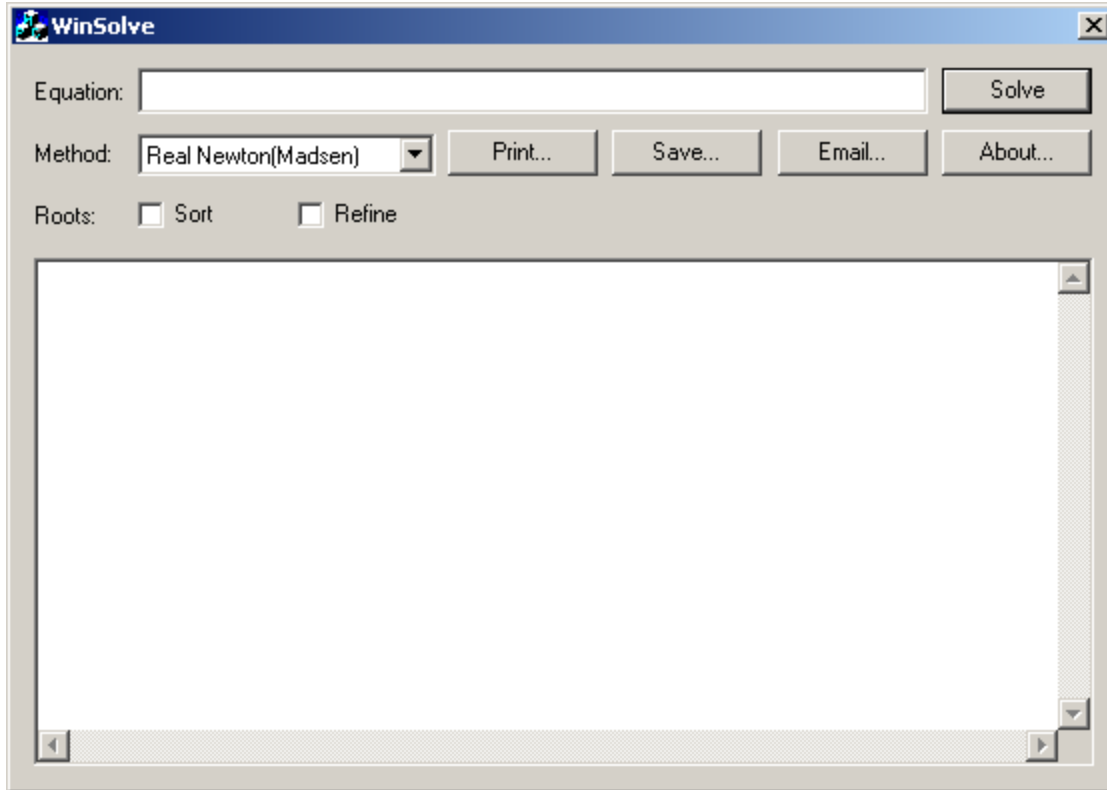


**WinSolve**  
**User's guide**  
Version 2.0

WinSolve.....	1
User's guide .....	1
Winsolve. ....	3
Input Equation:.....	3
Input Method:.....	4
Output: .....	5
Solver Methods: .....	6
Examples:.....	10
Multiple roots:.....	11
Root refinement: .....	12
The problem. ....	12
The solution. ....	13
The implementation. ....	14
Any drawbacks?.....	15
Any questions or comments:.....	15
Disclaimer: .....	15
Reference: .....	16

## WinSolve.

**WinSolve** finds all zeros (real or complex) of a polynomial of any degree with either real or complex coefficients.



### Input Equation:

The polynomial is typed in a straightforward manner. E.g.

$$\begin{aligned}2x^3+4x+7=0 \\ x^4-2x^3+3x^2-4x+5=3 \\ 1.2x^3+.25E+2x^2+2.234E-2x-1.23E3=0\end{aligned}$$

Even complex coefficient can be handled and specified with square brackets []: Fx.

$$[2+i3]x^5+[1-i2]x^3+5x^2-[-i2]x+[3+i4]=0$$

i denote the imaginary portion of a number.

The order of power is insignificant. E.g.  $2+3x^2+4x=12$  is accepted.

Also coefficients can be integer or floating point. Floating-point number following the syntax of:

$$[\text{digits}],[\text{digits}]E[+]\text{digits}$$

Digits prefixing or postfixing the "." i optional. However one must be present. E.g. ".E+1" is not legal but "1." , ".2" or "1.1" is all legal.

Also the sign of the exponent is optional so e.g. "1E+1", "1E-12", "1E3" is all legal.

Now the polynomial you type does not need to end with a "=0" it's perfectly legal to put a value on the right side of the equal sign e.g.  $x^2=12$  or  $x^3+3x=[1+i3]$ .

