

Non Linear Equations

**Practical methods for roots
finding**

Part II - Zeros of Polynomials

Leonardo Volpi

Zeros of Polynomials	3
Polynomials features	3
Synthetic division of polynomial	4
Polynomial and derivatives evaluation	6
Polynomial evaluation in limited precision	8
The underflow barrier	9
Deflation.....	11
Deflation with decimals	12
Inverse Deflation.....	13
Shifting.....	14
Roots Locating	17
Graphic Method	17
Companion Matrix Method.....	18
Circle Roots.....	19
Iterative Powers Method	21
Quotient-Difference Method.....	22
Multiple roots	27
Polynomial GCD.....	27
GCD with decimals.....	28
GCD - A practical Euclid's algorithm.....	32
GCD - the cofactor residual test.....	33
GCD - The Sylvester's matrix	34
The Derivatives method	35
Multiplicity estimation.....	36
Integer Roots finding	39
Brute Force Attack.....	39
Intervals Attack.....	41
Algorithms	42
Newton-Raphson method.....	42
Halley method.....	46
Lin-Bairstow method	48
Siljak method	53
Laguerre method	54
ADK method.....	56
Zeros of Orthogonal Polynomials.....	59
Jenkins-Traub method.....	64
QR method	67
Polynomial Test	70
Random Test	70
Wilkinson Test.....	71
Selectivity Test.....	72
Roots Finding Strategy.....	73
Attack Strategy.....	73
Examples of Polynomial Solving.....	73
Bibliography.....	77
Appendix.....	79
Complex power of the binomial $(x+iy)$	79

Zeros of Polynomials

The problem of finding the polynomial zeros is surely one of the most important topic of the numeric calculus. General speaking, a monovariable polynomial of n^{th} degree, can be written as:

$$P(z) = a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} \dots + a_2 z^2 + a_1 z + a_0 \quad (1)$$

For $P(x)$ we could adopt any zerofinding algorithm previous descript for the general real function $f(x)$. But for polynomials, usually, the roots problem needs searching on the entire complex domain. In addition they have same important features that can greatly help the root finding .

Polynomials features

- 1) A polynomials has always a number of zeros, real and complex, equal to its degree. We must count each zero as much as its multiplicity: we count one time a single zero; two times a double zero, etc.
- 2) If the coefficients are all real, then the complex zeros are always conjugate. That is, if a roots $z = a + i b$ exists, then it must exist also the root $\bar{z} = a - i b$. This reduces by half the computation effort for complex roots finding.
- 3) If the coefficients are all real , the sum of the roots is: $\sum z_i = -a_{n-1} / a_n$.
- 4) If the coefficients are all real , the product of the roots is: $\prod z_i = (-1)^n a_0 / a_n$
- 5) Found a real root z_1 is always possible to reduce the polynomial degree by the division $P(z) / (z - z_1) = P_1(z)$. The polynomial $P_1(z)$, of degree $n-1$, has exactly the same roots of $P(z)$ except the root z_1 . This method, called "deflation", allows to repeat iteratively the root finding. Deflation usually generates round-off errors and we should take in order to minimize those errors propagation.
- 6) Found a complex root $z_1 = a - i b$, if the polynomial coefficients are all real, then is always possible to reduce the polynomial degree by the division:
 $P_1(z) = P(z) / (z - a - i b)(z - a + i b)$. The polynomial $P_1(z)$, of degree $n-2$, has exactly the same roots of $P(z)$ except the complex conjugate roots z_1 e \bar{z}_1 . This method, called "complex deflation", allows to repeat iteratively the root finding. In spite the name, the complex deflation can be performed entirely with real arithmetic, being:
 $(z - a - i b)(z - a + i b) = (z^2 + 2a z + a^2 + b^2)$
- 7) If the coefficients are all integer, possible integer roots must be searched in the set of exact divisors of the last coefficients a_0 . This feature allows to extract integer roots by the "exact integer deflation" that has no round-off error.
- 8) If a root has multiplicity "m", then it is also root of all polynomial derivatives till $P^{(m-1)}$
- 9) The polynomial derivatives evaluation is simple

Synthetic division of polynomial

The Ruffini-Horner method performs the division of a polynomial for the binomial $(x - a)$ in a very fast and stable way. Generally, the division returns the quotient $Q(x)$ and the numerical remainder "r" verifying:

$$P(x) = (x - a) \cdot Q(x) + r \quad (1)$$

Example

$$(2x^4 + 11x^3 - 53x^2 - 176x + 336)/(x - 1)$$

Let's build the following table

1	2	11	-53	-176	336
<hr/>					

The number 1 (at the right) is the divisor.

The polynomial coefficients are in the first row, ordered from left to right by descendent degree

1	2	11	-53	-176	336
<hr/>					
	2				

Let's begin write down the number 2 of the first column

1	2	11	-53	-176	336
<hr/>					
	2	2			
	2	13			

Now multiply 2 for the divisor 1 and write the result under the 2° coefficient in order to make the sum 13

1	2	11	-53	-176	336
<hr/>					
	2	2	13		
	2	13	-40		

Now multiply 13 for the divisor 1 and write the result under the 3° coefficient in order to make the sum -40

1	2	11	-53	-176	336
<hr/>					
	2	2	13	-40	-216
	2	13	-40	-216	120

Repeat the process till the last coefficient. The coefficient of the quotient are in the last row; the remainder fills the last cell at the right-bottom

So we have : $Q(x) = 2x^3 + 13x^2 - 40x - 216$, $r = 120$

Other examples.

$$(2x^4 + 11x^3 - 53x^2 - 176x + 336)/(x + 2)$$

$$Q = 2x^3 + 7x^2 - 67x - 42 \quad , \quad r = 420$$

-2	2	11	-53	-176	336
<hr/>					
		-4	-14	134	84
	2	7	-67	-42	420

$$(2x^4 + 11x^3 - 53x^2 - 176x + 336)/(x + 4)$$

$$Q = 2x^3 + 3x^2 - 65x + 84 \quad , \quad r = 0$$

-4	2	11	-53	-176	336
<hr/>					
		-8	-12	260	-336
	2	3	-65	84	0

Note that, in the last example, the remainder is zero. This means that the number -4 is a root of the polynomial P(x) and, thus, the binomial $(x + 4)$ is a factors of the polynomial.

The synthetic division can be coded in a very efficient way. An example is the following VB code, having cost : $3n+2$

```
Sub PolyDiv1(a(), b(), r, p)
'Divide the polyn. A(x)= a0+a1*x+a2*x^2+... for (x-p)
'      A(x) = B(x)*(x-p)+R
'A()= coeff. of A(x)  A(0)=a0, A(1)=a1, A(2)=a2 ...
'B()= coeff. of B(x)  B(0)=b0, B(1)=b1, B(2)=b2 ...
'R = remainder
  Dim n, j
  n = UBound(a)
  b(n) = 0
  For i = n - 1 To 0 Step -1
    j = i + 1
    b(i) = a(j) + p * b(j)
  Next i
  r = a(0) + p * b(0)
End Sub
```

The division of the real polynomial: $P(z) = a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + \dots + a_2 z^2 + a_1 z + a_0$ for the quadratic polynomial $(z^2 - u z - v)$, characteristic of complex conjugate roots, can be realized by the following iterative algorithm.

$$b_n = 0, \quad b_{n-1} = 0 \quad b_i = a_i + u \cdot b_{i+1} + v \cdot b_{i+2} \quad \text{for: } i = n-2 \dots 2, 1, 0, -1, -2$$

where the quotient polynomial is: $Q(z) = b_{n-2} z^{n-2} + b_{n-3} z^{n-3} \dots + b_2 z^2 + b_1 z + b_0$
and the remainder polynomial is: $R(z) = b_{-1} z + b_{-2}$

This iterative algorithm can be easily arranged on spreadsheet.

Example. Divide the polynomial: $x^6 - 13x^5 + 106x^4 - 529x^3 + 863x^2 + 744x + 1378$
for the polynomial: $x^2 - 4x + 53$

		a_6	a_5	a_4	a_3	a_2	a_1	a_0
		1	-13	106	-529	863	744	1378
v	u	0	4	-36	68	64	104	0
-53	4	0	0	-53	477	-901	-848	-1378
		1	-9	17	16	26	0	0
		b_4	b_3	b_2	b_1	b_0	b_{-1}	b_{-2}

The cells of the 2nd row, compute the term " $u \cdot b_{i+1}$ ", that is the product of "u" for the value of the 4th row of the previous column.

		a_6	a_5	a_4	a_3
		1	-13	106	-529
v	u		4	1*F28	68
-53	4			0	-53
		1	-9	17	16
		b_4	b_3	b_2	b_1

The cells of the 3rd row, compute the term " $v \cdot b_{i+1}$ ", that is the product of "v" for the value of the 4th row of the previous column

		a_6	a_5	a_4	a_3
		1	-13	106	-529
v	u		4	-36	68
-53	4			0	1*E28
		1	-9	17	16
		b_4	b_3	b_2	b_1

The last 4th row is the sum of the column values.

The quadratic division algorithm can be efficiently implemented by adapt subroutines. An example is the following VB code, having cost : $6n$

```

Sub PolyDiv2(a(), b(), r1, r0, u, v)
'Divide the polyn. A(x)= a0+a1*x+a2*x^2+... for (x^2-u*x-v)
'      A(x) = B(x)*(x^2-u*x-v) + r1*x+r0
'A()= coeff. of A(x)  A(0)=a0, A(1)=a1, A(2)=a2 ...
'B()= coeff. of B(x)  B(0)=b0, B(1)=b1, B(2)=b2 ...
'r1 = 1st degree coeff. of the remainder
'r0 = constant coeff. of the remainder
  Dim n, j, i
  n = UBound(a)
  b(n) = 0
  b(n - 1) = 0
  For i = n - 2 To 0 Step -1
    j = i + 2
    b(i) = a(j) + u * b(i + 1) + v * b(j)
  Next
  r1 = a(1) + u * b(0) + v * b(1)
  r0 = a(0) + v * b(0)
End Sub

```

Polynomial and derivatives evaluation

The synthetic division is an efficient algorithm also for evaluating a polynomial and its derivatives. Remembering the relation:

$$P(x) = (x - a) \cdot Q(x) + r$$

setting $x = a$, we have

$$P(a) = (a - a) \cdot Q(a) + r \Rightarrow P(a) = r$$

Thus the remainder is the value of the polynomial at the point $x = a$. This method, called Ruffini-Horner schema, is very efficient and stable. Come back to the previous polynomial and evaluate it for $x = -2$, by both methods: the classic substitution method and the synthetic division.

Substitution method

$$\begin{aligned}
 P(x) &= 2x^4 + 11x^3 - 53x^2 - 176x + 336 \\
 P(-2) &= 2(-2)^4 + 11(-2)^3 - 53(-2)^2 - 176(-2) + 336 = \\
 &= 2 \cdot 16 + 11(-8) - 53 \cdot 4 - 176(-2) + 336 = \\
 &= 32 - 88 - 212 - 352 + 336 = 420
 \end{aligned}$$

Ruffini-Horner method

	2	11	-53	-176	336
-2		-4	-14	134	84
	2	7	-67	-42	420

It is evident the superiority of the Ruffini-Horner division method because it avoids to calculate directly the powers.

The following VB function computes a real polynomial of degree n; the cost is 2n

```
Function Poly_Eval(x, A())
'Compute the polyn. A(x)= a0+a1*x+a2*x^2+...an*x^n at the point x
Dim n, y, i
  n = UBound(A)
  y = A(n)
  For i = n - 1 To 0 Step -1
    y = A(i) + y * x
  Next
  Poly_Eval = y
End Function
```

The following VB subroutine computes the polynomial of degree n and its derivatives simultaneously

```
Sub DPolyn_Eval(x, A, D)
'compute the polynomial and its derivatives simultaneously
'D(0) = A(x) , D(1) = A'(x), D(2) = A''(x) ...
'A()= Coeff. of A(x)  A(0)=a0, A(1)=a1, A(2)=a2 ...
'D()= array of returned values
Dim n, y, i, k
  n = UBound(A)
  For k = n To 0 Step -1
    y = A(k)
    For i = 0 To k
      D(i) = D(i) * x + y
      y = y * (k - i)
    Next i
  Next k
End Sub
```

At the end, the vector D contains the value of the polynomial and its derivatives at the point x

$$D(0) = P(x), D(1) = P^{(1)}(x), D(2) = P^{(2)}(x), D(3) = P^{(3)}(x), \dots D(n) = P^{(n)}(x),$$

The computational cost is: $3(n^2+3n+2)$

Polynomial evaluation in limited precision

Let's take the polynomial of 10th degree having the roots: $x = 1$ ($m = 9$), $x = 2$ ($m = 1$), where m is the root multiplicity. This polynomial can be written and computed in three different ways

Formula

- 1) $(x-2)(x-1)^9$
- 2) $2-19x+81x^2-204x^3+336x^4-378x^5+294x^6-156x^7+54x^8-11x^9+x^{10}$
- 3) $(((((((((x-11)x+54)x-156)x+294)-378)x+336)x-204)x+81)x-19)x+2$

The first one is the factorized formula, that is the most compact form. It needs the explicit values of all polynomial roots.

The second formula is the sum of powers, that is the standard form which the polynomial is usually given. It needs the polynomial coefficients.

The last one is the Horner's form. As the previous one, it needs the polynomial coefficients but avoids the explicit power computation. The Ruffini-Horner iterative algorithm is numerically equivalent to this formula.

From the point of algebraic view, all these formulas are equivalent. But they are quite different for the numerical calculus in limited precision.

Many rootfinding algorithms evaluate the polynomial and its derivatives in order to approximate iteratively the roots. In this cases the formula (3) is used thanks its efficiency.

Now let's see the behavior of the three different formulas in a practical case.

We evaluate the given polynomial in a point very close to the exact multiple root using the standard arithmetic precision (15 significant digits).

Evaluation for $x = 1.0822 \Rightarrow P(x) = -1.5725136038295733584E-10$ ⁽¹⁾

Formula	$P(x)$ ²	Rel. Error
1)	-1.57251360383E-10	7.8904E-15
2)	-1.572 07580287 E-10	0.00027849
3)	-1.572 09356644 E-10	0.00026718

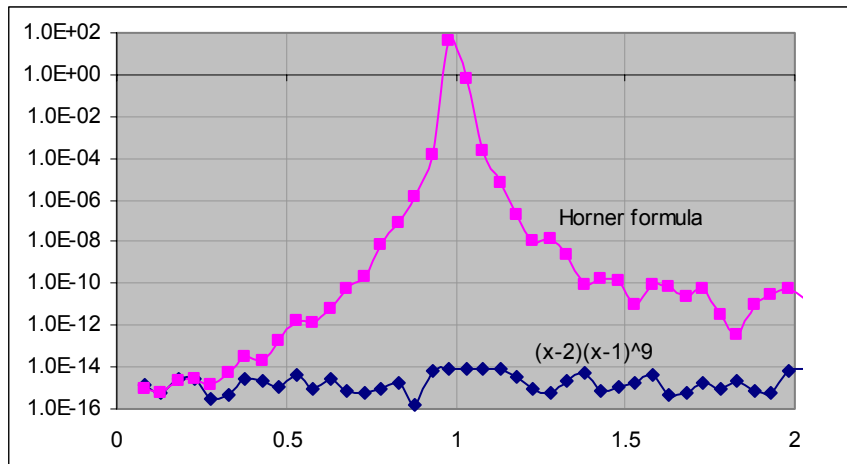
As we can see, the formulas (2) e (3) approximate the polynomial value with a relative error of about 1E-4 (0.1%), giving only 4 digits exact. The formula (1), on the contrary, gives a value practically exact (1E-14).

It's remarkable that there are more than 10 orders of accuracy between the formula (1) and the others. We can easily guess that the behavior of a rootfinder will be completely different calling the formula (1) instead of the (2) or (3). Unfortunately the factorized formula (1) cannot be used because the roots are unknown. Usually this is used by some authors - for convenience - in algorithm tests. Nevertheless, as we have shown, the response seems "too perfect" for performing a reliable test.

The following graph shows the relative error curve obtained by the factorized formula (1) and by the Horner's formula (3) in a close set around the multiple root $x = 1$

¹ Value computed with multiprecision of 30 digits

² The black digits are exact



The plot shows the difference between the polynomial evaluation obtained with the Horner's formula (3) and the factorized formula (1) performed in finite arithmetic (15 digits). The difference is remarkable just in proximity of the root. This fact shows that the rootfinder algorithm should always test with the worst conditions, thus using the formula (2) or (3). On the contrary, when exact roots are known, the factorized formula (1) is preferable.

The underflow barrier

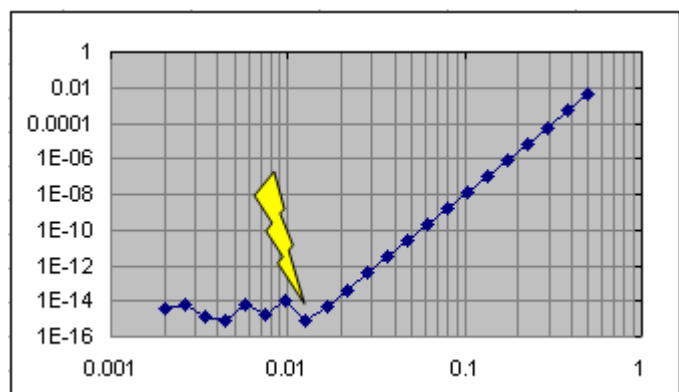
As we have shown, the polynomial evaluation using finite precision is affected by errors, that become noisier near a multiple root. Those errors limit the max accuracy of the root: bigger is the multiplicity and larger is the minimum error to which we can approximate the root. This phenomena is known as *underflow barrier* or *underflow breakdown*.

For example, assume to have the following 8th degree polynomial having the only roots $x = 1$, with multiplicity $m = 8$

$$P(x) = x^8 - 8x^7 + 28x^6 - 56x^5 + 70x^4 - 56x^3 + 28x^2 - 8x + 1$$

Using 15 digits, we evaluate the polynomial at the point $x = 1 + dx$, where "dx" is a small increment that we set progressively smaller. In this way we simulate the approaching of the rootfinder to this root.

For each point we take the absolute polynomial value $|P(x)|$, plotting the couple $(dx, |P(x)|)$ in a bi-logarithmic scale. The final plot will look like the one at the right

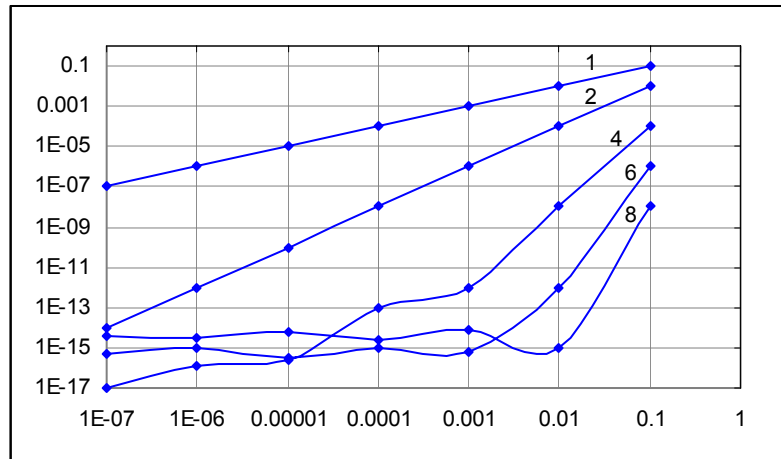


As we can see, just before the point $dx = 0.01$, the curve changes its linear trajectory and becomes random. The polynomial value is about $1E-15$ that it is near the machine limit.

The round-off errors, produced by the numerical evaluation of $P(x)$ for $dx < 0.01$, completely overcome the true value of the polynomial. In this situation no numerical algorithm can approach the root better than 0.01. It is a sort of accuracy "barrier" that we do not go any further evaluating the given polynomial.

This limit depends on several factors: mainly by the machine precision; by the root multiplicity; by the polynomial degree and, as we have seen, by the formula used in the calculation of the polynomial itself

The following graph shows several error trajectories for multiplicity $m = 1, 2, 4, 6, 8$



As we can see, for $m \leq 2$ there are no problem to reach a root accuracy better then $1E-7$.

For $m = 4$ the barrier is about $1E-5$; it shifts itself to 0.001 for $m = 6$, and to 0.01 for $m = 8$.

To overcome the barrier we have to use the multiprecision arithmetic or to adopt other methods that avoid the direct evaluation of the polynomial (see par. "Multiple Root Extraction")

Deflation

When we know a root of a polynomial, real or complex, we can extract it from the polynomial by the synthetic division obtaining a new polynomial with lower degree

Example

$$16x^6 - 58x^5 - 169x^4 + 326x^3 - 230x^2 + 384x - 45$$

Assume to have found, in any way, the real root $x = -3$ and the conjugate complex roots $x = \pm i$. Let's eliminate, at the first, the real root dividing for the linear factor $(x+3)$

-3	16	-58	-169	326	-230	384	-45
		-48	318	-447	363	-399	45
	16	-106	149	-121	133	-15	0

The conjugate complex roots $a \pm ib$ can be eliminated dividing for the quadratic factor

$$x^2 - u \cdot x - v, \text{ where } u = 2 \cdot a, \text{ } v = -(a^2 + b^2).$$

			16	-106	149	-121	133	-15
v	u			0	0	0	0	0
-1	0			0	-16	106	-133	15
			16	-106	133	-15	0	0

The reduced polynomial $16x^3 - 106x^2 + 133x - 15$ contains all the roots of the original polynomial except those eliminated. This process, called "*deflation*" or "*reduction*" is very common in the sequential root finding algorithms. When a root is found, the polynomial is reduced by deflation and the rootfinding algorithm is applied again to the reduce polynomial and so on, till the final polynomial has 1st o 2nd degree. This polynomial is finally solved by its specific formula.

Deflation with decimals

Deflation is always exact if all polynomial coefficients and all roots are integer. But, on the contrary, if at least one roots is real, or even if the root is affected by decimal errors, the deflation process propagates and, sometime, amplifies the round-off errors. The last root found can accumulate those errors and thus they can be quite different from the exact roots.

Example. $x^5 - 145x^4 + 4955x^3 - 46775x^2 + 128364x - 86400$

The exact roots are 1, 3, 9, 32, 100. Let's try to perform the deflation adding a reasonable relative error of about $1E-6$, (0.0001 %) at the first four roots. Perform the deflation in increasing order starting from the lowest root.

1.000001	1	-145	4955	-46775	128364	-86400
	0	1.000001	-144.0001	4811.005	-41964.04	86400.05
	1	-144	4811	-41964	86399.96	0.049104

3.000003	1	-144	4811	-41964	86399.96	
	0	3.000003	-423.0004	13164.01	-86400.04	
	1	-141	4387.999	-28799.98	-0.075186	

9.000009	1	-141	4387.999	-28799.98		
	0	9.000009	-1188.001	28800.01		
	1	-132	3199.998	0.030345		

32.00003	1	-132	3199.998			
	0	32.00003	-3200.002			
	1	-99.99996	-0.003385			

In order to study the stability, take the last polynomial of the deflation and solve it.

We have: $x - 99.99996 = 0 \Rightarrow x = 99.99996$ (err. rel. = $4.5E-07$)

As we can see, the error is comparable with those introduced in the roots. Therefore the algorithm has limited the error propagation. This does not happen if we start the deflation from the highest root in descendent order. Let's see.

100.0001	1	-145	4955	-46775	128364	-86400
	0	100.0001	-4499.99	45500.6	-127441	92342.36
	1	-44.9999	455.0055	-1274.4	923.4227	5942.364

32.00003	1	-44.9999	455.0055	-1274.4	923.4227	
	0	32.00003	-415.996	1248.299	-835.373	
	1	-12.9999	39.00931	-26.1054	88.04928	

9.000009	1	-12.9999	39.00931	-26.1054		
	0	9.000009	-35.9988	27.0949		
	1	-3.99986	3.010541	0.989503		

3.000003	1	-3.99986	3.010541			
	0	3.000003	-2.99957			
	1	-0.99986	0.01097			

Solving the last polynomial we have $x - 0.99986 = 0 \Rightarrow x = 0.99986$ (err. rel. = $1.4E-04$)

In this case, the error of the final root is much bigger than the errors introduced: the algorithm has amplified the error more than 100 times.

This result is general: the ascendent deflation is stable while the descendent deflation tends to amplify the round off errors.

This fact suggests the convenience to start the searching from the roots having the lowest modulo. Unfortunately this strategy is not always applicable. Some algorithms starts in random way or alternatively starts from the root having the highest module, because they can be isolated more easily

Inverse Deflation

In order to overcome to the intrinsic instability of the descendent deflation we can adopt the "*inverse deflation*", a variant that use the reciprocal polynomial

For example, found the first root $x = 100 \pm 1E-6$, invert $z = 1/x$ and reduce the reciprocal polynomial instead of the original one:

$$86400x^5 - 128364x^4 + 46775x^3 - 4955x^2 + 145x - 1, .$$

This trick comes in handy for each root found: 100, 32, 9, 3

The schema of the inverse deflation is thus:

x	1/x	86400	-128364	46775	-4955	145	-1
100.0001	0.01	0	863.999136	-1274.999	454.9996	-44.99996	0.999999
		86400	-127500	45500.001	-4500	100	-5.94E-07
x	1/x	86400	-127500	45500.001	-4500	100	
32.000032	0.03125	0	2699.9973	-3899.996	1299.999	-99.99995	
		86400	-124800	41600.005	-3200.002	9.11E-05	
x	1/x	86400	-124800	41600.005	-3200.002		
9.000009	0.111111	0	9599.9904	-12799.99	3199.999		
		86400	-115200.01	28800.016	-0.002963		
x	1/x	86400	-115200.01	28800.016			
3.000003	0.333333	0	28799.9712	-28799.99			
		86400	-86400.042	0.0312043			

Solving the last polynomial we have:

$$86400 \cdot x - 86400.042 = 0 \Rightarrow x = 1.0000005 \text{ (err. rel.} = 5E-07 \text{)}$$

As we can see, the error is comparable with those of the previous roots. So the inverse deflation algorithm is stable for roots in descendent order.

Shifting

The variable shifting is a frequent operation that is used for several different scopes: for reducing the polynomial coefficients; for simplifying the evaluation; for improving the factor decomposition; etc. For example, we can transform a general polynomial in a centered polynomial by variable shifting. A centered polynomial has the algebraic sum of its roots equal to zero and then, the coefficients a_{n-1} is always 0, being:

$$\sum_{i=1}^n x_i = 0 \Leftrightarrow a_{n-1} = -\sum_{i=1}^n x_i$$

Any general polynomial can be transformed into centered performing the following variable shifting:

$$x = z - \frac{a_{n-1}}{n \cdot a_n}$$

Example

$$P(x) = x^3 + 9x^2 + 28x + 30$$

setting $x = z - 3$

$$P(z) = (z - 6)^3 + 9(z - 6)^2 + 28(z - 6) + 30$$

Developing and rearranging we have:

$$P(z) = (z^3 - 9z^2 + 27z - 27) + (9z^2 - 54z + 81) + (28z - 84) + 30 = z^3 + z$$

As we can see, the shifted polynomial is much simpler than the original one in a such way that it can be easily factorized by hand.

$$P(z) = z^3 + z = z(z^2 + 1)$$

The roots of $P(x)$ are then:

$$z(z^2 + 1) = 0 \Rightarrow z = \{0, \pm i\} \Rightarrow z = x + 3 \Rightarrow x = \{3, 3 \pm i\}$$

The roots-center shifting is useful also for reducing the polynomial coefficients and simplifying the evaluation. For this purpose is not necessary to obtain the exact centered polynomial; we only need a polynomial closer to the centered one. This trick is sometime useful to avoid decimal values. For example:

$$P(x) = x^4 - 22x^3 + 179x^2 - 638x + 840$$

Let's try to reduce the coefficient by shifting to the root center. Nevertheless the exact shifting would be $z = x - (-22)/4 = x + 5.5$ introducing decimal values. Therefore we are satisfied with a "near centered" polynomial using the shifting $z = x + 5$.

$$P(x) = (x + 5)^4 - 22(x + 5)^3 + 179(x + 5)^2 - 638(x + 5) + 840$$

Developing and rearranging we have:

$$P(z) = z^4 - 2z^3 - z^2 + 2z$$

The shifting is a powerful method for polynomial simplification, but it needs the laborious explicit powers development. It is possible to avoid the power expansion by the following formula

$$b_k = P^{(k)}(a) / k! \quad \text{per } k = 0, 1, 2, \dots, n$$

where b_k are the coefficients of the shifted polynomial $P(x + a)$

This formula avoids the power development but, it requires the factorial computation

We can obtain the coefficients of $P(x + a)$ in a more efficient way by the synthetic division

Iterative Synthetic Division

We apply iteratively the synthetic division algorithm to the original polynomial, then to the quotient polynomial, to the next quotient, and so on, till the end. The divisor number "a" must be the shifting amount $x = z + a$

Let's see an practical example with the polynomial of the previous example

$$P(x) = x^4 - 22x^3 + 179x^2 - 638x + 840$$

We want to shift this polynomial by the substitution $z = x + 5$ getting the new polynomial

$$P(z) = b_4z^4 + b_3z^3 + b_2z^2 + b_1z + b_0$$

We observe that the first coefficients never change. So: $b_4 = a_4 = 1$

The other coefficients b_0, b_1, b_2, b_3 will be found by the iterative synthetic division, in this way.

