Roots of equations

1. Transcendental equations

A transcendental equation has the form

$$f(x) = 0,$$

where $f(x)$ is a transcendental function rather than a polynomial or rational function (ratio of polynomials). The latter can be solved by special methods that will be discussed in the following Section.

A typical context in which it becomes necessary to solve one or several transcendental equations is control theory. We often design governors with negative feedback: that is, we take part of the output, reverse its sign and apply it to the input of a system. This can have a stabilizing effect if all goes as we planned. The equation for such a system might be

$$\dot{x}(t) = -a\, x(t)\,,$$

where $x(t)$ represents some deviation from normal operation. This equation is stable since its solution represents damped perturbations:

$$x(t) = x(0)\, e^{-a\, t}\,.$$

However, in practice control systems include time delays resulting from signal propagation, hence can develop instabilities. That is, the preceding equation is actually

$$\dot{x}(t) = -a\, x(t - \tau)$$

where τ represents the delay. The standard substitution $x(t) = x(0)\, e^{\lambda\, t}$ yields a transcendental equation

$$\lambda\tau\, e^{\lambda\, \tau} = -a\tau\,.$$

While it is obvious that no real and positive values of λ can satisfy this equation, it is possible that for some range of $a\tau$ there are complex roots with positive real parts. In that case the control system could support oscillations that increase exponentially in magnitude—it would be wildly unstable!

There are several standard methods for finding a (or possibly, *the*) root of a transcendental equation (which might have many roots or none). The simplest of these is *Newton's method*. Under certain conditions the methods of *binary search* and *regula falsi* are more stable and therefore to be preferred. We discuss them in the order named.

Newton's method

Suppose $f(x)$ is a differentiable function; then if x_n is near a root we have

$$f(x_n) \approx f(x_{n-1}) + (x_n - x_{n-1}) f'(x_{n-1}) + O\left((x_n - x_{n-1})^2\right).$$

Setting $f(x_n) = 0$ and solving for x_n we have

$$x_n \approx x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})},$$

That is, we iterate until the sequence $\{x_n\}$ converges (or diverges).

Consider the problem of finding the square root of a positive number. In school we were taught synthetic division. But Newton's method offers a rapidly converging (and quite stable) iteration:

$$x^2 - a = 0$$

$$x_{n+1} = x_n + \frac{a - x_n^2}{2x_n} \equiv \frac{a/x_n + x_n}{2}.$$

Given a good initial guess the iteration converges extremely rapidly, as shown below for $\sqrt{2}$.

$$a = 2, \quad x_0 = a/2$$

n	Result
0	1.0000000000
1	1.5000000000
2	1.4166666667
3	1.4142156863
4	1.4142135624
5	1.4142135624

The computation of square roots is built into virtually every numeric co-procesor and software floating-point arithmetic package in modern use. Cube roots, however, are rarely included. The following program uses Newton's method to find cube roots.

```
\ Cube root of real number by Newton's method
\  ANS compatible version
\     (c) Copyright 1994  Julian V. Noble.     Permission is granted
\     by the author to use this software for any application provided
\     the copyright notice is preserved.
\
\  Algorithm:
\      x' = (N/x^2 + 2x)/3
\
\ This code conforms with ANS requiring:
\      FLOAT and FLOAT EXT word sets
\ Environmental dependence:
\      assumes separate floating point stack
\
\ Non STANDARD words: FTUCK
```

```
: undefined    BL WORD  FIND  NIP  0=  ;
undefined FTUCK  [IF]  : FTUCK   FSWAP  FOVER  ;  [THEN]

: x'       ( f: N x - - x')
   FTUCK   FDUP F*   F/  FSWAP  F2*  F+  3e0  F/  ;

: converged?    ( f: x' x x' - - x') ( - - f)
   F-   FOVER   F/   FABS   1.E-8  F<   ;

: fcbrt    ( f: N — N^1/3)
   FDUP  F0<  FABS               ( f: - - |N|)  ( - - f)
   FDUP  FSQRT                   ( f: - - N x0 )
   BEGIN    FOVER FOVER  x'   FTUCK   converged?
   UNTIL
   x'   IF   FNEGATE   THEN  ;
```

