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

Abstract:
Many practical issues arise from designing and implementing a polynomial root finder. If you look into this, you will discover that many different methods exist for finding the roots of a polynomial. Some are by today's standards considered to be obsolete others are still hanging around and some consider the state of the art.
This paper goes through practical issues arising from developing and implementing these different methods. The paper also highlights some of the many differences between the methods and discusses how to overcome the typical loss of accuracy when dealing with the 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.

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

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

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

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

Some methods require the evaluation of the polynomial at a real or complex point together with the first and second derivatives of the polynomial. Other typical matrix methods do not require 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 restart the search for the next root until all roots have 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 several iterations, you get closer and closer to the roots until some stopping criteria have 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 relatively 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 an algorithm in C++ it is understood that default headers are included 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
- Bairstow
- Ostrowski Square-root
- Steffensen
- Chebyshev
- Arithmetic Mean Newton (AN)
- Harmonic Mean Newton (HN)
- Geometric Mean Newton (GN)
- Midpoint Newton (MN)
- Heronian Mean Newton (HeN)
- Trapezoidal Newton (TN)
- Simpson Newton (SN)
- Root-mean square Newton (RMS)

Before we address any roots-finding methods in detail we first need to get the basic done. The first section shows 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 methods than what is presented here, but this will cover the most useful of the methods out there.

The appendix will cover 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 appropriate to move them to the Appendix.

Change log

16 February 2023, Cleaning up the grammar and smaller corrections.

Contents

Evaluation of Polynomials:

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

To evaluate a polynomial P(z) where

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

We use Horner [1] method given by the recurrence:

$$b_n = a_n \qquad 2$$
$$b_k = b_{k-1} z + a_k \qquad k = n-1, \dots, 0$$
$$P(z) = b_0$$

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 2*n* operations. The above mention recurrence works well for polynomials with real coefficients evaluated at a real point x, as well as for polynomials with complex coefficients evaluated at a complex point Z=x+*i*y in which case multiplication and addition are replaced with the complex multiplication and addition for complex arithmetic given by:

Complex multiplication: (a+*i*b)(c+*i*d) = (ac – bd) + *i*(ad+bc) 3

Complex addition: (a+ib)+(c+id) =(a+c) + *i(*b+d) 4

Since a Complex multiplication requires 4 'real' multiplications and 2 additions the total number of operations involving is 4*n*+2*n* or 6*n* '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:

$$Z = x + iy \qquad 5$$
$$p = -2x$$
$$q = x^2 + y^2$$

$$b_n = a_n$$
$$b_{n-1} = a_{n-1} - pb_n$$
$$b_k = a_k - pb_{k+1} - qb_{k+2} \qquad k = n-2,\ldots,1$$
$$b_0 = a_0 + xb_1 - qb_2$$
$$P(Z) = b_0 + iyb_1$$

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

Polynomial	**Real coefficient**	**Complex coefficients**
Number of operations:		
Real point	2n	4n
Complex point	4n	6n

Algorithm Horner with real coefficients at a real point

```
// Evaluate a polynomial with real coefficients a[] at a real point r
// and return the result fz
// Notice that a[0] is a_n, a[1] is a_n-1 and a[n]=a_0
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

```
// Evaluate a polynomial with real coefficients a[] at a complex point z
// and return the result fz
// Notice that a[0] is a_n, a[1] is a_n-1 and a[n]=a_0
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;
      }
```

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

Algorithm Horner with complex coefficients at a complex point

```
// Evaluate a polynomial with complex coefficients a[] at a complex point z
// and return the result fz
// Notice that a[0] is a_n, a[1] is a_n-1 and a[n]=a_0
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 find new roots until all the roots have been found.

For polynomials with real coefficients, you find either real roots or complex roots. There is a special property for Polynomials with real coefficients that the complex root always appears in pairs as the complex root and its complex conjugated root. For Polynomials with complex coefficients, you only find one root at a time. Therefore, we have three scenarios 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:

$$P(z) = a_n z^n + a_{n-1} z^{n-1}, \ldots, a_1 z + a_0 \qquad 6$$

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

$$P(z) = Q(z)(z - R) \qquad 7$$

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

$$and\ Q(z) = b_{n-1} z^{n-1} + b_{n-2} z^{n-2} + \ldots + b_1 z + b_0$$

Now to obtain the b's you can either start by finding the highest coefficient 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:

$$a_n z^n + a_{n-1} z^{n-1} + ... + a_1 z + a_0 = (b_{n-1} z^{n-1} + b_{n-2} z^{n-2} + ... + b_1 z + b_0)(z - R) \quad 8$$

The recurrence is given by:

$$\begin{aligned} a_n &= b_{n-1} \\ a_k &= b_{k-1} - R * b_k \quad k = n-1,...,1 \\ a_0 &= -R * b_0 \end{aligned} \quad 9$$

Now solve it for b's you get:

$$\begin{aligned} b_{n-1} &= a_n \\ b_k &= a_{k+1} + R * b_{k+1} \quad k = n-2,...,0 \end{aligned} \quad 10$$

This simple algorithm works well for polynomials with real coefficients and real roots and complex coefficients with complex roots using the same recurrence just using complex arithmetic instead. A special case is real coefficients with complex roots. A complex root and its complex conjugated root will be the same as dividing by the polynomial P(Z) with 2nd 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

$$P(z) = Q(z)(z^2 + rz + u) \quad 11$$

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 :

$$\begin{aligned} a_n &= b_{n-2} \\ a_{n-1} &= b_{n-3} + r b_{n-2} \\ a_{n-2} &= b_{n-4} + r b_{n-3} + u b_{n-2} \\ a_2 &= b_0 + r b_1 + u b_2 \\ a_1 &= r b_0 + u b_1 \\ a_0 &= u b_0 \end{aligned}$$

Now solve it for b's you get:

$$b_{n-2} = a_n$$
$$b_{n-3} = a_{n-1} - r * b_{n-2}$$
$$b_k = a_{k+2} - rb_{k+1} - ub_{k+2} \quad k = n-4,\ldots,0$$

Algorithm Forward deflation with real coefficients with a real root

```
// 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 aₙ, a[1] is aₙ₋₁ and a[n]=a₀
//
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

```
// 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 forwarddeflation( const int n, double a[], const complex<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

```
// 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 forwarddeflation( const int n, complex<double> a[], const complex<double> z
)
```

```
{
complex <double> z0 = 0;
for( int j = 0; j < n; j++ )
   a[ j ] = z0 = z0 * z + a[ j ];
return n-1;
}
```

Backward Deflation of Polynomials:

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

$$a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0 = (b_{n-1} z^{n-1} + b_{n-2} z^{n-2} + \ldots + b_1 z + b_0)(z - R) \quad (13)$$

The recurrence is given by:

$$a_0 = -R * b_0 \quad (14)$$
$$a_k = b_{k-1} - R * b_k \quad k = 1,\ldots n-1$$
$$a_n = b_{n-1}$$

Now solve it for b's you get:

$$b_0 = -\frac{a_0}{R} \quad (15)$$
$$b_k = (b_{k-1} - a_k)/R \quad k = 1,\ldots, n-2$$
$$b_{n-1} = a_n$$

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 $u= x^2+y^2$

$$P(z) = Q(z)(z^2 + rz + u) \quad (16)$$
$$where\ P(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0$$
$$and\ Q(z) = b_{n-2} z^{n-2} + b_{n-3} z^{n-3} + \ldots + b_1 z + b_0$$
$$The\ recurrence\ is\ giving\ by:$$
$$a_0 = ub_0$$
$$a_1 = ub_1 + rb_0$$
$$a_k = ub_k + rb_{k-1} + b_{k-2} \quad k = 2,\ldots, n-2$$
$$a_{n-1} = rb_{n-2} + b_{n-3}$$
$$a_n = b_{n-2}$$

Now solve it for b's and you get

$$b_0 = a_0 / u$$
$$b_1 = (a_1 - r * b_0) / u$$
$$b_k = (a_k - b_{k-2} - r b_{k-1}) / u \quad k = 2,...,n-2$$

Algorithm Backward deflation with real coefficients with a real root

```
// 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

```
// 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 = (z0 - s) / z;
			}
	return n - 1;
	}
```

Forward or Backward Deflation?

Wilkinson [2] has shown that to have a stable deflation process you should choose *forward* deflation if you find the roots of the polynomial in increasing magnitude and always deflate the polynomial with the lowest magnitude root first and of course, the opposite *backward* deflation when finding the roots with decreasing magnitude. Although most root-finding algorithms do find them in increasing order, it can't be guaranteed, and therefore 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 deflation and saving the new coefficients in an array B[]. Then do a backward deflation and say 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 newly 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 then take the average for the coefficients k as b_k= ½ (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];
b_k= ½ (B[k]+C[k])
For(i=k-1..0) b_i=C[i];

Algorithm composite deflation of real polynomial with real root

```
// 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
	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
	}
```

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]=a0
//
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 at a suitable point that is in the neighborhood of a root. Luckily, many people have studied this field and there are impressive 45+ methods for creating a priori bound of the roots as outlined by J.McNamee, Numerical Methods for roots of Polynomials [7]. Most a priori bounds are 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 where the strategy is to find the roots with increasing order of magnitude.

Priori for the root with the smallest magnitude.

Most root-finding implementations that I have seen do not pay too much attention to the starting point. E.g. [4] Grant-Hitchins 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:

$$z_0 = \frac{1}{2} min_{0<k} \sqrt[k]{\left|\frac{a_0}{a_k}\right| e^{i\theta}}, \qquad \theta = \arg\left(-\frac{P(0)}{P'(0)}\right) \tag{18}$$

The smallest root is located outside the circle with a 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 for the deflated Polynomial is z_0=600 (nearest root x=1,000)

Since 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:

$$Radius = \min\{n|\frac{a_0}{a_1}|, \sqrt[n]{|\frac{a_0}{a_n}|}\} \quad 19$$

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 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.

Formula	Start point	Priori Smallest
Polynomials	z_0	Radius
$1x^3-6x^2+11x-6=$ (x-1)(x-2)(x-3)	0.27	1.64
$X^4+2x^3-13x^2-14x+24=$ (x-1)(x-2)(x-3)(x-4)	0.68	2.2
$+x^3+2000x^2-5E6x-6E9=$ (x+1000)(x-2000)(x+3000)	600	1817
$x^4+2999x^3-10003E3x^2-2399E7x+24E9=$ (x-1)(x+2000)(x-3000)(x-4000)=	0.5	4.0
X^5-1	1	1

Algorithm for starting point for Polynomial with real coefficients

```
// 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 an, 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;
        return min;
        }
```

The start point algorithm has proven to be very useful since you will always start your search somehow close to the nearest root reducing the amount of iteration you would need to perform.

Priori Bounds for all roots

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

The two methods were Kalantari's formula and Deutsch's 'Simple' formula. Where it was found that Kalantari's method has 30% closer bounds than Deutsch's formula.

Kalantari's formula:

Radius for all roots, ρ (assuming Polynomial is in monic form):

$$\rho = Max_{i=1,\dots,n}|\rho_i| \tag{20}$$

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

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

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

$$|\rho| \leq |a_{n-1}| + Max_{i=0,\dots,n-2}\{|\frac{a_i}{c_{i+1}}|\} \tag{21}$$

Algorithm Kalantari formula for real coefficients

```
// Kalantaris formula for a 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;
		}
	max /= 0.682328;
	return max;
	}
```

Algorithm for Kalantaris formula for Complex coefficients

```
// Kalantaris formula for a priori upper bound for largest root
// 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 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

```
// Deutsch's simple formula for a priori upper bound for the 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 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;
	}
```

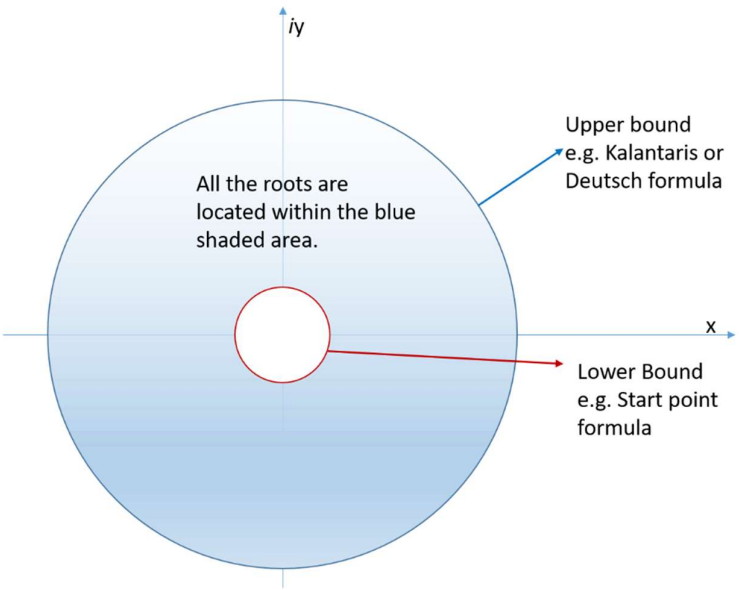
Algorithm for Deutsch formula for complex coefficients

```
// Deutsch's simple formula for a priori upper bound for the largest root
// 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 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 formulas stack up against each other.

Formula	**Kalantaris**	**Deutsch**
Polynomials	Radius	Radius
$1x^3-6x^2+11x-6=$ $(x-1)(x-2)(x-3)$	6.57	7.83
$+x^3+2000x^2-5E6x-6E9=$ $(x+1000)(x-2000)(x+3000)$	4,106	4,500
$x^4+2999x^3-10003E3x^2-2399E7x+24E9=$ $(x-1)(x+2000)(x-3000)(x-4000)=$	5,831	6,334
X^5-1	1.46	Infinity

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



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 root. 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 the iterative method you will at some point need to consider what stopping criteria you want to apply for your root finders. Since most iterative root finders use the evaluation of the polynomial to progress it is only natural to continue our search until the evaluation of P(z) is close enough to 0 to accept the root at that point. It is not all methods that use the value of P(z) as the stopping criteria. Typically Matrix method that does not rely on the evaluation of P(z) uses a different approach discuss later on.

Error in arithmetic operations:

J.H.Wilkinson in "Rounding errors in algebraic processes" [6] has shown that the errors in performing algebraic operations are bound by:

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

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

A simple upper bound:

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

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

A better upper bound.

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

$$b_n = a_n$$
$$b_k = b_{k-1}x + a_k \quad k = n-1,...,0 \qquad 22$$

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

$$e_n = |b_n|\frac{1}{2}$$
$$e_k = e_{k-1}|x| + |b_k| \quad k = n-1,...,0$$
$$e = (4e_0 - 2\,|\,b_0\,|)\varepsilon \quad where\ \varepsilon = \frac{1}{2}\beta^{1-t} \qquad 23$$

Algorithm Kahan Stopping Criteria

```
// 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.

$$
\begin{aligned}
Z &= x + iy \\
p &= -2x \\
q &= x^2 + y^2 \\
b_n &= a_n \\
b_{n-1} &= a_{n-1} - pb_n \\
b_k &= a_k - pb_{k+1} - qb_{k+2} \qquad k = n-2,\ldots,1 \\
b_0 &= a_0 + xb_1 - qb_2 \\
P(Z) &= b_0 + iyb_1
\end{aligned}
\tag{24}
$$

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

$$
\begin{aligned}
e_n &= |b_n|\frac{7}{9} \\
e_k &= e_{k-1}|Z| + |b_k| \quad k = n-1,\ldots,0 \\
e &= (4.5e_0 - 3.5(|b_0| + |b_1||Z|) + |x||b_1|)\varepsilon \quad \textit{where } \varepsilon = \frac{1}{2}\beta^{1-t}
\end{aligned}
\tag{25}
$$

Algorithm Adams Stopping Criteria

```
// 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

$$P(Z) = (a_n + ib_n)z^n + (a_{n-1} + ib_{n-1})z^{n-1} + \ldots + (a_1 + ib_1)z + (a_0 + b_0) \qquad 26$$

Using Horner's method and keeping track of the real component c_k and the imaginary component d_k of the coefficient separately we get:

$$\begin{aligned} c_n &= a_n, \quad d_n = b_n \\ c_k &= c_{k+1}x - yd_{k+1} + a_k \quad k = n-1,\ldots,0 \\ d_k &= d_{k+1}x + yc_{k+1} + b_k \quad k = n-1,\ldots,0 \end{aligned} \qquad 27$$

Using these values an error bound can now be calculated using the recurrence:

$$\begin{aligned} g_n &= 1, \quad h_n = 1 \\ g_k &= |x|(g_{k+1} + |c_{k+1}|) + |y|(h_{k+1} + |d_{k+1}|) + |a_k| + 2|c_k| \quad k = n-1,\ldots,0 \\ h_k &= |y|(g_{k+1} + |c_{k+1}|) + |x|(h_{k+1} + |d_{k+1}|) + |b_k| + 2|d_k| \end{aligned} \qquad 28$$

Now the error is $(g_0+ih_0)\varepsilon$, where $\varepsilon = \frac{1}{2}\beta^{1-t}$. 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:

$$e = (g_0 + ih_0)\varepsilon(1+\varepsilon)^{5n} \quad where\ \varepsilon = \frac{1}{2}\beta^{1-t} \qquad 29$$

Other methods in this category are Igarshi's, Garwick's, and Ward's. The nice parts of 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

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

Igarashi's Stopping criteria

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

$$P(z) = a_n z^n + a_{n-1} z^{n-1}, \dots, a_1 z + a_0 \qquad 30$$

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

$$|P(z_i) - B(z_i)| \geq min\ (|P(z_i)|, |B(z_i)|) \qquad 31$$

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

$$zP'(z) = na_n z^n + (n-1)a_{n-1} z^{n-1} + \cdots + a_1 z \qquad 32$$
$$C(z) = (n-1)a_n z^n + (n-2)a_{n-1} z^{n-1} + \cdots + a_2 z^2 - a_0$$

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 suitable stopping criteria for the root search.

Igarashi suggests 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

```
// 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];
		}
	horner(n, a, r, &px);
	horner(n, zP, r, &zpx);
	horner(n, C, r, &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 real coefficients at a complex point

```
// 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.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;
	}
```

Algorithm Igarashi with complex coefficients at a complex point

```
// 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]) introduces 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 the Garwick precondition issue and states the following stopping criterion:

$$z_{i-1}\ is\ a\ root\ if\ e_i > e_{i-1}, where\ e_i = |z_i - z_{i-1}| \qquad 33$$

JLN [8] replace it to:

$$z_{i-1}\ is\ a\ root\ if\ e_i \geq e_{i-1}, where\ e_i = |z_i - z_{i-1}| \qquad 34$$

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

In addition, the following preconditions hold:

$$(1) \quad e_i \leq 10^{-7}\ if\ |z_{i-1}| < 10^{-4}$$
$$(2) \quad \frac{e_i}{|z_{i-1}|} \leq 10^{-3} if |z_{i-1}| \geq 10^{-4}$$

Algorithm Garwick & Ward with real roots

```
 // 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 have been reached 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

```
// 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 have been reached 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 suggestion eliminates unnecessary function evaluations and immediately stops the iterations when no further improvement to the roots is possible. JLN outlines four possible stopping criteria capable of also handling the ill-conditioned root. The method works equally well for both real and complex roots. Instead of repeating JLN finding, I will just summarize the 4 different stopping criteria

JLN Sopping criterion 1

$$z_i \; is \; a \; root \; if \; \frac{s_i^2}{s_{i-1}} \geq s_m \qquad 35$$

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

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

$$z_{i+1} \; is \; a \; root \; if \; \frac{s_i^2}{s_{i-1}} > s_{i+1} \qquad 36$$

$$Preconditions: s_{i-1} \geq \frac{s_m}{q_m^2} \; and \; s_i - s_{i-1} \geq \frac{s_m}{q_m^2}$$

This stopping criterion is used when the 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

$$z_i \; is \; a \; root \; if \qquad 37$$
$$1: z_0 \neq 0 \; and \; s_1 \geq \frac{s_m}{2}$$
$$2: z_0 = 0 \; and \; s_1 \geq s_m$$

$$z_i \; is \; a \; root \; if$$
$$1: s_i - s_{i-1} \geq \frac{s_m}{2} \; \; or$$
$$2: s_1 - s_{i-1} \geq \frac{s_m}{4} \; and \; s_{i+1} - s_i < s_i - s_{i-1} \; when \; i \geq 2$$

JNL Stopping criterion 4

The last stopping criterion is.

$z_{i+1}\ is\ a\ root\ if s_{i+2} \leq s_{i+1}\ with\ the\ precondition$: (38)
$s_{i-1} \geq b, s_i \geq b\ and\ b = 8$

As already, mention I encourage readers to study JLN method [8] in detail and JLN a more elaborate 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 the Rutishauser QD method for finding Polynomial roots. One of the drawbacks of the Rutishauser QD method is that it requires all coefficients to be $a_i \neq 0\ for\ i$ =0,…,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,…,n and we can now find the roots of the new Polynomial to be:

X1=-0.9999999999999998
X2=(-2.8090169943749466+i0.5877852522924708)
X3=(-2.8090169943749466-i0.5877852522924708)
X4=(-1.6909830056250537-i0.951056516295154)
X5=(-1.6909830056250534+i0.9510565162951539)

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

X1=+0.9999999999999998
X2=(-0.8090169943749466+i0.5877852522924708)
X3=(-0.8090169943749466-i0.5877852522924708)
X4=(0.3090169943749745+i0.9510565162951536)
X5=(0.3090169943749745-i0.9510565162951535)

Which are the roots of 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 $P(z) = a_n z^n + a_{n-1} z^{n-1}, \dots, a_1 x + a_0$ (39)
We try to find Polynomial $Q(z) = q_n z^n + q_{n-1} z^{n-1}, \dots, q_1 x + q_0$
That represents the z_0 shifted Polynomial.
Arrange P(z) in a matrix form, where z_0 is the shift value:

$$M = \begin{bmatrix} a_{n-1}z_0^{n-1} & a_n z_0^n & & & \\ a_{n-2}z_0^{n-2} & & a_n z_0^n & & \\ \vdots & & & \ddots & \\ a_1 z_0^1 & & & a_n z_0^n & \\ a_0 z_0^0 & & & & a_n z_0^n \end{bmatrix}$$

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

Then $q_i = \frac{M[n,i+1]}{z_0^i}$ for i = 0,1, … , n − 1; $and\ q_n = a_n$

Algorithm for Polynomial Taylor shift with real coefficients

```
/*
Given the n - degree polynomial : p(x) = anx^n + an - 1x^n - 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) = anx^n + an - 1x^n - 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 algorithms. However, we still need a few helpful pieces of code.