5	1	-22	179	-638	840	
		5	-85	470	-840	
	1	-17	94	-168	0	
		5	-60	170		
	1	-12	34	2		
		5	-35			
	1	-7	-1			
		5				
	1	-2				

In the 1st row we write the original polynomial coefficients and, at the left, the amount to shift, that is 5.

In the 3rd row we obtain the coefficients of the quotient Q_1 and the remainder (yellow).

We repeat the division starting from the quotient Q_1 obtaining, in the 5th row, the quotient Q_2 and a new remainder. We continue this process until we get only one coefficient.

The coefficients b_0, b_1, b_2, b_3 are the remainders of all the division in crescent order

Then, the shifted polynomial is

$$P(z) = z^4 - 2z^3 - z^2 + 2z$$

that we have already obtained in the previous paragraph

This efficient method, always applicable, may be very useful for polynomial manipulating, overall for those of high degree.

Example. Find the roots of the given 4th degree polynomial

$$P(x) = 5x^4 - 120x^3 + 919x^2 - 2388x + 1980$$

Let's make the polynomial centered by the shift : $x = z - \frac{a_{n-1}}{n \cdot a_n} \Rightarrow x = z + 6$

The first coefficients $b_4 = 5$; the others are found by the iterative synthetic division

6	5	-120	919	-2388	1980	
		30	-540	2274	-684	
	5	-90	379	-114	1296	$\Rightarrow b_0 = 1296$
		30	-360	114		
	5	-60	19	0		$\Rightarrow b_1 = 0$
		30	-180			
	5	-30	-161			$\Rightarrow b_2 = -161$
		30				
	5	0				$\Rightarrow b_3 = 0$

Then, the shifted polynomial is

$$P(z) = 5 \cdot z^4 - 161 \cdot z^2 + 1296$$

that is a biquadratico polynomial, having the roots

$$\Delta = 161^2 - 4 \cdot 5 \cdot 1296 = 1 \Rightarrow z^2 = \frac{161 \pm 1}{10} \Rightarrow z^2 = \{16, 81/5\} \Rightarrow x = \{\pm 4, \pm 9\sqrt{5}/5\}$$

The following VB code performs the polynomial shifting in $P(x + x_0)$

```
Sub PolyShift(x0, A, B)
'Shifts the polynomial A(x) into A(x+x0) = > B(x)
'A()= Coefficients of A(x)=[a0, a1, a2 ...]
'B()= Coefficients of B(x)=[b0, b1, b2 ...]
Dim n, i, k, y
n = UBound(A)
For i = 0 To n
    B(i) = A(i)
Next i
For k = 0 To n
    y = 0
    For i = n To k Step -1
        y = y * x0 + B(i)
        B(i) = B(i) * (i - k) / (k + 1)
    Next i
    B(k) = y
Next k
End Sub
```

Computational cost. If the degree is n, the cost is $3(n^2+3n+2)$. For example, for a 5th degree polynomial, the cost is 126 op

Roots Locating

The most part of rootfinder algorithms works fine if the roots approximated location is known. For a real root it needs to know its bracketing interval while, for a complex root, it necessary to restrict a region of the complex plane containing it. There are several methods, with different accuracy and effort, for locating the polynomial roots.

Graphic Method

A real root may be localized by the same graphical method of the general real function $f(x)$. A complex root, on the contrary, requires a different approach, of course ,more complicated.

For example, we want to draw the roots position of the polynomial $z^3 - z^2 + 2z - 2$

Let's make the variable substitution: $z = x + \hat{i}y$.

$$(x + \hat{i}y)^3 - (x + \hat{i}y)^2 + 2(x + \hat{i}y) - 2 = [x^3 - x^2 + x(2 - 3y^2) + y^2 - 2] + \hat{i}[y(3x^2 - 2x - y^2 + 2)]$$

Setting the equation for both real and imaginary part, we have

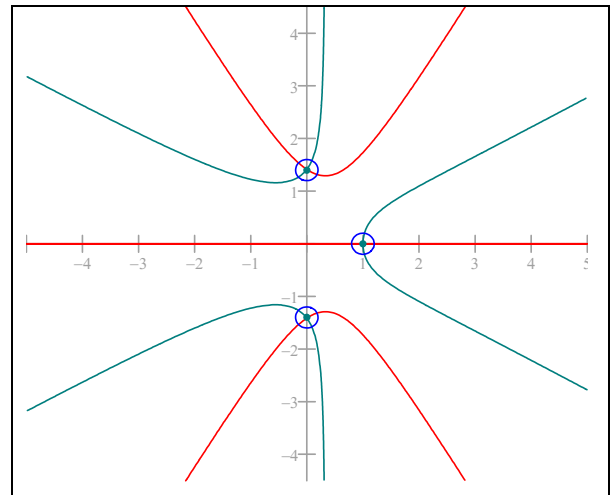
$$\text{Re}(x,y) = x^3 - x^2 + x(2 - 3y^2) + y^2 - 2 = 0 \quad (1)$$

$$\text{Im}(x,y) = y(3x^2 - 2x - y^2 + 2) = 0 \quad (2)$$

we can plot in the plane (x,y) the implicit function defined by (1) and (2). This task can be performed by math-graphical tools available also in the public domain, in Internet¹. The graph shows the curve $\text{Re}(x,y) = 0$ (blue) and the curve $\text{Im}(x,y) = 0$ (red). Observe that the x axe is a solution of the imaginary equation. The intersection points are the roots of the given polynomial

From the axes scale, we see that the roots are located near the points

$$P_i = (1, 0), (0, 1.4), (0, -1.4)$$



The graphic method is, of course, unavailable for automatic rootfinder programs and, moreover, it becomes quite cumbersome for high degree polynomial; but on the other hand, the result give us a complete and precise localization of the roots. We surely recommend it, when it is possible.

A list of the complex power expansions $(x + \hat{i}y)^n$, up to 9 degree, is showed in appendix

¹ The graphic was produced by WinPlot, a very good free program for math-graphical analysis by Peanut Software. Of course, here, works fine also other excellent programs.

Companion Matrix Method

A general method for localizing all the polynomial roots, treated in almost math books, is based on the companion matrix.

Given a monic polynomial (that is with $a_n = 1$)

$$a(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + x^n$$

set the following matrix

$$A = \begin{bmatrix} 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & -a_{n-1} \end{bmatrix}$$

As known, the eigenvalues of A are the roots of the given polynomial. Applying the Gershgorin's theorem at the rows, we have the following relations.

$$|\lambda| \leq |a_0| \quad |\lambda| \leq |a_i| + 1 \quad |\lambda - a_{n-1}| \leq 1 \quad \text{per } i = 1, 2, n-2$$

Applying the Gershgorin's theorem at the columns, we have the following relations.

$$|\lambda| \leq 1 \quad |\lambda - a_{n-1}| \leq \sum_{i=0}^{n-2} |a_i|$$

All these relations can be reassumed into the following general formula

$$|x| = \max \left\{ 1, |a_0|, \sum_{i=0}^{n-1} |a_i|, 1 + \max_{i=1}^{n-1} |a_i| \right\} \quad (1)$$

This formula is very important in theory field, but for numerical practical uses is not very useful because it is usually poor restrictive. Let's see this simple example

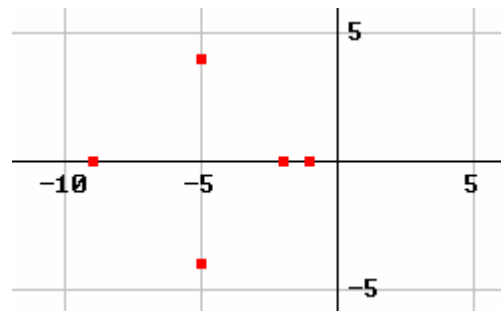
$$x^5 + 22 \cdot x^4 + 190 \cdot x^3 + 800 \cdot x^2 + 1369 \cdot x + 738$$

The roots are:

$$x = -5 - 4i, x = -5 + 4i, x = -9, x = -2, x = -1$$

The formula (1) gives:

$$|x|_{\max} = \max \{ 1, 738, 3119, 1370 \} = 3119$$



This means that all roots are located into the circle, centered at the origin, with radius $\rho = 3119$. Clearly a too large region, for practical scope!

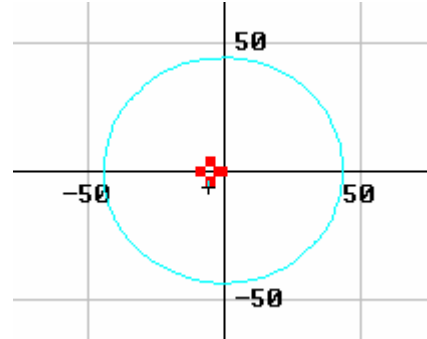
Circle Roots

This method exploits the following formula for a monic polynomial ($a_n = 1$)

$$\rho = 2 \max \{ |a_{n-k}|^{1/k} \} \text{ per } k = 1, 2, \dots, n \quad (2)$$

Applying it to the polynomial of the previous example gives the radius $\rho = 44$, that is still large but much more accurate than the previous estimation obtained with the companion method.

A remarkable improvement it is possible observing that the formula (2) works good if a_{n-1} is small respect to the other coefficients



Variant of the centered polynomial

In order to make the coefficient a_{n-1} equal to zero (or smaller, at the least) we can transform the given polynomial into a centered polynomial by shifting of:

$$b = \frac{1}{n} \sum x_i = -\frac{a_{n-1}}{n a_n} \quad (3)$$

In the example of $x^5 + 22x^4 + 190x^3 + 800x^2 + 1369x + 738$

The center is $C = (-22/5, 0) = (-4.4, 0)$.

Rounding 4.4 to the nearest integer to avoid decimal values, we perform the variable substitution $x = z - 4$ by the synthetic division

	1	22	190	800	1369	738
-4		-4	-72	-472	-1312	-228
	1	18	118	328	57	510
		-4	-56	-248	-320	
	1	14	62	80	-263	
		-4	-40	-88		
	1	10	22	-8		
		-4	-24			
	1	6	-2			
		-4				
	1	2				

The shifted polynomial is: $z^5 - 2z^4 - 2z^3 - 8z^2 - 263z + 510$

If we apply the formula (2) to this polynomial we have the radius estimation $r \cong 8$. Therefore the roots circle equation now becomes

$$(x + 4.4)^2 + y^2 = 8^2$$

The following graph shows a great improvement in the area restriction

The outer circle, having equation

$$x^2 + y^2 = 44^2$$

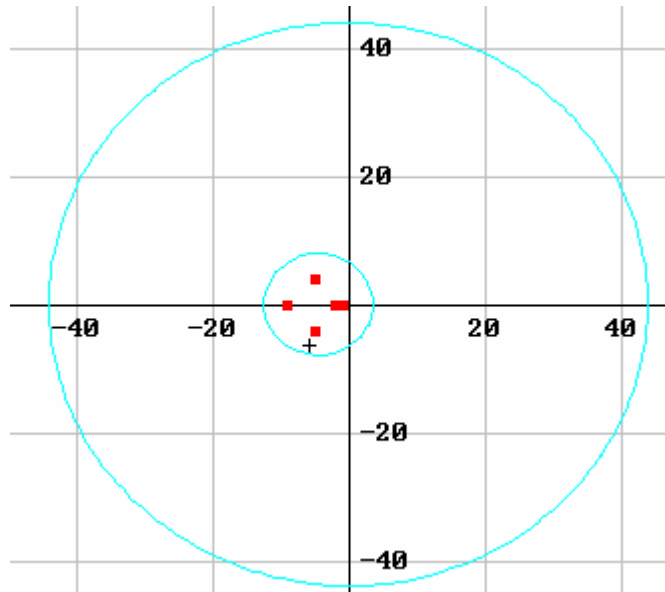
has been obtained applying the formula (2) to the original polynomial

The inner circle, of equation

$$(x + 4.4)^2 + y^2 = 8^2$$

has been obtained applying the formula (2) to the "almost" centered polynomial

As we can see, the clear improvement repays the effort for the polynomial shifting

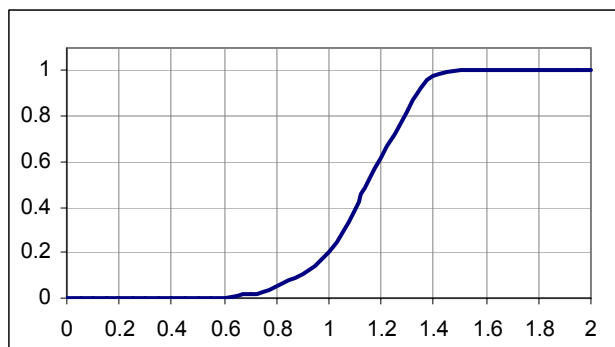


A statistical approach allows to get a closer circle choosing for the formula (2) a coefficient in the range: $1.3 \div 1.5$. In fact, if we consider the set of random polynomials of degree n^{th} , with $2 < n < 14$, the radius ρ returned by the formula (2) is a random variable, as it is the true radius ρ^* .

The variable ρ distributes itself around the ρ^* with some probability distribution

Defining the normalized random variable $t = 2 \rho^* / \rho$, we can get its experimental cumulative distribution (F),

Polin. 10° degree
 Samples = 790
 Avg = 1.13
 Dev.st = 0.17
 Max = 1.47
 Min = 0.63



For $t > 1.4$ the probability of enclosing all the roots is more than 95%. So, we have:

$$\rho^* = t \rho / 2 \Rightarrow \rho^* \cong 1.4 \rho / 2 \Rightarrow \rho^* = 1.4 \max \{ |a_{n-k}|^{1/k} \} \text{ per } k = 1, 2, \dots, n$$

That is the statistic formula for the radius estimation of the centered polynomials

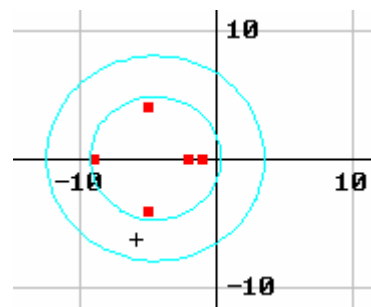
Applying this formula to the given polynomial we have

$$\rho^* \cong 1.4 \cdot 4 = 5.6$$

The roots circle now becomes:

$$(x + 4.4)^2 + y^2 = 5.6^2$$

with a sensible further restriction



Iterative Powers Method

This is the so called Bernoulli's method, modified for obtaining the roots, real or complex, having maximum and minimum module. Given the polynomial equation

$$a_n z^n + a_{n-1} z^{n-1} + a_{n-2} z^{n-2} + a_2 z^2 + a_1 z + a_0 = 0$$

we derive the associated difference equation

$$a_n s_n + a_{n-1} s_{n-1} + a_{n-2} s_{n-2} + a_2 s_2 + a_1 s_1 + a_0 s_0 = 0$$

solving

$$s_n = -(a_{n-1} s_{n-1} + a_{n-2} s_{n-2} + \dots + a_2 s_2 + a_1 s_1 + a_0 s_0) / a_n, \quad n = n+1, n+2, \dots (1)$$

The quotient s_n / s_{n-1} of the sequence $\{s_n\}$ converges to the dominant real root, that is the real root having maximum module.

$$\rho \cong |s_n / s_{n-1}| = \quad \text{per } n \gg 1 \quad (2)$$

If the dominant root is complex, the sequence (1) does not converge. In that case we can get the module of the complex root using the last 4 values and the following formula

$$\rho \cong ((s_{n-1})^2 - s_n s_{n-2}) / (s_{n-2})^2 - s_{n-1} s_{n-3})^{-1/2} \quad \text{per } n \gg 1 \quad (3)$$

The root of minimum module can be extracted transforming the given polynomial into its reciprocal by the variable substitution $z \rightarrow 1/z$. By this trick the smallest roots becomes the biggest one and we can apply the powers' method to the following sequence

$$q_n = -(a_1 q_{n-1} + a_2 q_{n-2} + a_3 q_2 + a_{n-1} q_1 + a_n q_0) / a_0, \quad n = n+1, n+2, \dots (4)$$

The formulas (2) e (3) applied to the sequence $\{q_n\}$ give the reciprocal ρ^{-1} of the minimum radius.

Lets' see how this method works with practical examples

Example: $x^5 + 22x^4 + 190x^3 + 800x^2 + 1369x + 738$

The starting sequence can be freely chosen. Usually it is the sequence $[0, 0, 0, 1]$. The first 3 iterations are shown in the following table

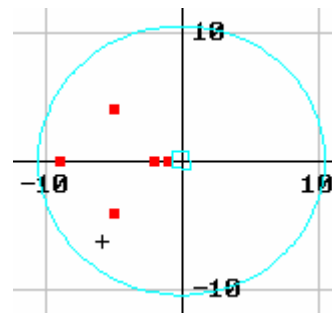
Polynomial. reciprocal		Polynomial original		
n	q _n	ρ min	s _n	ρ max
5	-1.85501	0.539	-22	22
6	2.35706	0.787	294	13.36
7	-2.61898	0.900	-3088	10.50

As we can see, the sequence $\{s_n\}$ grows quickly but usually only few iterations are needed:

In that case, we estimate the max radius as 10.5 and the min radius as 0.9. So all the roots must satisfy the relation

$$0.9 < |x| < 10.5$$

that is represented by two concentric circles of the figure



Now, let's see the case of complex dominant root

$$x^5 + 18x^4 + 135x^3 + 450x^2 + 644x + 312 = 0$$

The exact roots of the polynomial are $x = -6 - 4i$, $x = -6 + 4i$, $x = -3$, $x = -2$, $x = -1$

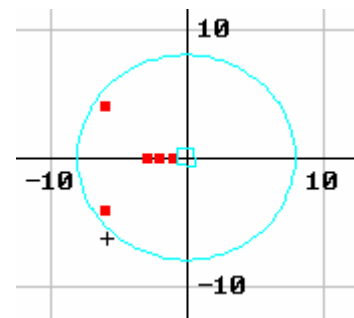
The powers method gives the following sequences, where for ρ_{\max} we have used the formula (3).

n	Reciprocal polynomial		Original polynomial	
	q_n	ρ_{\min}	s_n	ρ_{\max}
5	-2.06	-0.484	-18	
6	2.818	-0.732	189	11.61
7	-3.270	-0.861	-1422	8.660
8	3.525	-0.928	7537	7.682

In that case we estimate the max radius as 10.5 and the min radius as 0.9. So all the roots must satisfy the following relation

$$0.9 < |x| < 10.5$$

that is represented by two concentric circles of the figure



Quotient-Difference Method

The QD (quotient-difference) algorithm is a brilliant generalization of the powers' method, developed by Rutishauser, that converges simultaneously to all the roots of polynomial equation. In addition, it does not need any starting values. Under appropriate, reasonable conditions, it converges to the real roots giving an estimation of the accuracy reached. By a little variant it can also find the complex roots. The convergence order of QD method is linear, so its principal use is to provide a good initial approximation to other faster refinement algorithms such as Newton-Raphson or Halley.

The QD algorithm details are quite complicated and there are several different variants. Here we show only the details for an electronic worksheet implementation.

It consists in a table of rows and columns: the i -th row represents the correspondent iterate while the columns contain the differences (d) and the quotients (q), alternatively.

For clarity we show how to develop a QD table by the following example.

$$x^5 + 22x^4 + 190x^3 + 800x^2 + 1369x + 738$$

The QD table will be made with 11 columns, alternatively, $d_1, q_1, d_2, q_2, d_3, q_3, d_4, q_4, d_5, q_5, d_6$

The columns d_1 to d_6 are auxiliary and are always filled with 0. As each row corresponds to an iteration step, let's indicate as $d^{(i)}, q^{(i)}$ the coefficients at the i -th step. Therefore, for example, the coefficients at the starting step 0 will be $d_1^{(0)}, q_1^{(0)}, d_2^{(0)}, q_2^{(0)}$, etc. Those at the step 1 will be $d_1^{(1)}, q_1^{(1)}, d_2^{(1)}, q_2^{(1)}$, and so on.

Initialization. The first row, indicated with 0, is used to start the QD iteration and is built with only the polynomial coefficients in the following way.

$$q_1^{(0)} = -a_4/a_5 \quad q_2^{(0)} = q_3^{(0)} = q_4^{(0)} = q_5^{(0)} = 0$$

$$d_1^{(0)} = 0, \quad d_2^{(0)} = a_3/a_4, \quad d_3^{(0)} = a_2/a_3, \quad d_4^{(0)} = a_1/a_2, \quad d_5^{(0)} = a_0/a_1, \quad d_6^{(0)} = 0$$

Inspecting this formulas, we deduce that the polynomial must have no coefficient equal to zero. This is one of the QD algorithm constrains. We'll see later how to overcome this limit. The first row of the QD table looks like the following

n	d ₁	q ₁	d ₂	q ₂	d ₃	q ₃	d ₄	q ₄	d ₅	q ₅	d ₆
0	0	-22	8.6364	0	4.2105	0	1.7113	0	0.5391	0	0
1											

The next rows will be generate using the characteristic relation from which the algorithm takes its name.

$$q_k^{(i+1)} - q_k^{(i)} = d_{k+1}^{(i)} - d_k^{(i)} \quad (1)$$

$$d_k^{(i+1)} / d_k^{(i)} = q_k^{(i+1)} / q_{k-1}^{(i+1)} \quad (2)$$

From (1) we get the coefficients $q^{(i+1)}$ of the step $i+1$

$$q_k^{(i+1)} = q_k^{(i)} + d_{k+1}^{(i)} - d_k^{(i)} \quad (3)$$

with $k = 1, 2...5$

From (2) we get the coefficients $d^{(i+1)}$ of the step $i+1$

$$d_k^{(i+1)} = d_k^{(i)} \cdot (q_k^{(i+1)} / q_{k-1}^{(i+1)}) \quad (4)$$

with $k = 2, 3..5$

The values $d_1^{(i+1)} = d_6^{(i+1)}$, as before told, are always zero.

Synthetically, to generate a new row $(i+1)$, we alternates the following schemas

$d_k^{(i)}$	$q_k^{(i)}$	$d_{k+1}^{(i)}$
	$q_k^{(i+1)}$	

	$d_{k+1}^{(i)}$	
$q_k^{(i+1)}$	$d_{k+1}^{(i+1)}$	$q_{k+1}^{(i+1)}$

To calculate $q^{(i+1)}$ we use the adjacent values of the previous row. Afterwards, to generate $d^{(i+1)}$ we use the adjacent values of the same row and the value $d^{(i)}$ of the previous row.

The first 8 iterations are listed in the following table.

n	d ₁	q ₁	d ₂	q ₂	d ₃	q ₃	d ₄	q ₄	d ₅	q ₅	d ₆
0	0	-22	8.6364	0	4.2105	0	1.7113	0	0.5391	0	0
1	0	-13.36	2.8602	-4.426	2.3777	-2.499	0.8026	-1.172	0.2479	-0.539	0
2	0	-10.5	1.3366	-4.908	1.9737	-4.074	0.3402	-1.727	0.113	-0.787	0
3	0	-9.167	0.6228	-4.271	2.6375	-5.708	0.1164	-1.954	0.052	-0.9	0
4	0	-8.544	0.1645	-2.257	9.6179	-8.229	0.0286	-2.018	0.0245	-0.952	0
5	0	-8.379	-0.141	7.1968	-23.81	-17.82	0.0032	-2.022	0.0119	-0.977	0
6	0	-8.521	-0.273	-16.47	8.669	5.9976	-0.001	-2.014	0.0058	-0.988	0
7	0	-8.794	-0.234	-7.532	3.0758	-2.673	-8E-04	-2.007	0.0029	-0.994	0
8	0	-9.028	-0.109	-4.223	4.1878	-5.749	-3E-04	-2.003	0.0014	-0.997	0

At the first sight, we see the columns q_1, q_4, q_5 converging to the real roots, while, on the contrary, q_2, q_3 do not seem to converge. We suspect the presence of two conjugate complex roots.

The following paragraph gives the general rules for extracting the roots from a QD table

Real single roots - estimate the root error by the formula

$$e_i = 10(|d_{i,k}| + |d_{i+1,k}|) \quad (5)$$

if $(e_i / |q_i|) < \epsilon_r$ then accept the root q_i

Complex conjugate roots - calculate the new sequences $\{s_i\}$ e $\{p_i\}$

$$s_i = q_{i,k} + q_{i,k+1} \quad (6)$$

$$p_i = q_{i-1,k} \cdot q_{i,k+1} \quad (7)$$

estimate the root error by the formulas

$$e_{s_i} = |s_i - s_{i-1}| \quad (8)$$

$$e_{p_i} = |p_i - p_{i-1}|^{1/2} \quad (9)$$

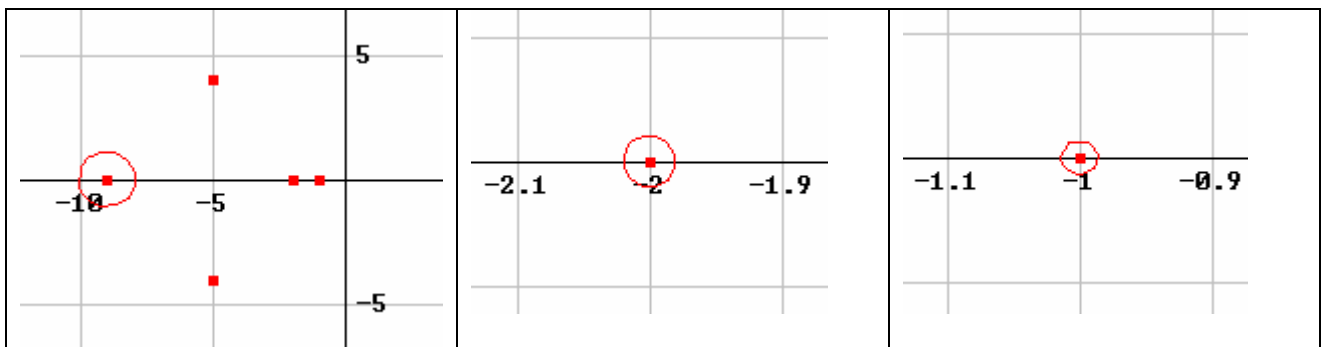
if $(e_{s_i} / |s_i|) < \epsilon_r$ and $(e_{p_i} / |p_i|) < \epsilon_r$ then

take the two values $s = s_i$, $p = p_i$ and solve the equation: $x^2 + s \cdot x + p = 0$

Let's stop the 8th row. According to the estimation formula (5), the relative errors are:

err 1	err 2	err 3	err 4	err 5
12.12%	1017.7%	728.5%	0.86%	1.44%

Taken the radius proportional to the relative errors, the roots circles are shown in the following graphs. Note the different scale of the first graph relatively to the others.