In some case you would like to solve polynomials that consist of multiplying several low order polynomials with each other. A regular paranthese "(" and ")" in between polynomial indicates that WinSolve will multiply the polynomial before searching for the roots. E.g.

$$(x^2+3x+2)(x^3+x+4)=0 \text{ is the same as } x^5+3x^4+3x^3+7x^2+14x+8=0$$
$$(x^2+3x+2)^2=0 \text{ is the same as } x^4+6x^3+13x^2+12x+4=0$$

Or in order to test WinSolve you can type in  $(x-1)(x-2)(x-3)(x-4)=0$  to check if the roots are correct.

Operators like '+', '-', '\*\*' are allowed fx.

$$(x^2+3x+2)*(x^3+x+4) \text{ \# polynomial multiplication}$$
$$(x^2+3x+2)-(x^3+x+4) \text{ \#polynomial subtraction}$$

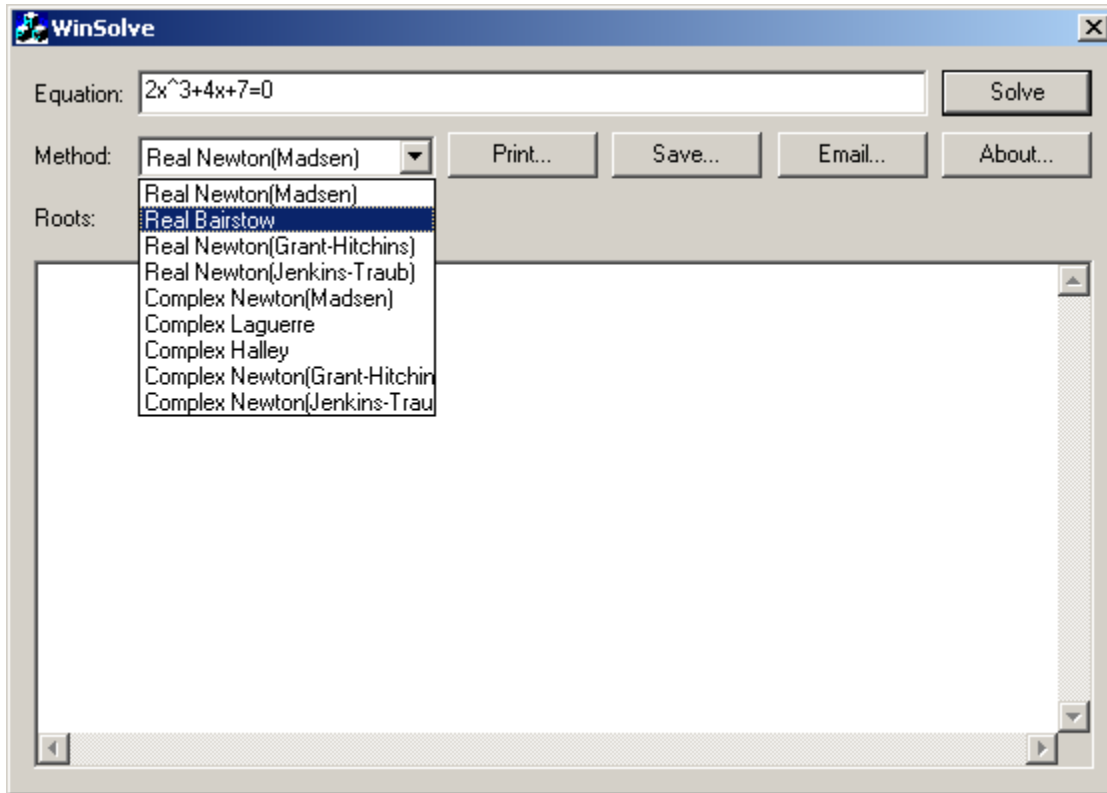
This allow input as complicated as:

$$(((((((x^2+x)^2+x)^2+x)^2+x)^2+x)^2+x)^2+x)^2+x=0$$

Winsolve check the coefficients for overflow and issue a warning if a coefficients can be represented correctly using IEEE754 standard.