Below we see the tabulated results of this algorithm:

$$a = 2, \quad x_0 = \sqrt{a}$$

n	Result
0	1.4142135624
1	1.2761423749
2	1.2601263691
3	1.2599210833
4	1.2599210499
5	1.2599210499

As our final example, consider how to compute the inverse of a number on a machine lacking a division instruction (as did many early computers). We might think of trying to find the root of

$$ax - 1 = 0$$

by Newton's method, *i.e.*

$$x_{n+1} = x_n - \frac{ax_n - 1}{a}$$

but this is a mere tautology. The proper approach is to apply Newton's method to

$$a - \frac{1}{x} = 0$$

giving

$$x_{n+1} = \left(2 - ax_n\right)x_n .$$

This iteration is unstable if, for $a > 1$ we choose $x_0 > 1$ or *vice-versa*. But it converges quite rapidly if a proper starting value is chosen. If $a > 1$ a good choice is to find the leading bit, *i.e.* the leading power of 2 contained in a, and take $x_0 = 2^{-k-1}$. The results for inverting 3.0 are shown below:

$a = 3, \quad x_0 = 0.25$

n	Result
0	0.2500000000000
1	0.3125000000000
2	0.3320312500000
3	0.3333282470703
4	0.3333333332557
5	0.3333333333333

When the stability of Newton's method is not known in advance (or is suspect), it is useful to choose a method that is guaranteed to find a root because we know that a root lies in a definite interval of the x-axis. Let us look first at binomial search, since its algorithm is easy to understand.

Binary search

We know that some interval, $x_L \le x \le x_R$, contains a root because $f(x)$ changes sign when x goes from $x_L \to x_R$. The method begins with upper and lower bounds on x that capture the root. Next we look at $f(\overline{x})$ where $\overline{x}$ lies halfway between x_L and x_R. If $f_{new} = f(\overline{x})$ has the same sign as $f_L = f(x_L)$, the new left end of the interval becomes $\overline{x}$. If the signs are opposite, $\overline{x}$ becomes the new right end of the interval. The algorithm is done when left and right ends agree within some predetermined accuracy. In pseudocode[1] the binomial search algorithm is

Binary search has the following virtues:

```
DECLARE FUNCTION dummy! (x!)
DECLARE FUNCTION binsrch! (a!, b!, eps!)
PRINT binsrch(0!, 1!, .00001)
END

FUNCTION binsrch (a, b, eps)
    fa = dummy(a)
    fb = dummy(b)
    DO UNTIL ABS(b - a) < eps
        xp = (b + a) / 2
        fx = dummy(xp)
        IF fa * fx  0 THEN
            fb = fx
            b = xp
        ELSE
            fa = fx
            a = xp
        END IF
    LOOP
```

1. Actually, in Microsoft QuickBasic®.

```
    binsrch = (b + a) / 2
END FUNCTION

FUNCTION dummy (x)
    dummy = x - EXP(-x)
END FUNCTION
```

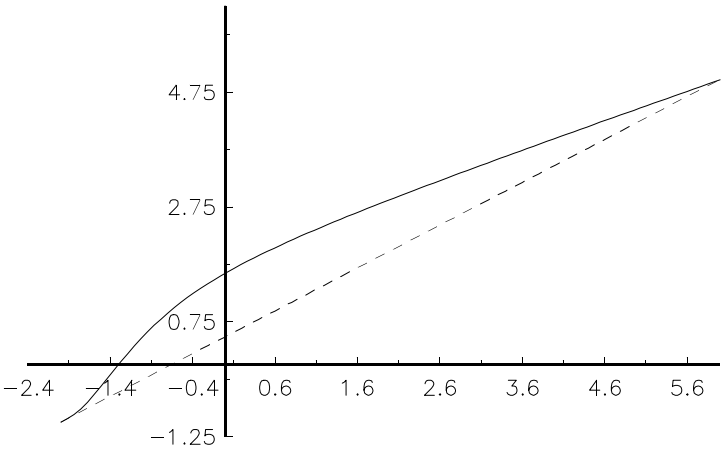
- the time it takes to achieve a given accuracy is predictable;
- it is guaranteed to find a captured root.