Let's see now the complex roots. From the last 2 rows of the QD table we calculate

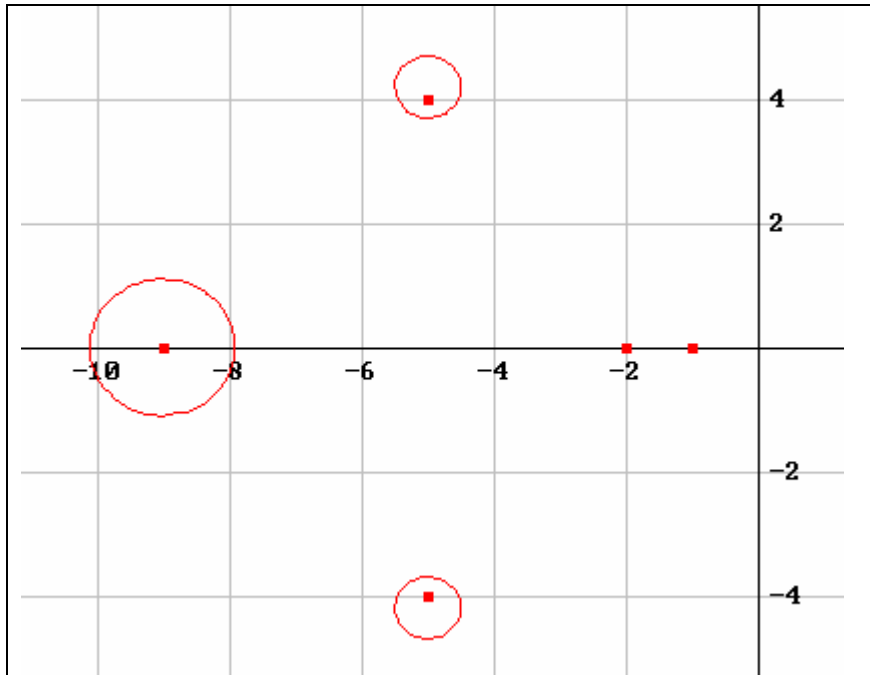
$$s = q_2^{(8)} + q_3^{(8)} \cong -10 \quad , \quad p = q_2^{(7)} q_3^{(8)} \cong 43$$

We obtain the roots solving the equation $(x^2 - 10x + 43) = 0$, $\Rightarrow x_{1,2} \cong 5 \pm 4.2 i$

The average estimation provided by (8) e (9) is about $er \cong 0.076$, giving an absolute error of about $er \cong 0.07 (5^2 + 4.2^2)^{1/2} \cong 0.5$. We can assume this estimation equal to the radius of the two complex conjugated roots. So, finally, the equations of the roots circles are:

$$(x + 5)^2 + (y - 4.2)^2 = 0.5^2 \quad e \quad (x + 5)^2 + (y + 4.2)^2 = 0.5^2$$

These equations are plotted in the following graph.



Null coefficients

As we have seen, the QD algorithm cannot start if one or more coefficients are null. In that case we shift the polynomial substituting the variable $x = z - a$ where "a" is a random value. This transformation eliminates the null coefficients and then we can apply the QD algorithm for finding the roots z_k of the shifted polynomial. After that, it is sufficient to perform the inverse shift $x_k = z_k - a$ to obtain the original roots

For example, the polynomial $x^5 - 8x^2 - 10x - 1$ can be transformed in a complete polynomial by the substitution $x = z + 1$ obtaining

$$(z+1)^5 - 8 \cdot (z+1)^2 - 10 \cdot (z+1) - 1 = z^5 + 5z^4 + 10z^3 + 2z^2 - 21z - 18$$

Of course, all the roots are also shifted by 1

Real distinct roots

Consider the 5th degree Wilkinson polynomial

$$P(x) = x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120$$

As known, it has all distinct real roots $x = 1, 2, 3, 4, 5$

Let's see how the QD algorithm works in that case.

The QD table containing the first 12 iterations, is:

n	d ₁	q ₁	d ₂	q ₂	d ₃	q ₃	d ₄	q ₄	d ₅	q ₅	d ₆
0	0	15	-5.667	0	-2.647	0	-1.218	0	-0.438	0	0
1	0	9.333	-1.833	3.019	-1.253	1.4293	-0.664	0.7798	-0.246	0.438	0
2	0	7.5	-0.88	3.6	-0.702	2.0178	-0.395	1.1983	-0.14	0.6839	0
3	0	6.62	-0.502	3.7777	-0.432	2.3255	-0.246	1.4525	-0.08	0.8243	0
4	0	6.1178	-0.316	3.8476	-0.282	2.5114	-0.159	1.6193	-0.044	0.904	0
5	0	5.802	-0.211	3.8813	-0.192	2.6346	-0.105	1.7337	-0.024	0.9484	0
6	0	5.5907	-0.147	3.901	-0.134	2.7216	-0.07	1.8139	-0.013	0.9728	0
7	0	5.4433	-0.106	3.9148	-0.095	2.7855	-0.047	1.8706	-0.007	0.9858	0

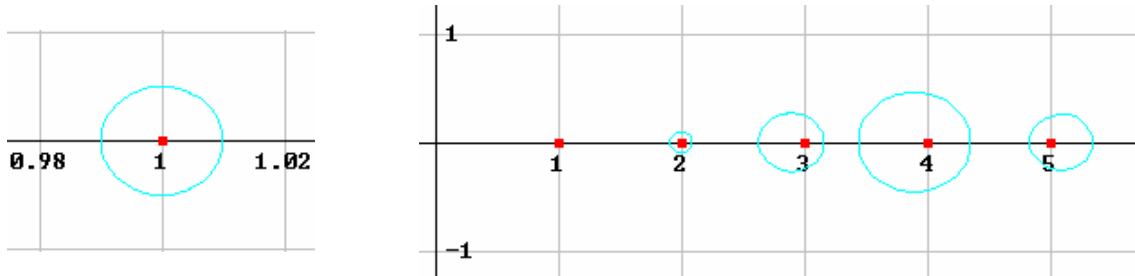
8	0	5.3373	-0.078	3.9257	-0.069	2.8338	-0.032	1.9105	-0.004	0.9927	0
9	0	5.2593	-0.058	3.9351	-0.05	2.8709	-0.021	1.9385	-0.002	0.9963	0
10	0	5.201	-0.044	3.9433	-0.037	2.8997	-0.014	1.9579	-9E-04	0.9981	0
11	0	5.1567	-0.034	3.9507	-0.027	2.9221	-0.01	1.9714	-5E-04	0.999	0
12	0	5.1228	-0.026	3.9574	-0.02	2.9397	-0.007	1.9806	-2E-04	0.9995	0

As we can see, all the columns "d" approach to zero while the roots appear clearly in the columns "q". It is surely the case where the algorithm shows all its power finding simultaneously all the real distinct roots..

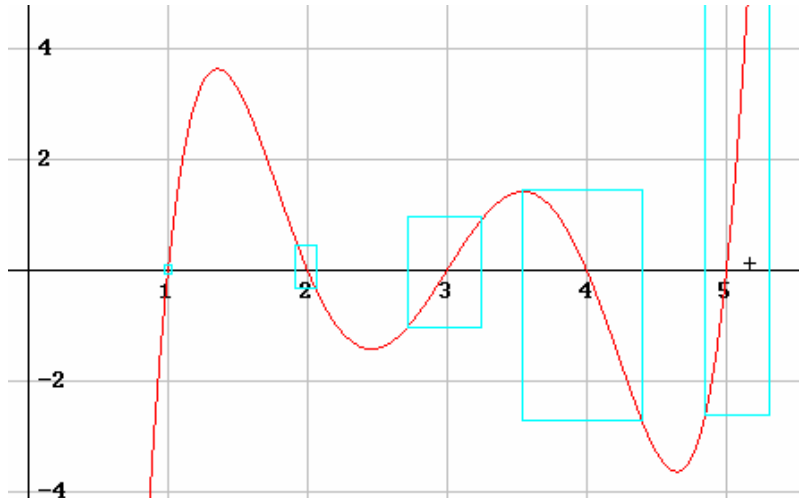
By the formula (5) we get these prudent error estimations

X ₁	X ₂	X ₃	X ₄	X ₅
5.13	3.96	2.94	1.98	1.00
5.1%	11.7%	9.1%	3.4%	0.24%

Plotting the usual root circles, we have:



In this picture we have plotted the polynomial curve and the bracketing intervals: the length of each interval is the diameter of the correspondent circle. Intervals can be used to start the rootfinder routines: if we use a one-point algorithm like the Newton, we take the middle point of the interval; if we use the a two points algorithm like the secant, we use the extremes. In both cases the convergence is guaranteed



Computational cost. Clearly the roots area searching using the QD algorithm can give good result, but it is more expensive. If "n" is the polynomial degree, then it takes $2(2n-1)$ op for each iteration (this does not include the operations for initializing).

For example, the total cost of the previous QD table is about $[2(2 \cdot 5 - 1)] \cdot 12 = 216$ op

Multiple roots

We have seen that the presence of multiple roots strongly deteriorates the general accuracy and the efficiency of almost rootfinder methods. This is true also for polynomials. But, for polynomials special strategies exist for minimize this effect. One of the most popular method consists to substitute the original polynomial with another polynomial having the same roots but all single. The problem of finding such polynomial can be solved in theory with the GCD (Greatest Common Divisor).

Polynomial GCD

We shell explain the GCD method with practical examples

Given the polynomial.

$$P(x) = x^5 + 12x^4 + 50x^3 + 88x^2 + 69x + 20$$

having the roots: $x_1 = -1$ with multiplicity $m = 3$, $x_2 = -4$ $x_3 = -5$ that is: $P(x) = (x+1)^3(x+4)(x+5)$ and we take its first derivative

$$P'(x) = 5x^4 + 48x^3 + 150x^2 + 176x + 69$$

Set: $P_0(x) = P(x)$ e $P_1(x) = P'(x)$, and perform the division of the these polynomial, obtaining a quotient and a remainder

$$P_0(x) / P_1(x) \Rightarrow Q_1(x), R_1(x)$$

Actually, of this division we are interesting for only the remainder: if its degree is greater then zero, make it monic¹ and set:

$$P_2(x) = (-25/76) \cdot R_1(x) = x^3 + (120/19)x^2 + (183/19)x + (82/19)$$

Continue to perform the division of the two polynomials $P_1(x) / P_2(x)$

$$P_3(x) = (-361/675) \cdot R_2(x) = x^2 + 2x + 1$$

and then, the division of the two polynomials $P_2(x) / P_3(x)$

$$P_4(x) = 0$$

The last polynomial is a constant (degree = 0) and then, the GCD algorithm stops itself. Because the constant is 0 this means that the polynomial P_3 is the GCD between the given polynomial and its derivative. Consequently, the roots of P_3 , being common to $P(x)$ and its derivative $P'(x)$, must be all the only multiple roots of the given polynomial $P(x)$.

In fact

$$P_3(x) = x^2 + 2 \cdot x + 1 = (x+1)^2$$

Note that the multiplicity here is less then one unit of the original polynomial root. This is a general result: if the root of the polynomial $P(x)$ has multiplicity "m", then the same root of the GCD will have multiplicity "m-1".

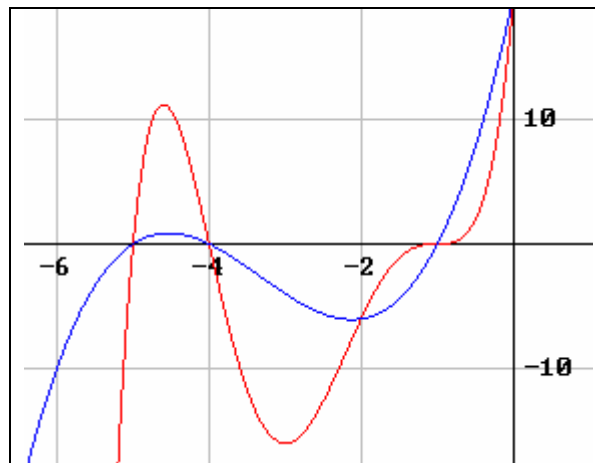
Let's divide the given polynomial for the GCD found.

$$C(x) = P(x) / P_3(x) = x^3 + 10x^2 + 29x + 20$$

The reduce polynomial $C(x)$, also called *cofactor* of $P(x)$, contains only the roots of the given polynomial $P(x)$ but with unitary multiplicity; therefore it is the wanted polynomial.

¹ This trick is not obligatory, but it will useful to reduce the round-off errors

The following graph shows the polynomial $P(x)$ (red) and its cofactor $C(x)$ (blue)



Around the roots $x_1 = -1$, $x_2 = -4$, $x_3 = -5$, the function $C(x)$ has always a slope different than zero. Consequently, applying a rootfinder algorithm to $C(x)$ we presume that we will get fast and accurate results for all the roots.

GCD with decimals

The method developed for the GCD is called Euclid's algorithm that, for its intrinsic elegance and good efficiency, is still used. Its computational cost is about $2n(n-1)$

Unfortunately, the calculus can be performed in exact fractional arithmetic only for small degree polynomials. In practical cases, the GCD computation is performed in floating point arithmetic and the round-off errors degrade the global accuracy of the GCD and the cofactor $C(x)$.

For example, the polynomial

$$x^6 + 46x^5 + 878x^4 + 8904x^3 + 50617x^2 + 152978x + 192080$$

has the roots: $x_1 = -7$, $m = 4$; $x_2 = -8$; $x_3 = -10$. That is: $P(x) = (x + 7)^4 \cdot (x + 8) \cdot (x + 10)$

Let's find its cofactor by the GCD method.

Synthetically, the polynomials coefficients are listed in the following table

	x^0	x^1	x^2	x^3	x^4	x^5	x^6
p0	192080	152978	50617	8904	878	46	1
p1	152978	101234	26712	3512	230	6	
p2	2775.1818	1532.364	316.90909	29.090909	1		
p3	343	147	21	1			
p4	4.186E-09	-2.4E-10	-1.01E-10				

The calculus has been performed by an electronic spreadsheet with 15 significant digits. We note that the values of the last row are about $1E-9$ instead of zero. This approximation, classic in the numerical calculus, is due to the round-off errors. Also the cells visualized as integer, if carefully examined, show the presence of decimal digits,

In fact the true coefficients of the GCD and the cofactor obtained are.

	a0	a1	a2	a3
CGD	343.0000000049060	147.0000000017460	21.0000000001279	1.0000000000000
cofactor	559.9999999913070	205.9999999977430	24.9999999998721	1.0000000000000

It is evident that the exact GCD polynomial should be: $x^3+21x^2+147x+343$, but this simple example shows that, in practical cases, the round-off errors may be very snaky and vanish also the most brilliant algorithms

It is clear that, in practical cases, the naive detection of the zero for stopping the algorithm cannot work anymore. We have to choose another approach. For example, defining the norm of the polynomial $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$

$$\| P(x) \| = (a_0^2 + a_1^2 + \dots + a_{n-1}^2 + a_n^2)^{1/2} \quad (1)$$

a possible stop criterion should check the following relation

$$\| P_k(x) \| < \varepsilon \| P_{k-1}(x) \|$$

where ε is a fixed tolerance. If the relation is true the algorithm must stop itself to the k -th iteration. In the previous example the stop happens at the 4th iteration because

$$\varepsilon = 1E-10, \| P_3(x) \| \cong 1000, \| P_4(x) \| \cong 4.2E-9.$$

Let's see now what precision the roots approximation can get. We compute all the single roots of the approximated cofactor $C(x)$ by the Newton-Raphson, choosing with the aid of the following graph, the starting values : $x_0 = -11, -8.5, -6.5$



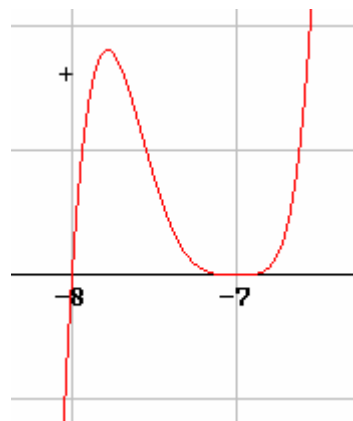
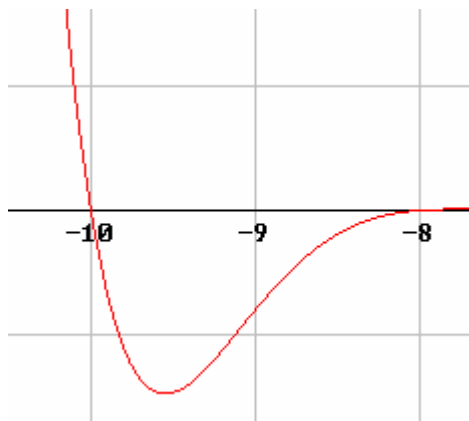
We get the following zeros

x	err
-7.00000000028028	2.8E-10
-7.99999999941018	5.9E-10
-10.0000000001817	1.82E-10

All the roots have a good global accuracy, overall for the quadruple root $x = -7$

What accuracy could we obtain using directly the rootfinder with the original polynomial?

From the $P(x)$ graph we choose the possible starting points: $x_0 = -10.2, -8.5, -6.8$

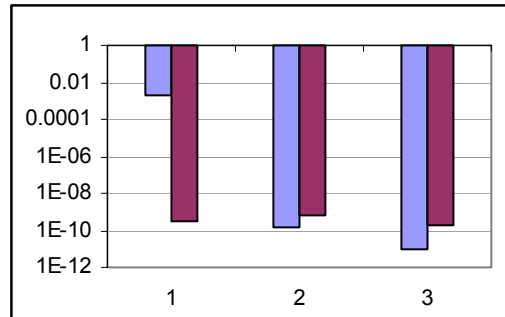


With the Newton-Raphson algorithm we find the following zeros.

x	err
-7.00223808742748	0.002238
-8.00000000015983	1.6E-10
-9.99999999999033	9.67E-12

In that case, we note a large difference of accuracy: from about 1E-11 to 1E-3 of the quadruple root.

The bar-graph compares the accuracy of the three roots obtained with the GCD method (red) and the direct method (light blue). As we can see, the GCD method gains great accuracy for the multiple root and loses a bit for the single roots. But the global gain is positive in any way.



Example. The following polynomial

$$x^9 - 10x^8 + 42x^7 - 100x^6 + 161x^5 - 202x^4 + 200x^3 - 144x^2 + 80x - 32$$

has the roots: $x_1 = 2$, $m = 5$; $x_2 = \pm i$, $m = 2$. That is: $P(x) = (x - 2)^5 \cdot (x^2 + 1)^2$

Find its cofactor by the GCD method.

Synthetically, the polynomials coefficients are listed in the following table

	x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9
p0	-32	80	-144	200	-202	161	-100	42	-10	1
p1	80	-288	600	-808	805	-600	294	-80	9	
p2	40.72727	-65.4545	69.81818	-61.8182	23.63636	4.636364	-5.45455	1		
p3	16	-32	40	-40	25	-8	1			
p4	7.74E-13	-1.3E-12	1.49E-12	-1.4E-12	7.32E-13	-1.4E-13				

We stop the algorithm at the 4th iteration because:

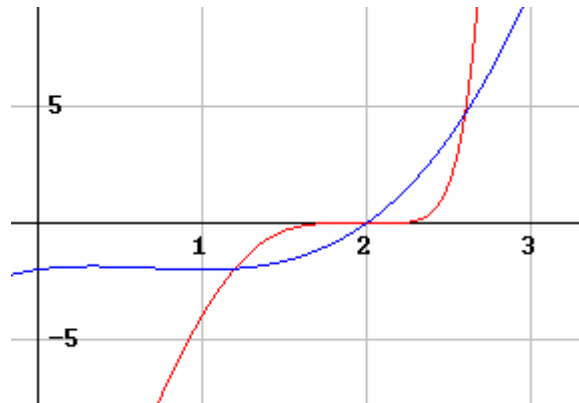
$$\varepsilon = 1E-12, \quad \|P_3(x)\| \cong 6016, \quad \|P_4(x)\| \cong 2.6E-12.$$

The coefficients of the GCD and the cofactors are

	a_0	a_1	a_2	a_3	a_4	a_5	a_6
GCD	16	-32	40	-40	25	-8	1
cofactor	-2	1	-2	1			

The figure shows the cofactor polynomial (blue) and the original polynomial (red). Let's approximate the only real root using both the cofactor and the original polynomials, choosing as starting point $x_0 = 2.5$

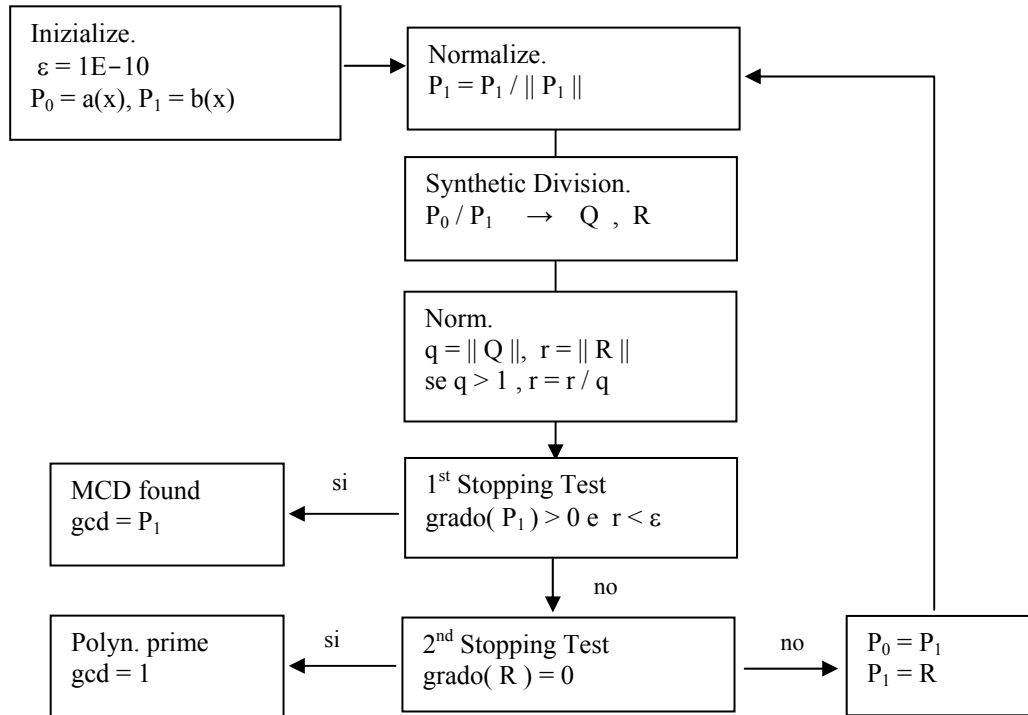
	iter.	x	err
original	28	2.001269	1.27E-03
cofactor	6	2	1.00E-15



For both computing we have used the Newton method. The difference is remarkable. Note that applying the Newton variant for multiplicity roots to the original polynomial could not take any improvement, because it only works on the convergence speed but not for the finally accuracy. To escape from the accuracy "barrier" of the multiplicity is necessary to change the polynomial

GCD - A practical Euclid's algorithm.

For computing the GCD of two polynomial $a(x)$ e $b(x)$, having $\text{degree}(a) \geq \text{degree}(b)$, in order to reduce the error propagation and to stop the process, it is convenient to adopt the following practical algorithm (for standard precision: 15 significant digits)



Example. For the following polynomial $a(x)$ and $b(x)$ the sequence of $R(x)$ is shown in the table

a(x)	b(x)
31680	-97488
-97488	227176
113588	-191469
-63823	74336
18584	-13450
-2690	888
148	7
1	

x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7
31680	-97488	113588	-63823	18584	-2690	148	1
-97488	227176	-191469	74336	-13450	888	7	
-94.5125	223.064	-191.106	75.63561	-14.081	1		
48.4334	-88.652	51.2438	-12.0271	1			
-16	24	-9	1				
-9.93E-13	1.27E-12	-2.78E-13					

For space saving, only a few decimals are visible.

The complete GCD coefficients are shown in the following table

coeff	coeff (arr.)
-15.99999999994830	-16
23.99999999993140	24
-8.99999999998298	-9
1.000000000000000	1

A very simple trick to improve the global accuracy is the integer rounding, as shown in the column (coeff. arr.). This can be done every time that the original polynomial $a(x)$ is integer and monic. Note that, in this case, the polynomial $b(x)$ is the derivative of $a(x)$.

GCD - the cofactor residual test

The Euclid's algorithm, illustrated in the previous paragraph, finds the GCD of two polynomials having similar degrees but can easily fail if the degrees are quite different. In a few cases the Euclid's algorithm returns a GCD even for relative prime polynomials.

Example. Calculate the GCD of the polynomials:

$$a(x) = 31680 - 97488x + 113588x^2 - 63823x^3 + 18584x^4 - 2690x^5 + 148x^6 + x^7$$

$$b(x) = 950 - 937x + x^2$$

x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7
31680	-97488	113588	-63823	18584	-2690	148	1
950	-937	1					
-1.0149735	1						
-2.2204E-16							

The remainder of about 2E-16, is very small, and then the second-last row should be contains the GCD coefficients. But this is completely wrong because the two polynomials are relative prime and their GCD must be 1.

The insufficient selectivity of the Euclid's algorithm is known and during the time several variants and methods to overcome this problem. A relative simple and cheap method for GCD testing exploits the cofactors residuals. Let's see how it works.

Being:

$$a(x) = \text{gcd} \cdot C_a + R_a \quad , \quad b(x) = \text{gcd} \cdot C_b + R_b$$

if "gcd" is exact , then the remainders R_a and R_b must be zero or very small, at least.

Practically, given an approximated "gcd" polynomial, we perform the divisions to obtain the cofactors C_a e C_b and their remainders R_a e R_b ; afterwards we compute the residuals with the formulas:

$$r_a = \| R_a \| / \| C_a \| \quad , \quad r_b = \| R_b \| / \| C_b \|$$

If the degree n_a of the polynomial $a(x)$ is close to the degree n_b of $b(x)$, then the weight of the two residuals r_a e r_b is equivalent and $r \cong \text{average}(r_a, r_b)$; on the contrary, if the degrees are quite different, then the average residual is better approximated by the following formulas:

$$r \cong (r_a \cdot n_a^2 + r_b \cdot n_b^2) / (n_a^2 + n_b^2)$$

Testing "r" we distinguish three cases

$r < 10^{-9}$	Positive test; GCD is accepted
$r > 10^{-7}$	Negative test; GCD is rejected
$10^{-9} \leq r \leq 10^{-7}$	Ambiguous test; multiprecision need

In the previous example we get the following values giving a weighted residual of about

$$r \cong 2.1E-5$$

Therefore, the GCD found is (correctly) rejected

	a	b
$\ R\ $	1.96519	1.016E-12
$\ C\ $	88115.4	935.98
r	2.23E-05	1.086E-15
n	7	2

GCD - The Sylvester's matrix

The GCD polynomial can be obtained also by the so called Sylvester's matrix

Let's see a practical example

Given two polynomials $a(x)$ e $b(x)$ that we want to compute the GCD

$$a(x) = x^5 - 13x^4 + 63x^3 - 139x^2 + 136x - 48$$

$$b(x) = 5x^4 - 52x^3 + 189x^2 - 278x + 136$$

The polynomial degrees are $n_a = 5$ e $n_b = 4$. We build the following square matrix S_n of dimension $n = n_a + n_b = 9$.