Simple roots

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

It is well known that before using any iteration methods you can eliminate simple zeros or zeros of polynomials that can be solved directly e.g. for one or two-degree polynomials. Simple zeros are where roots are x=0. Moreover, it is always when the last coefficient 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 apply 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 Polynomials with real coefficients

```
// For Polynomial with complex coefficients a[],
// Eliminate all zero roots from the polynomial
// N is the degree of the Polynomial
// The complex solutions are stored in res[i]
// The new degree is returned and the remaining coefficients are 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 Polynomials with complex coefficients

```
// For Polynomial with complex coefficients a[],
// Eliminate all zero roots from the polynomial
// N is the degree of the Polynomial
// The complex solutions are stored in res[i]
// The new degree is returned and the remaining coefficients are in a[]
```

```
// Notice that a[0] is an, a[1] is an-1 and a[n]=a0
//
int zeroroots(const int n, const complex<double> a[], complex<double> res[])
	{
	int i;
	for ( i=n; a[i] == complex<double>(0); --i )
		{
		res[i] = complex<double>(0.0);
		}
	return i;
	}
```

The 1st order and quadratic solutions are pretty straightforward and are presented without any explanation.

Algorithm for Quadratic roots for Polynomial with real coefficients

```
// Solve the 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].real(), 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]));
				res[2] = complex<double>(res[1].real(), -
res[1].imag());
				}
			else
				{
				res[1] = complex<double>((-1 - sqrt(r)) * a[1] / (2 *
a[0]), 0);
				res[2] = complex<double>(a[2] / (a[0] *
res[1].real()), 0);
				}
			}
		}
	else
		if (n == 1)
```

```
                        {
                        res[1] = complex<double>(-a[1] / a[0], 0);
                        }
        }
```

Algorithm for Quadratic roots for Polynomials 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[1] = sqrt(-a[2] / a[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]);
                        res[2] = a[2] / (a[0] * res[1]);
                        }
                }
        }
```

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 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 ∂_j, namely

$$m_j = \lim_{z \to \partial_j} \left\{ \frac{p'(z_n)^2}{p'(z_n)^2 - p(z_n)p''(z_n)} \right\} \quad 40$$

The drawback is of course that you also need access to the second derivative of P().

Traub (1964) uses:

$$m = \frac{ln\,(P(z_n))}{ln\,(\frac{P(z_n)}{P'(z_n)})} \quad 41$$

Rounded to the nearest integer.

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

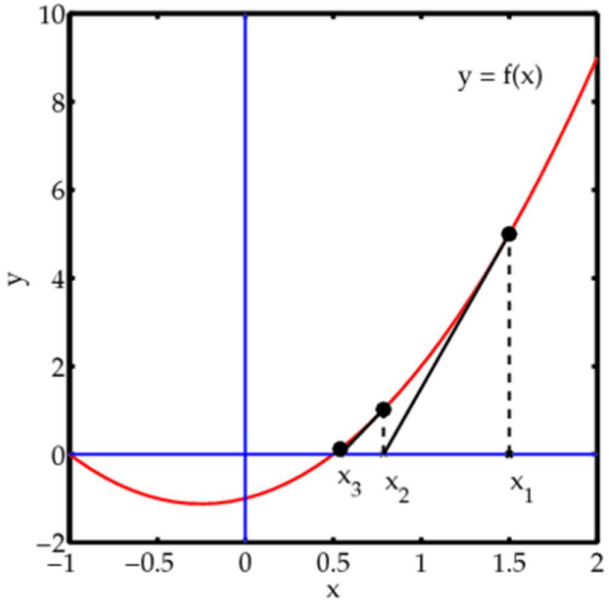
Newton and higher-order derivative-based methods

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 converge, particularly if you start a long way away from the root or do not use a special heuristic to make it converge. The Newton iterations algorithm looks like this.

$$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \qquad 42$$

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



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

Characteristic	Convergence order	Efficiency index
Newton	2	$2^{\frac{1}{2}} = 1.42$

However above formula suffer in convergence speed when dealing with the multiplicity of root >1 there we use the modified version that maintains convergence order even for multiple roots. See [11]

$$z_{n+1} = z_n - m\frac{P(z_n)}{P'(z_n)} \quad (43)$$

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

```
// 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=z=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] provides a very fast and efficient implementation that not only finds the roots in surprisingly few iterations but also handles the usual issues with the Newton method. I do not plan to repeat what is so excellent as described in [5] but just highly some interesting areas of his Newton implementation.

1) The first step is to find the $dz_n = \frac{P(z_n)}{P\prime(z_n)}$ and of course, decide what should happen if $P'(z_n) = 0$. Madsen concludes that when this condition arises 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 to rotating the direction with an odd degree of 53 degrees and multiplying the direction with the factor m. Madsen found that a suitable value for m =5 was reasonable when this happens.
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 $abs(dz_{n)} > 5 * abs(dz_{n-1})$ than the previous iteration's step size. Again he alters the direction with $dz_n = dz_n(0.6 + i0.8) * (\frac{5ab\ (dz_{n-1})}{abs(dz_n)})$

3) These two modifications make his method very resilient 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 looks 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 original z_{n+1}as the new starting point for the next iteration. If not then we accept z_{n+1} as a better choice and continue looking at a 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 altered with $dz_n = dz_n(0.6 + i0.8)$ and we use $z_{n+1} = z_n - dz_n$ as the new starting point for the next iteration.

    if $P(z_{n+1}) \leq P(z_n)$:
    b. Then we are looking in the right direction and we then continue stepping in that direction using $z_{n+1} = z_n - mdz_n \;\; m = 2,..,n$ as long as $P(z_{n+1}) \leq P(z_n)$ and use the best m for the next iterations. The benefit of this process is that if there is a root with the multiplicity of m then m will also be the best choice for the stepping size and this will maintain the 2nd-order convergence rate even for multiple roots.
5) Processes 1-5 continue until the stopping criteria are reached where after the root z_n is accepted and deflated up in the Polynomial and a new search for a root using the deflated Polynomial is initiated.

Madsen also divides the iterations into 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 steps 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 have been satisfied. In case we get outside the convergence circle, we switch back to stage 1 and continue the iteration. Madsen uses 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 $w - \frac{P(w)}{P\prime(w)}$ And radius $|\frac{P(w)}{P\prime(w)}|$ Then we have guarantee convergence if the following two conditions are satisfied:

$$p(w)p'(w) \neq 0 \quad and \qquad (44)$$
$$2|\frac{p(w)}{p'(w)}| \cdot \max_{z\varepsilon K}|p''(z)| \leq |p'(w)|$$

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

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

There are a few more tricks to this than the one described 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 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 in a few placed and ported from the original code in AlgolW to the below C++ implementation. E.g. we use a better upper bound (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

```
// Find all roots of a polynomial of n degree with complex coefficient 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 iteration
	a1 = new complex<double>[n];
	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);
		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 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)
                                {
                                dz *= complex<double>(0.6, 0.8) * (r0 / r); r
= abs(dz);
                                }
                        r0 = 5 * r;
                        }
                z0 = z; f0 = f; f0z = f1z;
                z = z0 - dz;
                fz = horner(n, a, z); ff = f = abs(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)
                                        {//Too 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 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);
            }

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

Halley method

Let's 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.

Halley's method uses the iteration:

$$z_{n+1} = z_n - \frac{2P(z_n)P'(z_n)}{2P'(z_n)^2 - P(z_n)P''(z_n)} \tag{45}$$

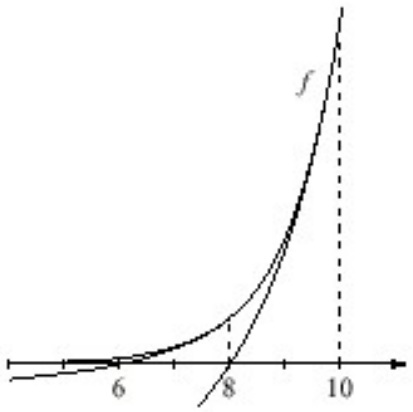
Or sometimes written as: ([13] Peter Acklam)

$$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}\left[1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1} \tag{46}$$

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

$$\left[1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1} \tag{47}$$

And are graphically shown below:



Characteristic	Convergence order	Efficiency index

Halley	3	$3^{\frac{1}{3}} = 1.44$

The efficiency index is slightly larger than the Newton method and 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:

$$z_{n+1} = z_n - \frac{P(z_n)}{\frac{m+1}{2m}P'(z_n) - \frac{P(z_n)P''(z_n)}{2P'(z_n)}} \quad 48$$

Alternatively, written in another way:

$$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}\left[\frac{m+1}{2m} - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1} \quad 49$$

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

$$\left[\frac{m+1}{2m} - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1} \quad 50$$

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

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

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.

Initial guess		0.5
	1	1.7743302038012400
	2	2.1000233483648700
	3	2.1274283810509800
	4	2.1276184100228100
	5	2.1276184100228100

Instead, I use my own modified version for the Halley methods:

$$z_{n+1} = z_n - \frac{m+1}{2}\frac{2P(z_n)P'(z_n)}{2P'(z_n)^2 - P(z_n)P''(z_n)} \quad 51$$

Alternatively, written in another way:

$$z_{n+1} = z_n - \frac{m+1}{2}\frac{P(z_n)}{P'(z_n)}\left[1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1} \quad 52$$

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

$$\frac{m+1}{2}\left[1 - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1} \qquad 53$$

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	1.9999999909793100
	5	1.9999999909793100

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 the below code for an efficient implementation of the Halley method.

Algorithm for the Halley method for Complex coefficients Polynomial

```
// Find all roots 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 iteration
	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);
					}
				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;
						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 errors 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.

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

The Householder's 3rd order method uses the following iteration:

$$z_{n+1} = z_n - \frac{6P(z_n)P'(z_n)^2 - 3P(z_n)^2P''(z_n)}{6P'(z_n)^3 - 6P(z_n)P'(z_n)P''(z_n) + P(z_n)^2P'''(z_n)} \quad 54$$

Substituting:

$$t = \frac{P(z_n)}{P'(z_n)}, u = \frac{P''(z_n)}{P'(z_n)}, v = \frac{P'''(z_n)}{P'(z_n)} \quad 55$$

We can now write the householder's 3rd order as follows:

$$z_{n+1} = z_n - \frac{t(1-0.5tu)}{1-t(u-\frac{vt}{6})} \quad 56$$

Characteristic	Convergence order	Efficiency index
Householder 3rd	4	$4^{\frac{1}{4}} = 1.41$

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

Our modified Householder 3rd order will be:

$$z_{n+1} = z_n - \frac{m+2}{3}\left[\frac{6P(z_n)P'(z_n)^2 - 3P(z_n)^2P''(z_n)}{6P'(z_n)^3 - 6P(z_n)P'(z_n)P''(z_n) + P(z_n)^2P'''(z_n)}\right] \quad 57$$

Or using the same substitution as before:

$$z_{n+1} = z_n - \frac{m+2}{3}\frac{t(1-0.5tu)}{1-t(u-\frac{vt}{6})} \quad 58$$

We use the same "template" for the code for the Householder 3rd order method as for the Newton method and get the below code for an efficient implementation of the Householder 3rd order method.

Algorithm for the Householder method for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with a 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 iteration
	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
* t;
				h = complex<double>(1.0) - t * (u - v * t *
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)
                                                break;
                                        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) *
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 errors 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;
                // Complex Forward deflation of root z
                n = complexdeflation(n, a, z);

                delete[] a3, a2, a1;
                }

        quadratic(n, a, res);
```

```
    delete[] a, a1, a2, a3;

    return;
    }
```

Ostrowski's multi-point method

Ostrowski's multi-point method for root finding is a two-step method (multi-point). The first step is a regular Newton step and the second step is a correction that only requires one extra Horner evaluation. Thereby the method has a very high-efficiency index of 1.59 and is a fourth-order method. The Ostrowski method has generated many 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 a name to another method called Ostrowski square root method, which is not the same as Ostrowski's multi-point method.

Characteristic	**Convergence order**	**Efficiency index**
Ostrowski Multi-point	4	$4^{\frac{1}{3}} = 1.59$

$$y_n = z_n - \frac{p(z_n)}{p'(z_n)} \qquad 59$$
$$z_{n+1} = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n)} \frac{p(y_n)}{p'(z_n)}$$

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

$$\text{Stage 1} \quad y_n = z_n - m\frac{p(z_n)}{p'(z_n)}, m \text{ is the multiplicity} \qquad 60$$
$$\text{Stage 2} \quad z_{n+1} = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n)} \frac{p(y_n)}{p'(z_n)}$$

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 <u>not</u> see any improvement using the multi-step check and or shortening of step, then do the second refinement and obtain a 4^{th}-order convergence for the remaining iterations. Well, what about multiplicity greater than one? 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 4^{th} order method, for simply root it will, however, be a 4^{th} order method.

Algorithm for Ostrowski's multi-point method for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with a complex coefficient
// using the modified Ostrowski
// 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 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
          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 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;
            z = z0 - dz; fz = horner(n, a, z); ff = f = abs(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)
                        {//Too 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 errors using Grant &
Hitchins's test
                eps = upperbound(n, a, z);
                }

           if (f==ff) // No stage 1 improvement
               { // Do the Ostrowski step as the 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 many Ostrowski "like" iterations, capitalizing on the same idea, see [19] E.g. the 6th order convergence:

61

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

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

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

Or

Stage 1 $$y_n = z_n - \frac{p(z_n)}{p'(z_n)}$$ 62

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

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

With a variation on β that gives and accelerated 6th-order convergence. When β=-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 let's see the 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 requires fewer numbers of iterations. However, also more work to be done per iteration.

Method	**Newton**	**Halley**	**Household 3rd**	**Ostrowski**
Iterations	Z	Z	Z	Z
Start guess	0.8320502943378436	0.8320502943378436	0.8320502943378436	0.8320502943378436
1	2.2536991416170737	1.6933271400922734	2.033435992687734	2.0863365344560694
2	1.9233571772166798	1.9899385955094577	1.9999990577501767	1.999968127551831
3	1.9973306906698116	1.9999993042509177	2	2
4	1.999996107736492	2		
5	1.9999999999916678			
6	2			

Laguerre's method

Another interesting method sees in McNamee [7] or [15]. Laguerre's method requires accepting 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.

Characteristic	Convergence order	Efficiency index
Laguerre	3	$3^{\frac{1}{3}} = 1.44$

$$z_{n+1} = z_n - a \qquad 63$$
$$where\ a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}}$$
$$sign\ \pm\ is\ chosen\ to\ maximize\ the\ denominator$$
$$G = \frac{p'(z_n)}{p(z_n)}\ and\ H = G^2 - \frac{p''(z_n)}{p(z_n)}$$

However, the 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.

$$z_{n+1} = z_n - a \qquad 64$$
$$where\ a = \frac{n}{G \pm \sqrt{\left(\frac{n}{m} - 1\right)(nH - G^2)}}$$
$$sign\ \pm\ is\ chosen\ to\ maximize\ the\ denominator$$
$$G = \frac{p'(z_n)}{p(z_n)}\ and\ H = G^2 - \frac{p''(z_n)}{p(z_n)}$$

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 the 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 works extremely well and is a very stable algorithm for finding Polynomial zeros.

Algorithm for Laguerre's method for Complex coefficients Polynomial

```
// Find all roots 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;
	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 iteration
	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; 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);
		fz = horner(n, a, z); f = abs(fz); r0 = 5 * u;
		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)
				{
				for (i = 1; f > fprev && i <= n; i++)
					{ // No improvement. try to shorten the steps
					dz *= 0.5; z = z0 - dz;
					fz = horner(n, a, z ); f = abs(fz);
					if (f > fprev && i == 3)
						{ // If shortening does not help
rotate and try other 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 direction (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 the 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 ways 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, 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 $O(n^2)$ operations. The QR algorithm was developed in the late 1950s. The basic idea is to 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

$$P(z) = a_n z^n + a_{n-1} z^{n-1} +, \dots, a_1 z + a_0 \tag{65}$$

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

$$\begin{bmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \cdots & -\frac{a_1}{a_n} & -\frac{a_0}{a_n} \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{66}$$

Or easier if the polynomial is already in a monic form as:

$$\begin{bmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{67}$$

Algorithm for the Eigenvalue method for Complex coefficients Polynomial

```
// Find all roots 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 a 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 predefine 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 a 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)
{
	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 sub-diagonal 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()),
				fabs(A[ien][ien - 1].imag()) + fabs(A[ien - 1][ien
- 2].imag()));
				}
			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;
				} // end if;

			for (int i = 0; i <= ien; i++) // for i in low..ien loop
				A[i][i] = A[i][i] - S;
			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]);
                    zr = 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 = zr;
                           zr = xr;
                           xr = sumabs(A[mm - 1][mm - 1]);
                           if (yr <= (eps * zr / yi *(zr + 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
                           {
                           x = A[i - 1][i - 1];
                           y = A[i][i - 1];
                           if (sumabs(x) >= sumabs(y))
                                  {
                                  z = y / x;
                                  lambda[i] = complex<double>(-1.0, 0.0);
                                  }
                           else
                                  {
                                  // interchange rows of A
                                  for (int jj = i - 1; jj<n; jj++) // for j in
i-1..n loop
                                         {
                                         z = A[i - 1][jj];
                                         A[i - 1][jj] = A[i][jj];
                                         A[i][jj] = z;
                                         } // end loop;
                                  z = x / y;
                                  lambda[i] = complex<double>(1.0, 0.0);
                                  } // end if;
                           A[i][i - 1] = z;
                           for (int jj = i; jj<n; jj++) // for j in i .. N
loop
                                  A[i][jj] = A[i][jj] - z*A[i - 1][jj];
                           } // end loop;

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

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

                            for (int i = 0; i <= jj; i++) // for i in low..j
loop
                                  {
                                  A[i][jj - 1] = A[i][jj - 1] + x*A[i][jj];
                                  }
                               // end accumulate transformations
                            }
                      }  // end while loop

               // a root is found
               lambda[ien] = A[ien][ien] + T;
               ien = ien - 1;
               } // end loop;
         return; // Success. All roots found
         } // end Complexeigenvalue
```

Example:

To see how it works I have run the Eigenvalue method against the 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]=

-2,	13,	14,	-24,
1,	0,	0,	0,
0,	1,	0,	0,
0,	0,	1,	0,

A Stopping criterion 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. The initial value is T=0.

Iteration	Matrix
0	-2, 13, 14, -24, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 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=0.9999999999775491; root z=a_{44}+T=0.9999999999299005
5	-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, Stopping criteria satisfied: |-4.121658e-22|<1.1E-16 T=0.9999999999775491, Root found at z=a_{44}+T=0.9999999999299005 Deflate n=n-1 and start a new search (notice you continue using the current value of the matrix but only work on the (n-1)x(n-1) elements
0	-5.100082, 19.64759, 24, -0.02638574, 3.91844, 6.007992, 0, -2.125897, -4.818358, T=0.9999999999775491; root z=a_{33}+T=-3.818358292455505
1	-1.895572, 11.96131, 24, 0.08192005, 4.808734, 5.711807, 0, 0.008662872, -0.02704938, T=-1.999297467539244; root z=a_{33}+T=-1.989086837319073
2	-2.385603, 12.00001, 24, -0.2368707, 5.384199, 6.7698, 0, -1.134544e-06, -0.0007037192, T=-1.999999760449937; root z=a_{33}+T=-2.000001186778308
3	-1.193048, 12, 24, 0.4164578, 4.193047, 4.386095, 0, 6.481681e-14, -2.3955e-07, T=-1.99999999999997; root z=a_{33}+T=-1.999999999999903
4	-5.381894, 12, 24, -2.925873, 8.381894, 12.76379, 0, -2.389547e-28, -3.090083e-14, Stopping criteria satisfied |- 2.389547e-28|<1.1E-16 T=-1.99999999999997, Root found at z=a_{33}+T=-2.000000000000001 Deflate n=n-1 and start a new search
0	-5.381894, 12,

	-2.925873, 8.381894, T=-1.99999999999997; root $z=a_{22}+T=6.381894370227082$
1	-7, 12, 0, 0, Stopping criteria satisfied |0|<1.1E-16 T=3.000000000000001, Root found at $z=a_{22}+T=3.000000000000001$ Deflate n=n-1 and start a new search
0	-7, T=3.000000000000001, Root found at $z=a_{11}+T=-4$ Finish searching for roots

Simultaneous method

As the name, imply simultaneous methods find all roots simultaneously. The benefit is you do not have to deal with the 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 rediscovered by Durand, Kerner, and others. Sometimes is goes by the name Durand-Kerner or by Wierstrass.

Characteristic	Convergence order	Efficiency index
Durand-Kerner	2	$2^{\frac{1}{2}} = 2$

$$z_i^{(k+1)} = z_i^{(k)} - \frac{P\left(z_i^{(k)}\right)}{\prod_{j=1, j\neq i}^{n}\left(z_i^{(k)} - z_j^{(k)}\right)} \quad i = 1, \dots, n \; and \; k = 0,1, \dots \tag{68}$$

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

$$z_i^{(0)} = (0.4 + i0.9)^{i-1} \quad i = 1, \dots, n \tag{69}$$

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

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

```
// Find all roots 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)
```

```
		{
		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);

	delete[] a;
```

```
        return;
        }
```

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 approaches roots with multiplicity greater than one with linear convergence.

Characteristic	Convergence order	Efficiency index
Aberth-Ehrlich	3	$3^{\frac{1}{2}} = 1.73$

$$z_i^{(k+1)} = z_i^{(k)} - \frac{\frac{P\left(z_i^{(k)}\right)}{p'\left(z_i^{(k)}\right)}}{1 - \frac{P\left(z_i^{(k)}\right)}{p'\left(z_i^{(k)}\right)} \sum_{j=1, j\neq i}^{n} \frac{1}{\left(z_i^{(k)} - z_j^{(k)}\right)}} \quad i = 1, \ldots, n\,; k = 0,1, \ldots \tag{70}$$

Aberth in his original paper [2] also describes 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 calls a function called startpoint() that calculates a suitable start position for all roots. That function can be found in the appendix, to avoid cluttering the implementation of the Aberth-Ehrich method below.