Regula falsi

Now we look at *regula falsi*, Latin for "rule of false approach". Here the basic premise is:

- Assume the root lies in the interval $(x_L\ ,\ x_R)$, and plot a straight line between the points $(x_L\ ,\ f_L)$ and $(x_R\ ,\ f_R)$.
- This line must intersect the x-axis somewhere in the interval, and we take that point, call it x', as our next guess.
- If x' is to the left of the root, adjust the interval accordingly, and the same if x' is to the right of the root.

As the figure to the right shows, the straight line is supposed to approximate the curve $f(x)$. The new guess may be much closer to the root than is the midpoint of the interval (which was the next guess in binomial search).



A straight line in the x-y plane has the analytic form

$$y = ax + b$$

where a and b are constants. The intercept of the straight line with the x-axis is gotten by setting $y=0$ and solving for x:

$$x' = -\frac{b}{a}.$$

To determine a and b we use the two equations

$$f_L = ax_L + b$$

$$f_R = ax_R + b$$

giving

$$a = \frac{1}{2}\left[f_L + f_R - \frac{f_R - f_L}{x_R - x_L}\left(x_L + x_R\right)\right]$$

and thus

$$x' = \frac{f_R\, x_L - f_L\, x_R}{f_R - f_L}\,.$$

Hybrid method

Sometimes regula falsi can be very slow—this happens if the function has a "knee" where the root is located, and is flat on either side of the knee. To speed things up when this is the case it is useful to insert a binary search step between the regula falsi steps, thereby giving rise to a hybrid approach. A simple Forth program that implements this strategy is given at the end of the chapter.

2. Roots of polynomials

For a certain type of function, namely a polynomial with real or complex coefficients, specialized methods have been devised that can find all the roots. The need to find the roots of polynomials arises in several contexts. For example, suppose the solution to a problem we are interested in may be expressed as an ordinary differential equation with constant coefficients,

$$a_n \frac{d^n x}{dt^n} + a_{n-1} \frac{d^{n-1} x}{dt^{n-1}} + \ldots + a_0\, x = f(t)\,;$$

then as is well known, when $f(t) = \alpha\, e^{\lambda t}$, where λ is a root of the equation

$$a_n\, \lambda^n + a_{n-1}\, \lambda^{n-1} + \ldots + a_0 = 0\,,$$

the solution $x(t)$ becomes infinite. Such equations arise when we treat structures as lumped masses connected by springs. Designers of structures subject to external vibrations must be cognizant of the resonant frequencies of their designs so that destructive loads do not develop.

The *fundamental theorem of algebra* asserts that a polynomial of degree n whose coefficients $a_0\,, a_1\,, \ldots\,, a_n$ are complex numbers has exactly n (complex) roots. The computation of these roots can be quite tedious, particularly if two roots are close. Many authors have devised algorithms for finding the roots of polynomials[2]. Here we explain the Laguerre algorithm, which is based on the following idea: write the polynomial as a function of the complex variable z in factored form.