5	-52	189	-278	136	0	0	0	0
1	-13	63	-139	136	-48	0	0	0
0	5	-52	189	-278	136	0	0	0
0	1	-13	63	-139	136	-48	0	0
0	0	5	-52	189	-278	136	0	0
0	0	1	-13	63	-139	136	-48	0
0	0	0	5	-52	189	-278	136	0
0	0	0	1	-13	63	-139	136	-48
0	0	0	0	5	-52	189	-278	136

In the first row, starting from the first column, we fill the coefficients of lowest degree polynomial $b(x)$, while in the second row the coefficients of the greatest degree polynomial $a(x)$. The next rows are built on the same way, shifting one column each time, till the right edge of the matrix (9 x 9) is reached. Now we transform it into an upper triangular matrix by any method that we want, for example, by the Gauss elimination algorithm. The reduced matrix will be as the following

1	-10.4	37.8	-55.6	27.2	0	0	0	0
0	1	-10.4	37.8	-55.6	27.2	0	0	0
0	0	1	-10.4	37.8	-55.6	27.2	0	0
0	0	0	1	-10.4	37.8	-55.6	27.2	0
0	0	0	0	1	-7.764	17.82	-11.06	0
0	0	0	0	0	1	-7.58	16.9	-10.32
0	0	0	0	0	0	1	-5	4
0	0	0	0	0	0	0	1E-14	5E-14
0	0	0	0	0	0	0	-1E-14	1E-14

As we note, the norm of the two last row is very small and then they can be totally eliminated.

Therefore, the matrix rank is $r = 7$ and its determinant is zero.

A theorem states that when the determinant of the Sylvester's matrix is zero, the polynomials $a(x)$ e $b(x)$ have a common divisor different from the trivial 1. On the contrary, if the determinant is not null, then the polynomials are relatively prime.

This method provides also the GCD coefficients in the last row of the reduced matrix. The global accuracy is comparable with those of the Euclid's method and the selectivity is light better

On the other hand, the computation cost is greater and the effort becomes relevant for high degree polynomials. An efficient and interesting variant of this method, but also more complex too, has been developed in order to obtain an accurate multiple roots structure of high degree polynomials¹

¹ "MultRoot - A Matlab package computing polynomial roots and multiplicities", Zhonggang Zeng, Northeastern Illinois University, 2003

The Derivatives method

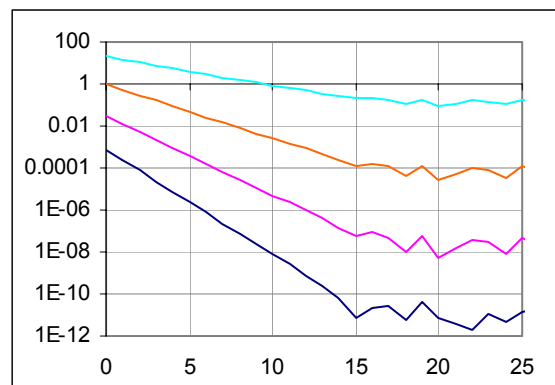
A less known, but very accurate method detects and reduces the root multiplicity by the derivatives evaluation. Conceptually speaking it is very simple: if the root $x = a$ of a polynomial $P(x)$ has multiplicity "m", then all its $m-1$ derivatives are zero in the same point $x = a$. That is:

$$P(a) = 0, P^{(1)}(a) = 0, P^{(2)}(a) = 0 \dots P^{(m-1)}(a) = 0 \quad (1)$$

We put in evidence that the derivatives of an integer coefficients polynomials still have integer coefficients. As no round-off errors are introduced, this method seems quite interesting. However, the relations (1) cannot be directly because they are simultaneously true only in the exact point of the root.

Usually we have not the exact root but only an approximation $x = a \pm \epsilon$. As we have shown, the barrier effect does not allow to approach the root beyond a minimum limit ϵ_{\min} . The graph shown the trajectories of a polynomial and its derivatives during a typical approach to a root of 4th multiplicity.

We note that after the 15th iteration all the curves stop to come down setting a minimum value greater than zero



The lowest curve is the polynomial $P(x)$ and, in increasing order, the derivatives P' , P'' , P''' .

As we can guess, it is a pure utopia to expect that the relations (1) may be verified.

In order to take advantage of the derivatives we have to adopt a different numerical approach.

The method, called "drill down", is articulated in consecutive steps. Let's suppose to have a root approximation¹. We define the following variables

r_0 , the best root approximation obtained with the polynomial $P(z)$

$D_0 = |P(r_0)|$, residual of $P(z)$ in r_0

$E_0 = |r_0 - x_0|$, initial root error estimation

We want to detect if the root has multiplicity $m > 1$. Repeat this process to the k^{th} derivative $P^{(k)}(z)$, defining in general:

r_k , the best root approximation obtained with the polynomial $P^{(k)}(z)$

$D_k = |P^{(k)}(r_k)|$, residual of $\|P^{(k)}(r_k)\|$

$E_k = |r_k - r_{k-1}|$, error estimation

The first test that we perform is for the multiplicity $m = 2$ ($k = 1$):

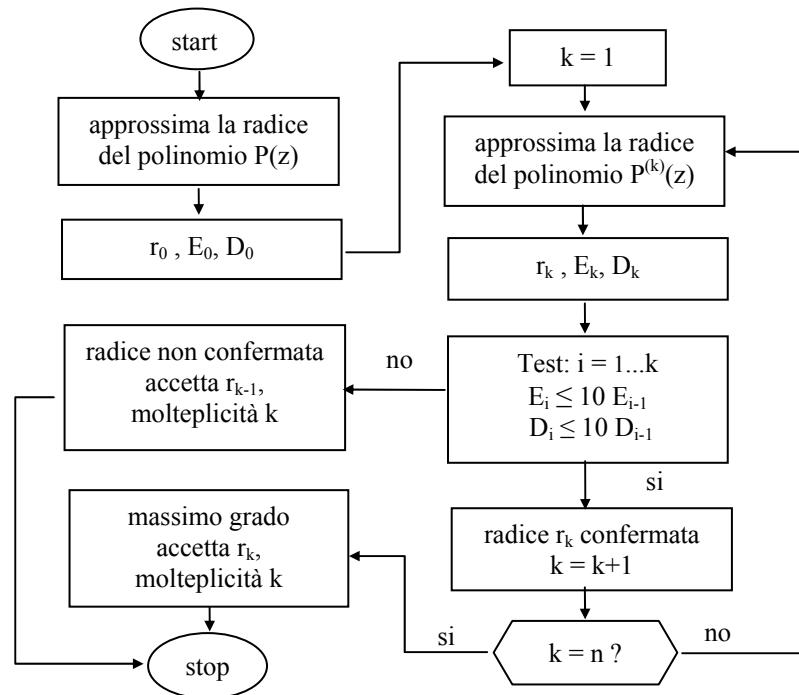
if $[(E_1 \leq 10 \cdot E_0) \cap (D_1 \leq 10 \cdot D_0)] \Rightarrow$ the root r_1 is confirmed with $m = 2$; else we take the root r_0 with $m = 1$.

The empirical coefficient 10 bypasses the numerical random noise near the machine limit ("bumping"). If the multiplicity 2 has been confirmed, we continue to apply the test to the next derivative $P^{(2)}(x)$ and so on. Generally the test for multiplicity $m = k+1$ is:

if $[(E_i \leq 10 \cdot E_{i-1}) \cap (10 \cdot D_i \leq D_{i-1}), i = 1 \dots k] \Rightarrow$ the root r_k is confirmed with multiplicity $m = k+1$; else, we take the root r_{k-1} with multiplicity $m = k$

¹ For example with the Newton method..

The following flow-chart explains, step-by-step, the drill-down process for multiple root refinement



We note that this test does not need any initial hypothesis about the root multiplicity, that it is accepted or rejected only to the end of the test itself. This process, that proceeds for "trials and errors", should be repeated for each root found but this is quite time consuming. But we have to put in evidence that is very important avoiding wrong multiplicity root, because this is reflected in the successive deflating process in a disastrous way. The reduced polynomial might be completely wrong. It is better to accept a root with inferior multiplicity than a root with wrong multiplicity. From this point of view, this method is quite safe. The computational cost is quite heavy but this method has been revealed robust and accurate at the same time.

Multiplicity estimation

We have seen that multiplicity testes are generally quite expensive; so we need a simpler criterion to decide when it is worth applying them.

The following method uses the sequential $\{x_n\}$ provided by the rootfinder algorithm itself. When "x" is sufficiently close to the roots "r" of multiplicity "m", we can approximate:

$$P(x) \cong k(x - r)^m$$

$$P'(x) \cong k \cdot m(x - r)^{m-1}$$

Dividing the first one for the second expression, we have

$$P(x) / P'(x) \cong (x - r) / m$$

$$\Rightarrow m \cdot \Delta x_n \cong r - x_n \tag{1}$$

where: $\Delta x_n = x_{n+1} - x_n = -P(x_n) / P'(x_n)$.

The relation (1) can be applied to consecutive values of the sequence.

$$m \cdot \Delta x_n \cong r - x_n$$

$$m \cdot \Delta x_{n+1} \cong r - x_{n+1}$$

removing r , we have the estimated value of m

$$m \cong -(x_{n+1} - x_n) / (\Delta x_{n+1} - \Delta x_n) = \Delta x_n / (\Delta x_n - \Delta x_{n+1}) \quad (2)$$

Note that, using the Newton or similar algorithms, the increments Δx_n are already computed; therefore the additional cost of the formula (2) is only 2 op for each iteration.

This formula gives reliable values only near the root and always just before the machine limit; in addition it can be easily tricked by clustered roots, and so, it should not be used as is, but only as a trigger for a more reliable multiple root test

For example consider the polynomial:

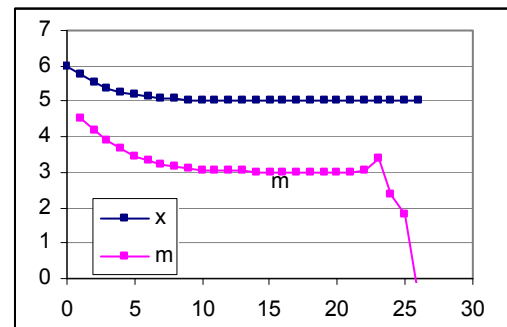
$$-1250 + 3000x - 3375x^2 + 2155x^3 - 818x^4 + 180x^5 - 21x^6 + x^7$$

Let's assume to have a real root bracketing in the interval $[4, 6]$

Applying the Newton's algorithm, starting from $x_0 = 6$, we have

n	x	P	P'	Δx	m	P
0	6	442	1704	-0.259		442
1	5.7406	142.96	708.16	-0.202	4.5097	142.96
2	5.5387	45.672	297.43	-0.154	4.1783	45.672
3	5.3852	14.404	126.26	-0.114	3.8901	14.404
4	5.2711	4.4856	54.137	-0.083	3.6532	4.4856
5	5.1882	1.3808	23.419	-0.059	3.4678	1.3808
6	5.1293	0.4209	10.204	-0.041	3.3286	0.4209
7	5.088	0.1273	4.4717	-0.028	3.2274	0.1273
8	5.0596	0.0383	1.9679	-0.019	3.1556	0.0383
9	5.0401	0.0115	0.8687	-0.013	3.1056	0.0115
10	5.0269	0.0034	0.3843	-0.009	3.0713	0.0034

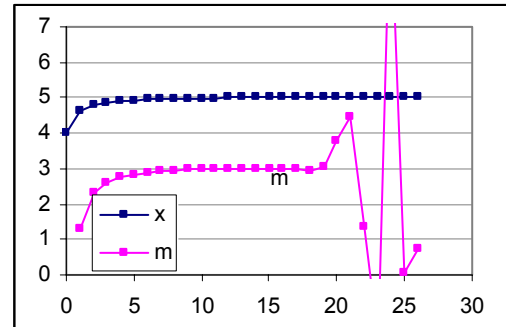
The graph shows the sequences of $\{x_n\}$ and $\{m\}$. After an initial transient, the curves converge to $x = 5$ and $m = 3$. The estimated multiplicity is 3. We note also that the final values of $\{m\}$ show large oscillations. This indicates that the machine limit is reached. In this situation further iterations are useless and the algorithm must be stopped. The last values of $\{m\}$, for $n > 20$, must be discharged.



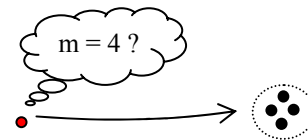
Setting a threshold of $m > 1.5$, after the initial 5 - 10 steps, the formula detects a multiple root and trigs the multiplicity drill-down test. In this way the true multiplicity test is called only when it need improving the efficiency of the entire rootfinder process.

Repeat now the rootfinder approximation, starting from $x_0 = 4$,

The graph shows the sequences of $\{x_n\}$ and $\{m\}$, starting from $x = 4$. After an initial transient, the curves converge to $x = 5$ and $m = 3$. Also in that case the multiplicity converges quickly after few iterations. Also here we note the final random oscillations due to the machine limit.

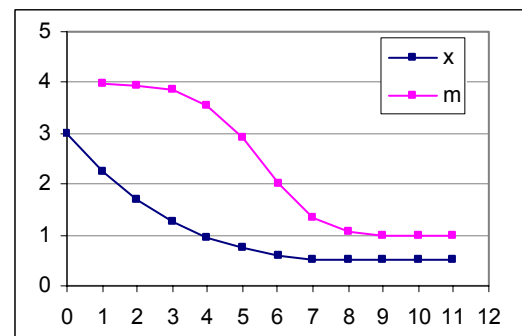


The multiplicity estimation formula is very simple and cheap, but it can easily fail. This happens, for example, when the sequence approaches a group of close roots (cluster roots)



At great distance, the formula "sees" the cluster as an only root having a bigger multiplicity; for example, if there are 4 single root, the "virtual" multiplicity appears about 4. As the distance becomes smaller, also the estimated multiplicity decreases consequently till to the unitary value near one of the four roots.

This phenomena can be easily evidenced . For example, the polynomial $P(x) = x^4 - 1/16$ has 4 root around a centered circle of radius 0.5. Start the Newton rootfinder from $x_0 = 3$ and trace the sequence of $\{x_n\}$ and the value $\{m_n\}$ returned by the estimation formula. As we can see, during the first iterations m is near 4 and decreases progressively to 1 as $\{x_n\}$ approaches the true root $r = 0.5$. For $x \cong 0.52$ (+4%), $m \cong 1.33$ (+33%)



In this situation the formula should give a "false trig" but, as we have explained, this is not a true problem because the multiplicity confirmation/rejection is performed by more reliable methods.

Integer Roots finding

Integer roots of a polynomial with integer coefficients, can be extracted with a simple and efficient way using the integer synthetic division. The reduced polynomial is virtually free of round-off errors so also the remaining decimal roots can greatly gain in accuracy.

The methods bases oneself on this entry: an integer root of a polynomial always divides exactly the constant term a_0 . Thus, we could find all the integer roots with a limited (hoping moderate) number of trials.

Let's see this example

$$P(x) = 16x^6 - 58x^5 - 169x^4 + 326x^3 - 230x^2 + 384x - 45$$

The exact divisors set of 45 is $Z_{45} = \{\pm 1, \pm 3, \pm 5, \pm 9, \pm 15, \pm 45\}$. The integer divisors must be taken with both signs¹. If the polynomial has an integer root, then it must be in this set.

Evaluating the polynomial for 12 values we are sure to find all its integer roots

$$x_i \in Z_{45}: P(x_i) \text{ for } i = 1, 2 \dots 12.$$

The computation is practically performed with the Ruffini-Horner algorithm that also gives the reduced polynomial coefficients if the test is positive. Starting from the lowest numbers, in increasing order and alternating the sign, we find the first integer root $x_1 = -3$.

		16	-58	-169	326	-230	384	-45
-3			-48	318	-447	363	-399	45
		16	-106	149	-121	133	-15	0

The reduced polynomial coefficients are in the last row. We apply again the process to the reduced polynomial, starting from the last integer root found. We find the new integer root $x_2 = 5$.

		0	16	-106	149	-121	133	-15
5			0	80	-130	95	-130	15
		0	16	-26	19	-26	3	0

Continuing we cannot find any more integer root; the only integer root found are $x_1 = -3$, $x_2 = 5$. Therefore the given polynomial can be factorized as

$$P(x) = P_1(x) (x + 3) (x - 5) = (16x^4 - 26x^3 + 19x^2 - 26x + 3) (x + 3) (x - 5)$$

We can apply to the residual polynomial $P_1 = 16x^4 - 26x^3 + 19x^2 - 26x + 3$ the usual rootfinding methods for finding the remaining roots.

Immediate benefits of the integer roots extraction process are clearly a degree reduction and an integer exact computation. This means that the remaining roots of the residual polynomial will be free of deflation round-off errors.

From a practical view, this method can be performed with two different algorithms: a "brute force attack" or an "intervals attack"

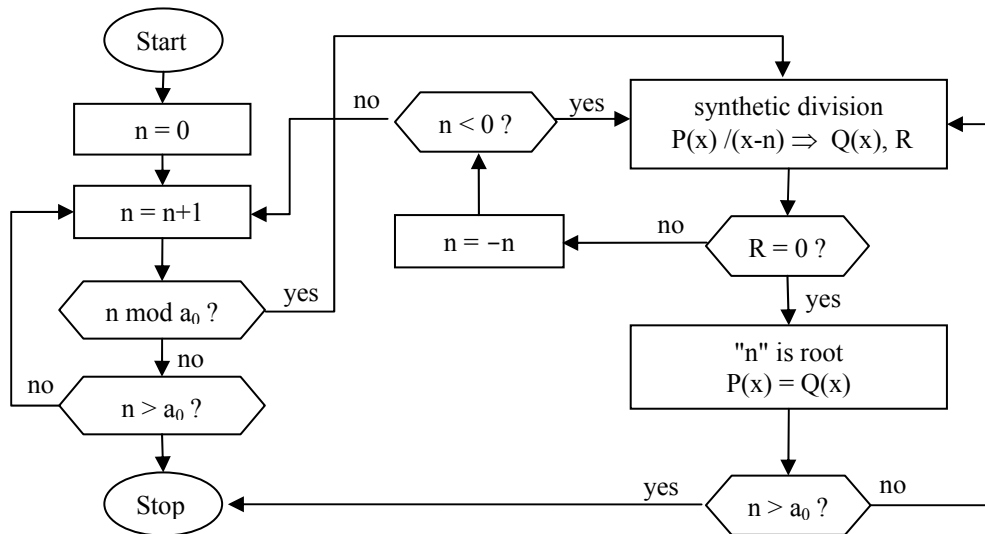
Brute Force Attack

The algorithm is conceptually simple and similar to that for prime numbers finding: each integer "n", between 1 and a_0 , is sequentially tested, alternating both positive and negative signs.

If $(n \bmod a_0) = 0$ then the synthetic division is performed $P(n)/(x-n)$.

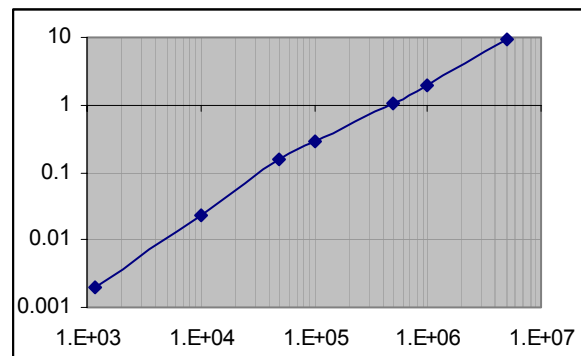
¹ A possible further reduction of the set can be made if the polynomial has all positive coefficients. In that case it is proved that all the roots stay in the negative half-plane and thus we can discharge all positive numbers.

If the remainder is zero ($R = 0$), then the quotient polynomial is substituted to the original polynomial and the process is repeated, starting again from the integer "n".



The searching time of this method grows about linearly with the minimum absolute integer root $|z_1|$. The following table and the graph show the elapsed time for several values of the magnitude $|z_1|$.

$ z_1 $	time (sec)
1200	0.002
10000	0.023
50000	0.156
100000	0.289
500000	1.031
1000000	1.953
5000000	9.273



We observe that, if the polynomial has no integer roots, the searching time grows with the constant terms $|a_0|$. As we can see, this method gives an acceptable response for $|z_1| < 10000$. For higher values the searching time becomes too long.

The computation effort is generally very high and it can be tolerated only for the relevant global accuracy gain. The computation cost depends on several factors such as the polynomial degree, the number of the integer roots, their spatial distributions, etc. Here we only give an estimation for finding the minimum root $|z_m|$, with $1 \leq |z_m| \leq |a_0|$.

The number " n_d " of exact divisors of a number "m" can be generally estimated by the formula $n_d \approx 10 \text{Log}_{10}(m/16)$. For each integer a modular test is performed (1 op) and for each divisor 2 synthetic division are performed ($3n+2$ op). So the total cost is about

$$C = [|z_m| + 20 \cdot (3n+2) \cdot \text{Log}_{10}(|z_m|/16)] \text{ op.}$$

For a 10^{th} degree polynomial having $|z_m| = 1000$, the cost is about 1610; if $|z_m| = 1\text{E}6$ the cost grows to about $1\text{E}6$. If the polynomial has no integer root the cost is evaluated setting $|z_m| = |a_0|$

Intervals Attack

For high degree polynomials or for big integer roots, the brute force attack becomes very long. To overcome this drawback the searching is restricted in adapt intervals containing one or more integer roots.

As we have seen, one of the better automatic algorithms for root locating is the QD.

For example, take the polynomial:

$$x^5 - 1840x^4 + 1064861x^3 - 224912240x^2 + 64952800x - 13712800000$$

Perform the first 4 iterations of the QD algorithm

n	d ₁	q ₁	d ₂	q ₂	d ₃	q ₃	d ₄	q ₄	d ₅	q ₅	d ₆
0	0	1840	-578.7	0	-211.2	0	-0.289	0	-211.1	0	0
1	0	1261.3	-168.6	367.52	-121.2	210.92	0.2887	-210.8	211.41	211.12	0
2	0	1092.6	-64.04	414.93	-97.12	332.43	0.0003	0.2893	-211.3	-0.289	0
3	0	1028.6	-23.77	381.85	-109.2	429.55	-1E-04	-211	211.32	211.03	0
4	0	1004.8	-7.012	296.38	-198.6	538.8	-7E-08	0.2891	-211.3	-0.289	0

We note that column q₁ converges to the real root $x = 1004.8 (\pm 70)$

This means that the root (the QD algorithm does not say if integer or not) is located in the interval $934 < x < 1075$.

Starting from $n = 934$, with only 66 trials, we can find the integer root $x = 1000$.

	1	-1840	1064861	-224912240	64952800	-13712800000
1000	0	1000	-840000	224861000	-51240000	13712800000
	1	-840	224861	-51240	13712800	0

In addition, as a sub-process, we have also the deflated polynomial coefficients

The estimated cost can be found calculating before the cost with $z_m = 1000$ and $z_m = 934$:

$$C_{1000} = [|1000| + 20(3.5+2) \cdot \text{Log}_{10}(|1000|/16)] \cong 1610 \text{ op}$$

$$C_{934} = [|934| + 20 \cdot (3.5+2) \cdot \text{Log}_{10}(|934|/16)] \cong 1534 \text{ op}$$

So the cost for searching into the interval $934 < x < 1000$ is about: $C_{1000} - C_{934} = 76 \text{ op}$

Adding the cost of 4 iterations of the QD, $4(4n-2) = 72 \text{ op}$, the total cost is about 150 op, that is about 10 times lower than the cost of the brute force attack from 1 to 1000 ($C_{1000} \cong 1610$)

Because the algorithm for integer root extraction is intrinsically free of round-off errors we can start from the biggest or the smallest root, indifferently

Algorithms

Zeros finding of polynomials is different from those of a general real function mainly for the definition domain that, in general, is the complex plane. Unfortunately, in that domain we cannot apply the concept of intervals bracketing used for real zeros. We can apply, in complex arithmetic, nearly all the one-point methods for general non-linear functions $f(x)$ such as Newton-Raphson, Halley, etc. The computational cost is much more than the standard arithmetic: we can estimate a complex operation¹ about 4÷5 times the cost of a standard operation.

Another important thing regards the derivatives that for a polynomials can be performed with high accuracy for any order and degree. Therefore, nearly all polynomial rootfinder algorithms largely exploit the derivatives.

Newton-Raphson method

This is far the most popular method for finding real and complex polynomial roots. It is a 2nd order method broadly used either in the automatic routine or in manual calculus

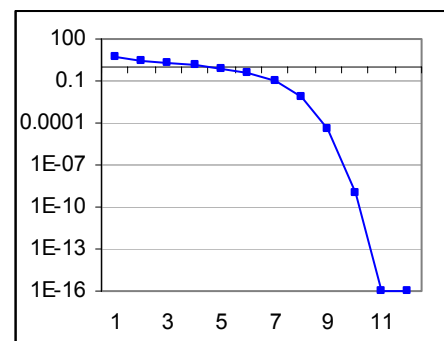
$$z_{i+1} = z_i - P(z_i) / P'(z_i)$$

For polynomials is almost used with complex arithmetic. Its computational cost is about $4(4n-1)$ for iteration. The algorithm implementation is quite simple with an electronic spreadsheet and the aid of a package performing complex calculus.²

A typical problem of the Newton method is a quite large sensibility to the starting point: this means that the convergence time exhibits large oscillations depending on the starting point. It is used to recommend to choose a starting point "sufficiently close" to the root but, as we have seen, this a quite ambiguous concept that can leads, in several cases, to unwanted situations. For example, we want to find the real root of the following polynomial, starting from $x_0 = -15$

$$x^3 + 4x^2 + 4x + 3$$

x	P(x)	P'(x)	error
-15	-2532	559	4.529517
-10.470483	-748.24755	249.12918	3.0034521
-7.4670309	-220.17781	111.53341	1.9740974
-5.4929335	-64.016999	50.573487	1.2658213
-4.2271122	-17.966597	23.788534	0.7552629
-3.4718493	-4.521207	12.386418	0.3650133
-3.106836	-0.8061411	8.1026018	0.0994916
-3.0073444	-0.0516807	7.0736055	0.0073061
-3.0000382	-0.0002677	7.0003824	3.824E-05
-3	-7.311E-09	7	1.044E-09



The graph shows the error trajectory $|x_i - r|$ versus the iterations. Typically the strong error reduction happens in the last 2÷3 iterations while the other iterations is spent in the root approaching.

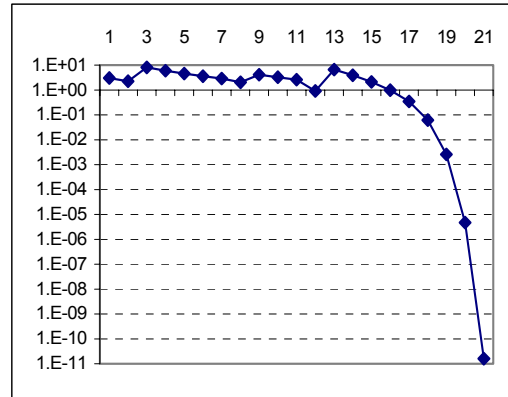
As we can note, the algorithm has approximated the root $r = -3$ with an error of about $1E-16$, in less than 10 iterations even if the starting point seems quite distant from the root.

Let's see, now, what happens if we start from the point $x_0 = -0.001$.

