffe's_method).

**Bairstow’s Method**

Bairstow’s method was invented by Bairstow and publish in 1914.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Convergence order</th>
<th>Efficiency index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bairstow’s</td>
<td>2</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Bairstow’s method is limited to polynomial with real coefficients, and to my knowledge has disappeared from serious numerical analysis, primarily due to its bad habit of lacking convergence and getting unstable when polynomials exceed the degree of 8-10th. However, the advantage of the method is that it always finds two roots at a time. This implementation is straightforward however, with the added twist...
that it will calculate an error bound on the residual portion (Rx+S) to find a stopping
criterion that depends on the actual rounding errors in Bairstow’s method.

### Algorithm for the Bairstow’s method for Real coefficients Polynomial

```cpp
// Find all root of a polynomial of n degree with real coefficients
// using the Bairstow's method
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
// The roots is stored in res[1..n] where res[n] is the first root found and

void Bairstow(int n, const double coeff[], complex<double> res[])
{
    int i, itercnt;
    double r, s, t, u, r_eps, s_eps;
    double p, dp, q, dq, d;
    double *a;
    double b1, b2;

    a = new double[n + 1]; // Copy the original coefficients
    for (i = 0; i <= n; i++) a[i] = coeff[i];
    // Eliminate zero roots
    n = zeroroots(n, a, res);

    while (n > 2)
    {
        p = 1; q = 1; r = 1; s = 1; dp = 1; dq = 1;
        r_eps = s_eps = pow(10.0, -10);
        for (itercnt = 0; (fabs(r) > 10 * r_eps || fabs(s) > 10 * s_eps)
            && itercnt < 4 * MAXITER; itercnt++)
        {
            calc_rstu(n, a, p, q, &r, &s, &t, &u);
            calc_eps(n, a, p, q, &r_eps, &s_eps);
            d = (u - p * t) * u - (-q * t) * t;
            if (d == 0.0)
            {
                // Stalled iterations. Restart with a new set of
                p += dp; q += dq;
                s = 1; r = 1;
                continue;
            }
            if (d < 1) // Check for ill conditions linear solution
                2x2 matrix and adjust eps accordingly
            {
                r_eps /= fabs(d); s_eps /= fabs(d);
            }
            dp = r * u - s * t;
            dq = (u - p * t) * s - (-q * t) * r;
            dp /= d; dq /= d;
            p += dp; q += dq;
            if (dp == 0 && dq == 0)
                break;
        }
        quadratic_pq(n, p, q, res); // Deflate polynomial by a quadratic factor x^2+px+q
        b1 = b2 = 0; n -= 2; b2 = a[0];
        for (i = 1; i <= n; i++)
        {
            // Perform operations...
        }
    }
}
```
a[i] = a[i] - p * b2 - q * b1;
b1 = b2;
b2 = a[i];
}
}
quadratic(n, a, res);
delete[] a;
return;
}

Notice the supporting function: calc_rstu(), calc_eps() and quadratic_pc() can be found in the appendix under the Bairstow section.
Reference

2. O. Aberth, Iteration Methods for finding all zeros of a Polynomial Simultaneously, Mathematics Computation, Vol 27, Number 122, April 1973
20. P. Henrici, Finding zeros of Polynomial by the Q-D Algorithm, Communications of the ACM, Volume 8, Number 9, September 1965
22. G Malajovich, Tangent Graeffe Iteration, 1999
   Informes de Matemática, Série B-118, IMPA
## Appendix

