[6.61] Find more accurate polynomial roots

solve() may return false solutions to ill-conditioned polynomials. An ill-conditioned polynomial is one in which small changes in the coefficients cause large changes in the roots. Since *zeros()* uses *solve()*, *zeros()* can return the same incorrect solutions. This tip gives an example, shows how to check the results and gives alternative methods for finding the better solutions. In general (and as usual), finding polynomial roots is not trivial, and the theoretic best attainable accuracy might be worse than you would expect, even for polynomials of relatively low degree.

For some examples in this tip, I define a 'best' solution for a root as *x* such that f(x+e) and f(x-e) are of opposite sign, or that one or both are zero, and *e* is the least significant digit of the floating-point mantissa. I will call this the *sign test*. There are better tests for root solutions, and I will discuss some of those as well.

The 'best' value for a polynomial root depends on what you want to do with it. Some common criteria for a root *x* are

1. *x* such that |f(x)| is as 'small' as possible. Ideally, |f(x)| would be zero, but this is unlikely with the finite resolution of floating-point arithmetic and its round-off errors.
2. *x* such that |f(x)| is as small as *required*. For problems based on physical measurements, it may be a waste of time to find the true minimum, since measurement error prevents such an accurate solution, anyway.
3. The polynomial coefficients can be reconstructed as accurately as possible from the roots. This is the same as saying that the errors in coefficients of the reconstructed polynomial are minimized. The polynomial is reconstructed with the roots z_0, z_1, ... z_n as f(x) = $(x-z_0)(x-z_1)...(x-z_n)$
4. Some other various conditions for polynomial roots are met.

The conditions in criteria 4 may include these properties of polynomials:

$$\sum_{i=1}^{n} z_i = \frac{a_{n-1}}{a_n} \qquad \sum_{i>j} z_i z_j = \frac{a_{n-2}}{a_n} \qquad z_1 z_2 z_3 ... z_n = (-1)^n \frac{a_0}{a_n}$$

for this polynomial, with roots z_1, z_2, ... z_n

$$f(x) = a^n x_n + a^{n-1} x_{n-1} + ... + a_1 x + a_0$$

Or, we may want to use the roots to evaluate the polynomial in this form:

$$f(x) = a_n (x - z_1)(x - z_2)...(x - z_n)$$

The point is that different root-finding algorithms can return slightly different roots (or fail completely to find all or any roots), so, as usual, you really need to know why you want the roots to get useful results.

Getting back to the failure of *solve()*, I'll use this polynomial as an example:

$$y1(x) = x^3 + 4.217E17 \cdot x^2 - 3.981E20 \cdot x - 6.494E22 \qquad [1]$$

Define this polynomial in the Y= editor, then `zeros(y1(x),x)`

returns these candidate solutions: `{-4.217E17, -141.81969643465, 0}`

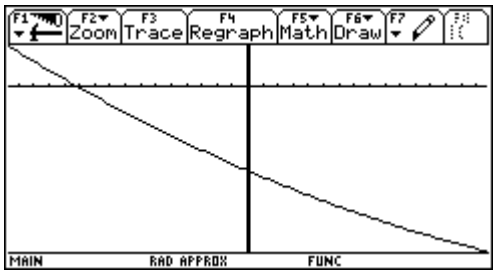
We find the function values at the solutions with `y1({-4.217E17, -141.81969643465, Ø})`

which returns (approximately) `{1.67879E38, 1E9, -6.494E22}`

These values may not seem 'close' to zero, still, they may be the best roots as defined above. The first step is to test each root. Command-line and TI Basic calculations on the TI-89 / TI-92 Plus are performed with a 14-digit mantissa, so we need to find the value of *e* for each root. If a root is expressed in scientific notation as a.bEc, where *a.b* is the mantissa, *a* is not equal to 0 and *c* is the exponent, then e = 10^(c-13). For example, for the first root of -4.217E17, e = 10^(17 - 13) = 1E4. The table below shows the verification results for all three roots.

root	y1(x-e)	y1(x+e)
-4.217E17	-1.610E39	1.946E39
-141.8196 9643 465	7E9	-3E9
0	-6.494E22	-6.494E22

Since the signs of y1(x-e) and y1(x+e) are different for the first two roots, they are at least plausible. However, the third root of zero fails the test. To verify that zero is not a root, we plot the function over the range of x = -200 to x = 200:



The graph shows the root at about x = -141.8, but x = 0 is clearly not a root.

