Boldly Going – Outsmarting PROOT

1. Introduction

Welcome to a new article belonging to my *"Boldly Going"* series, which strives to provide simple solutions for difficult subjects. This article features a subprogram, **PZER**, which deals once and for all with a glaring limitation of the *HP-71B Math Pac*'s keyword **PROOT**, the polynomial rootfinder which finds all roots of a polynomial of degree ***N***:

$$P(z) = a_N z^N + a_{N-1} z^{N-1} + \cdots + a_2 z^2 + a_1 z + a_0$$

The "*glaring limitation"* is that **PROOT** works exclusively for polynomials having *real* coefficients. On the other hand, **PZER** works efficiently and accurately for polynomials whose coefficients are *real* and or *complex* numbers.

2. Comparison with PROOT

As stated, **PROOT** only deals with *real* coefficients, so when working with polynomials having *complex* coefficients **PROOT** just can't cope, it's left completely out of the picture and **PZER** steals the show, period.

However, for polynomials with real coefficients we can compare them from a technical point of view, despite the obvious advantages in speed/accuracy that **PROOT** inherently has due to its assembly-language nature. Let's see:

PROOT	***PZER***
PROOT Advantages	
***Assembly**-language speed*	*Plain-vanilla **BASIC** code*
15-digit mantissa ± 50,000 exponent [1]	12-digit mantissa ± 499 exponent
Real polynomials so less arithmetic operations [2]	Complex polynomials so 2x-4x arithmetic operations
No need to compute conjugate roots [3]	Has to compute every root, conjugate or not
	PZER Advantages
*Only works for **real** polynomials* [4]	*Works for **real** and **complex** polynomials*
Computes roots one at a time, uses deflation [5]	Computes all roots at once, doesn't need deflation
Can't watch convergence to the roots [6]	Optionally can watch convergence on the fly
Doesn't return any extra info [7]	Optionally can return tolerance achieved and iterations used
Doesn't indicate if accuracy degraded [8]	Indicates if the specified tolerance wasn't achieved
User can't specify tolerance desired [9]	Optionally accepts any user-supplied tolerance
User can't specify a max. number of iterations	Optionally accepts any max. number of iterations
User can't provide initial guesses for the roots [10]	Optionally accepts initial guesses for all the roots

[1] The extended range improves accuracy and helps avoid internal underflows/overflows (see *HP Journal July 1984 pp.35*).
[2] For real arithmetic it takes 4x less operations, and for eval. at complex roots, only 2x *(see HP Journal July 1984 pp.34)*.
[3] As the polynomial is real, **PROOT** knows that complex roots come in conjugate pairs and computes just one of them.
[4] Can't be used to compute *eigenvalues* of complex matrices by finding the roots of the *Characteristic Polynomial.*
[5] Using deflation can seriously degrade accuracy for high-degree polynomials, only using 15-digit mantissas alleviates it.
[6] Watching convergence speed is important to determine whether there are multiple/clustered roots and degraded accuracy.
[7] **PROOT** doesn't return information such as the tolerance achieved and the iterations used, which can indicate problems.
[8] After it completes, you can't know if accuracy was degraded (probably due to multiple or very close roots).
[9] For real-world data or if you don't need more than, say, 5-6 digits, this saves time by avoiding unnecessary computation.
[10] For cases where the user has a fair idea of the location of the roots, there's no way to tell **PROOT** to speed up the process.

2.1. Preprocessing

While describing **PROOT**'s algorithm, the *Hewlett-Packard Journal July 1984* issue says in *p.33*:

> *"Before the root finder is initiated, leading zeros, trailing zeros, NaNs, and Infs alre all weeded out of the coefficient array."*

This means that the coefficient array is scanned and all these elements are taken care of if present. For instance, if **NaN**s are found, all roots are set to (**NaN**, **NaN**) and the process ends immediately. Next, if ±**Inf**s are found, every finite coefficient is set to **0**. Then *trailing zeros* are removed and as many (**0**, **0)** roots are placed in the roots array, decreasing the polynomial's degree by an equal amount. Same with *leading zeros*, but the roots are (**Inf**, **Inf)** instead.

PZER does nothing like this. Doing that kind of preprocessing, while perfectly possible, does increase the size of the subprogram, its complexity and the running time for no real gain. The goal of this article is to present a short, simple subprogram which works reliably and suitably fast for the kinds of polynomials that *do* appear in practice, and that doesn't include ones with **NaN** and **Inf** elements, which can be considered pretty *abnormal*.

Wasting time and memory catering for that is simply unwarranted. If the user or some process has included such elements among the coefficients, the execution will eventually *error out* or the results will obviously be *meaningless*, depending on the traps settings, and that will be indication enough that something's seriously *amiss* with the polynomial. Same with *leading zeros*. If the leading coefficient happens to be **0**, which isn't allowed, the polynomial's degree isn't N but *less* and the execution will terminate with a *Divide by Zero* error. Just don't do it.

2.2. Algorithm

PROOT's complicated and lengthy algorithm is thoroughly described in the *Hewlett-Packard Journal July 1984* issue *pp.33-36*. In particular, it says that **PROOT** computes the roots *one at a time*:

> *"The basic iteration used is Laguerre's method [...] It is the cubic convergence of this method that made it attractive [...]"*

and the details are pretty convoluted, to ensure convergence to each and every root, which is immensely facilitated by the mere fact that **PROOT** doesn't work with polynomials having *complex* coefficients, only with *real* coefficients., which has the following important advantages: (1) it enormously simplifies finding the annuli which contains each subsequent root (several theoretical bounds are computed) and thus obtaining a good initial approximation for each; (2) it also allows for much simpler and faster *real* arithmetic; (3) only half the complex roots need be computed, as they must necessarily appear as conjugate pairs. The algorithm also caters for the detection of when a root should be considered a root, plus lots of corrective measures if the iterations don't converge, the *Laguerre* step goes amiss, etc.

All of this is done *root by root*, and once found the polynomial is *deflated*, reducing its degree by one or two, then the whole process restarts for the next root. Deflation tends to degrade accuracy and stability very quickly and very seriously, easily resulting in losing digits and even turning *real* roots into *complex* ones, and the effect is quickly amplified for large-degree polynomials, which require many consecutive deflations. **PROOT** avoids the worse by taking advantage of the 15-digit mantissas and ±50,000 exponents available only to *assembly-language* keywords.