¹ Remember that our approach does not differentiate the operations

² The calculus has been performed here by MS Excel XP + the addin Xnumbers.xla v.4.5 (Foxes Team)

x	P(x)	P'(x)	error
-0.00100000	2.996004E+00	3.992003E+00	3.00E+00
-0.75150144	1.828598E+00	-3.177483E-01	2.25E+00
5.00336264	2.484004E+02	1.191278E+02	8.00E+00
2.91820419	7.358766E+01	5.289338E+01	5.92E+00
1.52695902	2.199451E+01	2.321048E+01	4.53E+00
0.57934779	6.854421E+00	9.641714E+00	3.58E+00
-0.13156536	2.540699E+00	2.999405E+00	2.87E+00
-0.97863290	1.979099E+00	-9.558961E-01	2.02E+00
1.09177951	1.343643E+01	1.631018E+01	4.09E+00
0.26797332	4.378375E+00	6.359216E+00	3.27E+00
-0.42053535	1.950887E+00	1.166267E+00	2.58E+00
-2.09329696	2.981779E+00	3.993008E-01	9.07E-01
-9.56079851	-5.435495E+02	2.017402E+02	6.56E+00
-6.86649441	-1.596176E+02	9.051428E+01	3.87E+00
-5.10304238	-4.613654E+01	4.129879E+01	2.10E+00
-3.98590198	-1.271963E+01	1.977503E+01	9.86E-01
-3.34268535	-3.026206E+00	1.077915E+01	3.43E-01
-3.06193915	-4.529939E-01	7.630901E+00	6.19E-02
-3.00257604	-1.806551E-02	7.025780E+00	2.58E-03
-3.00000473	-3.309246E-05	7.000047E+00	4.73E-06
-3.00000000	-1.117457E-10	7.000000E+00	1.60E-11



In that case, the Newton algorithm seems to spend the most part of the time in a chaotic search of an attack point.

This shows that a starting point closer to the root then the previous one can gives unwanted worse results. Paradoxical, the point $x_0 = -100$, very far from the root, converges in less then 20 iterations with no problems, while, from $x_0 = 0$, the convergence fails

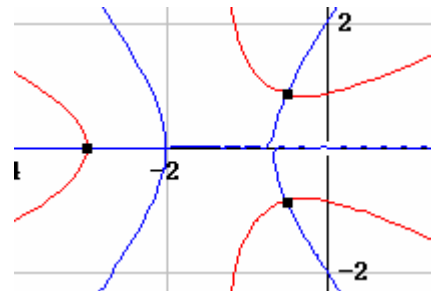
In order to converge at a complex root, the Newton method must be started from a complex root. For example, we want to find the complex roots of $x^3 + 4x^2 + 4x + 3$

First of all, we have to localize the roots. We use the graphical method plotting the curve implicitly defined by the following equation:

$$x^3 + 4x^2 + x(4 - 3y^2) - 4y^2 + 3 = 0$$

$$y(3x^2 + 8x - y^2 + 4) = 0$$

A possible starting point is $z_0 = -1+i$



i	z		P(z)		P'(z)		dz	
0	-1	1	1	-2	-4	2	-0.4	0.3
1	-0.6	0.7	0.746	-0.147	-1.19	3.08	-0.12295345	-0.19470305
2	-0.47704655	0.89470305	-0.1628118	0.058920035	-1.5351328	4.596734385	0.022173388	0.028013954
3	-0.49921994	0.866689095	-0.00404623	0.002385261	-1.49954782	4.33750191	0.000779278	0.000663439
4	-0.49999921	0.866025656	-2.2697E-06	3.02105E-06	-1.49999739	4.330132359	7.85053E-07	2.52223E-07
5	-0.5	0.866025404	3.53495E-13	2.42584E-12	-1.5	4.330127019	4.7495E-13	-2.4616E-13
6	-0.5	0.866025404	-4.4409E-16	0	-1.5	4.330127019	3.17207E-17	9.15697E-17

Few iterations are sufficient for finding the complex roots $z = -0.5 \pm i 0.866$ with an excellent accuracy. This algorithm can also converge to a real root starting from a complex starting point as for example: $z = -4 + i$

i	z		P(z)		P'(z)		dz	
0	-4	1	-5	19	17	-16	-0.71376147	0.44587156
1	-3.28623853	0.55412844	-0.63781703	5.431081824	9.187007828	-6.49296187	-0.3249332	0.361521729
2	-2.96130533	0.192606711	0.444614531	1.267438413	6.506253135	-1.88135	0.011080673	0.198007211
3	-2.97238601	-0.0054005	0.189649754	-0.0363244	6.726060173	0.053110224	0.028151863	-0.00562284

4	-3.00053787	0.000222339	-0.00376629	0.00155757	7.005379421	-0.00222411	-0.0005377	0.000222168
5	-3.00000017	1.70709E-07	-1.199E-06	1.19496E-06	7.000001713	-1.7071E-06	-1.7128E-07	1.70709E-07
6	-3	4.17699E-14	0	2.92389E-13	7	-4.177E-13	-2.4925E-27	4.17699E-14
7	-3	0	0	0	7	0	0	0

The inverse is not true. Starting from a real point, the algorithm never converge to a complex root. Therefore, a general starting point should be necessary complex..

The starting point

The convergence speed of the Newton algorithm is largely influenced by the starting point $P_0(x, y)$. There are points of which the convergence is extremely fast (6÷8 iterations); others point show a very slow convergence and other for which the convergence fails. In the past several studies was made in order to analyze the convergence region of the iterative algorithms. In particular, the studies about the Newton method have led, as known, to the wonderful results of the fractal geometry explored in the years '80 with the aid of supercomputers and analyzed by pioneers as Mandelbrot, Hubbard an many other.

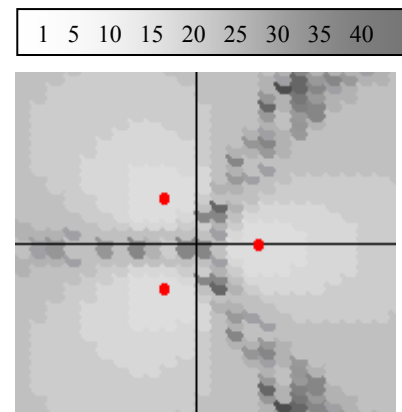
In this subject we are interested only to visualize the "good" and "bad" convergence region of the Newton algorithm.

The Convergence Spectrum

For that purpose we choose a simple test polynomial as for example z^3-1 . Then, we apply the Newton root finding method starting from the value $z_0 = x + i y$, where $P(x, y)$ is a generic point of the complex plane. We count the iterations need for the convergence, relating an arbitrary grey scale to the iteration number.

Thus, each point $P(x, y)$ is related to a given iterations number and then, to a grey tonality, more or less intense with the convergence difficulty.

Repeating the process for many points, a sort of spot configuration called *convergence spectrum* begins to appear

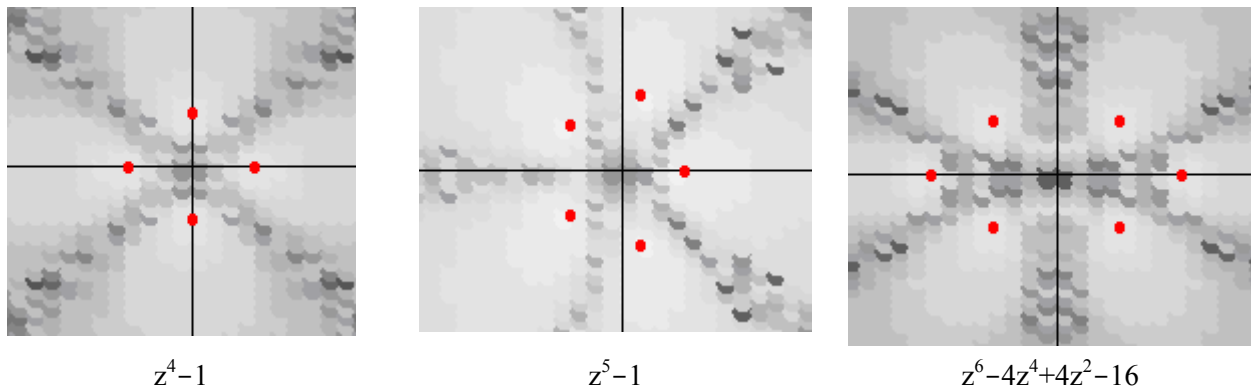


The characteristic is the non-defined borders: they shows continuously light and dark spots apparently mixed in a chaotic way. The three directions where the chaotic regions are more concentrated are the 1st and 4th bisectors and the negative x-axes. We note also that the central region, inside the root circle, exhibits an high "turbulence", that means a very low efficiency or even an higher probability of fail. On the contrary, starting from point outside the circle roots means a good probability of better performance speed.

This analysis regards a central polynomial but the considerations are the same for a general polynomial because it can be always transformed into a central polynomial by shifting

We wonder if these light and dark regions exist also for other polynomials. The answer is affirmative: not only they exists but the "turbulence" directions increase with the degree and the inner region of the roots shows more chaotic and intricate configurations

The following picture show the convergence spectrums of three different polynomials

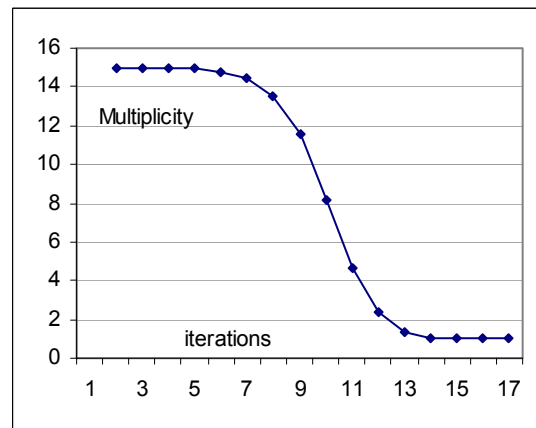
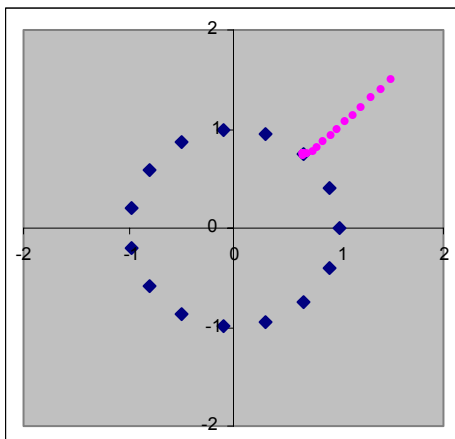


In order to avoid the "dangerous" inner region of the roots, it is convenient to start the rootfinding algorithms with a complex numbers external a circle having a radius 2 ÷ 3 times the radius of the roots. A drawback of this tactic is that the roots with higher magnitude will be extracted at the first and therefore it is better to adopt the reciprocal deflation to reduce the error propagation.

Multiplicity and clustering

In the previous pages we have shown the multiplicity estimation formula for the Newton' method: $m \cong \Delta x_n / (\Delta x_n - \Delta x_{n+1})$ and we have seen that it gives acceptable values only very near to the root; far from the root, this formula gives super estimations emphasized by the presence of other near single roots (clustering). Let's see this example..

Find a root of the polynomial $z^{15} - 1$, starting from $z_0 = (1.6, 1.6)$



The algorithm begins to approach as it were a single "virtual" root, having multiplicity 15, located in the center; only near the true root the influence of the other roots disappears and the formula gives a better estimation. The "clustering" effect influences also the convergence speed: very slow during the approaching iterations and faster in the last iterations .

Halley method

This efficient and accurate method of 3rd order uses the iterative formula

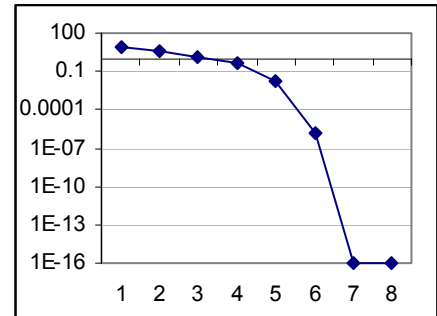
$$x_{i+1} = x_i - P(x_i) \cdot P'(x_i) / ([P'(x_i)]^2 - 0.5 \cdot P(x_i) \cdot P''(x_i))$$

Its computational cost, for a polynomial of degree "n" is about : $4(6n+1)$ for iteration.

Let's see how it works. For example we want to find the real root of the polynomial :

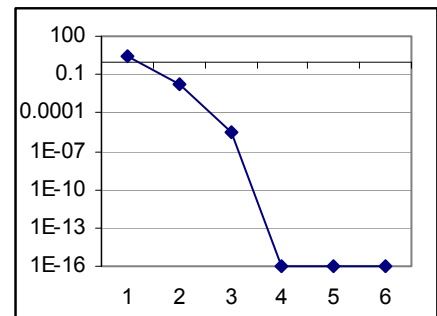
$$x^3 + 4x^2 + 4x + 3, \text{ starting from } x_0 = -15.$$

i	x	P(x)	P'(x)	P''(x)	dx
0	-15	-2532	559	-82	6.782933737
1	-8.217066263	-314.605	140.8240	-41.30239	3.322527165
2	-4.894539098	-38.00819	36.71322	-21.36723	1.481640878
3	-3.41289822	-3.813105	11.6404	-12.47738	0.397330795
4	-3.015567425	-0.110187	7.156401	-10.09340	0.015566067
5	-3.000001358	-9.50E-06	7.000013	-10.00000	1.35797E-06
6	-3	0	7	-10	0



In only 6 iterations this algorithm has found the root with the lowest possible error in standard precision (1E-16), with a global cost of 114 op. The Newton algorithm in the same condition has required 10 iterations, with a cost of 110 op. Thus, in spite of its faster cubic convergence, the Halley algorithm cannot overcome the efficiency of the Newton method. This is generally true, at least, in standard precision. For higher precision the cubic convergence make the Halley method more efficient. Practical experiences have shown that Halley method is much more stable than Newton. In fact, we see how it works in the critical point $x_0 = -0.001$.

x	P(x)	P'(x)	P''(x)	dx
-0.001	2.996003999	3.992003	7.994	-3.0194082
-3.020408201	-0.14494838	7.20533149	-10.1224492	0.020405161
-3.00000304	-2.1282E-05	7.0000304	-10.0000182	3.04028E-06
-3	0	7	-10	0



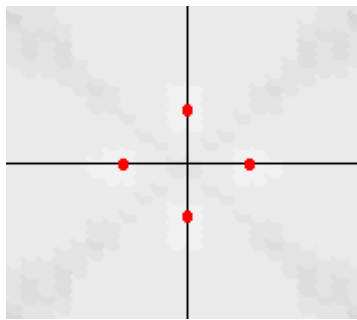
Only 4 steps for reaching the best possible accuracy in standard precision (1E-16)

It is remarkable that in the same situation the Newton method has spent 20 iterations and has failed for $x_0 = 0$. Here, on the contrary, for $x_0 = 0$ we get the exact root in a single step! Clearly the stability performance of the Halley method are superior. The stability and the cubic convergence make very interesting this method for polynomial rootfinding routine.

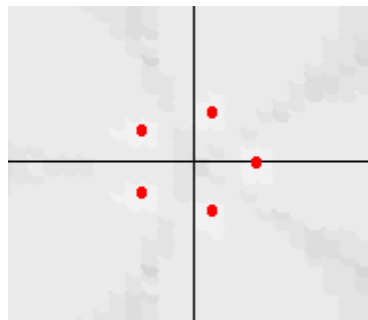
Starting from a complex value this method can converge to a complex root. In fact, if we choose the starting value $z_0 = -1+i$, we have

z		P(z)		P'(z)		P''(z)		dz	
-1	1	1	-2	-4	2	2	6	0.541401	-0.159235
-0.458599	0.8407643	0.05539	0.21461	-1.15850	4.41267	5.24840	5.0445	-0.04143	0.0252959
-0.500034	0.8660603	-9.97E-05	-0.0002	-1.50035	4.33012	4.99979	5.1963	3.41E-05	-3.487E-05
-0.5	0.8660254	2.11E-13	6.66E-14	-1.5	4.33012	5	5.1961	1.36E-15	4.834E-14
-0.5	0.8660254	-4.44E-16	0	-1.5	4.33012	5	5.1961	-3.17E-17	-9.157E-17

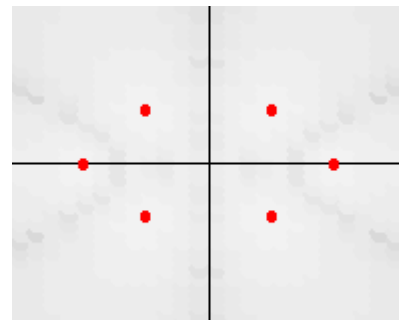
The following picture show the convergence spectrums of the Halley algorithm for three different polynomials



z^4-1



z^5-1



$z^6-4z^4+4z^2-16$

Clearly , the dark spots are here largely reduced : This confirms that the Halley method is very stable respect to the starting point.

Multiplicity Estimation

Also for this algorithm is possible to estimate the root multiplicity by the approaching sequence. In that case the estimation formula is:

$$m = (\Delta x_n + \Delta x_{n+1}) / (\Delta x_n - \Delta x_{n+1}) \quad , \quad \text{dove } \Delta x_n = (x_n - x_{n-1})$$

Also here, this formula should be regarded with the cautions already taken for Newton method

Lin-Bairstow method

This is a well known popular method used in many rootfinder routines for real polynomial, appreciated by its features of practicality, efficiency and simplicity. It seek for complex and real roots without using the complex arithmetic. In addition, it automatically generates the reduced polynomial as sub process. All that allows to implement a very compact and efficient code. This is enough to make this method one of the most preferred by programmers.

The Bairstow method is based on the fact that a real polynomial $P(x)$ can be factorized in linear and quadratic factors. The process starts with an approximated estimation of the quadratic factors $D(x) = (x^2 - ux - v)$, where "u" e "v" are the unknowns. Then, it computes the remainder of the division $P(x) / D(x)$ between the two polynomials. Generally, the remainder $R(u, v)$, function of both the variable "u" e "v", will not zero. The heart of the Lin-Bairstow algorithm consists on the iterative process to adjust the values "u" e "v" in order to have a zero remainder. In this way the approximated quadratic factor is transformed in an exact factor that can be easily solved by the 2nd degree equation formula. The process is applied again to the deflated polynomial. At each loop a couple of roots is found and the polynomial degree decreases by 2. The process ends when the remainder becomes a linear or quadratic polynomial (1st o 2nd degree).

According to this process, the general real polynomial, can be factorized as

$$P(x) = a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n = Q(x) \cdot (x^2 - ux - v) + R(u, v)$$

where: $R(u, v) = b_{n-1} x + b_n$, $Q(x) = b_0 x^{n-2} + \dots + b_{n-4} x^2 + b_{n-3} x + b_{n-2}$

The quadratic factor $(x^2 - ux - v)$ is exact, if both the remainder coefficients are zero. So we try to modify "u" e "v" so that $b_{n-1} = b_n = 0$. Setting $u = u_0$, $v = v_0$, as initial values, The algorithm starts with the following iterative formulas:

$b_0 = a_0$	$c_0 = b_0$
$b_1 = a_1 + u b_0$	$c_1 = b_1 + u c_0$
$b_2 = a_2 + u b_1 + v b_0$	$c_2 = b_2 + u c_1 + v c_0$
.....
$b_{n-1} = a_{n-1} + u b_{n-2} + v b_{n-3}$	$c_{n-2} = b_{n-2} + u c_{n-3} + v c_{n-4}$
$b_n = a_n + u b_{n-1} + v b_{n-2}$	$c_{n-1} = b_{n-1} + u c_{n-2} + v c_{n-3}$

The increments Δu , Δv are computed using the values b_n , b_{n-1} , c_{n-1} , c_{n-2} , c_{n-3}

$$\Delta u = \frac{b_n c_{n-3} - b_{n-1} c_{n-2}}{(c_{n-2})^2 - c_{n-1} c_{n-3}} \qquad \Delta v = \frac{b_{n-1} c_{n-1} - b_n c_{n-2}}{(c_{n-2})^2 - c_{n-1} c_{n-3}}$$

Thus, the new values becomes: $u + \Delta u$, $v + \Delta v$

The process repeats until $|\Delta u| + |\Delta v| < \epsilon$, where ϵ is a prefixed small number.

We note the vector $\mathbf{b} = [b_0 \dots b_i \dots b_n]$ contains the coefficients of the deflated polynomial; so, the process continues simple assigning the vector \mathbf{b} to the vector \mathbf{a} .

The Bairstow algorithm is based on the Taylor expansion of 2nd degree applied to the polynomial $R(u, v)$; thus the convergence speed is similar to the Newton method.

Experimental tests have shown that, generally, the convergence is neither monotonic nor regular. On the contrary, the increments Δu e Δv can exhibit large oscillation before to converge.

The computational cost is very low. In fact we have:

(coeff. b comp.) + (coeff. c comp.) + (increments comp.) = $4(n+2) + 4(n-6) + 13 = 8n+9$ for iterations. In addition the deflation cost is already enclosed. We can easily guess that the Bairstow methods is one of the more efficient general purpose rootfinder method for real polynomials.

Let's see now how it works. For example, find the roots of the equation test: $x^3 + 4x^2 + 4x + 3$, taking as initial values $(u_0, v_0) = (0, 0)$

The coefficients "b" e "c" are showed, alternated, in the following table

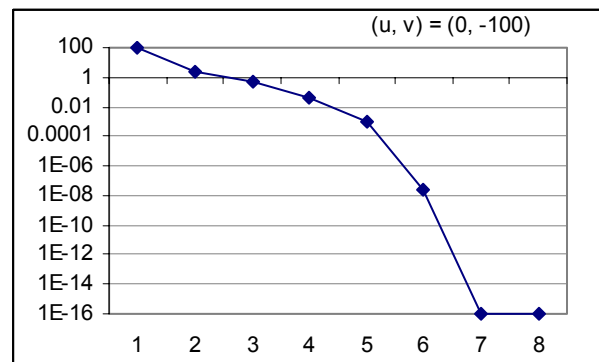
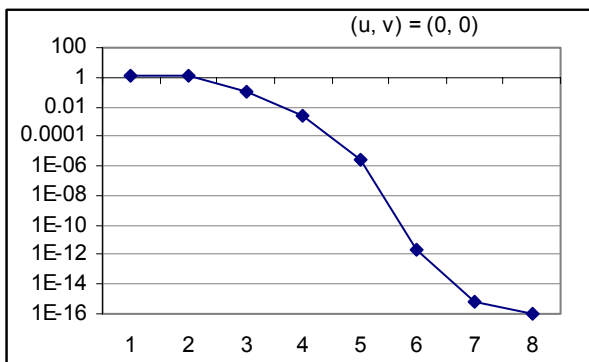
Iter		x^3	x^2	x^1	x^0	u	v	Δu	Δv	error
	a	1	4	4	3	0	0			
1	b	1	4	4	3					
	c	1	4	4	3	-1.0833333	0.33333333	-1.08333	0.333333	1.13E+00
	b	1	2.91666	1.17361	2.7008					
2	c	1	1.83333	-0.4791	3.83101	-0.9403255	-1.1024588	0.143008	-1.43579	1.44E+00
	b	1	3.05967	0.0204	-0.3923					
3	c	1	2.11934	-3.0748	0.16250	-0.9979133	-1.0008615	-0.05759	0.101597	1.17E-01
	b	1	3.00208	0.00331	-0.0079					
4	c	1	2.00417	-2.99753	0.97739	-0.9999989	-0.9999979	-0.00209	0.000864	2.26E-03
	b	1	3.00000	4.35E-06	7.66E-07					
5	c	1	2.00000	-2.9999	0.99999	-1	-1	-1.1E-06	-2.1E-06	2.37E-06
	b	1	3	1.2E-12	6.0E-12					
6	c	1	2	-3	1	-1	-1	4.01E-13	-2.3E-12	2.32E-12
	b	1	3	-1.1E-15	0					
7	c	1	2	-3	1	-1	-1	3.17E-16	4.76E-16	5.72E-16

In only 7 iterations this algorithm has found the quadratic factor $(x^2 + x + 1)$ with an excellent accuracy (1E-16) and, in addition, has given the deflated polynomial $(x + 3)$ in the last row of "b". Solving the equation: $x^2 + x + 1 = 0$ and $x + 3 = 0$, we find the roots:

$$x_1 = -0.5 + \sqrt{3}/2, \quad x_2 = -0.5 - \sqrt{3}/2, \quad x_3 = -3.$$

Practically, all the roots are found in only 7 iterations with a computation cost of about $7(24+9) = 161$ op. Solving the same polynomial by the Newton method, that is one of the very efficient method, would cost about 275 op

These graphs show the error trajectories for two different starting points: $(0,0)$, $(0,-100)$



As we can see, in both cases the convergence is reached in less than 8 iterations although the starting points are very different. and that the second starting value $(0,-100)$ is very far from the exact solution $(-1, -1)$.

But is the convergence always so fast and robust?. Unfortunately not. Like the Newton method, the convergence of the Bairstow algorithm depends on the starting point and the polynomial itself.

Also here the usual recommendation "sufficiently closer..." is very common in many books. For this algorithm, as we will see, besides being ambiguous it could be totally useless.

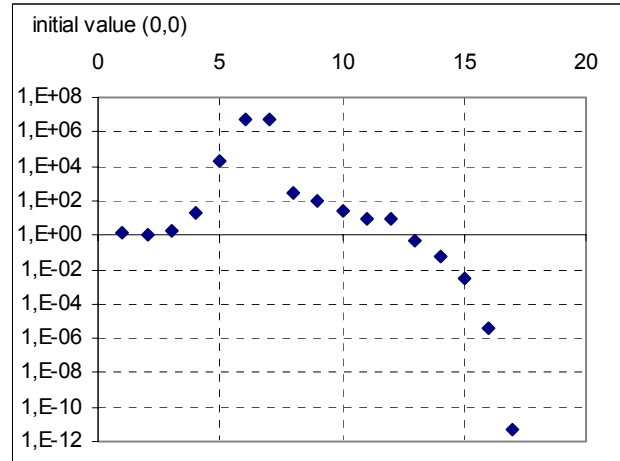
Let's see.

Solve the polynomial $x^3 + 3x^2 + 3x + 2$ starting from the point $(0, 0)$

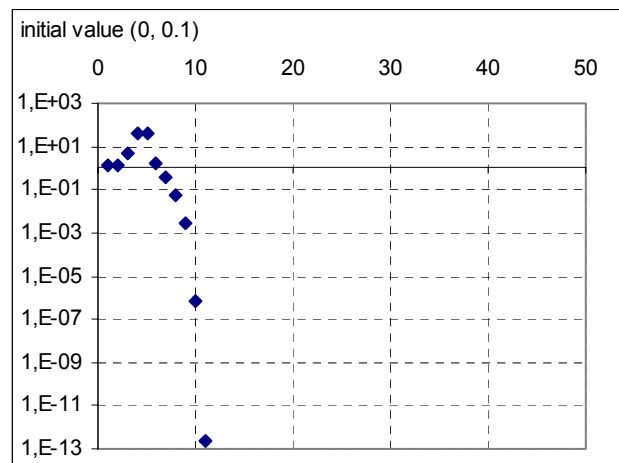
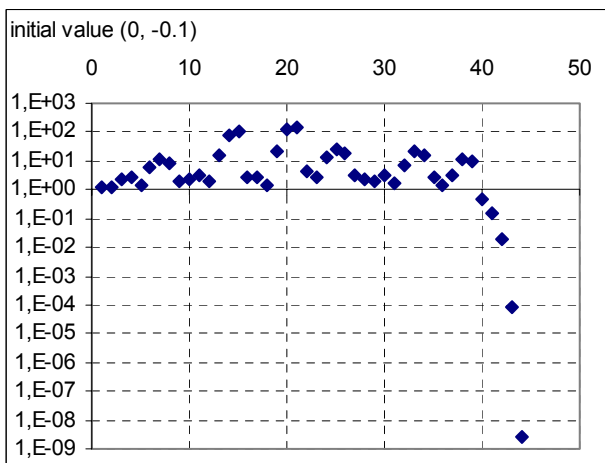
In that case, the trajectory is completely different. After few iterations, the algorithm begins to diverge largely reaching the maximum of about $1E6$ at the 7th iteration. Then, suddenly, the algorithm inverts the trend and begins to converge quickly after the 15th iteration to the exact solution $(-1, -1)$.