### Jenkins-Traub

**Algorithm for the Jenkins-Traub method for Complex coefficients Polynomial**

```c
/*
*******************************************************************************
*
*               Copyright (c) 2002-2020
*               Henrik Vestermark
*               Denmark&USA
*               
*               All Rights Reserved
*               
*               This source file is subject to the terms and conditions of the
*               Future Team Software License Agreement which restricts the manner
*               in which it may be used.
*               
*******************************************************************************
*
* Module name     :   cpoly.cpp
* Module ID Nbr   :
* Description     :   cpoly.cpp -- Jenkins-Traub real polynomial root finder.
*                     Translation of TOMS493 from FORTRAN to C. This
*                     implementation of Jenkins-Traub partially adapts
*                     the original code to a C environment by restruction
*                     many of the 'goto' controls to better fit a block
*                     structured form. It also eliminates the global memory
*                     Allocation in favor of local, dynamic memory management.
*                     
*                     The calling conventions are slightly modified to return
*                     the number of roots found as the function value.
*                     
*                     INPUT:
*                     opr - double precision vector of real coefficients in
*                     order of
*                     decreasing powers.
*                     opi - double precision vector of imaginary coefficients
*                     in order of
*                     decreasing powers.
*                     degree - integer degree of polynomial
*                     
*                     OUTPUT:
*                     zeror,zeroi - output double precision vectors of the
*                     real and imaginary parts of the zeros.
*                     to be consistent with rpoly.cpp the zeros is
*                     inthe index
*                     [0..max_degree-1]
*                     
*                     RETURN:
*                     returnval:  -1 if leading coefficient is zero,
```

3 May 2020
number of roots found.

Change Record :

* Version Author/Date Description of changes
* ------- ----------- ----------------------
* 01.01 HVE/021101 Initial release
* 01.02 HVE/23-Jan-2020 Fixed an error in the function fxshft where loop
count was set to 12 instead of 12
* 01.03 HVE/24-Jan-2020 Rewrote code to eliminate goto & label statement

End of Change Record

#define version string
static char _V_[] = "@(#)cpoly.cpp 01.03 -- Copyright (C) Henrik Vestermark";

#include "stdio.h"
#include "math.h"
#include <float.h>

static double sr, si, tr, ti, pvr, pvi, are, mre, eta, infin;
static int nn;
static int itercnt;     // HVE
static void noshft( const int l1 );
static void fxshft( const int l2, double *zr, double *zi, int *conv );
static void vrshft( const int l3, double *zr, double *zi, int *conv );
static void calct( int *bol );
static void nexth( const int bol );
static void polyev( const int nn, const double sr, const double si, const
double pr[], const double pi[], double qr[], double qi[], double *pvr, double
*pvi );
static double errev( const int nn, const double qr[], const double qi[], const
double ms, const double mp, const double are, const double mre );
static void cauchy( const int nn, double pt[], double q[], double *fn_val );
static double double_scale( const int nn, const double pt[], const double double
etar[], const double double smalno, const double double base );
static void cdivid( const double ar, const double ai, const double br, const
double bi, double *cr, double *ci );
static double cmod( const double r, const double i );
static void mcon( double *eta, double *infin, double *smalno, double *base );

int cpoly( const double *opr, const double *opi, int degree, double *zeror,
double *zeroi, int info[] )
{
    int cnt1, cnt2, idnn2, i, conv;
    double xx, yy, cosr, sinr, smalno, base, xxx, zr, zi, bnd;

    mcon( &eta, &infin, &smalno, &base );
    are = eta;
    mre = 2.0 * sqrt( 2.0 ) * eta;
    xx = 0.70710678;
    yy = -xx;
    cosr = -0.868756474;
    sinr = -0.99756405;
    nn = degree;

    // Algorithm fails if the leading coefficient is zero
if( opr[0] == 0 && opi[0] == 0 )
    return -1;

    // Allocate arrays
    pr = new double[degree+1];
    pi = new double[degree+1];
    hr = new double[degree+1];
    hi = new double[degree+1];
    qpr = new double[degree+1];
    qpi = new double[degree+1];
    qhr = new double[degree+1];
    qhi = new double[degree+1];
    shr = new double[degree+1];
    shi = new double[degree+1];

    // Remove the zeros at the origin if any
    while( opr[nn] == 0 && opi[nn] == 0 )
    {
        idnn2 = degree - nn;
        zeror[idnn2] = 0;
        zeroi[idnn2] = 0;
        nn--;
    }

    // Make a copy of the coefficients
    for( i = 0; i <= nn; i++ )
    {
        pr[i] = opr[i];
        pi[i] = opi[i];
        shr[i] = cmod(pr[i], pi[i]);
    }

    // Scale the polynomial
    bnd = scale(nn, shr, eta, infin, smalno, base);
    if( bnd != 1 )
        for( i = 0; i <= nn; i++ )
        {
            pr[i] *= bnd;
            pi[i] *= bnd;
        }