```
// Find all roots of a polynomial of n degree with a complex coefficient
// using the Aberth-Ehrlich 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.
// This is a port to C++ of D.Bini) original Fortran version, see below.
// NUMERICAL COMPUTATION OF THE ROOTS OF A POLYNOMIAL HAVING
//    COMPLEX COEFFICIENTS, BASED ON ABERTH'S METHOD.
//    Version 1.4, June   1996
//    (D.Bini, Dipartimento di Matematica, Universita' di Pisa)
//       (bini@dm.unipi.it)
// Is has been modified and simplified
//
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;
				zi = Z[i];
				fz0 = horner(n, a, zi); f0 = abs(fz0);
				fz1 = horner(n - 1, a1, zi ); 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);

				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, 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 detailed description, see P. Henrici [20].

Characteristic	Convergence order	Efficiency index
Rutishauser QD	linear	N.A.

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

$$P(z) = a_n z^n + a_{n-1} z^{n-1} +, \dots, a_1 z + a_0 \qquad 71$$

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

$$\begin{array}{lccccccccccc} \text{Row 1} & 0 & -a_{n-1} & & 0 & & 0 & & 0 & & 0 & \\ \text{Row 2} & 0 & & \frac{a_{n-2}}{a_{n-1}} & & \frac{a_{n-3}}{a_{n-2}} & & \frac{a_{n-4}}{a_{n-3}} & \cdots & \frac{a_0}{a_1} & & 0 \\ & 0 & q_1^1 & & q_1^2 & & q_1^3 & & \cdots & & q_1^n & \\ & e_1^0 & & e_1^1 & & e_1^2 & & e_1^3 & \cdots & e_1^{n-1} & & e_1^n \\ & 0 & q_2^1 & & q_2^2 & & q_2^3 & & \cdots & & q_2^n & \\ & e_1^0 & & e_2^1 & & e_2^2 & & e_2^3 & \cdots & e_2^{n-1} & & e_2^n \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{array} \qquad 72$$

For each iteration, 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.

$$q_{i+1}^k = q_i^k + \left(e_i^k - e_i^{k-1}\right) k = 1,2,\dots,n \quad (73)$$
$$e_{i+1}^k = e_i^k \frac{q_{i+1}^{k+1}}{q_{i+1}^k} \; for \; k = 1,2,n-1$$
$$e_i^0 = e_i^n = 0$$

The first row is the q_0^k the second row is the initial e_o^k row

Because new q's only require 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^{n-1} value is less than EPS and then the root is q_i^n. EPS is typically chosen as 2^{-53} (~1.1E-16) using 8-byte floating point numbers. (IEEE754). However, if the root is not simple and have the same magnitude e.g. complex numbers that appear 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 2nd-degree polynomial. The behavior indicating a quadratic root is when e_i^{n-2} gets less than the *EPS*. The Quadratic Polynomial x^2+Ax+B where:

$$A_i^n = q_i^n + q_i^{n-1} \quad (74)$$
$$B_i^n = q_i^n * q_{i-1}^{n-1}$$

This polynomial can then be solved 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^{n-2} for when to stop for the 2nd root. In case, the first root was a double root then of course you look for when e_i^{n-3} get sufficient small etc. until all roots are found.
There is another drawback of the Rutishauser QD method and that is that all coefficients a_n needs to be ≠ 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 described elsewhere in this paper.

Since the QD method is not in use nowadays then the source code is only shown 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 Polynomial

```
// Find all roots 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.
//
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 the next qV vrow vector
			for (i = 0; i < n; ++i )
				qV[i] += (i == n - 1 ? -eV[i] : eV[i + 1] - eV[i]);

			// Calculate the 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
```

```
                        {// 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 (a+da==a && b+db==b)  // Double root found pair
as either 2 real roots 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 of 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 outlined below.

$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n)} \frac{1}{\sqrt{1 - \frac{p(z_n)p''(z_n)}{p'(z_n)^2}}} \quad (75)$$

Due to the square root, the method has the popular name Ostrowski's square root method.

Characteristic	Convergence order	Efficiency index
Ostrowski- Square root	3	$3^{\frac{1}{3}}=1.44$

In the case of a root with a multiplicity greater than one, then Ostrowski also gives a modification, which maintains the cubic convergence rate.

$$z_{n+1} = z_n - \sqrt{m}\frac{p(z_n)}{p'(z_n)} \frac{1}{\sqrt{1 - \frac{p(z_n)p''(z_n)}{p'(z_n)^2}}} \quad (76)$$

Where m is the multiplicity of the root.

Algorithm for the Ostrowski's Square root method for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with complex coefficients
// using the modified Ostrowski
// square root 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 OstrowskiSQ(int n, const complex<double> coeff[], complex<double> res[])
        {
```

```
	int i; bool stage1;
	double r, r0, u, f, f0, eps, f1, f2, ff;
	complex<double> z0, f0z, z, dz, fz1, fz, fz2, fz0, g, h;
	complex<double> *a2, *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
iteration
	a1 = new complex<double>[n];
	a2 = new complex<double>[n-1];
	while (n > 2)  // Iterate for each root
		{
		// Calculate coefficients of p'(x)
		for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i, 0);
		// Calculate coefficients of p''(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]); 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);
		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 the iteration
		while (z + dz != z && f > eps)
			{
			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
				{
				g = fz / fz1;
				fz2 = horner(n - 2, a2, z); f2 = abs(fz2);
				h = fz2 / fz1;
				h = sqrt(complex<double>(1) - g * h );
				dz = g / h;
				// Check if converging
				stage1 = (f2 / f1 > f1 / f / 2) || (f != ff);
				// Check for overstep size of dz
				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 = fz1; fz0 = fz;
```

```
                    z = z0 - dz; fz = horner(n, a, z); ff = f = abs(fz);
                    if (stage1)
                           {  // Try multiple steps or shorten steps depending
of f is an improvement or not
                           bool div2;
                           double fn;
                           complex<double> zn, fzn;

                           zn = z;
                           for (i = 1, div2 = f > f0; i <= n; i++)
                                  {
                                  if (div2 == true )
                                         {  // Shorten steps
                                         dz *= 0.5; zn = z0 - dz;
                                         }
                                  else
                                         zn = z0 - sqrt(i+1)*dz;  // try
another step in the same direction using the Osrowski multiplier of sqrt(m);

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

                                  f = fn; fz = fzn; z = zn;
                                  if (div2 == true && i == 2)
                                         {//Too 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 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);
             }

      quadratic(n, a, res);
      delete[] a2, a1, a;
      }
```

Graeffe's Root-Squaring method

Graeffe's method was among the most popular methods for finding roots in the 19th and 20th centuries. Graeffe, Dandelin, and Lobachevsky (Householder 1959, Malajovich and Zubelli 1999) invented it independently. Graeffe's method has several

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 references [22] & [23]. For further reference, see http://en.wikipedia.org/wiki/Graeffe's_method.

Bairstow's Method

Bairstow's method was invented by Bairstow and published in 1914.

Characteristic	Convergence order	Efficiency index
Bairstow's	2	N.A.

Barstow's method is limited to polynomials 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.
For further information see http://en.wikipedia.org/wiki/Bairstow's_method

Algorithm for Bairstow's method for Real coefficients Polynomial

```
// Find all roots of a polynomial of n degree with real coefficients
// using 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
res[1] the last root.
//
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,q, r, s
                        p += dp; q += dq;
                        s = 1; r = 1;
                        continue;
                        }
                    if (d < 1)  // Check for ill conditions linear solution
2x2 matrix and adjust eps accodingly
                        {
                        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++)
                    {
                    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.

Jenkins-Traub method

Jenkins-Traub algorithm. It is the most used black-box method used in the industry today. Wikipedia has an introduction to the algorithm. See http://en.wikipedia.org/wiki/Jenkins-Traub_algorithm.
It is also complicated and for further study, I refer to the papers by Jenkins & Traub [24]. The Jenkins-Traub method is available for both real and complex coefficient polynomials. See also McNamee [7]. The Jenkins–Traub method has a convergence rate of 2.8, which is higher than the standard Newton method but also more complicated so it is slower than the comparable Newton method.
Due to the size of the algorithm, the source for a modified complex coefficients version is listed in Appendix A.

Chebyshev's method

The third order Chebyshev's method (1840/1841) is given by:

$$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}\left(1 + \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right) \quad 77$$

Luckily, we also have a modified version for cases with multiplicity > 1 as follow:

$$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}\left(\frac{m(3-m)}{2} + \frac{m^2}{2}\frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right) \quad 78$$

Characteristic	Convergence order	Efficiency index
Chebyshev's	3	$3^{\frac{1}{3}} = 1.44$

The drawback is the need for the 2nd prime of $P''(z_n)$.

Algorithm for Chebyshev's method for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with complex coefficients using the
// modified Chebyshev 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 Chebyshev( 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 iteration
	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 ); ff = 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);
					}
				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;
						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 = g * (complex<double>(m*(3-m)/2.0) +
complex<double>(m*m)*h);
```

```
                                        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 errors 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;
    }
```

Newton's method with Integral

Based on Newton's theorem applied for our Polynomial P(z):

$$P(z) = P(z_n) + \int_{z_n}^{z} P'(t)dt \tag{79}$$

You can approach the integral using different numeric methods.

By using the trapezoidal rule for the integral, we obtain:

$$\int_{z_n}^{z} P'(t)dt \approx \frac{z - z_n}{2m}\left[P'(z_n) + 2\sum_{i=1}^{m-1} P'\left(z_n - \frac{i}{m}\frac{P(z_n)}{P'(z_n)}\right) + P'(z)\right] \tag{80}$$

By setting m=1 you get the Arithmetic Mean Newton method (AN).

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \tag{81}$$

$$z_{n+1} = z_n - \frac{2P(z_n)}{P'(z_n) + P'(y_{n+1})} \tag{82}$$

In some other literature, see [25] & [26] they use a notation for y_{n+1} as x_{n+1}^{*} instead of the above notation of y_{n+1}. I found the y_{n+1} notation easier to understand and also you have to calculate that first anyway before you can find the next z_{n+1}. It has a similarity to the multi-point formula but is distinct. You can use a different method to approach an integral part. There exist the following:

- Arithmetic Mean Newton (AN)
- Harmonic Mean Newton (HN)
- Geometric Mean Newton (GN)
- Midpoint Newton (MN)
- Heronian Mean Newton (HeN)
- Trapezoidal Newton (TN)
- Simpson Newton (SN)
- Root-mean square Newton (RMS)

Many more can be developed.

If you use Harmonic Mean instead of Arithmetic mean, you get:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \tag{83}$$

$$z_{n+1} = z_n - \frac{P(z_n)(P'(z_n) + P'(y_{n+1}))}{2P'(z_n)P'(y_{n+1})} \tag{84}$$

If you use Geometric Mean, you obtain:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \tag{85}$$

$$z_{n+1} = z_n - \frac{P(z_n)}{sign(P(z_0))\sqrt{P'(z_n)P'(y_{n+1})}} \tag{86}$$

The Midpoint Mean method is:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \tag{87}$$

$$z_{n+1} = z_n - \frac{P(z_n)}{P'\left(\frac{z_n + y_{n+1}}{2}\right)} \tag{88}$$

The Heronian Mean method is:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \tag{89}$$

$$z_{n+1} = z_n - \frac{3P(z_n)}{P'(z_n) + P'(y_{n+1}) + sign(P(z_0))\sqrt{P'(z_n)P'(y_{n+1})}} \quad 90$$

The Trapezoidal Newton method can be found by setting (78) m=2:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \quad 91$$

$$z_{n+1} = z_n - \frac{4P(z_n)}{P'(z_n) + 2P'\left(\frac{y_{n+1} + z_n}{2}\right) + P'(y_{n+1})} \quad 92$$

Simpson Newton is obtained by using the Simpson's $\frac{1}{3}$ rule is given by:

$$\int_{z_n}^{z} P'(t)dt \approx \frac{z - z_n}{6}\left[P'(z_n) + 4P'\left(\frac{z + z_n}{2}\right) + P'(z)\right] \quad 93$$

You get:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \quad 94$$

$$z_{n+1} = z_n - \frac{6P(z_n)}{P'(z_n) + 4P'\left(\frac{y_{n+1} + z_n}{2}\right) + P'(y_{n+1})} \quad 95$$

Kalyanasundaram [25] furthermore introduces a Harmonic-Simpson-Newton method:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \quad 96$$

$$z_{n+1} = z_n - \frac{3P(z_n)}{\left[\frac{2P'(z_n)P'(y_{n+1})}{P'(z_n) + P'(y_{n+1})}\right] + 2P'\left(\frac{y_{n+1} + z_n}{2}\right)} \quad 97$$

Finally, we have the Root-mean Square Newton:

$$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)} \quad 98$$

$$z_{n+1} = z_n - \frac{\sqrt{2}P(z_n)}{sign(P'(z_0))\sqrt{P'(z_n)^2 + P'(y_{n+1})^2}} \quad 99$$

All these methods have third-order convergence for simple root and linear convergence when multiplicity > 1. On the positive side, you only need P(z) and P'(z). Since they all provide third-order convergence it does not make any sense to

use another method than the Arithmetic or Harmonic mean, which are simple than the other present.

Advantages of these methods:

- 3rd order convergence
- Use only P(z) and P'(z)

Disadvantages of these methods:

- Only linear convergence for multiplicity > 1 or lacking a modified version that can accelerate when multiplicity > 1

Now to overcome these disadvantages I recommend that you combined Newton and the e.g. Arithmetic mean methods. Recalling that the Newton method has an accelerated form for multiplicity greater than one and divides the iteration into two stages.

$$\text{Stage 1} \quad y_{n+1} = z_n - m\frac{P(z_n)}{P'(z_n)} \quad m = multiplicity \tag{100}$$

$$\text{Stage 2} \quad z_{n+1} = z_n - \frac{2P(z_n)}{P'(z_n) + P'(y_{n+1})} \tag{101}$$

Stage 1 is the stage where you use the modified Newton method and get automatically quadratic convergence when encountering a root with a multiplicity greater than one. You also don't waste time on the 2nd stage of extra calculation when you are not near a root.
Stage 2 is the stage where our search has reached a point where we are within the Newton convergence circle so we are sure that the Newton method will converge and then and only then we apply stage 2 of the arithmetic mean method to maintain 3rd order convergence in the final few iterations 3-5 iterations. This approach is equivalent to the way we have implemented Ostrowski multi-point method and below is the source code that implements our Arithmetic mean Newton method.

Algorithm for the Arithmetic Mean Newton's method for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with complex coefficients using
// // the modified Arithmetic Mean Newton 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 ArithmeticMean(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
       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
iteration
	a1 = new complex<double>[n];
	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);
		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 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;
			z = z0 - dz; fz = horner(n, a, z); ff = f = abs(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)
                                        {//Too 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 errors using Grant &
Hitchins's test
                               eps = upperbound(n, a, z);
                            }