---

2. See, *e.g.*, F.S. Acton, *Numerical Methods that Work* (Math. Ass'n of America, Washington, DC, 1990) for references to the literature.

$$p_n(z) \overset{df}{=} a_0 + a_1 z + a_2 z^2 + \ldots + a_n z^n \equiv a_n (z - z_1)\ (z - z_2) \ldots (z - z_n) .$$

Then

$$\log \left| p_n(z) \right| = \log \left| a_n \right| + \log|z - z_1| + \log|z - z_2| + \ldots + \log|z - z_n|$$

so that

$$\frac{d \log |p_n(z)|}{dz} = \frac{1}{z - z_1} + \frac{1}{z - z_2} + \ldots + \frac{1}{z - z_n} = \frac{p'_n(z)}{p_n(z)} \overset{df}{=} G ,$$

and

$$- \frac{d^2 \log |p_n(z)|}{dz^2} = \frac{1}{(z - z_1)^2} + \frac{1}{(z - z_2)^2} + \ldots + \frac{1}{(z - z_n)^2} = \left(\frac{p'_n(z)}{p_n(z)} \right)^2 - \frac{p''_n(z)}{p_n(z)} \overset{df}{=} H .$$

Now suppose we make the drastic assumption that for a given guess, z,

$$z - z_1 = a$$
$$z - z_k = b , \quad k = 2, \ldots , n .$$

In that case we may write

$$G = \frac{1}{a} + \frac{n-1}{b}$$

$$H = \frac{1}{a^2} + \frac{n-1}{b^2}$$

and upon eliminating b between the two equations, find

$$a = \frac{n}{G \pm \sqrt{(nH - G^2)(n-1)}} \equiv \frac{n\, p_n(z)}{p_n'(z) \pm p_n(z) \sqrt{(nH - G^2)(n-1)}} .$$

A **FORTRAN** routine that implements this is[3]

```
      SUBROUTINE LAGUER(A,M,X,EPS,POLISH)
      COMPLEX A(*),X,DX,X1,B,D,F,G,H,SQ,GP,GM,G2,ZERO
      LOGICAL POLISH
      PARAMETER (ZERO=(0.,0.),EPSS=6.E-8,MAXIT=100)
      DXOLD=CABS(X)
      DO 12 ITER=1,MAXIT
        B=A(M+1)
        ERR=CABS(B)
        D=ZERO
        F=ZERO
        ABX=CABS(X)
        DO 11 J=M,1,-1
```

3. W.H. Press, *et al.*, *Numerical Recipes* (Cambridge U. Press, Cambridge, 1986), p. 263ff.

```
          F=X*F+D
          D=X*D+B
          B=X*B+A(J)
          ERR=CABS(B)+ABX*ERR
11      CONTINUE
        ERR=EPSS*ERR
        IF(CABS(B).LE.ERR) THEN
          DX=ZERO
          RETURN
        ELSE
          G=D/B
          G2=G*G
          H=G2-2.*F/B
          SQ=CSQRT((M-1)*(M*H-G2))
          GP=G+SQ
          GM=G-SQ
          IF(CABS(GP).LT.CABS(GM)) GP=GM
          DX=M/GP
        ENDIF
        X1=X-DX
        IF(X.EQ.X1)RETURN
        X=X1
        CDX=CABS(DX)
        IF(ITER.GT.6.AND.CDX.GE.DXOLD)RETURN
        DXOLD=CDX
        IF(.NOT.POLISH)THEN
          IF(CABS(DX).LE.EPS*CABS(X))RETURN
        ENDIF
12    CONTINUE
      PAUSE 'too many iterations'
      RETURN
      END
```

It is worth noting that this subroutine is rather long because it includes an optional iterative procedure for polishing the root, controlled by the *flag* (*i.e.* the logical variable) `POLISH`. This is typical of languages like `FORTRAN` or `C` which exact a large run-time penalty for breaking procedures up into small, manageable modules.

Once we have found an estimate of a root, we can use Newton's method to polish it. To accomplish this expeditiously requires fast routines for evaluating a polynomial and its first derivative. We shall discuss these below.