Also, **PROOT** uses *Laguerre's method*, which converges cubically (i.e.: once cubic convergence starts, the number of correct digits triples at each iteration) but has a *very convoluted*, *costly* step per iteration, which includes evaluating the polynomial *and* its first derivative *and* its second derivative at each step, plus assorted arithmetic operations, including a square root. All of this is tolerable because **PROOT** is written in assembly-language and has an enormous advantage in execution speed, like the one it also has in accuracy/range, so making possible the use of costly steps.

PZER on the other hand doesn't have *any* of those advantages, as it's written in plain-vanilla *10x-100x* slower *BASIC* code and it's only 407-byte long. Further, it deals with *complex*-coefficients polynomials, which in itself implies *2x-4x* more arithmetic operations than **PROOT**'s *real*-coefficients ones. Finally, **PZER** can't use extended accuracy or range and has to make do with just the user's accessible 12-digit mantissas and ±499 exponents.

Taking all those limitations into account, **PZER** doesn't go for the complicated, costly cubically-convergent *Laguerre's method* (it would be too slow if implemented in *BASIC*) but for a *quadratically*-convergent method which *doesn't* require generating and evaluating the first and second derivatives at each step, just the polynomial.

This means that each step if much, much simpler, taking just a few arithmetic operations (and no square roots), and can be coded in very few lines, with less decisions to take at each step. In consequence, *it all runs much faster per step* than doing *Laguerre*'s, so that even if quadratic convergence takes more iterations than cubic convergence, each iteration runs *much faster*. That's an important consideration usually not taken into account.

PZER also doesn't go for carefully calculating annuli specific to each individual root (using various theoretical bounds), nor does it try to compute first the *smallest* root to minimize the adverse effects of the always-fearful subsequent *deflation*. What **PZER** does is to generate a simple initial approximation in bulk for the roots (see ***Appendix B***) and then proceeds to compute ***all roots at once***, so if it needs 15 iterations for a 30^{th} degree polynomial, those 15 iterations will get you *all* 30 complex root at once, i.e.: not 15 iterations for *one* root but 15 iterations in all for *all* 30 roots. Of course this means that *deflation isn't needed* (or even possible!) and thus frequently the roots are accurate to the *full 12 digits* available, despite the accuracy/range limitations.

2.3. Multiple roots

As most rootfinder procedures, **PZER** works at its best if the roots are *single* and preferably well *separated* from one another, which fortunately is usually the case for real-life and random polynomials.

However, if the roots are *clustered* or have *multiplicities* greater than one (double roots, triple roots, ...) then the running time increases (more iterations are needed) and the accuracy can severely degrade (less correct digits are obtained). As *HP* says of ***Math Pac***'s **PROOT** keyword:

> *"The general rule-of-thumb for PROOT is that for multiple or nearly-multiple zeros, resolution of the root is approximately 12/K significant digits, where K is the multiplicity of the root."*

and the same applies to **PZER**, so double roots will be found with about 12/2 = *6* correct digits, triple roots with 12/3 = *4* correct digits, and so on. Consider for instance the polynomial $P(z) = z^3 + 3z^2 + 3z + 1$, which has a *triple* root ***−1***, so *P(−1)* = ***0***. But *P(−1.0001)* and all the values in between *also* evaluate to ***0*** when computed using the 12-digit *HP-71B* so there's no way to tell apart ***−1.0001*** from the correct root ***−1*** and the same happens with *P(−**0.99993**)*, which also evaluates to ***0***, etc. Therefore you can't consistently expect to get more than *~4* correct digits in this case because, as far as the program can tell, all those values qualify as roots. See ***Examples***.

2.4. Performance

These are some results for **PZER** runtimes, iterations used and tolerances achieved for the polynomial of degree ***N*** defined as $P(z) = \sum_{k=0}^{N} z^k = z^N + z^{N-1} + \cdots + z^2 + z + 1$, obtained by using the following code:

```
10 DESTROY ALL @ INPUT "N=";N @ T=0 @ L=0
20 OPTION BASE 0 @ COMPLEX P(N) @ OPTION BASE 1 @ COMPLEX R(N)
30 MAT P=CON @ SETTIME 0 @ CALL PZER(P,R,L,T) @ DISP "Iters:";L;TIME @ DISP " Tmin:";T
```

N	*Time*	*Iterations*	*Tolerance achieved*	*N*	*Time*	*Iterations*	*Tolerance achieved*
3	0.07”	6	3.76 E−12	**30**	12.86”	15	2.15 E−12
5	0.20”	7	1.44 E−12	**50**	63.28”	27	1.50 E−12
10	0.72”	7	3.00 E−12	**70**	136.93”	30	1.48 E−12
15	2.89”	13	1.40 E−12	**100**	466.02”	52	4.85 E−12
20	6.59”	17	1.34 E−12	–	–	–	–

***Note**: All runtimes in this article are for Jean-François Garnier's Emu71/DOS running on a 2.4 Ghz single-core CPU pc.*
*For real-coefficients polynomials, the assembly-language **PROOT** keyword is 10x-60x faster, depending on the degree.*

3. Subprogram Description and Calling Syntax

PZER **Roots of a Real or Complex Polynomial**

SUB PZER(P(),R(),L, T)

Where **P** is a real or complex vector with N+1 elements, where N is the degree of the polynomial whose roots are sought, **R** is a complex vector, and **L** and **T** are real scalar variables.

P can be declared as either **REAL**, **SHORT**, **INTEGER**, **COMPLEX** or **COMPLEX SHORT** precision.
R can be declared as either **COMPLEX** or **COMPLEX SHORT** precision.

R will be assigned the (complex) values of the solutions of the equation *P(z) = 0*, where **P** is the polynomial of degree N whose coefficients are the values of the elements of **P**. The values for **L** and **T** may be passed as either real variables holding the value or real numeric expressions, and:

If **L** is passed a nonzero value, that will be the maximum number of iterations. Else the default is 100. Additionally, if **L** is passed *by reference* the number of iterations used will be returned there.

If **T** is passed a nonzero value, that will be the desired tolerance. Else the default tolerance is 10^{-10}. Additionally, if **T** is passed *by reference* the tolerance actually achieved will be returned there.