Practically, all the rootfinding job is performed in the last 3 iterations. The rest of the time, the algorithm has "walked" around the complex plane

This phenomenon is typical of this algorithm that exhibits a convergence even more wilder than those of the Newton method



This behavior is more evident choosing two closer starting points such as $(0, -0.1)$ and $(0, 0.1)$

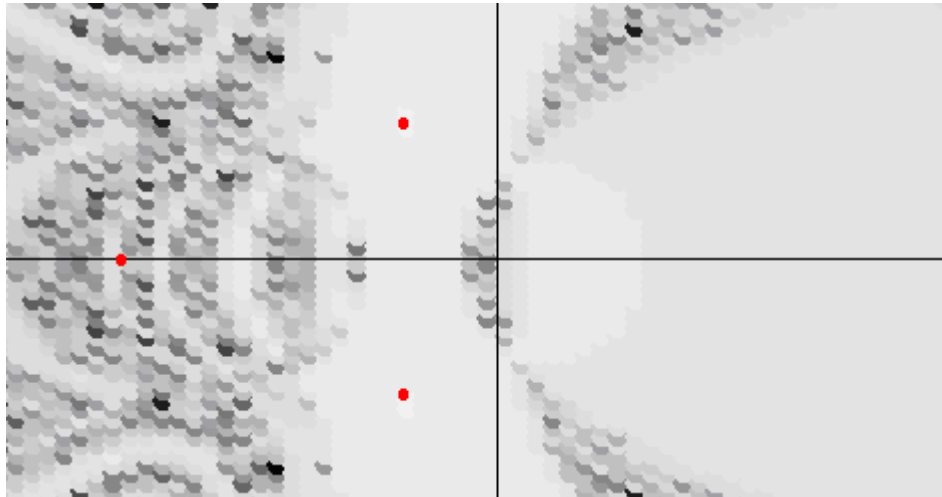


These graphs are quite eloquent. Starting from two very close initial values, the algorithm shows a completely different behavior: in the first case it needs more than 40 iterations for reaching the convergence, while in the second case it takes only 10 iterations; and, surprisingly, if we choose a very distant value, such as $(0, -100)$, the algorithm takes no more than 8 iterations.

Many studies are made for overcoming this intrinsically instability, but with not much success.

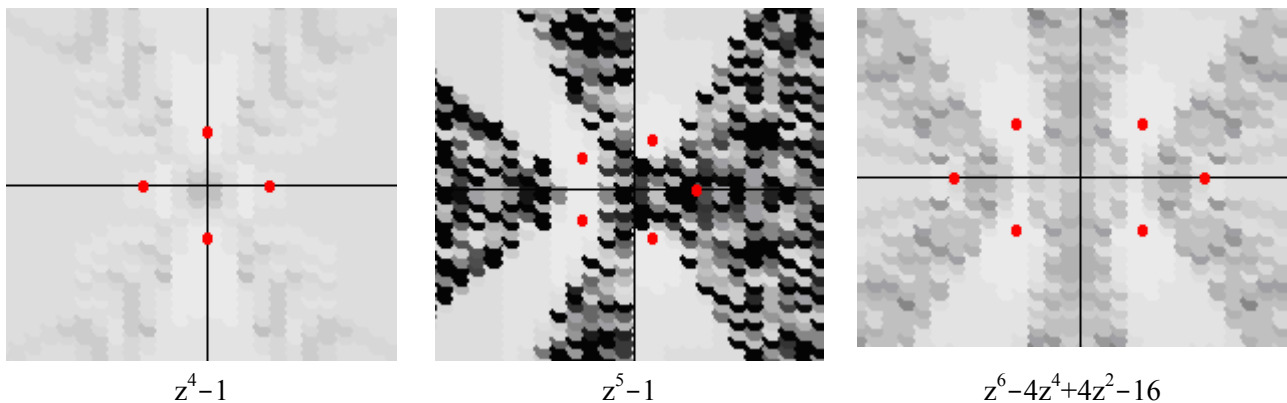
Perhaps the only true drawback of this algorithm is just this unpredictable convergence trajectory

It is interesting to see the convergence spectrum of this polynomial $x^3 + 3x^2 + 3x + 2$ for better explaining the convergence behavior.



The roots (red) are the points $(-0.5, 0.866)$, $(-0.5, -0.866)$, $(-3, 0)$. We see that the starting value $(0, -0.1)$ corresponding with the points $(0, 0.316)$, $(0, -0.316)$ falls in the dark "bad" region near the y-axis, while the starting value $(0, 0.1)$ corresponding with the points $(0.316, 0)$, $(-0.316, 0)$ falls in the light "good" region.

Others convergence spectrums for three different polynomials as



These pictures confirm what we have already intuited: the marked contrast between dark and light spots puts in evidence a remarkable instability of the Bairstow algorithm

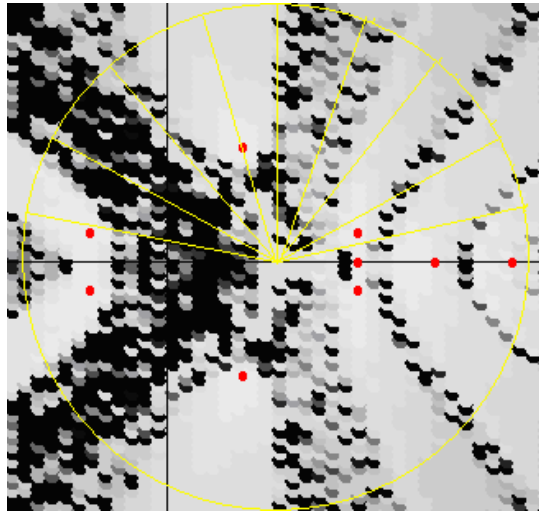
Observing the spectrum of $z^5 - 1$, we note that the only real root $x = 1$ is completely surrounded by dark spots. This means that the algorithm will never converge to this root. This always happens for odd degree polynomials when the numbers of roots is odd.

We note also that the inner area of the roots has an high "turbulence"

A simple and cheap method for the Bairstow algorithm starting chooses a random couple (u, v) in the range $-1 \leq u \leq 1$, $-1 \leq v \leq 1$. Together with the polynomial central shifting, this method, unless pathological cases, gives a reasonable probability of successes, (failure less than 5 %) for polynomial degree $n < 7$. For higher degree the failure probability increases much more. For example, the polynomial

$$-819000 + 195400x + 109270x^2 - 37748x^3 - 9169x^4 + 8519x^5 - 2510x^6 + 388x^7 - 31x^8 + x^9$$

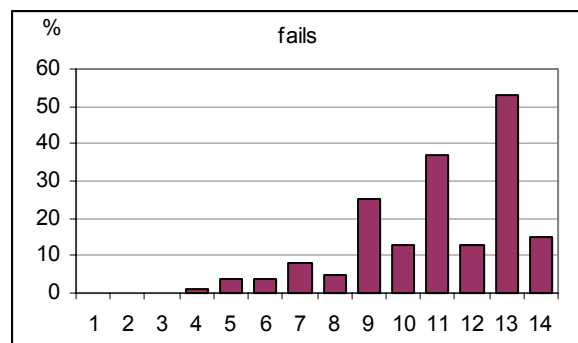
has the roots: $-2 \pm i$, $2 \pm 4i$, $5 \pm i$, $5, 7, 9$; its convergence spectrum and is shown in the following picture together with its roots circle.



We can see a large amount of dark spots inside and outside the roots circle. This means an higher probability of failure, in that case of about 25%.

For higher degree the convergence is more difficult. In the bar-graph has shown the failure probability versus the polynomial degree. The statistic test was performed with random polynomials with variable degree $2 < n < 15$ and mixed roots, real and complex, having $|z| < 200$.

Quaintly we note that odd polynomials are more difficult than even polynomials for this algorithm.



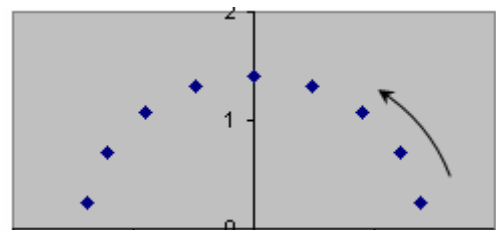
A method for improve the convergence stability without making heavy the algorithm is the following: if the convergence is not reached until a given number of iterations (30 for example), we performs a sort of systematic "fan-shaped" attack

Setting $R =$ roots radius, $xc =$ roots center,
 $n =$ degree, $\varphi = \pi/n$, we try successively the following starting points, for $k = 0, 1 \dots n-1$

$$x_k = R \cdot \cos(k \varphi + \varphi / 2) + xc$$

$$y_k = R \cdot \sin(k \varphi + \varphi / 2)$$

$$u = 2 \cdot x_k \quad , \quad v = -(x_k^2 + y_k^2)$$



For each point, we test the convergence until 30 iterations; then we continue with the next point, and so on. Generally after n trials, the algorithm can find a "channel" to reach a root with an high probability (typically more than 99%).

Siljak method

This a clever variant of the Newton-Raphson developed for complex polynomial without using complex arithmetic. Thanks to some tricks, it is a very efficient, stable and compact algorithm.

Here we show its main aspects.

Take the general complex polynomial of "n" degree:

$$P(x + iy) = \sum_{k=0}^n (a_k + ib_k)(x + iy)^k$$

This polynomial can be split into two real functions: $P(x + iy) = u(x,y) + i v(x, y)$

The Siljak method computes the polynomials $u(x, y)$, $v(x, y)$ and their partial derivatives du/dx , dv/dy by the following iterative algorithm.

Setting: $xs(0) = 1$, $ys(0) = 0$, $xs(1) = x$, $ys(1) = y$, $m = x^2 + y^2$, $t = 2x$
for $k = 2, 3 \dots n$

$$xs(k) = t \cdot xs(k-1) - m \cdot xs(k-2)$$

$$ys(k) = t \cdot ys(k-1) - m \cdot ys(k-2)$$

The coefficients $xs(k)$ e $ys(k)$, called Siljak's coefficients, are respectively the real and imaginary part of the power $(x + iy)^k$

Setting: $u = 0$, $v = 0$, for $k = 0, 1, 2 \dots n$

$$u = u + a(k) \cdot xs(k) - b(k) \cdot ys(k)$$

$$v = v + a(k) \cdot ys(k) + b(k) \cdot xs(k)$$

Setting: $p = 0$, $q = 0$, for $k = 1, 2 \dots n$

$$p = p + k \cdot [a(k) \cdot xs(k-1) - b(k) \cdot ys(k-1)]$$

$$q = q + k \cdot [a(k) \cdot ys(k-1) + b(k) \cdot xs(k-1)]$$

Finally, the increment $\Delta z = -P(z)/P'(z) = \Delta x + i \Delta y$, is computed by the formulas

$$\Delta x = - (u \cdot p + v \cdot q) / (p^2 + q^2) \quad , \quad \Delta y = (u \cdot q - v \cdot p) / (p^2 + q^2)$$

The performances of speed and convergence are similar to those of the Newton method. However, in order to improve the global convergence stability, the Siljak algorithm checks the convergence of the module $e^{(i)} = |P(z_i)|$ at each iteration. If the value $e^{(i)}$ at the i -th step is much more than the previous value $e^{(i-1)}$, then the increments Δz is halved and the algorithm loops until the value $e^{(i)}$ is lower or equal than the previous value.

This simple clever trick assures a good stability. This method has always overcomes our convergence tests for polynomial up to 14 degree using only the simple fixed starting point (0.1 , 1).

Laguerre method

A very good polynomial rootfinding algorithm is the Laguerre method.

For its heavy complexity, it is generally avoided in the manual calculus but its great performances of stability, robustness, accuracy and speed make this algorithm one of the most preferred for automatic rootfinding routines.

The Laguerre' method is based on the iterative formula:

$$z_{i+1} = z_i - \frac{n P(z_i)}{P'(z_i) \pm \sqrt{H(z_i)}} \quad (1)$$

where the complex function $H(z)$ is give by :

$$H(z) = (n-1)[(n-1)P'(z)^2 - n P(z) P''(z)]$$

The denominator of (1) should be chosen to minimize the increment $|dz|$; because, generally, both denominator can be complex, we choose the one with maximum module. The square root may be complex if $H < 0$. Generally the Laguerre formula requires complex arithmetic also starting from a real point.

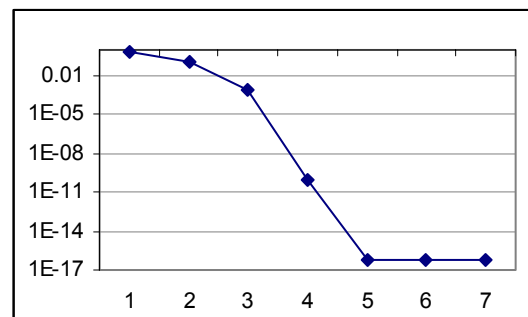
For example, find the root of the polynomial z^4+1 , starting from $z_0 = 1$. Note that this polynomial has only complex roots: $z = \pm\sqrt{2}/2 \pm i\sqrt{2}/2$, and that the starting point is real. The algorithm converges to the complex root in less then 5 iterations with the maximum accuracy possible (1E-16). The computation complexity suggests to use several auxiliary variables for a better comprehension.:
 $d_1 = P' + \sqrt{H}$, $d_2 = P' - \sqrt{H}$, $d = \max(d_1, d_2)$, $dx = -n \cdot P/d$

Variable.	iter 1	iter 2		iter 3		iter 4		iter 5		
x	1	0	0.8	0.6	0.707614	0.706599	0.707107	0.707107	0.707107	0.707107
P	2	0	0.1568	0.5376	4.12E-06	0.002871	0	3.7E-10	0	2.78E-16
P'	4	0	-1.408	3.744	-2.82233	2.834512	-2.82843	2.828427	-2.82843	2.828427
P''	12	0	3.36	11.52	0.017229	11.99999	2.22E-09	12	1.78E-15	12
H	-144	0	-40.32	-138.24	-0.20675	-144	-2.7E-08	-144	-1E-12	-144
\sqrt{H}	1.94E-14	12	7.2	-9.6	8.479188	-8.49137	8.485281	-8.48528	8.485281	-8.48528
d1	4	12	5.792	-5.856	5.656859	-5.65686	5.656854	-5.65685	5.656854	-5.65685
d2	4	-12	-8.608	13.344	-11.3015	11.32588	-11.3137	11.31371	-11.3137	11.31371
d1	12.64911		8.236504		8.000006		8		8	
d2	12.64911		15.87955		16		16		16	
d	4	12	-8.608	13.344	-11.3015	11.32588	-11.3137	11.31371	-11.3137	11.31371
dx	-0.2	0.6	-0.09239	0.106599	-0.00051	0.000508	-6.5E-11	6.54E-11	-4.9E-17	4.91E-17

For real root, the convergence is cubic while for multiple root the convergence is linear.

On of the most quality of this algorithm is the high starting stability.

It is not guaranteed, but, practically, the Laguerre method converges from almost every point of the complex plane, and the trajectory is very fast

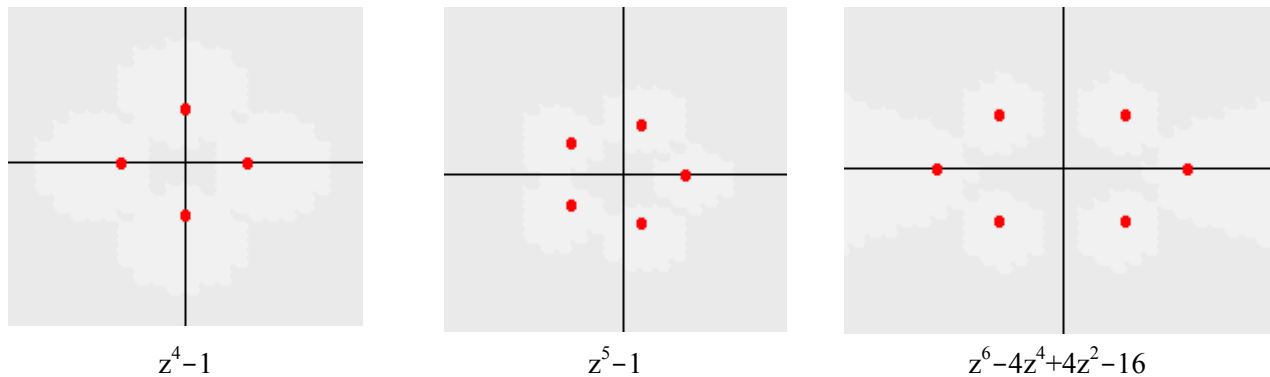


Of course, all that is detrimental to the computational cost that is high.

The complex square root operation requires the transcendental goniometric functions; its global cost can be estimated about 5 times a complex multiplication, that is 20 op. In addition we count the complex evaluation of the polynomials and its first and second derivatives. Totally, the cost for each iteration is about $12(2n+3)$, where n is the polynomial degree.

In the previous examples, $n = 4$, so the total cost for 5 iterations has been $12(2n+3)5 = 660$. Using the Newton method we get the same result with 9 iterations and a cost of $(16n-4)9 = 540$, while using the Bairstow method we have 10 iterations and a cost of $(8n+9)10 = 410$. Probably, the cost is the only drawback of the Laguerre method.

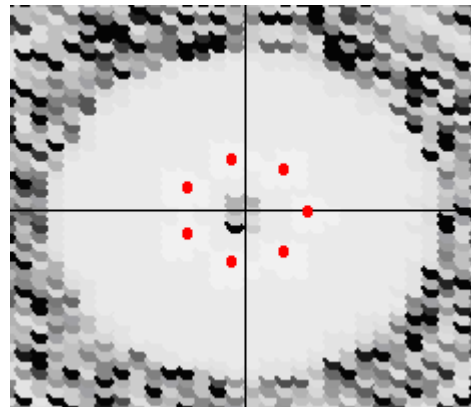
The globally convergence is very good. In the following pictures we show the convergence spectrums for three different polynomials



As we can see, for each starting points the algorithms converges in very few iterations (< 8)

From experimental tests it appears that choosing the starting point in the circular region having the radius "r" between 1 and 2 times the root radius is still a good choice.

In fact if we draw the convergence spectrum of the polynomial x^7-1 , we note a "turbulence" regions for $r > 3$, and, less accentuate, also inside the circle roots, near the origin.



ADK method

This algorithm was designed for finding simultaneously all the zeros of a real as well as complex polynomials. It apply the Newton iterative method to the ration functions:

$$F_i(z) = \frac{P(z)}{\prod_{j=1}^{i-1} (z - z_j^{(0)}) \cdot \prod_{j=i+1}^n (z - z_j^{(0)})}$$

where $z_j^{(0)}$ are root approximations of the polynomial $P(z)$. The functions $F_i(z)$ have the same zeros of the polynomial but, practically, the iterative method converges only to the i -th root z_i because the poles, very closer to the exact roots, prevent any approach to the roots different from z_i .

The iterative schema, really brilliant, has been developed and improved by many authors and it is known under several denominations such as: Aberth-Durand-Kerner method, Herlich formula, Mahely formula or also Weierstrass method.

The fastest variant requires the evaluation of the polynomial and its first derivative. If we name $z_i^{(k)}$ the i^{th} root at the step k , and $\Delta_i^{(k)} = P(z_i^{(k)}) / P'(z_i^{(k)})$, then the iterative formulas are:

$$z_i^{(k+1)} = z_i^{(k)} + \frac{\Delta_i^{(k)}}{1 + c_i^{(k)} \cdot \Delta_i^{(k)}} \quad , \quad i = 1, 2 \dots n \quad (1)$$

where the corrective coefficient $c_i^{(k)}$ is given by:

$$c_i^{(k)} = \sum_{j=1}^{i-1} \frac{1}{z_i^{(k)} - z_j^{(k+1)}} + \sum_{j=i+1}^n \frac{1}{z_i^{(k)} - z_j^{(k)}}$$

Note that if $c_i \cong 0$ then the iterative formula comes back to the Newton one

The formulas (1) are started with the vector $\mathbf{z}^{(0)} = [z_1^{(0)}, z_2^{(0)}, z_3^{(0)} \dots z_n^{(0)}]$ containing the best available estimations of all the polynomial roots.

From the first iteration we get the new value $z_1^{(1)}$ that substitutes the old value $z_1^{(0)}$ of the vector; successively we get $z_2^{(1)}$ substituting $z_2^{(0)}$ and so on. Synthetically, the iterations loop performs the following schema:

$$\begin{aligned} [z_1^{(0)}, z_2^{(0)}, z_3^{(0)} \dots z_{n-1}^{(0)}, z_n^{(0)}] &\Rightarrow z_1^{(1)} \\ [z_1^{(1)}, z_2^{(0)}, z_3^{(0)} \dots z_{n-1}^{(0)}, z_n^{(0)}] &\Rightarrow z_2^{(1)} \\ [z_1^{(1)}, z_2^{(1)}, z_3^{(0)} \dots z_{n-1}^{(0)}, z_n^{(0)}] &\Rightarrow z_3^{(1)} \\ [z_1^{(1)}, z_2^{(1)}, z_3^{(1)} \dots z_{n-1}^{(0)}, z_n^{(0)}] &\Rightarrow z_4^{(1)} \\ &\dots\dots\dots \\ [z_1^{(1)}, z_2^{(1)}, z_3^{(1)} \dots z_{n-1}^{(1)}, z_n^{(0)}] &\Rightarrow z_n^{(1)} \end{aligned}$$

At the end of the first loop, we have updated all the roots, that is we have obtained the new vector $\mathbf{z}^{(1)}$ and we can begin a new loop.

$$\begin{aligned} [z_1^{(1)}, z_2^{(1)}, z_3^{(1)} \dots z_{n-1}^{(1)}, z_n^{(1)}] &\Rightarrow z_1^{(2)} \\ &\dots\dots\dots \end{aligned}$$

After some loops, the sequence of the vector $\{\mathbf{z}^{(k)}\}$ converges simultaneously to all the real and complex roots of the polynomial. In the case of only real roots the algorithms does not need complex arithmetic

The global stability, even not proved, is very high. Experimental tests have shown a robust, fast convergence starting from initial estimations very far from the true roots.

The Starting Vector.

A very simple method takes a circular distribution of n points, centered in the roots center, with a radius $\rho < r < 2\rho$, being ρ the root radius.

Practically, for a polynomial of degree " n ", we calculate the root center $C = (x_c, 0)$ and the root radius ρ by one of the method shown in the previous chapters and then, we set

$$\theta = 2\pi / n,$$

$$q = 1/2 \text{ if } n \text{ is even, } q = 1 \text{ if } n \text{ is odd}$$

$$\text{for } i = 1, 2, \dots, n, \quad z_i = (\rho \cos(\theta \cdot (i - q)) + x_c, \rho \sin(\theta \cdot (i - q)))$$

For example, given the polynomial

$$z^6 - 18z^5 + 131z^4 - 492z^3 + 1003z^2 - 1050z + 425$$

The roots center is $x_c = 18/6 = 3$. Substituting $z = w+3$, we get the centered polynomial

$$w^6 - 4w^4 + 4w^2 - 16$$

that has the roots: $w = -1 \pm \hat{i}$, $w = 1 \pm \hat{i}$, $w = -2$, $w = 2$

Thus, the original roots are: $z = 2 \pm \hat{i}$, $z = 4 \pm \hat{i}$, $z = 1$, $z = 5$

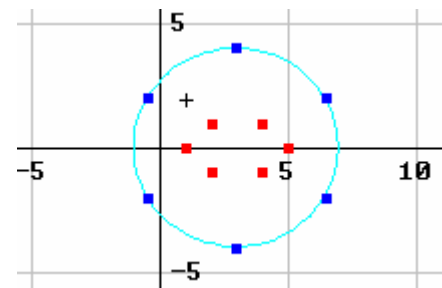
Let's see now how we choose the starting vector for the ADK algorithm

We calculate the estimation of the roots radius

$$\rho = 2 \max [4^{(1/2)}, 4^{(1/4)}, 16^{(1/6)}] = 4$$

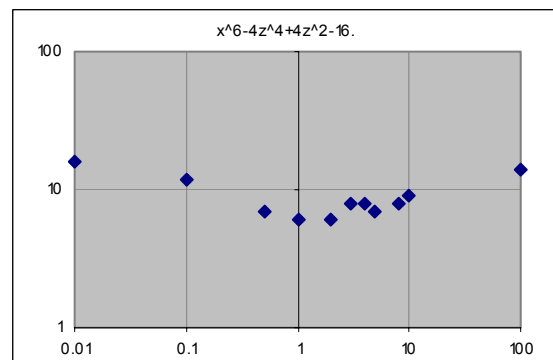
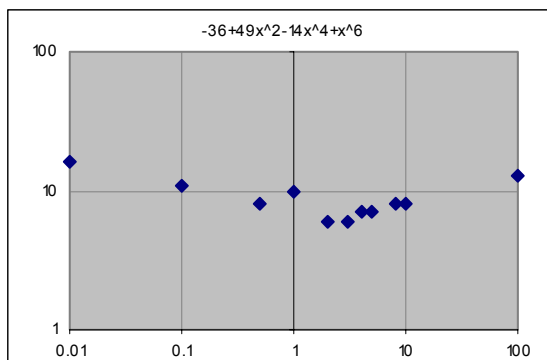
$$z_i = (4\cos(\pi/3 \cdot (i - 1/2)) + 3, 4\sin(\pi/3 \cdot (i - 1/2))), \quad i = 1, 2, \dots, 6$$

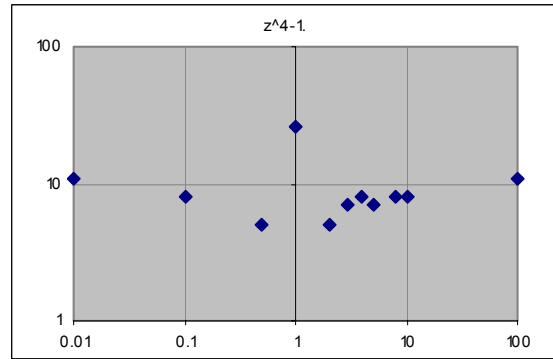
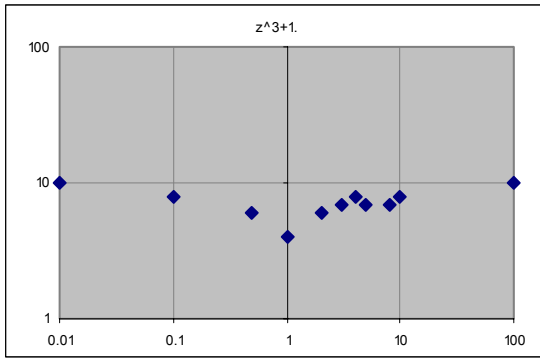
The true roots (red) and the starting points (blue) are shown in the graph



Starting from this values, the ADK algorithm converges simultaneously to the 6 roots in about 8 loops, with a global accuracy of about $1E-16$. Remember that each loop takes 6 iterations.

We wonder what would happen if we choose a different starting circle. In the following graphs is shown the total of loops for starting circles having a radius " r " in the range $0.01 < r < 100$, for different test-polynomials





As we can see the behaviors is very stable for every value of the starting circle radius. Clearly the starting values are not a problem for this algorithm.

Let's see the computational cost.

Formula (1) computation (complex):

$$16n+8$$

Corrective coefficient (complex):

$$3n-3$$

Total for iteration:

$$19n+5$$

Total for loop:

$$(19n+5)n = 19n^2+5n$$

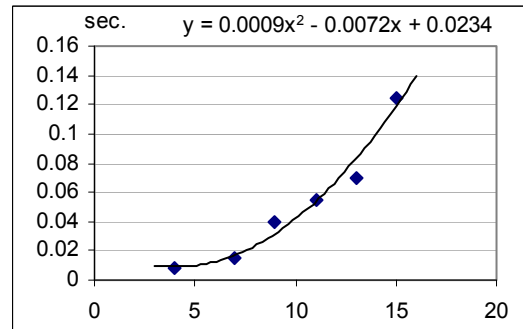
The cost of the 6th degree polynomial of the previous example is about 5700, against 3000 of the Newton-Rapshon method.

As guessed, the cost if the ADK method is very high, because, over all, it does not take advantage of the deflation degree reduction.

In the graph is shown the elaboration time (second) taken for solving polynomials with degree between 4 and 15.