                        if (f==ff) // No stage 1 improvement
                            { // Do the Arithmetic Mean step as the second part of
the multi-point iteration
                               complex<double> f1y = horner(n-1,a1,z);
                               z = z0 - 2.0*fz0 / (f1z+f1y);
                               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;
      }
```

Method without the use of derivatives

Steffensen Method

A Danish Mathematician, Steffesen invented a method for finding polynomial zeros without the use of polynomial derivatives in 1933. The method's strength is that it has quadratic convergence order (same as the Newton method) but without the use of polynomial derivatives. We have to remember that the secant method also does not use any derivatives but only a convergence order of approx. 1.62 versus Steffesen's 2nd order.

Steffesen's method uses the formula:

$$z_{n+1} = z_n - \frac{P(z_n)}{\frac{P\left(z_n + P(z_n)\right)}{P(z_n)} - 1} \quad 102$$

By replacing the Newton derivative with a forward finite difference.

You do need to make two evaluations and therefore the Efficiency index (EI) is the same as for the Newton method.

Characteristic	Convergence order	Efficiency index
Steffeson's Method	2	$2^{\frac{1}{2}}$ =1.41

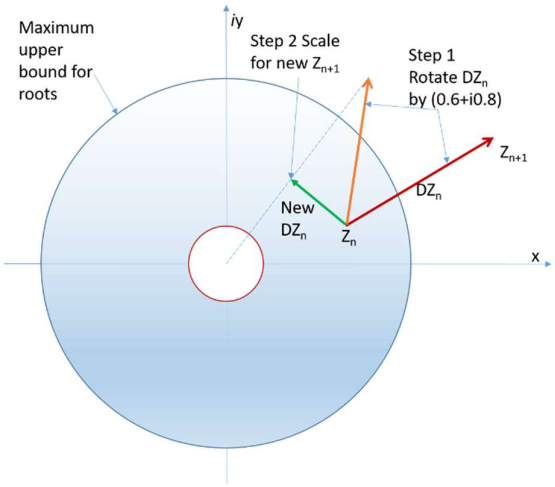
Steffesen's method is relevant, particularly in areas where it is hard to find the first derivative of the function. However, that is not the case for polynomials, where it is easy to find the derivatives of a polynomial. Now for multiplicity greater than one we can cheat a little bit by observing that the denominator is an approximation for the Polynomial derivative and we can therefore use the same multiplier as for Newton methods:

$$z_{n+1} = z_n - m\frac{P(z_n)}{\frac{P\left(z_n + P(z_n)\right)}{P(z_n)} - 1}; where\ m = multiplicity \quad 103$$

Unfortunately, the Steffensen method has a habit it sometimes diverging instead of converging to a root same as the secant method. Therefore, I do not recommend using the Steffensen method.

To avoid the habit of diverging you can apply a safeguard trick when iterating. The trick is that we make a special adjustment of the search direction when an iteration step leads us outside the maximum circle where all the roots are located within the circle (Kalantaris a priori bound for maximum root) then we make a special adjustment of search direction by first rotating the direction (same as we do when we encounter a saddle point) and then we scale the next search point z, so it is located with a radius of the midpoint of Kalantaris a priory bound and our start guesses circle. This makes the implementation more robust and the result is implemented in our general solver program.

The figure below gives a graphic depiction of the process.



The red arrow is the step that leads us outside the maximum circle where roots are located within. The orange arrow is where we would be after a simple rotation and the green arrow is the new DZ_n that leads us to the new Z_{n+1}

Algorithm for Steffesen's method for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with complex coefficients
// using the modified Steffesen
// 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 Steffensen(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,t;
       complex<double> *a;
       int itercnt = 0; int alter;
       double min_radius, max_radius;

       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);
       while (n > 2)  // Iterate for each root
              {// Calculate a suitable starting point
              min_radius = u = startpoint(n, a);
              max_radius = prioriKalantaris(n, a);
              u = prioriSmallest(n, a);
              if (max_radius > 1.5*u)
                     max_radius = 1.5*u;

              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);
              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 the iteration
```

```
              itercnt = 0; alter = 0;
              while (z + dz != z && f > eps)
                     {
                     itercnt++;

                     f1z = horner(n, a, z + fz) / fz - complex<double>(1); f1 =
abs(f1z); // Steffensen
                     if (f1 == 0.0)  // True Saddlepoint
                            dz *= complex<double>(0.6, 0.8) * 5.0;
                     else
                            {
                            dz = fz / f1z;
                            u = abs((f0z - f1z) / (z0 - z));
                            stage1 = (u / f1 > f1 / f / 2) || (f != ff);
                            if(stage1==false)
                                   stage1 = true;
                            r = abs(dz);
                            if (r > r0)
                                   {
                                   dz *= complex<double>(0.6, 0.8) * (r0 / r); r
= abs(dz);
                                   }
                            r0 = 5 * r;
                            // Inside or outside root circle band
                            u = abs(z - dz);
                            if (u<min_radius || u> max_radius)
                                   {
                                   dz *= complex<double>(0.6, 0.8);
                                   dz = (z - dz)*((min_radius + max_radius) / (2
* u) );
                                   r = abs(dz);
                                   }
                        }
                     z0 = z; f0 = f; f0z = f1z; fz0 = fz;
                     z = z0 - dz; fz = horner(n, a, z); ff = f = abs(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)
                                       {//Too many shorten steps try another
direction
```

```
                                        dz *= complex<double>(0.6, 0.8)*(-
pow(2,alter));
                                        z = z0 - dz;
                                        fz = horner(n, a, z); f = abs(fz);
                                        alter++;
                                        break;
                                    }
                                }
                            if (div2 == false)
                                    alter = 0;
                        }
                    else
                        {// 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);
        }

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

Other methods without the use of derivative

However, it is remarkable that M. Kumar Ref [27] construct a 7th-order convergence Newton-type method without using a derivative. In [27] there are several other references to other authors' construction of cubic, sixth, and eighth-order derivative-free methods. The efficiency index is 1.44, 1.56, and 1.68 respectively.

M.Kumar 7th order non-derivative methods also make use of multi-point iterations as follows with an efficient index of 1.63:

104

Stage 1 $$y_n = z_n - \frac{p(z_n)}{p[w_n, z_n]}$$

Stage 2 $$v_n = y_n - [1 + (\frac{P(y_n)}{P(z_n)})^2] \frac{p(y_n)P[z_n, w_n]}{P[y_n, x_n]P[y_n, w_n]}$$

Stage 3 $$z_{n+1} = v_n - [1 + 2\left(\frac{P(y_n)}{P(z_n)}\right)^2 - 4\frac{P(v_n)}{P(y_n)}] \frac{p(v_n)P([,w_n]}{P[y_n,z_n]P[y_n,w_n]}$$, where

$$P[z_n, w_n] = \frac{P(w_n) - P(z_n)}{P(z_n)};\ w_n = z_n + P(z_n)$$

Other Multi-point Method

The Ostrowski multi-point iteration has given rise to many Ostrowski "like" multi-point iterations, capitalizing on the same idea, see H. Nor, A Rahman, A Ismail, A Majid [19] E.g. the 6th order convergence with an Efficient index of 1.56:

$$\text{Stage 1} \quad y_n = z_n - \frac{p(z_n)}{p'(z_n)} \tag{105}$$

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

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

Or

$$\text{Stage 1} \quad y_n = z_n - \frac{p(z_n)}{p'(z_n)} \tag{106}$$

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

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

With a variation on β that gives an accelerated 6th-order convergence. When β=-2 you have the previous 6th-order convergence method.

A.Cordero [28] present an eight-order convergence with an Efficient index of 1.68:

$$\text{Stage 1} \quad y_n = z_n - \frac{p(z_n)}{p'(z_n)} \tag{107}$$

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

$$\text{Stage 3} \quad u_n = v_n - \frac{p(v_n)}{p'(z_n)} \left(\frac{p(z_n) - p(y_n)}{p(z_n) - 2p(y_n)} + \frac{1}{2} \frac{p(v_n)}{p(y_n) - 2p(v_n)}\right)^2$$

$$\text{Stage 4} \quad z_{n+1} = u_n - 3 \frac{p(v_n)}{p'(z_n)} \frac{u_n - v_n}{y_n - z_n}$$

M. Kumar [27] list a ninth-order convergence as follows:

$$\text{Stage 1} \quad y_n = z_n - \frac{p(z_n)}{p'(z_n)} \tag{108}$$

$$\text{Stage 2} \quad v_n = y_n - \left[1 + \left(\frac{p(y_n)}{p(z_n)}\right)^2\right] \frac{p(y_n)}{p'(y_n)}$$

$$\text{Stage 3} \quad z_{n+1} = v_n - \left[1 + 2\left(\frac{p(y_n)}{p(z_n)}\right)^2 - 4 \frac{p(v_n)}{p(y_n)}\right] \frac{p(v_n)}{p'(y_n)}$$

With an Efficient Index (EI) of 1.55

On the positive side, we have a very high-Efficiency Index on the negative side none of these methods exist in a version that maintains the convergence order for multiplicity greater than one.

Solemani, Babajee, and Lotfi see [29] give a list of a few 4^{th}-order multi-point methods that can maintain a 4^{th}-order convergence for multiplicity greater than one.

A general solver program

If you go through most of the code for each method you will notice a lot of similarities in the way we have implemented each method. Most methods share so many commonalities that we can create a supper solver that can handle a variety of the methods present. With the exception that the Matrix, The simultaneous methods, Jenkins Traub, Tangent Graeffe, and a few others are so different in their approach that they can't be incorporated into our general super solver.

We define the following general stages we are within our search for the root.

Stage 0: Initial Setup and Per root Initialization
Stage 1: Search per the method with variable stepsize analysis
Stage 2: Within the converging circle. Use the per method step without exception
Stage 3: Determine the final root, deflate, and repeat the search for the next root.

If we take a look at the initial pseudo-code presented in the chapter Finding the roots of a polynomial. (see below) we can begin to expand to be more general.

```
// 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 RootFinder(int method, int n, double a[], complex<double> res[] )
       {
       // Stage 0 - Global initialization
       while(n>2)
              {
              // Stage 0 - Per root initialization
              dz=z=startpoint(n,a);
              fz=horner(n,a,z);    // fz=P(z)
              EPS=…  // Termination value of |P(z)|
              // Do Per method initialization
              Switch(method)
                     {
                     Case method1: …  break;
                     Case method2: …  break;
                     …
                     }

              //  Stage 1 - Loop until z does not change or |fz|<EPS
              Stage_1=true;
              while(z+dz!=z||abs(fz)<EPS)
                     {
                     // Do Newton, Halley, Ostrowski, Householder
                     // and many other steps per the variable method
                     Switch(method)
                            {
                            Case method1: …  break;
                            Case method2: …  break;
                            …
                            }

                     // Determine which stage
                     If( within_converging_circle())
                            Stage_1=false, Stage_2=true;
```

```
                    Else   Stage_1=true, Stage_2=false;
                    If( Stage 1==true)
                           {
                           // Do variable steps analysis
                           }
                    Else  // Stage 2
                           {
                           // Set Final EPS (a more precise upperbound
                           // than in Stage 0.
                           EPS=upperbound(n,a,z);
                           // Do the remaining Multi-point steps if any
                           Switch(method)
                                  {
                                  Case method1: …  break;
                                  Case method2: …  break;
                    …

                                  }
                           }
                    }

             // Stage 3 - Root found, determine final root, deflate
             // and restart the iterations
             Res[n]=z;  // Save Root found
             n=deflation(n,a,z);  // Deflate Polynomial with found root
             }
      Quadratic(n,a,res);
      }
```

General Solver for Complex coefficients Polynomial

```
// Find all roots of a polynomial of n degree with complex coefficients using the
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 MultiSolver(const int Method, int n, const complex<double> coeff[],
complex<double> res[])
   {
   int i; bool stage1;
   double r, r0, u, f, f0, eps, f3, f2, f1, ff;
   complex<double> z0, f0z, z, dz, fz, fz0, fz1, fz2, fz3, fzprev, s, t, v, g, h,
gm, gp;
   complex<double> *a3=NULL, *a2=NULL, *a1=NULL, *a;
   double min_radius, max_radius;
   int icount;

   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);
   // Allocate temporary memory to hold the derivatives
   switch (Method)
       {
       case MSteffensen:
              min_radius = u = startpoint(n, a);  // Calculate a suitable start
point
              max_radius = prioriKalantaris(n, a);
              u = prioriSmallest(n, a);
                if (max_radius > 1.5*u)
```

```
                    max_radius = 1.5*u;
                break;
          case MNewton:
          case MOstrowskiMP:
          case MArithmeticMean:
                // Create a1 to hold the derivative of the Polynomial a for each
iteration
                a1 = new complex<double>[n];
                break;
          case MHalley:
          case MChebyshev:
          case MOstrowskiSQ:
          case MLaguerre:
                // Create a1, a2 to hold the derivative of the Polynomial a for
each iteration
                a1 = new complex<double>[n];
                a2 = new complex<double>[n-1];
                break;
          case MHouseHolder:
                // Create a1, a2, a3 to hold the derivative of the Polynomial a for
each iteration
                a1 = new complex<double>[n];
                a2 = new complex<double>[n - 1];
                a3 = new complex<double>[n - 2];
                break;
          }

       while (n > 2)  // Iterate for each root
          {
          // Stage 0
          // Calculate the derivatives
          switch (Method)
              {
              case MNewton:
              case MOstrowskiMP:
              case MArithmeticMean:
                    // Calculate coefficients of f'(x)
                    for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i,
0);
                    break;
              case MHalley:
              case MChebyshev:
              case MOstrowskiSQ:
              case MLaguerre:
                    // Calculate coefficients of f'(x)
                    for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i,
0);
                          for (i = 0; i < n-1; i++) a2[i] = a1[i] *
complex<double>(n - i -1, 0);
                    break;
                case MHouseHolder:
                    // Calculate coefficients of f'(x)
                    for (i = 0; i < n; i++) a1[i] = a[i] * complex<double>(n - i,
0);
                       for (i = 0; i < n - 1; i++) a2[i] = a1[i] *
complex<double>(n - i - 1, 0);
                       for (i = 0; i < n - 2; i++) a3[i] = a2[i] *
complex<double>(n - i - 2, 0);
                    break;
                }

          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);
          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 = 2 * n * f0 * pow((double)_DBL_RADIX, -DBL_MANT_DIG);
          // Stage 1 - Start the iteration
          for(icount=0; z + dz != z && f > eps && icount<MAXITER; ++icount)
                 {
                 switch (Method)
                        {
                        case MSteffensen:
                               fz1 = horner(n, a, z + fz) / fz -
complex<double>(1); f1 = abs(fz1);
                               break;
                        default:
                               fz1 = horner(n - 1, a1, z); f1 = abs(fz1);
                               break;
                        }
                 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
                        {
                        switch (Method)
                               {
                               case MNewton:
                               case MOstrowskiMP:
                               case MArithmeticMean:
                               case MSteffensen:
                                      dz = fz / fz1;
                                      break;
                               case MHalley:
                                      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);
                                      break;
                               case MHouseHolder:
                                      t = fz / fz1;
                                      fz2 = horner(n - 2, a2, z); f2 = abs(fz2);
                                      s = fz2 / fz1;
                                      fz3 = horner(n - 3, a3, z); f3 = abs(fz3);
                                      v = fz3 / fz1;
                                      g = complex<double>(1.0) -
complex<double>(0.5) * s * t;
                                      h = complex<double>(1.0) - t * (s - v * t *
complex<double>(1.0 / 6.0));
                                      dz = t * (g / h);
                                      break;
                               case MChebyshev:
                                      g = fz/fz1;
                                      fz2 = horner(n-2,a2,z); f2 = abs(fz2);
                                      h = fz2 / fz1;
                                      h = complex<double>(0.5)*g*h;
                                      dz = g*(complex<double>(1)+h);
                                      break;
                               case MLaguerre:
                                      fz2 = horner(n-2,a2,z); f2 = abs(fz2);
                                      g = fz1/fz;
                                      t = g*g;
                                      h = (t-fz2/fz);
```

```
                                        h = complex<double>(n)*h- t;  // Reused for
later                                   t = complex<double>(n - 1)* h;
                                        t = sqrt(t);
                                        gp = g+t;
                                        gm = g-t;
                                        // Find the maximum value
                                        if (abs(gp)<abs(gm))
                                                gp = gm;
                                        // Calculate dz, change directions if zero
                                        if (abs(gp) == 0.0)
                                                dz = dz*complex<double>(0.6*5.0,
0.8*5.0);
                                        else
                                                dz = complex<double>(n)/ gp;
                                        break;
                                case MOstrowskiSQ:
                                        g = fz/fz1;
                                        fz2 = horner(n-2,a2,z); f2 = abs(fz2);
                                        h = fz2/fz1;
                                        h = sqrt(1.0-g*h);
                                        dz = g/h;
                                        break;
                         }
                         u = abs( (f0z - fz1) / (z0 - z) );
                         stage1 = (u / f1 > f1 / f / 2) || (f != ff);
                         // Check for oversize steps. Rotate if encounter
                         r = abs(dz);
                         if (r > r0)
                             {
                                dz *= complex<double>(0.6, 0.8) * (r0 / r);
                                r = abs(dz);
                             }
                         r0 = 5 * r;
                            }
                        // Inside or outside root circle band. Only Steffensen
method
                        if (Method == MSteffensen)
                                {
                                u = abs(z - dz);
                                if (u<min_radius || u> max_radius)
                                           {
                                           dz *= complex<double>(0.6, 0.8);
                                           dz = (z - dz)*((min_radius + max_radius)
/ (2 * u));
                                           }
                                }
                         z0 = z; f0 = f; f0z = fz1; fzprev=fz0 = fz;
                         z = z0 - dz; fz = horner(n, a, z); ff = f = abs(fz);
                         if (stage1==true)
                                {  // Try multiple steps or shorten steps depending
of f is an improvement or not
                                 double fn;
                                    complex<double> zn, fzn;
                                    zn = z;
                                    if(f>f0)
                                       { // Try shorten the steps
                                       for (i = 1; i <= n; i++)
                                                  {
                                                  dz *= 0.5; zn = z0 - dz;
                                                  fzn = horner(n, a, zn); fn =
abs(fzn);
                                                  if (fn >= f)
                                                         break; // Break if no
improvement
                                                  // Otherwise take the improvement
and try again
```

```
                                                f = fn; fz = fzn; z = zn;
                                                if (i == 2)
                                                        {//Too 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
                                    {  // Try multiple steps in the same direction.
Makes Multiplicity> 1 converge with the method convergence order
                                        for (int m = 2; m <= n; m++)
                                                {
                                                switch (Method)
                                                        {
                                                        case MNewton:
                                                        case MOstrowskiMP:
                                                        case MArithmeticMean:
                                                        case MSteffensen:
                                                                t = z0 -
complex<double>(m)*dz;  // try another step in the same direction
                                                                break;
                                                        case MHalley:
                                                                t =
complex<double>(0.5*(m + 1))*dz;
                                                                break;
                                                        case MHouseHolder:
                                                                t =
complex<double>((m + 2) / 3.0)*dz;
                                                                break;
                                                        case MChebyshev:
                                                                t = g*(
complex<double>(m*(3 - m) / 2.0)+complex<double>(m*m)*h);
                                                                break;
                                                        case MLaguerre:
                                                                t =
sqrt((double(n)/( m - 1))*h);
                                                                gp = g+t;
                                                                gm = g-t;
                                                                // Find the
maximum value
                                                                if
(abs(gp)<abs(gm))
                                                                        gp = gm;
                                                                t =
complex<double>(n)/ gp;
                                                                break;
                                                        case MOstrowskiSQ:
                                                                // try another
step in the same direction using the Osrowski multiplier of sqrt(m);
                                                                t = sqrt(m)*dz;
                                                                break;
                                                        default:  // No
Multiplicity>1 formula exist
                                                                t = z0 -
complex<double>(m)*dz;
                                                                break;
                                                        }
                                                zn = z0 - t;
                                                fzn = horner(n, a, zn); fn =
abs(fzn);
```

```
                                                if (fn >= f)
                                                   break; // Break if no
improvement
                                                f = fn; fz = fzn; z = zn;
                                                {
                                                       f = f;
                                                }
                                                {
                                                       f = f;
                                                }
                                                }
                                         }
                                }
                            else
                                {// Stage 2 calculates the upper bound of error.
                                   eps = upperbound(n, a, z);
                                }

                            if (f==ff) // No stage 1 improvement or In stage 2
                                { // Do the Method multi-step portion if any
                                   switch (Method)
                                          {
                                          case MOstrowskiMP:
                                                 // Do the Ostrowski Step as the
second part of the multi-point iteration
                                                 z = z - fz0 / (fz0 -
complex<double>(2) * fz) * fz / fz1;
                                                 fz = horner(n, a, z); ff = f =
abs(fz);
                                                 break;
                                          case MArithmeticMean:
                                                 // Do the Arithmetic Mean step as the
second part of the multi-point iteration
                                                 t = horner(n-1,a1,z); t = t+fz1;
                                                 t = 2.0*fzprev/t;
                                                 z = z0-t;
                                                 fz = horner(n, a, z); ff = f =
abs(fz); dz = t;
                                          default: // Nothing to do
                                                 break;
                                          }
                                   }
                        }