## Input Method:

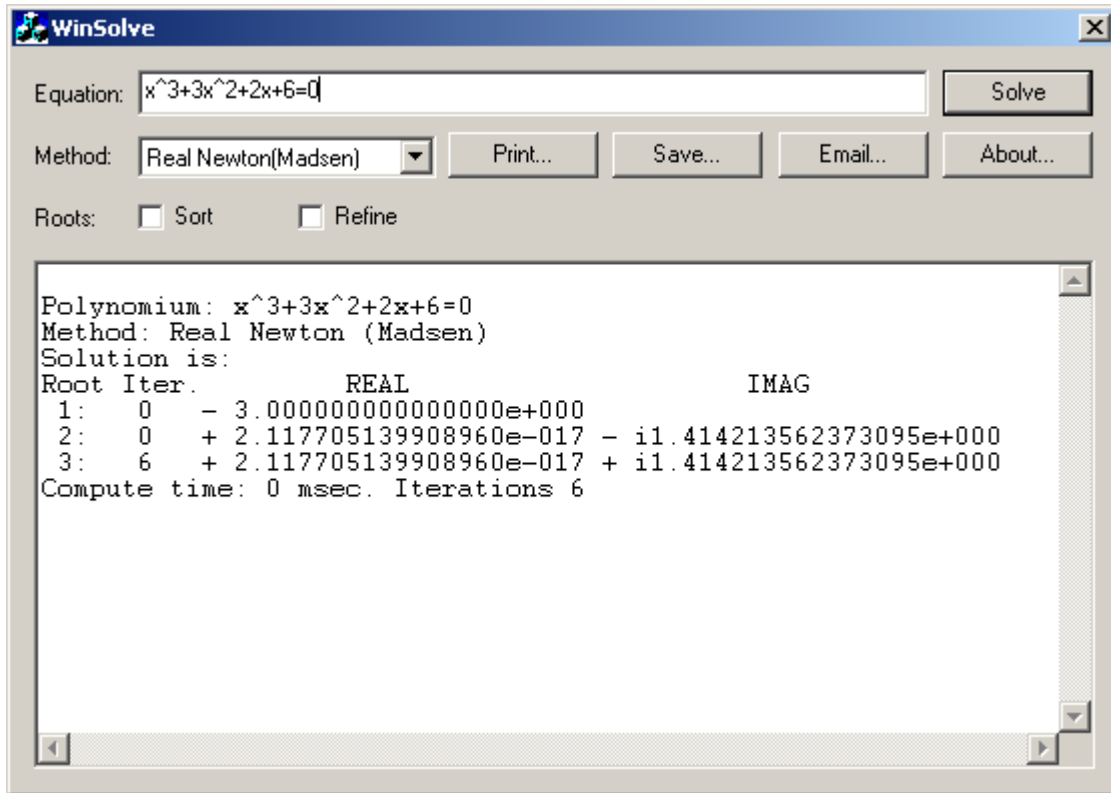
From the Method pull down menu you can select which method to use for solving the polynomial equation.



When ready hit the solve button to find the zeros of the polynomial.

## Output:

The roots will be showed in the window below and the user can now print, save or email the solution.



## Solver Methods:

WinSolve make the user choose between 11 different methods of solving this equation. The first four can only handle real coefficient:

Real Newton (Madsen), Real Bairstow, Real Bairstow (Bond), Real Newton (Grant-Hitchins) and Real Newton (Jenkins-Traub)

The remaining 5 can handle both real and complex coefficient

Complex Newton (Madsen), Complex Laguerre, Complex Halley, Complex Newton (Grant-Hitchins) and Complex(Jenkins-Traub), Complex Graeffe (Malajovich), Durand-Kerne

Bairstow and Newton has quadratic convergence meaning that for every iteration the number of significant digits will double.

Laguerre and Halley have cubic convergence meaning that the number of significant digits triple for every iteration. All methods use a modified version of the 'base' algorithm. In most cases the base method will fail to convergence unless you make modification to the algorithm to bypass numerical limitation or shortage. The modified algorithm can find new roots in a surprisingly few numbers of iteration.

**Barstow's** method is limited to real coefficients and to my knowledge has disappeared from serious numerical analysis, primarily due to its bad habit of lacking convergence. However the advantage of the method is that it always find two zeros at a time. This implementation is straight forward however with the added twist that it will calculate an error bound on the residual portion (  $Rx+S$ ) to find a stopping criteria that depends on the actual rounding errors in Bairstow's method.

**Newton's** method:

Although all the Newton's method basically is the same the three Newton methods: Madsen, Grant-Hitchins and Jenkins-Traub differ significantly in their approach. My personal favorite is the method by Madsen. [Bit 13 (1973) page 71-75] The reason is simplicity and elegance in all phases of reaching the root. As have been pointed out by Wilkinson it's important to have a stable deflations of the polynomial by either finding the root in increasing magnitude and use forward deflation or decreasing magnitude and use backward deflations. All method I have look at is trying to find the root in increasing magnitude and so do Madsen's. However instead of using the mind less zero as the starting point, Madsen instead use a starting point  $|z_0| = \frac{1}{2} \text{Min}(\sqrt[k]{|a_0/a_k|})$ ,  $k=1..n$ ; that guarantees that all roots has a higher magnitude than this starting point. The benefit is of course that the starting point is somehow close to the lowest magnitude of the root. Now Madsen define that if the root is not close to zero it's in stage 1 otherwise the process is in stage 2. Define the usual Newton step:

$$Dz_k = - f(z_k)/f'(z_k); \quad k \geq 0$$

The next step  $Z_{k+1}$  is found from  $z_k$  in the following way:

Stage1:

- A) if  $|f(z_k+d_{zk})| < |f(z_k)|$  Madsen calculate the numbers  $|f(z_k+pd_{zk})|$ ,  $p=1,2..$ ; as long as these are decreasing. i.e. Madsen stop when  $|f(z_k+(p-1)d_{zk})| \leq |f(z_k+pd_{zk})|$  and put  $Z_{k+1}=Z_k+(p-1)d_{zk}$
- B) if  $|f(z_k+d_{zk})| \geq |f(z_k)|$  the numbers  $|f(z_k+2^{-p}d_{zk})|$ ,  $p= 0,1,2..p_0$  are examined in the same way . If the sequence starts increasing we put  $Z_{k+1}=z_k+2^{-(p-1)}d_{zk}$ ; otherwise we put  $Z_{k+1}=z_k+2^{-p_0}d_{zk}e^{i\theta}$ . Here  $\pi/4 < \theta < 3\pi/4$  and  $1 \leq p_0 \leq 5$ .  $\theta = \arctan(.75)$  and  $p_0=2$

Stage 2:  $Z_{k+1}=Z_k+d_{zk}$

The Newton iteration at stage 2 is started if:

- 1) the stage 1 iteration has led to  $Z_{k+1}=Z_k+d_{zk}$
- 2) the fixed point theorem by Ostrowski ensures convergence of a Newton iteration with initial value of  $Z_{k+1}$

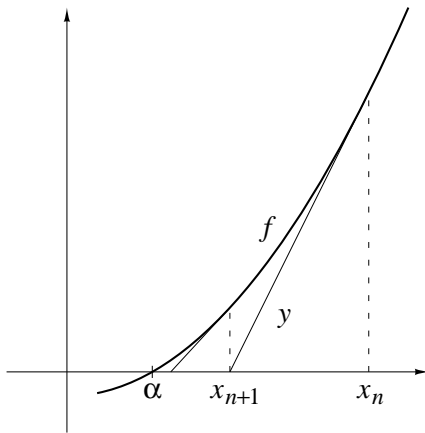
The search is stopped by either an error bound of evaluation of  $f(z_k)$  or  $d_{zk}$  is less than certain machine dependent value. For polynomial with real coefficients Adams stopping criteria is used. One of the nice part of using Madsen's variable step size is that the Newton method maintain is quadratic convergence even for multiple roots. As an example of Madsen's efficient Newton iterations consider the polynomial  $(x-1)^3(x+1)^3(x-10)(x-100)=0$ . Clearly we have a triple root at 1. The following table list the number of iterations needed for finding the 3 triple roots for each of the Newton methods:

Method	Madsen	Grant-Hitchins	Jenkins-Traub
Iterations	31	99	32

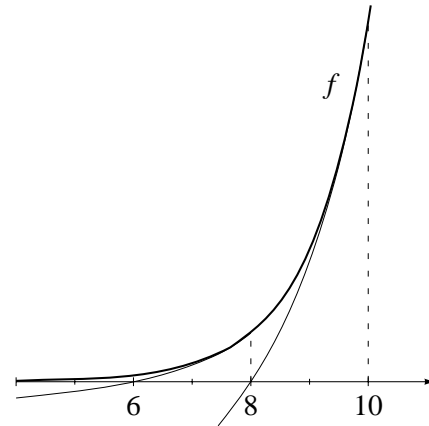
**Grant-Hitchins** [Computer Journal August 1973] is also a very solid Newton iteration although it also tries to find roots with increasing magnitude it use *composite deflation* to ensure stable deflations. Hitchins use a fixed starting point of  $0.001+i0.1$  and go from there. It also have some clever detection of possible saddle points, but instead of dealing with the problem as Madsen algorithm do it return with a failed search. Likewise if  $f'(x_k)$  is close to zero Hitchins give up the search. Again Madsen's approach is better where it just alter the directions of the search using the angle  $\arctan(.75)$  and starts a new Newton iteration. Grant-Hitchins has no special detection for handling multiple roots.

**Jenkins-Traub** algorithm. I haven't had time to study the algorithm in detail, so I will postpone my comments, but it's the most used black-box methods used in the industry today. It's also complicated and for further study I refer to the papers by Jenkins & Traub (See reference at the last page).

**Halley's** method distinguished itself from the Newton by required the calculation of the second derivate of the polynomial in order to calculate the  $d_{z_k}$  value of the next step. Hence the calculation time required per iterations is longer yet the convergence is cubic and therefore require fewer iteration than the Newton method. One way to describe the Halley's method against Newton is that a Newton step is the tangent interception of the x-axis while a Halley step is a tangent Hyperbola interception of the x-axis as depicted below.



**Figure 1** Newton Steps



**Figure 2** Halley Steps

A Halley step is giving by:

$$Dz_k = - 2f(z_k)f'(z_k)/(2f'(z_k)^2-f(z_k)f''(z_k)); k \geq 0$$

Of course Halley's method suffers the same weakness as the Newton and needs to be modified slightly to overcome the usual numeric shortcomings. Halley's method has been enhanced by adding the starting point algorithm as in the Newton Method by Madsen and the method of altering directions near saddle point and multi steps  $Dz_k$  etc as outlined by Madsen.