    // Loop until all roots are found
    for(bool root_found=false;nn>=1;root_found=false)
    {
        itercnt = 0;
        if( nn <= 1 )
        {
            cdivid( -pr[1], -pi[1], pr[0], pi[0], &zeror[degree-1],
            &zeroi[degree-1] );
            if( info != NULL ) info[degree] = 0;    // HVE
            break;
        }

        // Calculate bnd, a lower bound on the modulus of the zeros
        for( i = 0; i <= nn; i++ )
            shr[i] = cmod(pr[i], pi[i]);

        cauchy(nn, shr, shi, &bnd);

        // Outer loop to control 2 Major passes with different sequences of
        // shifts
        for( cnt1 = 1; cnt1 <= 2; cnt1++ )
        { ... }
{  
  // First stage calculation, no shift  
noshft( 5 );  
  // Inner loop to select a shift  
  for( cnt2 = 1; cnt2 <= 9; cnt2++ )  
  {  
    // Shift is chosen with modulus bnd and amplitude rotated by 94 degree from the previous shift  
    xxx = cosr * xx - sinr * yy;  
    yy = sinr * xx + cosr * yy;  
    xx = xxx;  
    sr = bnd * xx;  
    si = bnd * yy;  
    // Second stage calculation, fixed shift  
    fxshft( 10 * cnt2, &zr, &zi, &conv );  
    if( conv )  
    {  
      // The second stage jumps directly to the third stage iteration  
      // If successful the zero is stored and the polynomial deflated  
      idnn2 = degree - nn;  
      zeror[ idnn2 ] = zr;  
      zeroi[ idnn2 ] = zi;  
      if( info != NULL ) info[ idnn2 + 1 ] = itercnt;  
      // HVE  
      nn--;  
      for( i = 0; i <= nn; i++ )  
      {  
        pr[ i ] = qpr[ i ];  
        pi[ i ] = qpi[ i ];  
      }  
      root_found=true;  
      break;  // Root found. break the inner loop goto restart a search for the next root  
    }  
    // If the iteration is unsuccessful another shift is chosen  
  }  
  // Check if inner loop found a root  
  if(root_found=true)  
  {  
    // Break outer loop if root is found and restart a new search  
    // if 9 shifts fail, the outer loop is repeated with another sequence of shifts  
    if(root_found==false)  
    {  
      // The zerofinder has failed on two major passes  
      // return empty handed with the number of roots found (less than the original degree)  
      degree -= nn;  
    }  
  }  
  // Deallocate arrays  
  delete [] pr;  
  delete [] pi;  
  delete [] hr;  
  delete [] hi;  
  delete [] apr;  
  delete [] api;  
  delete [] qhr;  
  delete [] qhi;  
}
delete [] shr;
delete [] shi;

return degree;
}

// COMPUTES THE DERIVATIVE POLYNOMIAL AS THE INITIAL H POLYNOMIAL AND COMPUTES L1 NO-SHIFT H POLYNOMIALS.
//
static void noshft( const int l1 )
{
    int i, j, jj, n, nm1;
double xni, t1, t2;

    n = nn;
nm1 = n - 1;
for( i = 0; i < n; i++ )
{
    xni = nn - i;
    hr[i] = xni * pr[i] / n;
    hi[i] = xni * pi[i] / n;
}
for( jj = 1; jj <= l1; jj++ )
{
    if( cmod( hr[n - 1], hi[n - 1] ) > eta * 10 * cmod( pr[n - 1], pi[n - 1] ) )
    {
        cdivid( -pr[nn], -pi[nn], hr[n - 1], hi[n - 1], &tr, &ti );
    for( i = 0; i < nm1; i++ )
    {
        j = nn - i - 1;
t1 = hr[j - 1];
t2 = hi[j - 1];
hr[j] = tr * t1 - ti * t2 + pr[j];
hi[j] = tr * t2 + ti * t1 + pi[j];
    }
hr[0] = pr[0];
hi[0] = pi[0];
    }
else
    {
        // If the constant term is essentially zero, shift H coefficients
        for( i = 0; i < nm1; i++ )
        {
            j = nn - i - 1;
hr[j] = hr[j - 1];
hi[j] = hi[j - 1];
        }
hr[0] = 0;
hi[0] = 0;
    }
}