                    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);

                    if(icount>=MAXITER)
                           throw std::runtime_error("Solver did not reach a root.
Exceeded Max Iteration steps. Result unreliable");
                 }

              quadratic(n, a, res);
              delete[] a;
              if (a1 != NULL) delete[] a1;
              if (a2 != NULL) delete[] a2;
              if (a3 != NULL) delete[] a3;
          }
```

Reference

1. Wikipedia Horner's Method: https://en.wikipedia.org/wiki/Horner%27s_method
2. O. Aberth, Iteration Methods for finding all zeros of a Polynomial Simultaneously, Mathematics Computation, Vol 27, Number 122, April 1973
3. Adams, D A stopping criterion for polynomial root finding. Communication of the ACM Volume 10/Number 10/ October 1967 Page 655-658
4. Grant, J A & Hitchins, G D. Two algorithms for the solution of polynomial equations to limiting machine precision. The computer Journal Volume 18 Number 3, pages 258-264
5. Madsen. A root-finding algorithm based on Newton Method, Bit 13 (1973) 71-75.
6. Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood Cliffs, NJ 1963
7. McNamee, J.M., Numerical Methods for Roots of Polynomials, Part I & II, Elsevier, Kidlington, Oxford 2009
8. Jorgen L. Nikolajsen New Stopping criteria for iterative root finding, Royal Society Open Science, 16 September 2014 http://dx.doi.org/10.1098/rsos.140206
9. Igarashi 1989, A termination criterion for iterative methods used to find the zeros of polynomials. Mathematical Computation, Volume 42, Page 165-171.
10. Kahan W and Farkas I, Algorithm 168 and Algorithm 169. Comm. ACM 6 (Apr. 19G3), 165.
11. H. Vestermark, "A Modified Newton and higher orders Iteration for multiple roots.", www.hvks.com/Numerical/papers.html
12. A. Ostrowski, Solution of equations and systems of equations, Academic Press, 1966.
13. Peter Acklam, A small paper on Halley's method, http://home/online.no~pjacklam, 23rd December 2002
14. E Hansen & M. Patrick, A family of root finding methods, Numerical. Math 27 (1077) 257-269
15. Wikipedia Lageurre's method, https://en.wikipedia.org/wiki/Laguerre%27s_method
16. Wikipedia Eigenvalue algorithm, https://en.wikipedia.org/wiki/Eigenvalue_algorithm
17. W.H. Press & others, Numerical Recipes, 3rd edition 2007, Cambridge University press
18. Wikipedia QR Algorithm, https://en.wikipedia.org/wiki/QR_algorithm
19. H. Nor, A Rahman, A Ismail, A Majid, Numerical Solution of Polynomial Equations using Ostrowski Homotopy Continuation method, MAMTEMATIKA, 2014, Volume 30, Number 1, 47-57
20. P. Henrici, Finding zeros of Polynomial by the Q-D Algorithm, Communications of the ACM, Volume 8, Number 9, September 1965
21. J Gathen, Jürgen Fast Algorithm for Taylor sifts and certain Difference Equations.
22. G Malajovich, Tangent Graeffe Iteration, 1999

23. Malajovich, G. & Zubelli, J. P. "On the Geometry of Graeffe Iteration.", Informes de Mathemática, Série B-118, IMPA
24. M.A. Jenkins & J.F. Traub, ”A three-stage Algorithm for Real Polynomials using Quadratic iteration”, SIAM J Numerical Analysis, Vol. 7, No.4, December 1970.
25. J.J. Kalyanasundaram.M, Modified Newton’s method using Harmonic mean for solving nonlinear equations, IOSR Journal of Mathematics Volume 7, issue 4 (jul-aug 2013), PP 93-97
26. Syamsudhuka, M, Imran, Root mean square Newton’s method, Seminar UNRI-UKM ke 5, 19-21 august 2008 Pekanbaru
27. M Kumar, A.K. Singh, A Srivastava, Various Newton-type iterative methods for solving nonlinear equations
28. A Cordero, J. Torregrossa, M. Vassileva, Three-steps iterative methods with optimal eight-order convergence, Journal of Computational and Applied Mathematics 235 (2011) 3189-3194
29. Soleymani, Babajee, Lotfi, On a numerical technic for finding multiple zeros and its dynamic, Journal of Egyptian Mathematical Society (2013) 21, 346-353
30. Bairstow method, http://en.wikipedia.org/wiki/Bairstow's_method

Appendix

Summarise various methods and their order of convergence

The below list is a summary of the various numerical method for finding zeros of a polynomial. *m* is the multiplicity of the root, and *n* is the degree of the polynomial.

Method	Formula	Order
Newton	$z_{n+1} = z_n - m\frac{P(z_n)}{P'(z_n)}$	2
Halley	$z_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}\left[\frac{m+1}{2m} - \frac{P(z_n)P''(z_n)}{2P'(z_n)^2}\right]^{-1}$	3
Householder 3rd order	$z_{n+1} = z_n - \frac{m+2}{3}\left[\frac{6P(z_n)P'(z_n)^2 - 3P(z_n)^2P''(z_n)}{6P'(z_n)^3 - 6P(z_n)P'(z_n)P''(z_n) + P(z_n)^2P'''(z_n)}\right]$	4
Ostrowski multi-point	$y_n = z_n - \frac{p(z_n)}{p'(z_n)}$ $z_{n+1} = y_n - \frac{p(z_n)}{p(z_n) - 2p(y_n)}\frac{p(y_n)}{p'(z_n)}$	4*
Laguerre	$z_{n+1} = z_n - a$ $where\ a = \frac{n}{G \pm \sqrt{\left(\frac{n}{m} - 1\right)(nH - G^2)}}$ $sign\ \pm\ is\ chosen\ to\ maximize\ the\ denominator$ $G = \frac{p'(z_n)}{p(z_n)}\ and\ H = G^2 - \frac{p''(z_n)}{p(z_n)}$	3
Jenkins-Traub	See Algorithm in ref [24]	2.64
Eigenvalue	See ref [16-17] for the QR method	2
Ostrowski Square-root	$z_{n+1} = z_n - \sqrt{m}\frac{p(z_n)}{p'(z_n)}\frac{1}{\sqrt{1 - \frac{p(z_n)p''(z_n)}{p'(z_n)^2}}}$	3
Graeffe's	See ref [22-23]	2
Steffensen	$z_{n+1} = z_n - m\frac{P(z_n)}{\frac{P(z_n + P(z_n))}{P(z_n)} - 1}$	2
Arithmetic Mean Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{2P(z_n)}{P'(z_n) + P'(y_{n+1})}$	3*
Harmonic Mean Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$	3*

	$z_{n+1} = z_n - \frac{P(z_n)(P'(z_n) + P'(y_{n+1}))}{2P'(z_n)P'(y_{n+1})}$	
Geometric Mean Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{P(z_n)}{sign(P(z_0))\sqrt{P'(z_n)P'(y_{n+1})}}$	3*
Midpoint Mean Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{P(z_n)}{P'\left(\frac{z_n + y_{n+1}}{2}\right)}$	3*
Heronian Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{3P(z_n)}{P'(z_n) + P'(y_{n+1}) + sign(P(z_0))\sqrt{P'(z_n)P'(y_{n+1})}}$	3*
Trapezoidal Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{4P(z_n)}{P'(z_n) + 2P'\left(\frac{y_{n+1} + z_n}{2}\right) + P'(y_{n+1})}$	3*
Simpson Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{6P(z_n)}{P'(z_n) + 4P'\left(\frac{y_{n+1} + z_n}{2}\right) + P'(y_{n+1})}$	3*
Harmonic Simpson Newton	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{3P(z_n)}{\left[\frac{2P'(z_n)P'(y_{n+1})}{P'(z_n) + P'(y_{n+1})}\right] + 2P'\left(\frac{y_{n+1} + z_n}{2}\right)}$	3*
Root mean Square	$y_{n+1} = z_n - \frac{P(z_n)}{P'(z_n)}$ $z_{n+1} = z_n - \frac{\sqrt{2}P(z_n)}{sign(P'(z_0))\sqrt{P'(z_n)^2 + P'(y_{n+1})^2}}$	3*
Bairstow	See ref [30]	2
Kumar	$y_n = z_n - \frac{p(z_n)}{p[w_n, z_n]}$ $v_n = y_n - [1 + (\frac{P(y_n)}{P(z_n)})^2] \frac{p(y_n)P[z_n, w_n]}{P[y_n, x_n]P[y_n, w_n]}$ $z_{n+1} = v_n - [1 + 2\left(\frac{P(y_n)}{P(z_n)}\right)^2 - 4\frac{P(v_n)}{P(y_n)}] \frac{p(v_n)P([,w_n]}{P[y_n,z_n]P[y_n,w_n]}$, where P$[z_n, w_n]$=$\frac{P(w_n)-P(z_n)}{P(z_n)}$; $w_n = z_n + P(z_n)$	9*
Codero	$y_n = z_n - \frac{p(z_n)}{p'(z_n)}$ $v_n = z_n - \frac{p(z_n)}{p'(z_n)} \frac{p(z_n) - p(y_n)}{p(z_n) - 2p(y_n)}$	8*

	$u_n = v_n - \frac{p(v_n)}{p'(z_n)}(\frac{p(z_n)-p(y_n)}{p(z_n)-2p(y_n)} + \frac{1}{2}\frac{p(v_n)}{p(y_n)-2p(v_n)})^2$ $z_{n+1} = u_n - 3\frac{p(v_n)}{p'(z_n)}\frac{u_n - v_n}{y_n - z_n}$	
Nor	$y_n = z_n - \frac{p(z_n)}{p'(z_n)}$ $v_n = y_n - \frac{p(z_n)}{p(z_n)-2p(y_n)}\frac{p(y_n)}{p'(z_n)}$ $z_{n+1} = v_n - \frac{p(z_n)}{p(z_n)-2p(y_n)}\frac{p(v_n)}{p'(z_n)}$	6*

*) When multiplicity>1 then the order reduces to linear

Jenkins-Traub

Algorithm for the Jenkins-Traub method for Complex coefficients Polynomial

```
/*

********************************************************************************
 *
 *
 *                       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,
otherwise
 *                                    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 l2
 *  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 double *pr, *pi, *hr, *hi, *qpr, *qpi, *qhr, *qhi, *shr, *shi;
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 scale( const int nn, const double pt[], const double eta, const
double infin, const double smalno, const 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 *infiny, 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 = -0.060756474;
   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 alll 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, alower 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 shif
            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 ieration
               // 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;  //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 [] qpr;
   delete [] qpi;
   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 passwed 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;
```

```
               //The iteration failed to converge. Turn off testing and restore
h,s,pv and T
               test = 0;
               for( i = 0; i < n; i++ )
                  {
                  hr[ i ] = shr[ i ];
                  hi[ i ] = shi[ i ];
                  }
               sr = svsr;
               si = svsi;
               polyev( nn, sr, si, pr, pi, qpr, qpi, &pvr, &pvi );
               calct( &bol );
               continue;
               }
            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 onthe error
         // in evaluationg P, terminate the ietartion
         *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 RECURRENCE
// 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 RECURRENCE.
// 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

```
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[i];
	s1*=ir-i;
	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<xsmall)
				{
				nz+=nzeros;
				}
			if(temp>xbig)
				{
				r=DBL_MAX;
				nz+=nzeros;
				}
			if(temp<=xbig && temp>MAX(-xbig,xsmall))
				{
```

```
                        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+SI
GMA));
                        }
                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++ )
```

```
                {
                b3 = a[ i ] - p * b2 - q * b1;
                b1 = b2;
                b2 = b3;
                e3 = fabs( b3 ) * 2 * beta + fabs( p ) * e2 * 3 + fabs( q ) * e1
* 2;
                e1 = e2;
                e2 = e3;
                }
        *r_eps = e3;
        *s_eps  = fabs( a[ n ] ) * beta + fabs( q ) * e1 * 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 = complex<double>(0, sqrt(-r));
                                s2 = 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;
                }
        }
```

Tangent Graeffe

The below source code is from Malajovich [27] & [28]. Tangent Graeffe Iteration, from 1999. The source below is under a GNU license and I have only compiled it into a single file