The sign test is automated with the following function *proot_t1()*. The arguments are the list of polynomial coefficients and a list of at least one root, and it returns a list of boolean *true* and *false* results: *true* if the corresponding root passes the sign test, and *false* if it fails.

```
proot_t1(c,r)
Func
©(coefList,rootList)
©Sign test for polynomial roots
©Calls math\mantexp()
©14junØ2/dburkett@infinet.com

local i,o,e

{}→o                                © Initialize result list
for i,1,dim(r)                      © Loop to test each root
 1Ø^(math\mantexp(r[i])[2]-13)→e    © Find least significant digit
 © Test passes if f(x)=Ø, or f(x-e), f(x+e) have opposite sign or are zero
 polyeval(c,r[i])=Ø or polyeval(c,r[i]-e)*polyeval(c,r[i]+e)≤Ø→o[i]
endfor

return o

EndFunc
```

This call tests the roots returned by *solve()*:

```
proot_t1({1,4.217E17,-3.981E20,-6.494E22},{-4.217E17,-141.81969643465,0})
```

which returns {*true,true,false*}, indicating that the first two roots pass the sign test and the last root fails. Note the *proot_t1()* also tests for f(x) = 0 for each root, and returns *true* in that case.

proot_t1() can only be used to test real roots of polynomials with real coefficients. The method used to perturb the roots for the test does not make sense in the complex plane.

Another method to check polynomial solutions is to expand the roots into a polynomial and compare the resulting coefficients with the original coefficients. One way to do this is

```
expand(product(x-{zn, ... z0}))
```

where {zn, ... z0} is a list of the roots and x is a symbolic variable. Applying this method to the roots above with

```
expand(product(x-{0,-4.217E17,-141.81969643465}))
```

gives

$$x^3 + 4.217E17 \cdot x^2 + 5.9805365986492e19 \cdot x$$

The two highest-order coefficients are the same, but the original constant term has disappeared and the coefficient for *x* has the wrong sign, let alone the right value. By now you should be starting to suspect that *solve()* has failed us.

You might be tempted to try *factor()* or *cFactor()* to find the roots, but they fail in the same way as *solve()*, returning (x+0) as one of the factors.

The following sections show a few methods to find the correct result for this root:

Method 1: Invert polynomial coefficients	(routine *prooti()*)
Method 2: Laguerre's algorithm	(routine *polyroot()*)
Method 3: Eigenvalue methods	(routine *proots()*)

These methods have trade-offs in speed, accuracy and code size, as well as their abilities to find all the real and complex roots.

Method 1: Invert polynomial coefficients

An interesting property of polynomials is that the roots map onto the reciprocals of the roots of the polynomial with the coefficients reversed, that is, if

$$f_1(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \quad \text{and} \quad f_2(x) = a_0x^3 + a_1x^2 + a_2x + a_3$$

and the solutions to $f_2(x)$ are z_1, z_2 and z_3, then the solutions to $f_1(x)$ are $1/z_1$, $1/z_2$ and $1/z_3$. Sometimes solving this 'reversed' polynomial gives better solutions than the original polynomial. Using this method and *zeros()* with our example polynomial, we get

z1 = -141.8196 9643 465 (7E9, -3E9)

z2 = -4.2169 9999 9999 9 E17 (1.68E38, 3.72E39)
z3 = 1085.8557 4101 61 (-5E10, 4.9E10)

where the numbers in parentheses are y1(x-e) and y1(x+e). This method returns the correct first and third roots, but the second root fails the sign test.

If you save your polynomials in coefficient list form and evaluate them with *polyEval()*, it is easy to use the list reversal method in tip [3.4] to reverse the coefficients. Bhuvanesh Bhatt's MathTools package also include a list reversal function, *reverse()*.