**Laguerre's** method shared the same need for calculation of the second derivate as with Halley's method. Laguerre has also been enhanced with the starting point and direction search change as outline by Madsen in his Newton method.

A Laguerre step is giving by:

$$Dz_k = - n / (\max(G \pm \sqrt{(n-1)(nH-G^2)})); k > 0$$

Where  $G=f'(z_k)/f(z_k)$  and  $H=(f'(z_k)/f(z_k))^2 - f''(z_k)/f(z_k)$

**Graeffe's** method was among the most popular methods for finding roots of in the 19th and 20th centuries. It was invented independently by Graeffe, Dandelin, and Lobachevsky (Householder 1959, Malajovich and Zubelli 1999). Graeffe's method has a number of drawbacks, among which are that its usual formulation leads to exponents exceeding the maximum allowed by floating-point arithmetic and also that it can map well-conditioned polynomials into ill-conditioned ones. However, these limitations are

avoided in an efficient implementation by Malajovich and Zubelli (1999), which is the method implemented here. See the two reference at the last page on this document for further reference.

**Durand-Kerner** method is a method that solves all roots simultaneously. (see [http://en.wikipedia.org/wiki/Durand-Kerner\\_method](http://en.wikipedia.org/wiki/Durand-Kerner_method)) avoiding the polynomial deflation steps of the other methods.

Two multi-precision (40 digits) version of the Madsen Newton has been added to test solving polynomials with high precision. Be aware that they take considerable longer than any of the other methods.

## Examples:

Example: This is a 10<sup>th</sup> degree polynomial:  
 $(x-1)(x-2)(x-3)(x-4)(x-5)(x-6)(x-7)(x-8)(x-9)(x-10)=0$

The number of iterations needed per method is:

Method	Iteration	Error
Laguerre	31	7.37E-11
Newton (Madsen) Real	37	1.11E-10
Newton (Madsen) Complex	38	1.45E-10
Halleys	52	3.82E-11
Bairstow	60	1.58E-11
Newton (Jenkins) Real	75	1.51E-10
Newton (Jenkins) Complex	86	1.56E-10
Newton (Hitchins) Real	91	8.07E-11
Newton (Hitchins) Complex	104	2.48E-11

Example: This is a 8<sup>th</sup> degree polynomial:  
 $(x-1)^3(x+1)^3(x-10)(x-100)=0$

The number of iterations needed per method is:

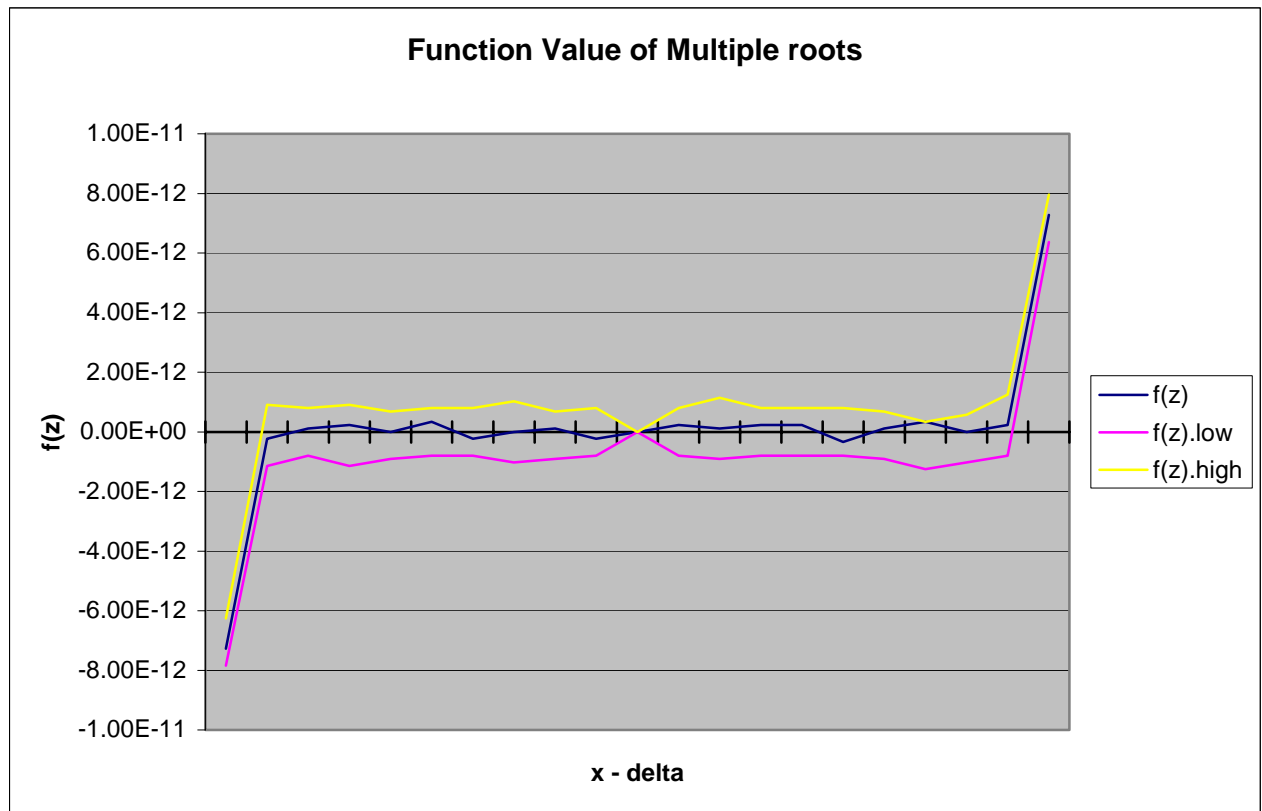
Method	Iteration	Error
Newton (Madsen) Real	31	6.75E-08
Newton (Jenkins) Real	32	3.69E-10
Newton (Madsen) Complex	42	4.38E-08
Laguerre	57	1.38E-05
Newton (Jenkins) Complex	65	5.74E-09
Halleys	74	1.23E-05
Newton (Hitchins) Real	99	2.14E-05
Newton (Hitchins) Complex	102	1.44E-05
Bairstow*)	239	6.01E-04