Suppose we have found a root[4] z_1 by Laguerre's method—what next? Clearly the next step is to *deflate* the polynomial by dividing out the factor $(z - z_1)$, thereby producing a polynomial of degree $n - 1$. The process of deflation can be carried out by *synthetic division*, that is, by writing

$$p_n(z) \equiv (z - z_0)\, q_{n-1}(z) + p_n(z_0) .$$

4. …or a complex conjugate pair of roots if the coefficients a_k are real.

The process of synthetic division[5] yields both the quotient polynomial $q_{n-1}(z)$ and the remainder,

$$R \overset{df}{=} p_n(z_0) .$$

It is easiest to illustrate with an example. Consider $3x^5 - 14x^3 + x^2 - 5x + 7$. To divide it by $(x - 5)$ we arrange its coefficients in a tableau as shown below:

	3	0	−14	1	−5	7
5	0	15	75	305	1530	7625
	3	**15**	**61**	**306**	**1525**	**7632**

To the left we place the value 5, and below the leading coefficient (that is, a_5) we put 0. Now we add the two entries in the first column to get **3**. Next multiply by 5 and place the result in the second column, and add these two entries to get **15**. Proceeding rightward as indicated by the arrows, we get the bottom row. The first five numbers are the coefficients of the quotient polynomial, that is,

$$q_{n-1}(x) = 3x^4 + 15x^3 + 61x^2 + 306x + 1525 ,$$

and the last entry is the polynomial evaluated at $x = 5$. That is, we can accomplish deflation and evaluation with a single subroutine.

Similarly, we can evaluate the derivative $p'_n(x)$ at a given value of x by applying synthetic division to the quotient polynomial.

A Forth program to solve for the roots of well-behaved polynomials is found at the end of this chapter.