// COMPUTES L2 FIXED-SHIFT H POLYNOMIALS AND TESTS FOR CONVERGENCE.
// INITIATES A VARIABLE-SHIFT ITERATION AND RETURNS WITH THE
// APPROXIMATE ZERO IF SUCCESSFUL.
// L2 - LIMIT OF FIXED SHIFT STEPS
// ZR,ZI - APPROXIMATE ZERO IF CONV IS .TRUE.
// CONV - LOGICAL INDICATING CONVERGENCE OF STAGE 3 ITERATION
//
static void fxshft( const int l2, double *zr, double *zi, int *conv )
{
    int i, j, n;
    int test, pasd, bol;
    double otr, oti, svsr, svsi;

    n = nn;
polyev( nn, sr, si, pr, pi, qpr, qpi, &pvr, &pvi );
test = 1;
pasd = 0;

    // Calculate first T = -P(S)/H(S)
calct( &bol );

    // Main loop for second stage
    for( j = 1; j <= l2; j++ )
    {
        otr = tr;
        oti = ti;

        // Compute the next H Polynomial and new t
        nexth( bol );
calct( &bol );
        *zr = sr + tr;
        *zi = si + ti;
    itercnt++;      // HVE

        // Test for convergence unless stage 3 has failed once or this
        // is the last H Polynomial
        if( !( bol || !test || j == l2 ) )
            if( cmod( tr - otr, ti - oti ) < 0.5 * cmod( *zr, *zi ) )
                {
                    if( pasd )
                        {
                            // The weak convergence test has been passed twice, start the
                            third stage

                            // Iteration, after saving the current H polynomial and shift
                            for( i = 0; i < n; i++ )
                            {
                                shr[ i ] = hr[ i ];
                                shi[ i ] = hi[ i ];
                            }
                            svsr = sr;
                            svsi = si;
                            vrshft( 10, zr, zi, conv );
                            if( *conv ) return;
                        }
                }
        pasd = 1;
else
    pasd = 0;
}

// Attempt an iteration with final H polynomial from second stage
vrshft( 10, zr, zi, conv );
}

// CARRIES OUT THE THIRD STAGE ITERATION.
// L3 - LIMIT OF STEPS IN STAGE 3.
// ZR,ZI - ON ENTRY CONTAINS THE INITIAL ITERATE, IF THE
//         ITERATION CONVERGES IT CONTAINS THE FINAL ITERATE ON EXIT.
// CONV - .TRUE. IF ITERATION CONVERGES
static void vrshft( const int l3, double *zr, double *zi, int *conv )
{
    int b, bol;
    int i, j;
    double mp, ms, omp, relstp, r1, r2, tp;

    *conv = 0;
    b = 0;
    sr = *zr;
    si = *zi;

    // Main loop for stage three
    for( i = 1; i <= l3; i++ )
    {
        itercnt++;     // HVE
        // Evaluate P at S and test for convergence
        polyev( nn, sr, si, pr, pi, qpr, qpi, &pvr, &pvi );
        mp = cmod( pvr, pvi );
        ms = cmod( sr, si );
        if( mp <= 20 * errev( nn, qpr, qpi, ms, mp, are, mre ) )
        {
            // Polynomial value is smaller in value than a bound on the error
            // in evaluating P, terminate the iteration
            *conv = 1;
            *zr = sr;
            *zi = si;
            return;
        }
        if( i != 1 )
        {
            if( !( b || mp < omp || relstp >= 0.05 ) )
            {
                // Iteration has stalled. Probably a cluster of zeros. Do 5 fixed
                // shift steps into the cluster to force one zero to dominate
                tp = relstp;
                b = 1;
                if( relstp < eta ) tp = eta;
                r1 = sqrt( tp );
                r2 = sr * ( 1 + r1 ) - si * r1;
                si = sr * r1 + si * ( 1 + r1 );
                sr = r2;
                polyev( nn, sr, si, pr, pi, qpr, qpi, &pvr, &pvi );
                for( j = 1; j <= 5; j++ )
                {
                    calct( &bol );
                    nexth( bol );
                }
            }
        }
    }
omp = infin;
// Calculate next iterate
calct( &bol );
nexth( bol );
calct( &bol );
if( !bol )
{
  relstp = cmod( tr, ti ) / cmod( sr, si );
sr += tr;
si += ti;
}
continue;
}