4. Source Code Listing and Subprogram Characteristics

```
 100  SUB PZER(P(),R(),L,T) @ N=UBND(R,1) @ OPTION BASE 1 @ COMPLEX S(N),D,U,V,Z
 110  W=INF @ U=P(0) @ IF FLAG(0) THEN S$="4D,2X,K" ELSE S$="#2(^)"
 120  T=T+(T=0)/10^10 @ L=L+(L=0)*100 @ IF RNORM(R) THEN 140
 130  FOR I=1 TO N @ R(I)=(.4,.9)^(I-1) @ NEXT I
*140  FOR K=1 TO L @ M=0 @ FOR I=1 TO N @ Z=R(I) @ D=1
 150  FOR J=1 TO N @ IF I#J THEN D=D*(Z-R(J))
 160  NEXT J @ V=U @ FOR J=1 TO N @ V=V*Z+P(J) @ NEXT J
 170  V=V/((D+NOT ABS(D))*U) @ R(I)=Z-RES @ M=MAX(M,ABS(V)) @ NEXT I
 180  IF M<=T THEN T=M @ L=K @ END
 190  IF M<W THEN W=M @ H=K @ MAT S=R @ DISP USING S$;K,M
 200  NEXT K @ T=-W @ L=H @ MAT R=S
```

- This *BASIC* subprogram is 11 lines (407 bytes) long and uses several matrix- and complex-related keywords from the ***Math Pac*** so the *Math ROM* must be available (either physically plugged in or virtual *ROM* image.)
- It can be called either directly from the command line or from another program or subprogram.
- It accepts four parameters, namely:

  P( ) *input*, real or complex array declared with **OPTION BASE 0**, holds the N+1 polynomial's coefficients. Returns unaltered.

  R( ) *optional input*, complex array declared with **OPTION BASE 1**. If not empty, specifies the initial approximations for each root. If you don't want to specify them, just leave it *empty* (all elements (**0**, **0**), as just dimensioned) and they'll be automatically generated.

  output, holds the N roots computed and returned by **PZER**.

L *input*, specifies the max. number of iterations to use. If you don't want to specify the max.numbers of iterations, pass either some numeric expression which evaluates to **0** or a variable holding **0**, and the default maximum (up to 100 iterations) will be used.

optional output, if you pass a variable *by reference* the number of iterations actually used will be returned in it. If you don't want this optional output, pass a numeric value or numeric expression in the call, or a variable enclosed in parentheses (or simply disregard).

T *input*, specifies the min. tolerance[1] to achieve. If you don't want to specify a min. tolerance pass either some numeric expression which evaluates to **0** or a variable holding **0** and the default tolerance (10^{-10}) will be used.

optional output, if you pass a variable *by reference* the tolerance actually achieved will be returned in it, and will be marked *negative* if greater than the tolerance specified, to let the user know that it wasn't achieved. If you don't want this optional output, pass a numeric value or numeric expression in the call, or a variable enclosed in parentheses (or disregard).

5. Call Syntax Examples

```
CALL PZER(P,R,0,0)
```

Specifies the default min. tolerance (10^{-10}) and default max. iterations (up to 100), but doesn't return the tolerance actually achieved or the iterations used, just the roots (*aka "global mode" .)*

```
CALL PZER(P,R,50,0)
```

Specifies default min. tolerance (10^{-10}) and max. 50 iterations, but doesn't return the tolerance achieved or the iterations used.

```
CALL PZER(P,R,50,1E-5)
```

Specifies 10^{-5} as the min. tolerance and max. 50 iterations, but doesn't return the tolerance achieved or the iterations used.

```
L=0 @ T=0 @ CALL PZER(P,R,L,T)
```

Specifies the default min. tolerance (10^{-10}) and default max. iterations (up to 100), and also returns both the tolerance achieved and the iterations used.

```
L=50 @ T=0 @ CALL PZER(P,R,L,T)
```

Specifies the default min. tolerance (10^{-10}) and max. 50 iterations, and also returns both the tolerance achieved and the iterations used.

```
L=50 @ T=1E-5 @ CALL PZER(P,R,L,T)
```

Specifies 10^{-5} as the min. tolerance and max. 50 iterations, and also returns both the tolerance achieved and the iterations used.

You can also use one default but not the other and/or return the tol. achieved but not the iterations used, etc.

[1] Specifying a tolerance smaller (say 10^{-12}) than the default (10^{-10}) isn't recommended and might result in unnecessarily longer runtimes and/or reduced accuracy. Due to the quadratic convergence, the default tolerance will usually give full 12-digit accuracy (give or take a few *ulps)* for polynomials with single, adequately separated roots

6. Silent and Verbose modes

PZER can perform its computations in either a ***Silent*** mode (never displaying or printing anything, just like **PROOT**) or a ***Verbose*** mode in which relevant information is output on the fly while the process is running.

- to specify ***Silent*** mode, previously execute: **CFLAG 0** (usually the default state)
- to specify ***Verbose*** mode, previously execute: **SFLAG 0**

The chosen mode will remain in effect until you execute another **SFLAG** / **CFLAG 0** statement. When *Verbose* mode has been specified, each time an iteration achieves a *smaller* error than the previous ones *the iteration number* and the *smallest error* so far are output on the fly while the process continues. For example:

Find all roots of $P(x) = x^3 - 3x^2 + 3x - 5$ *and use* ***Verbose*** *mode to show the convergence:*

We execute the following directly from the command line:

```
>DESTROY ALL                          { clear all variables to prevent conflicts with former definitions }
>OPTION BASE 0 @ REAL P(3)            { declare the coefficients array }
>OPTION BASE 1 @ COMPLEX R(3)         { declare the roots array }
>MAT INPUT P                          { ask for the coefficients }
P(0)?
        1, -3, 3, -5                  { enter the coefficients }

>SFLAG 0 @ FIX 4                      { specify Verbose mode and 4 decimals for the output }
>CALL PZER(P,R,0,0)                   { call PZER passing just the coeffs.  and returning just the roots }

         1   2.0542                   { 1st iteration, error = 2.0542 }
         4   0.4867                   { 4th iteration, smaller error   }
         5   0.0472                   { 5th iteration, ditto }
         6   2.0598E-5                { 6th iteration, ditto }
```

and the very next iteration results in an error of $2.7122 \cdot 10^{-11}$ (notice the *quadratic* convergence), which is less than the default tolerance (10^{-10}), so the process ends and we proceed to display the computed roots:

```
>MAT DISP R

       (2.5874,  1.0000E-22)          { real root, disregard the negligible 10^-22 imaginary part }
       (0.2063,  1.3747)              { complex conjugate pair:   first root }
       (0.2063, -1.3747)              { complex conjugate pair:   second root }
```

Verbose mode can be very useful for diagnostic purposes or simply to check the convergence of the procedure, which eventually must be *quadratic*. If convergence seems to be slow (*linear),* this indicates that probably there are roots of multiplicity greater than one or closely clustered, and in that case the max. number of iterations specified should be increased though the improved roots might still not be accurate to full 10-12 digits.