5. F.S. Acton, *Numerical Methods that Work* (Math. Ass'n of America, Washington, DC, 1990), p. 181 ff.

```
\ Regula Falsi - - ANS compatible version of September 10th, 1997
\              - - tested with WinForth v. 3.5 on September 10th, 1997
\ Finds roots of real transcendental functions by hybrid
\       secant/binary search method
\ Usage example:
\       : F1    ( F: x - - [x-e**-x])   FDUP  FNEGATE  FEXP  F-  ;
\       USE( F1  % 0 % 1 % 1.E-5 )FALSI  5.671432E-1  ok
\ Environmental dependencies:
\       Separate floating point stack
\       ANS FLOAT and FLOAT EXT wordsets
\       ANS TOOLS EXT wordsets
\
\     (c) Copyright 1994  Julian V. Noble.  Permission is granted
\     by the author to use this software for any application provided
\     the copyright notice is preserved.
MARKER -falsi
\ say "-falsi" to evaporate

\ Vectoring wordset (conditionally compile if not present)
DEFINED use(   NIP  0=  [IF]
    : use(    ’  STATE @  IF  POSTPONE LITERAL  THEN  ;  IMMEDIATE
    : v:   CREATE   [’]  NOOP  ,  DOES>  @  EXECUTE  ;
    : defines     ( xt - -)   ’ ( name )  >BODY
            STATE @  IF   POSTPONE  LITERAL  POSTPONE !
                     ELSE  !  THEN  ; IMMEDIATE [THEN]
\ Data structures
FVARIABLE A                      \ f(xa)
FVARIABLE B                      \ f(xb)
FVARIABLE XA                     \ lower end of interval
FVARIABLE XB                     \ upper end of interval
FVARIABLE EPSILON                \ precision
v: dummy                         \ vectored function name
\ End data structures

: x’    ( F: - - x’)               \ secant extrapolation
\         F" XA + (XA - XB) * A / (B - A) "    ;
          XA F@  FDUP   XB F@  F-       ( F: xa xa-xb )
          A  F@   B F@   FOVER  F-  F/  F*  F+   ;

: <x’>  ( F: - - <x’>)             \ binary search extrapolation
\         F" (XA + XB) / 2  "  ;
          XA F@  XB F@  F+  F2/  ;

: same-sign?   ( F: x y - -)  ( - - f)    F*   F0>  ;

: !end    ( F: x - -)
   FDUP  dummy   FDUP  ( F: - - x f[x] f[x] )
   A F@  same-sign?
          IF   A F!  XA F!   ELSE   B F!  XB F!   THEN   ;

: shrink   x’  !end   <x’>  !end  ;     \ combine extrapolations
```

```
: initialize    ( xt - -) ( F: lower upper precision - -)
        epsilon F!    XB F!    XA F!    \ store parameters
        defines dummy                   \ xt -> DUMMY
        XA F@  dummy  A F!              \ compute fn at endpts
        XB F@  dummy  B F!
        A  F@  B F@
        SAME-SIGN?  ABORT" EVEN # OF ROOTS IN INTERVAL!"  ;

: converged?    ( - - f)
\       F" ABS( XA - XB ) < EPSILON  "   ;
        XA F@   XB F@  F-  FABS   EPSILON F@  F<   ;

: )falsi        ( xt - -) ( F: upper lower precision - -)
        initialize
        BEGIN    shrink  converged?    UNTIL
        <x'>    ;
```

```
\ Laguerre method for finding polynomial roots
\ -------------------------------------------------
\     (c) Copyright 1999  Julian V. Noble.          \
\       Permission is granted by the author to      \
\       use this software for any application pro-  \
\       vided this copyright notice is preserved.   \
\ -------------------------------------------------
\ This is an ANS Forth program requiring the
\   FLOAT, FLOAT EXT, FILE and TOOLS EXT wordsets.
\
\ Environmental dependences:
\       Assumes PARSE can be used interpretively.
\
\       Assumes independent floating point stack
\       Complex numbers reside on the fp stack as
\       ( f: x y) where z = x + iy  (Im above Re).
\       The complex sqrt function, ZSQRT, is assumed to map
\       (0, 2*pi) into (0, pi). That is, its branch cut is the
\       positive real axis.
\ Non-Standard words