\*) Although the root where correct it did exceed the maximum number of iteration

The result is as expected. Halley's and Laguerre has cubic convergence while Newton and Bairstow has quadratic convergence only when dealing with multiple roots Laguerre and Halley's show some weakness. Also since we are dealing with single roots only in this example the Bairstow's algorithm has the advantages that it always finds two roots at a time. In real life, where we have complex conjugated roots, the Bairstow's method will fall far behind the other methods.

## Multiple roots:

All of the present methods face difficulties when dealing with multiple roots, as can be seen from the below example the errors on found multiple roots is significant or very imprecise. This lies in the nature of the function value of  $f(z)$  near multiple roots. In order to see how difficult it is a graph showing the function value as a function of how far away we are for the 'real' root is shown. In this example the polynomial with two triple multiple roots is used:  $(x-1)^3(x+1)^3(x-10)(x-100)=0$



The y-axis is the value of  $f(z)$ . The x-axis indicates the distance from the real root +1. Each tick mark is a power of ten away from the root starting at  $-10^{-14}$ ,  $-10^{-13}$ ,  $-10^{-12}$  going left at the graph and  $+10^{-14}$ ,  $+10^{-13}$ ,  $+10^{-12}$  going to the right of the graph. The range is between  $-10^{-5} \dots +10^{-5}$  etc. With this "low" calculated value of  $f(z)$  around  $10^{-12}$  from 0.99999..1.00001 we see that even when we are "far" away from the root the calculate value of  $f(z)$  is so low that most method would accept it as a root for the polynomial. Two other curves is also plotted  $f(z).low$  and  $f(z).high$  these curves indicate the upper and lower bound when calculating the value of  $f(z)$  using interval arithmetic. Also note that the curve  $f(z)$  several times cross the 0 value which can lead to false detection of a 'perfect' root.

## Root refinement:

In order to polish or refine already found roots, you will need to understand the limitation of standard numerical search methods and the limitation in the precision of floating point arithmetic. Standard search methods will limited the possible precision based on a number of factors.

### The problem.

- 1) First all methods implemented in WinSolve use standard IEEE754 floating point precision which limited the precision to approximate 16 decimals digits.
- 2) Secondly the stopping criteria for each root is usual some worst case bounds of the errors in evaluation of the function value  $f(x)$ . Even the famous Adam's stopping criteria for polynomial with real coefficients is an upper bound for the evaluation of  $f(x)$ . So in essence non of the methods really does take the precision to its potential limit.
- 3) When a root is found the deflation of that root yield a new polynomial with a lower degree with new coefficients that is not exact due to arithmetic errors in the deflation process and the imprecision of the deflating root. Errors from 1-3 does not usual build up to large errors when we have well isolated roots.  $Fx (x-1)(x-2)(x-3)(x-4)(x+1)(x+2)(x+3)(x+4)=0$  give 14-15 correct digits for all found roots.