Algorithm for the Tangent Graeffe method for Complex coefficients Polynomial

```
/*
 ****************************************************************************
 *
 *
 *                       Copyright (c) 2002-2020
 *                       Henrik Vestermark
 *                       Denmark
 *
 *                       All Rights Reserved
 *
 *   This source file is subject to the terms and conditions of the
 *   Henrik Vestermark Software License Agreement which restricts the manner
 *   in which it may be used.
 *
 *
 ****************************************************************************
*/

/*
 ****************************************************************************
 *
 *
 * Module name     :   TangentGraeffe.cpp
 * Module ID Nbr   :
 * Description     :   Solve n degree polynomial using Tangent Graeffe methode
 * --------------------------------------------------------------------------
 * Change Record   :
 *
 * Version     Author/Date           Description of changes
 * -------  -----------              ---------------------
 * 01.01       HVE/17-Mar-2017       Initial release
 * 01.02       HVE/19-May-2020       Restructured Source code and create a single
file for the method
 *
 * End of Change Record
 * --------------------------------------------------------------------------
*/


/* define version string */
static char _V_[] = "@(#)TangentGraeffe.cpp 01.02 -- Copyright (C) Henrik
Vestermark";


//#include "stdafx.h"
#include <malloc.h>
#include <time.h>
#include <float.h>
#include <complex>

using namespace std;

#include "globals.h"
```

```
/****************************************************************************
Roots, Version 1.0, May 1998.
By Gregorio Malajovich, gregorio@labma.ufrj.br
Copyright (c) 1997, Gregorio Malajovich.

This  program  solves  univariate  polynomials  using  Renormalized   Graeffe
Iteration. This algorithm was developed by Jorge P. Zubelli and  myself.  See
the man page for references.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, Version 2, June 1991.

Certain files (all the fortran files) are public domain,  published  software
instead, so GNU GPL does not apply to those files.

This program is distributed in the hope that it will be useful,  but  WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public Licence for more details

You should have received a copy of the GNU General Public Licence along  with
this program; if not, write to the Free Software Foundation, Inc. , 675  Mass
Ave, Cambridge, MA 02139, USA.

****************************************************************************/

/****************************************************************************
This file:    globals.c

Created by:   Gregorio Malajovich.
Date:         June 1, 1997.
Purpose:      Contains some global routines, initialization, and global
variables.

Modified by:  Gregorio Malajovich
Date:         June 1997 to May 1998
Changes made: Algorithm improvements, debugging, preparation of a version
suitable for distribution. (1.0)

Modified by:  (Your name)
Date:         (Date)
Changes made: (Short description of changes)

****************************************************************************/

int UNSAFE = 0;
int REALFLAG = 0;
int OUTPUTMODE = 0;
FILE *err = stderr;

MYDOUBLE MACHEPS;
MYDOUBLE MYNAN;
MYDOUBLE GRINFINITY;
MYDOUBLE MYDSIGNIF;
double  MINSEP = 0;
double  MYPROB = 0.01; /* Acceptable probability of failure in
                                       a random complex polynomial */
MYDOUBLE MAXSTEPS;
int      MINSTEPS = 0;
/*
MYDOUBLE MAXMAXSTEPS = 22;
*/
MYDOUBLE MAXMAXSTEPS = 0;

SAFEDOUBLE SAFENAN;
SAFEDOUBLE SAFEMACHEPS;
```

```
SAFEDOUBLE ANGLE = 0;
SAFEDOUBLE CANGLES[3];
int CHEAT = 1;
int NEWTON_ITERATES;

#ifdef COMPARE
char AVAILABLE[MAXALGORITHMS][40] = {
       "Renormalized Tangent Graeffe Iteration",
       "Jenkins and Traub",
       "Aberth (By D. Bini)" };
#else
char AVAILABLE[MAXALGORITHMS][40] = {
       "Renormalized Tangent Graeffe Iteration" };
#endif

void fatal_error(char *routine, char *why)
{

       fprintf(err,
              "\nSorry ! A fatal error has occurred.\nRoutine = %s\nReason
= %s\n",
              routine, why);
       exit(1);
}

void warning(char *routine, char *why)
{

       fprintf(err,
              "Warning at %s ! %s\n",
              routine, why);
}


void *safe_alloc(size_t size)
{
       void *ptr;

       ptr = malloc(size);
       if (ptr == NULL) fatal_error("safe_alloc(size)", "Not enough memory");
       return ptr;
}


struct poly *newpoly(int degree)
{
       struct poly *p;
       int i;

       p = (struct poly *) safe_alloc(sizeof(struct poly)
              + 2 * (degree + 1) * sizeof(MYDOUBLE));
       p->degree = degree;
       p->renorm = 0;
       p->log = &(p->here);
       p->arg = p->log + degree + 1;
       p->sgn = (byte *)p->arg;
       for (i = 0; i <= degree; i++) p->log[i] = p->arg[i] = 0.0;
       return p;
}

void copypoly(struct poly *f, struct poly *g)
{
       int i;

       g->degree = f->degree;
```

```
	g->renorm = f->renorm;

	for (i = 0; i <= f->degree; i++) g->log[i] = f->log[i];
	if (REALFLAG)
		for (i = 0; i <= f->degree; i++) g->sgn[i] = f->sgn[i];
	else
		for (i = 0; i <= f->degree; i++) g->arg[i] = f->arg[i];
}

void init_graeffe(void)		/* Initialize global constants, etc... */

{
	MYDOUBLE one = 1.0, eps = 0.5, sum;
	SAFEDOUBLE safeone = 1.0, safeeps = 0.5, safesum;
	SAFEDOUBLE d0 = 0.0;

	/* Find MACHEPS */
	if (MYMACHEPS == 0)
	{
		sum = 1 + eps;
		while (sum != one)
		{
			eps = eps / 2.0;
			sum = 1 + eps;
			if (eps == 0) fatal_error("init()", "Epsilon=0");


		}
		MACHEPS = eps;
	}
	else MACHEPS = MYMACHEPS;



	MYDSIGNIF = 1 + floor(-log(MACHEPS) / log((double)2.0));

	MYNAN = (MYDOUBLE)sqrt((double)-1.0);
	GRINFINITY = (long double)(1.0 / d0);


	safesum = 1 + safeeps;
	while (safesum != safeone)
	{
		safeeps = safeeps / 2.0;
		safesum = 1 + safeeps;
		if (safeeps == 0) fatal_error("init()", "Epsilon=0 - 2");
	}
	SAFEMACHEPS = eps;


	SAFENAN = (MYDOUBLE)sqrt((double)-1.0);


	/*
	Another possibility is to set:

	MACHEPS = pow(2,-DSIGNIF) ;

	*/
	if (MAXMAXSTEPS == 0) MAXMAXSTEPS = GRINFINITY;

}

void available_algorithms()
{
	int i;
```

```
	fprintf(stderr, "Available algorithms:\n");
	for (i = 0; i< MAXALGORITHMS; i++)
		fprintf(stderr, " -a%d   %s\n", i, AVAILABLE[i]);
	fprintf(stderr, "\n\n");
}



int isinf(SAFEDOUBLE d)
{
	if (d == GRINFINITY)
		return 1;
	else
		if (d == -GRINFINITY)
			return -1;

	return 0;
}

SAFEDOUBLE drand48()
{
	return ((double)rand()) / (double)RAND_MAX;
}

/*****************************************************************************
Roots, Version 1.0, May 1998.
By Gregorio Malajovich, gregorio@labma.ufrj.br
Copyright (c) 1997, Gregorio Malajovich.

This  program  solves  univariate  polynomials  using  Renormalized   Graeffe
Iteration. This algorithm was developed by Jorge P. Zubelli and  myself.  See
the man page for references.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, Version 2, June 1991.

Certain files (all the fortran files) are public domain,  published  software
instead, so GNU GPL does not apply to those files.

This program is distributed in the hope that it will be useful,  but  WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public Licence for more details

You should have received a copy of the GNU General Public Licence along  with
this program; if not, write to the Free Software Foundation, Inc. , 675  Mass
Ave, Cambridge, MA 02139, USA.

*****************************************************************************/

/*****************************************************************************
This file:    random.c

Created by:   Gregorio Malajovich.
Date:         June 1, 1997.
Purpose:      Generates random complex numbers and polynomials.

Modified by:  Gregorio Malajovich
Date:         June 1997 to May 1998
Changes made: Algorithm improvements, debugging, preparation of a version
suitable for distribution. (1.0)

Modified by:  (Your name)
Date:         (Date)
Changes made: (Short description of changes)
```

```
*****************************************************************************/

/* Congruential pseudo-random number generator drand48() is used
here. In order to obtain normal (Gaussian) distribution, the
Box-Muller method is used in normal_random_complex and
normal_random_real.

Random polynomials are obtained by setting each coefficient with
normal distribution, times the correct scaling. (This is called
Kostlan probability measure, or yet U(2)-invariant probability)
Then the polynomial is normalized.

This corresponds to the usual area of a unit sphere in R^n or
C^n, with respect to the Weyl norm:

|f_i|^2
|| f ||^2  = SUM ( --------------- )
Binomial(d,i)


*/


void normal_random_complex(SAFEDOUBLE *re, SAFEDOUBLE *im)
{
       SAFEDOUBLE ang, rad;

       ang = (SAFEDOUBLE)drand48() * M_PI * 2;
       rad = (SAFEDOUBLE)sqrt(-2 * log(drand48()));

       *re = rad * cos(ang);
       *im = rad * sin(ang);

       /* Need: make the distribution of rad gaussian, avg=0 and sigma=1 */

}

void uniform_random_real(SAFEDOUBLE *re)
{
       *re = (SAFEDOUBLE)drand48();
}

void normal_random_real(SAFEDOUBLE *re)
{
       double x, y;

       x = drand48();
       y = drand48() * 2 * M_PI;

       *re = (SAFEDOUBLE)(sqrt(-2 * log(x))*cos(y));
}


void get_binomials(int degree, SAFEDOUBLE *t)
{
       int i;

       t[0] = 1;
       for (i = 0; i<degree; i++) t[i + 1] = t[i] * (degree - i) / (i + 1);

}


void random_poly(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im)
{
```

```
	int i;
	SAFEDOUBLE n;
	SAFEDOUBLE *t;

	t = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));

	/* Get binomial coefficients */

	get_binomials(degree, t);

	/* Get random coefficients */

	for (i = 0; i <= degree; i++)
	{
		normal_random_complex(re + i, im + i);
		re[i] *= sqrt(t[i]);
		im[i] *= sqrt(t[i]);
	}

	/* Normalize */

	n = 0;
	for (i = 0; i <= degree; i++)
		n += (re[i] * re[i] + im[i] * im[i]) / t[i];
	n = sqrt(n);
	for (i = 0; i <= degree; i++)
	{
		re[i] /= n;
		im[i] /= n;
	}
	free(t);
}

void real_random_poly(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im)
{
	int i;
	SAFEDOUBLE n;
	SAFEDOUBLE *t;

	t = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));

	/* Get binomial coefficients */

	get_binomials(degree, t);

	/* Get random coefficients */

	for (i = 0; i <= degree; i++)
	{
		normal_random_real(re + i);
		re[i] *= sqrt(t[i]);
		im[i] = 0;
	}

	/* Normalize */

	n = 0;
	for (i = 0; i <= degree; i++)
		n += (re[i] * re[i] + im[i] * im[i]) / t[i];
	n = sqrt(n);
	for (i = 0; i <= degree; i++)
	{
		re[i] /= n;
		im[i] /= n;
	}
	free(t);
```

```
}

/*****************************************************************************
Roots, Version 1.0, May 1998.
By Gregorio Malajovich, gregorio@labma.ufrj.br
Copyright (c) 1997, Gregorio Malajovich.

This  program  solves  univariate  polynomials  using  Renormalized   Graeffe
Iteration. This algorithm was developed by Jorge P. Zubelli and  myself.  See
the man page for references.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, Version 2, June 1991.

Certain files (all the fortran files) are public domain,  published  software
instead, so GNU GPL does not apply to those files.

This program is distributed in the hope that it will be useful,  but  WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public Licence for more details

You should have received a copy of the GNU General Public Licence along  with
this program; if not, write to the Free Software Foundation, Inc. , 675  Mass
Ave, Cambridge, MA 02139, USA.

*****************************************************************************/

/*****************************************************************************
This file:

Created by:   Gregorio Malajovich.
Date:         January 1998.
Purpose:      Conformal transforms.

Modified by:  Gregorio Malajovich
Date:         June 1997 to May 1998
Changes made: Algorithm improvements, debugging, preparation of a version
suitable for distribution. (1.0)

Modified by:  (Your name)
Date:         (Date)
Changes made: (Short description of changes)

*****************************************************************************/


/*                      cos(angle) x - sin(angle)
Below, Phi(x) = ----------------------------
sin(angle) x + cos(angle)

*/

void real_conformal(int degree, SAFEDOUBLE *zre, SAFEDOUBLE *zim,
       SAFEDOUBLE angle)
{
       SAFEDOUBLE c, s, num_re, num_im, den_re, den_im, n2;
       int i;

       if (angle == 0) return;

       c = cos(angle); s = sin(angle);

       for (i = 0; i<degree; i++)
       {
              num_re = c * zre[i] - s;
```

```
                num_im = c * zim[i];

                den_re = s * zre[i] + c;
                den_im = s * zim[i];

                n2 = den_re * den_re + den_im * den_im;

                zre[i] = (num_re * den_re + num_im * den_im) / n2;
                zim[i] = (-num_re * den_im + num_im * den_re) / n2;
        }

}


void conformal(int degree, SAFEDOUBLE *zre, SAFEDOUBLE *zim,
        SAFEDOUBLE angle[3])
{
        SAFEDOUBLE c, s, num_re, num_im, den_re, den_im, n2;
        int i;

        c = cos(angle[0]); s = sin(angle[0]);
        for (i = 0; i<degree; i++)
        {
                num_re = zre[i] * c - zim[i] * s;
                num_im = zre[i] * s + zim[i] * c;
                zre[i] = num_re;
                zim[i] = num_im;
        }

        c = cos(angle[1]); s = sin(angle[1]);

        for (i = 0; i<degree; i++)
        {
                num_re = c * zre[i] - s;
                num_im = c * zim[i];

                den_re = s * zre[i] + c;
                den_im = s * zim[i];

                n2 = den_re * den_re + den_im * den_im;

                zre[i] = (num_re * den_re + num_im * den_im) / n2;
                zim[i] = (-num_re * den_im + num_im * den_re) / n2;
        }

        c = cos(angle[2]); s = sin(angle[2]);
        for (i = 0; i<degree; i++)
        {
                num_re = zre[i] * c - zim[i] * s;
                num_im = zre[i] * s + zim[i] * c;
                zre[i] = num_re;
                zim[i] = num_im;
        }
}



void real_multiply_1(int degree, SAFEDOUBLE *f, SAFEDOUBLE a, SAFEDOUBLE b)
{ /* f gets multiplied by (ax + b) ; hope there is room for that ! */

        int i;


        f[degree + 1] = 0;
        for (i = degree + 1; i >= 1; i--)
        {
```

```
                f[i] = (f[i - 1]) * a + f[i] * b;
        }
        f[0] = f[0] * b;

}

void multiply_1(int degree, SAFEDOUBLE *fre, SAFEDOUBLE *fim, SAFEDOUBLE a,
SAFEDOUBLE b)
{ /* f gets multiplied by (ax + b) ; hope there is room for that ! */

        int i;

        fre[degree + 1] = 0;
        fim[degree + 1] = 0;
        for (i = degree + 1; i >= 1; i--)
        {
                fre[i] = (fre[i - 1]) * a + fre[i] * b;
                fim[i] = (fim[i - 1]) * a + fim[i] * b;
        }
        fre[0] = fre[0] * b;
        fim[0] = fim[0] * b;

}




void real_horner_transform(int degree, SAFEDOUBLE *f, SAFEDOUBLE a[4])
{
        SAFEDOUBLE *p, *q;
        int i, j;

        p = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
        q = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));

        for (i = 1; i <= degree; i++) p[i] = 0;
        for (i = 1; i <= degree; i++) q[i] = 0;

        p[0] = f[degree];
        q[0] = 1;

        for (i = degree; i >= 1; i--)
        {
                real_multiply_1(degree - i, q, a[2], a[3]);
                real_multiply_1(degree - i, p, a[0], a[1]);
                for (j = 0; j <= degree - i + 1; j++) p[j] += f[i - 1] * q[j];
        }

        for (i = 0; i <= degree; i++) f[i] = p[i];
        free(q);
        free(p);
}

void horner_transform(int degree, SAFEDOUBLE *fre, SAFEDOUBLE *fim, SAFEDOUBLE
a[4])
{
        SAFEDOUBLE *pre, *qre;
        SAFEDOUBLE *pim, *qim;
        int i, j;

        pre = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
        qre = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
        pim = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
        qim = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));

        for (i = 1; i <= degree; i++) pre[i] = 0;
```

```
	for (i = 1; i <= degree; i++) pim[i] = 0;
	for (i = 1; i <= degree; i++) qre[i] = 0;
	for (i = 1; i <= degree; i++) qim[i] = 0;

	pre[0] = fre[degree];
	pim[0] = fim[degree];
	qre[0] = 1;
	qim[0] = 0;

	for (i = degree; i >= 1; i--)
	{
		multiply_1(degree - i, qre, qim, a[2], a[3]);
		multiply_1(degree - i, pre, pim, a[0], a[1]);
		for (j = 0; j <= degree - i + 1; j++) pre[j] += fre[i - 1] * qre[j] -
fim[i - 1] * qim[j];
		for (j = 0; j <= degree - i + 1; j++) pim[j] += fre[i - 1] * qim[j] +
fim[i - 1] * qre[j];
	}

	for (i = 0; i <= degree; i++) fre[i] = pre[i];
	for (i = 0; i <= degree; i++) fim[i] = pim[i];
	free(qim);
	free(pim);
	free(qre);
	free(pre);
}


void pull_real_conformal
(int degree, SAFEDOUBLE *re,
	SAFEDOUBLE angle)
{
	SAFEDOUBLE a[4];

	a[0] = (SAFEDOUBLE)cos(angle);
	a[1] = (SAFEDOUBLE)-sin(angle);
	a[2] = -a[1];
	a[3] = a[0];

	real_horner_transform(degree, re, a);
}

void pull_conformal
(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im,
	SAFEDOUBLE angle[3])
{
	SAFEDOUBLE a[4];
	SAFEDOUBLE c, s, t;
	int i;



	for (i = 0; i <= degree; i++)
	{
		c = cos(i * angle[2]); s = sin(i * angle[2]);

		t = re[i] * c - im[i] * s;
		im[i] = re[i] * s + im[i] * c;
		re[i] = t;
	}




	a[0] = (SAFEDOUBLE)cos(angle[1]);
	a[1] = (SAFEDOUBLE)-sin(angle[1]);
```

```
        a[2] = -a[1];
        a[3] = a[0];

        horner_transform(degree, re, im, a);

        for (i = 0; i <= degree; i++)
        {
                c = cos(i * angle[0]); s = sin(i * angle[0]);

                t = re[i] * c - im[i] * s;
                im[i] = re[i] * s + im[i] * c;
                re[i] = t;
        }
}

/*****************************************************************************
Roots, Version 1.0, May 1998.
By Gregorio Malajovich, gregorio@labma.ufrj.br
Copyright (c) 1997, Gregorio Malajovich.

This  program  solves  univariate  polynomials  using  Renormalized   Graeffe
Iteration. This algorithm was developed by Jorge P. Zubelli and  myself.  See
the man page for references.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, Version 2, June 1991.

Certain files (all the fortran files) are public domain,  published  software
instead, so GNU GPL does not apply to those files.

This program is distributed in the hope that it will be useful,  but  WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public Licence for more details

You should have received a copy of the GNU General Public Licence along  with
this program; if not, write to the Free Software Foundation, Inc. , 675  Mass
Ave, Cambridge, MA 02139, USA.

*****************************************************************************/

/*****************************************************************************
This file:    witness.c

Created by:   Gregorio Malajovich.
Date:         June 1, 1997.
Purpose:      Newton iteration, alpha theory and certification.

Modified by:  Gregorio Malajovich
Date:         June 1997 to May 1998
Changes made: Algorithm improvements, debugging, preparation of a version
suitable for distribution. (1.0)

Modified by:  (Your name)
Date:         (Date)
Changes made: (Short description of changes)

*****************************************************************************/


SAFEDOUBLE real_eval_eta(int degree, SAFEDOUBLE *re,
        SAFEDOUBLE zre, SAFEDOUBLE zim,
        SAFEDOUBLE *mu, SAFEDOUBLE *alpha,
        SAFEDOUBLE *newzre, SAFEDOUBLE *newzim)
        /* We estimate here:
        | f(z) |
```

```
        eta(f,x) =   -------------------
        d || (z,1) ||^d

        If one assumes ||f||=1 in the U(2) invariant norm,
        the above value is the "right" notion of the value
        of the polynomial.

        Also, invariants mu and alpha are estimated.

        Optionally, newzre and newzim receive the iterate of
        zre,zim
        */


{
        SAFEDOUBLE wre, wim, wre_z, wim_z, wre_w, wim_w, t, n, m, r, df_dz, df_dw;
        SAFEDOUBLE norm, denominator, deltaz_re, deltaz_im, deltaw_re, deltaw_im;

        int i;



        if ((zre == GRINFINITY && zim == 0.0))
        {

                *newzre = zre;
                *newzim = zim;
                *alpha = 0.0;
                *mu = SAFENAN;
                return 0.0;
        }

        /* n is the norm of (z,1).
        */

        n = hypot(hypot(zre, zim), 1.0l); // HVE 2015 June 20. change 1.0 to 1.0l



        wre = re[degree];
        wim = 0;
        m = 1.0;

        wre_z = re[degree] * degree;
        wim_z = 0;

        wre_w = re[degree - 1];
        wim_w = 0;

        for (i = degree - 1; i >= 0; i--)
        {
                m *= n;

                t = (wre * zre - wim * zim) / n;
                wim = (wre * zim + wim * zre) / n;
                wre = t;
                wre += re[i] / m;

                if (i != 0)
                {
                        t = (wre_z * zre - wim_z * zim) / n;
                        wim_z = (wre_z * zim + wim_z * zre) / n;
                        wre_z = t;
                        wre_z += re[i] / m *i;
                }
```

```
                if (i != 0)
                {
                        t = (wre_w * zre - wim_w * zim) / n;
                        wim_w = (wre_w * zim + wim_w * zre) / n;
                        wre_w = t;
                        wre_w += re[i - 1] / m * (degree - i + 1);
                }
        }

        r = hypot(wre, wim) / degree;
        df_dz = hypot(wre_z, wim_z);
        df_dw = hypot(wre_w, wim_w);




        norm = (df_dz * df_dz + df_dw * df_dw) / n;

        deltaz_re = (wre * wre_z + wim * wim_z) / norm;
        deltaz_im = (-wre * wim_z + wim * wre_z) / norm;

        deltaw_re = (wre * wre_w + wim * wim_w) / norm;
        deltaw_im = (-wre * wim_w + wim * wre_w) / norm;

        denominator = (1 - deltaw_re) * (1 - deltaw_re) + deltaw_im * deltaw_im;

        *newzre = ((zre - deltaz_re)  * (1 - deltaw_re)
                + (zim - deltaz_im) * (-deltaw_im)) / denominator;

        *newzim = ((zre - deltaz_re)  * deltaw_im
                + (zim - deltaz_im) * (1 - deltaw_re)) / denominator;

        *mu = sqrt((double)degree) / hypot(df_dz, df_dw);
        if (*mu < 1) *mu = 1;

        *alpha = r * *mu * *mu * pow(degree, 1.5) / 2;

        return r;
}


SAFEDOUBLE eval_eta(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im,
        SAFEDOUBLE zre, SAFEDOUBLE zim,
        SAFEDOUBLE *mu, SAFEDOUBLE *alpha,
        SAFEDOUBLE *newzre, SAFEDOUBLE *newzim)
        /* We estimate here:
        | f(z) |
        eta(f,x) =   -------------------
        d || (z,1) ||^d

        If one assumes ||f||=1 in the U(2) invariant norm,
        the above value is the "right" notion of the value
        of the polynomial.

        Also, invariants mu and alpha are estimated.

        Optionally, newzre and newzim receive the iterate of
        zre,zim
        */


{
        SAFEDOUBLE wre, wim, wre_z, wim_z, wre_w, wim_w, t, n, m, r, df_dz, df_dw;
        SAFEDOUBLE norm, denominator, deltaz_re, deltaz_im, deltaw_re, deltaw_im;

        int i;
```

```
	if (REALFLAG)
	{
		return real_eval_eta(degree, re,
			zre, zim,
			mu, alpha,
			newzre, newzim);
	}