\       Most of these are conditionally compiled.
\       However as there is as yet no agreement as to the
\       names of certain complex functions, I have chosen
\       names that seemed sensible. Thus instead of
\       ZABS I have defined |z| (more telegraphic).
\       I also use the function |z|^2 which computes
\       x^2 + y^2 = conjg(z) * z .
\
\       The lexicon for arrays (arrays.f) builds both the number of
\       elements and the data size into the header of an array. This
\       information is required by the word }mov that moves data
\       from one array to another array, since I have written it
\       generically (that is, it works for any 2 arrays). }mov
\       could easily be replaced by one that works only for complex
\       arrays, if another array definition is preferred.
\
\       Definitions using |z|, |z|^2 or }mov have been marked
\       with **** for easy reference.
FALSE [IF]
    Reference:  F.S. Acton, "Numerical Methods that (Usually) Work"
                (Mathematical Ass'n of America, Washington, DC, 1990)
  Algorithm:
        For a given z, assumes z - z1 = a, and for all other
        roots, z - zn = b ; then
            G = p'(z)/p(z) = 1/a + (n-1)/b
        and
            H = G^2 - p"(z)/p(z) = 1/a^2 + (n-1)/b^2 .
        Eliminate b to get
            a = n/( G +- sqrt((nH-G^2)*(n-1)) ) .
        The next guess is z' = z - a .
        Iterate until converged, then deflate polynomial
        by the factor (z - root) and repeat.
[THEN]
```

```
MARKER -laguer
\ define "undefined" if it does not already exist
BL PARSE undefined DUP PAD C! PAD CHAR+ SWAP CHARS MOVE PAD FIND NIP 0=
[IF]  : undefined   BL WORD  FIND  NIP  0=  ;  [THEN]

include complex.f               \ lexicon for complex arithmetic
undefined s>f       [IF]  : s>f     S>D  D>F  ;     [THEN]
undefined f-rot     [IF]  : f-rot    FROT  FROT  ;  [THEN]
undefined fnip      [IF]  : fnip   FSWAP  FDROP  ;  [THEN]
undefined ftuck     [IF]  : ftuck    FSWAP  FOVER ; [THEN]
undefined 1/f       [IF]  : 1/f   F1.0  FSWAP  F/ ; [THEN]
undefined f^2       [IF]  : f^2   FDUP  F*  ;       [THEN]
undefined z^2       [IF]  : z^2    zdup z*  ;       [THEN]
undefined z*f       [IF]  : z*f   FROT  FOVER  F*  f-rot  F*  ; [THEN]
undefined z2*       [IF]  : z2*   F2*  FSWAP  F2*  FSWAP  ;     [THEN]

include arrays.f                \ lexicon for arrays
\ these are complex fp arrays
20 long 2 FLOATS 1array a{      \ coefficients of input polynomial
20 long 2 FLOATS 1array b{      \ coefficients of quotient polynomial
20 long 2 FLOATS 1array c{      \ coefficients of 1st derivative
20 long 2 FLOATS 1array d{      \ coefficients of 2nd derivative
\ complex variables
: zvariables    ( n --)  0 DO     CREATE   2 FLOATS  ALLOT  LOOP  ;
3 zvariables  G  zz  zp
fvariable epsilon
0 VALUE dummy{
0 VALUE chummy{
0 VALUE #iter
6 VALUE max_iter
\ ------------------------------------------ synthetic division
\   p[z] = (z-s) * q[z] + p[s]
\   adr1 is address of coeff array of input polynomial p[z]
\   adr2 is address of coeff array of quotient polynomial q[z]
\   n is degree of polynomial
: }zsynth  ( adr1 adr2 n -- ) ( f: s -- p[s])
    >R                  \ save N on rstack
    TO chummy{          \ vector array names
    TO dummy{
    dummy{ R@ }  z@                 ( f: -- z sum)
    0 R> 1-     DO                  \ count down from N to 1
        zdup   chummy{ I }  z!      \ b{ I } = sum
        zover  z*
        dummy{ I }  z@  z+          \ sum = sum * x + a{ I }
    -1 +LOOP                        ( f: s p[s])
    znip ;
```

```
FALSE [IF]
Test case for synthetic division:
    : }.    ( adr n -- )    \ display complex 1array