If after the call to **PZER** you're not satisfied with the achieved tolerance (it returns *negative*) and would like to perform more iterations, the work already done isn't wasted as you can immediately repeat the call to **PZER** with the same parameters or different ones (e.g.: decrease the tolerance and/or increase the max. number of iterations) and, most important, with the **R** array still holding the just computed roots, *which will be used as initial approximations by the new call*, thus reducing the total computing time and hopefully improving the results.

Essentially this means that the new batch of iterations is starting just where the previous one ended, it doesn't redo those iterations again so no work wasted. This can be repeated as often as desired, see ***Example 8.3a/ 8.3d***.

7. A Simple Driver

As seen above, **PZER** can be called from the command line or from another program/subprogram, but you might find it convenient to use the following simple 221-byte *driver* program[1] which does all the initializations, asks you for the required inputs, validates them, calls **PZER** to compute the roots, and finally display them for you:

```
10 DESTROY ALL @ INPUT "N,Iter,Tol,Verb Y/N: ","3,0,0,N";N,L,T,V$
20 N=MAX(2,ABS(IP(N))) @ T=ABS(T) @ L=ABS(IP(L)) @ CFLAG 0 @ IF UPRC$(V$)="Y" THEN SFLAG 0
30 OPTION BASE 0 @ COMPLEX P(N) @ OPTION BASE 1 @ COMPLEX R(N)
40 MAT INPUT P @ CALL PZER(P,R,L,T) @ DISP "Iters:";L @ DISP " Tmin:";T
50 DISP "Roots: [press CONT]" @ PAUSE @ DELAY INF @ MAT DISP R @ DELAY 0,0
```

To use it, simply:

RUN → *N,Iter,Tol,Verb Y/N: 3,0,0,N*

It asks for the polynomial's degree *(N, default=3)*, the max. number of iterations *(Iter, default=100)*, the tolerance *(Tol, default=10^{-10})*, and whether you want *Verbose* mode *(Verb Y/N, default=N)*. If you're fine with the defaults, simply press **END LINE**, else edit the parameters you want (e.g.: the degree *N [change the **3**]* or activate *Verbose* mode *[change the **N** to **Y**]*) and press **END LINE**.

You'll be asked next for the coefficients of the polynomial. Enter them, beginning with the one for z^N (which can't be **0** or (**0**, **0**), lest the polynomial's degree wouldn't be *N*):

P(0)? a_n, a_{n-1}, a_{n-2}, ... , a_0 **END LINE**

The computation starts (relevant iteration info will be output on the fly if *Verbose* mode was specified) and upon termination the results will be displayed:

→ *Iters: iterations actually performed*
Tmin: minimum tolerance achieved *{ if > specified tol., it will be marked negative }*
Roots: [press CONT]

You *may* now specify the display mode you want (**STD**, **FIX n**, **SCI n**, etc.) and then press **CONT**:

CONT	→	*(real part$_1$, imag. part$_1$)*	*{ root #1 }*
SPC	→	*(real part$_2$, imag. part$_2$)*	*{ root #2 }*
		. . .	
SPC	→	*(real part$_N$, imag. part$_N$)*	*{ root #N }*
SPC	→	**>**	*{finished, show command line prompt }*

As you can see, execution *stops*[2] after displaying each root so that you can write it down at leisure, then press most any key (e.g.: **SPC** , as above) to display the next one. When all have been displayed, the command line prompt (**>**) appears (and the **DELAY INF** is automatically reset to **DELAY 0,0**).

If desired, you can redisplay or print all the roots by executing either **MAT DISP R** or **MAT PRINT R**.

---

[1] This convenient driver program is similar to (but much simpler than) the one for **PROOT** featured in the *HP-71 Math Pac Owner's Manual p. 121-124.*

[2] If execution stops because of a run-time error (such as *Division by Zero*), if desired you can *first examine* the subprogram's variables/parameters to look for the cause, *then end* the subprogram and return to the caller by executing from the command line either **END** (subprogram environment is lost and execution returns) or **END ALL** (the driver program is also terminated).

8. Examples

8.1 . Three Typical Examples:

(a) Use the driver (assume **STD** mode) to find all roots of: $P(z) = (1 + i)\, z^3 + (2 + i)\, z^2 + (3 + i)\, z + (4 + i)$

```
RUN   →  N,Iter,Tol,Verb Y/N: 3,0,0,N   END LINE     { all defaults suits us fine }
      →  P(0) ?  (1,1),(2,1),(3,1),(4,1) END LINE

      →  Iters:  6                      { 0.07" }
         Tmin:   5.08..E-12             { which is less than the default tolerance 10^-10, so it's Ok }
         Roots:  [press CONT]

CONT  →   ( -0.284985631787 , -1.3037864029   )
SPC   →   (  0.186345015088 ,  1.51551674976  )
SPC   →   ( -1.4013593833   ,  0.288269653138 ) ,   which are accurate to full 12 digits.
```

(b) Use the driver (**STD** mode) to find all roots of: $P(z) = z^3 - (9 + 12\, i)\, z^2 + (-21 + 64\, i)\, z + (85 - 20i)$

```
RUN   →  N,Iter,Tol,Verb Y/N: 3,0,0,N          END LINE  { all defaults suits us fine }
      →  P(0) ?   1,-(9,12),(-21,64),(85,-20)  END LINE

      →  Iters:  9                      { 0.11" }
         Tmin:   1.76..E-11             { which again it's less than the default tolerance: 10^-10 }
         Roots:  [press CONT]

CONT  →   ( 5             ,  6 )
SPC   →   ( 1.00000000001 ,  2 )
SPC   →   ( 3.00000000001 ,  4 ) ,   which are accurate to full 12 digits, give or take an ulp[1].
```

(c) My *HP-41C* program featured in *PPC Technical Notes V1N3 p6* is used there to find all roots of the complex polynomial: $P(z) = (2 + 8i)\, z^6 + 3\, z^5 + (-1 + 2i)\, z^4 + 2i\, z^3 - (3 + 3i)\, z^2 + (1 + 2i)\, z - 2 + 3i = 0$

Let's solve it with my current creation using the driver in *Verbose* mode to check convergence (assume **FIX 7**):

```
RUN   →  N,Iter,Tol,Verb Y/N: 3,0,0,N

6,0,0,Y   END LINE                  { specify degree 6 and Verbose, default tol. and max. iterations }

      →  P(0) ?  (2,8),3,(-1,2),(0,2),-(3,3),(1,2),(-2,3)   END LINE

      →  1   0.6257565              { iteration 1, err = 0.6257565 }
      →  4   0.3368252              { iteration 4, err = smaller }
      →  5   0.1249454              { iteration 5, ditto }
      →  6   0.0035500              { quadratic convergence sets in }
      →  7   0.0000011              { ditto }

      →  Iters:  8                  { 0.32" }
         Tmin:   1.91..E-12         { once more, this is less than the default tolerance: 10^-10 }
         Roots:  [press CONT]

CONT  →  ( -0.0715576,  1.1235559 ) ,   SPC  →  (  0.5688927,  0.5464170 )
SPC   →  ( -0.4721457, -0.3777269 ) ,   SPC  →  ( -0.9724260,  0.3032192 )
SPC   →  (  0.0323977, -0.8883400 ) ,   SPC  →  (  0.8266036, -0.3541840 )
```

which exactly match the results given in *p.9* of said issue. The other example there matches as well.