The following function *prooti()* can be used to find the roots with the inversion method.

```
prooti(c)
Func
©(coefList) Poly roots by inversion
©11junØ2/dburkett@infinet.com

local ï

seq(c[ï],ï,dim(c),1,⁻1)→c            © reverse coefficients
return 1/(zeros(polyeval(c,ä),ä))    © solve for roots and invert

EndFunc
```

Method 2: Laguerre's algorithm

Another alternative is to use a completely different algorithm to find the polynomial roots. *solve()* may recognize a polynomial and so use a specific algorithm to find its roots, in any event, it doesn't work for our example, and Laguerre's algorithm does. The function shown below, *polyroot()*, combines the code for the programs *laguer()* and *zroots()* from *Numerical Recipes in FORTRAN*. Call *polyroot()* with the polynomial coefficients as a list:

```
polyroot({an, ... aØ})
```

and the roots are returned as a list. You *must* set the Complex Format mode (with [MODE]) to RECTANGULAR before running *polyroot()*, since it uses complex arithmetic even for real roots. Also set the Exact/Approx mode to APPROX, or *polyroot()* will take an extremely long time.

polyroot() may return the string *"polyroot too many its"* if a solution does not converge in 80 iterations or less, for each root. The call for our example is

```
polyroot({1,4.217E17,-3.981E2Ø,-6.494E22})
```

which returns these roots in about 7 seconds:

```
{-4.217ØØØØØØØØØ1,1Ø85.8557410161,-141.81969643465}
```

All of these roots pass the sign test, and the coefficients are reconstructed to, at worst, 5 least significant digits.

Some comments on the operation of *polyroot()* follow this code listing.

```
polyroot(a)
Func
©({an,...,aØ}) return roots of polynomial
```

```
©Set Complex format to Rectangular
©12may02/dburkett@infinet.com

local m,roots,eps,maxm,i,j,jj,ad,x,b,c,root


©-------------------------------------------------------------
© Define local function to find 1 root with Laguerre'e method

Define root(a,x)=Func
©({coeffs},guess) polynomial root
©NOTE: {a} is reversed!

local m,maxit,mr,mt,epss,iter,j,abx,abp,abm,err,frac,dx,x1,b,d,f,g,h,sq,gp,gm,g2

dim(a)-1→m                © Find order of polynomial
2E⁻12→epss                © Set estimated fractional error
8→mr                      © Set number of fractional values to break limit cycles
10→mt                     © Set number of steps to break limit cycle
mt*mr→maxit               © Maximum number of iterations

{.5,.25,.75,.13,.38,.62,.88,1.}→frac      © Fractions used to break limit cycle

for iter,1,maxit          © Loop to find a root
 a[m+1]→b
 abs(b)→err
 0→d
 0→f
 abs(x)→abx
 for j,m,1,⁻1             © Evaluate the polynomial and first two derivatives
  x*f+d→f
  x*d+b→d
  x*b+a[j]→b
  abs(b)+abx*err→err
 endfor
 epss*err→err             © Estimate roundoff error in polynomial evaluation
 if abs(b)≤err then       © Return root if error condition met
  return x
 else                     © Execute Laguerre's method
  d/b→g
  g^2→g2
  g2-2.*f/b→h
  √((m-1)*(m*h-g2))→sq
  g+sq→gp
  g-sq→gm
  abs(gp)→abp
  abs(gm)→abm
  if abp < abm: gm→gp
  if max(abp,abm)>0 then
   m/gp→dx
  else
   e^(ln(1+abx)+iter*i)→dx
  endif
 endif
 x-dx→x1
 if x=x1:return x         © Converged to root
 if mod(iter,mt)≠0 then   © Every mt steps take a fractional step to break (rare) limit cycle
  x1→x
 else
  x-dx*frac[iter/mt]→x
 endif
endfor

return "polyroot too many its"    © May have this error for complex roots
EndFunc

©-------------------------------------------------------------
© Begin polyroot()

dim(a)-1→m                        © Find order of polynomial
```

```
1E-12→eps                          © Set complex part limit
newlist(m)→roots                   © Create list to hold roots

seq(a[i],i,dim(a),1,-1)→a          © reverse coefficients for simpler looping

a→ad                               © Copy coefficients for deflation

for j,m,1,-1                       © Loop to find each root
 Ø→x                               © Set guess to zero to find smallest remaining root
 root(ad,x)→x                      © Find the root
 if gettype(x)="STR":return x      © Check for error from root()
 if abs(imag(x)) ≤ 2*eps^2*abs(real(x))   © Remove imaginary part of root if very small
  real(x)→x
 x→roots[j]                        © Save the root
 ad[j+1]→b                         © Perform forward deflation on remaining coefficients
 for jj,j,1,-1
  ad[jj]→c
  b→ad[jj]
  x*b+c→b
 endfor
 left(ad,j)→ad                     © Keep only deflated coefficients
endfor

for j,1,m                          © Polish roots with undeflated coefficients
 root(a,roots[j])→roots[j]
 if gettype(roots[j])="STR":return roots  © On error, return roots so far
endfor

return roots

EndFunc
```

polyroot() will also work for polynomials with complex coefficients.