	if ((zre == GRINFINITY && zim == 0.0))
	{

		*newzre = zre;
		*newzim = zim;
		*alpha = 0.0;
		*mu = SAFENAN;
		return 0.0;
	}

	/* n is the norm of (z,1).
	*/

	n = hypot(hypot(zre, zim), 1.0l);  // HVE 2015 June 20. change 1.0 to 1.0l



	wre = re[degree];
	wim = im[degree];
	m = 1.0;

	wre_z = re[degree] * degree;
	wim_z = im[degree] * degree;

	wre_w = re[degree - 1];
	wim_w = im[degree - 1];

	for (i = degree - 1; i >= 0; i--)
	{
		m *= n;

		t = (wre * zre - wim * zim) / n;
		wim = (wre * zim + wim * zre) / n;
		wre = t;
		wre += re[i] / m;
		wim += im[i] / m;

		if (i != 0)
		{
			t = (wre_z * zre - wim_z * zim) / n;
			wim_z = (wre_z * zim + wim_z * zre) / n;
			wre_z = t;
			wre_z += re[i] / m *i;
			wim_z += im[i] / m *i;
		}

		if (i != 0)
		{
			t = (wre_w * zre - wim_w * zim) / n;
			wim_w = (wre_w * zim + wim_w * zre) / n;
			wre_w = t;
			wre_w += re[i - 1] / m * (degree - i + 1);
			wim_w += im[i - 1] / m * (degree - i + 1);
		}
	}

	r = hypot(wre, wim) / degree;
	df_dz = hypot(wre_z, wim_z);
```

```
        df_dw = hypot(wre_w, wim_w);




        norm = (df_dz * df_dz + df_dw * df_dw) / n;

        deltaz_re = (wre * wre_z + wim * wim_z) / norm;
        deltaz_im = (-wre * wim_z + wim * wre_z) / norm;

        deltaw_re = (wre * wre_w + wim * wim_w) / norm;
        deltaw_im = (-wre * wim_w + wim * wre_w) / norm;

        denominator = (1 - deltaw_re) * (1 - deltaw_re) + deltaw_im * deltaw_im;

        *newzre = ((zre - deltaz_re)  * (1 - deltaw_re)
                + (zim - deltaz_im) * (-deltaw_im)) / denominator;

        *newzim = ((zre - deltaz_re)  * deltaw_im
                + (zim - deltaz_im) * (1 - deltaw_re)) / denominator;

        *mu = sqrt((double)degree) / hypot(df_dz, df_dw);
        if (*mu < 1) *mu = 1;

        *alpha = r * *mu * *mu * pow(degree, 1.5) / 2;

        return r;
}

int compreals(SAFEDOUBLE r1, SAFEDOUBLE r2)
{
        SAFEDOUBLE tol = 1000 * SAFEMACHEPS;
        SAFEDOUBLE m = fabs(r1) + fabs(r2);

        if (isinf(m)) m = 0;

        if (r1 < r2 - m * tol) return -1;
        if (r1 > r2 + m * tol) return  1;
        return 0;
}

int comproots(const void *p1, const void *p2)
{
        SAFEDOUBLE *r1 = (SAFEDOUBLE *)p1;
        SAFEDOUBLE *r2 = (SAFEDOUBLE *)p2;
        int res;

        res = compreals(r1[0] * r1[0] + r1[1] * r1[1],
                r2[0] * r2[0] + r2[1] * r2[1]);
        if (res != 0) return res;

        res = compreals(r1[0], r2[0]);
        if (res != 0) return res;

        res = compreals(fabs(r1[1]), fabs(r2[1]));
        if (res != 0) return res;

        res = compreals(r1[1], r2[1]);
        return res;

}


void reorder(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im, int *multiplicity)
{
        SAFEDOUBLE *roots;
```

```
	int i;

	roots = (SAFEDOUBLE *)malloc((3 * degree) * sizeof(SAFEDOUBLE));
	for (i = 0; i<degree; i++)
	{
		roots[3 * i] = re[i];
		roots[3 * i + 1] = im[i];
		roots[3 * i + 2] = multiplicity[i];
	}

	qsort(roots, degree, 3 * sizeof(SAFEDOUBLE), comproots);

	for (i = 0; i<degree; i++)
	{
		re[i] = roots[3 * i];
		im[i] = roots[3 * i + 1];
		multiplicity[i] = (int)roots[3 * i + 2];
	}
	free(roots);
}


SAFEDOUBLE norm(int d, SAFEDOUBLE *re, SAFEDOUBLE *im)

/*    Computes the norm of a polynomial, given by

||f||^2 = sum ( |f_i|^2 / binomial(d,i) )

This norm is invariant under right action of
O(n) (resp. U(n)).

See: Hermann Weyl, The Theory of Groups and Quantum
Mechanics, Dover, 1950.
*/


{
	SAFEDOUBLE n = 0;
	SAFEDOUBLE binomial, a;
	int i;

	binomial = 1;
	for (i = 0; i <= d; i++)
	{
		if (REALFLAG) a = re[i] * re[i];
		else	a = re[i] * re[i] + im[i] * im[i];

		n += a / binomial;

		binomial *= (((SAFEDOUBLE)d) - i) / (i + 1.0);
	}

	return sqrt(n);
}

void normalize(int d, SAFEDOUBLE *re, SAFEDOUBLE *im)
{
	SAFEDOUBLE n;
	int i;

	n = norm(d, re, im);
	for (i = 0; i <= d; i++) re[i] /= n;
	if (!REALFLAG)
		for (i = 0; i <= d; i++) im[i] /= n;
}
```

```
/****************************************************************************
Roots, Version 1.0, May 1998.
By Gregorio Malajovich, gregorio@labma.ufrj.br
Copyright (c) 1997, Gregorio Malajovich.

This  program  solves  univariate  polynomials  using  Renormalized  Graeffe
Iteration. This algorithm was developed by Jorge P. Zubelli and  myself.  See
the man page for references.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, Version 2, June 1991.

Certain files (all the fortran files) are public domain,  published  software
instead, so GNU GPL does not apply to those files.

This program is distributed in the hope that it will be useful,  but  WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public Licence for more details

You should have received a copy of the GNU General Public Licence along  with
this program; if not, write to the Free Software Foundation, Inc. , 675  Mass
Ave, Cambridge, MA 02139, USA.

****************************************************************************/

/****************************************************************************
This file:    renorm.c

Created by:   Gregorio Malajovich.
Date:         June 1, 1997.
Purpose:      Renormalized Graeffe Iteration.

Modified by:  Gregorio Malajovich
Date:         October 1997
Changes made: Algorithm for factors of roots of same modulus.

Modified by:  Gregorio Malajovich
Date:         June 1997 to May 1998
Changes made: Algorithm improvements, debugging, preparation of a version
suitable for distribution. (1.0)

Modified by:  (Your name)
Date:         (Date)
Changes made: (Short description of changes)

****************************************************************************/

#ifdef TESTING
static int long COUNT = 0;
static int long TRCOUNT = 0;
#endif

void u2r(int degree, int renorm, struct poly *r,
       SAFEDOUBLE *re, SAFEDOUBLE *im)

{
       SAFEDOUBLE pow_of_2;
       int i;

       r->degree = degree;
       r->renorm = renorm;
       pow_of_2 = pow(2.0, renorm);

       if (REALFLAG)
              for (i = 0; i <= r->degree; i++)
```

```
		{
			r->log[i] = (MYDOUBLE)log(fabs(re[i])) / pow_of_2;
			r->sgn[i] = signed_sgn(re[i]);
		}
	else
		for (i = 0; i <= r->degree; i++)
		{
			r->log[i] = (MYDOUBLE)log(hypot(re[i], im[i])) / pow_of_2;
			r->arg[i] = (MYDOUBLE)atan2(im[i], re[i]);
		}

}


void u2er(int degree, int renorm, struct poly *r, struct poly *r_dot,
	SAFEDOUBLE *re, SAFEDOUBLE *im)

{
	SAFEDOUBLE *re_dot, *im_dot;
	int i;

	re_dot = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
	im_dot = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
	for (i = 0; i <= degree - 1; i++)
	{
		re_dot[i] = re[i + 1] * (i + 1);
		if (!REALFLAG) im_dot[i] = im[i + 1] * (i + 1);
	}

	re_dot[degree] = 0;
	if (!REALFLAG) im_dot[degree] = 0;

	u2r(degree, renorm, r, re, im);
	u2r(degree, renorm, r_dot, re_dot, im_dot);
	free(im_dot);
	free(re_dot);
}


//inline
void real_rensum(MYDOUBLE a, byte alpha,
	MYDOUBLE b, byte beta,
	MYDOUBLE *c, byte *gamma,
	MYDOUBLE pow_of_2)
{
	register MYDOUBLE t, diff = a - b;
	/* Important: diff can be a real, -inf, +inf or NaN */

	if (diff >= 0   /*a>=b*/)
	{
		/* Below, the bound 1+MYDSIGNIF) / pow_of_2 * log(2) would be
sharper,
		but more expensive to compute */

		if (diff <= (1 + MYDSIGNIF) / pow_of_2)
		{
			if ((beta^alpha) == 1)
				t = 1 - exp(-pow_of_2 * diff);
			else
				t = 1 + exp(-pow_of_2 * diff);
			*c = a + log(t) / pow_of_2;
			*gamma = alpha;
			return;
		}
		else
		{			/* a finite, b=-inf comes here */
```

```
                        *c = a;
                        *gamma = alpha;
                        return;
                }

        }
        else /* real_rensum(b,beta,a,alpha,c,gamma) ; */
        {

                if ((-diff) <= (1 + MYDSIGNIF) / pow_of_2)
                {
                        if ((beta^alpha) == 1)
                                t = 1 - exp(pow_of_2 * diff);
                        else
                                t = 1 + exp(pow_of_2 * diff);
                        *c = b + log(t) / pow_of_2;
                        *gamma = beta;
                        return;
                }

                else
                { /* a = inf, b finite comes here ; also, in case a = b = -inf,
                  diff is NaN and hence all previous comparisons failed,
                  so we are in the line below */
                        *c = b;
                        *gamma = beta;
                        return;
                }
        }

}

//inline
void rensum(MYDOUBLE a, MYDOUBLE alpha, MYDOUBLE b, MYDOUBLE beta,
        MYDOUBLE *c, MYDOUBLE *gamma, MYDOUBLE pow_of_2)
{
        register MYDOUBLE s1, s2, t, diff = a - b;

        if (diff >= 0)
        {

                if (diff <= 1 + MYDSIGNIF / pow_of_2)
                {
                        t = exp(-pow_of_2 * diff);
                        s1 = 1 + t * cos(beta - alpha);
                        s2 = t * sin(beta - alpha);
                        *c = a + log(hypot(s1, s2)) / pow_of_2;
                        *gamma = alpha + atan2(s2, s1);
                }
                else
                {
                        *c = a;
                        *gamma = alpha;
                }


        }
        else /* rensum(renorm,b,beta,a,alpha,c,gamma) ; */
        {

                if (-diff <= (1 + MYDSIGNIF) / pow_of_2)
                {
                        t = exp(pow_of_2 * diff);
                        s1 = 1 + t * cos(alpha - beta);
                        s2 = t * sin(alpha - beta);
                        *c = b + log(hypot(s1, s2)) / pow_of_2;
```

```
                        *gamma = beta + atan2(s2, s1);
                }
                else
                {
                        *c = b;
                        *gamma = beta;
                }

        }

}


void real_rengra(struct poly *f, struct poly *f_dot,
        struct poly *h, struct poly *h_dot, int *farthest)

{
        register MYDOUBLE t, diff;
        register MYDOUBLE *a, b, *b1, *b2;
        register byte *alpha, beta, *beta1, *beta2;
        int degree, renorm, i, j, max;
        int nontriv1 = 0, nontriv2 = 0, nontriv3 = 0;
        MYDOUBLE twice, pow_of_2;
        byte minus, sign;
        int DISCARD;
        MYDOUBLE mindiff;

        renorm = h->renorm = f->renorm + 1;
        degree = h->degree = f->degree;
        pow_of_2 = pow(2.0, renorm);
        twice = log((double)2.0) / pow_of_2;


        if (CHEAT) DISCARD = 10; /*This reduces the precision of renormalized
                                                        sums by the corresponding factor
of 2 */
        else    DISCARD = 0;


        mindiff = (-DISCARD + MYDSIGNIF + 1) / pow_of_2 *log((double)2.0);

        /*    This would guarantee a precision of
        around 2^DISCARD * MACHEPS to the trivial rensums.
        However, this may introduce some numerical instability !
        That is why the no-cheat option.
        */
        minus = degree & 1;
        /* negative = 1, positive = 0, multiplication = xor (^) */

        for (i = 0; i <= degree; i++, minus = !minus)
        {
                h->log[i] = f->log[i];
                h->sgn[i] = minus;

                h_dot->log[i] = (f->log[i] + f_dot->log[i]) / 2 + twice;
                h_dot->sgn[i] = f->sgn[i] ^ f_dot->sgn[i] ^ minus;

                max = degree - i;
                if (i < max) max = i;

                if (max > farthest[i] && CHEAT) max = farthest[i];

                sign = !minus;

                a = h->log + i;
                alpha = h->sgn + i;
```

```
		b1 = f->log + i;
		beta1 = f->sgn + i;
		b2 = f->log + i;
		beta2 = f->sgn + i;

		for (j = 1; j <= max; j++, sign = !sign)
		{
			b1++; b2--; beta1++; beta2--;
			/* clog = (f->log[i+j] + f->log[i-j]) / 2 + twice; */
			b = (*b1 + *b2) / 2 + twice;
			/* csgn = f->sgn[i+j] ^ f->sgn[i-j] ^ sign ; */
			beta = *beta1 ^ *beta2 ^ sign;
#ifdef TESTING
			COUNT += 3;
#endif
			diff = *a - b;

			if (diff >= 0   /*a>=b*/)
			{
				if (diff <= mindiff)
				{
#ifdef TESTING
					TRCOUNT++;
#endif
					nontriv1 = j;
					if (beta^ *alpha)
						t = 1 - exp(-pow_of_2 * diff);
					else
						t = 1 + exp(-pow_of_2 * diff);
					*a += log(t) / pow_of_2;
				}
			}
			else /* real_rensum(b,beta,a,alpha,c,gamma) ; */
			{
				nontriv1 = j;
				if ((-diff) <= mindiff)
				{
#ifdef TESTING
					TRCOUNT++;
#endif
					if (beta^ *alpha)
						t = 1 - exp(pow_of_2 * diff);
					else
						t = 1 + exp(pow_of_2 * diff);
					*a = b + log(t) / pow_of_2;
					*alpha = beta;
				}
				else
				{
					*a = b;
					*alpha = beta;
				}
			}

		}

		a = h_dot->log + i;
		alpha = h_dot->sgn + i;
		b1 = f->log + i;
		beta1 = f->sgn + i;
		b2 = f_dot->log + i;
		beta2 = f_dot->sgn + i;

		sign = !minus;
		for (j = 1; j <= max; j++, sign = !sign)
		{
```

```
                        b1++; b2--; beta1++; beta2--;
                        /* clog = (f->log[i+j] + f_dot->log[i-j]) / 2 + twice ; */
                        b = (*b1 + *b2) / 2 + twice;
                        /* csgn = f->sgn[i+j] ^ f_dot->sgn[i-j] ^sign ;         */
                        beta = *beta1 ^ *beta2 ^ sign;

                        diff = *a - b;

                        if (diff >= 0   /*a>=b*/)
                        {
                                if (diff <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        nontriv2 = j;
                                        if (beta^ *alpha)
                                                t = 1 - exp(-pow_of_2 * diff);
                                        else
                                                t = 1 + exp(-pow_of_2 * diff);
                                        *a += log(t) / pow_of_2;
                                }
                        }
                        else /* real_rensum(b,beta,a,alpha,c,gamma) ; */
                        {
                                nontriv2 = j;
                                if ((-diff) <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        if (beta^ *alpha)
                                                t = 1 - exp(pow_of_2 * diff);
                                        else
                                                t = 1 + exp(pow_of_2 * diff);
                                        *a = b + log(t) / pow_of_2;
                                        *alpha = beta;
                                }
                                else
                                {
                                        *a = b;
                                        *alpha = beta;
                                }
                        }


                }

                a = h_dot->log + i;
                alpha = h_dot->sgn + i;
                b1 = f->log + i;
                beta1 = f->sgn + i;
                b2 = f_dot->log + i;
                beta2 = f_dot->sgn + i;
                sign = !minus;
                for (j = -1; j >= -max; j--, sign = !sign)
                {
                        b1--; b2++; beta1--; beta2++;
                        /* clog = (f->log[i+j] + f_dot->log[i-j]) / 2 + twice ; */
                        b = (*b1 + *b2) / 2 + twice;
                        /* csgn = f->sgn[i+j] ^ f_dot->sgn[i-j] ^sign ;         */
                        beta = *beta1 ^ *beta2 ^ sign;

                        diff = *a - b;

                        if (diff >= 0   /*a>=b*/)
```

```
                        {
                                if (diff <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        nontriv3 = -j;
                                        if (beta^ *alpha)
                                                t = 1 - exp(-pow_of_2 * diff);
                                        else
                                                t = 1 + exp(-pow_of_2 * diff);
                                        *a += log(t) / pow_of_2;
                                }
                        }
                        else /* real_rensum(b,beta,a,alpha,c,gamma) ; */
                        {
                                nontriv3 = -j;
                                if ((-diff) <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        if (beta^ *alpha)
                                                t = 1 - exp(pow_of_2 * diff);
                                        else
                                                t = 1 + exp(pow_of_2 * diff);
                                        *a = b + log(t) / pow_of_2;
                                        *alpha = beta;
                                }
                                else
                                {
                                        *a = b;
                                        *alpha = beta;
                                }
                        }

                } /*j*/

                if (nontriv2 > nontriv1) nontriv1 = nontriv2;
                if (nontriv3 > nontriv1) nontriv1 = nontriv3;
                farthest[i] = nontriv1;
        } /*i*/
}


void rengra(struct poly *f, struct poly *f_dot,
        struct poly *h, struct poly *h_dot, int *farthest)

{
        register MYDOUBLE s1, s2, t, diff;
        register MYDOUBLE *a, b, *b1, *b2;
        register MYDOUBLE *alpha, beta, *beta1, *beta2;
        int degree, renorm, i, j, max;
        int nontriv1 = 0, nontriv2 = 0, nontriv3 = 0;
        MYDOUBLE twice, pow_of_2;
        byte minus, sign;
        int DISCARD;
        MYDOUBLE mindiff;

        renorm = h->renorm = f->renorm + 1;
        degree = h->degree = f->degree;
        pow_of_2 = pow(2.0, renorm);
        twice = log((double)2.0) / pow_of_2;


        if (CHEAT) DISCARD = 10; /*This reduces the precision of renormalized
```

```
                                                        sums by the corresponding factor
of 2 */
        else    DISCARD = 0;

        mindiff = (-DISCARD + MYDSIGNIF + 1) / pow_of_2 *log((double)2.0);

        /*    This would guarantee a precision of
        around 2^DISCARD * MACHEPS to the trivial rensums.
        However, this may introduce some numerical instability !
        That is why the no-cheat option.
        */
        minus = degree & 1;
        /* negative = 1, positive = 0, multiplication = xor (^) */

        for (i = 0; i <= degree; i++, minus = !minus)
        {
                h->log[i] = f->log[i];
                h->arg[i] = 2 * f->arg[i] + M_PI * minus;

                h_dot->log[i] = (f->log[i] + f_dot->log[i]) / 2 + twice;
                h_dot->arg[i] = f->arg[i] + f_dot->arg[i] + minus*M_PI;

                max = degree - i;
                if (i < max) max = i;

                if (max > farthest[i] && CHEAT) max = farthest[i];

                sign = !minus;

                a = h->log + i;
                alpha = h->arg + i;
                b1 = f->log + i;
                beta1 = f->arg + i;
                b2 = f->log + i;
                beta2 = f->arg + i;

                for (j = 1; j <= max; j++, sign = !sign)
                {
                        b1++; b2--; beta1++; beta2--;
                        /* clog = (f->log[i+j] + f->log[i-j]) / 2 + twice; */
                        b = (*b1 + *b2) / 2 + twice;
                        /* csgn = f->arg[i+j] f->arg[i-j] sign ; */
                        beta = *beta1 + *beta2 + sign * M_PI;
#ifdef TESTING
                        COUNT += 3;
#endif
                        diff = *a - b;

                        if (diff >= 0   /*a>=b*/)
                        {
                                if (diff <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        nontriv1 = j;
                                        t = exp(-pow_of_2 * diff);
                                        s1 = 1.0 + t * cos(beta - *alpha);
                                        s2 = t * sin(beta - *alpha);
                                        *a += log(hypot(s1, s2)) / pow_of_2;
                                        *alpha += atan2(s2, s1);
                                }
                        }
                        else /* rensum(b,beta,a,alpha,c,gamma) ; */
                        {
                                nontriv1 = j;
```

```
                                if ((-diff) <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        t = exp(pow_of_2 * diff);
                                        s1 = 1.0 + t * cos(-beta + *alpha);
                                        s2 = t * sin(-beta + *alpha);
                                        *a = b + log(hypot(s1, s2)) / pow_of_2;
                                        *alpha = beta + atan2(s2, s1);
                                }
                                else
                                {
                                        *a = b;
                                        *alpha = beta;
                                }
                        }

                }

                a = h_dot->log + i;
                alpha = h_dot->arg + i;
                b1 = f->log + i;
                beta1 = f->arg + i;
                b2 = f_dot->log + i;
                beta2 = f_dot->arg + i;

                sign = !minus;
                for (j = 1; j <= max; j++, sign = !sign)
                {
                        b1++; b2--; beta1++; beta2--;
                        /* clog = (f->log[i+j] + f_dot->log[i-j]) / 2 + twice ; */
                        b = (*b1 + *b2) / 2 + twice;
                        /* csgn = f->arg[i+j] ^ f_dot->arg[i-j] ^sign ;         */
                        beta = *beta1 + *beta2 + sign * M_PI;

                        diff = *a - b;

                        if (diff >= 0   /*a>=b*/)
                        {
                                if (diff <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        nontriv2 = j;
                                        t = exp(-pow_of_2 * diff);
                                        s1 = 1.0 + t * cos(beta - *alpha);
                                        s2 = t * sin(beta - *alpha);
                                        *a += log(hypot(s1, s2)) / pow_of_2;
                                        *alpha += atan2(s2, s1);
                                }
                        }
                        else /* rensum(b,beta,a,alpha,c,gamma) ; */
                        {
                                nontriv2 = j;
                                if ((-diff) <= mindiff)
                                {
#ifdef TESTING
                                        TRCOUNT++;
#endif
                                        t = exp(pow_of_2 * diff);
                                        s1 = 1.0 + t * cos(-beta + *alpha);
                                        s2 = t * sin(-beta + *alpha);
                                        *a = b + log(hypot(s1, s2)) / pow_of_2;
                                        *alpha = beta + atan2(s2, s1);
```

```
                        }
                        else
                        {
                                *a = b;
                                *alpha = beta;
                        }
                }


        }

        a = h_dot->log + i;
        alpha = h_dot->arg + i;
        b1 = f->log + i;
        beta1 = f->arg + i;
        b2 = f_dot->log + i;
        beta2 = f_dot->arg + i;
        sign = !minus;
        for (j = -1; j >= -max; j--, sign = !sign)
        {
                b1--; b2++; beta1--; beta2++;
                /* clog = (f->log[i+j] + f_dot->log[i-j]) / 2 + twice ; */