[1] An *ulp* is a *unit in the last place*. For instance, using 12 digits we have that *23.0000000003* differs from *23* by *3 ulps*.

8.2 . Two Examples comparing PZER to PROOT *(real coefficients)*:

(a) Let's try a **PROOT** usage example from the *HP Math Pac Owner's Manual p.124*, featuring the polynomial:

$$P(z) = z^6 + z^5 + z^4 + z^3 + z^2 + z + 1$$

From the command line, execute the following:

```
>DESTROY ALL
>OPTION BASE 0 @ REAL P(6)
>OPTION BASE 1 @ COMPLEX R(6)
>MAT INPUT P
P(0)?       1,1,1,1,1,1,1
```

First let's check that **PROOT** produces these roots:

```
>MAT R=PROOT(P) @ STD @ MAT DISP R

        ( -0.222520933956 ,  -0.974927912182 )
        ( -0.222520933956 ,   0.974927912182 )
        ( -0.900968867902 ,  -0.433883739118 )
        ( -0.900968867902 ,   0.433883739118 )
        (  0.623489801859 ,   0.781831482468 )
        (  0.623489801859 ,  -0.781831482468 )
```

Now let's see how **PZER** fares in its **PROOT**-like *"global"* fashion[1] (i.e.: all defaults, *Silent* mode, no returned info):

```
>MAT R=ZER @ CFLAG 0     { we first delete the roots PROOT produced and make sure Verbose mode is Off }
>CALL PZER(P,R,0,0)      { now we call PZER with all the defaults; the process takes 0.35" to complete }
>MAT DISP R

        (-0.900968867903 ,   0.433883739117 )
        ( 0.623489801859 ,   0.781831482468 )
        (-0.222520933956 ,   0.974927912183 )
        (-0.900968867902 ,  -0.433883739118 )
        (-0.222520933956 ,  -0.974927912182 )
        ( 0.623489801859 ,  -0.781831482468 )
```

which (apart from ordering) are the same roots that **PROOT** produced to *12-digit accuracy* save *1 ulp* here and there.

Also, the process was *"global"*: the user didn't have to supply initial approximations or a stopping criterion, and no information other than the roots was returned, so in this regard **PZER** behaved just as **PROOT** does.

(b) Now let's try something much more extreme, namely finding all 100 complex roots of the ***100th***-*degree* polynomial whose coefficients are all ***1***, as mentioned in the *HP Math Pac Owner's Manual p.129.*

In the same page, *HP* also goes on to say of **PROOT** 's performance:

> *"Of the 200 real and imaginary components of the calculated roots, about half were found to 10 digit accuracy. Of the rest, the error did not exceed a few counts in the 12th digit".*

[1] *"The* **PROOT** *function is global in the sense that the user is not required to supply either an initial guess or a stopping criterion"* (quoted from the *HP Math Pac Owner's Manual p.128.*)

Again, let's see how **PZER** does in its *"global"* mode (all defaults, *Silent* mode, no returned info):

```
>DESTROY ALL @ STD
>OPTION BASE 0 @ REAL P(100)
>OPTION BASE 1 @ COMPLEX R(100)
>MAT P=CON @ CALL PZER(P,R,0,0) @ MAT DISP R          { 466" }

        ( -0.987930439741 ,  -0.154898180214 )        { root #1 }
        (  0.435884418475 ,   0.900002652069 )        { root #2 }
        ( -0.712583964148 ,   0.701586839985 )        { root #3 }
              ...
        (  0.435884418475 ,  -0.900002652068 )        { root #98 }
        (  0.969198999200 ,   0.246278906832 )        { root #99 }
        ( -0.350126449192 ,   0.936702444524 )        { root #100 }
```

Now if we go on and compute those 100 roots using **PROOT** and compare the outputs ...

```
        ( -0.987930439740 ,  -0.154898180214 )        { root #37 }
        (  0.435884418474 ,   0.900002652068 )        { root #50 }
        ( -0.712583964147 ,   0.701586839985 )        { root #20 }
              ...
        (  0.435884418474 ,  -0.900002652068 )        { root #49 }
        (  0.969198999199 ,   0.246278906832 )        { root #16 }
        ( -0.350126449191 ,   0.936702444522 )        { root #54 }
```

... we find that **PZER** 's roots *#1, 2, 3, ... , 98, 99, 100* are **PROOT** 's roots *#37, 50, 20, ... , 49, 16, 54*, respectively, which are produced in a different order but are virtually the same to full 12-digit accuracy: among the 12 components above, as many as four are *identical*, seven differ by just *1 ulp* and only one differs by a mere *2 ulps*.

Considering **PROOT**'s higher numerical precision and extended range advantages, and that this is a ***100th-degree*** polynomial with 100 quite clustered complex roots, the accuracy achieved by **PZER** on its own is utterly *amazing*.

8.3 . Four Examples with Multiple roots:

(a) Use the driver (in **FIX 7**) to find the *triple* root of: $P(z) = z^3 + 3\,z^2 + 3\,z + 1$ as accurately as possible.

```
RUN   →  N,Iter,Tol,Verb Y/N: 3,0,0,N     END LINE     { all defaults suits us fine }
      →  P(0) ?  1,3,3,1                  END LINE

      →  Iters:   76                      { 1.16" }
          Tmin:  -0.0000090               { negative, so it didn't achieve the default tol., 10^-10 }
         Roots:  [press CONT]

CONT  →   ( -0.9999811 ,  0.0000120 )     {  P(z) = (   0    ,  1.12766 · 10^-14   )  }
SPC   →   ( -0.9999772 , -0.0000380 )     {  P(z) = ( -10^-11 , -4.36340 · 10^-15  )  }
SPC   →   ( -1.0000252 ,  0.0000051 )     {  P(z) = (  10^-12 ,  9.62306 · 10^-15  )  }
```