I will not describe Laguerre's algoithm in detail, but you can find additional description in the references for Acton and Press below.

polyroot() first finds estimates for all the roots, then the roots are 'polished' to improve accuracy. For the estimates, *polyroot()* finds the smallest remaining root, then creates a new polynomial by dividing that root out of the polynomial. This process is called *deflation*. The new polynomial has degree of one less than the original, since one root is divided out. Finding the smallest roots first reduces errors from round-off; otherwise the small roots might disappear completely. Since deflating the polynomial can introduce rounding errors by changing the coefficients, the second pass of polishing the roots finds the roots again, this time using the original, undeflated coefficients, with the roots just found as initial guesses.

As with any feedback-based iterative method, it is possible for the algorithm to get stuck, oscillating between two solutions which do not meet the termination criteria. This oscillation is called a limit cycle, and *polyroot()* attempts to break limit cycles by periodically perturbing the current solution by some fractional amount. *mt*, *mr* and *frac* implement the cycle-breaking process such that a fractional step size from *frac* is taken every *mt* steps. As shown, the total number of iterations to find a root is 80. If a root isn't found in 80 iterations, *polyroot()* gives up and returns an error message. This would be unusual but may occur with complex roots. If this error occurs, *polyroot()* returns a list of the roots found up to the point of error, and the last element is the string "*polyroot too many its*". For example, if the program finds three roots 1, 2 and 3, but fails on the fourth root, the returned list is

```
{1,2,3,"polyroot too many its"}
```

If you call *polyroot()* from another program, you can test for this condition with

```
root[dim(root)]→r
```

```
getType(r)="STR"
```

polyroot() uses Laguerre's algorithm for both the initial root-finding and the polishing, however, this is not mandatory: you could use some other method for the polishing. The references at the end of this tip discuss this more and give some suggestions.

Since *polyroot()* uses complex arithmetic even to find the real roots, we have a dilemma I have not encountered before: when do we decide that a complex root is *actually* real, and only has a small complex part because of round-off error? It may be that the root really does have a small complex part, in that case we don't want to throw it away. Conversely, we don't really want to return a small complex component when the root is real. Press and his co-authors use this criteria, for a complex root of x = a + b*i:

$$\text{if } |b| \le 2 \cdot \text{eps}^2 \cdot |a| \text{ then } a \to x$$

eps is the desired relative accuracy of f(x). Press gives no justification for this criteria, and I could not derive it from several reasonable assumptions. So, I posted this as a question on the Usenet newsgroup sci.math.num-analysis. Mr. Peter Spellucci was kind enough to answer with this:

> *"I think this is one of the typical ad hoc "equal to zero within the roundoff level" decisions one often is forced to use. First of all, this makes sense only for a real polynomial, and I assume that this decision is applied simultaneously for the conjugate complex value. In Laguerre's method there appears the following:*
>
> *denom= p'+sign(p')*sqrt((n-1)*((n-1)*p'^2-n*p*p'')))*
>
> *and if the radicand is negative, it branches into the complex plane. Hence a sound decision would be based on the possible roundoff level in the evaluation of this expression, in the first position on the possible roundoff errors in the evaluation of p, p' and p''. Bounds on this can be computed within the evaluation for example using Wilkinson's techniques. But this bounds may be too pessimistic and hence one usually relies on some rules of thumb. In the real case with a complex zero z, also conj(z) will be a zero. Multiplying out we get a quadratic factor*
>
> *x^2-2*re(z)*x+abs(z)^2*
>
> *and if this factor would not change within the computing precision by neglecting the imaginary part I would set it to zero. This however would mean |b| < |a|*sqrt(eps) , if eps represents the computing precision. Hence I wonder a bit about the settings you mention."*

