[6.12] Find faster numerical solutions for polynomials

As shown in tip [11.5], *nsolve()* is accurate and reliable in solving polynomial equations, but the speed leaves something to be desired. This tip shows a way to find the solution faster. This method requires that the equation to be solved is a polynomial, and that an estimating function can be found. This estimating function must be a polynomial, as well.

The basic method is to replace *nsolve()* with a TI Basic program that uses Newton's method, coupled with an accurate estimating function. Further speed improvements are possible since you control the accuracy of the solution.

The function is called *fipoly(),* and here is the code:

```
fipoly(clist,fguess,yval,yemax,yetype)
func
©Fast inverse polynomial solver
©26 nov 99 dburkett@infinet.com
©Find x, given f(x)
©clist: list of polynomial coefficients
©fguess: list of guess generating polynomial coefficients
©yval: point at which to find 'x'
©yemax: max error in 'y'
©yetype: "rel": yemax is relative error; "abs": yemax is absolute error.

local fd,dm,xg,yerr,n,erstr,nm

©Set maximum iterations & error string
30→nm
"fipoly iterations"→erstr

©Find list of derivative coefficients
dim(clist)→dm
seq(clist[k]*(dm-k),k,1,dm-1)→fd

©Find first guess and absolute error
polyeval(fguess,yval)→xg
polyeval(clist,xg)-yval→yerr

©Loop to find solution 'x'
0→n

if yetype="abs" then
 while abs(yerr)>yemax
  xg-(polyeval(clist,xg)-yval)/polyeval(fd,xg)→xg
  polyeval(clist,xg)-yval→yerr
  n+1→n
  if n=nm:return erstr
 endwhile
else
 yerr/yval→yerr
 while abs(yerr)>yemax
  xg-(polyeval(clist,xg)-yval)/polyeval(fd,xg)→xg
  (polyeval(clist,xg)-yval)/yval→yerr
  n+1→n
  if n=nm:return erstr
 endwhile

endif

xg

Endfunc
```

Again, this routine will only work if your function to be solved is a polynomial, and you can find a fairly accurate estimating function for the solution. (If you can find a *very* accurate estimating function, you don't need this routine at all!)

The function parameters are

clist	List of coefficients for the polynomial that is to be solved.
fguess	List of coefficients of the estimating (guess) polynomial
yval	The point at which to solve for x
yemax	The maximum desired y-error at the solution for x, must be >0
yetype	A string that specifies whether yemax is absolute error or relative error: "abs" means absolute error "rel" means relative error

fipoly() returns the solution as a numeric value, if it can find one. If it cannot find a solution, it returns the string "fipoly iterations". If you call *fipoly()* from another program, that program can use *gettype()* to detect the error, like this:

```
fipoly(...)→x
if getype(x)≠"NUM" then
 {handle error here}
endif
{otherwise proceed}
```

With *fipoly(),* you can specify the y-error *yemax* as either a relative or absolute error. If y_a is the approximate value and y is the actual value, then

$$\text{absolute error} = |y - y_a| \qquad \text{relative error} = \left|\frac{y - y_a}{y}\right|$$

You will usually want to use the relative error, because this is the same as specifying the number of significant digits. For example, if you specify a relative error of 0.0001, and the y-values are on the order of 1000, then *fipoly()* will stop when the y-error is less than 0.1, giving you 4 significant digits in the answer.

However, suppose you specify an absolute error of 1E-12, and the y-values are on the order of 1000 as above. In this case, *fipoly()* will try to find a solution to an accuracy in y of 1E-12, but the y-values only have a resolution of 1E-10. *fipoly()* won't be able to do this, and will return the error message instead of the answer.

As an example, I'll use the same gamma function approximation function from tip [11.5]. The function to be solved is

$$y = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6 + hx^7 + ix^8$$

where these coefficients are saved in a list variable called *fclist*:

a = 4.44240042385	b = -10.1483412133	c = 13.4835814713
d = -11.0699337662	e = 6.01503554007	f = -2.15531523837
g = 0.494033458314	h = -.0656632350273	i = 0.00388944540448

Using curve fitting software for a PC, I found this estimating function:

$$x = p + qy + ry^2 + sy^3 + ty^4 + uy^5$$

where these coefficients are saved in a variable called *fglist*:

p = -788.977246657 q = 3506.8808748 r = -6213.31596202
s = 5493.68334077 t = -2422.15013853 u = 425.883370029

So, to find x when y = 1.5, with a relative error of 1E-8, the function call looks like this:

fipoly(fclist,fglist,1.5,1E-8,"rel") which returns x = 2.6627...

Using the same test cases as in tip [11.5], the table below shows the execution times and errors in *x* for various maximum *y* error limits, for both the relative and absolute error conditions.

yemax	"abs" max x-error	"rel" max x-error	"abs" mean execution time, sec	"rel" mean execution time, sec
1 E-1	3.03 E-2	3.03 E-2	0.70	0.72
1 E-2	3.03 E-2	3.03 E-2	0.70	0.71
1 E-3	6.56 E-3	6.56 E-3	0.90	0.92
1 E-4	6.96 E-4	6.96 E-4	0.95	0.99
1 E-5	1.74 E-5	1.74 E-5	1.12	1.15
1 E-6	3.19 E-6	3.19 E-6	1.16	1.20
1 E-7	2.82 E-6	3.21 E-7	1.23	1.30
1 E-8	1.35 E-8	1.35 E-8	1.27	1.32
1 E-9	6.67 E-10	6.67 E-10	1.30	1.33
1 E-10	6.67 E-10	6.67 E-10	1.38	1.43
1 E-11	6.67 E-10	5.84 E-10	1.41	1.45
1 E-12	5.84 E-10	5.84 E-10	1.77	1.83

There is little point to setting the error tolerance to less than 1E-12, since the 89/92+ only use 14 significant digits for floating point numbers and calculations. For this function, we don't gain much by setting the error limit to less than 1E-9.

Note that this program is much faster than using *nsolve():* compare these execution times of about 1.3 seconds, to those of about 4 seconds in tip [11.5].

The code is straightforward. The variable *nm* is the maximum number of iterations that *fipoly()* will execute to try to find a solution. It is set to 30, but this is higher than needed in almost all cases. If Newton's method can find an answer at all, it can find it very quickly. However, I set *nm* to 30 so that it will be more likely to return a solution if a poor estimating function is used.

I use separate loops to handle the relative and absolute error cases, because this runs a little faster than using a single loop and testing for the type of error each loop pass.