The time remains under 1/10 sec. until degree is less then 14, but grows quickly for higher degrees

The test is performed on a Pentium PC, 1.8GHz, in VBA.



In spite of is low efficiency, this method is diffuse in many automatic rootfinding routine for its performances of high accuracy, stability and robustness and, last but not least for its easy, straight, implementation

Zeros of Orthogonal Polynomials

The NR method

A common method for finding the zeros of orthogonal polynomials follows three basic steps

Root estimation
 x_i^*

This task is performed with several approximated formulas: polynomials, rational, or power function are usually adopted.

Polynomial and derivative evaluation
 $L(x), L'(x)$

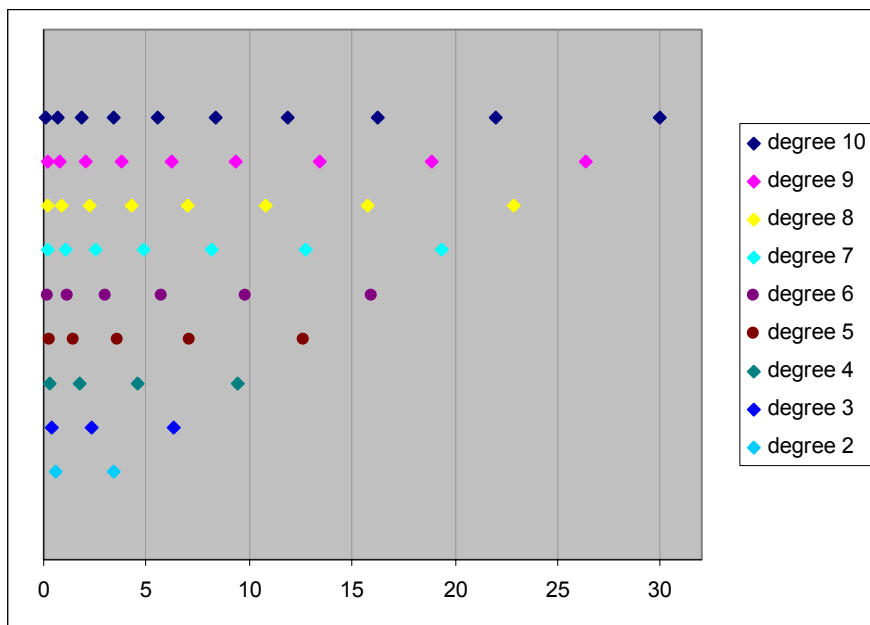
This task is performed with recursive functions both for the polynomial and the 1st derivative values. They do not use the polynomial coefficients and are very fast and efficient

Root refinement with Newton-Raphson method
 $x_{n+1} = x_n - L(x_n)/L'(x_n)$

This task is performed with the iterative use of the Newton-Raphson formula. If the initial root approximation is sufficiently close the convergence is very fast. But on the other hand it could converge to another root if the initial estimation is not sufficiently accurate

The first step is key of success. If the initial estimation of the root x_i^* is not sufficiently close the algorithm will converge to another unwanted root. This may raise problems when the degree of the polynomial increase because all the roots are more close each others.

The following picture shows the zeros of the Laguerre polynomials from 2 to 10 degree



As we can see, the distance between the roots increase at the right but decrease at left. The distance between x_1 and x_2 becomes very small for higher degree. This requires a good accuracy for the initial starting point of the NR algorithm; otherwise it cannot find all the polynomial roots.

This phenomenon explains why the estimation formulas are so complicated. For example, good estimation formulas for the generalized Laguerre polynomials roots known in the literature¹ are the following

$$z_1 = \frac{(1+m)(3+0.92m)}{1+2.4n+1.8m} \quad z_2 = z_1 + \frac{15+6.25m}{1+0.9n+2.5m}$$

$$z_i = z_{i-1} + \frac{\left(\frac{1+2.55(i-2)}{1.9(i-2)} + \frac{1.26(i-2)m}{1+3.5(i-2)} (z_{i-1} - z_{i-2}) \right)}{1+0.3m}$$

The VB implementation is:

```

If i = 1 Then
    z(1) = (1 + m) * (3 + 0.92 * m) / (1 + 2.4 * n + 1.8 * m)
ElseIf i = 2 Then
    z(2) = z(1) + (15 + 6.25 * m) / (1 + 0.9 * m + 2.5 * n)
Else
    z(i) = z(i - 1) + ((1 + 2.55 * (i - 2)) / (1.9 * (i - 2)) + 1.26 * (i - 2) *
        m / (1 + 3.5 * (i - 2))) * (z(i - 1) - z(i - 2)) / (1 + 0.3 * m)

```

Quite frightening, isn't? To compensate this intrinsically complication, this formula is very fast and accurate, giving a global efficient method to calculate the zeros of the generalized Laguerre polynomials

Fortunately similar formulas exist also for other orthogonal polynomials..

The ADK method

An alternative approach for finding all zero of orthogonal polynomials use the so called "Newton algorithm with implicit deflating", also called ADK formula².

$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n) - p(z_n) \sum_{j=1}^r \left(\frac{1}{z_n - z_j} \right)}$$

where z_1, z_2, \dots, z_r are a subset of the polynomial roots. The iterative formula will converges to a polynomial root surely different from the roots of the given subset. The thing works as these roots would be "extracted" from the polynomial $p(z)$. This explains the "implicit deflating" term. This method does not require any accurate estimation of the starting point and it is also as fast as the Newton method above described. It requires only a rough estimation of the roots radius. This method is valid for any orthogonal polynomial family, (or better, for any polynomial having all real roots).

Attack strategy for orthogonal polynomials

Assume to have to solve the 4th degree Laguerre polynomial.

$$L_4(z) = \frac{1}{24}z^4 - \frac{2}{3}z^3 + 3z - 4z + 1$$

We know that all its roots are positive, real and distinct: $0 < z_1 < z_2 < z_3 < z_4 < R$

The radius R can be roughly estimated, for $n \leq 20$, by the simple formula $R \cong 3.5 \cdot (n - 1)$

Or with the more accurate formula: $R \cong 3.14 \cdot (n - 1) + 0.018 \cdot (n - 1)^2$

¹ "Numerical Recipes in Fortran 77", Cambridge University Press, 1999

² Aberth-Durand-Kerner formula

At the beginning, the roots subset is empty ($r = 0$) the ADK formula comes back to the Newton one

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

If we choose a starting point greater (but not too far) then the max root x_4 , for example:
 $x_0 = R = 3.5 \cdot (4 - 1) = 10.5$, the iterative formula will converge monotonically to the max root x_4

$$x_4 \cong 9.39507091230113$$

Note that the starting point is not critical at all. The algorithm will work fine also for $x_0 = 12, 15$, etc. We have to point out that we should not exceed with the starting point because orthogonal polynomials raise sharply.

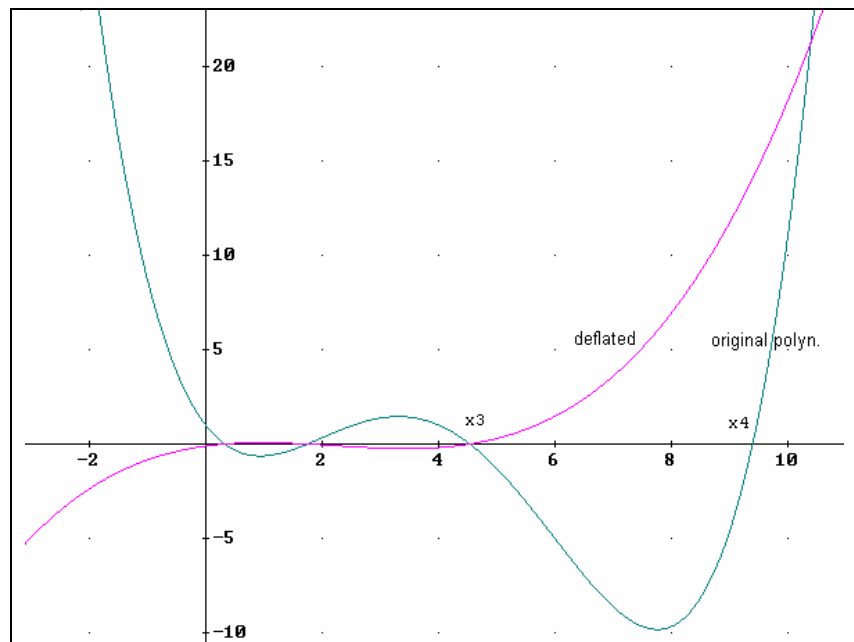
When the max root is found the ADK formula changes in:

$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n) - p(z_n) \frac{1}{z_n - z_4}}$$

Now the new starting point can be chosen near to x_4 , and greater of x_3 , that it's the max root of the deflated polynomial.

$$L_{d4}(z) = \frac{L_4(z)}{z - z_4}$$

This situation is shown in the following graph.



Note that we do not perform truly the deflating operation. The ADK formula will simply skip the x_4 root. A new starting point satisfying the starting conditions $x_0 > x_3$ is:

$$x_0 = x_4 + h$$

Where "h" is a little value (example 5% of x_4). This is necessary because the ADK formula cannot started from a root itself (1/0 division error). Substituting we get the new estimate starting point:
 $x_0 \cong 9.4 + 0.47 = 9.87$. The formula will converge to the root x_3 , obtaining

$$x_3 \cong 4.53662029692113$$

Again, inserting this value into the ADK formula we have

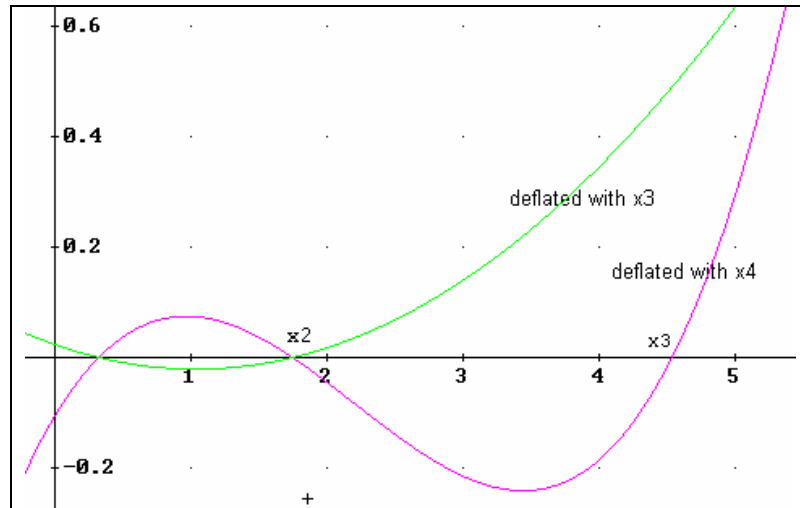
$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n) - p(z_n) \left(\frac{1}{z_n - z_4} + \frac{1}{z_n - z_3} \right)}$$

Taking another starting point

$$x_0 \cong 4.5 + 0.225 = 4.725$$

The formula will converge to the root x_2 , obtaining

$$x_2 \cong 1.74576110115835$$



Again, inserting the x_2 value into the ADK formula, we have

$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n) - p(z_n) \left(\frac{1}{z_n - z_4} + \frac{1}{z_n - z_3} + \frac{1}{z_n - z_2} \right)}$$

With the starting point $x_0 \cong 1.7.5 + 0.05 = 1.785$, It will finally converge to the last root x_1 , obtaining

$$x_1 \cong 0.322547689619392$$

So, the roots of 4th degree Laguerre polynomial, are

$x_4 =$	9.39507091230113
$x_3 =$	4.53662029692113
$x_2 =$	1.74576110115835
$x_1 =$	0.32254768961939

Now is clear how this algorithm works and because the accuracy of all the starting points is not as important as in the NR method. We have only to choose an upper bound of the greatest root and the ADK algorithm will find all roots in sequence from the maximum to the minimum.

Maximum root estimation
 $x_n < R$

This task is performed with several approximate simple formulas. Accuracy of R is not required

Polynomial and derivative
evaluation
 $L(x), L'(x)$

This task is performed with recursive functions both for the polynomial and the 1st derivative values. They do not use the polynomial coefficients and are very fast and efficient

Root finding with ADK
method

This task is performed with the iterative use of the ADK formula. If the starting point is greater than the max root the convergence to the max root is guaranteed.

Incidentally we note that all roots are already sorted. This is useful to save extra time spent in the sorting step

Jenkins-Traub method

This is an "hybrid" method, that is composed by several different algorithms reaching the global convergence for every polynomials, keeping also an average good efficiency. The first version of this excellent polynomial rootfinder, become a standard reference by now, has been developed in FORTRAN IV. Updated versions are in IMSL library with the name "CPOLY" (for complex polynomials) and "RPOLY" (for real polynomials) or in Mathematica¹ with the name "Nsolve". The JT rootfinder program is very elaborate. The FORTRAN version has almost 1000 code lines; its C++ translation reaches about 700 lines. Many tricks are included for assuring the convergence in any situation and many stratagemms are adopted for containing the error propagation.

Here we want only to show the basic concepts and explain the heart of the algorithm.

It finds one root at each time, starting from the smaller roots. After one root is found, the polynomial is deflated and the process is applied again to the reduced polynomial.

The heart of the JT method uses appropriate polynomials $H(z)$ of degree $n-1$ having the same roots z_2, z_3, \dots, z_n , of the original polynomial $P(z)$, except the smallest root z_1 . Then, the quotient $P(z)/H(z)$ will be a linear function having the only single root z_1

Of course the polynomial $H(z)$ cannot be exactly known but approximated by an iterative process. Stopping the process at the k^{th} iteration, (we hope) the roots of $H_k(z)$ will be very close to the exact roots of $P(z)$. The poles of the rational function $P(z)/H_k(z)$ push the convergence away from all the roots except z_1 , that can be quickly found using a simple fixed point iterative algorithm.

The conceptually iterative algorithm for finding the polynomials $H_k(z)$, apart special trick, is the following.

$$H(z) = h_{n-1} \cdot z^{n-1} + h_{n-2} z^{n-2} \dots + h_2 \cdot z^2 + h_1 \cdot z + h_0 +$$

Setting $H_0(z) = P'(z)$

$$H_{k+1}(z) = \frac{1}{z} \left(H_k(z) - \frac{H_k(0)}{P(0)} P(z) \right) , k = 1, 2 \dots K$$

Usually, only few iterations are need ($K = 4 \div 8$) to obtain a satisfactory polynomial

Let's see an example. Take the polynomial $P(z) = z^3 - 8z^2 + 17z - 10$ having all real distinct roots
The sequence of H polynomials will be:

$$\begin{aligned} H_0(z) &= 17 - 16z + 3z^2 \\ H_1(z) &= 12.9 - 10.6z + 1.7z^2 \\ H_2(z) &= 11.33 - 8.62z + 1.29z^2 \\ H_3(z) &= 10.641 - 7.774z + 1.133z^2 \end{aligned}$$

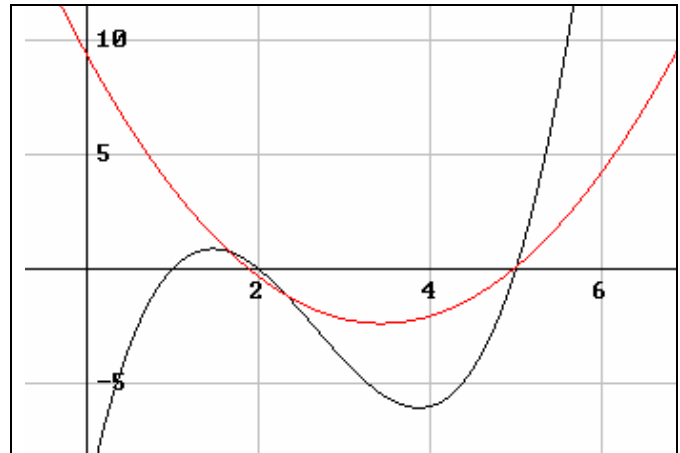
In this case, the sequence of $\{H_k\}$ is composed by parabolas converging to the one having its zeros coinciding with the two largest roots of the given polynomial

The following graph illustrates better what happens

¹ Mathematica , WoframResearch

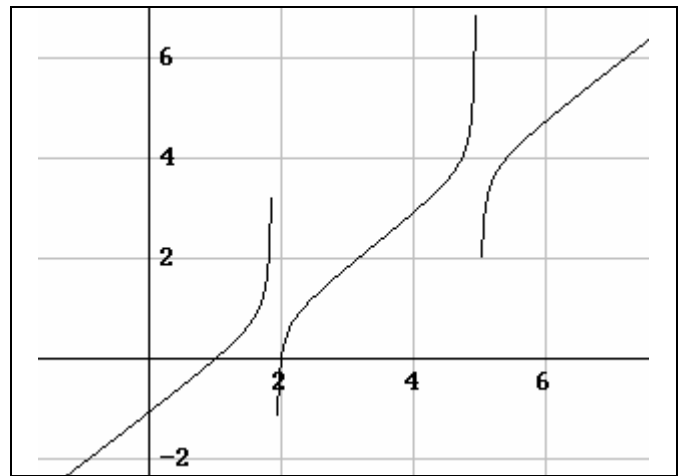
In the graph are shown the plots of the polynomial and the parabola $H_3(z)$. We can clearly see that the parabola crosses the x-axis very close to the two polynomial roots of largest magnitude.

Continuing, the zeros of the parabola approach much more the polynomial roots. But the zeros of H_3 , thus the poles of the rational function $1/H_3$, are usually sufficient to get the convergence toward the smallest root. (JT usually computes up to H_5)



The smallest root is approximated by using the fixed point iterative method applied to the rational function $P(z) / H_3(z)$, that is shown in the graph.

We note clearly that this function has a linear trend except at the two poles $x = 2$ e $x = 5$, where are approximately located the largest roots. We note also that around the smallest root $x = 1$ the trend is about linear.



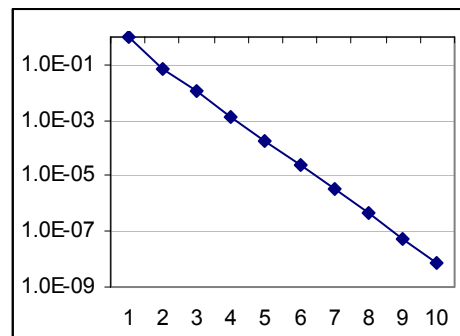
This transformation is the heart of the JT method that tries to come out the smallest root neutralizing the other roots by appropriate poles. So the JT iterative schema is:

$$z_{i+1} = z_i - \frac{P(z_i)}{\tilde{H}_k(z_i)}$$

where \tilde{H} is the monic polynomial of $H(z)$: $\tilde{H}_k(z_i) = \frac{H_k(z)}{h_{n-1}}$

Starting with $x_0 = 0$ we get the following iteration values

x	P	\tilde{H}	dx	dx
0	-10	9.3918799	1.0647496	1.0647496
1.0647496	0.2383072	3.2198672	-0.0740115	0.0740115
0.9907381	-0.0374774	3.5755621	0.0104816	0.0104816
1.0012196	0.004871	3.5245225	-0.001382	0.001382
0.9998376	-0.0006498	3.5312397	0.000184	0.000184
1.0000216	8.639E-05	3.5303451	-2.447E-05	2.447E-05
0.9999971	-1.149E-05	3.530464	3.255E-06	3.255E-06
1.0000004	1.528E-06	3.5304482	-4.329E-07	4.329E-07
0.9999999	-2.033E-07	3.5304503	5.757E-08	5.757E-08
1	2.703E-08	3.5304501	-7.657E-09	7.657E-09



The linear convergence to the root $x_1 = 1$ is evident. The convergence order, in that case, is 1.

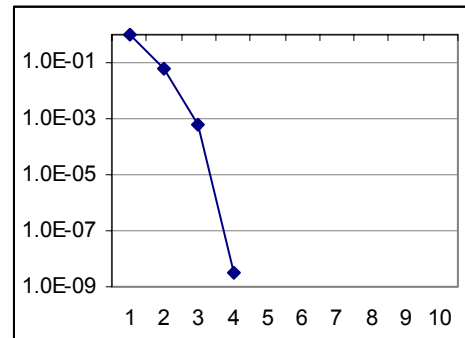
The JT method increases the convergence order with the polynomial shifting. It has been demonstrated that the convergence order raises to the Newton-Raphson order. In the second step of the process, the iterative formula of the polynomial H_k changes as:

$$z_{i+1} = z_i - \frac{P(z_i)}{\tilde{H}_k(z_i)}$$

$$H_{k+1}(z) = \frac{1}{z - z_{i+1}} \left(H_k(z) - \frac{H_k(z_{i+1})}{P(z_{i+1})} P(z) \right)$$

Using this trick, the shifting is enclosed in the iterative process in a very stable and efficient way.

x	P	\tilde{H}	dx	dx
0	-10	9.39188	1.06474955	1.0647496
1.0647496	0.2383072	3.715299	-0.0641421	0.0641421
1.0006074	0.0024279	3.996942	-0.0006074	0.0006074
1	-1.28E-08	3.999979	3.2092E-09	3.209E-09
1	6.75E-14	3.999979	-1.688E-14	1.688E-14



The convergence is greatly increased. In only 3 iterations we have reached the accuracy of about 1E-9

The JT method must be started from a complex value for approaching to a complex root. The POLY program has many specialized subroutines detecting complex root and, consequently, choosing the appropriate starting point. Other functions check the global convergence, and other limit the round-off errors propagation. A specific attention has been dedicated for the cluster-roots detection, that, as we have seen, slow down the convergence speed similarly to the multiple roots. To list all the implemented tricks is beyond the scope.

Computation cost. Over the years, the JT method JT is constantly improved and, in spite its high number of code lines, it is one of the most efficient method for polynomial solving. It is far less expensive than the Laguerre or the Durand-Kerner algorithm and even cheaper of the Newton one; only the Bairstow algorithm, when it converges, may compete with the JT method.

QR method

Another method for finding simultaneously all the zeros of a real polynomials bases oneself on the eigenvalues of the companion matrix

Given a monic polynomial ($a_n = 1$), we set the following matrix

$$a(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + x^n$$

$$A = \begin{bmatrix} 0 & 0 & \dots & 0 & -a_0 \\ 1 & 0 & \dots & 0 & -a_1 \\ 0 & 1 & \dots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & -a_{n-1} \end{bmatrix}$$

The eigenvalues of $[A]$ are, as known, the polynomial roots. Therefore we can adopt any eigenvalues solving method to obtain the polynomial roots. One of the most used is the QR factorization, where Q is a orthogonal matrix and R is an upper-triangular matrix.

Conceptually speaking, the algorithm is simple and it adopts the following iterative schema.

Set $A_0 = A$, we split the matrix A_0 in the product $Q_1 \cdot R_1$ and then we build the matrix $A_1 = R_1 \cdot Q_1$. We repeat the split of A_1 in $Q_2 \cdot R_2$, obtaining $A_2 = R_2 \cdot Q_2$ and so on.

The sequence of matrices $\{A_n\}$ converges to a block upper-triangular matrix. If all eigenvalues are real, the diagonal elements are also real. If there is a couple of complex conjugate eigenvalues, then the correspondent block converges to a 2×2 matrix. It can happen, for example, the following cases

λ_1	#	#	#	#	#
0	λ_2	#	#	#	#
0	0	λ_3	#	#	#
0	0	0	λ_4	#	#
0	0	0	0	λ_5	#
0	0	0	0	0	λ_6

λ_1	#	#	#	#	#
0	λ_2	#	#	#	#
0	0	α_{11}	α_{12}	#	#
0	0	α_{21}	α_{22}	#	#
0	0	0	0	β_{11}	β_{12}
0	0	0	0	β_{21}	β_{22}

The left matrix has all real eigenvalues. The right matrix has 2 real eigenvalues (λ_1 e λ_2) and two couples of complex conjugate eigenvalues located in the 2×2 sub-matrices " α " e " β "

To extract them, we have only to solve the characteristic polynomial of each sub-matrix.

$$\lambda^2 - (\alpha_{11} + \alpha_{22}) \lambda + (\alpha_{11} \cdot \alpha_{22} - \alpha_{21} \cdot \alpha_{12}) = 0$$

$$\lambda^2 - (\beta_{11} + \beta_{22}) \lambda + (\beta_{11} \cdot \beta_{22} - \beta_{21} \cdot \beta_{12}) = 0$$

The QR method requires the matrix calculus, but it has two very important advantages: it does not need any starting value and its is intrinsically robust and stable.

Though there is no proof about is global convergence, practical experiments have shown that the QR algorithm converges in almost every cases.

These characteristics have made this algorithm very popular with the programmers: the program HQR of the EISPAK Fortran library, is in fact one of the best routine for finding eigenvalues of real standard matrices.

Example. Real Roots

Given $x^3 - 8x^2 + 17x - 10$ let's set the associated companion matrix and find its eigenvalues by the QR factorization¹

$$\begin{array}{|c|c|c|} \hline \text{A0} & & \\ \hline 0 & 0 & 10 \\ \hline 1 & 0 & -17 \\ \hline 0 & 1 & 8 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \text{Q1} & & \text{R1} \\ \hline 0 & 0 & 1 & | & 1 & 0 & -17 \\ \hline 1 & 0 & 0 & | & 0 & 1 & 8 \\ \hline 0 & 1 & 0 & | & 0 & 0 & 10 \\ \hline \end{array}$$

Multiplying R₁ and Q₁ we get the new matrix A₁ to which we apply again the factorization

$$\begin{array}{|c|c|c|} \hline \text{A1} & & \\ \hline 0 & -17 & 1 \\ \hline 1 & 8 & 0 \\ \hline 0 & 10 & 0 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline \text{Q2} & & \text{R2} \\ \hline 0 & -0.862 & 0.507 & | & 1 & 8 & 0 \\ \hline 1 & 0 & 0 & | & 0 & 19.723 & -0.862 \\ \hline 0 & 0.507 & 0.8619 & | & 0 & 0 & 0.507 \\ \hline \end{array}$$

After 6 iterations we note A₆ converges to a triangular matrix. The eigenvalues begin to appear along the first diagonal

$$\begin{array}{|c|c|c|} \hline \text{A6} & & \\ \hline 5.1028 & -17.04 & 11.435 \\ \hline 0.0186 & 1.9466 & -1.773 \\ \hline 0 & 0.0277 & 0.9506 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline \text{Q7} & & \text{R7} \\ \hline 1 & -0.004 & 5E-05 & | & 5.1029 & -17.03 & 11.428 \\ \hline 0.0036 & 0.9999 & -0.014 & | & 0 & 2.0088 & -1.801 \\ \hline 0 & 0.0138 & 0.9999 & | & 0 & 0 & 0.9755 \\ \hline \end{array}$$

Continuing, after 20 iterations we get the matrix A₂₀

$$\begin{array}{|c|c|c|} \hline \text{A20} & & \text{iter. } 20 \\ \hline 5 & -16.74 & 11.881 \\ \hline 5E-08 & 2 & -1.871 \\ \hline -3E-16 & 2E-06 & 1 \\ \hline \end{array}$$

We note the progressive zeroing of the values under the first diagonal and the convergence of the first diagonal to the values $\lambda_1 = 5$, $\lambda_2 = 2$, $\lambda_3 = 1$. We deduce that all the eigenvalues are real and thus we have got all the roots of the given polynomial. The max residual value under the diagonal (2E-6) gives also a raw estimation of the global accuracy.

Example. Complex Conjugate Roots

Given $x^4 - 7x^3 + 21x^2 - 37x + 30$ set the companion matrix and find its eigenvalues. By 100 iterations of the QR algorithm we get the following matrix A₁₀₀

$$\begin{array}{|c|c|c|c|} \hline \text{A0} & & & \\ \hline 0 & 0 & 0 & -30 \\ \hline 1 & 0 & 0 & 37 \\ \hline 0 & 1 & 0 & -21 \\ \hline 0 & 0 & 1 & 7 \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|} \hline \text{A100} & & & \\ \hline 3 & -2.91838 & -25.1713 & -44.6849 \\ \hline 9.02E-14 & -0.33519 & -4.67545 & -7.93471 \\ \hline 2.72E-21 & 1.236835 & 2.335192 & 4.270452 \\ \hline 3.34E-16 & -6.8E-22 & 6.8E-06 & 2 \\ \hline \end{array}$$

We note the progressive zeroing of the values under the first diagonal only for the values $\lambda_1 = 3$, $\lambda_2 = 2$. We deduce that two eigenvalues are real while the other eigenvalue λ_2 e λ_3 are complex conjugate. To obtain them we set the characteristic polynomial $a_2\lambda^2 + a_1\lambda + a_0$ of the following (2x2) sub-matrix.