There is quite a difference between Peter's criteria and that used in NRIF: compare 1.4E-6 to 2E-24, if we use *eps* = 2E-12. Peter's criteria, while theoretically quite sound, seems rather extreme in throwing away complex roots. At this point, I discovered that the authors of *Numerical Recipes* run a forum for this type of question, so I posted:

> *In the xroots() code on p367 of NRIF, a complex root x = a + bi is forced real if |b| <= 2*eps**2*|a|. I cannot derive this from a few different assumptions; the closest I can come is b^2 <= 2*eps*|a|, assuming that the root is forced real if |x| - |a| <= eps. How is this criteria derived?*

One of the authors of *Numerical Recipes*, Saul Teukolsky, answered:

> *The criterion in the Recipe, using eps**2, is purely empirical. After all, you may well have a root that has a very small imaginary part that is meaningful. But your criterion, with eps, is perfectly*

reasonable to use instead. It assumes that any small imaginary part cannot be determined with an accuracy better than eps relative to the real part.

In the meantime, I had tested *polyroot()* with about 30 polynomials from textbooks and numerical methods books, and in all cases it returned both real and complex roots, as appropriate. While empirical, the criteria in *polyroot()* seems to work pretty well. Perhaps the best approach is to try the criteria now used in *polyroot()*, but if you can't find the roots you want, then try one of the other two criteria.

*Method 3: Eigenvalue method*s

We can find the roots of a polynomial by finding the eigenvalues of this *m* x *m* companion matrix:

$$\begin{bmatrix} -\frac{a_m}{a_{m+1}} & -\frac{a_{m-1}}{a_{m+1}} & \dots & -\frac{a_2}{a_{m+1}} & -\frac{a_1}{a_{m+1}} \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}$$

where the eigenvalues will be the roots of the polynomial

$$P(x) = \sum_{i=1}^{m+1} a_i x^{i-1}$$

cybernesto has written a function called *proots()* which implements this method:

```
proots(aa)
Func
Local nn,am
dim(aa)→nn
aa[1]→am
right(aa,nn-1)→aa
list▸mat(aa/(⁻am))→aa
augment(aa;augment(identity(nn-2),newMat(nn-2,1)))→aa
eigVl(aa)
EndFunc
```

The input argument *aa* is the list of polynomial coefficients. For our example polynomial,

$$y1(x) = x^3 + 4.217E17 \cdot x^2 - 3.981E20 \cdot x - 6.494E22$$

proots() returns the candidate roots in about 1 second, compared to 6.4 seconds required for Laguerre's method as implemented in *polyroot()* above. Also, *polyroot()* is about 1773 bytes, while *proots()* is only 156. Unfortunately, the root at about -141.8 returned by *proots()* fails the sign test.

As another example, consider the polynomial

$$p(x) = (x+1)^{20} =$$

$$x20 + 20x^{19} + 190x^{18} + 1140x^{17} + 4845x^{16} + 15504x^{15} + 38760x^{14} + 77520x^{13} + 125970x^{12} +$$

$$167960x^{11} + 184756x^{10} + 167960x^9 + 125970x^8 + 77520x^7 + 38760x^6 +$$

$$15504x^5 + 4845x^4 + 1140x^3 + 190x^2 + 20x + 1$$

Obviously, there are 20 repeated roots of -1, but *proots()* returns (approximately) these roots in about 100 seconds:

-1.49339 ± 0.102038i	-0.848193 ± 0.295003i
-1.40133 ± 0.280223i	-0.843498 ± 0.25989i
-1.24788 ± 0.388669i	-0.750275 ± 0.212911i
-1.08333 ± 0.41496i	-0.703918 ± 0.130283i
-0.945259 ± 0.38717i	-0.682926 ± 0.043651i

If we substitute these roots in the original polynomial, we get 20 complex results near zero, but *proots()* has returned 10 complex root pairs, instead of the expected 20 real results. In about the same 100 seconds, *polyroot()* (Laguerre's method) returns the correct roots as a list of twenty 1's. The results from *proots()* are disturbed from the correct solutions by round-off error in calculating the eigenvalues of a 20 x 20 element matrix.