The default iterations got us about **5** correct digits (though as shown after each root *z*, *P(z)* is *essentially* ***(0,0)*** to *11-12* digits so the values found are pretty accurate roots as far as **PZER** can tell), but we know better and can attempt to improve them by using *50 additional* iterations. From the command line, execute:

```
>T=0 @ CALL PZER(P,R,50,T)        { the just computed roots are now used as the initial approximations }
```

which takes just *0.58"* , and we now display the new achieved tolerance and the improved roots:

```
>T
      -0.0000003        { new achieved tolerance, about 30 times better but still > 10^-10, thus negative }

>MAT DISP R

      ( -0.9999963,  0.0000003    )      { P(z) = (  0  , −1.2712 · 10^−15  )
      ( -0.9999939, -0.0000002    )      { P(z) = (  0  , −2.3789 · 10^−17  )
      ( -1.0000098,  2.2834334E-9 )      { P(z) = (  0  ,  6.6389 · 10^−19  )
```

which get us ~ **6** correct digits and *P(z)* is now ***(0, 0)*** to *15-17* digits or better. The 50 extra iterations were indeed worth it but we can't achieve significantly more accuracy using 12-digit arithmetic. Quoting Zhonggang Zeng:

> *"All standard softwares output similar inaccurate results, due to the "attainable accuracy" barrier they are subject to."*

(b) Use the driver (in **`FIX 7`**) to find all roots of:

$$P(z) = z^9 - (3 + 4i)\,z^8 + (3 + 16i)\,z^7 + (11 - 44i)\,z^6 + (-61 + 80i)\,z^5 + (159 - 92i)\,z^4 + (-267 + 32i)\,z^3 + (277 + 92i)\,z^2 - (156 + 128i) + (36+48i)$$

```
RUN    →  N,Iter,Tol,Verb Y/N: 3,0,0,N

9,0,0,Y    END LINE    →          { specify deg. 9 and Verbose, default tolerance, max. iterations }

P(0)?  1,-(3,4),(3,16),(11,-44),(-61,80),(159,-92),(-267,32),(277,92),-(156,128),(36,48)

           END LINE    →          { now Verbose mode allows us to check convergence }

       →  1    11.0805302
       →  2     7.2399942
                  ...
       →  14    0.0001299
       →  19    0.0000656        { and indeed we observe very slow convergence, quite possibly
       →  89    0.0000236        indicating several multiple roots }

       →  Iters:  89             { 8.40" }
          Tmin:  -0.0000236      { negative, so it didn't achieve the default tolerance, 10^-10 }
          Roots:  [press CONT]

CONT   →  ( 1.0000000      ,  0.0000000 )   { alleged triple root :   1          }
SPC    →  ( 1.0000000      , -1.4142136 )   { conjugate pair #1  :   1 − √2  i }
SPC    →  (- 1.0000017     ,  2.0000022 )   { alleged double root:  −1 + 2i     }
SPC    →  ( 1.0000000      ,  1.4142136 )   { conjugate pair #1  :   1 + √2  i }
SPC    →  ( 1.6022543E-13  , -2.0000000 )   { conjugate pair #2  :  −2i         }
SPC    →  ( 1.0000115      , -0.0001513 )   { alleged triple root :   1          }
SPC    →  ( 1.0000150      ,  0.0001694 )   { alleged triple root :   1          }
SPC    →  ( -1.0470053E-12 ,  2.0000000 )   { conjugate pair #2  :   2i         }
SPC    →  ( -1.0000001     ,  1.9999986 )   { alleged double root:  −1 + 2i     }
```

So we have:
- the *single conjugate roots* have full *12* correct digits
- the alleged *double root* has about *6-7* correct digits
- the alleged *triple root* has about *4-5* correct digits (but the first one is *12*-digit *exact*)

This is about as good as it gets using 12-digit arithmetic so there's no point in performing extra iterations.

(c) Use the driver (in **FIX 5**) to find all roots of the 11th degree polynomial:

$$P(x) = x^{11} - 44\,x^{10} + 852\,x^{9} - 9{,}576\,x^{8} + 69{,}306\,x^{7} - 338{,}376\,x^{6} + 1{,}133{,}768\,x^{5} - 2{,}596{,}984\,x^{4} + 3{,}966{,}573\,x^{3} - 3{,}826{,}620\,x^{2} + 2{,}087{,}100\,x - 486{,}000$$

RUN → *N,Iter,Tol,Verb Y/N: 3,0,0,N*

11,0,0,N END LINE → *{ specify degree 11 and all the defaults }*

P(0)? 1,-44,852,-9576,69306,-338376,1133768,-2596984,3966573,-3826620,2087100,-486000

END LINE → *Iters: 75* *{ 12.24" }*
Tmin: -1.02946 E-2
Roots: [Press CONT]

It took many iterations (75) and the tolerance achieved is only $1.02 \cdot 10^{-2}$, much worse than the default 10^{-10} specified, so we suspect *several* multiple roots and thus we won't display the just computed roots but instead we'll immediately repeat the process specifying more iterations *(128)* and watching convergence:

>DELAY 0,0 END LINE RUN → *N,Iter,Tol,Verb Y/N: 3,0,0,N*

11,128,0,Y END LINE → *{ we also specify Verbose mode (**Y**) to check convergence }*

P(0)? *{ there's no need to key in the many lengthy coefficients again, just use the command stack to recall them all to the prompt and press* END LINE *}*

→ ...
→ *26 0.01315*
→ *27 0.01251*
→ *75 0.01029* *{ as expected, we observe very slow convergence,*
→ *113 0.00765* *probably due to more than one multiple roots }*
→ *Iters: 113* *{ 15.68" }*
→ *Tmin: -0.00765*
→ *Roots: [Press CONT]*

CONT	→ (1.00000, 0.00000)	{ double : **1**, 12 digits;	PROOT: ~ same }
SPC	→ (5.00686, 0.01448)	{ triple : **5**, 2-3 digits;	PROOT: (4.95044, 0.08799) }
SPC	→ (4.98292, -0.00013)	{ triple : **5**, 3-4 digits;	PROOT: (5.10033, 0.00000) }
SPC	→ (3.00007, -0.00083)	{ double : **3**, 3-4 digits;	PROOT: ~ same (complex) }
SPC	→ (9.00000, 1.16255E-12)	{ single : **9**, 9 digits;	PROOT: ~ same }
SPC	→ (2.00000, 6.49191E-89)	{ single : **2**, 10 digits;	PROOT: ~ same }
SPC	→ (3.00004, 0.00011)	{ double. : **3**, 4-5 digits;	PROOT: ~ same (complex) }
SPC	→ (6.00000, -3.57906E-8)	{ single : **6**, 7 digits;	PROOT: (5.99954, 0.00000) }
SPC	→ (5.00694, -0.01235)	{ triple : **5**, 2-3 digits;	PROOT: (4.95044,−0.08799) }
SPC	→ (4.00001, -3.39592E-9)	{ single : **4**, 6 digits;	PROOT: (3.99918, 0.00000) }
SPC	→ (1.00001, -0.00002)	{ double : **1**, 5-6 digits;	PROOT: ~ same }

The roots are: ***3***, ***1***, ***4***, ***1***, ***5***, ***9***, ***2***, ***6***, ***5***, ***3***, ***5***, so ***2***,***4***,***6***,***9*** are *single* roots, ***1***,***3*** are *double* roots and ***5*** is a *triple* root.