// Exit if polynomial value increase significantly
if( mp *0.1 > omp ) return;
}
omp = mp;

// Calculate next iterate
calct( &bol );
nexth( bol );
calct( &bol );
if( !bol )
{
  relstp = cmod( tr, ti ) / cmod( sr, si );
sr += tr;
si += ti;
}
}

// COMPUTES T = -P(S)/H(S).
// BOOL - LOGICAL, SET TRUE IF H(S) IS ESSENTIALLY ZERO.
static void calct( int *bol )
{
  int n;
  double hvr, hvi;
  
  n = nn;

  // evaluate h(s)
polyev( n - 1, sr, si, hr, hi, qhr, qhi, &hvr, &hvi );
  *bol = cmod( hvr, hvi ) <= are * 10 * cmod( hr[ n - 1 ], hi[ n - 1 ] ) ? 1 : 0;
  if( !*bol )
  {
    cdivid( -pvr, -pvi, hvr, hvi, &tr, &ti );
    return;
  }
  
  tr = 0;
  ti = 0;
}

// CALCULATES THE NEXT SHIFTED H POLYNOMIAL.
// BOOL - LOGICAL, IF .TRUE. H(S) IS ESSENTIALLY ZERO
//
static void nexth( const int bol )
{
  int j, n;
double t1, t2;

n = nn;
if( !bol )
{
    for( j = 1; j < n; j++ )
    {
        t1 = qhr[ j - 1 ];
        t2 = qhi[ j - 1 ];
        hr[ j ] = tr * t1 - ti * t2 + qpr[ j ];
        hi[ j ] = tr * t2 + ti * t1 + qpi[ j ];
    }
    hr[ 0 ] = qpr[ 0 ];
    hi[ 0 ] = qpi[ 0 ];
    return;
}

// If h[s] is zero replace H with qh
for( j = 1; j < n; j++ )
{
    hr[ j ] = qhr[ j - 1 ];
    hi[ j ] = qhi[ j - 1 ];
}
hr[ 0 ] = 0;
hi[ 0 ] = 0;
}

// EVALUATES A POLYNOMIAL P AT S BY THE HORNER RECURRANCE
// PLACING THE PARTIAL SUMS IN Q AND THE COMPUTED VALUE IN PV.
//
static void polyev( const int nn, const double sr, const double si, const double pr[], const double pi[], double qr[], double qi[], double *pvr, double *pvi )
{
    int i;
    double t;

    qr[ 0 ] = pr[ 0 ];
    qi[ 0 ] = pi[ 0 ];
    *pvr = qr[ 0 ];
    *pvi = qi[ 0 ];

    for( i = 1; i <= nn; i++ )
    {
        t = ( *pvr ) * sr - ( *pvi ) * si + pr[ i ];
        *pvi = ( *pvr ) * si + ( *pvi ) * sr + pi[ i ];
        *pvr = t;
        qr[ i ] = *pvr;
        qi[ i ] = *pvi;
    }
}

// BOUNDS THE ERROR IN EVALUATING THE POLYNOMIAL BY THE HORNER RECURRANCE.
// QR,QI - THE PARTIAL SUMS
// MS    -MODULUS OF THE POINT
// MP    -MODULUS OF POLYNOMIAL VALUE
// ARE, MRE -ERROR BOUNDS ON COMPLEX ADDITION AND MULTIPLICATION
//
static double errev( const int nn, const double qr[], const double qi[], const double ms, const double mp, const double are, const double mre )
{
    int i;
double e;

e = cmod( qr[ 0 ], qi[ 0 ] ) * mre / ( are + mre );
for( i = 0; i <= nn; i++ )
e = e * ms + cmod( qr[ i ], qi[ i ] );
return e * ( are + mre ) - mp * mre;