TI offers a flash application, *Polynomial Root Finder*, (I'll call it TI PRF) which also uses this eigenvalue method according to its PDF documentation. TI PRF also fails in a rather spectacular fashion with the 20-degree polynomial above, but with different results than *proots()*. The flash application returns two real roots of about -1.49821 and -0.869716, and 9 complex conjugate pairs. If these roots are evaluated in the polynomial, we again get real and complex results that are in the vicinity of zero. The magnitudes range from about 1E-6 to 2E-10.

The real drawback to these eigenvalue methods is that they give no estimate for the root errors, and no warning that the roots are wrong. Since we usually do not know in advance what the roots are, we might be led to believe that the returned roots were correct. The main advantage to *polyroot()* is that it attempts to find the roots to meet an error criteria, and, if it cannot, it returns an error message instead of a wrong solution.

Comments

This procedure can be used to find polynomial roots with a fair degree of success:

1. First, try *solve()* or *csolve()* (or *zeros()* or *czeros()*) to find the roots. If they work, they will be faster than *polyroot()*, and they can find some complex solutions which *polyroot()* cannot.
2. Check the solutions with the sign test method (*proot_t1()*), or by creating the original polynomial with the roots, or just by evaluating y(x) at each root. You can also plot the polynomial in the vicinity of the roots for additional assurance.
3. If the tests in step 2 fail, try *proots()* and repeat the tests.
4. If *proots()* fails, try *polyroots()*.

The limited floating-point resolution can result in an effect called destructive cancellation, the symptom of which is that the polynomial may not be strictly monotonic (increasing or decreasing) over a very narrow range of x. For this reason, the sign test is not always conclusive. Plotting f(x) - f(xc), where *xc* is a point in the center of the x-range, may help indicate this problem.

Solving polynomials is not trivial, particularly when the polynomial is ill-conditioned, and polynomials with multiple identical roots are always ill-conditioned. An ill-conditioned polynomial has a large dynamic range of coefficients, and the coefficient signs alternate. Forman S. Acton (reference below), has this to say about polynomial-solving systems in general:

"A system cannot do everything: There will always be some extremely nasty polynomials which will frustrate it and cause it to fail to find some of the roots. A system should handle most of the polynomials it is apt to encounter, but we cannot insist on perfection, lest the program become a monster that demands more than its share of library space and requires large amounts of running time looking for subtle difficulties that are not usually present. We must, however, insist that in the nasty situations the system should announce its difficulty, lest the trusting user be misled into thinking that all the roots have been found, and with the precision he desires."

Considering that the TI-89 and TI-92 Plus are intended for educational applications, it is perhaps reasonable that it fail to solve our example polynomial, which is, however, taken from a problem in mechanical engineering (strength of materials). However, as Mr. Acton says, it is not acceptable that it returns a clearly incorrect root without a warning message.

This situation is somewhat puzzling considering that the TI-86 returns, very quickly, three correct solutions. The TI-89 / TI-92 Plus probably use a more general algorithm, while the TI-86 uses an algorithm specifically designed for polynomials.

Some references claim that it is better to normalize the polynomial coefficients by dividing them by the leading coefficient, so that coefficient becomes one. Other references claim that normalization introduces additional rounding errors, and so is best avoided.

Sometimes your polynomial may be in the form of an expression instead of a list of coefficients. There are available TI Basic which extract the coefficients from an expression with repeated differentiation. This is a clever technique which works great for symbolic coefficients, but it adds round-off error which will affect the solutions. The ideal solution would be to extract the coefficients with *part()*, which avoids any round-off error. I am working on such a program, but as it is not finished, the derivative method is better than nothing. Here is one version, written by J.M. Ferrard:

```
polyc(p,x)
Func
©(f(x),x) polynomial coefficients
©J.M. Ferrard

Local c,k
Ø→k

While when(p=Ø,false,true,true)
 p/k!|x=Ø→c[k+1]
 d(p,x)→p
 k+1→k
EndWhile

seq(c[k],k,dim(c),1,⁻1)

EndFunc
```

I have added the final *seq()* line so that the coefficients are returned in descending order of the independent variable powers. A call of

```
polyc(3*x^2+2*x-1,x)
```

returns {3,2,-1}

References

Polynomial root-finding is an important, well-developed topic in numerical analysis, so most books on numerical methods or numerical analysis include a section on finding polynomial roots. Those that have been most helpful to me follow.