                b = (*b1 + *b2) / 2 + twice;
                /* csgn = f->arg[i+j] + f_dot->arg[i-j] +sign*M_PI ;
*/
                beta = *beta1 + *beta2 + sign*M_PI;

                diff = *a - b;

                if (diff >= 0   /*a>=b*/)
                {
                        if (diff <= mindiff)
                        {
#ifdef TESTING
                                TRCOUNT++;
#endif
                                nontriv3 = -j;
                                t = exp(-pow_of_2 * diff);
                                s1 = 1.0 + t * cos(beta - *alpha);
                                s2 = t * sin(beta - *alpha);
                                *a += log(hypot(s1, s2)) / pow_of_2;
                                *alpha += atan2(s2, s1);
                        }
                }
                else /* rensum(b,beta,a,alpha,c,gamma) ; */
                {
                        nontriv3 = -j;
                        if ((-diff) <= mindiff)
                        {
#ifdef TESTING
                                TRCOUNT++;
#endif
                                t = exp(pow_of_2 * diff);
                                s1 = 1.0 + t * cos(-beta + *alpha);
                                s2 = t * sin(-beta + *alpha);
                                *a = b + log(hypot(s1, s2)) / pow_of_2;
                                *alpha = beta + atan2(s2, s1);
                        }
                        else
                        {
                                *a = b;
                                *alpha = beta;
                        }
                }

        } /*j*/
```

```
		if (nontriv2 > nontriv1) nontriv1 = nontriv2;
		if (nontriv3 > nontriv1) nontriv1 = nontriv3;
		farthest[i] = nontriv1;
	} /*i*/

	for (i = 0; i <= degree; i++)
	{
		h->arg[i] = fmod(h->arg[i], (long double)(2 * M_PI));
		h_dot->arg[i] = fmod(h_dot->arg[i], (long double)(2 * M_PI));
	}
}

void old_rengra(struct poly *f, struct poly *f_dot,
	struct poly *h, struct poly *h_dot)

{
	int degree, renorm, i, j, max, minus, sign;
	MYDOUBLE twice, pow_of_2;
	MYDOUBLE clog, carg;


	renorm = h->renorm = f->renorm + 1;
	degree = h->degree = f->degree;
	pow_of_2 = pow(2.0, renorm);
	twice = log((double)2.0) / pow_of_2;

	if (_isnan((double)twice)) warning("rengra()", "NaN");



	minus = (degree & 1) ? -1 : 1;
	for (i = 0; i <= degree; i++, minus = -minus)
	{
		h->log[i] = f->log[i];
		h->arg[i] = 2 * f->arg[i];
		if (minus == -1) h->arg[i] += M_PI;

		h_dot->log[i] = (f->log[i] + f_dot->log[i]) / 2 + twice;
		h_dot->arg[i] = (f->arg[i] + f_dot->arg[i]);
		if (minus == -1) h_dot->arg[i] += M_PI;

		max = degree - i;
		if (i < max) max = i;


		sign = -minus;
		for (j = 1; j <= max; j++, sign = -sign)
		{
			clog = (f->log[i + j] + f->log[i - j]) / 2 + twice;
			carg = (f->arg[i + j] + f->arg[i - j]);
			if (sign == -1) carg += M_PI;
			rensum(h->log[i], h->arg[i], clog, carg, &(h->log[i]), &(h-
>arg[i]),
				pow_of_2);
		}

		sign = -minus;
		for (j = 1; j <= max; j++, sign = -sign)
		{
			clog = (f->log[i + j] + f_dot->log[i - j]) / 2 + twice;
			carg = f->arg[i + j] + f_dot->arg[i - j];
			if (sign == -1) carg += M_PI;
			rensum(h_dot->log[i], h_dot->arg[i], clog, carg,
				&(h_dot->log[i]), &(h_dot->arg[i]), pow_of_2);
```

```
                }
                sign = -minus;
                for (j = -1; j >= -max; j--, sign = -sign)
                {
                        clog = (f->log[i + j] + f_dot->log[i - j]) / 2 + twice;
                        carg = f->arg[i + j] + f_dot->arg[i - j];
                        if (sign == -1) carg += M_PI;
                        rensum(h_dot->log[i], h_dot->arg[i], clog, carg,
                                &(h_dot->log[i]), &(h_dot->arg[i]), pow_of_2);
                }




        }


        for (i = 0; i <= degree; i++)
        {
                h->arg[i] = fmod(h->arg[i], (long double)(2 * M_PI));
                h_dot->arg[i] = fmod(h_dot->arg[i], (long double)(2 * M_PI));
        }

}


MYDOUBLE dist(struct poly *f, struct poly *g, int *subf, int *subg)
{ /* This routine computes a "distance" between f and g;
  at output subdegree[i] gets a 1 for close coordinates
  and 0 for far-away coordinates */
        int i;
        MYDOUBLE d, x;

        d = 0.0;

        for (i = 0; i <= f->degree; i++)
                if (subf[i] != subg[i]) return GRINFINITY;

        for (i = 0; i <= f->degree; i += subg[i])
        {
                x = fabs(f->log[i] - g->log[i]);
                if (x > d) d = x;
        }

        return d;
}

int find_subdeg(struct poly *f, int *subdegree)
{
        int      degree = f->degree;
        int      *point;
        int      i, numpoints;
        MYDOUBLE error;      /* Security factor */
        MYDOUBLE R, two_to_d, two_to_N;
        int acceptable = 1;
        MYDOUBLE N = f->renorm;

        point = (int *)malloc((degree + 1) * sizeof(int));
        two_to_N = pow((long double)2.0, N);

        two_to_d = pow((double)2.0, degree);

        R = exp(log(MINSEP + 1) * two_to_N);

        error = (log(two_to_d * (1 + 1 / R)) / two_to_N
                - log(1 - two_to_d / R) * 2 / two_to_N
```

```
               + log(MINSEP + 1) / 4);


       if (_isnan(error) || error >= log(MINSEP + 1) / 2)
       {

               error = log(MINSEP + 1) / 2;

               acceptable = 0;
       }


       /* Add the initial point */
       point[0] = 0;
       numpoints = 1;

       for (i = 1; i <= degree; i++)
       {
               while (numpoints > 1
                      && (-f->log[point[numpoints - 1]] + f->log[point[numpoints -
2]])
                      / (point[numpoints - 1] - point[numpoints - 2])
                      >  (-f->log[i] + f->log[point[numpoints - 1]])
                      / (i - point[numpoints - 1])
                      - error) numpoints--;
               point[numpoints] = i;
               numpoints++;
       }

       for (i = 0; i< degree; i++) subdegree[i] = 0;

       for (i = 0; i<numpoints - 1; i++)
               subdegree[point[i]] = point[i + 1] - point[i];


       free(point);

       return acceptable;
}



#ifdef NEVER


void find_subdeg(struct poly *f, int *subdegree)
{
       /* This routine is intended to find the extreme points of the
       convex hull of - f->log, also called "Newton Diagram"
       (See Ostrowskii's paper).

       The bound MINSEP was fixed in order to assume a reasonable
       separation of the modulus of the roots. Therefore, points
       that are "almost inside" the convex hull can be discarded.
       For instance, in case of a multiple root, the points of f
       cannot be supposed to be exactly on a line.

       The term pow(2,-f->renorm+1) * 10 is arbitrary.
       */

       int      degree = f->degree;
       int      point[degree + 1];
       int      numpoints;
       MYDOUBLE incl[degree];
       MYDOUBLE FACTOR = 10;
       MYDOUBLE pow_of_half = pow(2.0, -f->renorm);
```

```
	int	i, j, j0, j1;

	/* Add the initial point */
	point[0] = 0;
	numpoints = 1;

	for (i = 1; i <= degree; i++)
	{ /* In order to include the point i, previous points may
	  need to be removed. We seek the largest 0<=j<=numpoints such that
	  ( -f->log[i] + f->log[point[j]] ) / (i-point[j]) > incl[j-1] + tolerance

	  All points after j will be removed.

	  We will use bissection, so this algorithm takes time
	  O(n log(n) )
	  */

	  /* Find j */
		j0 = 0;
		j1 = numpoints;
		while (j0 < j1 - 1)
		{
			j = (j0 + j1) / 2;
			if ((-f->log[i] + f->log[point[j]]) / (i - point[j]) > incl[j
- 1]
				+ (i - point[j]) * (MINSEP + pow_of_half * FACTOR))
			{ /* Increase j */
				j0 = j;
			}
			else
			{ /* Decrease j */
				j1 = j;
			}
		}
		j = j0;

		/* Remove extra points */
		numpoints = j + 2;
		point[j + 1] = i;
		incl[j] = (-f->log[i] + f->log[point[j]]) / (i - point[j]);
	}

	for (i = 0; i< degree; i++) subdegree[i] = 0;

	for (j = 0; j<numpoints - 1; j++)
		subdegree[point[j]] = point[j + 1] - point[j];

}

#endif

#define EXP(x)   ((isinf(x)==-1) ? 0 : exp(x))



void solvemagic2(SAFEDOUBLE *z_re, SAFEDOUBLE *z_im,
	struct poly *f, struct poly *f_dot,
	struct poly *g, struct poly *g_dot,
	int *subdegree, int info[])
{
	int degree = f->degree;
	int i, j, t, incr;
	MYDOUBLE d;
	MYDOUBLE modulus, clog, cang;
	byte csgn;
	int *farthest;
```

```
	int *subg;
	int acceptable;

#ifdef DEBUG
	rprint(stdout, f);
#endif

	farthest = (int *)malloc((degree + 1) * sizeof(int));
	subg = (int *)malloc((degree + 1) * sizeof(int));
	/* Iteration part */


	d = 1;

	for (i = 0; i <= degree; i++) subdegree[i] = 1;
	for (i = 0; i <= degree; i++) subg[i] = 1;
	for (i = 0; i <= degree; i++) farthest[i] = degree;

	for (t = 1;
		t <= MINSTEPS || (d>MINSEP / 100 && t <= MAXSTEPS);
		t++)
	{			/* Iterate */
		if (REALFLAG)
			real_rengra(f, f_dot, g, g_dot, farthest);
		else
			rengra(f, f_dot, g, g_dot, farthest);

		/* Compute Newton polytope */
		acceptable = find_subdeg(g, subg);

		/* Distance from previous iterate */
		d = dist(f, g, subdegree, subg);
		if ((!acceptable) && (!CHEAT)) d = 1;

		/* Copy the polynomial */
		copypoly(g, f);
		copypoly(g_dot, f_dot);
		for (i = 0; i <= degree; i++) subdegree[i] = subg[i];

	}

	info[1] = t - 1;
	//fprintf(err,"%3d iteration steps.\n",t-1) ;

	/* Interpretation part */

	if (REALFLAG)
	{
		for (i = 0; i<degree; i += incr)
		{
			incr = subdegree[i];
			/* Find the actual modulus of the root */
			modulus = EXP(f->log[i] / incr) / EXP(f->log[i + incr] /
incr);
			/* Find out (log f_i)'-(log g_i)' */

			real_rensum(f_dot->log[i] - f->log[i],
				f_dot->sgn[i] ^ f->sgn[i],
				f_dot->log[i + incr] - f->log[i + incr],
				!(f_dot->sgn[i + incr] ^ f->sgn[i + incr]),
				&clog, &csgn, pow(2.0, f->renorm));

			/* Scale back */
			z_re[i] = modulus * modulus * EXP(clog *pow(2.0, f->renorm))
				/ pow(2.0, f->renorm) / incr;
```

```
                        if ((incr & 1) && fabs(fabs(z_re[i]) - modulus) > modulus
*0.001)
                        {
                                warning("SolveMagic", "Loss of stability");
                                fprintf(stderr, "%e %e \n",
                                        (double)modulus, (double)z_re[i]);
                                if (z_re[i]>0) z_re[i] = modulus;
                                else        z_re[i] = -modulus;
                        }
                        if (fabs(z_re[i]) > modulus)
                        {
                                if (z_re[i]>0) z_re[i] = modulus;
                                else        z_re[i] = -modulus;
                        }

                        /* Sign */
                        if ((csgn & 1) == 0) z_re[i] = -z_re[i];
                        /* Imaginary part */
                        if ((incr & 1) == 1)
                                z_im[i] = 0;   /* Odd degree factor, root should be
real */
                        else
                        {
                                z_im[i] = sqrt(modulus * modulus - z_re[i] * z_re[i]);
                                if (_isnan(z_im[i])) z_im[i] = 0;
                                if (fabs(z_im[i] - modulus)<modulus*1e-8)
                                        warning("Solvemagic2", "Pure imaginary value
found");

                                /* Generic polynomials do not have pure
                                imaginary roots, so I placed a warning
                                here. */
                                if (CHEAT && (fabs(fabs(z_re[i]) - modulus)
<modulus*1e-8))
                                {
                                        warning("Solvemagic2", "Imaginary part forced
to zero");
                                        z_im[i] = 0;
                                }
                        }
                        /* Other roots */
                        for (j = 1; j<incr; j++)
                        {
                                z_re[i + j] = z_re[i + j - 1];
                                z_im[i + j] = -z_im[i + j - 1];
                        }
                }
        }
        else
        {
                for (i = 0; i<degree; i += incr)
                {
                        incr = subdegree[i];

                        /* Find the actual modulus of the root */
                        modulus = EXP(f->log[i] / incr) / EXP(f->log[i + incr] /
incr);

                        /* Find out (log f_i)'-(log g_i)' */

                        rensum(f_dot->log[i] - f->log[i],
                                f_dot->arg[i] - f->arg[i],
                                f_dot->log[i + incr] - f->log[i + incr],
                                f_dot->arg[i + incr] - f->arg[i + incr] + M_PI,
                                &clog, &cang, pow(2.0, f->renorm));
```

```
                        /* Find out the angle of zeta. As simple as that ! */
                        cang = -cang + M_PI;
                        z_re[i] = cos(cang) * modulus;
                        z_im[i] = sin(cang) * modulus;

                        /* Other roots */
                        for (j = 1; j<incr; j++)
                        {
                                z_re[i + j] = z_re[i + j - 1];
                                z_im[i + j] = z_im[i + j - 1];
                        }
                }
        }

        free(farthest);
        free(subg);
}




void solvemagic(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im,
        SAFEDOUBLE *z_re, SAFEDOUBLE *z_im,
        struct poly *f, struct poly *f_dot,
        struct poly *g, struct poly *g_dot,
        int *subdegree, int info[])
{
        u2er(degree, 0, f, f_dot, re, im);
        solvemagic2(z_re, z_im, f, f_dot, g, g_dot, subdegree, info);

}



int compress(int degree,
        SAFEDOUBLE *root_re, SAFEDOUBLE *root_im,
        int *multiplicity)
{
        int i, j, number;

        for (i = 0, j = 0; j<degree; i++)
        {
                root_re[i] = root_re[j];
                root_im[i] = root_im[j];
                multiplicity[i] = multiplicity[j];

                if (multiplicity[i]>0) j += multiplicity[i];
                else                j += -2 * multiplicity[i];
        }
        number = i;
        for (; i<degree; i++) multiplicity[i] = 0;

        return number;
}

int  solve(int degree, SAFEDOUBLE *re, SAFEDOUBLE *im,
        SAFEDOUBLE *root_re, SAFEDOUBLE *root_im, int *multiplicity, int info[])
{
        struct poly *f, *g, *f_dot, *g_dot;
        int *subdegree;
        int i, j, mult, mult_zero, mult_infinity, number;

        SAFEDOUBLE *old_re;
        SAFEDOUBLE *old_im;

        subdegree = (int *)malloc((degree + 1) * sizeof(int));
```

```
	old_re = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));
	old_im = (SAFEDOUBLE *)malloc((degree + 1) * sizeof(SAFEDOUBLE));

	for (mult_zero = 0;
		mult_zero <= degree && re[mult_zero] == 0 && (REALFLAG ||
im[mult_zero] == 0);
		mult_zero++);

	if (mult_zero == degree + 1)
		fatal_error("solve()", "Zero polynomial");

	for (mult_infinity = 0;
		mult_infinity <= degree && re[degree - mult_infinity] == 0
		&& (REALFLAG || im[degree - mult_infinity] == 0);
		mult_infinity++);

	if (mult_infinity == degree + 1)
		fatal_error("solve()", "Internal and unknown.");

	if (mult_zero != 0 || mult_infinity != 0)
	{
		number = solve(degree - mult_zero - mult_infinity,
			re + mult_zero, im + mult_zero,
			root_re + ((mult_zero == 0) ? 0 : 1),
			root_im + ((mult_zero == 0) ? 0 : 1),
			multiplicity + ((mult_zero == 0) ? 0 : 1), info
		);
		if (mult_zero != 0)
		{
			root_re[0] = 0;
			root_im[0] = 0;
			multiplicity[0] = mult_zero;
			number++;
		}
		if (mult_infinity != 0)
		{
			root_re[number] = GRINFINITY; // 1.0/0 ;
			root_im[number] = 0;
			multiplicity[number] = mult_infinity;
			number++;
		}
		free(old_re);
		free(old_im);
		free(subdegree);
		return number;
	}

	for (i = 0; i <= degree; i++)
		if (!(_finite(re[i]) && (REALFLAG || _finite(im[i]))))
			fatal_error("solve()", "Invalid polynomial");


	f = newpoly(degree);
	f_dot = newpoly(degree);
	g = newpoly(degree);
	g_dot = newpoly(degree);


	for (i = 0; i <= degree; i++) old_re[i] = re[i];
	if (!REALFLAG) for (i = 0; i <= degree; i++) old_im[i] = im[i];

	if (REALFLAG)
	{
		if (ANGLE != 0)
		{
```

```
                        //        warning("Solve","Starting real conformal
transform") ;
                        pull_real_conformal(degree, re, ANGLE);
                        //        warning("Solve","Conformal transform done") ;
                }
        }
        else
        {
                if (ANGLE != 0)
                {
                        //        warning("Solve","Starting complex conformal
transform") ;
                        pull_conformal(degree, re, im, CANGLES);
                        //        warning("Solve","Conformal transform done") ;
                }
        }


        /* See Lemma 7 in Malajovich and Zubelli, On the Geometry
        of Graeffe Iteration, for a justification of the line
        below */
        if (MINSEP == 0) /* The user did NOT fix minsep */
        {
                if (degree>1) MINSEP = 6 * MYPROB / (2 * degree * (degree - 1));
                else MINSEP = 1;
        }


#ifdef NEVER
        if (degree > 100)
                MINSEP = 6 * MYPROB / (2 * 100 * 99);
#endif


        /* See Theorem 2 ibid for the line below */

        MAXSTEPS = ceil((2 * log((double)degree) - log(MINSEP) - log(MYPROB)) /
log((double)2.0) + 1);
        if (isinf(MAXSTEPS) || _isnan(MAXSTEPS) || MAXSTEPS > MAXMAXSTEPS)
                MAXSTEPS = MAXMAXSTEPS;

        solvemagic(degree, re, im, root_re, root_im, f, f_dot, g, g_dot, subdegree,
info);

        for (i = 0; i<degree; i += subdegree[i])
        {

                mult = subdegree[i];
                if (REALFLAG && root_im[i] != 0) mult = -subdegree[i] / 2;
                /* Conjugate roots */

                for (j = i; j<i + subdegree[i]; j++) multiplicity[j] = mult;

        }

        number = compress(degree, root_re, root_im, multiplicity);

        if (REALFLAG)
        {
                if (ANGLE != 0)
                        real_conformal(number, root_re, root_im, ANGLE);
        }
        else
        {
                if (ANGLE != 0)
                        conformal(number, root_re, root_im, CANGLES);
```

```
       }

       for (i = 0; i <= degree; i++) re[i] = old_re[i];
       if (!REALFLAG) for (i = 0; i <= degree; i++) im[i] = old_im[i];

       reorder(number, root_re, root_im, multiplicity);


       free(g);
       free(g_dot);
       free(f);
       free(f_dot);

#ifdef TESTING
       fprintf(stderr, "Total rensums                %9ld\n", COUNT);
       fprintf(stderr, "Total transcendental rensums %9ld\n", TRCOUNT);
       fprintf(stderr, "Total trivial rensums        %9ld\n", COUNT - TRCOUNT);
#endif
       free(old_re);
       free(old_im);
       free(subdegree);

       return number;
}

///    @author Henrik Vestermark (hve@hvks.com)
///    @date  9/2/2005
///    @brief                graeffe
///    @return                      int    -
///    @param "DEGREE"       -      The degree of the polynominal
///    @param "coeff[]"      -      The polynominasl complex coefficients
///    @param "res[]" -         The roots from res[1..n]
///    @param "info[]"       -      informationa e.g. number of iterations per root
etc
///
///    @todo  Add to do things
///
/// Description:
///   This is the main call function to the greaffe iterations
///   Note that coefficients is stored reversed e.g. coeff[0]==a(n), coeff[1]=a(n-
1), ..., coeff[n]=a0
///
int graeffe( const int DEGREE, const std::complex<double> coeff[],
std::complex<double> res[], int info[] )
   {
   int error = 0;
   int i,j, k ;
   SAFEDOUBLE *root_re, *root_im;
   SAFEDOUBLE *re, *im;
   SAFEDOUBLE eta, mu, alpha, maxeta = 0,maxmu = 0, maxalpha=0, sep, maxsep;
   SAFEDOUBLE t1,t2 ;
   int cert_count = 0, inv_count = 0 ;
   int *multiplicity;
   int found ;
   long int seed=0 ;

   init_graeffe();

   if (NEWTON_ITERATES ==0) NEWTON_ITERATES=DEFAULT_NEWTON_ITERATES ;

   srand( seed );
   uniform_random_real (&ANGLE)      ; /* For the conformal transform */
   uniform_random_real (CANGLES)     ; /* For the conformal transform */
   uniform_random_real (CANGLES + 1) ; /* For the conformal transform */
   uniform_random_real (CANGLES + 2) ; /* For the conformal transform */
   ANGLE      *= MAXANGLE ;
```

```
   CANGLES[0] *= MAXANGLE ;
   CANGLES[1] *= MAXANGLE ;
   CANGLES[2] *= MAXANGLE ;

   re = new SAFEDOUBLE [ DEGREE + 1 ];
   im = new SAFEDOUBLE [DEGREE + 1 ];
   for( REALFLAG=true, i = 0; i <= DEGREE; i++ )
      {
      re[ i ] = coeff[ DEGREE - i ].real();
      im[ i ] = coeff[ DEGREE - i ].imag();
      if( im[ i ] != 0.0 ) REALFLAG = false;
      }

   root_re = new SAFEDOUBLE [ DEGREE + 1 ];
   root_im = new SAFEDOUBLE [DEGREE + 1 ];
   multiplicity = new int [ DEGREE ];

       /* First of all, ensure that the polynomial has
          norm 1 */
   normalize(DEGREE,re,im) ;

                       /* Call the appropriate algorithm */

   found = solve(DEGREE,re,im,root_re,root_im,multiplicity, info) ;

                       /* Improve the result using a few Newton iterations.
                       Additionally, estimate a few invariants */

   for (i=0 ; i<found ; i++)
      {
      if(UNSAFE==0 && abs(multiplicity[i]) == 1)
         {
         for (j=0 ; j < NEWTON_ITERATES ; j++)
            eta =
eval_eta(DEGREE,re,im,root_re[i],root_im[i],&mu,&alpha,root_re+i,root_im+i) ;
         info[ i + 1 ] += NEWTON_ITERATES;
         }
       }

                       /* Results should be printed in a fixed, special
                          ordering so we can compare them */

   reorder (found, root_re, root_im, multiplicity) ;


                       /* Output the results */
   for( k=0, i = 0; i < found; i++ )
      {
      res[ ++k ] = complex<double>( root_re[ i ], root_im[ i ] );
      if( multiplicity[i] < 0 )
         res[ ++k ] = complex<double>( root_re[ i ], -root_im[ i ] );
      for( j = abs(multiplicity[i])-1; j > 0; j-- )
         {
         res[ ++k ] = complex<double>( root_re[ i ], root_im[ i ] );
         if( multiplicity[i] < 0 )
            res[ ++k ] = complex<double>( root_re[ i ], -root_im[ i ] );
         }
      }


                       /* Zeros */
  for (i=0 ; i<found ; i++)
    {
    if (UNSAFE==0 && abs(multiplicity[i]) == 1) /* Certify the results */
      {
      eta = eval_eta(DEGREE,re,im,root_re[i],root_im[i],&mu,&alpha,&t1,&t2) ;
```

```
    if (abs(multiplicity[i])==1)
       {
       if (eta > maxeta) maxeta = eta ;
       if (mu > maxmu) maxmu = mu;
       if (alpha > maxalpha) maxalpha = alpha ;
       }
    }
  }

maxsep=MYNAN ;
for(i=0 ; i<found-1 ; i++)
  {
  sep = (hypot(root_re[i+1],root_im[i+1])
      - hypot(root_re[i],root_im[i]))
        / hypot(root_re[i],root_im[i]) ;
  if (! (sep > maxsep)) maxsep = sep ;
  if (sep < MINSEP)
     {
     inv_count ++ ;
     warning("run","Moduli are too close.") ;
     }
  }
                            /* Overall statistics */

if ((UNSAFE==0) && (!(maxsep <=0)) && (! (maxalpha > maxsep / 2))
    && maxalpha < 1e-1 )
    error = 0;
   else
    {
    if (maxsep<0)
       error = -1;
    if (fabs(maxsep)/2<maxalpha)
       error = -2;
    if (maxalpha>1e-1)
       error = -3;
    }

delete [] multiplicity;
delete [] root_im;
delete [] root_re;
delete [] re;
delete [] im;

return error;
}
```