// CAUCHY COMPUTES A LOWER BOUND ON THE MODULI OF THE ZEROS OF A
// POLYNOMIAL - PT IS THE MODULUS OF THE COEFFICIENTS.
//
static void cauchy( const int nn, double pt[], double q[], double *fn_val )
{
    int i, n;
double x, xm, f, dx, df;

    pt[ nn ] = -pt[ nn ];

    // Compute upper estimate bound
    n = nn;
x = exp( log( -pt[ nn ] ) - log( pt[ 0 ] ) ) / n;
if( pt[ n - 1 ] != 0 )
{
    // Newton step at the origin is better, use it
    xm = -pt[ nn ] / pt[ n - 1 ];
    if( xm < x ) x = xm;
}

    // Chop the interval (0,x) until f < 0
    while(1)
    {
        xm = x * 0.1;
f = pt[ 0 ];
for( i = 1; i <= nn; i++ )
f = f * xm + pt[ i ];
if( f <= 0 )
    break;
x = xm;
    }
dx = x;

    // Do Newton iteration until x converges to two decimal places
    while( fabs( dx / x ) > 0.005 )
    {
        q[ 0 ] = pt[ 0 ];
for( i = 1; i <= nn; i++ )
    q[ i ] = q[ i - 1 ] * x + pt[ i ];
f = q[ nn ];
df = q[ 0 ];
for( i = 1; i < n; i++ )
df = df * x + q[ i ];
dx = f / df;
x -= dx;
itercnt++; }

    *fn_val = x;
}

// RETURNS A SCALE FACTOR TO MULTIPLY THE COEFFICIENTS OF THE POLYNOMIAL.
// THE SCALING IS DONE TO AVOID OVERFLOW AND TO AVOID UNDETECTED UNDERFLOW
INTERFERING WITH THE CONVERGENCE CRITERION. THE FACTOR IS A POWER OF THE BASE.
PT - MODULUS OF COEFFICIENTS OF P
ETA, INFIN, SMALNO, BASE - CONSTANTS DESCRIBING THE FLOATING POINT ARITHMETIC.

static double scale( const int nn, const double pt[], const double eta, const double infin, const double smalno, const double base )
{
    int i, l;
    double hi, lo, max, min, x, sc;
    double fn_val;

    // Find largest and smallest moduli of coefficients
    hi = sqrt( infin );
    lo = smalno / eta;
    max = 0;
    min = infin;

    for( i = 0; i <= nn; i++ )
    {
        x = pt[ i ];
        if( x > max ) max = x;
        if( x != 0 && x < min ) min = x;
    }

    // Scale only if there are very large or very small components
    fn_val = 1;
    if( min >= lo && max <= hi ) return fn_val;
    x = lo / min;
    if( x <= 1 )
    {
        sc = 1 / ( sqrt( max )* sqrt( min ) );
    }
    else
    {
        sc = x;
        if( infin / sc > max ) sc = 1;
    }
    l = (int)( log( sc ) / log(base ) + 0.5 );
    fn_val = pow( base , l );
    return fn_val;
}

// COMPLEX DIVISION C = A/B, AVOIDING OVERFLOW.
//
static void cdivid( const double ar, const double ai, const double br, const double bi, double *cr, double *ci )
{
    double r, d, t, infin;

    if( br == 0 && bi == 0 )
    {
        // Division by zero, c = infinity
        mcon( &t, &infin, &t, &t );
        *cr = infin;
        *ci = infin;
        return;
    }

    if( fabs( br ) < fabs( bi ) )
    {
        r = br / bi;
        d = bi + r * br;
*cr = ( ar * r + ai ) / d;
*ci = ( ai * r - ar ) / d;
return;
}

r = bi / br;
d = br + r * bi;
*cr = ( ar + ai * r ) / d;
*ci = ( ai - ar * r ) / d;
}

// MODULUS OF A COMPLEX NUMBER AVOIDING OVERFLOW.
//
static double cmod( const double r, const double i )
{
    double ar, ai;
    ar = fabs( r );
    ai = fabs( i );
    if( ar < ai )
        return ai * sqrt( 1.0 + pow( ( ar / ai ), 2.0 ) );
    if( ar > ai )
        return ar * sqrt( 1.0 + pow( ( ai / ar ), 2.0 ) );
    return ar * sqrt( 2.0 );
}