Polynomial:  $x^8-30x^6+273x^4-820x^2+576=0$

Method: Real Newton (Madsen)

Solution is:

Root	Iter.	REAL	IMAG
1:	0	- 3.9999999999999998e+000	
2:	0	+ 2.9999999999999996e+000	
3:	6	+ 4.0000000000000004e+000	
4:	5	- 1.9999999999999999e+000	
5:	5	- 3.0000000000000002e+000	
6:	4	+ 1.0000000000000000e+000	
7:	5	- 1.0000000000000000e+000	
8:	5	+ 2.0000000000000000e+000	

Compute time: 0 msec. Iterations 30

- 4) When it gets to multiple roots the problem get worse. Multiple roots suffer severely from precision and as we deflate multiple roots the errors propagated down the polynomial to create more and more imprecise roots.  $Fx. (x-1)^5=0$  has one root with 16 correct digits, 2 roots with 5 correct digits one with 3 correct digits and finally one with only 2 correct digits:

Polynomial:  $x^5-5x^4+10x^3-10x^2+5x-1=0$

Method: Real Newton (Madsen)

Solution is:

Root	Iter.	REAL	IMAG
1:	0	+ 1.000199636548539e+000	
2:	0	+ 9.997950748092688e-001	
3:	0	+ 1.000002644321096e+000	- i2.023154325176408e-004
4:	13	+ 1.000002644321096e+000	+ i2.023154325176408e-004
5:	1	+ 1.0000000000000000e+000	

Compute time: 0 msec. Iterations 14

$(x-1)^8=0$  has one correct to 16 digits and everything else between 1 and 2 correct digits.

Polynomial:  $x^8-8x^7+28x^6-56x^5+70x^4-56x^3+28x^2-8x+1=0$

Method: Real Newton (Madsen)

```

Solution is:
Root Iter.          REAL                      IMAG
1:  0  + 9.983239343691635e-001 - i9.370282978126239e-003
2:  0  + 9.983239343691635e-001 + i9.370282978126239e-003
3:  0  + 1.005995707441628e+000 - i6.820254962098577e-003
4:  7  + 1.005995707441628e+000 + i6.820254962098577e-003
5:  4  + 1.008809339132710e+000
6:  0  + 9.912756886228529e-001 + i4.399484045926204e-003
7:  8  + 9.912756886228529e-001 - i4.399484045926204e-003
8:  1  + 1.000000000000000e+000
Compute time: 0 msec. Iterations 20

```

## The solution.

The solution is to use a combination of several different approaches.

First to deal with IEEE754 limited precision we would need to polish the roots using higher precision than IEEE754 approx. 16 decimal digits precision. We use the author own implementation of an arbitrary floating point precision packages. Furthermore there is a twist to using higher precision arithmetic. Instead of just selecting let's say twice the precision of the standard IEEE754 e.g. 32 decimal digits we instead are using a variable precision throughout the Newton iteration process that makes the final Newton iteration using between 24-48 decimal digits precision. The reason is that we don't want to waste calculation using higher than necessary precision, so when dealing with well isolated roots we only need 1 or 2 iterations to get the result using only 24-32 decimal precision, while when dealing with multiple root we will need a lot more. Most multiple roots I have tested need between 40-48 decimal precision before we are satisfied with the result.

Secondly we use Interval arithmetic to really *bound* the error in the evaluation of  $f(x)$  thereby having a much more precise estimate of the true rounding errors in the calculation of  $f(x)$ .

Thirdly we do all calculation using the original polynomial completely avoiding any deflation or the first, second or  $n-1$  derivate of the polynomial. However by always using the original polynomial we need to take special care that a root does not snap to another root thereby reducing the number of roots found instead of improving them!

And finally special handling of multiple roots is done to ensure maximum precision.

Now all previous three examples yields:

```

Solution after root refinement is:
Root Iter.          REAL                      IMAG
1:  39  + 1.0000000000000001e+000
2:   0  + 1.0000000000000000e+000
3:   0  + 1.0000000000000000e+000
4:  13  + 1.0000000000000000e+000
5:   1  + 1.0000000000000000e+000
Compute time: 19167 msec. Iterations 53

```

```

Solution after root refinement is:
Root Iter.          REAL                      IMAG
1:   0  + 1.0000000000000000e+000
2:   0  + 1.0000000000000000e+000
3:  34  + 1.0000000000000000e+000

```

```

4: 41 + 1.0000000000000000e+000
5: 67 + 1.0000000000000002e+000
6: 0 + 1.0000000000000000e+000
7: 8 + 1.0000000000000000e+000
8: 1 + 1.0000000000000000e+000
Compute time: 108466 msec. Iterations 151

```

```

Solution after root refinement is:
Root Iter.          REAL          IMAG
1: 2 - 4.0000000000000000e+000
2: 2 + 3.0000000000000000e+000
3: 8 + 4.0000000000000000e+000
4: 7 - 2.0000000000000000e+000
5: 7 - 3.0000000000000000e+000
6: 4 + 1.0000000000000000e+000
7: 5 - 1.0000000000000000e+000
8: 5 + 2.0000000000000000e+000
Compute time: 16404 msec. Iterations 40

```

Which is where we want it!

## The implementation.

Root polishing tries to improve the founded roots by applying the method using the original polynomial for all roots. This will avoid the errors introduce in the deflation of the polynomial however special provision has to be made to ensure that the refine algorithm does not suddenly snap to another of the found root thereby loosing a root instead of improving it. WinSolve use the following method to refine the roots found by any of the selected methods irregardless of whether the polynomial used real or complex coefficients.