That **PZER** manages to achieve about 140 correct digits out of 264 while dealing with that many multiple roots is a truly excellent result for an implementation mostly suited to isolated, single roots. Actually, it beats **PROOT** at its own game by obtaining more accurate roots in *five* cases. Furthermore, **PROOT** also reports several *real* roots as being *complex*, despite what it's said in the *"Hewlett-Packard Journal" July 1984 p.34*, namely:

> *"Also, <u>when a complex root is found, there is 100% confidence that it is actually complex</u> (and not a real root contaminated by round-off) so that both it and its complex conjugate can be declared as roots."*

(d) Use the driver (in **FIX 7**) to find all the roots of this polynomial, taken from "*On 4th order simultaneously zero-finding method for multiple roots of complex polynomial equations":*

$$P(z) = z^7 - 3z^6 + 5z^5 - 7z^4 + 7z^3 - 5z^2 + 3z - 1$$

```
RUN   →  N,Iter,Tol,Verb Y/N: 3,0,0,N

7,0,0,Y     END LINE    →          { we specify Verbose mode (Y) to check convergence }

P(0)?       1,-3,5,-7,7,-5,3,-1    END LINE

   →          ...
        13  0.0000812
        22  0.0000604
        31  0.0000072
        32  0.0000050              { slow convergence, thus probably multiple roots }

        Iters:   32                { 5.22" }
         Tmin:  -0.0000050         { negative, so it didn't achieve the default tolerance, 10^-10 }
        Roots:   [Press CONT]

CONT  →  ( 1.0000000 ,  0.0000000 )
SPC   →  ( -0.0000003 ,  1.0000002 )
SPC   →  ( -0.0000011 ,  0.9999997 )
SPC   →  ( 0.0000007 , -0.9999998 )
SPC   →  ( -0.0000003 , -0.9999997 )
SPC   →  ( 0.9999877 , -0.0000010 )
SPC   →  ( 1.0000060 , -0.0000102 )
```

The actual integer roots are perfectly recognizable and have about ***5-7*** correct digits (though the first one has full *12* correct digits) but we can attempt to get some extra precision just because, by performing an exotic ***222*** additional iterations, like this:

```
>CALL PZER(P,R,222,0) @ MAT DISP R

     1   0.00019324     { notice the small first error because we're automatically using the just computed
                          roots as initial guesses for this new batch of iterations, which speeds it all up }
          ...
   203   0.00000006     { 11.57" }

 ( 1.0000000,  0.0000000 )  { triple root:     1 ;  PROOT:  ( 0.9999399,    0.0000000 )
 ( 0.0000003,  1.0000009 )  { double root:     i ;  PROOT:  ~ same
 ( -0.0000002,  0.9999998 )  { double root:    i ;  PROOT:  ~ same
 ( 0.0000001, -1.0000008 )  { double root:    −i ;  PROOT:  ~ same
 ( 0.0000004, -0.9999997 )  { double root:    −i ;  PROOT:  ~ same
 ( 0.9999976,   2.06..E-10 )  { triple root:    1 ;  PROOT:  (1.0000300,    0.0000520 ), complex
 ( 1.0000024,   8.90..E-14 )  { triple root:    1 ;  PROOT:  (1.0000300, −0.0000520 ), complex
```

which got us *8 times better* achieved tolerance and significantly more accurate roots, despite their various multiplicities. Matter of fact, there wasn't even a single *single* root in sight.

Notice also that **PROOT** did significantly *worse*, despite its extended accuracy/range: it didn't get the first instance of the triple root (or any other for that matter) accurate to full 12 digits (just ~ *4*), and the second and third instances were found to ~ *5* digits while **PZER** got *6-7* digits. Worst of all, it reports them as a *complex* conjugate pair instead of *real*, despite what *HP* stated (see the ***Example*** in the previous page) . Sorry but *no*.

Appendix A

For checking purposes, these are all the roots of the 37th-degree polynomial $P(z) = z^{37} + z^{36} + ... + z^2 + z + 1$