// MCON PROVIDES MACHINE CONSTANTS USED IN VARIOUS PARTS OF THE PROGRAM.
// THE USER MAY EITHER SET THEM DIRECTLY OR USE THE STATEMENTS BELOW TO
// COMPUTE THEM. THE MEANING OF THE FOUR CONSTANTS ARE -
// ETA       THE MAXIMUM RELATIVE REPRESENTATION ERROR WHICH CAN BE DESCRIBED
// AS THE SMALLEST POSITIVE FLOATING-POINT NUMBER SUCH THAT
// 1.0_dp + ETA &gt; 1.0.
// INFINY    THE LARGEST FLOATING-POINT NUMBER
// SMALNO    THE SMALLEST POSITIVE FLOATING-POINT NUMBER
// BASE      THE BASE OF THE FLOATING-POINT NUMBER SYSTEM USED
//
static void mcon( double *eta, double *infiny, double *smalno, double *base )
{
    *base = DBL_RADIX;
    *eta = DBL_EPSILON;
    *infiny = DBL_MAX;
    *smalno = DBL_MIN;
}
Aberth-Ehrlich supporting functions

```c
static bool ctest(const int n, double a[], int il, int i, int ir)
{
    double s1, s2;
    s1=a[i]-a[il];
    s2=a[ir]-a[1];
    s1*=ir-il;
    s2*=i-il;
    if(s1>(s2+0.4)) return true;
    return false;
}

static int cleft(const int n, bool h[], int i)
{
    int il;
    for (il = i - 1; il >= 0; il--)
        if (h[il]) break;
    return il;
}

static int cright(const int n, bool h[], int i)
{
    int ir;
    for (ir = i + 1; ir <= n; ir++)
        if (h[ir]) break;
    return ir;
}

static void cmerge(const int n, double a[], int i, int m, bool h[])
{
    int ir, il, irr, ill;
    bool tstl, tstr;
    il=cleft(n,h,i);
    ir=cright(n,h,i);
    if(ctest(n,a,il,i,ir)) return;
    h[i]=false;

    for (;;)
    {
        if(il==i-m) tstl=true;
        else
        {
            ill=cleft(n,h,il);
            tstl=ctest(n,a,ill,il,ir);
        }
        if(ir==(n<i+m?n:i+m))
            tstr=true;
        else
        {
            irr=cright(n,h,ir);
            tstr=ctest(n,a,il,ir,irr);
        }
    }
    h[il]=tstl;
    h[ir]=tstr;
    if(tstl && tstr ) return;
    if(tstl==false) il=ill;
    if(tstr==false) ir=irr;
}
```
static void convex(const int n, double a[], bool h[])  
{  
    int m, nj, jc, k, i, j;  
    for (i=1; i<=n; i++)  
        h[i]=true;  
    k=(int)(log(n-2.0)/log(2.0));  
    if (pow(2.0, k+1)<=n-2)  
        k++;  
    m=1;  
    for (i=0; i<k; i++)  
    {  
        nj=(int)((n-2.0-m)/(m+m))<0? 0: (int)((n-2.0-m)/(m+m));  
        for (j=0; j<=nj; j++)  
        {  
            jc=(j+j+1)*m+1;  
            cmerge(n, a, jc, m, h);  
        }  
        m+=m;  
    }  
}  

static void startpoints(const int n, double a[], complex<double> start[], *, double radius[]*/ )  
{  
    int iold, i, nz, nzeros, j, jj;  
    double th;  
    double temp, r, ang;  
    const double xsmall=log(DBL_MIN);  
    const double xbig=log(DBL_MAX);  
    const double SIGMA=0.7;  
    bool *h = new bool[n + 1];  
    for (i=0; i<=n; i++)  
    {  
        if (a[i]!=0)  
            a[i]=log(a[i]);  
        else  
            a[i]=1e-30;  
        convex(n, a, h);  
        iold=0;  
        th=PI*2/n;  
        for (i=1; i<=n; i++)  
        {  
            if (h[i]==true)  
            {  
                nzeros=i-iold;  
                temp=(a[iold]-a[i])/nzeros;  
                if (temp<xbig && temp >= xsmall)  
                {  
                    nz=nzeros;  
                    r=1.0/DBL_MAX;  
                }  
                if (temp<xsall)  
                {  
                    nz=nzeros;  
                }  
                if (temp>xbig)  
                {  
                    r=DBL_MAX;  
                    nz=nzeros;  
                }  
                if (temp>xbig && temp>MAX(-xbig,xsmall))  
                {  
                    