¹ The QR factorization can be easily obtained in Excel by the addin Matrix.xla (by Foxes Team)

-0.33519	-4.67545
1.236835	2.33519

a2	a1	a0
1	-2	5.0000

Solving we have: $\lambda^2 - 2\lambda + 5 = 0 \Rightarrow \lambda = 1 \pm 2i$

Example. Multiple Roots

Given $x^4 - 7x^3 + 17x^2 - 17x + 6$ build the companion matrix and find its eigenvalues. By 100 iterations of the QR we get the following matrix A_{100}

A0			
0	0	0	-6
1	0	0	17
0	1	0	-17
0	0	1	7

 \Rightarrow

A100			
3	-13.8636	16.82906	-12.6343
4.77E-15	2	-3.41298	1.872992
2.28E-16	3.85E-15	1.010308	-0.89431
-1E-16	3.83E-16	0.000119	0.989692

We note the progressive zeroing of the values under the first diagonal and only for the values $\lambda_1 = 3, \lambda_2 = 2$. We deduce that the eigenvalues λ_3, λ_4 could be conjugate complex. Let's see. Setting the characteristic polynomial $a_2\lambda^2 + a_1\lambda + a_0$ of the (2x2) sub-matrix

1.010308	-0.89431
0.000119	0.989692

a2	a1	a0
1	-2	1

and solving, we have : $\lambda^2 - 2\lambda + 1 = 0 \Rightarrow (\lambda - 1)^2 = 0 \Rightarrow \lambda = 1, m = 2$

Surprisingly we have got a single multiple root instead of two complex roots

Multiple roots slow down the convergence speed and decrease the global accuracy for almost algorithm. But using this trick we can obtain accurate multiple roots using a reasonable number of iterations.

In order to accelerate the convergence and increase the efficiency of the standard QR method, many ingenious variant have been excogitated such as the "shifting" technique.

It has been proved, in fact, that the convergence to zero of the under-diagonal elements follows the relation

$$a_{ij}^{(n)} \approx (\lambda_i / \lambda_j)^n.$$

Consequently, the convergence may be very slow if λ_i e λ_j are very closer each other. Performing the matrix transformation $\mathbf{B} = \mathbf{A} - k\mathbf{I}$ where "k" is an appropriate real or complex constant, also the eigenvalues will be shifted of the same amount and, in that case, the convergence speed of \mathbf{B} is given by the relation:

$$b_{ij}^{(n)} \approx [(\lambda_i - k) / (\lambda_j - k)]^n$$

A good choice for "k" is a value near to the lowest eigenvalue.

We note that without the shifting the eigenvalues appear along the diagonal in reverse order of magnitude while, with the shifting, they can appear in any order.

Polynomial Test

Random Test

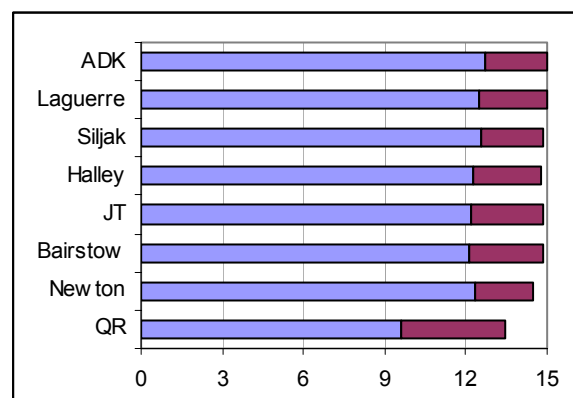
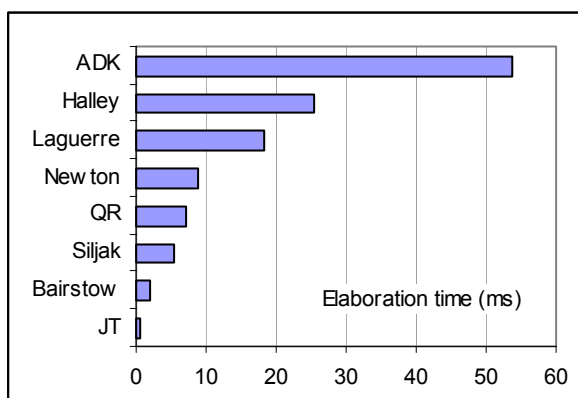
The measure of the computational cost and, thus, the efficiency of a rootfinding method for polynomial requires a different approach from those of the real function $f(x)$. The reason is the polynomial rootfinding algorithm may be quite bulky containing many code lines: from hundred of the most compact algorithms to thousand of the "hybrid" algorithms. Thus, the exact counting of each operation would be quite difficult and heavy. Besides, it would be useless because we are interested in a global measure of the computational cost for solving all the polynomial roots and not for one single root.

The total computation time depends on many factors such as the polynomial degree, the spatial distribution of its roots, the roots multiplicity, the starting values, the precision required, etc. All these factors cannot be generally known therefore we prefer a statistic approach.

The statistic samples. It is composed by 1100 random polynomials of degree between 3 and 14, with mixed real and complex roots, having module $|z| < 100$. For each group of 100 polynomial having the same degree, has been measured the elaboration time¹, the average logarithm error (LRE)² and the error standard deviation. It has also been reported the number of non convergence (fails).

The statistic is summarized in the following table and bar-graphs.

Method	time (ms)	LRE	Dev. LRE	fails	Starting method
JT	0.68	13.5	1.32	0%	-
Bairstow	1.87	13.5	1.38	0.2%	random + fun
Siljak	5.37	13.7	1.16	0%	fixed
QR	7.23	11.5	1.95	0%	-
Newton	8.97	13.4	1.05	0%	Roots circle
Laguerre	18.16	13.8	1.32	0%	Roots circle
Halley	25.36	13.5	1.28	0%	Roots circle
ADK	53.66	14.0	1.22	0%	Roots circle



Elaboration time (ms)

LRE

As we can see, almost all the methods have overcome the convergence test (only 2 fails for Bairstow). The "hybrid" method JT, despite of its 1000 code-lines results both very fast and accurate. On the contrary the methods with cubic convergence (Laguerre e Halley) suffer from their

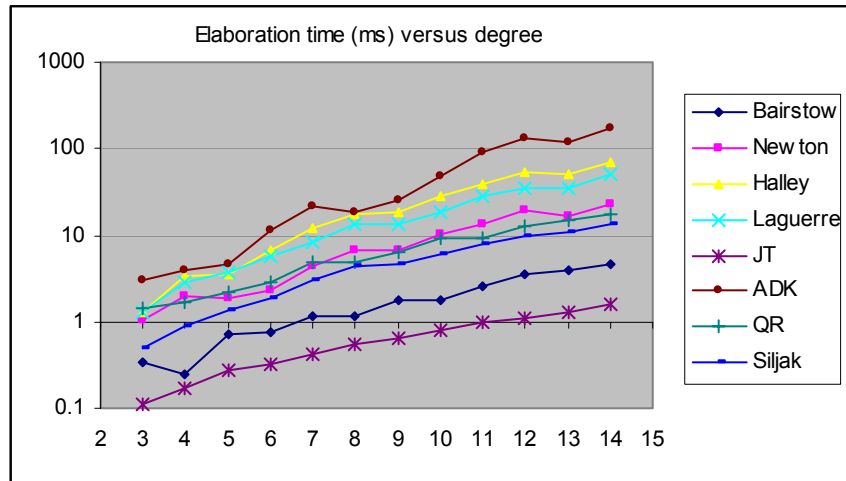
¹ The time, in ms, was measured with a CPU Pentium, 1.2GHz

² LRE : (Log. relative error) is defined as $LRE = \min(-\text{Log}_{10}(e), 15)$

heavy calculation. The ADK method has been the slowest because it does not take advantage of the deflation, but it has also been one of the most accurate.

The average LRE error is about $13.3 (\pm 1.3)$.

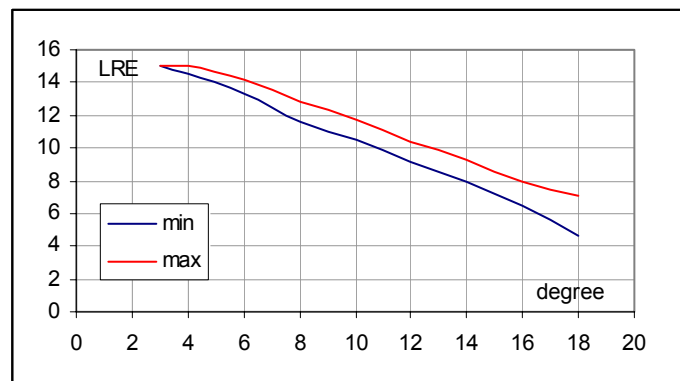
The following graph shows, for each method, the elaboration time based on the degree.



Wilkinson Test

The polynomial family having all integer roots $x = \{1, 2, 3 \dots n\}$, is usually used to test the accuracy of the rootfinding algorithms. These polynomials, studied by Wilkinson and others, are very difficult for almost methods. We have applied the previous methods to the W. polynomials from degree $3 \leq n \leq 18$, and measured the correspondent accuracy (LRE). The computation was entirely performed in standard arithmetic (15 digits)

n	LRE min	LRE average	LRE max
3	15	15	15
4	14.5	14.9	15
6	13.3	13.7	14.2
8	11.6	12.2	12.8
10	10.5	11.2	11.7
12	9.2	10	10.4
14	7.9	8.7	9.3
16	6.5	7.2	8
18	4.6	6	7.1



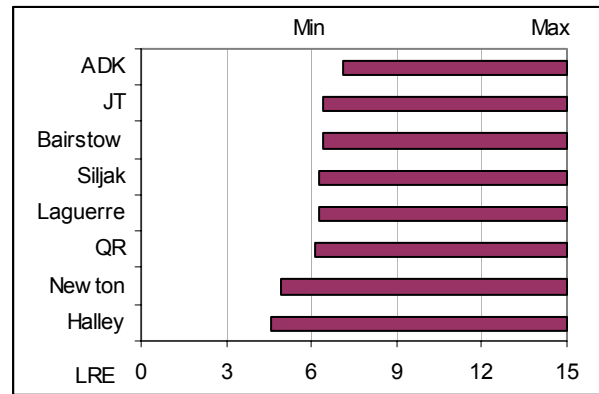
The accuracy obtained in this test (LRE = 8.7) is much less than the previous one (LRE = 13.3). It is lower than 4 order.

The test put in evidence a global equilibrium between all the methods.

Note that W. polynomial with $n > 16$ can be written only approximated because some coefficients overcome the standard 15 significant digits.

The global behavior of the rootfinder versus the W. polynomial is summarized in the following bar-graph showing the max e min LRE of each method.

The light advantage of ADK is mainly acquired because this method avoids the round-off error derived from the deflation process that is common to the other methods, except the QR



Selectivity Test

In literature we can find many "test-suit" for polynomial rootfinder¹ specialized for testing their performances of stability, efficiency and accuracy. Generally, these test-polynomials are chosen to test the ability of approximating multiple roots or cluster roots for high polynomial degree (up to several hundred). Same test-polynomials have also approximated coefficients in order to test the algorithm robustness.

Among hundred of test-cases, the following test-polynomial, extracted from the Jenkins-Traub test, appears adapted to test the algorithms selectivity for roots very close each other (cluster-roots).

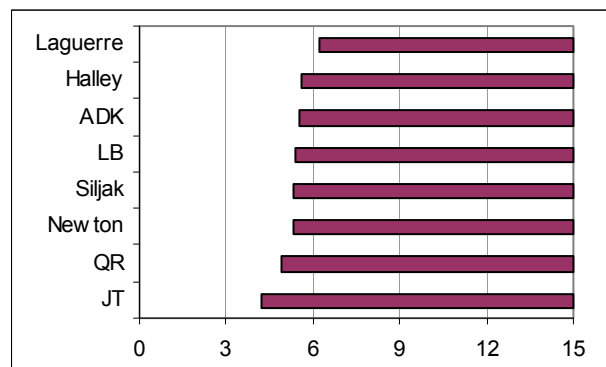
$$jt06 \quad (x-0.1)(x-1.001)(x-.998)(x-1.00002)(x-.99999)$$

These "almost multiple" roots are very common in applied science. The given polynomial has 4 roots with a very short relative distance, between $3E-5$ and $2E-4$. If we apply the rootfinder methods we observe a lack of accuracy just as if the polynomial would have a multiple root.

The bar-graph shows the max e min LRE, obtained by each algorithm applied to the jt06 test-polynomial.

Of course the minimum values are obtained for the roots near 1, while the maximum LRE = 15 it obtained for the root $x = 0.1$

The minimum LRE is about $4 \div 6$; this means that, in standard precision (15 digits), the minimum detectable distance between two roots is about $10^{-4} \div 10^{-6}$



¹ Jenkins et al. 1975, Loizou 1983, Brugnano et al. 1985, Dvorkuc 1969, Farmer et al. 1975, Goedecker 1984, Igarashi et al. 1995, Iliev 2000, Miyakoda 1989, Petkovic 1989, Stolan 1995, Toh et al 1994, Uhlig 1995, Zeng 2003, Zhang 2001

Roots Finding Strategy

Attack Strategy

Several strategies exist for finding polynomial roots. They change according to degree, the accuracy required, the coefficient scale, the topological root configuration, the multiplicity, etc. The searching domain, complex plane or real axes, represents also an important aspect. In this chapter we will see a general process for finding all the roots of a real polynomial in order to achieve the maximum accuracy possible.

The process is divided into the following steps

1. Search for the possible integer roots. For those roots, as we have seen, methods exist to extract them virtually without error independently from their multiplicity. The integer deflation can reduce the degree without introducing any round-off errors
2. Search for the possible multiple roots by GCD. If it is different from 1, we proceed to the polynomial splitting into factors with lower degree having single roots. Generally, this process is error prone, but the global accuracy gain is usually remarkable.
3. Search for the real and complex roots by any general purpose rootfinding algorithm. In particular cases we can proceed for selective search (for example only real roots) by appropriate starting values

Beside of the general strategy just exposed, we can apply, if necessary, all the transformation methods already seen such as the shifting, the scale changing, the coefficient reduction, the rounding, etc.

Examples of Polynomial Solving

Example 1. Given the polynomial¹ of 9th degree.

$$P = [200, 140, 222, 181, 46, 43, 26, 3, 2, 1]$$

We search for possible integer roots by the Ruffini's method²

We find the root $x = -2$ with multiplicity 3; performing three times the synthetic division for $(x+2)$, we get the reduce polynomial of 6th degree.

$$P_1 = [25, -20, 39, -24, 15, -4, 1]$$

We try to detect other multiple roots performing the factorization by the GCD method. We get

$$P_2 = [5, -2, 6, -2, 1] \quad , \quad P_3 = [5, -2, 1]$$

We note that the polynomial P_3 is the GCD between P_1 and its derivative P_1'

The polynomials P_2 e P_3 have all single roots that can be good approximated with a general rootfinder method³

¹ In these examples, a polynomial can be given either by its coefficients vector or by its algebraic representation depending of the more suitable form. Coefficients are in ascending degree, from left to right or from top to bottom.

² A rootfinder using this method, called "Rootfinder RF", can be found in the Excel addin Xnumbers.xla

³ In these examples we use the rootfinder JT (Jenkins-Traub), if not differently specified

The final result is thus:

From the polynomial P	from P ₂	from P ₃
-2 , -2, -2	$1 \pm 2 i$, $\pm i$	$1 \pm 2 i$

All the roots are obtained with the highest possible accuracy¹ of 1E-16

Let's see now what happens if we apply the rootfinder directly to the given polynomial P(x).

The accuracy of the triple root -2 is now about 1.3E-5 , while the double complex root $1 \pm 2 i$ shows an accuracy of about 2E-6.

The single roots $\pm i$ have still the maximum accuracy. The global precision is still acceptable but certainly not comparable with those obtained in the previous strategy

z re	z im	err
-2.000009686	1.67763E-05	1.323E-05
-2.000009686	-1.67763E-05	1.323E-05
-1.999980628	0	9.686E-06
0	1	6.172E-16
0	-1	6.172E-16
0.999999056	1.999996904	2.02E-06
0.999999056	-1.999996904	2.02E-06
1.000000944	2.000003096	2.02E-06
1.000000944	-2.000003096	2.02E-06

If we consider the accuracy based on the elapsed time, the direct method is still more efficient.

There are cases, however, in which the errors effect is catastrophic that the values returned by the direct method are completely useless. In that cases the attack strategy just descript is the only way to give acceptable results.

Another suitable method for extracting multiple roots without losing efficiency is the "drill-down"².

Example 2.

$$P = [3888, -40176, 174096, -413028, 587577, -514960, 272600, -80000, 10000]$$

This polynomial has no integer roots because the Ruffini's method returns no positive result. Let's see if there are multiple roots. A factorization exists and it is:

$$P_1 = [36, -216, 443, -360, 100] \quad , \quad P_2 = [-18, 63, -64, 20] \quad , \quad P_3 = [-6, 5]$$

The last polynomial is linear thus its root is: $x = 6/5 = 1.2$

The single roots of the other polynomials can be found using a rootfinder

from P ₁	from P ₂	from P ₃
0.4, 0.5, 1.2, 1.5	0.5, 1.2, 1.5	1.2

So, all the roots are: 0.4, 0.5 (m = 2) , 1.2 (m = 3) , 1,5 (m = 2). The global accuracy is about 1E-15

¹ In these example we use the standard precision of 15 significant digits, if not differently specified

² In Xnumbers.xla this algorithm is called "Rootfinder GN" (Generalized Newton-Raphson)

if we apply the rootfinder directly to the given polynomial P(x) we get the following approximated values

The single root has a good accuracy and also the double roots are acceptable, but the triple root 1.2 falls to about 2E-4.

z re	z im	err
0.4	0	7.22E-15
0.499999999	0	1.38E-09
0.500000001	0	1.38E-09
1.199941593	0.00010111	0.00016
1.199941593	-0.00010111	0.00016
1.200116814	0	0.000117
1.499997919	0	2.08E-06
1.500002081	0	2.08E-06

Example 3. Using the GCD method, factorize the standard form of the following polynomial¹

$$P(x) = (x - 4)^2 \cdot (x - 3)^4 \cdot (x - 2)^6 \cdot (x - 1)^8$$

The standard form of the given polynomial is shown in the column at the left. In the other columns, from left to right as in a spreadsheet, are the coefficients of the polynomial factors²

P(x)	P1	P1	P2	P2	P3	P3	P4	P4
82944	24	24	-6	-6	2	2	-1	-1
-1064448	-50	-50	11	11	-3	-3	1	1
6412608	35	35	-6	-6	1	1		
-24105792	-10	-10	1	1				
63397936	1	1						
-123968128								
186963852								
-222655300	The polynomial can be factorized as:							
212617033	P = (P1 · P2 · P3 · P4) ²							
-164382924								
103450582								
-53083024								
22168911	The roots of each polynomial are:							
-7494136								
2030608	P1		P2		P3		P4	
-434244	1, 2, 3, 4		1, 2, 3		1, 2		1	
71575								
-8764								
750								
-40								
1								
	0.4531							

Then, putting all together, we find the following roots: 1 (m = 8), 2 (m = 6), 3 (m = 4), 4 (m = 2). We note, incidentally, that all the roots are integer so they would be extracted by the Ruffini's method as well.

¹ Of course, the factorized form is given only for clarity. In practical cases this form is never known and we always start from the standard polynomial form.

² This is just the output of the macro "Polynomial Factors" in Xnumbers.xls

Example 4.

Find the roots of the 6th degree polynomial

We note that the coefficients are scaled very different each other: they start from 1 to 10⁻²¹ of the last coefficient. The intermediate coefficients have values progressively descendent.

degree	coeff.
0	1E-21
1	-1.11111E-15
2	1.122322110E-10
3	-1.123333211E-06
4	0.00112232211
5	-0.1111111
6	1

In such case, it may be suitable for the sake of accuracy, to equalize the coefficients by the scaling method. Setting $x = y / k$, the polynomial becomes:

$$P(y) = y^6 + k \cdot a_5 y^5 + k^2 \cdot a_4 y^4 + k^3 \cdot a_3 y^3 + k^4 \cdot a_2 y^2 + k^5 \cdot a_1 y + k^6 \cdot a_0$$

The value "k" is not critical at all. Taking $k = 1000$, the coefficients change as shown in the following left table. Solving by the Newton method we get the roots y_i that, successively divided by 1000, gives the original roots x_i .

n	k ⁿ	k ⁿ a _n
0	1.E+18	0.001
1	1.E+15	-1.11111
2	1.E+12	112.232211
3	1.E+09	-1123.333211
4	1.E+06	1122.32211
5	1.E+03	-111.111
6	1	1

y	x	x (err. rel.)
0.000999999999968	9.99999996774E-07	3.22574E-10
0.010000000000000	0.0000100000000000	1e-16
0.09999999999986	9.99999999986E-05	1.42464E-12
1.000000000000047	0.0010000000000005	4.6946E-13
10	0.01	1e-16
100	0.1	1e-16

As we can see, the relative accuracy is quite good

Let's see what happens if we apply directly the Newton¹ algorithm to the given polynomial

In that case the result is quite different. The smaller roots are affected by a large relative error, that progressively decreases in the larger roots.

x	x (rel. err)
9.5559951020E-07	0.04440
1.0049360690E-05	0.00494
9.9994989752E-05	5.010E-05
0.00100000005010	5.010E-08
0.00999999999995	4.960E-12
0.1	0

¹ The JT algorithm automatically performs the coefficients equalization so the scaling is not necessary

Bibliography

"Numerical Recipes in Fortran", W.H. press et al., Cambridge University Press, 1992

"Numerical Analysis" F. Sheid, McGraw-Hill Book Company, New-York, 1968

"Numerical Methods that Work", Forman S. Acton

"Iterative Methods for the solution of Equations", J. F. Traub, Prentice Hill, 1964

"Numerical Mathematics in Scientific Computation", Germud Dahlquist, Åke Björck, 2005, [//www.mai.liu.se/~akbjo](http://www.mai.liu.se/~akbjo)

"Graphic and numerical comparison between iterative methods", Juan L. Varona, in The "Mathematical Intelligencer 24", 2002, no. 1, 37–46.

"Scientific Computing - An Introductory Survey", Michael T. Heath, in "Lecture Notes to Accompany", 2001, Second Edition

"An acceleration of the iterative processes", Gyurhan H. Nedzhibov, Milko G. Petkov, Shumen University, Bulgaria , 2002

"Lezioni - Equazioni non lineari", Lorenzo Pareschi, Università di Ferrara

"On a few Iterative Methods for Solving Nonlinear Equations", Gyurhan Nedzhibov, Laboratory of Mathematical Modelling, Shumen University, Bulgaria, 2002

"The Pegasus method for computing the root of an equation", M. Dowell and P. Jarratt, BIT 12 (1972) 503--508.

"The Java Programmer's Guide to Numerical Computation", Ronald Mak, Prentice-Hall. [//www.apropos-logic.com/nc/](http://www.apropos-logic.com/nc/)

"Numerical Analysis Topics Outline", S.-Sum Chow, 2002, [//www.math.byu.edu/~schow/work](http://www.math.byu.edu/~schow/work)

"Newton's method and high order iterations" , Xavier Gourdon , Pascal Sebah, in Number Costant and Computation , Oct. 2000, [//numbers.computation.free.fr/Constants](http://numbers.computation.free.fr/Constants)

"Radici di Equazioni Non Lineari con MathCad", A. M. Ferrari, in Appunti di LPCAC, Università di Torino

"Numerical Method - Rootfinding", in Applied Mathematics, Wolfram Reserch, [//mathworld.wolfram.com/](http://mathworld.wolfram.com/)

"Roots of Equations", John Burkardt, School of Computational Science (SCS), Florida State University (FSU). [//www.csit.fsu.edu/~burkardt/math2070/lab_03.html](http://www.csit.fsu.edu/~burkardt/math2070/lab_03.html)

"Zoomin" a miscellanea of rootfinding subroutines in Fortran 90 by John Burkardt, 2002

"ESFR Fortran 77 Numeric Calculus Library", ESFR, 1985

"Handbook of Mathematical Functions", by M. Abramowitz and I. Stegun, Dover Publication Inc., New York.

"Zeros of Orthogonal Polynomials" , Leonardo Volpi, Foxes Team ([//digilander.libero.it/foxes/Documents](http://digilander.libero.it/foxes/Documents))

"Solutions Numeriques des Equations Algebriques", E., Durand, Tome I, Masson,Paris ,1960.

"A modified Newton method for Polynomials" , W.,Ehrlich, Comm., ACM, 1967, 10, 107-108.

"Iteration methods for finding all the zeros of a polynomial simultaneously", O. Aberth, Math. Comp. ,1973, 27, 339-344.

"The Ehrlich-Aberth Method for the nonsymmetric tridiagonal eigenvalue problem", D. A. Bini, L. Gemignani, F. Tisseur, AMS subject classifications. 65F15

"Progress on the implementation of Aberth's method", D. A. Bini, G. Fiorentino, , 2002, The FRISCO Consortium (LTR 21.024)

"Numerical Computation of Polynomials Roots: MPSolve v. 2" , D. A. Bini, G. Fiorentino, Università di Pisa, 2003

"Geometries of a few single point numerical methods", Bryan McReynolds, US Military Academy, West Point, New York, 2005

"Calculo Numérico", Neide M. B. Franco, 2002

"Metodos Numericos" Sergio R. De Freitas, Universidade Federal de Mato Grosso do Sul, 2000

"Appunti di Calcolo Numerico", Enrico Bertolazzi, Gianmarco Manzini, Università di Trento, C.N.R. di Pavia

"Zeros of orthogonal polynomials", Leonardo Volpi, Foxes Team, 2003, //digilander.libero.it/foxes

"Iterative Methods for Roots of Polynomials", Wankere R. Mekwi, University of Oxford, Trinity, 2001.

"MultRoot - A Matlab package computing polynomial roots and multiplicities", Zhonggang Zeng, Northeastern Illinois University, 2003

Appendix

Complex power of the binomial (x+iy)

$$(x + iy)^2 = x^2 - y^2 + 2ixy;$$

$$(x + iy)^3 = x(x^2 - 3y^2) + iy(3x^2 - y^2);$$

$$(x + iy)^4 = x^4 - 6x^2y^2 + y^4 + 4ixy(x^2 - y^2);$$

$$(x + iy)^5 = x(x^4 - 10x^2y^2 + 5y^4) + iy(5x^4 - 10x^2y^2 + y^4);$$

$$(x + iy)^6 = (x^2 - y^2)(x^4 - 14x^2y^2 + y^4) + 2ixy(3x^4 - 10x^2y^2 + 3y^4);$$

$$(x + iy)^7 = x(x^6 - 21x^4y^2 + 35x^2y^4 - 7y^6) + iy(7x^6 - 35x^4y^2 + 21x^2y^4 - y^6);$$

$$(x + iy)^8 = x^8 - 28x^6y^2 + 70x^4y^4 - 28x^2y^6 + y^8 + 8ixy(x^2 - y^2)(x^4 - 6x^2y^2 + y^4);$$

$$(x + iy)^9 = x(x^8 - 36x^6y^2 + 126x^4y^4 - 84x^2y^6 + 9y^8) + iy(9x^8 - 84x^6y^2 + 126x^4y^4 - 36x^2y^6 + y^8)$$



© Jan. 2006, by Foxes Team
ITALY

1. Edition