To obtain them, we execute the following directly from the command line:

```
>DESTROY ALL @ STD                { clear all variables to avoid potential conflicts and set STD display mode }
>OPTION BASE 0 @ REAL P(37)       { dimension the coefficients array, which for this polynomial it's real }
>OPTION BASE 1 @ COMPLEX R(37)    { dimension the roots array }
>MAT P=CON                        { all coefficients are 1 }
>CFLAG 0                          { ensure no Verbose mode }
>L=0 @ T=0                        { specify default max. iterations (100) and default tolerance (10^-10)
>CALL PZER(P,R,L,T)               { call PZER passing the coefficients and returning the roots as well as
                                    the iterations used to compute all 37 roots and the tolerance achieved }
    ...                           { the computation proceeds for 21.32" }
>L;T                              { display the number of iterations used and the tolerance achieved }

        17   1.15690811496E-12    { just 17 iterations achieved a tolerance of 1.1569·10^-12 < 10^-10 }

>DELAY INF @ MAT DISP R           { display each root in turn; press any key to display the next one and
                                    don't forget to afterwards reset the DELAY to your preferred setting }
```

```
( -0.9458172417   ,   0.324699469204 )      {  conjugate pair   #01 }
(  0.401695424653 ,   0.915773326655 )      {  conjugate pair   #02 }
( -0.789140509396 ,   0.614212712689 )      {  conjugate pair   #03 }
( -0.9458172417   ,  -0.324699469204 )      {  conjugate pair   #01 }
( -0.245485487141 ,  -0.969400265939 )      {  conjugate pair   #04 }
(  0.945817241701 ,  -0.324699469205 )      {  conjugate pair   #05 }
(  0.546948158123 ,   0.837166478262 )      {  conjugate pair   #06 }
( -0.677281571625 ,   0.735723910672 )      {  conjugate pair   #07 }
( -0.789140509397 ,  -0.61421271269  )      {  conjugate pair   #03 }
(  0.245485487141 ,  -0.96940026594  )      {  conjugate pair   #08 }
(  0.0825793454728 , -0.996584493007 )      {  conjugate pair   #09 }
(  0.789140509396 ,  -0.614212712689 )      {  conjugate pair   #10 }
(  0.677281571626 ,   0.735723910673 )      {  conjugate pair   #11 }
( -0.245485487141 ,   0.96940026594  )      {  conjugate pair   #04 }
( -1              ,   0              )      {  real root            }
(  0.789140509396 ,   0.614212712689 )      {  conjugate pair   #10 }
(  0.401695424653 ,  -0.915773326655 )      {  conjugate pair   #02 }
(  0.0825793454728 ,  0.996584493007 )      {  conjugate pair   #09 }
( -0.401695424653 ,   0.915773326655 )      {  conjugate pair   #12 }
( -0.986361303403 ,  -0.164594590281 )      {  conjugate pair   #13 }
( -0.401695424653 ,  -0.915773326655 )      {  conjugate pair   #12 }
(  0.546948158122 ,  -0.837166478262 )      {  conjugate pair   #06 }
(  0.879473751206 ,   0.475947393037 )      {  conjugate pair   #14 }
( -0.0825793454727,   0.996584493007 )      {  conjugate pair   #15 }
( -0.879473751206 ,   0.475947393037 )      {  conjugate pair   #16 }
( -0.879473751206 ,  -0.475947393037 )      {  conjugate pair   #16 }
( -0.0825793454722,  -0.996584493006 )      {  conjugate pair   #15 }
(  0.879473751206 ,  -0.475947393037 )      {  conjugate pair   #14 }
(  0.245485487141 ,   0.96940026594  )      {  conjugate pair   #08 }
( -0.546948158122 ,   0.837166478263 )      {  conjugate pair   #17 }
(  0.677281571626 ,  -0.735723910673 )      {  conjugate pair   #11 }
( -0.546948158123 ,  -0.837166478263 )      {  conjugate pair   #17 }
(  0.986361303403 ,   0.164594590281 )      {  conjugate pair   #18 }
(  0.945817241701 ,   0.324699469205 )      {  conjugate pair   #05 }
( -0.986361303403 ,   0.164594590281 )      {  conjugate pair   #13 }
(  0.986361303403 ,  -0.164594590281 )      {  conjugate pair   #18 }
( -0.677281571626 ,  -0.735723910673 )      {  conjugate pair   #07 }
```

Appendix B

PZER uses a simple procedure to generate the initial guesses for the roots, which basically consist of spreading the guesses uniformly along a slowly decreasing spiral within the unit disk, but much more elaborate procedures are possible, such as those described in the *"Hewlett-Packard Journal July 1984"* issue *pp 33-36*, which among many other finesses strive to find annuli known to contain the smallest root of the current[1] polynomial.

On the other hand, **PZER** doesn't compute the roots one by one but *all simultaneously* so the annulus would have to contain all roots and then the initial guesses would be assigned by, say, generating random values uniformly distributed within said annulus. There are many methods to compute the bounds of all roots (i.e.: the annulus radii, ***R1*** and ***R2***) from very simple ones (but rather unsharp) to very complicated ones (but sharper).

I did try a version of the well-known *Cauchy criterion*, where ***R1*** and ***R2*** are the unique positive roots of:

$$|a_n|\, z^n + |a_{n-1}|\, z^{n-1} + \cdots + |a_1|\, z - |a_0| = 0 \qquad (R1)$$

and

$$|a_n|\, z^n - |a_{n-1}|\, z^{n-1} - \cdots - |a_1|\, z - |a_0| = 0 \qquad (R2)$$

respectively. They can be computed relatively quickly and accurately using **FNROOT** but I found that in practice the upper bound ***R2*** tends to be overly *large* and nothing is gained in terms of significantly less iterations (thus shorter running times) or improved accuracy to compensate for the added complexity, and matter of fact often enough the process ran *slower*, not faster. For instance, let's consider the polynomial from ***Example 1b***:

$$P(z) = z^3 + (-9 - 12\,i)\, z^2 + (-21 + 64\,i)\, z + (85 - 20i)$$

whose roots are *(1,2), (3,4)* and *(5,6)*, which required 9 iterations with the current, simple implementation. Using *Cauchy criterion* we find:

R1 is the root of: $x^3 + |(-9,-12)|\, x^2 + |(-21,64)|\, x - |(85,-20)| = 0 \;\rightarrow\; \mathbf{R1} = 1.03922225823$
R2 is the root of: $x^3 - |(-9,-12)|\, x^2 - |(-21,64)|\, x - |(85,-20)| = 0 \;\rightarrow\; \mathbf{R2} = 18.8245724264$

and with ***R1***, ***R2*** in hand we would randomly assign the initial guess for each root within the annulus, like this:

```
RADIANS @ FOR I=1 TO N @ R(I)=RECT((RND*(R2-R1)+R1,2*PI*RND)) @ NEXT I
```

However, once the solving procedure was executed with these new initial guesses I found that it *still* required 9 iterations and the tolerance achieved ($3.54\cdot10^{-11}$) was *2x worse* than the one obtained using the simple guesses ($1.77\cdot10^{-11}$) and the computed roots were ever- so-slightly *less* accurate.

I conducted many such tests and tried various other criterions and bounds and ultimately decided that it just wasn't worth the complications and little was gained at all, quite the contrary, so the simpler procedure stayed.

Final Conclusions

Subprogram **PZER** is a very short piece of code which fills up a glaring **PROOT** limitation, as it can quickly and accurately find *all* roots of an N^{th} degree polynomial with *complex* coefficients, which **PROOT** just can't touch.

Further, it can work as a *"global"* procedure which requires no inputs from the user save the coefficients and returns nothing but the roots, as **PROOT** does, but additionally it also gives the user a lot of optional extra control, allowing the specification of desired tolerance, max. number of iterations, distinct initial guesses for each of the roots and the possibility of watching convergence to detect problems, as well as optionally returning the tolerance actually achieved (which will be marked *negative* if it didn't meet the one specified) and the iterations used to achieve it. Last but not least, it frequently deals better with *multiple* roots than **PROOT** does.

All in all, a worthy addition to the *HP-71B* math capabilities, which can be called directly from the command line or from your own programs/subprograms, as well as using the convenient driver program also featured herein.

[1] **PROOT** finds the roots one by one, not all at once like **PZER** does, beginning with the smallest one in magnitude and once found using *deflation* to reduce the polynomial's degree by one or two, thus the *current* (deflated) polynomial varies.