                }  
            }  
        }  
        }  
    }  
    delete h;  
}
Practical Implementation of Polynomial Root Finders

```c
r=exp(temp);
}
ang=2.0*PI/nzeros;
for( j=iold;j<i;j++)
{
    jj=j-iold+1;
    start[j+1]=complex<double>(r*cos(ang*jj+th*i+SIGMA),r*sin(ang*jj+th*i+SIGMA));
    iold=i;
}
}

Bairstow supporting functions

// Calculate new r,s,t,u values
//
static void calc_rstu( const int n, const double a[], const double p, const double q, double *r, double *s, double *t, double *u )
{
    int i;
    double b1, b2, b3;
    double c1, c2, c3;

    b1 = 0; b2 = 0;
    c1 = 0; c2 = 0;
    for( i = 0; i <= n - 1; i++ )
    {
        b3 = a[ i ]- p * b2 - q * b1;
        b1 = b2;
        b2 = b3;
        if( i <= n - 3 )
        {
            c3 = b3 - p * c2 - q * c1;
            c1 = c2;
            c2 = c3;
        }
    }
    *r = b3;
    *s = a[ n ] - q * b1;
    *t = c3;
    *u = b1 - q * c1;
}

// Calculate new upper bounds for the errors in evaluation p & q
//
static void calc_eps( const int n, const double a[], const double p, const double q, double *r_eps, double *s_eps )
{
    int i;
    double b1, b2, b3;
    double e1, e2, e3;
    double beta;

    b1 = 0; b2 = 0;
    e1 = 0; e2 = 0;
    beta = pow( 2.0, -52 );
    for( i = 0; i <= n - 1; i++ )
    {
```

\[
\begin{align*}
\{ \\
b_3 &= a[i] - p \cdot b_2 - q \cdot b_1; \\
b_1 &= b_2; \\
b_2 &= b_3; \\
e_3 &= \text{fabs}( b_3 ) \cdot 2 \cdot \beta + \text{fabs}( p ) \cdot e_2 \cdot 3 + \text{fabs}( q ) \cdot e_1 \cdot 2; \\
e_1 &= e_2; \\
e_2 &= e_3; \\
\}
\]

*r_eps = e_3; 
*s_eps = \text{fabs}( a[n] ) \cdot \beta + \text{fabs}( q ) \cdot e_1 \cdot 2;

---

// Solve linear or quadratic equation
//
static void quadratic_pq(const int n, const double p, const double q, std::complex<double> res[]) {
    double r, r2; 
    complex<double> s1, s2; 

    if (n >= 2) 
    {
        if (p == 0) 
        {
            r = -q; 
            if (r < 0) 
            {
                s1 = std::complex<double>(0, sqrt(-r)); 
                s2 = std::complex<double>(0, -s1.imag()); 
            }
            else 
            {
                s1 = std::complex<double>(sqrt(r), 0); 
                s2 = std::complex<double>(-s1.real(), 0); 
            }
        }
        else 
        {
            r = 1 - 4 * 1 * q / (p * p); 
            if (r < 0) 
            {
                s1 = std::complex<double>(-p / 2.0, p * sqrt(-r) / 2); 
                s2 = std::complex<double>(-p / 2.0, -s1.imag()); 
            }
            else 
            {
                r = (-1 - sqrt(r)) * p / 2.0; 
                r2 = q / r; 
                s1 = std::complex<double>(r, 0); 
                s2 = std::complex<double>(r2, 0); 
            }
        }
    }

    res[n - 1] = s1; 
    res[n] = s2; 
}