Numerical Recipes in FORTRAN; The Art of Scientific Computing
Press, Teukolsky, Vetterling and Flannery, 1992, Cambridge University Press.
Section 9.5 is devoted to finding roots of polynomials, and describes the processes of deflation and inverting the coefficients. Some other methods are discussed, including the eigenvalue technique (used by cybernesto's *proots()*) and mention of the Jenkins-Traub and Lehmer-Schur algorithms.

Numerical Methods that (usually) Work
Forman S. Acton, 1990, The Mathematical Association of America.
Acton discusses polynomial root-finding in all of chapter 7, *Strategy versus Tactics*. He dismisses Laguerre's method as "not sufficiently compatible with our other algorithms', but discusses it, anyway. Mr. Acton's objection seems to be the requirement for complex arithmetic, but points out that Laguerre's method is guaranteed to converge to a real root when all coefficients are real.

A survey of numerical mathematics, volume 1
David M. Young, Robert Todd Gregory, 1988, Dover.
Chapter 5 is a thorough (70 page) development and examination of the entire process of finding polynomial roots and determining if the roots are 'good enough'. Starting with general properties of polynomials, developed in a theorem and proof format, Young and Gregory continue by examining the methods of Newton, Lin and Lin-Bairstow, and the secant method. They proceed to the methods of Muller and Cauchy before developing procedures for finding approximate values of roots, including Descarte's rule of signs, Sturm sequences, and the Lehmer-Schur method. Acceptance criteria for real and complex roots are derived. Matrix-related (eigenvalue) methods are described, including Bernoulli, modified Bernoulli and inverse-power (IP). The chapter continues with a discussion of polyalgorithms, which consist of two phases of root-finding: an initial phase, and an assessment/refinement phase. In the final section, other methods are very briefly discussed, including Jenkins-Traub and Laguerre, which the authors describe as "An excellent method for determining zeroes of a polynomial having only real zeroes ..."

Rounding errors in Algebraic Processes
J.H. Wilkinson, 1994, Dover Publications.
Wilkinson devotes chapter 2 to polynomials and polynomial roots, and develops several important ideas in solving polynomials in general. This book is a 'classic' in numerical analysis.

Validating polynomial numerical computations with complementary automatic methods
Phillipe Langlois, Nathalie Revol, June 2001, INRIA Research Report No. 4205, Institut National de Recherche en Informatique et en Automatique.
(This paper is available at http://www.inria.fr/index.en.html.)
While the focus of this paper is on using stochastic, deterministic and interval arithmetic methods to estimate the number of significant digits in calculated polynomial roots, section 3 summarizes the best attainable accuracy of multiple root solutions. This paper is also the source of the example polynomial $p(x) = 1.47x^3 + 1.19x^2 - 1.83x + 0.45$.

The following web sites are useful or at least interesting:

Mcnamee's bibliography on roots of polynomials
http://www.elsevier.com/homepage/sac/cam/mcnamee/
No actual contents here, but an exhaustive bibliography which would be even more helpful with a little annotation. The categories are

Bracketing methods (real roots only). Newton's method. Simultaneous root-finding methods. Graeffe's method. Integral methods, esp. Lehmer's. Bernoulli's and QD method. Interpolation methods such as secant, Muller's. Minimization methods. Jenkins-Traub method. Sturm sequences, greatest common divisors, resultants. Stability questions (Routh-Hurwitz criterion, etc.). Interval methods. Miscellaneous. Lin and Bairstow methods. Methods involving derivatives higher than first. Complexity, convergence and efficiency questions. Evaluation of polynomials and derivatives. A priori bounds. Low-order polynomials (special methods). Integer and rational arithmetic. Special cases such as Bessel polynomials. Vincent's method. Mechanical devices. Acceleration techniques. Existence questions. Error estimates, deflation, sensitivity, continuity. Roots of random polynomials. Relation between roots of a polynomial and those of its derivative. Nth roots.

Polynomial sweep at the WWW Interactive Mathematics Server
http://wims.unice.fr/wims/wims.cgi?session=RNC912CC8A.3&+lang=en&+module=tool%2Falgebra%2Fsweeppoly.en

This is a very illuminating animation of polynomial roots plotted in the complex plane. The polynomial is expressed parametrically, then the polynomial and its roots are plotted as the parameter varies. It is fascinating to watch the roots move as the polynomial changes. You can use example polynomials, or enter your own. Very cool!