- 1) For any root of zero we accept is a real root of the original polynomial and zero roots is therefore skipped throughout the refinement process.
- 2) To avoid a root to snap to another root and thereby loosing the root we first established an isolation circle around each root. This circle represents the limited of the search area and is guaranteeing to be isolated from any other root.
- 3) Each root is tried using the polynomial original coefficients with a standard Newton iteration. All roots found are typical very close to the real roots and therefore is not critical which method e.g. Newton, Laguerre or Halley to use. However the function evaluation is done using interval arithmetic that by definition always will give us a correct upper bound of the error in the calculation of  $f(x)$ . So when we can't improve the root due to calculating error we stop the improvement for that root. Also if a Newton steps goes outside the isolation circle defined for that root the Newton will returned with an indication of a possible multiple root.
- 4) Step 3 works well for isolated roots however when dealing with multiple roots the circle gets very small and therefore will not be very useful. Instead we use a special improvement for multiple root original suggested by C. Bond. The method in short is for multiple roots both the  $f(x)$  and its  $f'(x)$  has the same zero. E.g. a double root for  $f(x)$  contains only one of its double roots for its first derivate  $f'(x)$ . By differentiate a number of times you essentially strip of the multiple root until there is only one left in which case it will easy to solve and the Newton iteration will quickly *bound* the root.
- 5) Now how do we detect multiple roots? Well if a Newton iterations walks out of the circle of isolation from the other roots we are typically dealing with a multiple root problem and we use C. Bond approach to reduce the multiplicity of the root and solved that. However the Newton() function also use another criteria for determine of multiple roots. Since we are using interval arithmetic it will be easy to detect if the interval in the evaluation of  $f(x)$

and  $f'(x)$  contain zero. If it does it will mean that both  $f(x)$  and  $f'(x)$  has the same zero or is so close that you can't distinguished these roots from each other. In which case we use C. Bond suggestion to finding the multiple root.

- 6) Now when iterating using Newton method we usual start out using 24 digits decimal precision, which for well isolated roots is enough. When encountered multiple root we will need higher precision to really "*get the root*". So the Newton method has been enhanced so that after we exhaust the precision of a giving search and we still are not satisfied with the result. We simple increase the precision with 8 decimal digits and repeat the Newton iteration. This continuous until we no longer can detect a change in the root based on its standard IEEE754 representation. For most root refined we typically end up with a final round of Newton iterations between 24-40 digits precisions.
- 7) The stopping criteria for our root refinement procedure are when the result does not change the root using the standard IEEE754 representation. e.g. when two successive Newton iterations using  $p$  and  $p+8$  decimal digits precisions does not change the root as represented by the IEEE754 standard.

### **Any drawbacks?**

The refinement process is slow compare to the basis solver methods. Both the higher precision arithmetic and interval arithmetic really eats up a lot of CPU time. On a Windows 2000 PC with a 700MHz Pentium III CPU various test polynomial between the degree of 5 to 10 took about 2-10 seconds to refined while the basic solver found the initial roots in less than 10msec!

### **Any questions or comments:**

Please email me at: [hve@hvks.com](mailto:hve@hvks.com) or alternative [hve@contex.com](mailto:hve@contex.com)

### **Disclaimer:**

Permission to use, copy and distribute this software and documentation for any purpose is hereby granted without fee, provided:

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, XPRESS, IMPLIED OR THERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL Henrik Vestermark, Future Team Aps, BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

## Reference:

Kaj Madsen

"A Root-finding Algorithm based on Newton's Method"  
Bit 13 (1973) pp 71-75.

J.A.Grant & G.D.Hitching

"Two Algorithms for the solution of Polynomial Equations to Limiting Machine Precision"  
Computer Journal Volume 18 Number 3 (August 1973)

J.H.Wilkinson

"Rounding Errors in Algebraic Processes"  
Prentice-Hall, 1963

A. M. Ostrowski

"Solution of Equations and Systems of Equations"  
Academic Press, 1966

D. Adam

"A stopping criteria for polynomial root-finding"  
Comm. Acm 10 (1967) pp 655-658

C. Bond

A Method for finding Real and Complex Roots of Real Polynomials with Multiple roots  
(2002)

C. Bond

A Robust Strategy for Finding All Real and Complex Roots of Real Polynomials  
(2003)

W Press, S Teukolsky, W Vetterling, B Flannery  
Numerical recipes in C. 2<sup>nd</sup> edition

Malajovich, G. & Zubelli, J. P.

"On the Geometry of Graeffe Iteration."  
*Informes de Matemática*, Série B-118, IMPA.

Malajovich, G. & Zubelli, J. P.

"Tangent Graeffe Iteration."  
27 Aug 1999. <http://xxx.lanl.gov/abs/math.AG/9908150/>.

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

Durand-Kerner

[http://en.wikipedia.org/wiki/Durand-Kerner\\_method](http://en.wikipedia.org/wiki/Durand-Kerner_method))