        0 SWAP  DO   DUP  I }  z@  CR  I .  z.  -1  +LOOP  DROP  ;

        7.e0 0e0 a{ 0 } z!
       -5.e0 0e0 a{ 1 } z!
        1.e0 0e0 a{ 2 } z!
      -14.e0 0e0 a{ 3 } z!
        0.e0 0e0 a{ 4 } z!
        3.e0 0e0 a{ 5 } z!
        a{ b{ 5 5.e0 0e0 )zsynth CR z.
        b{ 4 }.

    answers should be
    7.63200E3  + i 0.00000E-1  ok
    4 3.00000E0  + i 0.00000E-1
    3 1.50000E1  + i 0.00000E-1
    2 6.10000E1  + i 0.00000E-1
    1 3.06000E2  + i 0.00000E-1
    0 1.52500E3  + i 0.00000E-1  ok
[THEN]
\ ------------------------------------ end synthetic division
: guessed?    zdup  |z|  epsilon F@  F  ;  \ uses |z|      ****
: zmax     ( f: z1 z2 -- z1 | z2)   \ leave value with larger |z|
    zover zover                    ( f: z1 z2 z1 z2)
    |z|^2  f-rot |z|^2             ( f: -- z1 z2 |z2|^2  |z1|^2)
    F<   IF zdrop  ELSE  znip  THEN  ;      \ uses |z|^2    ****

: guess    ( n --) ( f: z -- a)
    >R                          \ save N on rstack
    zdup zz z!                  \ save initial guess
    a{ b{ R@ }zsynth            ( f: -- p[z])
    guessed?  IF  R DROP  EXIT  THEN
    zz z@  b{ c{ R@ 1- }zsynth  ( f: -- p[z] p'[z] )
    zover  z/  G z!             \ G = p'[z] / p[z]
    zz z@  c{ d{ R@ 2 - }zsynth
    z2*                         ( f: -- p[z] p"[z] )
    zswap  z/  znegate          ( f: --  -p"/p)
    G z@  z^2  z+               ( f: -- H)
    R@ sf  z*f  G z@  z^2 z-    ( f: -- n*H-G^2)
    R@ 1-  s>f  z*f  zsqrt      ( f: -- R)
    zdup  znegate               ( f: -- R  -R)
    G z@  z+                    ( f: -- R G-R)
    zswap G z@  z+              ( f: -- G-R G+R)
    zmax
    1/z  R> s>f  z*f   ;    \ a = n/( G +|- sqrt((nH-G^2)*(n-1)) )

0 VALUE #bytes
: }mov      ( src dst n --)     \ array_dst = array_src
    >R    2DUP  @  TO #bytes  @  #bytes    ABORT" Inconsistent data types"
    0 R>  DO  OVER  I }  OVER  I }   ( src[0] dst[0] src[I] dst[I])
              #bytes MOVE
    -1 +LOOP    2DROP  ;
```

```
: new_z     ( f: a --)   znegate  zz z@  z+ zp z!  ;

: apart?    zz z@ zp z@  z-  |z|  epsilon f@  F>  ;     \ uses |z| ****

: <root>      ( n --) ( f: -- root)
    >R
    zz z@  R@ guess         ( f: -- a)
    new_z
    0 TO #iter
    BEGIN   apart?   #iter max_iter < AND
    WHILE  zp z@  zdup  zz z!      \ zz = zp
           R@ guess
           new_z
           #iter 1+ TO #iter
    REPEAT
    R> DROP  ;

: quadroots
    a{ 2 } z@  |z|^2  F0=               \ uses |z|^2    ****
    IF  a{ 1 } z@ |z|^2  F0=
        IF  CR  ." no roots!"
        ELSE  CR  ." only one root!  "
              a{ 0 } z@  a{ 1 } z@  z/  znegate  z.
        THEN
    ELSE
        a{ 1 } z@  znegate    ( f: -b)
        zdup  z^2
        a{ 0 } z@  a{ 2 } z@
        z*  z2*  z2*  z-
        zsqrt                 ( f: -b d)
        zover zover           ( f: -b d -b d)
        z+  z2/  a{ 2 } z@ z/  CR  z.
        z-  z2/  a{ 2 } z@ z/  CR  z.
    THEN  ;

: roots     ( n --) ( f: z0 epsilon --)
    epsilon F!  zz z!
    BEGIN   DUP  2 >
    WHILE   DUP  <root>   CR  zp z@ z.
            1-  DUP  R  b{ a{ R }mov      \ uses }mov     ****
    REPEAT  DROP
    quadroots ;
```

```
FALSE [IF]
Test case for roots:
   p(z) = z^6 + 4*z^5 - 6*z^4 - 4*z^3 - 7*z^2 - 48*z + 60
        1.e0 0e0 a{ 6 } z!
        4.e0 0e0 a{ 5 } z!
       -6.e0 0e0 a{ 4 } z!
       -4.e0 0e0 a{ 3 } z!
       -7.e0 0e0 a{ 2 } z!
      -48.e0 0e0 a{ 1 } z!
       60.e0 0e0 a{ 0 } z!

Say:    6 -10e0 0e0 1e-9 roots

Should get
-5.00000E0  + i 0.00000E-1
 -2.00000E0  + i 0.00000E-1
 7.09585E-11  + i 1.73205E0
 1.00000E0  + i 6.14840E-12
 2.00000E0  + i -3.36886E-12
 2.93416E-11  + i -1.73205E0  ok
[THEN]
```